On-Demand Distributed Image Processing Over An Adaptive Campus-Grid

A thesis submitted in partial fulfilment

of the requirement for the degree of Doctor of Philosophy

Simon James Caton

Cardiff University
School of Computer Science

September 2009

Form: PGR_Submission_200701

NOTICE OF SUBMISSION OF THESIS FORM:
POSTGRADUATE RESEARCH

CARDIFF
UNIVERSITY
PRIFYSGOL
CAERDYDD

APPENDIX 1:
Specimen layout for Thesis Summary and Declaration/Statements page to be included in a Thesis

DECLARATION

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed [illegible] (candidate) Date 27th April 2010

STATEMENT 1

This thesis is being submitted in partial fulfillment of the requirements for the degree of ....PhD....(insert MCh, MD, MPhil, PhD etc, as appropriate)

Signed [illegible] (candidate) Date 27th April 2010

STATEMENT 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references.

Signed [illegible] (candidate) Date 27th April 2010

STATEMENT 3

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed [illegible] (candidate) Date 27th April 2010

~~STATEMENT 4: PREVIOUSLY APPROVED BAR ON ACCESS~~

~~I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loans **after expiry of a bar on access previously approved by the Graduate Development Committee.**~~

~~Signed .... (candidate) Date ....~~

Abstract

This thesis explores how scientific applications, which are based upon short jobs (seconds and minutes) can capitalize upon the idle workstations of a Campus-Grid. These resources are donated on a voluntary basis, and consequently, the Campus-Grid is constantly adapting and the availability of workstations changes. Typically, to utilize these resources a Condor system or equivalent would be used. However, such systems are designed with different trade-offs and incentives in mind and therefore do not provide intrinsic support for short jobs. The motivation for creating a provisioning scenario for short jobs is that Image Processing, as well as other areas of scientific analysis, are typically composed of short running jobs, but still require parallel solutions.

Much of the literature in this area comments on the challenges of performing such analysis efficiently and effectively; even when dedicated resources are in use. The main challenges are: latency and scheduling penalties, granularity and the potential for very short jobs. A volunteer Grid retains these challenges but also adds further challenges. These can be summarized as: unpredictable resource availability and longevity, multiple machine owners and administrators who directly affect the operating environment. Ultimately, this creates the requirement for well conceived and effective fault management strategies. However, these are tyically not in place to enable transparent fault-free job administration for the user.

This research demonstrates that these challenges are answerable, and that in doing so opportunistically sourced Campus-Grid resources can host disparate applications constituted of short running jobs, of as little as one second in length. This is demonstrated by the significant improvements in performance when the system presented here was compared to a well established Condor system. Here, improvements are increased job efficiency from 60–70% to 95%–100%, up to a 99% reduction in application makespan and up to a 13000% increase in the efficiency of resource utilization. The Condor pool in use is approximately 1,600 workstations distributed across 27 administrative domains of Cardiff University. The application domain of this research is Matlab-based image processing, and the application area used to demonstrate the approach is the analysis of Magnetic Resonance Imagery (MRI). However, the presented approach is generalizable to any application domain with similar characteristics.

Acknowledgements

A lot of people have been of great help and support during the work for this thesis, and here I would like to say thanks. To my wife Anke for her support and for proof reading the entire thesis. To my PhD supervisors Profs. Bruce Batchelor and Omer Rana for their help and guidance. To Matthan Caan and Dr. Silvia Olabarriaga for their collaboration, ideas and proof reading of Chapter 6. To Dr. James Osborne for his help with Condor. To the ARCCA team for permitting the use of the Cardiff University Condor Pool.

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A strange paradox has emerged in scientific computing. Computers have become faster and cheaper,[1] but the computation models that researchers are developing now take longer to solve [1]. The increase in capacity has enabled researchers' computational models to become much larger and more complex. This results in increased execution times, despite an increase in computational capabilities. In many cases, a single workstation quickly becomes inadequate and the success of an application or piece of research becomes dependent upon the ability to perform the analysis quickly, accurately and cheaply. Simply using more computational resources is not as simple as it may sound.

Traditionally, dedicated parallel architectures were used to provide a source of computational power. However, accessing large quantities of dedicated compute power with low user effort is challenging and not always possible [2] for reasons of price, availability, accessibility, security restraints and portability. In addition, it is seldom the case that a single researcher or research group possess or have access to such resources. Over the last two decades much research has been conducted to try and ease the transition of migrating an application or workflow from one node to many, especially when existing resources can be leveraged. Despite much research, many challenges remain unanswered when the problem domain reaches beyond the general case.

The discussion in this Chapter will focus on the key characteristics that have made Campus-Grids and the associated paradigm of Opportunistic Computing so popular a source of resources. Here, two of the predominant reasons are presented in Section 1.1, namely, the cost of prior investment vs. the cost new investment, and the environmental costs of these approaches. The application context and scenarios that this research is motivated by will be introduced in Section 1.2. In addition, this Section will introduce that in these areas of research, definitions that are taken as common fact in distributed computing, are still very much used incorrectly to the detriment of the standard of the literature in these fields. In Section 1.3 the challenges of performing such analysis in a pragmatic and efficient manner will be introduced, and later serve as a means to evaluate to what extent the solutions this research has uncovered have been able to lessen the impact of these challenges. Sub-

[1]The performance of a modern and reasonably priced desktop machine is comparable to a supercomputer that was in the list of the top 500 most powerful supercomputers throughout the world in the early '90s.

sequently, in Sections 1.4, 1.5 and 1.6 the research hypothesis, key research questions that will be used to support and test the hypothesis and the contributions of this research are presented. Lastly, in Section 1.7 an outline of this thesis is provided.

1.1 Opportunistic Computing and Campus-Grids

Opportunistic computing is essentially the leverage, or exploitation, of computational resources that would have otherwise been wasted. The principal resource is idle CPU cycles, but memory and disk space are other examples. The paradigm has a long history of success, primarily due to the availability of cheap or sometimes free resources. BOINC [3] powered projects, such as the well-known SETI@HOME [4] and other @HOME derivatives,[2] are prime examples of the paradigm's success and paramount to its legacy. The core philosophy behind these approaches is to encourage resource owners to donate their unused resources, in this case idle cycles of their CPU(s).[3] They may then participate in, and contribute to, a large experiment or project. The use of league tables provides further incentives as participants can form teams to compete with others, which enables them to show off their computing resources. Resources can be a standard desktop PC or laptop, a beowulf cluster or even an HPC machine. The notion of donating resources is often referred to as volunteer computing, but it is important to note that this is not a philanthropic environment, i.e. resources are lost as well as gained over time.

There are varying statistics regarding the levels of a workstation's utilization. Ranging from 5 - 40% [5–7]. In other words, as much as 95% of a workstation's utility (CPU, memory etc.) is wasted, leaving workstations to sit around idle gathering dust, their value depreciating daily. Even when a workstation is in use, it normally has a large portion of idle resources and at night it is hardly used at all. This situation lives in contradiction with the large demand for computational resources in certain areas. The need for and waste of resources often coexist in the same institution [8]. This is a strange situation if the demand for computational resources is high, as the initial investment of purchasing computational resources has already occurred, but the utility of the resources is not being maximized.

Large institutions may soon come under pressure to reduce their environmental impact. There are rumours that the computing industry could soon become a larger polluter than the aeronautical industry,[4] with large dedicated computing facilities and data centers being at the forefront. In contrast to the top500 list,[5] an environmentally friendly equivalent has emerged: the Green500.[6] Here, supercomputers are ranked according to the number of MFlops they can produce per Watt of power consumed. Even if the environmental impact of computing resources is put aside, the cost of basic

[2]Examples include: FightAIDS@HOME, Folding@HOME, ClimateModeling@HOME etc.

[3]This is achieved by downloading a worker daemon. Work is downloaded in units, when possible and required, and executed either as a screen saver or by detecting and consuming spare cycles.

[4]Announced by Dr. Pirahesh of the IBM Almaden Research Center in San Jose, during his keynote at the CCGRID '08: the Eighth IEEE International Symposium on Cluster Computing and the Grid.

[5]The list of the 500 most powerful supercomputers across the world: http://www.top500.org

[6]http://www.green500.org

utilities like gas, oil and electricity is a significant portion of the cost of computing capabilities. The cost of these utilities has also risen significantly in recent months. In general, dedicated infrastructures demand a lot of physical space, a temperature-controlled environment, and measures to deal with the noise produced by cluster nodes [8]. This can not only be expensive, but is becoming increasingly environmentally controversial.

Supercomputers and other dedicated resources are not the only computing resources. General purpose workstations also contribute to utility bills and environmental impact. There are two basic options for these workstations: (1) switch them off, hibernate them or place them into a low power mode such as standby when they are not in use, or (2) use them more efficiently. A combination of the two is of course also possible. The benefits of powering down a workstation when it is not required are obvious, but it might not be worthwhile. Switching the workstation off and then back on again may consume more power than leaving it on in the first place. Chein et al. [9] highlighted that the exploitation of idle cycles on pervasive desktop systems provides the opportunity to increase the available computing power by orders of magnitude (10x – 1000x). Osbourne and Hardisty [10] determined the average computational output when an average of 800 workstations were available (i.e. idle) to be 0.5 TFlops of computational power. To put this into context, a supercomputer near the bottom of the top500 list, at the same time, was measured at 2.03 TFlops. Considering that these workstations were up to four years old, this demonstration of computational ability is quite extraordinary. Chein et al. [9] noted that these computational resources can be as much as 5 to 10 times more cost effective (in terms of capital spent to performance achievable) than the cheapest hardware equivalents.[7] The rapid growth in Campus-Grids strongly reflects this processing capability and cost.

A Campus-Grid enables the collaboration of multiple departments, labs, and centers within an institution [11]. These groups are often in their own administrative domains but also share some common infrastructure. Existing security and administration procedures, for example, greatly appease the challenges of accessing and modifying resources, with the latter enabling a customisable computational environment. Campus-Grids provide the opportunity for researchers to utilize unused resources from anywhere on campus. Resources can include general purpose workstations available in a variety of open access, teaching and research computer labs, administration offices, lecture theaters and libraries, but can also include beowulf clusters, SMP and HPC machines as well as supercomputers. The Condor system [12, 13] is perhaps the forerunner of this domain and can provide access to sources of untapped computational power. Osbourne and Hardisty [10] also conducted a study of the Cardiff University Condor Pool, which, at the time, was the third largest Windows-based pool in the UK. They concluded that a Condor pool (or equivalent) is a very cheap source of raw computational power. Approximately £0.005 per CPU hour. In other words, for £1 a Condor user can consume more than a week of CPU time. This has lead to an architecture that allows Cardiff University to have the equivalent of a £500,000 supercomputer, but at a significantly

[7]Since this research, Graphical Processing Units (GPUs) have become a viable and cost effective option for HPC. GPU clusters are likely to reshape the HPC community sooner or later, even if it is only for specific application types. Their availability now contradicts this finding, although it is yet to be seen by how much.

lower cost. It is also a much greener solution than the equivalent computing power sourced from dedicated resources. Unpublished findings by Osbourne also identify that computing power sourced from idle workstations is up to 5 times more environmentally friendly, than equivalent dedicated resources. The saving comes from a reduced need for cooling, storage and the reuse of a previous investment. The Condor pool at Cardiff Univesity enables a saving of £$\frac{1}{3}$M per year by not using dedicated computational resources [14].

Some examples of Campus-Grids include: Alchemi [15], Condor@Cardiff [14], GreenGrid [11], Harmony [16], OxGRID [17], UoBGrid [18] and UVaCG [19].

1.2 Application Context

Image processing is an important part of many applications, scenarios and workflows, which span a wide range of academic and industrial domains. As a discipline it continues to permeate into new areas of research, and in domains where it is already a critical component increase its indispensability. It may be found in subject areas as disparate as manufacturing, engineering, medicine, site security and law enforcement, meteorology and robotics. Within these domains it performs a variety of roles as disparate as the domains themselves.

There is no shortage of example image processing applications that cannot be executed on a single workstation. Practically every medical imaging workflow requires higher performance than can be achieved with a desktop workstation at one or more stages of the computation. This is commonly the case during data visualization or the analytic processing of data. Todd-Pokropek [20] highlighted the need for high performance now that the analogue image capture processes for radiology have been largely replaced by digital equivalents. MRI (Magnetic Resonance Imaging) data is one such example, where studies frequently involve processing across several hundreds or thousands of relatively small (a few megabytes in size) images [21]. Bioimagery follows this same line, modern microscopes can produce images with resolutions of 40,000 by 40,000 pixels or higher and sizes range from several hundred megabytes to several tens of gigabytes [21]. Such images cannot even be held in the memory of a single workstation. Visualization of images of such a large scale is itself challenging. Typically, the image data is decomposed into tiles. Then sampling and montage techniques are employed to create a view of the data, which can be rendered on the user's workstation.

Multi-perspective vision systems demonstrate that a single workstation can soon become overwhelmed. Small scale systems (involving 3 cameras or less) can be handled by a modern desktop computer, but larger scale applications could be quite challenging even for HPC machines [22]. Object recognition is another example. Consider the processing of surveillance data for event analysis or crime detection, recognition techniques may be adopted to identify items of interest or significance. Such approaches are very computationally expensive.

The need for increased resource capabilities is not always related to computational capacity, memory restrictions can also elicit an increased need for resources. Image processing is a data intensive domain and thus the time required to manipulate and transfer data can be high in relation to the time required to process it. Applications with this requirement typically have a low flops to byte ratio. Consider comparative analysis of anatomical image data from a source such as an MRI scanner. Simple anatomical differences and variation in patient orientation during the image acquisition create a non-uniform space. For comparative analysis (e.g. comparing fibre bundles or identifying regions of interest based upon certain criteria or similarity) to be possible all data sets must be aligned (registered). This process may involve transformations in multiple dimensions and involve large data sets. Similarly, it may not be possible to hold the complete data set in memory, in such a case more resources are needed to distribute the memory requirements. Satellite and radar imagery provides another example where data sets can easily reach hundreds of gigabytes in size [23, 24], and therefore require far more substantial amounts of memory than can be provided by a single workstation. The end result is that as contextual and conceptual models develop, the quantity of data and the computational complexity of the underlying image processing functionality also increases. Consequently, it is common for an application to reach beyond the capabilities of a single workstation. Such scenarios present many interesting challenges and research questions. For example how can the amount of data transferred be minimised, what transfer techniques should be employed, when should throttling be introduced, how many nodes should be used and what type of architecture should be used?

Many researchers have turned to Grid Computing [25] for an answer to their resource requirements, from many different disciplines. Plenty of approaches exist in the literature that have used Grid Computing for image processing too. Some have been much more successful than others, which gives rise to interesting observations within the literature for this domain. There is the clear disagreement of whether Grid Computing is suitable for image processing or not. Tackling and evaluating this objectively is very difficult, as the terminology is not consistent throughout the literature. Many approaches simply refer to Foster and Kesselman's definition [25] and state that they use "The Grid". In a similar way to the power grid and other large utility frameworks there is no one universal and global Grid, but many. In the UK alone there is more than one. Companies like HP and IBM have private Enterprise Grids, universities have Campus-Grids and the UK e-Science project provides the National Grid Service (NGS). In some cases the use of specific middleware is enough for a research paper to define its context as a Grid approach. For example, is using the Globus Toolkit [26] and Condor-G [27] enough to be considered a Grid approach, or do the resources in use, their locations, the administrative policies and variability of quality of service that surround these resources define the context? The literature suggests both, but the original definitions of Grid only permit the second.

One of the things that has been lost in Grid Computing is the way in which resources are utilized. Middleware provides the means to access resources, but middleware is not always as transparent as it was intended to be. Researchers have commented on the complexities of middleware and how this inevitably stops the casual users (non-computer scientists) from accessing these resources [2, 16, 19, 28–31]. In addition, no two image processing applications are the same, there can be strict environmental setup requirements (see Section 1.3.7) and the terminology is, at times, used very loosely. It is not surprising that there are significant differences in opinion over the usefulness of Grid Computing for image processing. There are two identifiable subsets of image processing applications: (1) those which can execute in most execution environments,[8] and (2) those which have specific environmental constraints, for example the requirement for specific libraries or software.

Any application within the first subset should fit nicely into a Grid context, such as custom applications that are written in one of the standard programming languages. However, those in the second subset are typically built upon and developed with specific libraries (e.g. R) and/or problem solving environments such as Matlab, Mathmatica and LabVIEW. These applications cannot easily fit into the Grid Computing paradigm, as they can require software licenses (Matlab, Mathmatica and LabVIEW). Even in the case of their open source counterparts, namely Octave, SciLab, etc. these applications need to be available to the compute nodes and correctly installed.[9] In this second subset more local, and thus controllable and customisable, resources are required. The ability to use a Campus-Grid for image processing or other domains with similar attributes is therefore an attractive option.

1.2.1 Application Types

There are two kinds of application which are considered in this research: (1) distributed applications, and (2) scientific analysis which require or would benefit from parallelization. Distributed applications are ones in which the programmer is aware of the distributed nature of the computational platform when the application is written [32]. Such applications aim to utilize multiple resources to improve their productivity. Both distributed and scientific applications may have the ability or requirement to generate new work at runtime.

1.3 Challenges of Using Campus-Grids

The requirement for faster, more powerful image processing systems and environments capable of high performance, which can adapt autonomously and dynamically to suit the changing demands of applications [33] has been identified, yet remains insatiable. Much work and research from many different perspectives has been pursued toward this goal and is presented in Chapter 3. Yet, the challenges of this domain, as outlined below, have resulted in numerous partial solutions, many of which are application specific. The main challenge is finding a balance between many conflicting and sometimes paradoxical goals.

[8]This does not mean that they are platform-independent.

[9]It cannot be assumed that they are in fact correctly installed. Even when they are, the user may not have sufficient access to tune the installation to their specific job. This is a common problem in Matlab where environment variables like the search path and Java `classpath` need changing, but the user cannot introduce persistent changes to do this.

1.3.1 Resource Volatility

Resources within Campus-Grid infrastructures are volatile [9, 34–36]. There are several sources of volatility, some context dependant and other architecturally dependent, and therefore not all attributes of volatility are experienced in all Campus-Grids. The main attributes are: (1) the volunteer nature of resources, (2) resource autonomy, (3) network and resource failures, (4) multiple resource owners, (5) hostile environment, (6) multiple sources of administration, and (7) heterogeneity of hardware and software.

As resources are provided on a voluntary basis, they are only available haphazardly and for a finite time. The variability of resource availability makes creating a stable environment particularly challenging [37], because workstations can freely establish and revoke their voluntary status without any constraints [35]. This can occur as a result of both active and passive decisions and events. Active decisions can be categorized through the autonomy of the owner (denoted as volunteer autonomy in [35]). Owner autonomy encapsulates a few scenarios, but the most common is the reclaiming of their workstation for private use.[10] The consequence of this is the suspension or termination of a remote job running on their workstation. The commonly adopted solution is to reschedule the job to another workstation, where it can be restarted from scratch or a previously saved state: called a checkpoint. The challenge here is in the retention of throughput, because as workstations are lost, throughput ultimately suffers. This is a consequence of losing the processing time, which occurred prior to the failure or eviction and the cost of rescheduling the job. This process becomes more expensive toward the end of an application's execution, as there are fewer other jobs executing concurrently and therefore reduces the efficiency of parallel execution.

The passive actions responsible for similar challenges are: failures, workstation autonomy and the unintentional reclaiming of a workstation.[11] Failures can occur at three levels: (1) software, (2) hardware, and (3) network, whereas workstation autonomy is exercised by the operating system, and workstation management software. The events that have a negative impact upon a workstation are virus checks, software updates and other administrative actions. These actions are unlikely to cause a job to be suspended or terminated, which makes them much harder to detect. Instead, they can produce less desirable scenarios such as reduced performance, yet, more subtle scenarios may also occur. One such example is independent[12] livelock, where workstations do not generate a result for a job even though they appear to be executing it [35, 38] (see Section 2.3 for an example). Handling jobs in this state is exceptionally challenging, as without resubmission, there are little or no symptoms that can be used to identify whether the job is stuck[13] or the workstation is at

[10]Others include powering down or rebooting the workstation and opting out as a provider of volunteered resources.

[11]Typically workstations highlight that they are idle using a combination of a lack of mouse and keyboard activity and a low CPU activity. As the mouse becomes more sensitive and lighter, and therefore a lot easier to move it is becoming a lot easier to wake up a workstation accidently. Similarly, in open access or teaching labs, it is not uncommon to see groups of workstations woken up by a user, so that they can use the first one to respond. This kind of owner behaviour will cause jobs running on these workstations to be suspended. However, in some cases job suspension is not possible and therefore the job will be evicted or fail (see: Section 2.3).

[12]In many cases these errors do not affect other jobs, but only the application as a whole, which is hence independent [35]. The only exception is when jobs intercommunicate; in such a case the traditional meaning of livelock would apply.

[13]For example an infinite loop, uncaught error or the user forgetting to terminate job.

fault.[14] For these reasons many existing scheduling mechanisms cannot detect failures stemming from workstation (and some volunteer) autonomy.

Campus-Grids have one core difference to other opportunistic approaches, namely, that each workstation can have more than one owner. In theory there is only one physical owner, which is the university or institution that paid for the workstation. However, using the assumption that the owner is anyone that could use the workstation, a very different model is created. Here, a workstation can have thousands of owners. Some workstations can also have many orders of magnitude more owners than others. Multiple owners increase the impact of owner autonomy, but it also means that owners are not in direct control of the workstations they use.

As a Campus-Grid spans multiple administrative domains, administration comes mainly from teams of local (i.e. school or department) and central administrators (university administration). Two further classes of administrator exist that have an equal, if not greater, impact upon resource volatility. **(1)** Those that have rights but are not necessarily administrators, and **(2)** owners with administrative access to their workstation. Consequently, this means that workstations can be upgraded haphazardly and from multiple sources. In addition, administrators in the different classes outlined here are unlikely to have strong communicative links, and possibly may not even be aware of each other. This occurs as each party has their own set of rules and agendas, which can be mutually exclusive, contradicting, dynamic and unique. This is also true between administrative domains [17], i.e. departments, schools and research groups. Over time, it therefore cannot be guaranteed that any software dependencies remain persistent, and that workstation updates can induce anomalous behaviour, which may not be permanent, and only affect specific users and jobs. The four Condor admonitions [39] are a good set of guidelines for volunteer computing, as they have been compiled from two decades of experience of the domain. In the context of multiple sources of administration the most fitting admonition is the assumption that a workstation's state is not persistent or correct. However, it is the user who must inevitably determine why their job has failed. As mentioned above, few systems can provide a distinction between failures due to job error and errors which stem from the operating environment. The latter may even be unique to a specific workstation for a specific job or set of jobs. The possibility that workstations can be modified also impacts upon job execution in other less obvious and invasive ways. State switching,[15] unanticipated background processing, changes in IP address or host name are all examples. These potential mutations in a workstation's state only exacerbate an already hostile environment for a job to execute within. Hostility, typically, stems from a workstation's finite availability, possible heterogeneity and that the job may only have a reduced set of access rights [39]. The final result is that job management challenges even the most capable of users, but in many cases it is nearly impossible to map and manage the execution of non-trivial applications by hand [2, 32].

To demonstrate the impact that volatility has on the utility of Campus-Grids Chein et al. [34]

[14]For example a missing or corrupt library, data corrupted during transfer, insufficient space, or high background processing causing a reduction in a job's performance.

[15]State switching is a change in software availability and/or integrity. It usually occurs as a consequence of a software update, such that the software entity changes from being perfectly usable to unusable or vice versa.

used Acharya et al.'s [7] cluster equivalence metric on a 220 node Entropia [9] system. The purpose of this research was to quantitatively identify what the performance of volunteered resources would be if they were actually a dedicated cluster. The system was determined to be the equivalent of a 209-node cluster on weekends, and a 160-node cluster on weekdays. An interesting result was the observation that the longer a job takes to complete the lower the cluster equivalence ratio. Initially this seems strange, as shorter jobs should have a higher scheduling overhead relative to their execution time, therefore reducing the efficiency of resource utilization. However, this is also the case on a cluster. The decrease in equivalence is reported to be because longer jobs are more likely to be affected by owner autonomy and failure. Resource management and application scheduling issues have been thoroughly studied, but most of this work considers a resource failure as a rare event [36]. In a Campus-Grid context resources do not necessarily fail more often, but resource revocation leads to the same eventuality: the job must be rescheduled. For this reason, failure can be used synonymously with job eviction[16] as the result of a workstation being reclaimed is the same as if the workstation would have failed. In this research they are not considered synonyms. The main reason for this is that a job can fail for many reasons other than being evicted from a workstation. This is demonstrated in Chapter 2.

1.3.2 Computational requirements are not static

The demand for computational resources is ad hoc and dynamic. This can occur at two levels: **(1)** the global level, and **(2)** the application level. The global level is where users and applications exist. Here, many simultaneous applications from one or more users can be executed. The central attributes that create dynamic demand are that no two applications are the same, and that users may run the same application several times as testing, preliminary runs or for result verification. The end effect is that over time resource requirements change, as applications complete and require less resources, which are released accordingly. However, an application may be causative and generate one or more other applications. Dynamic demand is also a consequence of how the execution of a job space must adapt given the nature of the resources in use. In other words, requirements can change with time, either as new or better resources become available [40], or as other resources are lost.

Dynamic resource requirements are also identifiable at the application level [41, 42]. For example during data transfers, computational requirements are lower than during the analysis phases. Even simple applications are composed of heterogenous, transient and finite functional entities, and therefore requirements may not only be variable, but also change spontaneously, which can give rise to dynamic job spaces.

Consider a surveillance system: should an event be detected, further processing is required to determine whether there is a risk associated with the event. This may involve views from other cameras and thus could create a large amount of additional computation to be completed as soon as possible. A similar example could arise in the analysis of MRI data or satellite data. If an anomalous

[16]This is the case in [36].

object (a tumor for example) is discovered, in the first instance, further analysis would be required. There could also be the need for a resolution different to the current view or orientation, in order to give a physician a deeper understanding of the anomaly. A 3D model may also be generated, to aid a radiologist or other specialist in diagnosis. Should a specific cloud formation or region of interest be identified, in the second case, then the data may need to be resampled to provide a higher resolution.

Similar circumstances are also possible in an industrial context. Consider an industrial inspection system for a manufacturing line that produces engine components: if an artifact is located somewhere on the component, further analysis may be used to determine whether the build quality is affected by this anomaly. Similarly, if component alinement is incorrect, analysis may be introduced autonomously to detect how much the components are offset and the result fed into a decision making model to decide whether the fault can be rectified. As these applications grow and their complexity increases, so too does the necessity to use a significant number of resources to support their execution [2]. However, because these scenarios also exhibit dynamic job spaces (one which can change during runtime), they not only require additional resources, but furthermore the ability to autonomously tailor the quantity of resources available.

1.3.3 User ability

As distributed computing continues to permeate and extend into more research fields, the proportion of developers that are adept at distributed computing is inevitably decreasing. For image processing, developers are scientists and engineers (in disciplines such as: maths, physics, vision, medicine etc.) and not parallel and distributed computing experts [23,24,31,40,43]. Such users might not need, or desire, to always know or understand where the data or software they are using actually resides [23]. Their requirements will be more of the sort: *"I want to do this, to that"* and possibly within some time constraint.

The architecture and software solutions that the user chooses should enrich their research rather than introduce new constraints and challenges. For these reasons, developers place their preference with systems that are easy to use and have good performance over those with peak performance, but which are hard to use or program [44]. Any gain in performance from the latter is strongly negated by the additional time and effort required for development.

Resource volatility (Section 1.3.1) is another contributing factor, as it increases the complexities of job management for users. Research conducted by Pancake et al. [43] determined that users can spend in the order of 10% of their time performing job setup. The measured 10% entails simply describing their job properties for the architecture they wish to use and then getting the jobs to run. For large applications this is likely to be a significant period of time and for inexperienced users far greater than 10%. Accessing distributed and parallel resources has become easier in recent years, but in-depth knowledge of the particular system, front-end or middleware is still required and in many cases the learning curve is steep.

1.3.4 Users' typical working environment

Applications developers are, as would be expected, application-centric. Therefore, most users are concerned solely with *what* their application does, not *how* a given architecture services their request(s) or where resources are located [23]. Problem solving environments, such as Matlab, have thrived because of their simple user interfaces and the reduced development time this enables. Encouraging users to step out of this comfort zone and into distributed computing, a paradigm full of unknowns and new challenges, is itself challenging [45]. There are several reasons for this difficulty, namely: **(1)** trust, **(2)** time and effort, **(3)** awareness, and as previously mentioned **(4)** ability.

Trust is a virtue that is very difficult to establish. In distributed computing trust is either assumed or intrinsically certified through authentication. Users are, however, much more sceptical, and therefore less willing to send their applications and data to remote locations. In a Campus-Grid environment there are thousands of people from a myriad of disciplines and backgrounds. Each of whom use the workstations consituting the Campus-Grid on a day to day basis, and could theoretically steal or corrupt the user's data and application.

In order to utilise distributed resources the user must spend time and effort. This can occur during the porting of code to a new environment. Examples include additional error handling, functionality decomposition,[17] and pre- and post-processing. Similarly, each approach requires some time to install and learn. This is time that could be spent doing research. Ultimately, users are more willing to wait hours, days and weeks for applications to complete rather than learn new tools and techniques, which could improve productivity.

Not all users are aware of what computing resources are available to them or what help they can receive in using them. However, approaches that enable researchers to treat large quantities of resources as a black box have been successful as this fits well into their development environment.

1.3.5 Determining resource requirements

The volatile nature of the resources in use means that the user needs to think carefully about the constitution of their jobs to minimize the time wasted as overhead. This can be easily achieved by increasing execution time, but if a job is too long the user runs the risk of the host workstation being revoked. Should this occur, jobs must be rescheduled and this may negate any gain in performance by increasing execution times. On the other hand, the user may receive more resources if a scheduler sees that they have more jobs. This brings forward questions that the user must answer, for example: how long should a job be, should job granularity be changed to reduce overhead, execution time or the risk of revocation, should jobs be clustered together to decrease overheads? There are many job/workflow management tools around but few can automate this process, placing performance in the hands of the user. In addition, clustering jobs means that should a workstation fail, a cluster of jobs may need rescheduling instead of just one job.

To answer even these simple questions, additional knowledge is required. Irrespective of user

[17]The breakdown of functionality into work units for parallel execution.

ability, this knowledge is not available prior to runtime, as no clairvoyant services are available and resource availability and allocation is difficult for the user to predict. The core point here is that whilst all these options and considerations can be made, a user is unlikely to either have the necessary information or expertise to correctly optimize their job space. Instead, the underlying architecture, which does in fact have most of the necessary information[18] and potential ability, should take care of these intricacies. The quantity of resources to be used can then be considered.

The dynamic nature of resource demands and availability mean that determining the number of resources required can be difficult. The use of remote resources only makes this process yet more challenging, because their state and computing capabilities are not necessarily known prior to execution. Estimates can be made using monitoring software to feed profiling tools and techniques, but ultimately changes in workstation state are difficult to predict. Similarly, judging the requirements for the set of jobs may be difficult. In many situations it can be impossible to determine their requirements prior to runtime (also noted by [41, 46]). For example the size and quantities of input/output data, runtime errors like running out of memory or disk space and, as mentioned above, job sets are not necessarily static over an application's runtime. In this sense the best indicator is the application itself [47].

Users' ability to identify their own requirements is a well known problem. Even with an unlimited supply of resources available, the user would still try to use everything. During this time resources would be used very inefficiently and much of the applications' runtime would be time wasted as latencies and overheads. A similar eventuality occurs when users are asked to estimate for how long they require a resource or set of resources, or for how long their application will run. This is also made more difficult by not knowing what workstations will be received in advance. Batch processing systems simplify this issue, but they cannot guarantee good performance, or an efficient utilisation of resources. The user is ultimately responsible for the latter. Given that the user has limited control over what resources are allocated or how many, expecting them to use resources efficiently is fundamentally flawed. This is reinforced by the possibility that users are domain experts rather than adept users of distributed computing infrastructures (Section 1.3.3). The resources themselves can also provide other challenges (see Chapter 2).

It is not unusual for vast amounts of compute power to be required at short notice, sporadically, temporarily, or for the requirements of an application to change with time. Such attributes make planning, resource reservation and fair allocation particularly difficult. Grid Computing sites built up of dedicated resources often require resources to be reserved (e.g. Grid5000 [48] and DAS-3 [49]) and plenty of toolkits and middleware facilitate this ability (e.g. the Globus toolkit [50]). Yet, this ability does not remove the challenge of determining how many resources should be used. Large computing sites such as Grid5000 are if anything more likely to encourage the user to use more resources than they require, simply because more are available. Similarly, online applications and those which require resources on-demand do not fit the model of reservation. It is more a case

[18]For example: statistics on previous job executions such as: CPU time, memory, disk and network bandwidth required, current resource availability, user priority, resource contention etc.

of using whatever is available at the time. In a dynamic application scenario, reservation may lead to under used resources or situations where more resources are needed, but cannot be used because they are reserved for another user. The benefits of reservation, however, is ensuring that resources are used consistently, and thus more efficiently. It can also be an easy metric to measure the demand for resources.

Research in Grid Computing answers the question of resource quantity by using one simple metric: cost. By allocating a budget to a set of jobs the execution time, and therefore the quantity of resources can be optimised according to cost. Buyya et al. [37] describe how this works in practice for Nimrod-G [51] and the Gridbus broker [52]. However, in the context of a Campus-Grid determining resource quantity based on cost is of little use. Firstly, resources are so cheap that the cost is almost negligible. Secondly, the volatility of resources and their dynamic availability makes planning very difficult. Finally, there are no service providers to instantiate agreements over quality of service and provide guarantees of resource availability, which accompany costing strategies.

As the complexity of an application grows, so too does the necessity to use a significant number of resources to support its execution [2]. However, the quantity of resources to be used becomes dependant on the user and how they partition their job space, which in turn lives in contradiction to a user's ability.

1.3.6 Achieving parallelism

Given the typical user expertise (Section 1.3.3), it would also be unreasonable and unrealistic to expect users to adopt complex low-level parallel programming approaches, such as the use of MPI libraries [53] that are typically used to speed up applications [31]. If this were the case, the ability of the user to improve computational performance would be linked to their ability to implement a software solution, and identify where parallelism can be added to improve performance. In addition, MPI-based approaches cannot withstand the potentially high variance in resource availability that comes with the use of opportunistic computing. Approaches such as OpenMP [54] may not be as demanding, but the developer must still need to know where their application can be parallelized. They would also require either a multicore workstation or access to shared memory parallel machine.

Aside from how speed up is achieved, many domains of scientific analysis are data intensive and therefore it is very difficult to maintain efficiency using a fine-grained abstraction. This is only made more difficult in the context of a Campus-Grid. Moler, [55] who here defends the decision to not create a parallel version of Matlab, reasons that if a data set fits into memory, little or no performance can be achieved with a fine-grained parallel model. In some cases partitioning and subsequently sending the data can take as long as processing it. Scalability can also present additional challenges, as adding more worker nodes does not always mean a reduction in execution time. The literature presents cases where adding more worker nodes could increase rather than decrease execution time (e.g. [31, 44, 56]). These scenarios arise as a consequence of an increase in communication overhead from adding more nodes. As the number of nodes exceeds an application specific threshold, communication overheads begin to saturate the total wall clock execution time. If left unchecked this will eventually surpass the amount of time that is consumed by the execution of an application. At this point resources are being wasted. Friedman and Hadad [57] observed that the distributed computing research community frequently provide detailed breakdowns of latency costs and how this relates to the general approach. Yet, they do not consider the throughput limiting factors. Some examples include: using too many nodes or scheduling without considering data dependencies or provenance.

In an image processing context a fine-grained parallel model can also limit the number of image processing functions that can be facilitated. Tasks such as convolution, morphology and local operators can receive great improvements in performance from a fine-grained model. This has been thoroughly demonstrated by the parallel processing community. However, these same operations can be vastly improved in a serial context too, just by changing the execution strategy.[19] Not all image processing operations can be simply performed using a fine grained model without the user writing specific parallel routines. In many cases it is possible to adopt a coarse-grained parallel approach quickly and easily, with minimal changes to application code. This was demonstrated in [59].

1.3.7 Operating environment

Image processing has the requirement for very specific environmental conditions. The need for commodity software and legacy systems such as Matlab, Octave [60], the GIMP [61] and LabVIEW [62] is needed to load, manipulate, interpret and process the image data. Such software can be stringently licensed as well as expensive. Moving the computational environment to a remote site would not normally have many issues other than portability. In an image processing context, however, licenses must also be in place to facilitate the application. Systems like Condor can describe the availability of local licenses (see [63]),[20] but currently, no solutions are in place for transferring licenses between sites or subdomains [64, 65].[21] In this case a job that requires a software license must either be run within the licensing domain, or it must borrow a license at the remote site. Given the cost of licenses it it unlikely that the latter will be an acceptable solution.

Similarly, mechanisms for efficient and meaningful data manipulation and control are required. This need arises for the simple reason that if the infrastructure cannot adequately handle data, processing it and extracting usable results will become a lot harder. Such requirements mean that high level control over the operating environment is essential. In this case, if licensing is not an issue, the required software libraries must still be available to the job. There are two realistic options here: (1) the software is pre-installed and declarable as a job requirement, (2) the software is sent to the remote node with the job. The latter of course assumes that the software can be installed, either

[19]For example see Skipsm [58]; an execution model for improving the serial execution of morphological operators.

[20]However, this approach cannot describe how many licenses there are or if any are currently available.

[21]Recently, a 7th Framework European project SmartLM has begun to investigate how the licenses of software products using license managers such as FlexLM (e.g. Matlab) can be transferred between sites. See: [66] or `http://www.smartlm.eu` (last accessed April 2009), for further details.

silently[22] or that the job has sufficient access to install it for the session. Naturally, this is not always an option. The data requirements and security implications being the most obvious examples. A job's requirement for software libraries is not unusual, but it is one that quickly reduces the number of potential sites where the job can be run.

The requirement for software libraries actually motivates the use of Campus-Grids. The existing administration mechanisms can be used to install any required software. The only issue is the unpredictable nature of a workstation, which comes with multiple sources of administration as mentioned in Section 1.3.1.

1.3.8 Latency and Overhead

The distributed computing research and development communities have provided a large repository of tools and techniques for users to access resources (e.g. clusters, supercomputers, networks of workstations) at one or more remote sites. However, sometimes there can be so much middleware and intermediary software components that latency and bottlenecks can become overwhelming. In fact, Bovenkamp et al. [67] highlighted that the image processing module and its control is often the most major bottleneck in (distributed) image processing.

The longevity of a job also impacts upon overheads. The shorter a job is, the more likely that scheduling overheads[23] and data transfer overheads[24] will limit performance. Different architectures have different recommendations[25] for a minimum job length. Any less, and a good proportion of the turnaround time for a job can be attributed to scheduler overhead. When a user has jobs below this recommendation, they have only two options: **(1)** rethink, and tailor the job space to improve performance, or **(2)** leave things as they are, thereby ignoring any reduction in performance. In many cases users will opt for the second option, for reasons of convenience, insufficient information and expertise.

The overhead experienced by a user also increases as a result of the software requirements mentioned in Section 1.3.7. This impact upon performance is the time required for software to initialize. Initialization, which must occur before any execution can be performed, is wasted time. Typically, in job sets that need additional resources for computation, many jobs will share similar software requirements. However, each job will need to initialize this software, because for each job a new working session is created. This is a fundamental property of general distributed computing infrastructures, because they cannot control the software in use. Only in very specific architectures can this be alleviated, yet these approaches are commonly not portable. The problem here is that the node and software are being used for the duration of the job and not the duration of the node's availability. Therefore, a node may initialize the same software multiple times. For short jobs this

[22]A silent installation refers to the ability to make software available without using any specific administration tools. For example, a runtime environment just needs to be accessible, it does not matter where it is.

[23]Scheduling overhead is the time required to allocate a job to a node, transfer the required libraries and data and start execution.

[24]The time required to transfer data to the remote node.

[25]For example, the recommended minimum time for a Condor job is 10 minutes.

can quickly become a large, and avoidable, proportion of a job's makespan.

1.3.9 Job type

This research concentrates on a specific job type. However, this job type is in fact quite common. Whilst image processing is the primary application area of this research, it is interchangeable with any other area of scientific analysis, where a job's constitution is compatible with the attributes listed below. This is possible, as many of the challenges presented above can be applied to many different domains. These challenges are mainly infrastructural and conceptual, but they stem from the type of jobs which constitute an application's context (see: Section 1.2). Below are the key attributes which identify the type of job that is considered by this research. Note that a job does not need to have all these attributes, to benefit from this research.

Attribute(1): Jobs are short running, seconds to minutes in length.

Attribute(2): They use a coarse-grained abstraction as defined in Section 4.9.1.

Attribute(3): They can be causative and therefore instigate dynamic job spaces. In other words, the result(s) of one job may induce the need for the execution of additional jobs.

Attribute(4): Have rigid software requirements, such as the need for applications like Matlab to be installed and available.

Attribute(5): May require interactive use of resources, i.e. batch processing is not an option. For example, short running jobs may need interactive use of resources in order to avoid high overheads.

1.3.10 Summary

The challenges which have been outlined in this Section illustrate the complexity of using Campus-Grids. Their complexity largely stems from the volatility and variability of the resources themselves, which adds additional challenges. This means that applications which use Campus-Grids must be able to adapt within and respond to the environment they are using. They must also be capable of handling these challenges to produce effective methods for job management to control the potential for high overheads from the volatility and availability of resources and the potential for job failure. Users cannot be expected to do this, because they are domain experts rather than distributed computing experts. For the same reason, users cannot be expected, and are unwilling, to significantly modify their models to use distributed resources. Their role is to further their research, not perfect their resource administration and parallel programming skills. Finally, determining the number of workstations to use raises many questions: about job constitution, the workstations that will be allocated, whether those that are allocated will work correctly and whether the quantity allocated will be detrimental to performance due to over allocation.

These challenges are also self deprecating. The fact that they exist, makes it more challenging to convince users that Campus-Grids and distributed computing are useful assets to their research. Essentially, the grand challenge is to hide the complexities and bring forward a familiar environment, where they can conduct their research. For this reason the research conducted throughout this thesis is not based upon the results of simulators, but upon a real system implemented as a core part of this research. The implemented system can not only be used for research into this area, but also for the facilitation of external research.

In returning to the application context, even without using a Campus-Grid, distributed image processing is hard [29, 31, 32, 45, 68]. This is also true for many other areas of scientific computing. However, as distributed infrastructures and [image processing] tools mature and become more readily available, merging the two into reliable, open and accessible systems is a challenging task [31].

1.4 Research Hypothesis

The volatility of a Campus-Grid places many hurdles in front of users, but the power and utility of such an architecture should not be discounted for such reasons. The hypothesis of this research is that a Campus-Grid can be made reliable, efficient and stable even for very short jobs (seconds), if the conceptual models of job and resource management are changed to directly consider the volatile nature of the resources in use. By changing these models, a virtual, but dedicated, cluster of workstations can be constructed. The workstations can therefore be used for as long as they are available, or needed, instead of the length of a single job.

The modification of the job and resource management models will enable other models to be changed and introduced. A key aspect of the hypothesis is the modification of the fault tolerance models. They must be sensitive toward the environment's context rather than just identifying failed and successful jobs. Further motivations and real life examples of where the current models falter are presented in Chapter 2. These examples can only exist in real systems. Simulated environments cannot accurately model these scenarios because they can be so diverse that it is difficult to imagine to what extent they occur and then understand how, why and when they occur. Scheduling is another example where this time an application's context can be considered. The aim here is that instead of packing potentially disparate jobs into a global plan over time, the scheduling decisions can be given to the application itself, to enable a more directed and application and context aware approach.

Once the conceptual models are in place to enable stability, reliability and efficiency, then the performance of an application can be increased without compromising on either the fiscal or temporal costs of performing the analysis or the accuracy of the produced results. In the worst case, performance will be improved without increasing the number of workstations used. However, in the best case, performance will be improved with a reduction in the number of workstations used. This will occur thanks to a decrease in overheads, which occurs as a direct consequence of the existing conceptual models. The combination of the conceptual models that creates the idea of a Virtual Cluster will also enable support for applications that require resources spontaneously and interactively. Interactivity is possible because workstations are held on to, and therefore can be used interactively themselves. Finally, this is achievable with an increase in resource transparency to the user, i.e. a reduction in a job set's administrative procedures such as the remapping or wrapping of code, job description and reusability of components.

1.5 Research Questions

From this hypothesis several core research questions are needed to evaluate the hypothesis.
Question 1: What measures are needed to make a Campus-Grid feasible for the job definition given in Section 1.3.9?

> **Motivation:** There are two questions here, the first is stated above. The second question is: how do the current conceptual models (job and resource management) in place need to be extended? It is the adaptive and volatile environment, which makes using a Campus-Grid challenging. Therefore, fault tolerance and prevention is one of the key areas, as is dynamic and autonomous management to keep up with the ever changing environment. This variability of resources makes creating a stable and reliable environment particularly challenging [37]. It also makes it nearly impossible for a user to map and then manage the execution of non-trivial applications by hand [2,32].
>
> **Relation to hypothesis:** The creation of a stable environment for the execution of jobs demonstrates the feasibility of the hypothesis. Additionally, it also enables research questions 2, 3 and 4 to be answered.
>
> **Analysis required to answer question:** The building and critical evaluation of a working computational model will provide a solid platform to answer this question, and raise others.

Question 2: How can the number of resources to be used for the execution of a set of jobs be determined?

> **Motivation:** This question relates back to the challenge outlined in Section 1.3.5, which stated that it is difficult if not impossible to determine the resource requirements of an application. This process is only made harder by the volatility (Section 1.3.1) of the resources in use, and the opportunistic context in which resources are used.
>
> The consequence of variations in availability and resource volatility mean that allocated resources are dynamically lost and gained over time, and those which are allocated are not guaranteed to function as expected. In addition, the changes in modelling create a different working scenario, where fewer workstations can potentially provide an improved throughput. For these reasons a requirements model needs to be able to signal that if more workstations become available, then it would be possible for an application to consume these additional resources. Here, the idea is to provide a general indicator of the number of workstations for an autonomic system manager to aim for. Such an estimate may never be achieved, but even an ambitious estimate is a useful indicator.

Relation to hypothesis: The models introduced by the hypothesis will impact upon performance, and a requirements model is needed to encapsulate this.

Analysis required to answer question: By monitoring the efficiency of resource utilization as well as performance as a whole, it can be ascertained whether the resources that are supplied by the system are both sensible, i.e. not detrimental to performance, and adequate to yield an improvement in application makespan.

Question 3: What differences does the user experience as a result of introducing these models?

Motivation: In answering this question the proposed research can be validated and its usefulness demonstrated.

Relation to hypothesis: Answering this question will illustrate the applicability of the approach by demonstrating the ways in which users are able to do something that was previously either not possible or feasible, or how they can improve upon previous methods.

Analysis required to answer question: The development and facilitation of a real life application, will be used to illustrate where gains in performance and useability can be achieved.

Question 4: What are the bounds of these models?

Motivation: This approach is primarily targeted at short running jobs and those which meet the criteria in Section 1.3.9. There are two aims of this question. Firstly, to identify how short a job can be to feasibly be executed within a Campus-Grid environment. Secondly, to determine the range of jobs which do not fit this criteria but can still work within this context. In other words, does this approach negatively impact the performance of longer running jobs?

Relation to hypothesis: The outcome for this analysis will demonstrate the applicability of the approach but also determine scenarios where more research is required to capitalize upon the utility of a Campus-Grid. Similarly it will enable the identification of other scenarios that simply do not fit this model.

Analysis required to answer question: The easiest way to evaluate this question is through application development and synthesized applications.[26] Applications demonstrate the feasibility of the approach within a realistic context. However, they are difficult to reproduce as the underlying workstations are inherently heterogeneous, making

[26]A synthesized application is one contrived to suit the purposes of evaluation. It differs from a simulation in that real jobs are still performed on the Campus-Grid. A deeper explanation and motivation for such an application is given in Section 2.4.

it difficult to create the same execution environment. A synthesized application, sidesteps this problem because it is controllable and can be made independent of workstation performance. This ability means that an approach can be evaluated fairly and objectively.

These are only a few possible research questions that can be raised from the hypothesis. Some other questions which have not been attempted, but which would be useful future contributions are outlined in Sections 6.10 and 8.3.

1.6 Contributions of this Research

This research has demonstrated the feasibility of Campus-Grids for even very short running jobs (seconds). It has also documented how this can be achieved. An adaptive model has been developed and deployed within a real Campus-Grid environment. This model enables the creation of a stable environment despite the volatile context in which the workstations exist.

The result is an improved platform for users, allowing them to perfect the applications at the heart of their research and not their resource administration skills. This is achieved by isolating the user from the administrative, pragmatic and sometimes tedious roles of job and resource management. Users can harness the computational resources of the Campus-Grid without leaving their familiar operating environment. This has been demonstrated through the creation of a Matlab-based system for distributed image processing [33, 38, 59]. The utility provided by this approach can be leveraged directly from Matlab, with the result that the user can harness computational resources from their Matlab session transparently. This enables the user to directly integrate computational resources directly into an application's context, and therefore consider application specific trade offs and agendas. Similarly, an application can dynamically change its resource requirements over time, causing resources to be acquired and relinquished transparently, based upon demand and availability.

The final contribution of this research has been the demonstration that performance can increase without increasing cost(s) or reducing the accuracy of the results. The approach is evaluated alongside a managed Condor system, showing significant improvements in performance over a standard Condor system, even when fewer workstations were used.

1.7 Overview of the Thesis

Chapter 2 presents an empirical study of Condor from a user's perspective. This study has been performed to illustrate that the challenges presented in this introduction are not only real, but have significant impacts upon performance and raise further challenges in job administration. This Chapter also sets the initial benchmarks for the system developed as a part of this research to be compared against, in order to quantify the improvements in performance that have been gained as a direct result of this research.

Chapter 3 reviews existing techniques of creating a distributed image processing environment (not necessarily with Matlab) and aims to illustrate how this approach differs from other attempts. It also details other relevant work in the areas of volunteer, opportunistic and distributed computing, commenting on the areas within these domains that cannot support the type of applications presented earlier in this introduction.

Chapter 4 presents the methodology, philosophy and core assumptions that the approach adopts and commences the discussion on where the limitations, paradoxes and trade-offs exist in this approach. It also introduces the core aspects of the solutions to the answerable challenges of Chapters 1 and 2 and how this research differs from what is presented and discussed in Chapter 3.

Chapter 5 presents how the system has been built, what components constitute the solutions to the presented challenges and where the limitations of this approach lie.

Chapter 6 documents the real-world application of the system documented in Chapter 5 on a medical image processing workflow, which has been developed and aided by using this approach. The aim of this Chapter is to show how the user can benefit from this research and what lessons have been learnt through this development.

In Chapter 7 the approach is evaluated using a series of experiments comprising of synthesized jobs are presented. The results are discussed and compared to the results of the same analysis performed using the current Campus-Grid middleware (Chapter 2).

Finally, in Chapter 8 the final results, contributions and lessons learnt from this research are presented as well as future work.

Chapter 2

Analysing the limitations of Condor within a Campus-Grid

This Chapter aims to demonstrate in a real scenario the challenges that were identified in Section 1.3. Three studies are presented each to convey a different aspect of the use of such an infrastructure. The first demonstrates the challenges of job management and the importance of "*sensible*" strategies for job submission. The second introduces the impact of resource volatility upon job management. Here, emphasis is placed upon the scenario whereby the software requirements (Matlab in this case) are advertised by workstations but are either not available at all, or incorrectly installed. Both of these studies present challenges that the user must face in job and resource administration and also in application steering. Ultimately they demonstrate the volatility of the resources in use. The third study captures Condor's performance when short jobs are to be performed. Here, Matlab jobs with predefined runtimes of as little as one second are executed, and the performance of the system analysed.

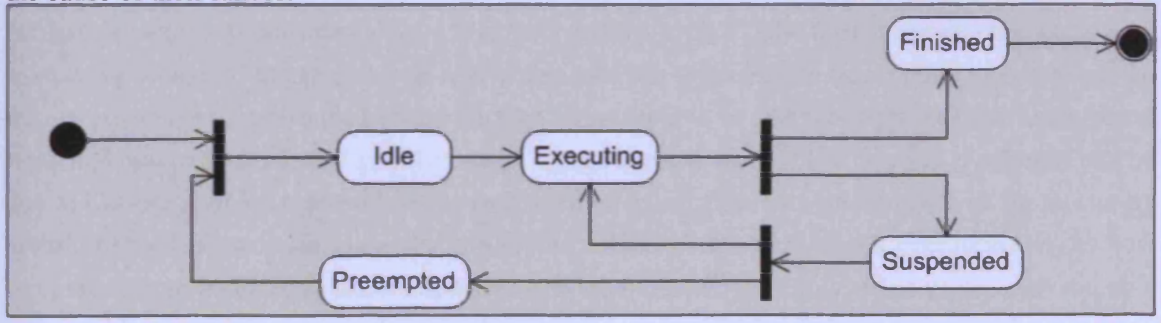
These three studies are not a criticism of Condor, but rather intend to motivate the need for research to improve the performance for applications that require further resources from infrastructures such as a Campus-Grid, and especially when the jobs are quite short in length. The motivation for this work is therefore, firstly, to demonstrate that there is room for improvement and, secondly, where improvement is needed.

The structure of this Chapter is as follows: in Section 2.1, the method for the extraction of experimental data from Condor logs is presented. In Section 2.2, the first study: the overhead related to job queue size, is presented and the results discussed. In Section 2.3, the second study: the difficulty in successfully executing Matlab jobs, is presented and the results discussed. In Section 2.4.1 the methodology behind the analysis which constitutes the third study is presented, as it forms the benchmark for the performance-related comparison of this research (Chapter 7). In Section 2.4.2 an Autonomic Job Manager is defined, as it is needed to run and steer the experiments for the third study. In Section 2.4.3, the third study: application-centric performance for short jobs, is presented and the results discussed. Finally, in Section 2.5 a summary of the Chapter is presented and its findings discussed.

2.1 Acquisition and Manipulation of Experimental Data

In order to discuss the results and implications of the execution of Condor-based experimental analysis, data describing the behaviour of job execution is paramount. The primary source of data detailing job execution are the Condor job logs, which are stored on the user's submit machine.[1] Condor uses these log files to record changes in state of jobs and also to provide some additional information of what happened during the execution of each job. However, for intuitive job analysis of large numbers of jobs (in the order of thousands) this data is fairly difficult to use by hand. Essentially, it is a record of all state transitions (Figure 2.1 shows the basic state transitions) of all jobs with a date/time stamp. Entries are made chronologically and therefore events for specific jobs are not necessarily near each other in the log file.

Figure 2.1: Most Command State Transitions of a Condor Job. Note: that here failed is a sub-state of finished. Such jobs may or may not be rescheduled automatically by Condor, depending on the the cause of their failure.



To obtain usable data from the Condor logs they must be parsed to extract the state transitions of each job. This data can then be used to build an object model representing each job and its associated state transitions. By converting the data of job space in this manner, a normalized representation of the state of the job space as time series data can be generated. This is achieved by sampling the state of each job for the duration of the execution of the job space. Once the job space has been represented in this manner it can be formatted to suit any persistent medium like a database, where it can be searched and graphically represented. Similarly, characteristic metrics such as the average job runtime or the number of concurrently executing jobs at a specific time, can be easily calculated. Such metrics are not normally available to the user, although Condor Quill [69] can provide this information to a certain extent. However, these services are not standard Condor features, but optional additions and are not always available.

This data can be extracted both during and after runtime, and can have uses other than just retrospective analysis. For example, in application steering, monitoring and error handling. This

[1] Other sources would be the *condor_history* utility, Quill [69] and Hawkeye [70].

information can also be used by autonomous processes, for example Section 2.4.2 describes an Autonomic Job Manager which is used to steer the third study performed in this Chapter.

2.2 Job Overheads

Whenever a Condor job is started on a remote workstation it is subject to overhead. This stems from two primary sources: **(1)** pre-staging: the initialisation routines, for example environment setup (setting the path, mirroring the user's environment, starting external applications etc.), input file transfers or modification prior to execution (unextracting compressed archives etc.) and any Condor and architecture specific behaviours that are required to physically start job execution. **(2)** post-staging: the cleaning up after the job has finished, the preparation to vacate the workstation and transfer of results.

It is clear that this overhead is related to application complexity and context. Yet, even the simplest of jobs are subject to Condor and architecture specific overheads. If additional factors such as input and output file transfers and job-specific pre- and post-staging are removed it is possible to measure the time required for a job to start. To measure this time a *Dummy Job* can be used, as can a job with a predetermined execution time that is independent of workstation performance. A Dummy Job in this instance is one that calls the Windows command `exit` and thus terminates the job session as soon as it begins. Condor *should* record this as a job that requires 0 seconds to execute, as the minimum time unit used in the Condor log files is one second. In addition, because there are minimal input and output files to transfer, no overhead should occur due to file I/O. Instead, overheads will be due to Condor's job start procedure, as well as other external factors stemming from the operating system of the host workstation and strain on the submit node. Detrimental operations on the host workstation can include: Windows updates, anti-virus checks, the applications of the workstation's owner and any other background processing that the workstation is currently undertaking.

The questions that this experimental analysis is trying to answer are listed below. The aims of this analysis are to demonstrate the administrative aspects of job management that a user must undertake, and furthermore, to demonstrate that Condor, when left unchecked, can waste large quantities of resources unnecessarily.

Question: Is the overhead a significant factor in a job's execution time?

> **Motivation:** The identification of the overhead for a job is an important factor in accessing the performance of any distributed system. For short jobs overhead can be a large proportion of its makespan, thus making resource utilization inefficient. If overhead was static its impact upon performance would be inversely proportional to the job's execution time. Unfortunately, overhead is not static, but dynamic. How dynamic is dependant on many factors, but the variance in the overhead is a good indicator of how resources change over time and also demonstrates their volatility. This overhead, when combined with an estimate of the scheduling overhead, can also be used in predictive models to determine minimum job lengths for a given resource utilization

efficiency target.

The Dummy Job being used is sufficiently light weight so that workstation performance is not a factor in determining its execution time. Only external workstation specific conditions and Condor overheads can impact upon its execution time. The greater the influence of these factors, the higher job execution time will be.

Question: Does overhead increase in relation to the number of concurrent jobs?

Motivation: In theory this should be an inconsequential factor for starting remote execution. The number of jobs waiting in the queue for a specific user should only influence the scheduling overhead, i.e. the time to match a job to a workstation. The `condor_shadow` daemon is used to represent the user on the remote workstation [39]. Casual observation of submit nodes over time has hinted toward the hypothesis that if the number of jobs increases so too does the load on the submit node. Therefore the performance on the remote workstation is reduced. If this is the case, increasing the size of the local queue should impact upon execution times. Such an eventuality would be significant, because without local regulation of submissions, resources are being unnecessarily wasted.

Question: How often do jobs runaway and how expensive is a runaway job?

Motivation: A *runaway* job is one that requires an unexplainably large amount of time to complete (with respect to its anticipated runtime). One important and distinguishing attribute of a runaway job is that is does not fail, as given the opportunity it will eventually complete. Therefore, a job that requires a long time to run, but then fails is not classified as a runaway job. By quantifying this and determining the variance in the results, the volatility of the resources in use can be demonstrated. This is due to the difficulty in detecting the cause of a runaway job, as a number of potential contributing factors exist. The problem here is that many fault tolerance approaches assume that the job is at fault, and will monitor the job in future (rescheduled) executions when the host workstation may in fact be at fault.

If a Dummy Job becomes a runaway job, it is not possible for it to be at fault if a significant proportion of other runs complete as expected. Thus, runaway jobs could only be symptomatic of external workstation specific factors and/or Condor faults. Analysis of the frequency of runaway jobs will further demonstrate the administrative challenges faced by users. In this case the job is not representative of normal job complexity, but it is clear that the job cannot be at fault. Outside of these controlled conditions it becomes increasingly more difficult to determine where the fault lies, especially considering the methods employed to describe failures.

2.2.1 Experimental Analysis

The number of jobs per experiment is varied to 100, 200, 300, 400, 500, 750, 1000 and 2000 concurrently submitted jobs[2] and each experiment has been repeated ten times at different times of day, resulting in 52,500 submissions of identical Dummy Job. Figure 2.1 showed the state changes possible for a job. However, should a job be preempted, the job model will reset its execution time so that any additional time spent in the queue does not contribute to the monitored execution time. Each experiment is monitored by an autonomous manager, which monitors the Condor log to determine when an experiment has finished[3] to enable back to back experiment runs. This ensures that the Condor pool is as consistent as possible.

These experiments are not concerned with fault detection and autonomic correction: if a job fails the manager will leave Condor to reschedule the job. Changes in idle time due to an increased local queue are not taken into account. Similarly, the total runtime is also not considered here. However, should an experiment of a larger number of jobs take significantly longer to execute than the equivalent number of sequentially executed smaller experiments, then this will be presented. Such an eventuality would demonstrate that the size of the local queue is an influential factor on the total makespan of an application if a similar number of workstations were used.

2.2.2 Results

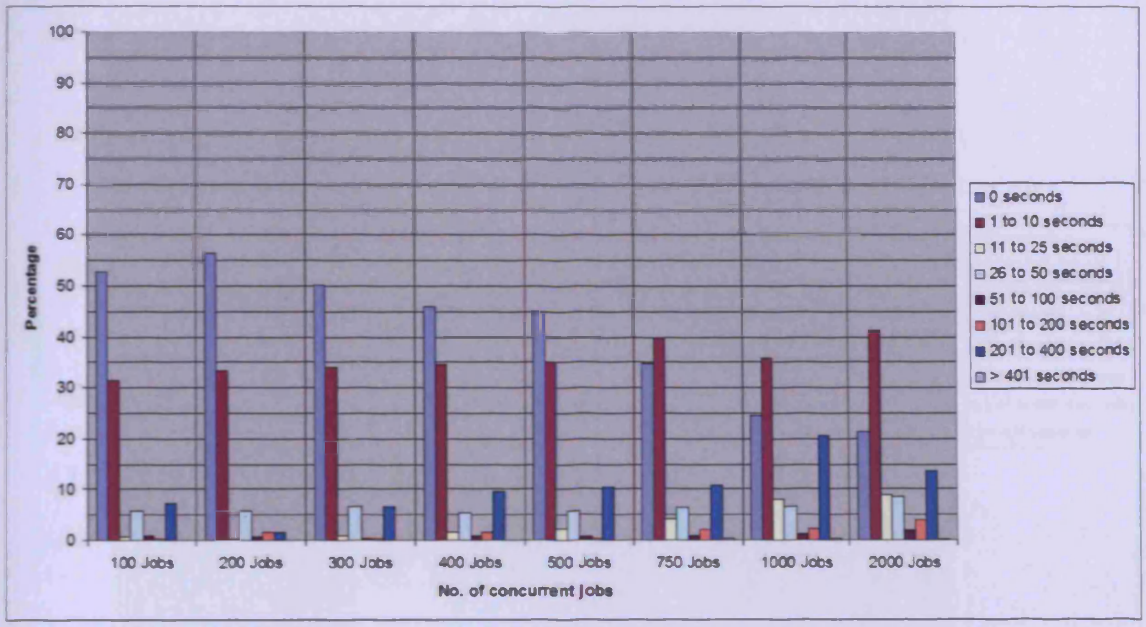
Figures 2.2 and 2.3 depict the execution times of all jobs that constituted this analysis. As expected the vast majority of jobs are recorded as requiring 0 seconds or less than 10 seconds (approximately 70%). One very noticeable characteristic is that as the queue size increases the number of jobs requiring 0 seconds decreases and the number requiring 1 - 10 seconds and 11 - 25 seconds increase. This would suggest that the overhead caused by starting a job increases, as the size of the local queue increases. This is an interesting development because, typically, users submit all their jobs at the same time. Yet, these results suggest that this is not a good idea. Instead, to improve efficiency an equilibrium between the number of jobs in queue, to ensure workstations are allocated, and the effect that the queue size has on overhead must be retained. This becomes even more important when jobs are short. One other factor that must be considered is the time required to allocate a workstation to a specific job. As the number of jobs waiting to be assigned a workstation increases the demands placed upon the scheduler also increase and therefore allocation requires more time per job. Figure 2.4 shows how the average execution time of jobs and the standard deviation also increases with an increase in the size of the job queue.

One simple answer to this problem is in job clustering, where small jobs are grouped into a single job cluster. A job cluster is then submitted to Condor as a single job, which returns one or more results. Clustering has challenges associated with it and the main challenges are: determining cluster size, recognizing affinity in jobs to cluster similar jobs together and thus reduce data transfers

[2]In other words, the Condor queue size is varied for each experiment.

[3]The manager has read only access to the Condor logs to ensure that the Condor Shadow daemon has priority access to the job logs. Thus, preventing bottlenecked access to the log file.

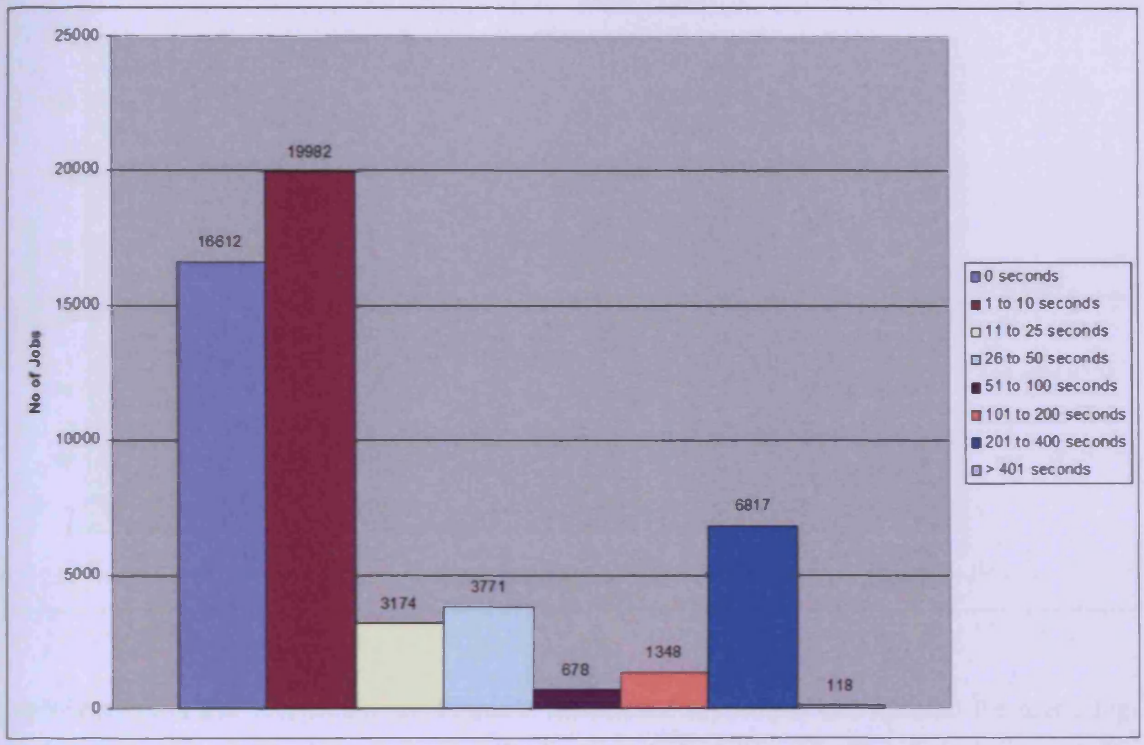
Figure 2.2: Frequency of execution categories for a varied number of concurrently submitted zero length jobs



and failure recovery. This analysis has identified a new problem that must be considered if clustering were put into action. This is manifested by the moderate amount (approximately 13%) of jobs that required more than 200 seconds to complete and a very small percentage (0.22%) that required over 400 seconds. These may seem marginal quantities, but when the amount of time required by these jobs to complete is taken into account a very different picture emerges. Figure 2.5 illustrates the total CPU time that each of the runtime categories consumed. The detrimental effect of the jobs that required between 201 and 400 seconds is significant (nearly 500 CPU hours), as they represent 78% of the total recorded CPU time for all jobs (disregarding the time jobs spent in the queue). Note that very few of these jobs failed. In total, 9 jobs (0.0017%) failed and of these only three required more than 400 seconds.

Given that most jobs required between 0 and 10 seconds, *normal* behaviour is within these bounds. A further 13% required between 11 and 50 seconds, and 4% between 51 and 200 seconds. The jobs above 200 seconds can be classified as runaway jobs because there is no information in the logs to suggest that these jobs encountered problems. This would suggest that such jobs are the effect of factors that are independent from the job itself. In other words, the fact that these jobs runaway is not related to their complexity or to some environment precondition not being met. Instead, it can only stem from either Condor, the submit node or the host workstation. The connotations this has for job clustering are troublesome, on the one hand clustering jobs improves efficiency as overheads become less influential upon performance. One the other, given the number of jobs that have runaway and their effect upon makespan, what indicators are there to distinguish between an uncaught failure and a runaway job?

Figure 2.3: Execution time categories of all zero length jobs



To answer this question, a little more information on the most extreme jobs is needed. Of 52,500 jobs, 118 jobs required more than 400 seconds, 19,982 required between 1 and 10 seconds, but the latter used less CPU time. The job with the highest recorded runtime was 3,493 seconds, or approximately 58 minutes. There is no information available in the Condor log to explain why this occurred. It is possible the remote workstation's log could have given some indication, but this is neither easily acquired[4] nor likely to retain the information long enough to be of use.[5] Throughout this analysis the worst recorded runtime was 7,205 seconds, approximately 2 hours. This job also failed, as Condor lost contact with the host workstation, the likely cause being a power down of the workstation. In brief, there are little or no indicators to identify a runaway job, unless some estimate of the execution time of the job is known. Even in this case the user must then write a manager to monitor all jobs, provide estimates and define rules or behaviours which can be mounted against such jobs.

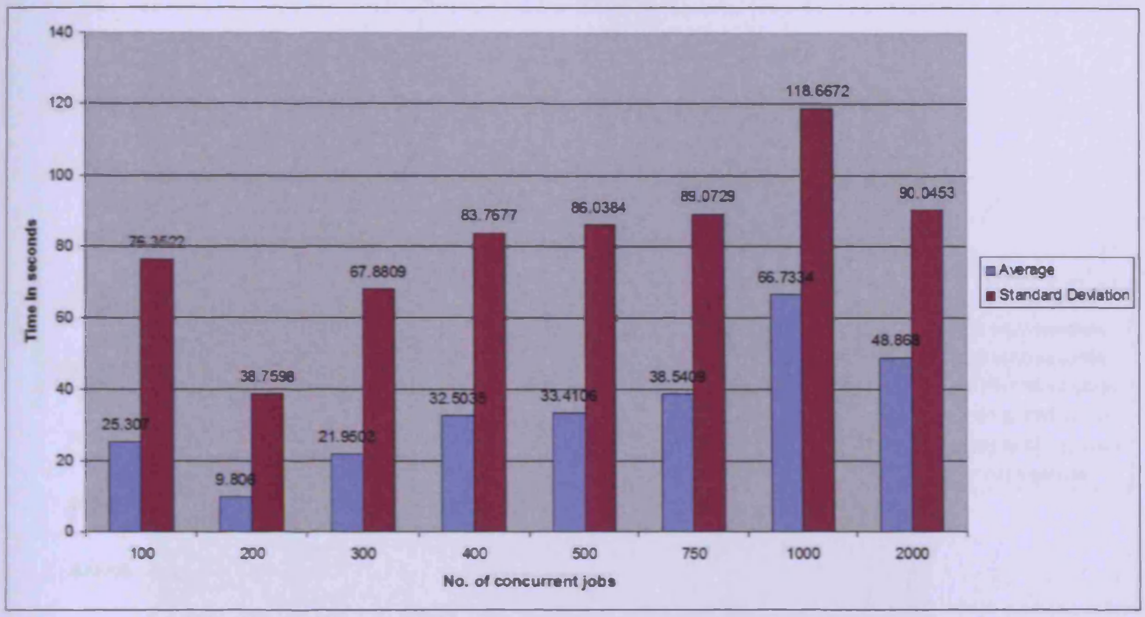
This does still leave one problem: Condor logs are not updated in real time, so this kind of reactive analysis and steering can create unfortunate situations. For example, a job can be killed even if it has finished.[6] When a workstation finishes a job, it cannot start the next until the Condor

[4]There are three ways of acquiring this data: **(1)** Logging into the workstation manually, however note that workstations are not co-located with the submit machines. **(2)** Remote access to the workstation. However, having the access to do this even for an administrator is unlikely and may not be possible at all. **(3)** Use a Condor job to access the log and transfer it to another workstation.

[5]Workstation logs have a predefined size limit, when a log gets too large older data is deleted.

[6]In such a case no results would be returned.

Figure 2.4: Category Statistics

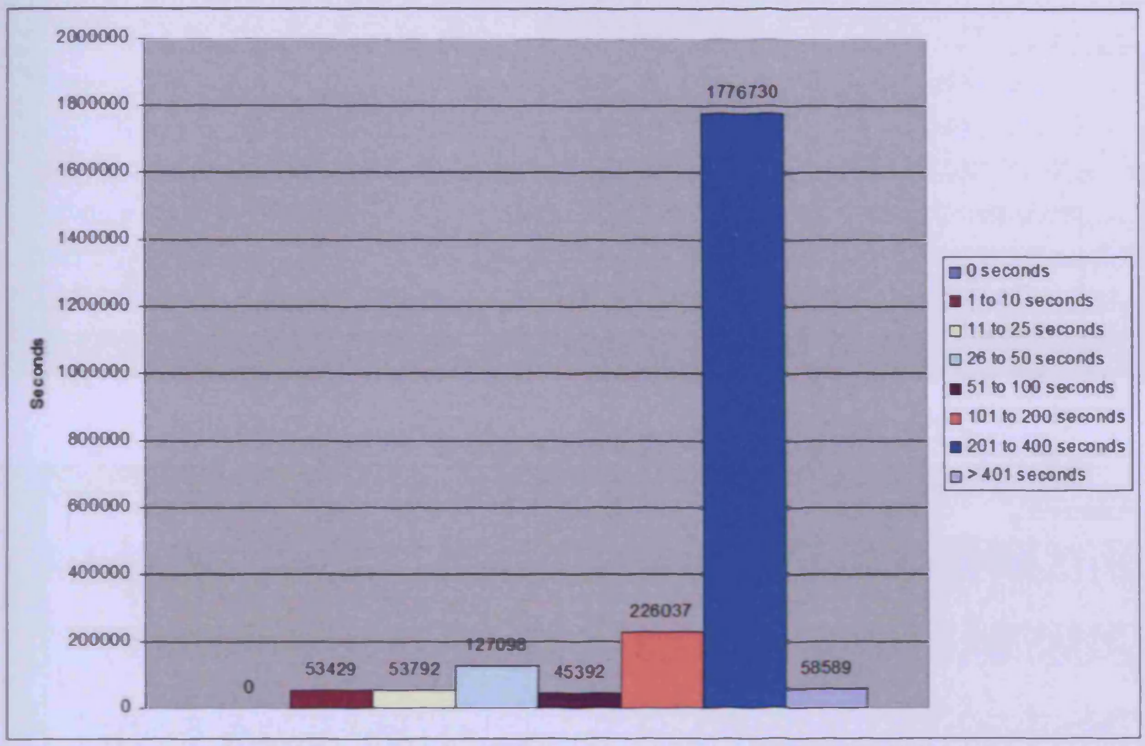


shadow daemon has vacated the workstation, transferred any output and updated the user's logs. At this point, discrepancies between what the logs say and what is actually occurring are possible, sometimes significantly. What this means is that a job may take X time units to complete, but the log may not reflect this for X + *k* time units, where *k* is the delay in updating the log. Hence, using real time analysis of job state using only the Condor logs can lead to reactive measures that do not consider this delay, and thus kill a job which has already finished, but not yet checked in its results.

If real time diagnostics cannot be used reliably, then the next natural option is retrospective profiling. Figures 2.6, 2.7 and 2.8 illustrate an execution cycle of experiments with 100, 500 and 1000 jobs respectively. What is especially interesting in these graphs is the repetitive cycle of poorly performing (runaway) jobs, indicated by the blue line (average runtime of all jobs currently running). This has occurred in every experiment, but with varied frequency and amplitude.

The first question that is raised by these graphs is whether the same workstations are in use. If so, then do these workstations consistently perform badly, and therefore cause their jobs to consistently runaway and detriment performance. Surprisingly, this is not the case. If this had been the case, workstation profiling could have been used to instruct Condor not to use poorly performing workstations. In Figure 2.6 61 different workstations were used, and 32 jobs (32%) required more than 200 seconds to complete. The maximum number of jobs that a single workstation completed was 5. This particular workstation had 3 jobs that took over 200 seconds to complete but also one that took only 1 second. In Figure 2.7 62 jobs (12%) required more than 200 seconds to complete. The workstation with the highest number of >200 second jobs, in this case 3, also performed 6 jobs with a recorded runtime of 0 seconds. In Figure 2.8 158 jobs (16%) required more than 200 seconds to complete. The most >200 second jobs performed by a single workstation was 6, the

Figure 2.5: Total time required for each of the empty job categories



same workstation also performed 10 jobs requiring 2 seconds or less.

The next question that arises is: do runaway jobs occur sequentially on a workstation? In other words, if a job runs away, is it likely that the next job will do the same? Again, surprisingly, the answer is no. 787 workstations were used during this experimental analysis and of these 725 performed more than one job. Of the 6,935 jobs that required over 200 seconds, 1,278 (18%) were sequential occurrences when the time between occurrences is not considered. The time is not considered because it is difficult to know at what point the gap is too large. However, note that during this time a job from another Condor user could run and the workstation could be reclaimed or restarted, which may affect the state of the workstation. Without a monitoring system such as Hawkeye or Quill in place it is difficult to ascertain whether this be the case. There are only two possible other sources capable of providing this information: **(1)** the log files of all other Condor users, and **(2)** the log file of the Condor workstation. Both are difficult to acquire, but as previously mentioned the latter is also transient.

Of these two options the latter was employed, as it may provide some insight as to why some jobs runaway. To obtain the relevant log file a Condor job is used to initialize an FTP session from the workstation and then transfer the log to a central point for analysis. Not all logs could be

Figure 2.6: Execution Profile of 100 Empty Jobs

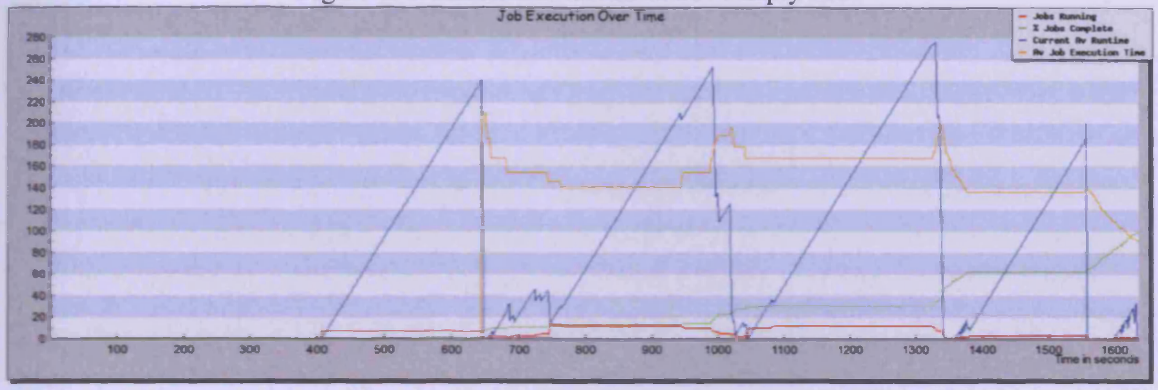
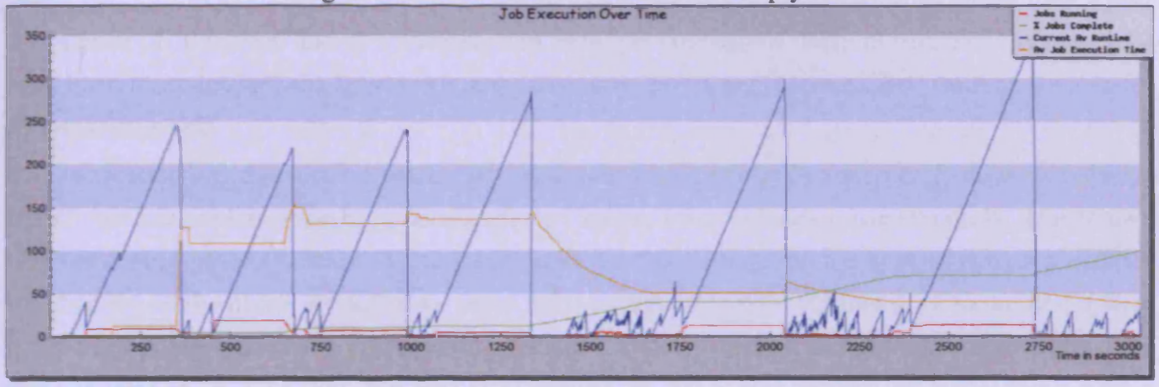


Figure 2.7: Execution Profile of 500 Empty Jobs



collected, and there are two reasons for this: (1) workstation identification[7] and (2) availability.[8] In total 50% were obtained, but this relates to a small representation of the total number of jobs (approximately 9,500 or 18%)[9] that this analysis involved. Whilst this is not representative of all jobs, no better solution is possible with the current architecture, without administrative access to all workstations. Nevertheless, enough evidence[10] is present to determine that another user's job be-

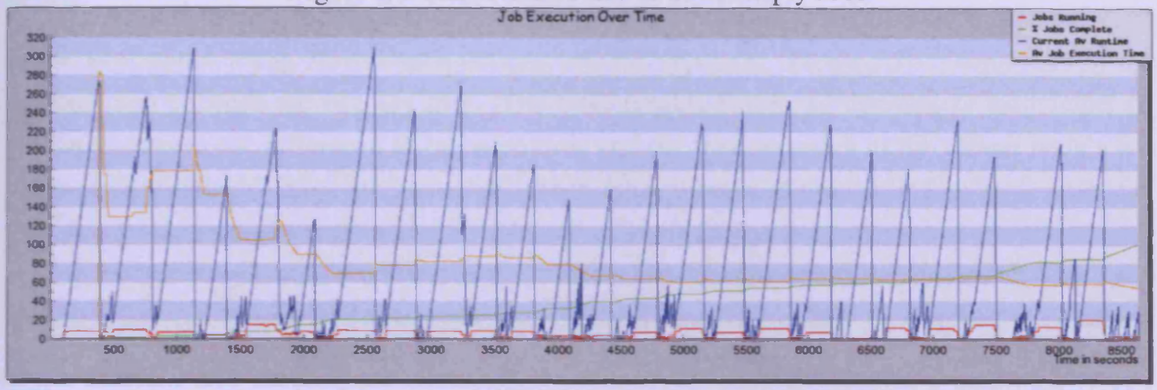
[7]The difficulty here is the method by which workstations are identified. A workstation has 4 identifiers, (1) a MAC address, (2) an IP address, (3) a DNS name, and (4) a NetBIOS name. The first three are self explanatory, but the last is not. It is the name assigned to a workstation by the network management software and it is typically, the MAC address prefixed by an X. For some reason not all schools and departments follow this naming scheme, giving rise to an inconsistent naming schema, where repetitions can occur. This name is also presented when the windows command `hostname` is executed. Condor uses this command to identify a workstation, and uses its results in the workstation's ClassAd under machine name. When a user wishes a job to run on a specific workstation they must provide this name. In itself this would not be a problem, but when Condor assigns a job to a workstation it identifies that workstation in log files with its IP address. The University does make the database which provides the map between IP addresses and NetBIOS names available to users, but only to network administrators. In order to acquire this information the user must extract it from the workstation's ClassAd, which is only available when the workstation is connected to Condor.

[8]Not all workstations were available within a week of the experiments completing. It was assumed that after this time little or no useful information would be contained in the log.

[9]Note that even with 100% representation of workstations it is very unlikely to achieve 100% of the job records. This is primarily because of the amount of jobs some machines have performed, the time span for this analysis and because old data is removed from the logs periodically.

[10]The log in use is the *StarterLog*, which details all jobs that have been started, within the monitored time frame. The documented data of interest is: when a job was submitted, which submit node was in use, when the job finished and what

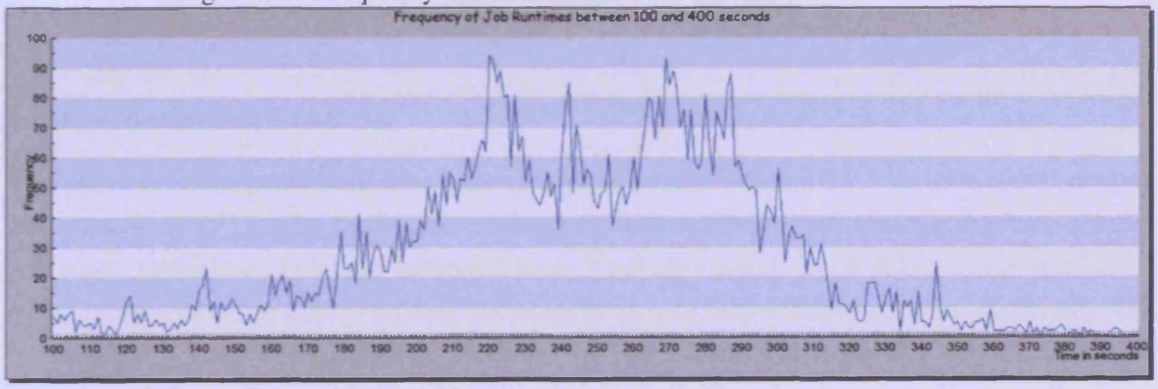
Figure 2.8: Execution Profile of 1000 Empty Jobs



tween two jobs in this analysis cannot be linked to runaway jobs. All possible combinations of jobs are present i.e. 0 second and up to 468 second runtimes are present. Both normal and runaway jobs have been preceded by jobs from other users that have been long running, short running, successful and unsuccessful.

One resounding questions remains: why do some jobs runaway in a seemingly random fashion? Many possible explanations have been explored above, yet no concrete reason can be determined. One remaining possible reason is that the Condor worker daemon on the host workstation updates the shadow, on the user's workstation, every 300 seconds. It is possible that prior to the first update the job is waiting for further information or that the update is used as a synchronization point. The average time of all jobs, that fell into the category of 201 - 400 seconds is 270 seconds with a standard deviation of 35.9. Within these bounds this is a possible explanation. Figure 2.9 shows the frequency of each runtime between the values of 100 and 400 seconds. If the daemon updates were a direct cause of runaway jobs a peak should be visible around 300 seconds. Yet, this is not the case, as peaks are present around 220 and 270 seconds. Other possible reasons beyond startup overheads and variations in workstation load are difficult to conceive.

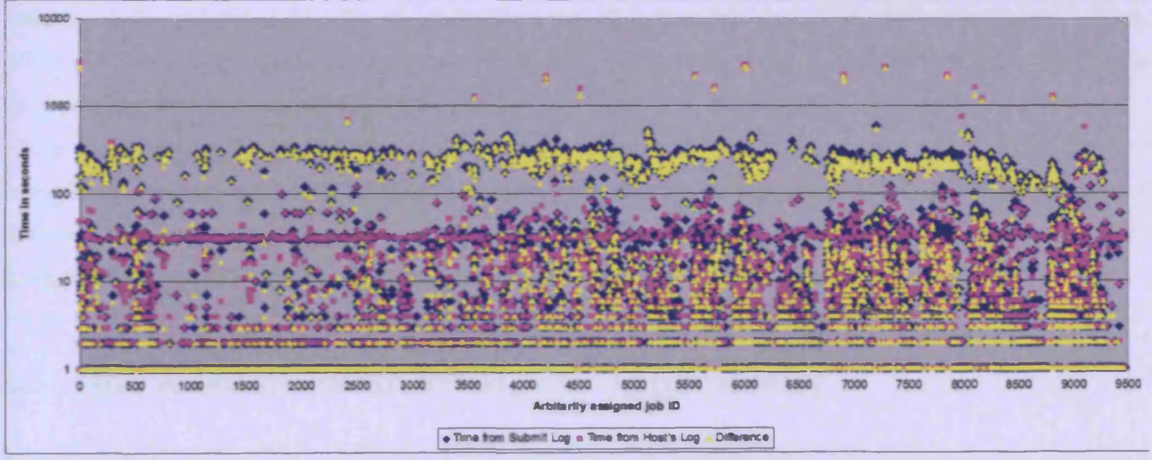
Figure 2.9: Frequency of Job runtimes between 100 and 400 seconds.



exit code the job generated. This data is in a very different format to the logs generated by a submit node, but the same approach for generating a job model can still be used.

An interesting result came from the workstation logs with regard to a job's execution time. When the execution time recorded in the submit node's log is compared to the execution time recorded in the workstation's log, some contradictions become apparent. Figure 2.10[11] demonstrates the contradiction between the two log files. The most notable feature of this data is the number of overlaps between the difference in recorded time and the time recorded in the submit node's log (16% of the available data). In these cases the difference between the two logs is more than one standard deviation of the runtime for all jobs (see: Figure 2.11). In many cases the workstation's log reports that the job required between 0 and 10 seconds to complete when the submit node's log records a job that qualifies as a runaway job. Of the data available 22% of all jobs do not have a consistently recorded runtime[12] in both logs. One final observation can be made from this data. In rare cases (1%) the workstation's log has recorded a significantly longer runtime than that of the submit node. This eventuality gives reason to the disagreement between the realtime job runtime and that which is recorded in the Submit node's logs. Why this happens, however, is beyond the scope of this analysis and would require further and more detailed investigation into the Condor monitoring systems.

Figure 2.10: Contradiction in recorded execution time between a submit node's log and condor node's job log.



2.2.3 Concluding Remarks

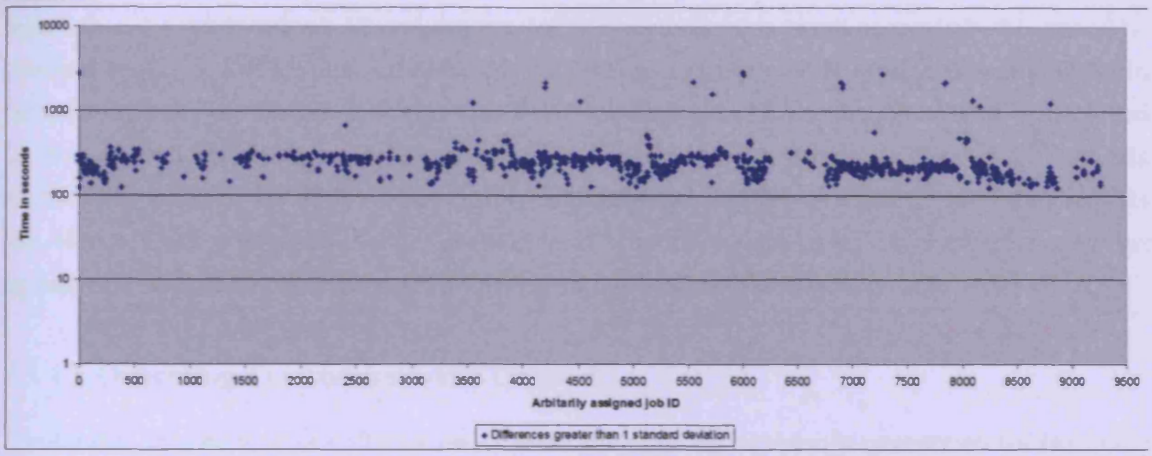
This analysis has identified that jobs have a startup overhead of around 6.5 seconds and a standard deviation of 12.5,[13] which for longer running jobs is insignificant, but for shorter jobs (seconds) will play a critical part in a job's execution time. It has also been found that as the size of the local queue increases, so too does overhead. Increases were also seen in the average execution time and the standard deviation of runtime as the queue size increased. This work has also identified that

[11]Note the logarithm scale of the y-axis

[12]A tolerance of 10 seconds is used to determine consistency.

[13]These values are calculated incorporating only jobs which required less than 100 seconds to complete.

Figure 2.11: Differences from Figure 2.10 whose value is greater than 1 standard deviation of runtime.



Condor jobs can runaway for reasons that are independent of job complexity and when this occurs it can be very expensive. There is also an increased chance that a job will runaway if the local queue is larger.

Clustering would not necessarily be of benefit in such a situation thanks to the penalties of runaway jobs and the difficulty of identifying them; remember that of 52,500 jobs only 2 failed. What would be beneficial is to ascertain stability prior to commencing a cluster of jobs, but this is not possible with the current Condor system.

In this analysis job complexity is superficially simple. When more realistic job complexities are considered, one question really stands out from the results of this analysis: If overhead increases with the size of the job space, even with the simplest of jobs, how can a runaway job be distinguished from a faulty workstation or an uncaught error?

2.3 Job Administration for Matlab Applications

As mentioned in Section 1.3 licensed software such as Matlab does not always fit the model of Campus-Grids. In the case of Matlab, licensing issues are the problem, but this can be alleviated using the Matlab Compiler to create license-free stand-alone applications. This not only removes the need for multiple costly software licenses,[14] but also means that workstations in other Campus-Grid domains can be utilized without the need to transfer and manage software licenses. A compiled Matlab application has only one software requirement, which is the availability of the Matlab Component Runtime (MCR), which does not require a license. The MCR installation is not complex, as it does not need to write to the Windows Registry or require a specific installer. Installation is simply the extraction of a `zip` archive and additions to the search path. The latter can be done

[14]For example, at the time of writing, the educational cost of a single Matlab license is 375. In addition, each toolbox costs 150 (there are 35 of them). The exceptions here are the Matlab compiler, the Calibration Toolbox and SimBiology each of which costs 375. Therefore a single instance of Matlab costs between 525 and 6300.

during a Condor job's pre-staging, if more than one version is available.

The problem is the way in which a workstation's state changes with time. For Condor users, this means that if a workstation's ClassAd states MCR availability this is not necessarily the case. The problem here is that if a job is submitted to a workstation stating MCR availability, and this is in fact not true, the job will fail. In such a case Condor will not reschedule the job because it has failed for reasons which Condor cannot associate with the workstation. Similarly, Condor will use this workstation again in the future, and possibly immediately, even for an exact copy of the previous job. Hence, Condor assumes that the job is at fault, which is not the case. The consequence of this assumption, as will be demonstrated in this Section, is that job blackholes emerge.

2.3.1 Operating Environment and Context

Availability of a service or software entity can change or be incorrectly announced for two main reasons. **(1)** An error occurred during installation, but the ClassAd was still updated to reflect availability. Examples include file corruption, mishandled transfer error, incorrect directory structure, installation termination/restart, users interrupting the installation process by logging in and/or out. **(2)** The state of the workstation has changed since the ClassAd was last updated. This can occur from multiple sources but it is most likely that another installation process has corrupted or unintentionally changed the initial installation, or an administrator has modified the workstation.

On Windows workstations there are many methods of software distribution such as Peer-to-Peer and BitTorrent, customization of the Windows installer, which can include networks transfers, resource virtualization (e.g. Xen [71] and VMWare [72]), where a virtual machine is transferred with the software already present and correctly installed, and Network administration tools (e.g. Novell's ConsoleOne) that push software out onto workstations. However, regardless of the method of distribution it cannot be expected that campus wide software distributions be error free and uninvasive on software already present. It is also common place that there be no single authoritative administrator for a given workstation, but many, all of whom can update a workstation without notifying the others. The result is that no persistent list of software is upheld, making it very difficult to identify the provenance of a software failure. Similarly, no fault tolerant procedures to ensure that an installation has been correctly performed are in place, unless an administrator includes customized functionality explicitly for this reason. On Linux, however, the installation of software, even on a large scale, is by comparison almost effortless. This is primarily due to the software and dependency database which is unheld by the operating system. Here, simple system tools such as *apt-get* are of great utility for the distribution of software.

Novell's ConsoleOne is used to install software on Windows workstations of Cardiff University's network. The process to install software involves the creation of an application object, which relates to a customized Windows batch script or an automatically generated script built using some other administrative software tool. This script is then flagged with meta-data to describe its actions, when these actions are to be undertaken and any other application object dependencies. Typically, application objects run only once, unless an error is detected by ConsoleOne. The accuracy of detection, however, is dependent on the manner in which the software installation process reports errors. For example a custom Windows batch script can complete with a satisfactory exit code of zero even when an error has occurred, if it is not correctly handled. The use of ConsoleOne enables entire domains and subdomains of workstations to receive software updates remotely. However, as previously mentioned ConsoleOne provides no inherent support for the analysis of whether previously installed applications have been damaged, overwritten or if the installation process was itself successful. This is unfortunate, as in the hands of an expert ConsoleOne is very powerful and useful. Yet, this is not always the case, as the level of training received by administrators is inconsistent. In some cases the training provided is insufficient for the background and adeptness of the recipient and the consequences of this are demonstrated by this analysis.

When a lack of intuitive fault tolerance is combined with an environment which changes dynamically from multiple semi-independent sources, often without warning, workstation consistency cannot be assumed. A similar idea is presented in the four Condor admonitions [39], but this provides no solace to the user, who must inevitably administrate their jobs. In addition, Condor provides no distinction between a job that has failed due to user error such as a programming fault and one which has failed due to workstation error such as a change in software availability or incorrectly configured software. Instead, an exit code is provided in the Condor log, which, without a detailed understanding of the application in question, is relatively uninformative.

The MCR was pushed out by ConsoleOne onto several domains of the Condor pool, and of the 2,500 Condor workstations. At the time of this analysis, approximately 20% announce the availability of the MCR in their ClassAds. This analysis will demonstrate the challenges facing users in administrating their Compiled Matlab jobs. These challenges stem from incorrect ClassAds, where in fact no installation is in place and those where the installation is incomplete or faulty. Such scenarios will result in a failed job. This analysis will also build upon the previous analysis by highlighting the increase in startup overhead from initialising the compiled Matlab engine (MCR) for a non-trivial application. However, instead of performing any analysis the application will terminate the MCR upon startup. This will enable Condor to record the time required only to start a job, initialise the environment, start and then shutdown the MCR and vacate the workstation. The effect of this action is to be able to determine the overhead incurred by Condor to launch a compiled Matlab job.

A compiled Matlab application is an executable and an archive of compiled Matlab code packaged into a `CTF`, which is comparable to a `zip` archive. It may also be accompanied with data and other application-specific libraries. The executable initialises the MCR and begins execution. The `CTF` contains an application's (compiled) Matlab code and some other deployment and application specific Matlab functionality, which are created by the Matlab compiler. The time that is being measured includes the time to extract the CTF archive, so that job execution can commence. Therefore, job time in this analysis is based upon a certain amount of workstation performance as well as external Condor factors.

2.3.2 Experimental Analysis

The aim of this analysis is to use as many different MCR-enabled Condor workstations as possible, by replicating an application with a large number of jobs: 5,000 in this case. To do this the application manager from the previous analysis has been modified to take into account the results from the previous analysis. Instead of submitting a large quantity of jobs at once it will aim to retain between 100 and 200 jobs in the queue until all jobs are completed. As yet, it will not try to identify poorly performing workstations or help Condor in steering the application, by killing and resubmitting jobs which have obviously runaway i.e. run for hours or days. There are two reasons for this choice. Firstly, by not aiding Condor, with application specific knowledge such as expected runtime or monitored performance leading to the same knowledge, the true extent of the challenges can be demonstrated. Secondly, because of the likelihood of not having real time job information, decisions based upon unreliable data are, at this stage, themselves not reliable. Instead, some pre-staging functionality has been incorporated into the job to identify inaccuracies in the ClassAds.

To ensure that a workstation has a valid installation of the MCR a simple check can be performed; if it fails the job can be forced to fail, with a specific exit code, which can be captured by the log readers. The employed test verifies the MCR installation by checking the size of the MCR directory, where it is located and the number of files within the directory. This is a simple yet effective way to test the MCR's integrity. In all tests of this type there is a common trade off between accuracy and cost. This test will not permit the use of any installation which does not contain the correct number of files or have the correct data size. Whilst this provides some assurance it cannot guarantee that the contents of the directory hierarchy are in fact correct, only that they have adequate size and that there are enough of them. In this sense, the only real test that can provide a concrete answer is the job itself. However, this test is a quick and effective way to catch workstations where the MCR installation is incomplete.

It goes without saying that this analysis will use a much reduced proportion of the Condor pool, as only 20% are available for Matlab-related jobs. Considering that the pool consists of around 2,500 workstations and that in the previous study only around 31% were used, it will be difficult to obtain representative results of the whole MCR-ready portion of the pool. However, this is likely to be more representative of user experience. A job space of 5,000 requires a few days to complete giving an increased potential to use a larger variety of workstations. It has been repeated 5 times (making 25,000 jobs), to try and provide as rich a representation as possible.

2.3.3 Results

The first attribute of this analysis and quite possibly the most important, is the ratio of failed jobs to successful ones. In total 45% of all jobs failed the MCR integrity test, performed during the pre-staging of the job. Note that no job, which passed the pre-staging test failed, giving an incorrectly configured MCR as the reason. However, all the jobs which did fail this test, were run on only 11% of the workstations used in this analysis. This means, that Condor ran the exact same job on a small subset of workstations over and over again, where each one consistently failed, without Condor realizing or reacting in any way. Whilst, as a user, this is not a difficult problem to fix in the short term,[15] it is, however, challenging to administrate over time. The challenge comes from the dynamic nature of network where workstation identifiers such as IP addresses, DNS names and NetBIOS workstation names can change spontaneously and often without notification as can configurations and installed software.

During this analysis some interesting changes in workstation performance occurred, which demonstrate the challenge of job administration. Two workstations, both of which had a track record of consistent jobs failures, 524 and 161 consecutive occurrences, suddenly changed. Within a space of a day and half, for the first and less than a day for the second, the workstations' configuration changed sufficiently to enable them to pass the MCR integration test and complete all subsequent jobs successfully. Note that both these workstations are in the same IP range and possibly in the same computer lab and that both changes occurred between similar dates. Contrary to these two workstations at the same time a third workstation experienced the opposite, going from 142 consecutive successful jobs to 103 consecutive failures. It was also in the same IP range. This is a clear demonstration of how workstations can change dramatically over time. However, had the user used persistent exclusion (such as static blacklisting) the two workstations which changed for the better would have gone unnoticed, and therefore not capitalised upon for future jobs. Had this been the case an ever shrinking pool of available resources would develop over time.

On average failed jobs ran for 6 seconds, but in line with the last analysis some required a similarly lengthy period of time. In fact, the minimum runtime for one of these jobs was 0 seconds, illustrating the unobtrusive nature of the pre-staging MCR test, the maximum was 1100 seconds and the standard deviation was 21. Of course, there were also some anomalies, which were not used to calculate these values. One job in particular, which eventually failed as a consequence of being *unsuspended*,[16] ran for over 28 hours. It was not rescheduled by Condor.

The successful jobs ran on average for 130 seconds, with a minimum of 45, a maximum of 2782, and a standard deviation of 56. Here similarly expensive runaway jobs were experienced. These jobs are several orders of magnitude greater than in the previous analysis, the maximum time required for a job to complete was 4 days and 5 hours. There is one other attribute of this analysis, which was not seen in the previous. It stems from the number of times Condor needed to schedule a job before it successfully completed or failed. The maximum number of times this occurred for a single job was 5, the average is 1. Essentially, what happens is that a job would runaway, be suspended, then evicted and rescheduled. This process could then be repeated several times. What this means is that some workstations have not managed to run a job to completion regardless of whether their jobs fail or not. Such a job will be referred to as a terminated job, to provide a distinction from a failed job.

In total, 47 workstations have been allocated jobs but have not managed to steer one to either

[15]The Condor submit script can contain a blacklist.

[16]Unsuspension occurs when the user logs out of, or otherwise releases, the workstation within 10 minutes of interrupting (suspending) a job.

a failed or successful MCR initialisation. This relates to 76 jobs (<<1%) and nearly 11 days of CPU time. Some of these workstations are, however, unfortunate in that there is a high probability that they have been used only once and that their job was terminated during either the pre-staging check or during MCR initialisation. To eliminate such a scenario, any terminated job that required less than 20 minutes has been removed from the set of terminated jobs. This reduces the number of workstations to 24 and the number of jobs to 38, but it does not significantly impact upon the total CPU time used; now 10 and a half days. Of these workstations, 8 have recorded two or more occurrences.

Two clear possibilities can explain why terminated jobs occur. The first, is that the job simply runs away. The second possibility, is that the MCR integration test is succeeding, but that an underlying error in the MCR installation remains, but, which has not been detected by the test. When this occurs within an interactive context the MCR will alert the user with a message box detailing the problem. On Condor, however, a message box will either be displayed to the Condor virtual desktop, where it shall never be seen, or to the desktop of an idle workstation. In the second case only the next user of the workstation shall be aware of a problem with a software component they have probably never heard of, and the fault will go unreported. The problem with the use of message box signalling is that the MCR will not shutdown until this message box is closed, or the job is forcibly terminated by Condor. This means that the job cannot complete normally without user interaction, which is likely to not be received. Hence, the job is eventually evicted.

There is a distinct possibility that both of these scenarios are represented in the set of terminated jobs. For the 38 jobs in question the minimum runtime is 25 minutes, the average is 9.5 hours, with a standard deviation of 6.5 hours, and the maximum is 2 days. Without physically observing the workstations in question it is not possible to determine which of the two eventualities a workstation is experiencing. Even the workstation logs cannot provide adequate information to differentiate between the two possibilities. Thus, the line between a job which is stuck, and one which has runaway, is blurred.

2.3.4 Concluding Remarks

This analysis has quantified the overhead of initialising the MCR for a large Matlab application, which is, on average, 130 seconds. Realistically this overhead should not increase, it would take a great deal of Matlab code to do so. The application used in this analysis contains over 1,500 user functions, to say nothing of the Matlab code that is required for these functions to operate. Given the time required to initialise the MCR, how can short jobs fit into this context, especially considering the number of failures and the unpredictable nature of the resources? One of the challenges, that must first be answered, is the need for a clearer differentiation of the distorted and ambiguous characteristics which separate runaway jobs and uncaught failures. Only in solving this can job management strategies be put into force to enable shorter jobs.

One factor that has, still, not been considered in this analysis is job complexity. Again, the test job in this analysis was superficially simple. What this means is that errors that stem from programmer error, or other code faults have not been taken into account. Put simply, these attributes of jobs make management more complex. This is because not only are workstations volatile and dynamic, but programmer error can add new reasons for job failure. One scenario that is not inconceivable is the user forgetting to terminate their job, using the Matlab `quit` command. The Matlab Engine can be kept alive even after execution has finished, if a figure or some other visual output remains open. In such a case the job would appear to have runaway, and would eventually be evicted by Condor. Such possibilities make job management even more complex.

2.4 Performance Analysis for Matlab Jobs of Various Lengths

In this last study the performance of the Campus-Grid will be examined for jobs of a known length, so that it can be clearly demonstrated where the performance of the Campus-Grid is inadequate and so that a benchmark can be created for current performance of the Campus-Grid. The latter will be used again in Chapter 7 to compare the Campus-Grid's performance to the approach and prototype documented in Chapters 4 – 5.

The previous two studies have highlighted that in the context of a Campus-Grid it is impossible to create the same execution environment twice; this has also been noted in [37]. Therefore, due to the heterogeneity of context, some control conditions are required to enable meaningful and objective analysis. In the literature this is typically achieved through the use of simulators e.g. GridSim [73]. However, for analysis such as that performed in this Chapter, simulators would be inadequate, as they require a depth of knowledge of the environment that cannot be expected. In order to retain an objective and accurate perspective toward the heterogeneity and volatility of the execution environment the evaluation of a distributed system needs to be run in the real environmental context. Therefore, to present the performance of the Campus-Grid for later comparison synthesized applications are employed.

A synthesized application is one which has been contrived to suit the purposes of the experiment being performed. This is achieved through a strictly controlled domain to enable fairer (comparative) analysis. The number of workstations available varies with time and of the workstations available, some are more powerful, have more memory, faster network connections, hard disks etc. The controlled domain of a synthesized application means that many of these factors are inconsequential. The job space for a synthesized application is a collection of dummy jobs with a pre-defined execution time and data.[17] In other words, jobs are homogeneous and made independent of workstation performance. To achieve this a job's execution time is defined using the Matlab `pause` function, which causes the Matlab engine to *sleep* for a specified period of time.

Predefining the job space in this manner means that the identification of the overhead the job receives is easier and more accurate. For example if a job has a known length of 60 seconds and its makespan is 100 seconds, it is clear how much time is required for the administration of this job's

[17]Data can be easily be fabricated, for example using a lossless and uncompressed image format to enable precise data quantities.

execution. A synthesized application therefore ensures that the implementation of the job does not cloud the performance data. It is subject only to heterogeneity during the initialization process of Matlab that a workstation must undergo before a job can be executed. The reduce the impact of this process to performance synthesized applications are run multiple times to ensure a diversity in workstation performance and enable a general picture to be drawn. With this generalization and the isolation of the effect of heterogeneity, only the resource management model directly affects a job's makespan. Resource volatility is experienced by both approaches and can be generalized through the repetition of applications.

The consequence of the lack of environment consistency means that each application cannot realistically be expected to acquire the same number of workstations. This is even the case when applications are run sequentially. In addition, the time of day, week, month and term also impact upon the performance of the Campus-Grid. The effect of these conditions means that making direct comparisons between the Campus-Grid and other approaches is difficult. However, repetition enables a global picture to be produced and also provides a gain in statistical power. Ultimately there is no way of knowing if every corner case of the Campus-Grid has been captured and realistically, this is unlikely. However, a good range of behaviour has been captured in this study to demonstrate both the volatility of the Campus-Grid and cases when performance is good. This general picture will in Chapter 7 be compared to the performance of prototype documented in this thesis to demonstrate its feasibility and improvement in performance.

The analysis in this study has two goals and builds upon the previous two studies. Firstly, it aims to illustrate that for short running jobs, the traditional Campus-Grid model does not work. Secondly, it demonstrates the effort required to steer a job set to completion i.e. 100% completion. In order to achieve 100% job completion an Autonomic Job Manager has been implemented to steer the Condor system. It is defined in Section 2.4.2.

2.4.1 Experimental Analysis

The synthesized applications have predefined job times of: 1, 60, 600 and 1800 seconds. Each of these times has been chosen to reflect a specific set of attributes. The 1 and 60 second jobs have been chosen to capture the performance of the quickest jobs that a user could feasibly wish to have. In Chapter 6, an application scenario is presented where the average job time is between 4 and 7 seconds in length. The 600 second jobs have been chosen as 600 seconds is the minimum recommended job time for a Condor job. The 1800 second jobs have been chosen to validate that the approach is fair as Condor should perform well with jobs of this length. All jobs are independent and are not associated with any data. The reason for this choice is to simplify the functionality of the Autonomic Job Manager. If data is considered as an intrinsic part of the job the Autonomic Job Manager must act at the data level and not the job level. This differentiation is important because when resubmitting jobs that have failed, the Autonomic Job Manager must be aware of the associated data to ensure that all jobs are completed. If data is not considered then the Autonomic Job Manager can be orientated toward the single goal of completing a pre-defined number of jobs. The aim of the synthesized experiments is to demonstrate the factors that cannot be controlled and their effect upon performance. Namely: the initialization time of Matlab, the number of workstations available and their volatility, data handling can cloud these metrics. In addition, the Autonomic Job Manager must be more sophisticated, but this extra functionality has little, if any, significant benefit with regard to the results.

To evaluate the performance four metrics are used: **(1)** Job Efficiency, **(2)** Job Runtime, **(3)** Application Makespan, and **(4)** Efficiency of Resource Utilization. The metrics used here are are commonplace in distributed, Grid and parallel computing, therefore enabling comparisons to other approaches. Below the motivation for each metric is presented and, if required, the means to determine its value.

Job Efficiency

The job efficiency captures the ability of a resource management approach to execute jobs successfully. This is measured by the number of attempts needed to complete the job space in relation to the target number of jobs. This metric characterizes the dependability of an approach, and 100% is the ideal target for an approach to achieve. However, a clear separation is needed between failures due to typical opportunistic events, such as user and workstation autonomy and those which are a consequence of resource volatility (see: Section 1.3.1). The first category are not failures as such but transient events which are symptomatic of the paradigm, whereas the second category are problematic and frequent failures of the Campus-Grid and directly affect the job efficiency to a much greater extent. This separation means that the cause of job inefficiency can be categorized into two classes failures due to volatility (unreliability) of the workstations and workstation autonomy (normal behaviour). The purpose of classifying failures in this way is to reinforce that most failures are due to workstation volatility and not workstation autonomy. However, note that longer running jobs will be subject to an increased probability of preemption due to workstation autonomy. Job efficiency is calculated as shown in 2.1.

$$Efficiency = \frac{Jobs\ Successfull}{Jobs\ Submitted\ -\ Autonomy\text{-}related\ Failures} * 100\% \tag{2.1}$$

Job Runtime

Job Runtime is the time required by an approach to complete a job, but it should not be confused with the predefined job time. It is the time from when the job commences execution to the time that a result is received. To deduce this value the times as recorded in the Condor log file are used.[18] Only the execution times of successful jobs contribute toward this metric.

The motivation for the use of this metric is that it is a good demonstration of the effectiveness of an approach in running a job. It specifically identifies the overhead incurred by the scheduling strategy of the resource model, as the runtime of a synthesized job is predetermined. Here, overhead

[18]Note that it has been observed in Section 2.2.2 that the time recorded by Condor may not always be an accurate representation of the actual time needed for a job to run. However, no better alternative is available.

is the combination of the costs to executing the job, staging in and out of data and software executables from the remote workstation, and if required job initialization and/or post-execution clean up. Therefore, it illustrates the feasibility of the Campus-Grid for short jobs.

Application Makespan

The application makespan (denoted as AM) is the time from when the very first job was submitted, to the time that any results of the very last job were received. This metric gives an indication of whether the use of multiple resources actually provided some benefit. Here, the results can be compared to the time required to execute the job space on one workstation to identify if there was any utility in the use of a parallel approach for the specified application. This metric could also be compared to the number of workstations in use to calculate the speed up received by parallelizing the application. However, where speed up shows the direct improvement of using multiple workstations it does not illustrate whether the resources were used effectively, this is captured by the *Efficiency of Resource Utilization* (described below).

Efficiency of Resource Utilization

The efficiency of resource utilization (denoted as ERU) characterizes how well the resource management model has used the resources that were available. It is important to characterize efficiency due to the initial problem statement presented in Chapter 1, which specified that the current solutions for Campus-Grids and Grid Computing in general cannot adequately cater for short jobs. Therefore, through this metric the feasibility of an approach can be identified. In addition, the benefit of the approach in comparison to a traditional model can be presented. The ERU is calculated using 2.2, the expected application makespan (denoted as EAM) calculation is shown in 2.3.

$$ERU = \frac{EAM}{Observed\ Application\ Makespan} * 100\% \tag{2.2}$$

$$EAM = \frac{No\ of\ jobs * Predetermined\ Runtime}{Average\ No\ of\ Workstations} \tag{2.3}$$

Due to the variability in workstation allocation over the experiment runtime the average number of workstations is used instead of the total number of workstations. This is calculated by building a time series representation of all jobs submitted to the Campus-Grid from the Condor logs, as described in Section 2.1. Then the time series data is sampled at one second intervals to determine the total number of Campus-Grid jobs running at that time. The average number of workstations is then the average number of Campus-Grid jobs running throughout the application's execution.

2.4.2 Autonomic Campus-Grid Job Manager

To enable a fair analysis of a regular Campus-Grid approach using Condor, an Autonomic Job Manager has been created. The motivation behind its development is that over time a user would develop some knowledge of the Campus-Grid and modify their submission behaviour to reflect these observations. Therefore, the Autonomic Job Manager models a "*competent*" user who is aware of the issues surrounding job management and is proactive with respect to these issues. Naturally there is also some utility in developing an autonomic Autonomic Job Manager, namely the ability to run experiments without the need for supervision.

To make the Autonomic Job Manager self-governing (autonomic) several key abilities are incorporated. Log monitors permit job monitoring and the creation of a job model, as described in Section 2.1. This model enables jobs to be (re)scheduled, terminated (for example if it runs away) and monitored for completion. Similarly, post execution this information can be used to construct a graphical representation of the execution and extract other statistics to demonstrate certain aspects of the execution life-cycle. The Condor pool is also monitored to identify how many workstations are available in relation to how many are in use and how many other jobs were in the pool at the time. This information enables the identification of the Campus-Grid resource management software i.e. Condor's ability to provide resources.

The features that have been incorporated into the Autonomic Job Manager are: (1) automated job submission, taking into consideration the nature of the synthesized jobs to be performed by the Campus-Grid, (2) log monitoring, (3) reactive job resubmission upon failure, (4) proactive job termination upon the identification of anomalous (runaway jobs) or unwanted (suspended or held jobs) behaviour,[19] (5) post execution log dissemination to identify key performance information and generate a graphical representation of the execution life-cycle, and (6) automated post-experiment cleanup and repetition to gain statistical accuracy. Note that autonomous workstation blacklisting is not incorporated at this stage.

Implementation of the Autonomic Job Manager

To perform an experiment the Autonomic Job Manager is passed as input arguments: (1) the number of jobs that need to be performed, (2) the length of the jobs, (3) the number of repetitions that should be performed, and (4) a naming standard for the Condor log files. Upon startup it checks to see if any previous progress has been made with respect to the input arguments, for example if any logs with the same naming standard have been used in the past it will check for progress in relation to the passed parameters. This enables the Autonomic Job Manager to be restarted without upsetting the progress of an experiment. Once the Autonomic Job Manager has identified the current experiment to manage it will inspect its progress, and then take the necessary action to steer the experiment to completion.

When a new experiment is started the first task that the Autonomic Job Manager undertakes is to create the Matlab job template file; an example is shown in Listing 2.1. The template file contains the Matlab code to describe the tasks to be performed on all remote workstations.

Then the Autonomic Job Manager can create and execute the Condor submission script for all jobs. Here, it must identify and handle all Matlab-related dependencies of the job, establish the

[19]Details of how this is performed will be presented in Chapters 4 and 5.

Listing 2.1: An example Matlab job template for a synthesized job of 600 seconds to be executed by the Campus-Grid

```
pause(600); %sleep for 600 seconds
quit; %shutdown the MCR with the default exit code - 0
```

correct initialization routine for a Matlab-based batch job and highlight the specific requirements of the job with respect to the Campus-Grid workstations that should be used for the matching process. The submission script is then executed using the `condor_submit` command.

The processes that the job performs are, firstly, check that the allocated workstation is capable of running a Matlab job, through a pilot test of the workstation, as performed in Section 2.3. If this is successful, the second action is to initialise the MCR and invoke a predefined function, which acts as a entry point into the compiled application. Thirdly, the entry function searches for any job descriptor files, extracts the contents and passes the Matlab code to be executed to a meta-interpreter (meta-interpreters will be described in Section 5.1.2) specifically created for this batch mode. The meta-interpreter enables batch jobs to be executed in a uniform manner include the result gathering phases. Fourthly, results are saved into a `.mat` file with a random name and error messages are piped to the standard error stream. Should an error arise Matlab will be terminated using a specific exit code to identify that the meta-interpreter caught an error, this error code can then be picked up the the Autonomic Job Manager for analysis. Finally, the entry function terminates Matlab allowing Condor to return the results file or highlight an error.

After the experiment's jobs have been submitted to the Campus-Grid the Autonomic Job Manager begins to monitor all corresponding jobs and the Campus-Grid as a whole. The code and libraries needed for these processes are borrowed from one of the prototype components (the Condor Resource Manager, which will be presented in Section 5.3). In comparison to a regular Campus-Grid user, the Autonomic Job Manager has one useful extra ability: namely the ability to proactively identify anomalous job behaviour. Here, an anomalous job is one, which is either running away or otherwise become non-responsive or stuck in a state transition. To identify such jobs the Autonomic Job Manager employs the termination heuristic shown in Figure 2.12. This can be easily extended for other applications to consider a maximum permitted execution time rather than a known or expected execution time. Extending the Autonomic Job Manager in this way would enable other Campus-Grid users to manage their experiments and benefit from the functionality of the Autonomic Job Manager. The reasoning behind this heuristic will be presented in Section 4.4, but, in short, such a heuristic would have been capable of accurately terminating 100% of all runaway jobs and incorrectly terminated 0.03% of normal jobs for the Campus-Grid study presented in Section 2.3. A 10 minute minimum runtime is used here to prevent the heuristic, in its early stages, from terminating jobs too eagerly. This can occur, for example, when only more powerful workstations have been used and consequently the average runtime is low.

The monitoring process continues until the required number of jobs have been successfully completed. After this point any remaining jobs e.g. those which were submitted through mistake,

Figure 2.12: The Autonomic Job Manager's job termination heuristic

IF

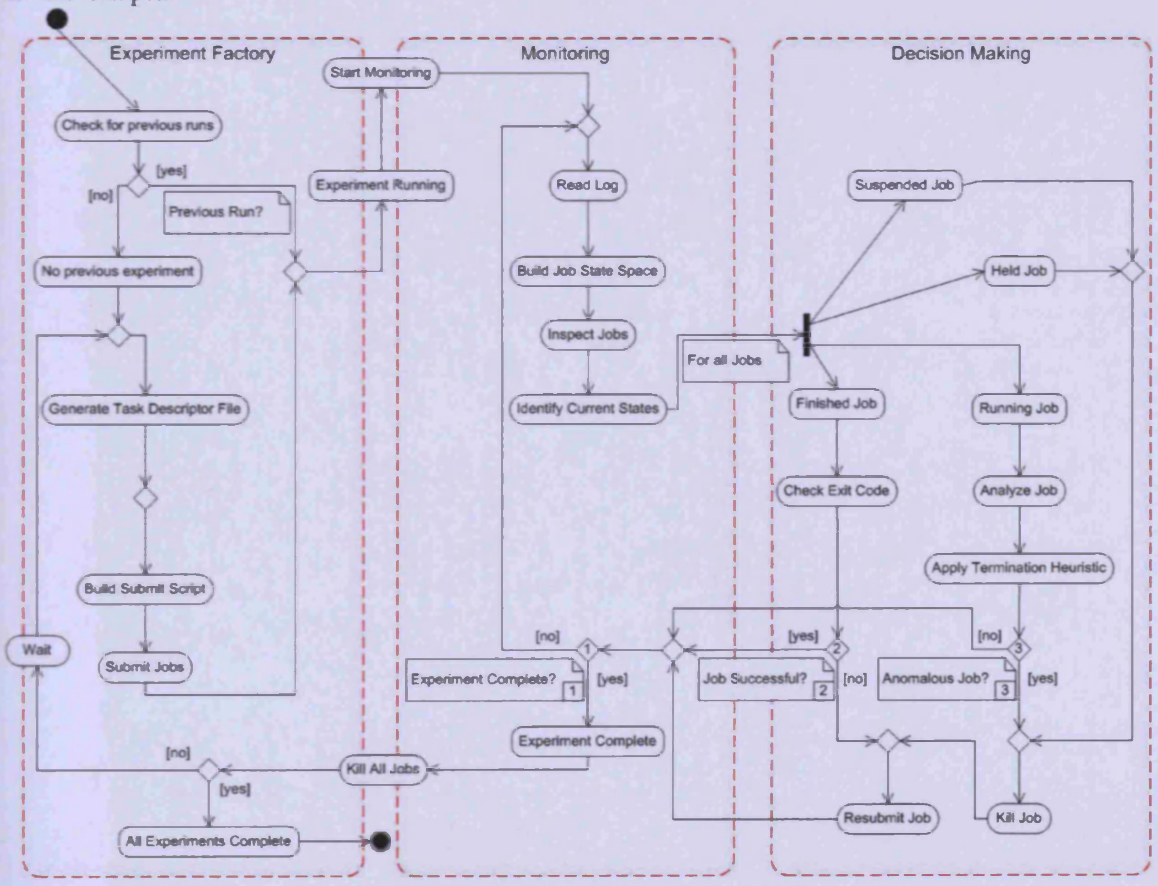
$Runtime > Max(10\,minutes, avg(Matlab\,Initialisation\,Time) + 5StandardDeviations) + predefined\,jobruntime$

$THEN$

$terminate\ and\ resubmit$

possibly as a consequence of the Condor logs not being up to date, a fault in the parsing of the log or those which were resubmitted by Condor in response to an eviction, are terminated. The Autonomic Job Manager then waits for ten minutes to allow the Condor negotiator (whose cycle periodicity is typically five minutes on the Cardiff University Campus-Grid) to identify that the workstations previously in use are now available again. Then if more experiments are to be performed the Autonomic Job Manager will begin the next experiment and the cycle presented in this Section restarts. A summary of the functionality of the Autonomic Job Manager is presented in Figure 2.13.

Figure 2.13: The Autonomic Autonomic Job Manager for the Campus-Grid experiments presented in this Chapter



In some cases the log readers cannot detect all jobs from the logs during runtime, typically this is due to the condor shadow daemon not being able to write to the log. When this scenario arises there is little that the Autonomic Job Manager can do other than wait for the situation to resolve itself, which it will eventually. This is the one corner case when the Autonomic Job Manager cannot perform to its best with regard to the steering of an experiment.

2.4.3 Results

In this Section the results of the Campus-Grid's typical behaviour and performance are presented as the base case for all synthesized experiments. The results presented here are intended as a benchmark of the Campus-Grid's performance for short jobs and will be used to show the benefit and improve in performance through the use of the prototype presented in Chapter 5.

In Table 2.1 the results of ten runs of each experiment are presented, and it can be seen that the Autonomic Job Manager's primary responsibility is to ensure that all one hundred jobs are completed through the identification of failures and subsequent resubmission.

Table 2.1: Table of the performance of 100 synthesized jobs using the Autonomic Job Manager of the Campus-Grid.

	Predefined Job Time in Seconds			
	1	60	600	1800
Min Jobs Submitted	120	104	157	117
Avg Jobs Submitted	133	137.7	384.4	246.222
Max Jobs Submitted	157	186	690	880
Min Efficiency	63.694 %	53.763 %	14.638 %	11.364 %
Avg Efficiency	75.898 %	76.496 %	34.459 %	62.504 %
Max Efficiency	83.333 %	100 %	63.694 %	85.47 %
Volatility Failures	91.496 %	95.798 %	98.101 %	97.186 %
Autonomy Failures	8.504 %	4.202 %	1.899 %	2.814 %
No. of Jobs Failed	341 %	357 %	2843 %	1315 %
Min Job Runtime	32	94	633	1838
Avg Job Runtime	100.903	154.109	718.752	1926.931
Max Job Runtime	574	583	1164	2219
Standard Deviation	67.309	57.263	72.871	74.106
Min AM	642	946	2560	5896
Avg AM	2220.909	1865.5	3961	8187.889
Max AM	12830	7345	7741	14868
AM 1 workstation	100	6000	60000	180000
Min ERU	0.676 %	27.95 %	70.388 %	57.06 %
Avg ERU	0.898 %	35.581 %	77.058 %	87.757 %
Max ERU	1.172 %	39.976 %	80.107 %	93.98 %

Job Efficiency

The performance for the Campus-Grid for the job efficiency is on the whole what would be expected, only a few of the experimental runs were stricken by one or more job blackholes. However, what is clear is that most failures were due to resource volatility and not workstation autonomy, which illustrates that some improvement in the resource model is necessary. Something that should be mentioned is that for one experiment an efficiency of 100% was achieved, which should be commended.[20] Table 2.1 also shows that as the job length increases the efficiency generally decreases, which also is to be expected. The main issue here is that when the Campus-Grid is left to its own devices it cannot provide many (if any) guarantees on performance, therefore making it somewhat unreliable in terms of performance. Such an outcome is to be expected in opportunistic and volunteer computing with respect to workstation availability. However, Table 2.1 identifies that most job failures are a consequence of resource volatility and not autonomy. The latter of which would normally be expected to be detrimental to performance, which for the Campus-Grid is indicative of resource unreliability (volatility).

Job Runtime

The period of time required for the Campus-Grid to complete a job is quite poor for the shorter jobs. This is primarily due to the necessity to initialize Matlab for each job, and consequently increase execution time. The overhead experienced by each job ranges from 31 to 573 seconds, which shows that the time required for initialization is not consistent, but can be expensive, especially if the job runtime is low. In addition, the standard deviation for the 1 and 60 second jobs is greater than or near to their predefined run times. For these two job lengths their predefined runtime is a percentage of their overhead rather than the overhead being a percentage of their runtime, which is not a good sign. This also further motivates for improvements in the conceptual model if short jobs are to be executed in a Campus-Grid context. It is also interesting to note that the maximum runtime of these two jobs is similar. For the longer running jobs performance is improved and the impact of the overhead is reduced, which is to be expected. This also validates the fairness of the experiments in use.

Application Makespan

The first and most obvious point to be made about the observations in AM is that the one second jobs actually required longer in all experiments on the Campus-Grid than they would normally require under sequential execution on a single workstation. The same is also true in a small number of the sixty second job experiments. For all remaining job times the AM is an improvement upon the sequential execution on one workstation illustrating the utility of the Campus-Grid for these job times. The second point to be raised is the variability in the AM, but this is normal behaviour for

[20]The Autonomic Job Manager did not report 100 submitted jobs here, due to a peculiarity in the Condor log, which caused it to submit an additional 4 jobs. These jobs are not considered in the job efficiency, but were nevertheless submitted.

the Campus-Grid. However, what it does show is that there can be no guarantees for when a job set completes regardless of a job's runtime; compare the Max AM values for the 1 and 1800 second jobs, and also for the 60 and 600 second jobs.

Efficiency of Resource Utilization

It is clear from the ERU measurements that the Campus-Grid is not suitable for short jobs of less than or equal to 60 seconds. For the longer jobs the performance is satisfactory, which is to be expected. What is surprising, however, is just how bad the performance is for the shorter two job times. The poor results stem primarily from the need to initialize Matlab for every job, which for the shorter jobs is particularily expensive. Job clustering would be beneficial here, but the results of the first study highlighted that this may not be a viable option. Instead it would be more beneficial to be able to test the workstation prior to submitting jobs to it. Therefore, alleaviating the risk of submitting an entire cluster to a faulty workstation. However, job clustering in this manner does not remove the need to initialize the workstation for every cluster, and therefore a trade off between cluster size verses the risk of preemption will emerge.

Summary

These results have shown shown that the Campus-Grid is not a good platform for short running jobs, and that an improved model is required. The results also show that the performance in most metrics improves as job length increases, which verifies that they fairly and accurately reflect the performance of the Campus-Grid. In addition, these results also show the basic abilities of the Autonomic Job Manager, to react to the Campus-Grid and the intricacies of job management in order to execute Matlab-based jobs, are both useful and valid, as in all experiments 100% were successfully completed without user intervention. This is despite the number of failures due to inaccuracies in workstation classAds and Matlab installations.

In Figures 2.14 – 2.17 a selected graphical representation for each of the experiments is presented. Note the variability with which the jobs start (time 0 is when the job submission took place), which further highlights the unpredictability in resource acquisition. In Figures 2.14 and 2.15 the time required by the Campus-Grid to acquire resources is significantly longer than the experiments depicted in Figures 2.16 and 2.17.

2.5 Discussion

This analysis has demonstrated the complexities of using real Campus-Grid resources. Scenarios like this cannot be simulated and are always presenting new, sometimes perplexing, situations for users to handle. Having presented these three empirical studies into the complexities and challenges job of management and the many scenarios which make this difficult, one question presents itself: Given the difficulties that users must face, is the the Campus-Grid worth the effort for short jobs?

Figure 2.14: Graphical representation of the runtime for 100 synthesized jobs of one second in length. In this experiment the average number of workstations used was: 7.141 and the average job execution time was: 83.19 seconds.

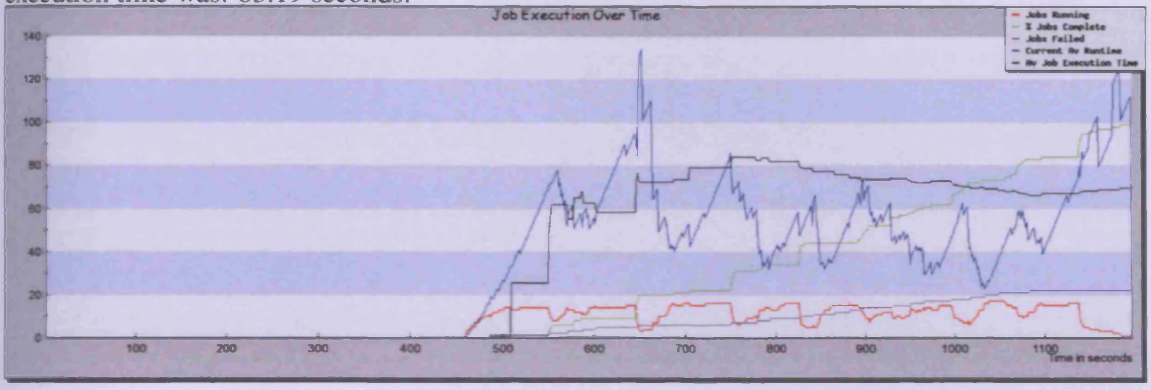


Figure 2.15: Graphical representation of the runtime for 100 synthesized jobs of sixty seconds in length. In this experiment the average number of workstations used was: 16.408 and the average job execution time was: 152.53 seconds.

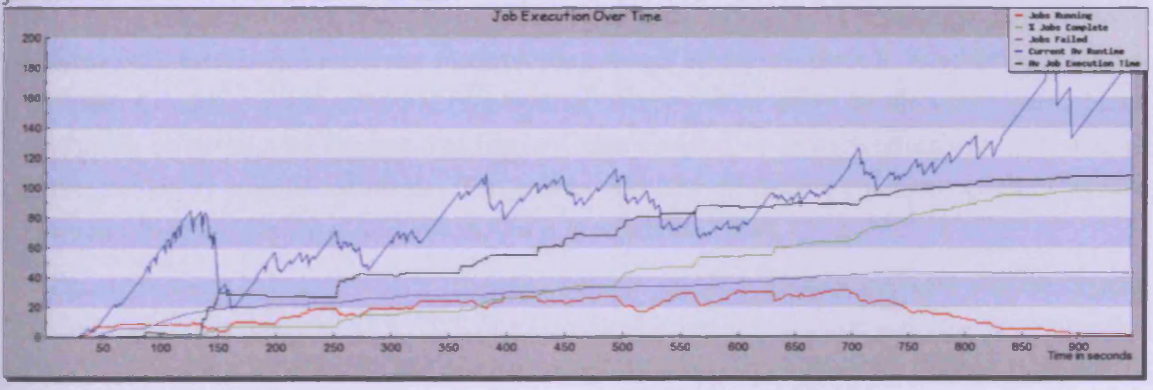


Figure 2.16: Graphical representation of the runtime for 100 synthesized jobs of six hundred seconds in length. In this experiment the average number of workstations used was: 28.49 and the average job execution time was: 703.19 seconds.

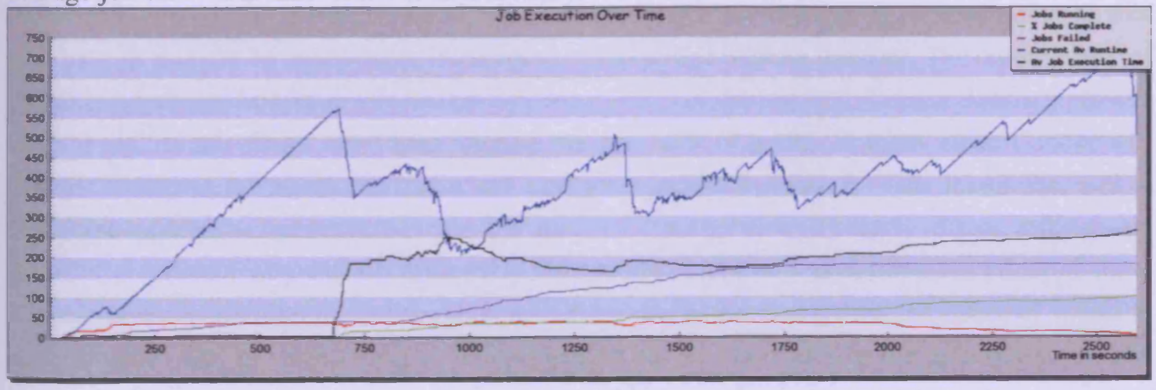
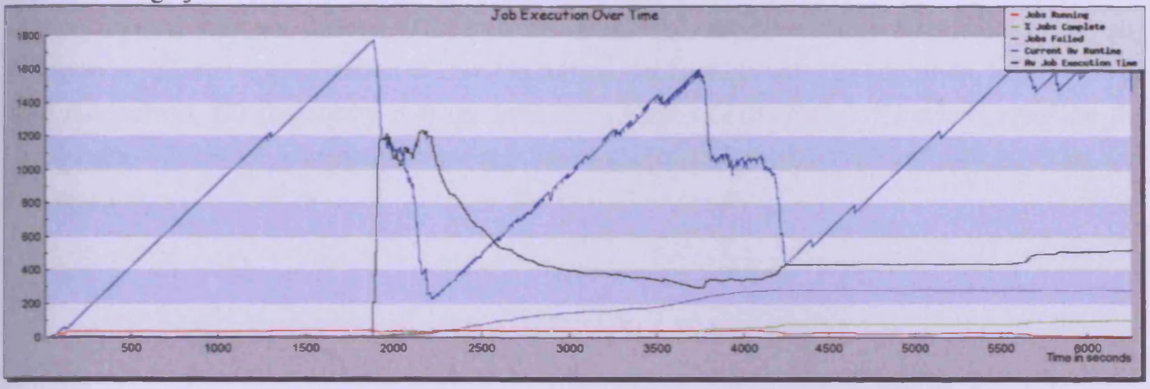


Figure 2.17: Graphical representation of the runtime for 100 synthesized jobs of one thousand eight hundred seconds in length. In this experiment the average number of workstations used was: 31.76 and the average job execution time was: 1912.86 seconds.



The Condor pool provides a wealth of computational power, which would otherwise may not be available to the users at all. To sum up the contribution to research projects that the Condor pool provides can be put in monetary terms. The existence of the pool allows the university to save a third of a million pounds a year compared to if dedicated resources of equivalent computational resources were used [10]. Few, if any research projects can afford to replace it. In addition, to deploy a job onto Condor requires minimal (application) development effort on the user's part. Many applications do not require alteration to run on Condor nodes, as they are effectively workstations of other university staff and students. This means that they all share a common operating system and similar base installations. So in short, yes it is worth the effort.

The challenge is to improve the management of jobs and how the user interacts with the system. The first study illustrated that up to 78% of job runtime is unnecessary and just how much time this relates to, for even a simple job. The findings were that jobs can runaway for reasons independent of their complexity. The time that is lost when this occurs could have been used for other jobs and is lost unnecessarily. No concrete evidence could be found to explain why jobs runaway. The second study took this further and presented what happens in a more realistic context, without increasing job complexity, but by increasing the basic job requirements. The findings were that, not only do users have to contend with runaway jobs, but also noticeable changes in workstation state, and installations that can be overwritten, interrupted and modified without warning. The consequence of which is a series of knock on effects that detriment job completion rates. Condor does not provide any support on this front, other than logging the exit code of a job. In many cases Condor will allocate the same job again and again and thus job blackholes emerge. This leaves the user to determine what error has occurred and why. In some cases jobs ran for days, with no information available to explain why this occured. The third study empricially quantified the effect of these difficulties on the performance of the Campus-Grid to the extent that even if the issues of volatility were removed the resource model is ill-equipped to enable short jobs to run effectively, as only successful jobs were used to measure performance. The overall conclusion is that job management in a large Campus-Grid is challenging and beyond the ability of non-computer scientists.

During these three studies autonomous job managers were used to steer experiments. Their extension can provide many additional features and provide new opportunities to the user. There are two abilities that are needed to address the challenges raised in these three studies: (1) the ability to verify that a workstation is suitable for job deployment prior to execution, and once this been ascertained, (2) the ability manage jobs and resources separately. Therefore, enabling the introduction of new job management models which can resolve the trade offs of job clustering and management.

Chapter 3

Related Work

This Chapter discusses the literature related to the current state of the art, and is divided into two main Sections. First, distributed image processing approaches will be presented, with an orientation toward approaches which include Campus-Grids, Matlab, or forms of distributed computing. Second, a review of more general work in the scope of resource management will be presented. The aim of the second section will be to relate the current research to the challenges outlined by the three empirical studies performed in Chapter 2.

3.1 The Facilitation of Distributed Image Processing

In this Section an overview of work, which has an emphasis on image processing in a distributed or parallel context is presented and discussed. Due to the orientation toward a Matlab-based approach, special attention is given to approaches that also provide a Matlab solution for distributed computing.

3.1.1 Matlab-based Approaches

The choice of Matlab as an application container and platform is due to several different achievements of the Matlab system. Matlab is one of the forerunners in the domain of scientific computing and analysis, thanks to the ease of use and the ability to rapidly develop analytic tools and solutions. Since the original inception in the 1970s [44] of an interactive interface to EISPACK [74] and LINPACK [75], Matlab has been extended to include a vast repository of toolboxes for many different disciplines. The importance of these features is expressed by the user base (well exceeding 500,000 [76]) Matlab has built up over the years, and the number of books published (over 400) about it.

Another reason for the choice of Matlab is its competitiveness in the domain of scientific computing. There are other license free and open source equivalents, with SciLab [77] and Octave [60] being the most well known. Wilkinson [78] reported that for those who wish to avoid the expense of Matlab licenses and the challenge of compilation, the conversion from Matlab to Octave is quite straightforward, as is its use on systems such as Condor. However, no evaluation or empirical study was performed to document the cost in performance of this migration. Steinhaus [79], however, performed several comparative studies of the mainstream scientific computing products, which include: Matlab, Mathematica, Octave, Scilab and some lesser known others. His analysis covers a wide spectrum of features that includes: the range of functionality, performance and the return on investment each product gives. His results identify that Matlab is far superior to its open source and (license free) equivalents in all areas of analysis. He also observed that the two industry forerunners are Matlab and Mathematica and that there is little that can be used to separate the two.

Matlab's popularity and capability has enabled it to entrench itself into a wide range of domains and therefore soon became a staple of scientific computing. The question of a parallel version of Matlab was soon raised, as applications stretched beyond a single workstation. In 1995, the Mathworks released a statement [55] expressing that they had no intention of producing such a toolbox, as it did not make good business sense. One of the reasons for this is that Matlab is an interpreted language, and its performance does not scale with problem size [30, 80]. Instead, Matlab spends much more time in places like the parser, the interpreter, and graphics routines, which are harder to parallelize. The first (commonly accepted) parallel approach for Matlab, MultiMatlab [81], was published by the scientific community in 1997 and since then many other approaches (approximately 40) have followed. Their emergence demonstrates that this was a missed opportunity for The Mathworks.

In this Section an overview of the approaches for distributed and parallel Matlab, which are relevant to this research is presented. Efforts in this area generally attempt to either provide MPI-like message passing to Matlab, built in methods that will partition work among multiple sessions, or automatically convert Matlab scripts into parallel programs [82]. Strangely, very few approaches attempt to extend or utilize the Matlab compiler in order to create a license free option, preferring instead to perform simple compilations. Therefore the approach is either to compile often, simply accept that more licenses are required, or perform very elaborate customized compilation procedures.

Matlab Distributed Computing Toolbox

The Mathworks eventually released a toolbox for parallel computing: The Matlab Distributed Computing Toolbox, which enables Matlab users to distribute and control a set of computational tasks across several nodes. In this context a node corresponds to a stripped down version of the Matlab interpreter and is denoted as a distributed computing engine. In more recent versions, the engines are simply MPI processes, and the toolbox provides a limited MPI implementation, which consists of commands to perform nonblocking sends, blocking receives, broadcasts, and global reductions. The toolbox also contains additional job-control functions, with plug-ins to many different queue and resource management systems, including Condor. However, each node must hold a valid license and as previously mentioned, the management of licenses across a Campus-Grid environment is not always possible [64,65]. Newer versions of the toolbox have removed the need for an MPI-like programming style, but it is not possible to simply map code to the toolbox, as explicit modifications to the code are required to signal where parallelization should be performed.

An advantage of the distributed computing toolbox/engine combination is that it lets users utilize existing Matlab and Simulink toolboxes, provided the data is available locally. In its current form (without support for existing parallel numerical linear algebra libraries), this tool is most useful for embarrassingly parallel problems, despite the design being orientated toward MPI-like parallelism. A key disadvantage of this toolbox is that the nodes hosting the distributed compute engines require access to a licensing server. Thus, even though the toolbox supports back end systems like Condor, a workstation in many Campus-Grid scenarios cannot, currently, be used. This is due to the outstanding issues of license locality and the scope of a license server. For example, even if licenses were available either locally at the head node or remotely at the compute node, but both the head node and compute node fall under the scope of different licensing servers, it is not possible to run the job. In such scenarios, the job could only be run when the compute nodes are in the same domain as the head node. This therefore dramatically reduces the usability of the toolbox. The performance of the toolbox has also been questioned in [1]. Another key limitation with the toolbox is that it does not allow for the dynamic distribution of the user's code [30].

SmartLM

SmartLM[1] is a framework 7 European project driven by the challenges in license management for applications such as Matlab. The main aim of the project is to enable the establishment of Service Level Agreements (SLA), which provide the basis for license migration between sites. Matlab and many other commercial software packages are managed using the FlexLM license manager software, where license servers hold a specific number of license tickets. When a user requests a license they check out the required number of tickets for their application context,[2] when the available tickets reaches zero, a user can no longer launch the application. The core idea of SmartLM is to provide the means to transfer licenses between sites, which would mean that when a user wishes to run an application remotely they can transfer their own licenses with the request. Here, the SLA can provide a legal foundation to govern how this transfer is undertaken. There are two main goals here, firstly to protect the license owner from illicit or malicious activities, and secondly to define how the transfer is to be undertaken, for example the physical actions needed to transfer the license ticket securely.

The SmartLM project is still in its infancy at the time of writing, and hence cannot be harnessed for this research, however when the software is ready there is no doubt that it will be very useful and popular; especially in Campus-Grid contexts. For instance it would alleviate the need for central license servers, which are always surrounded by inter-department politics concerning usage and funding. In addition, it would provide the means for university departments to sell spare license

[1]SmartLM project website: `http://www.smartlm.eu` (last accessed April 2009)

[2]Matlab for example, has two modes of operation in this respect depending on the license type: student license, which permits the user to use any installed toolbox, and regular license, where two types of ticket are in place; one for the session as a whole, and one for each toolbox in use. In the latter case a user will check out multiple licenses.

capacity, here the SLA would also contain the necessary economic data required for trading, and its protocol stack the necessary methodology for negotiating its constitutional aspects such as price, usage allowances and quality of service. Further details on the SmartLM prototype and its use and definition of SLAs can be found in Ref [66].

A Survey of Parallel Matlab approaches

Choy and Edelman [44] present a thorough survey of parallel Matlab approaches. The main contribution of this work is the definition and discussion of what constitutes a *good* parallel Matlab. They also present an example of what they consider *could* be a good parallel Matlab approach: Matlab*P, later becoming the commercial product Star-P (both of which are discussed below). They define that a good parallel Matlab should address at least one of these areas well:

1. A lot of small problems that require no communication.
2. A lot of small problems that require some communication.
3. Large problems.

One of the most interesting observations of this survey is the number of approaches that have been documented (27 reported), as well as how many are now defunct and exist only in Google's cache. Those which are not defunct and relevant to this research will be further discussed in this Section. Choy and Edelman identified four main areas of parallel Matlab, which are: **(1)** Embarrassingly Parallel, **(2)** MPI, **(3)** Compiled, and **(4)** Back-end support. The first two categories are self explanatory, but the latter two are perhaps not.

The compiled category refers to two types of approaches, firstly, those which use the Matlab Compiler to displace the requirement for multiple Matlab licenses, and secondly, those which use or have created additional tools to convert the Matlab code into a form of parallel code. The idea of back-end support is to connect a version of Matlab to a parallel computation engine, which utilizes optimized parallel numerical computation libraries such as ScaLAPACK [83]. This approach is a means to remove the need for multiple Matlab licenses and prevent the need for compilation. Typically, an HPC machine is used, and those discussed in the survey all utilize this approach. However, more recent efforts in this category have utilized both cluster and Grid Computing approaches.

The survey discusses only fine-grained approaches, and illustrates that fine-grained parallelism is a difficult task to perform for Matlab-based applications. This again is because Matlab is an interpreted language, but also because the difference between the time required for execution and communication can be negligible. The result is that it is very difficult to retain an effective use of resources, especially if large numbers of resources are in use or available. Although compilation can improve the interpretation phases of Matlab, this improvement is only minor.

Choy and Edelman present what *could* be a good approach for parallel Matlab, however, their performance analysis illustrates that in some cases speed up was only achievable when the number of machines used was a small factor of the number of jobs to be performed. This is perhaps a consequence of granularity from the fine-grained model adopted, which Moler [55] also raised concerns over when defending the Mathworks' decision not to produce a parallel Matlab. This would suggest that Matlab may be better suited to a coarse-grained parallel model, using the granularity definition given in Section 4.9.1.

Matlab*P

Matlab*P [56, 84][3] is a transparent environment built on top of Matlab. The motivation behind this work is that many widely used algorithms can be described as matrix operations, which on small data sets have satisfactory performance, but do not always scale for larger data sets which can dramatically increase computation time. Therefore, the identification of the need to perform analysis on large data sets interactively (to test new ideas) was made. Their philosophy is that the client should not have to be concerned with the administrative aspects of data handling and that the client's machine does not need to be used for computations.

The architecture of Matlab*P is a client/server model, where an instance of Matlab transparently communicates with a Parallel Problems Server (PPServer) [80]. The PPServer is essentially an MPI-based collection of linear algebra algorithms for large dense and sparse matrices. It runs on any high performance machine (e.g. one of SGI origin) with a *nix operating system that supports an MPI implementation. Communication is performed by standard Unix sockets and parallelism is achieved through the polymorphic overloading of Matlab functions and operators. A Matlab `layout` object `*p`[4] is used to specify that a variable is to be used in a distributed context (see Listing 3.1 for use).[5] It is reported that when Matlab*P was integrated with the Matrix Computation Toolbox (developed for [85]), 70% of the toolbox could be integrated without any modification to the underlying code, which is commendable.

Listing 3.1: Matlab*P of adding two random matrices.

```
A = rand(10*p, 10); % Create a 10 by 10 random matrix
B = rand(10*p, 10); % The *p specifying that A & B are distributed variables
C = A + B; % Perform A plus B (in parallel) C will also be a distributed variable
```

The main advantages of Matlab*P are: **(1)** That all data is held and created on the server. This means that data transfers only occur between processors and upon client request. **(2)** The leverage of established high performance parallel numerical libraries e.g. ScaLAPACK, SuperLU, PARPACK etc. and the transparent hooks from Matlab to the back end PPServer allow the user to benefit from a reduced computation time. This is also the case if only processor of the HPC machine is used. **(3)** The user doesn't require any knowledge of parallel computing, provided that they stay within

[3]The reference [84] is the commonly used reference for Matlab*P, however, no such paper exists. The paper that should be referenced is: [56]. To be consistent with other published works, both references are supplied and the name Matlab*P is used throughout this thesis instead of the original name: MITMatlab. MITMatlab contains the same functionality and basic theory as Matlab*P.

[4]The Matlab documentation is unclear about how such objects work or if they are even supported in the newer versions of Matlab. In fact, no documentation of such objects exists in the documentation of more recent versions of Matlab. It is possible, however, that this concept has been renamed since it was first described in 1999.

[5]This also enables the propagation of parallelism; if a distributed variable is used in a calculation, the result will also be treated a distributed variable.

the scope of Matlab*P. (4) Data is created on the server and remains there (unless requested by the client), allowing for no data transfer between client and server. Data communication and transfer only occurs between processors.

The main disadvantages are: (1) The numerical libraries are not portable. (2) There are no administrative tools, making installation difficult. (3) If the libraries are not already installed on the HPC architecture the user must do this. Note that there may well be bugs in the installation libraries, the administrators of these machines may not be willing to add additional libraries or if the user is the administrator they may be unfamiliar with the nuances of the administration of the libraries. (4) The capabilities and applications that the approach can serve is limited by the available libraries. Hence, not all Matlab functionality can simply be mapped to this approach. If no library exists, the user must implement a parallel approach (with an MPI implementation) to suit their algorithm. (5) Only two-dimensional and single precision matrices are supported. Here, there are ambiguities in the literature, as Husbands [56] states that double precision and complex numbers are not supported. Choy [86], however, says that they are. Points (4) and (5) greatly limit the range of applications which can be adopted by this approach. However, for the applications which can be simply mapped to Matlab*P the improvement performance is good.

The use case of Matlab*P does not fit into a Campus-Grid context due to the reliance on dedicated and (homogeneous) high performance hardware. In addition, its basis of MPI may struggle in a volatile and opportunistic resource context, where resource availability and consistency cannot be guaranteed. Finally, moving away from the fine-grained parallel model would require significant architectural changes.

Star-P

Star-P [1] is a commercial implementation of Matlab*P by Interactive Supercomputing. Much of Star-P is taken from Matlab*P, but the exact architecture of Star-P is difficult to present, as a commercial solution there are no papers describing Star-P in full for obvious reasons. However, there are papers which describe small facets of the system and some describing user experiences.

The basic structure of Star-P is quite simple: an instance of Matlab transparently communicates with a server front end of a high-performance machine in the same way as Matlab*P. Many papers which refer to Star-P, refer to it as a package which sits on top of a Matlab instance, where it transparently handles all parallelism for the user. However, Star-P still assumes that the user has a HPC machine to host its back end PPServers. Wong et al. [87] used Star-P to perform a computational chemistry application using Molpro [88]. However, only one part of their experiment was parallelized using Star-P. The experiment consisted of two parts, the first part of the experiment was not executed using Star-P, but on a custom built machine. The second part, which was executed on Star-P, required Wong et al. to borrow computational resources (an HPC machine from interactive supercomputing), despite having local HPC resource. The questions that immediately arise here are: firstly, why was separate hardware needed to run Star-P when existing resources were in place, and secondly, why could the first part of the experiment not be run on Star-P? [89] is another example which used Star-P; here the application was parallelized with help from interactive super computing staff to make the application Star-P compliant.

Star-P is documented as achieving sub-linear speed-up (see the Star-P demonstration video[6]), which is due to the optimized parallel libraries run on the back-end. This enables Star-P to achieve the performance of many commonly used mathematical operations in Matlab to match the performance of programs written in low-level languages transparently [90]. However, the analysis does not accurately demonstrate the capabilities or scalability of Star-P. In addition, Ref [80] documents experiments performed using Matlab*P where adding further nodes increased execution time. A Star-P user does not specify the number of nodes/cores to use and therefore it is unclear from the literature how Star-P determines the number of nodes to use when a large numbers of processors are available.

The concluding remarks for Star-P are the same as Matlab*P: it will not fit the Campus-Grid context for the same reasons. It is, however, still a useful and interesting approach for the parallelization of Matlab and also demonstrates that parallel solutions for Matlab are in high demand.

On-demand and Interactive Matlab Using Condor

In [76] Reuther et al. describe an approach for parallel applications which require Matlab instances. Their approach can be used on a variety of resource managers, but they are currently bound to Condor. The resources in this work are a rack of dedicated servers consisting of approximately 160 cores. This infrastructure is called the LLGrid, but it is in fact just a cluster at MIT. In order to facilitate parallel Matlab, two toolboxes both from MIT, pMatlab [91] and MatlabMPI [92] are combined to adopt polymorphic implicit parallelism and provide an MPI-like communications protocol respectively. There are also possible cross overs between pMatlab and Matlab*P. This amalgamation is called the gridMatlab toolbox. It interfaces with the underlying resource manager (Condor) to provide three core abilities: cluster status monitoring (how many processors are available), job launching, and job termination. Or in other words, it provides the ability to call `condor_status`, `condor_submit` and `condor_rm` for each of the respective tasks from a Matlab session. Each job then initializes a (licensed) version of Matlab with the two toolkits running and providing a front end for interactive use.

The gridMatlab toolkit determines (using `condor_status`) whether enough resources are available to satisfy a user's request, and if a sufficient number of resources are available the jobs are submitted. However, it is up to the user to identify their resource requirements. If an insufficient number is available, the toolbox queries the user whether their application can be run on fewer processors or whether the job should be resubmitted when a sufficient number is available. The system mostly provides transparent access to the resources, however, despite a reported implicit parallelization model users must still manually parallelize their code. No information is given about what this entails. The typical applications that are performed in this system are long running applications, where individual jobs are on average 31 minutes in length. However, some support is given for

[6]Available at http://www.interactivesupercomputing.com – last accessed June 2008.

shorter jobs, but this is not discussed in any detail.

The main achievement of this work is that Matlab sessions could be launched from Matlab using Condor, which were then interactively used for MPI applications, where the user's workstation is node 0. However, firstly, this was performed on dedicated resources, and therefore failure rates are minimal, and not even mentioned in the work. This limits the ability to map this work onto a Campus-Grid, even though this approach is based upon Condor. This is potentially the most limiting factor of this approach, because MPI applications cannot normally tolerate loosing one or more nodes. The authors' suggestion for ensuring dedicated access to a set of resources (so that job eviction does not occur), is to place Condor into its execute always job model. Here, jobs cannot be preempted even if another user or even the owner wishes to use the resource. In the context of a Campus-Grid, such an approach is usually frowned upon. Secondly, application run times are long, potentially enabling low overheads relative to execution times. Again, no performance data in this respect is provided, and therefore it cannot be ascertained whether this approach would be feasible in the context of short running jobs. Thirdly, licensed Matlab sessions are used, and consequently the approach will be limited to the number of available Matlab licenses. No discussion is given concerning what happens if resources are available, but licenses are not, and how this affects the resource inspection process, which is performed to test whether the facilitation of on-demand access to Matlab instances is possible for a given request.

Matlab*G

Matlab*G [93] is a fine grained explicitly parallel Matlab approach for the ALiCE Grid [94]. Here, licensed Matlab sessions are transparently launched as ALiCE jobs to provide image processing capabilities. The number launched is in response to explicit user requests, i.e. the user must know in advance how many Matlab sessions they require. Each session supports multiple user tasks and can be used interactively, just like regular Matlab. Therefore, only one initialisation phase is required per session. However, the approach is limited by the available number of Matlab licenses and enforces static allocation, which relates to the original user request. This suggests that should the user request an inappropriate number of Matlab sessions, they must either rerun their application or wait for it to complete. The user must also know in advance what resources are required for their specific application. No information is given about what would happen should this number of sessions not be available. However, the parallelization of Matlab code is very intuitive. Parallel calls are explicitly included by the user, however, the user requires no knowledge of how the call is serviced.

The idea behind Matlab*G is good, but it is not thoroughly thought through, and, ultimately, if there are no Matlab licenses available, the system cannot serve the user. It is also an expensive approach to support when large numbers of nodes are required. In a Campus-Grid context this approach cannot work, because Matlab licenses are not in abundance, and are potentially tightly bound to the workstation owners.

JavaPorts

JavaPorts [82] is an approach for distributing Matlab applications to a network of workstations. The Matlab compiler is used at distribution time to compile a user's application, which is then distributed transparently onto the network. This is a great idea in theory, as license requirements are removed, and an opportunistic context is intrinsically included in the model. However, in practice this is not a practical solution. It will work very well for simple applications, as users can simply submit their Matlab source code to the central daemon, and wait for the results. The problem with this approach is that as applications become more complicated several issues and complications will arise during the compilation process.

Firstly, as the dependencies of the application increase the user or some manager application must identify what needs to be transferred to the central daemon, so that all dependencies can be resolved and the compilation begun. When compilation is performed in a local context, Matlab performs this process transparently. Therefore, the JavaPorts approach must either assume that all functionality is available on the central daemon, or that the user or some autonomic manager can identify and transfer all dependencies. This will be exacerbated significantly if third party tools are in use, or if the application is large.

Secondly, many third party toolboxes are built using external operating system-specific libraries, which are subsequently integrated into Matlab using wrapper scripts. This is a transparent process in the regular Matlab context to the user and each toolbox invariably contains the necessary OS-specific scripts for all platforms. To integrate such toolboxes into compiled Matlab code, external tools are often required. This will be discussed in more detail in Section 5.2.1, and will include a method of transparently handling such scenarios.

Thirdly, for large or complex applications, compilation is non-trivial, as a myriad of disparate conditions can cause the compiler to fail. The most common examples, from direct experience in complex Matlab compilation, include (but are not limited to): **(1)** unbalanced parenthesis, **(2)** an m-file being a script and not a function, **(3)** an external library that cannot be included without additional administration, **(4)** the code contains uncompilable Matlab functionality,[7] **(5)** two functions that exist with the same name but are not polymorphic, and **(6)** the scope of a variable modified in a nested function or through an implicit call of either the `load` and `eval` functions causing compiler failures when parsing a function to be complied. In these cases compilation is invariably an interactive process which requires both autonomy and supervision. This again will be discussed in more detail in Section 5.2.1.

Finally, the consequence of compiling and then distributing each submission successfully is an increase in an application's makespan and each compiled instance cannot be extended beyond the scope of its original conception and intention, i.e. its reusability is limited. A much better solution than the Javaports approach would be to perform the compilation on the user's workstation and generate a general application container which contains one or more entire toolboxes to increase

[7]see: `http://www.mathworks.com/products/compiler/compiler_support.html` for a detailed list of what this includes

reusability. The output of this process can then be transferred to the central daemon and propagated from there to the worker nodes. Such a process could be performed in a similar manner to how the code was distributed for the third study in Section 2.4 was performed.

Grid-enabled Hyperspectral Image Analysis Toolbox

Rivera et al. [95] present a Hyperspectral Image Analysis Matlab toolbox, which is accessed through a portal interface. This approach claims to be a Grid approach, but this is debatable. In either case Rivera et al. report that they use a Grid portal,[8] but no information is given concerning resource discovery or acquisition, how the Matlab Toolbox is controlled and no experimental or performance data is provided.

The key advantage of this approach is that from a users' perspective it is both operating system independent and programming language independent. There are, however, two conceptual limitations of this approach. The first limitation is the use and reliance upon Matlab licenses. The approach aims to provide public access to the Matlab functionality, which is a legal grey area in relation to the terms of licensing. It is also questionable whether the license holder would allow such an approach. Here, compiling the toolbox would result in a much more feasible solution. The second limitation is that the portal interface uses Java servlets hosted within a Tomcat servlet container. The performance of this front end can be questionable, depending on the implementation chosen and upon the performance of the server upon which it resides. If implemented correctly, this will work nicely as a form of resource broker (see [97, 98] for examples). However, if not, the overheads of the approach will be excessive. In addition, the portal acts as a single point of failure for the system and a performance bottleneck.

A Parallel Matlab for Grid-based Telescience

Lather et al. [65] describe an infrastructure to create parallel Grid-based telescience applications, which originate from a Matlab development environment. Using the Pegasus planner [99–101] with Condor, they were able to launch the Matlab M-files on a number of heterogeneous systems and achieve an increase of approximately six times in throughput, from 12+ days to just under 2 days. All their programs and function libraries were developed and implemented solely as Matlab M-files. Yet, they question the applicability of compiling their Matlab code, which would ultimately have yielded a more cost efficient approach. Their concern revolves around the additional effort required to compile their code, and that compilation comes at the cost of refining the algorithm itself. Here, (they say) that the application developer is no longer refining the algorithm but rather spending time on programmatic language conversions. These concerns are inaccurate, granted Matlab compilation is not a trivial undertaking for an approach such as their's, but nevertheless, it would not have required significant effort or time. Perhaps the key reason for Lather et al. to not investigate this

[8] A Web-based user interface that provides seamless access to Grid heterogeneous computing resources. The goal of using a Web portal is to allow scientists to focus completely on the science by making the Grid a transparent extension of the user's desktop computing environment [96].

possibility is the fortunate procedure for license administration at their institution. Their Matlab licenses are issued as part of a campus wide license management system, which means that all campus workstations can check out licenses. However, had this not been the case, the logistics of rights to their licenses between Virtual Organizations (they give TeraGrid as an example) would have had to be examined (or at least negotiated). They also identified that this is an issue that remains a challenge facing the Grid community.

Network performance also provided a critical hurdle. They used 128 processors and the computation time alone was approximately 2 hours. This does not include any queue wait times, which they identify as a considerable limiting factor on the application makespan. They required nearly 1.5 hours to transfer their input data set to the computational end point and 3 hours to return the output data. This translates to over 50% of the application's makespan solely due to data transfers. The problem, however, is that this data transfer overhead would be conserved even if the level of parallelism were increased. In other words, even if the entire application could be computed instantaneously the data transfer overhead (approximately 4.5 hours) would remain unchanged. The time necessary to transfer this amount of data is unwieldy and partially or totally negates any gains in computational times attained by using additional resources. Like many other approaches in this Section this approach is limited by the number of available Matlab licenses, and is only successful in this respect due to the fortuitous use of a Campus-wide license server, which is not the case for many institutions. In [102] Dobson reports that the GreenGrid [11] also runs Matlab as batch jobs using Sun's Grid Engine [103] and a campus-wide license server.

MultiMatlab

MultiMatlab [81] is an MPI based approach for parallel Matlab. To permit parallel/remote execution of a function seamlessly, an overloaded `eval` function is used. Matlab-based RPC calls are used to load new Matlab instances, which can then be initialised as a MultiMatlab daemon. Workstations are sourced either from a predefined list or are defined by the user at the point of the request, i.e. static resource binding. The user is therefore responsible for initialising the required number of sessions and terminating them when no longer required. Parallel routines can be integrated into Matlab by using MPI-based MultiMatlab implementations. This is generally facilitated through the writing or rewriting of routines as parallel MEX routines. Matlab's MEX interface allows C and Fortran programs to be called directly from Matlab. For simple or common routines this is fine, as such functions are likely to be built into MultiMatlab or are accessible online. However, should users wish to define new custom functionality, they must write their own C or Fortran code with MPI calls. Despite this, good performance was observed, albeit dependant on the competency of the user, which cannot be assumed. This approach is also dependent on the availability of Matlab licenses, and assumes the user has access to and knowledge of resources upon which the MultiMatlab daemons can be launched.

Users-Grid

Users-Grid [30, 104] is a framework built on Globus [26] and Condor-G [27]. Jobs are created on the fly in response to a custom monitoring system that detects and reacts to heavy load on the user's workstation. The system then identifies which application currently running can be stopped; the state is then caught and one or more jobs submitted using Condor-G. In order to evaluate an application in this way and to disseminate the required information for job creation, a customized plug-in must first be integrated for the application. For Matlab the current command, workspace variables and toolboxes in use are identified and represented in XML, so that the migration can be performed. A Condor job is then submitted to perform the migration, where the job is an instance of the application and will be started using the captured runtime information as a form of checkpoint.

The benefit of Users-Grid is that the user does have to be concerned with job submission. However, the complexity of job generation and submission means that the approach is slower than regular Condor. This may also be due to the use of Condor-G. Users-Grid was tested on a dedicated local 4-node Condor pool, and therefore has no demonstration of scalability, the effect of resource contention or "normal" Condor operation. The use of a local Condor pool is important because it means that Matlab licenses could be easily transferred between nodes, and, as already mentioned in this Section, outside of these conditions this may not be possible and licenses may also not be available. In addition, this approach does not actually perform any form of parallel execution, but rather releaves a wokstation of a demanding application by migrating the application to another workstation accessed through Condor and Globus.

The Matlab Compiler for Heterogeneous Computing Systems

The objective of the MATlab Compiler for Heterogeneous computing systems [105] (MATCH) is to make it easier for the users to develop efficient code for distributed, heterogeneous, reconfigurable computing systems, specifically DSPs (digital signal processors) and FGPAs (field-programmable gate arrays). To achieve this, the compiler translates Matlab code into C code for each of the target systems. As Matlab is an interpreted language and performs just in time compilation, it is not strictly bound to specific data types. This complicates the compilation, and, consequently, the user must insert a series of directives to enable the compiler to identify the data types in use. During compilation the user indicates whether the code should be parallelized or not, as this affects the compilation process, specifically for loops. It also means that output from the compilation process is not as reusable and less general. Parallel support is provided by embedding MPI-like calls, which invoke communication routines in order to pass instructions to the hardware. The compiler offers two modes of compilation, an automated approach and a user driven approach, where the latter is intended to enable experienced programmers to fine tune and optimize the produced programs.

The cost of the optimization process is a further reduction in generality, whereby the exact usage of the code is defined as well as what the typical input will be. The demonstrated effect of this process is a significant improvement in performance, but even a slight modification to the application or its context require the compilation process to be performed again. There are no studies in this work which compare the total cost of recompilation against the performance of a more general program. Here, a sub-optimal program could reduce the overall job latency because it does not require the program to be recompiled for changes in program logic, the data or data types in use. Whilst this work is orientated toward DSPs and FGPAs, it could potentially be employed upon Campus-Grid resources, but there are no details of whether any benefit would be achieved in this approach over the built-in Matlab compiler. Similarly, this approach is intended to target systems such as inspection components in a machine vision system rather than for the facilitation of research applications.

Grid Enabled Optimisation and Design Search for Engineering – GEODISE

The GEODISE [106] project has a collection of toolboxes for engineering and physical sciences, which can be exposed to scripting environments such as Jython and Matlab. Of particular relevance is the compute toolbox, which enables users to submit jobs to back-end resources or run packages that are available as services. The philosophy behind the use of environments such as Matlab is that they are familiar to the end-user and therefore ease the transition from sequential and local computing to parallel and distributed computing [107]. There are different options for job submission, however, the most relevant is the ability to harness Condor reported in [108]. In this work the functionality and utility of Condor is exposed through the definition of Matlab functions. This approach also enables the use of Condor programmatically, and, interactively, directly from the Matlab environment. The introduced functions are not necessarily Condor-specific and could therefore be mapped to other middlewares. Condor is exposed to Matlab through the definition of a web service, which provides the abilities of language independence, streamlined access and the verification of access credentials. However, the web service also acts as a single point of failure and as was demonstrated in the first study conducted in Chapter 2 also becomes a serious performance bottleneck and instigator for erroneous job behaviour.

Condor and Compiled Matlab

There are many research papers published that report the ability to run compiled Matlab jobs on Condor and other systems, but there is little that can be said about these approaches, other than their basic functionality. This is because very little time is spent discussing the performance of Condor outside of the general observation that the experiment runs faster using Condor. In these approaches, a developer must compile the Matlab application, prepare the submission script and submit the jobs. These approaches are also inherently application-dependent and specific, i.e. when the application context changes the Matlab stand-alone application must be recompiled.

Many approaches also hard code job clusters, in order to increase the time needed by Condor to execute a job, which therefore improves efficiency; [109–111] are examples. However, when the job size is increased in this manner, the user must simply hope that their jobs are not interrupted by the workstation owner. If they are, then it is likely that the job will fail, as compiled Matlab jobs cannot be successfully suspended by Condor on Windows platforms unless they are in an idle state, which for approaches in this area is uncommon and rare. Therefore, if suspension occurs and the job fails upon unsuspension, the job must be restarted either from scratch or a manually programmed check point. These jobs are also subject to all of the issues raised in Chapter 2, most notably with respect to the efficiency of resource utilization.

In [112], Zhorin and Stef-Praun describe their experience running compiled Matlab applications in a Grid Computing context (they use Condor-G) and the set up of the MCR on the remote machines. They report that the process of initializing the workstations to be Matlab-ready is a one time effort, which may be true in their environment. However, Chapter 2 demonstrated otherwise. One interesting observation relates to the minimum (3 – 5 minutes) and maximum (up to 2 hours) periods of the time that is recommended for the length of a job in such a scenario in order to avoid "excessive" overheads from the middleware. The authors provide no performance data to demonstrate their approach or experiences in a real context or define the computational environment in which they run their applications, which is unfortunate.

Compiled Matlab with other Resource Managers

In [113] Teodoro et al. describe an extension to the Anthill system [114] for the management of workflows where the processing units are performed by compiled Matlab scripts. They present an approach which is not commonly seen in this area, where each Matlab script is individually compiled into a single stand-alone executable and is described using a custom XML schema. In this respect the compilation is relatively simple, and therefore the potential issues raised in the discussion of JavaPorts are not applicable, as there is only ever one function and its Matlab dependencies which must be evaluated by the compiler. The idea is to be able to provide Matlab functionality to external applications, which may not be Matlab-based. The performance of this approach is also good: near linear speed up in the presented experiments. However, they do not compare their analysis to a single machine executing the entire workflow, and therefore the overhead involved in invoking each compiled function, deserializing and serializing the input and output parameters is not presented. It is also not clear what the costs of the compilation and XML definition processes are. This approach seems to be overly complicated for what is intrinsically a simple idea. The work unit descriptions must contain which Matlab filter produces the inputs, and which filter the output must be directed to. Furthermore, the definition of the input/output parameters is quite elaborate, as the means to define the value or source/destination of parameters is also unclear. In addition, each description may contain one or more Matlab scripts to be evaluated and the workflow is defined using multiple description files. It is not clear why libraries of functionality are not created instead.

Summary of Matlab-based Approaches

There are several key desirable attributes that are needed based upon the analysis performed in Chapter 2 and outlined in Section 1.3, which are simply not considered by many of the approaches discussed in this Section. Instead, many approaches include an orientation toward a fine-grained parallel model, a dependency on Matlab licenses, a setup which is often coupled with the assumption that Matlab licenses will be available at the time of the request, the expectation that the user can and will administrate the resources in use or the assumption that resources are reliable and persistent, and the expectation that the user will modify their code (for example to a MPI-like implementation) in order to use an approach. The approaches discussed in this Section are summarized in Table 3.1 according to the following criteria:

C1: Ability to run an engine on a Campus-Grid workstation
C2: License requirements
C3: Can handle user defined Matlab code
C4: Transparency of code distribution
C5: Ability/orientation to run short jobs
C6: Granularity
C7: Resource utilization model
C8: Node acquisition model

It is evident that there is no single approach discussed in this section that meets the minimum criteria needed for a volatile Campus-Grid environment. However, JavaPorts meets many of the criteria, but lacks a sensible model for compilation, and the ability to run jobs interactively.

3.1.2 Approaches for Parallel and Distributed Image Processing

The Distributed Image Processing Shell - DIPS

The Distributed Image Processing Shell (DIPS) [115] and its successor CDIP, Concurrent and Distributed Image Processing [23, 47, 116] (discussed below), used the NetSolve system [28] to create a distributed image processing system. These two approaches are similar to the work presented in this thesis and provide several significant contributions that should be considered when conceiving distributed image processing systems.

DIPS comprises of three software elements: **(1)** ImageVision [117], which could only be run on SGI machines, and for this reason making it remotely available was of particular interest. **(2)** Image/J [118], which was chosen because it contained a plug-in for ImageVision. **(3)** NetSolve, which enabled access to remote resources from multiple programming languages and was considered to be the best approach at the time. NetSolve also permitted dynamic applications without additional compilation, provided load balancing and *best* choice machine allocation.

In a similar vain to the results presented in Chapter 2 NetSolve proved to be an inhibitor of progress. There were many different reasons for this but the key reason, which is also apparent

[9]No volatility-related awareness or tolerance.
[10]Extension through the inclusion of new parallel numerical libraries, either written by the user or provided by a third party.
[11]Only when recompiled.
[12]User defined quantity.
[13]1 per job.
[14]Tolerance for resource volatility

Table 3.1: A summary of the attributes of each approach, which are important for the application context of this research. Acronyms: Y - Yes, N - No, Y? - Unclear from publications, but is likely, N? - Unclear from publications, but is unlikely, B - Batch, I - Interactive, A - Automatic, S - Static, D - Dynamic, U - Performed by user.

Approach Name	Criteria							
	C1	C2	C3	C4	C5	C6	C7	C8
Matlab DCT	Y[9]	n+1	Y	Poor	Y	Fine	I	AS/D[12]
SmartLM	Y[9]	n	Y	n/a	n/a	n/a	n/a	n/a
Matlab*P	N	1	N[10]	Good	Y	Fine	I	AS/D
Star-P	N	1	N[10]	Good	Y	Fine	I	AS/D
Rether et al.	Y[9]	n+1	Y	Poor	N	Fine	I	AD[13]
Matlab*G	Y[9]	n+1	Y	Good	Y?	Fine	I	AS[12][13]
JavaPorts	Y[9]	1	Y	Good	Y?	Coarse	B	AD[13]
Rivera et al.	Y[9]	n	Y	N?	N?	Any	B	U[14]
Lather et al.	Y[9]	n	Y	Poor	N	Any	B	U
Matlab on GreenGrid	Y[9]	n	Y	Poor	N	Any	B	AD[13]
MultiMatlab	N	n+1	N[10]	Poor	Y	Fine	I	U
Users-Grid	Y[9]	1	Y	Poor	N	n/a	B	AD
MATCH	N	1	Y[11]	Poor	Y	Fine	I	U
Geodise	Y[9]	1	n/a	Fair	Y	Any	B	AD
Condor+Compiled Matlab	Y[9]	1	Y[11]	Poor	N	Coarse	B	AD[13]
Teodoro et al.	Y[9]	1	Y[11]	Poor	N	Coarse	B	AD[13]
Ideal	Y[14]	[0,1]	Y	Good	Y	Coarse	I	AD

in many other middleware solutions is that NetSolve mandated that each functional request be a self-contained entity, i.e., a single job and consequently, data sets and the relevant application data (for example program code) were frequently transferred multiple times to the same nodes. For applications such as parameter sweeps this is an expensive requirement. Oberhuber accounts that most of the problems with DIPS were NetSolve-related, other issues of relevance were with regard to flexibility, and reliability. The issue of flexibility relates to the complexity of generating (even automatically) the calls to the NetSolve infrastructure. For example data sizes must be known in advance for both input and output. In addition, NetSolve's resource descriptor language was too limited for image processing applications. In terms of reliability, NetSolve would loose without documentation 3% of all jobs.

Oberhuber identified that in order to achieve better performance, schedulers need to have access to application specific information e.g. workload and available bandwidth, and that good performance can only be obtained when scheduling is explicitly integrated within the application. Here, crossovers with other approaches can be identified such as AppLeS [119] and DIANE [120, 121]. Hence, monitoring is an important facility, but, horizontal tools like NWS [122] and Ganglia [123] can only provide an overview of the scenario, for deep analysis of the current situation or state, custom monitoring tools are required to augment the off-the-shelf equivalents. Essentially, the idea here is to include application level information into the scheduling strategies, and make the scheduler application-driven/aware.

However, there is a clear challenge with this observation that Oberhuber has not considered. Many platforms such as NetSolve, Condor and the others presented in Section 3.2 use generic scheduling routines, such as LSF [124], LoadLeveler [125] and PBS [126], or custom scheduling routines to fulfil the agendas and intentions of the middleware. In order to perform application level scheduling these generic schedulers need to be: side-stepped, replaced, or used only for resource allocation. This would enable the application to steer the resources allocated to it based upon its own agendas and integrated scheduling routines. The first two solutions are out of the question without upsetting other users of the resource management tools. Therefore, a change in job model and the isolation of control is the only remaining option. Adding monitoring systems at this point is relatively simple, as they can be intrinsically integrated into the application itself, or hosted externally as required.

This is also closely related to the descriptor or interface definition language for the resource management software. In many cases they are either too limited for image processing applications or enforce unrealistic requirements on the application developer. Here, Oberhuber hints toward (but does not explicitly state) the need to separate the methods of acquiring computational resources from the job description. It is in the aim of generality that work description languages[15] become problematic, especially for numerical data. Consider this example, when Matlab is called from the

[15]Examples include: the XML-based JSDL [127] specification, from the Open Grid Forum, gLite's [128] Job Definition Language, which describes resource and job preferences using the *GLUE*-Schema [129], the Common Information Model [130], Globus's Resource Specification Language as well as its XML-based Job Description Document, and Condor's submission scripts and schema-free ClassAd methodology.

command line, and passed numerical input, the input is read as a number. However, if the same call is made for a compiled application, Matlab will use the ASCII values for the parameters to construct the input. Many other application engines would fail at this point, but as Matlab binds its variable types at runtime, it will assume that the input is a one-dimensional matrix. There is no solution to this problem using a general work description language, without transferring the data as a flat file and reading it from within the application itself.

The most detrimental issue of DIPS was NetSolve starting a new transfer and image processing session for every request. This would have even been the case when the same job was performed on the same data and on the same node. No implementable solution is given on how this can be avoided. This issue is not specific only to NetSolve; Condor too enforces that each request be self contained. Even when Condor is used in conjunction with systems such as Pegasus [99–101] and Condor DAGMan such issues are not necessarily resolved. Nimrod imposes the same restriction, as do many of the other systems presented in Section 3.2. The issue here is that a workstation or node is being used for the life of a job and not for the time that it is available or needed. The development of DIPS was carried out such that it became tightly coupled to, and therefore dependent on, NetSolve, with the consequence that to replace NetSolve with another resource manager, significant changes to the DIPS implementation would be necessary. This also highlights the importance of the implementation to be able to swap the underlying resource managers with minimal effort. Here, GridSAM [131] could be a useful option for production systems. However, the use of GridSAM means adopting a general job description language, which is less desirable.

Multi-institutional networks also provided a problem, inter-site communication overheads made DIPS unfeasible. Oberhuber acknowledges that DIPS and other similar approaches are best suited to LANs and single site networks. He also accounts that clusters are too tightly coupled, at the time metacomputing was not yet mature enough and loosely coupled architectures could not usually achieve good performance for fine grained parallelism. Some of these points are debatable today, however, Grid Computing as well as wide area distributed systems are still not well suited to image processing unless the job granularity is coarse, using the definition of granularity in Section 4.9.1, and a runtime of reasonable length, i.e. tens of minutes or greater.

The outcome of this research was the demonstration of the potential for seamless access to distributed image processing capabilities, using Campus-Grid like infrastructures. However, the concluding remarks were that more research is required in order to maximize their benefits and on how this should be achieved. Many publications, including the NetSolve website, often reference DIPS as a success, which was not the case. DIPS was by no means a failure, as much was learnt from the project.

Concurrent Distributed Image Processing - CDIP

Following DIPS, CDIP was developed to learn how to support the user with remotely located data and remotely available image processing methods running on parallel computers, thereby exploiting the possibility of concurrent execution of algorithms [23]. CDIP was constructed using Java, CORBA and NetSolve. It used a variety of image processing packages, but kept the original DIPS Image/J front end. Its applications revolve around the processing of large SAR (satellite imagery) data sets, which exhibited the requirement for a fine-grained parallel model.

Multiple Image Processing packages were amalgamated in order to overcome a lack of image processing algorithms and to provide a highly optimised parallel method repository [116]. However, the interleaving of several IP packages did not drastically improve the cohesion of a system as a whole. Instead, many challenges, some which CDIP could not answer, stemmed from this aim. A detailed knowledge of the procedures was essential, particularly which algorithms are available on which machines and also how to map between them. This meant that the scheduler required a large repository of data in order to convert a request into a scheduleable entity [47]. This was a similar idea to other approaches such as Condor's ClassAd scheme, but differs in that where a ClassAd typically publicises the availability of an application in general, CDIP was trying to publicise the functional availability within an application platform. Scheduling complexity was intransigently increased due to the need to consider algorithm locality and ensuring that deadlock was avoided. Inevitably, adding new functionality was a timely and complicated process, as metadata for each new function needed to be generated.

Despite the added complexity of scheduling tasks for multiple image processing libraries and operating systems, a simple but effective scheduling and allocation mechanism was implemented. Decisions were based upon estimates sourced from past experiences and interactions. Oberhuber's suggestion of application level scheduling was also worked on to some extent. Here, application level data was used to classify a machine's performance. Then based upon this data their scheduler would identify the "*best*" match for a job, rather than to improve the management of the application as a whole, as Oberhuber intended. This, however, required a deep prior knowledge of the application, which included: **(1)** execution time of the whole job, **(2)** cost, accumulated by network traffic, processor hours, and potentially algorithm/software license fees, and **(3)** some user-level metrics such as job priority and whether the use of special hardware is permitted, which may reduce the number of possible configurations.

This approach is documented in [47], but not validated through experimentation. Yet, the effectiveness of the approach is demonstrated in [23]. However, in these experiments HPC machines were used and not clusters of workstations, which was the initial aim. In this case the scheduling requirements could be pruned, but it is not mentioned how this was done, or the effect. For example the scheduler no longer needs to know if special hardware can be used as only one HPC machine was available. This also reduced the number of potential candidate machines to run jobs on, again simplifying the scheduling strategy. Details about a method's implementation are also not required at the same level, as resources are now homogeneous in terms of hardware, available software and image processing functionality. However, the use of dynamic scheduling and nodes requesting more work as they start to run out enabled good performance to be achieved. This, however, only occurred when the following was considered: speedup increased linearly as more processors were added until the number of processors exceeded $\frac{1}{4}$ of the number of jobs. Past this point the load balancing strategies started to loose track and become too opportunistic, with the consequence that the speed up began to flatten out and eventually decrease [23, 116]. What this means is that the scheduling strategy would permit too many machines to be used, which in turn could decrease performance and efficiency and increase the application makespan.

The conclusion of the CDIP project was that more research is necessary to prove that this simple approach satisfies more generic application, or to find more elaborate strategies. In addition, it was concluded that Java and Netsolve could indeed achieve an ease of access and a gain in efficiency and throughput in distributed image processing. Remember that this work is based on an HPC environment and was not actually evaluated using the clusters of workstations it was intended for. Such an environment is much more volatile than an HPC environment, most notably due to the change from dedicated to voluntarily provided resources. One of the limiting factors of the approach was the need for very high bandwidth connections [116] and running a fine-grained approach with the huge amounts of data used in SAR processing[16] may draw out the requirement for a different approach to load balancing. In such a case it is not unlikely for the retransfer of data to be more expensive (timely) than choosing not to reallocate a job. The CDIP publications also report no improvements or differences with respect to the use and practicality of the NetSolve backend.

Virtual Resource Browser

The Virtual Resource Browser (VBrowser) [132] is a part of the Virtual Laboratory for e-Science (VL-e) [133] project and acts as a front end to Grid resources for a variety of medically-based applications. The resources in question are several dedicated clusters of Linux nodes, where resource management is performed by a single administration entity, and every node is a carbon copy of a single master machine image. Here, faults such as those shown in Section 2.4 are not apparent and therefore the scheduling strategies and resource management software does not need to consider the challenges of resource volatility and owner or workstation autonomy.

The use case of the VBrowser is to provide a simple and transparent point of access to resources for use by medical researchers. Typically, this is performed in batches of independent jobs or by workflow engines when dependencies exist [134]. The current implementation adopts the use of the Moteur [135] system. Many of the applications are Matlab-based, where one example is the processing of Magnetic Resonance Imaging (MRI) data using a variety of third party and in-house toolboxes. The VBrowser uses the gLite [128] middleware to access the resource fabrics and, as with most middlewares of this type, cannot perform job submission outside of the batch processing paradigm. This is also the case for Matlab applications, which until recently[17] have been through the use of licensed instances of Matlab.

A significant attribute of the VBrowser stems from its graphical user interface, which has the look and feel of the native operating system. Jobs are submitted through a drag and drop approach,

[16]In: [116] each task is associated to 0.5Gb of data.

[17]As part of the research for this thesis some collaboration was performed, which enabled the submission of compiled Matlab applications using the vBrowser. These applications can be compiled using approaches such as the semi-autonomic compiler presented in Section 5.2.1.

where the user drops in their job definition, which in most cases is a Moteur workflow description and any required data or executables. The VBrowser then handles the intricacies of job submission and result gathering for the user, delivering results as required. The future challenges for the VBrowser include the capitialization and harnessing of spare workstations on campus, however, as the VBrowser and the associated software has had no prior need for the fault management techniques needed to side step the idiosyncrasies of campus-based resources, it is currently not in a position to be simply mapped to such an infrastructure. In addition, the current approach does not permit an interactive use of resources and therefore the performance guarantees in terms of efficiency for short jobs cannot be made.

PIMA(GE)^2

Clementis et al. [136] present an approach to transparently access a distributed image processing architecture realized through a parallel CORBA front-end for an MPI-based image processing back-end. Henning reported in [137] the many problems associated with CORBA, both in terms of application development and the significant problems with CORBA's component model. It is not surprising that this approach to distributed image processing is hampered by high communications overhead, which based upon their experimental results reaches 38% of the total wall clock computation time when using just 8 machines. This may arise from the fact that CORBA was never intended for the development of parallel applications [136] or that the CORBA is architecturally flawed on many levels [137]. This is further accentuated by Goller [116] who identified the need for a reliable version of CORBA. This paper also shows the importance of feedback-orientated scheduling and resource allocation: if left unchecked this system would allow the user to create a scenario where adding more nodes would start to increase computation time rather than reduce it.

Remote Sensing Information Modelling with Condor

Cai et al. [138] describe a simplistic approach to the processing of satellite data by using a small Condor pool. The authors unintentionally present a key concept which confirms the observations made in Chapter 2. Namely, the challenge of running short jobs on a Condor pool efficiently. They performed comparative analysis on a range of data sets with different sizes on one workstation and compared this to a 5 node Condor pool. The experimental set up uses Condor as a scheduler for a set of dedicated resources, rather than in its typical cycle stealing context. This makes for quite a useful comparison, because there is no excuse for Condor to perform badly. In fact, this situation should favour the performance results, as no resource volatility or autonomy can hamper the performance of the Condor system.

Cai et al's jobs are each of 180 seconds or less in length when run on a single workstation, which is one of the workstations used in the pool. In addition, all workstations are of a very similar nature and set up, making the pool largely homogeneous. In each of their application scenarios their Condor pool required more time to complete the job space than a single machine, when the entire application makespan is considered. This was primarily due to the overhead in scheduling, where between 60 and 70% of the application makespan was spent in Condor's administration of the job space. If, however, only the time Condor spent executing the jobs is considered, then their application received a good speed up factor. What this shows is that even in a clean and practically dedicated context Condor cannot service short jobs efficiently, but that the potential is there if the right solution can be identified.

ProteomeGRID

ProteomeGRID [139] uses a combination of Condor and CORBA to perform parallel image processing. The constitution of the Condor pool is 200 heterogeneous workstations within the same computer lab, and therefore same administrative domain. The approach is to distribute a CORBA-based worker daemon as a regular Condor job onto host workstations. Once initialized, a worker daemon calls back to a specified point to request image processing functions to execute. Here, each image processing function is a CORBA ORB with an `execute` method, provided by the application developer. This means that in order to use the system, existing code must be mapped to or encapsulated by an ORB. It also assumes that the image processing function does not have dependencies on other ORBs or third party libraries. However, the benefit is that no underlying libraries need to be installed on the Condor workstations. Therefore, there is no need for mechanisms to provide stability in response to workstation volatility.

The system is deployed using the standard Condor model where one Condor job relates to one worker instance. Workers are distributed with all the necessary libraries needed to initialize the basic daemon instance, capable of accepting and executing ORBs. All other program code and data to be processed is transferred to the host workstation at runtime, using an external scheduler. The ability to transfer all application dependencies at runtime also translates into another benefit: namely, that the Condor scheduler does not need to match, as many resource requirements for job execution and therefore the time required for matchmaking is reduced.

The resource utilization efficiency of this approach is reported to be up to 81%. However, it is not clear how this value was calculated or what the runtime of a single job is. What is clear, however, is that the time required to acquire idle workstations and then initialize the worker daemon is not included in the calculation. By using the provided performance data and the brief experimental description, it was not possible to achieve the same values presented in the paper, using the equation for calculating the efficiency of resource utilization (Equation 2.2 in Section 2.4). Instead, an efficiency of 65% was determined, which is a sizable difference to the reported efficiency. When the time required to acquire the necessary resources (135 seconds) was also included, this further reduced to 54%. However, even with this consideration the general performance of the approach is good. This is due to the capability to schedule jobs directly to a workstation without the need to invoke the Condor scheduler. It also means that the workstations are used for as long as they are needed or available rather than for the runtime of a job, which may be short.

This approach cannot be simply mapped into the application scenario of this work, as there are three critical incompatibilities: **(1)** no mechanisms for the management of workstation volatility are included. They are not needed, as workstations reside in a single administrative domain and no software libraries other than Condor need to be installed for the application to run. (2) Matlab functions have deep dependencies not only within the MCR, but also other with user defined code and other third party functions, as Matlab employs a functional programming model. Therefore, identifying and transferring the necessary functionality could be quite challenging to perform efficiently, especially if the code must also be compiled, which, as discussed in the review of JavaPorts, is not a trivial process to perform in an automated manner. The increase in communications overhead experienced as a consequence of the transfer of functionality could also be significant. (3) The Matlab MCR cannot be transferred at runtime in the same manner as the CORBA libraries, as it is many orders of magnitude larger and would generate a sizable networking bottleneck, even with replicas on the network. It would also be prone to file transfer errors and significantly increase the initialization time of a job.

Image Processing for Grid (IP4G)

Image Processing for Grid (IP4G) [21] is a toolkit for developing Grid-enabled medical image processing applications. In this approach jobs are represented using an XML-based description language. Consequently, the job description file is somewhat complex even in the simple example presented. In fact, defining the job requires more lines of XML than image processing function provided as an example. Here, it seems an XML representation is complicating the approach, but not aiding it in any way. In addition, processing time will be increased in the generation and parsing of the description.

Three toolkits are used in this approach, DataCutter [140], VTK [141] and ITK [142], which are used for representing image processing filters, data visulisation and algorithms for image segmentation and registration respectively. The experimental results presented indicate that overhead in this approach is a minimum of 25%, as this was the overhead experienced when the system was run on a single machine. The high overhead is reportedly due to data serialization and copying, which stems from the splitting of processing into separate filters, which is imposed by the DataCutter model. Bovenkamp et al. [67] observed that the control of image processing models is the most common major bottleneck in distributed and parallel image processing systems. Here, this illustrates the importance of choosing an application container that will not impose unrealistic conceptual models. As the number of nodes increases, it can be expected that the overhead will also increase. However, the authors report that this can be reduced if transparent copies of the data are placed on cluster nodes prior to execution, which, they suggest, could be performed manually by the user. When such a scenario is in place the system schedulers can then attempt, but not guarantee, to map jobs to nodes where image data is already located. This assumes, firstly, that the user knows in advance where their jobs are going to run, which is based upon the scheduling model and availability. Secondly, it assumes that the users have sufficient access to pre-stage the data manually.

Image Processing Grid Platform (IPGP)

Image Processing Grid Platform (IPGP) [143] is a middleware, which is built on the existing web services technology and targets image analysis and visualization applications on a Grid. It is a layered framework and uses service-oriented components and workflow techniques to specify a model of Grid applications. Here, an application is described as a Grid workflow composing of component activities, where each activity has one or more resources satisfying the service. For this reason, resource selection is a scheduling process.

The performance results of IPGP raises questions about the suitability of Grid Computing for image processing. The analysis of the characteristics of Grid environments and image processing applications found some inconsistencies. Grid resources are widely distributed with low bandwidth, high latency and non-stable performance. Image-processing applications, however, are data-intensive and consequently, this leads to an abundance of data transfers between interconnected components of an application workflow, which causes performance to be inhibited by infrastructural limitations. The IPGP authors concluded that in order to optimize performance, the conjoint activities should be scheduled to single or low-price Grid nodes, if possible. Here, a single node could be a cluster or physical machine depending on application requirements or context. If the first alternative were employed, this can significantly increase the scheduling complexity and directly impact upon any economic attributes that may be associated to the submission. Should the latter case be adopted it does raise the question of why the job should be distributed, as any improvements in performance are likely to be negated by data transfer costs. Given the challenge of image transfer in distributed image processing, is Grid Computing really suited to image processing? In situations where costs are based on time rather than load it seems apparent that distributed image processing approaches are not well suited to commercial Grids. Facilities like the NGS,[18] however, provide a very cost effective solution, but do not address the specific scheduling challenges that are resounding challenges in distributed image processing.

Grid Services with the GIMP

Petcu and Iordan [24] used the GIMP in combination with the Globus Toolkit [50] to create a completely license free distributed image processing system. The Globus toolkit is used to launch GIMP daemons onto remote machines to serve a client's request. A GIMP command is sent to the remote GIMP server, which contains a GIMP operation or script to be applied on specified remote image file(s). In the case of a script, this must lie on the remote computing node. It is not clear how scripts are transferred or if the system is limited solely to preinstalled functionality. The approach also assumes that any required data is accessible or already available at the remote site.

The inherent GIMP functionality provides the means to be run as an interactive daemon, and therefore enables users to have interactive sessions with a remote server. The reported initialization times are very good, a matter of seconds, which demonstrates the effectiveness of the GIMP for

[18]National Grid Service: http://www.ngs.ac.uk

use in such a context. The authors specify that consecutive operations can be performed, but only in reference to the previous operation. This immediately questions whether the services can be used for heterogeneous jobs, and how reusable the services are. The main issue with this work is that it appears to be a system which provides remote access to a single image processing resource, rather than to multiple image processing resources. This question arises for two reasons, firstly, due to the briefly discussed application, which is indicative of a sequential programming model, and, secondly, because the paper reports no experimental evaluation. Similarly, there is no methodology which describes how resources are chosen.

Parallel Image Processing over a Cluster using the GIMP

Czarnul et al. [31] document an approach that utilizes a cluster of workstations in order to setup a pipelined architecture for processing sequences of image processing calls. A load monitoring system is used to select the least loaded nodes to launch GIMP daemons upon. Here, the system aims to optimize the application makespan and minimize wait times. This is apparent in their results, as significant speedup through the selection of least loaded nodes has been observed. In the pipelined architecture they note that even small amounts of additional background processing can create bottlenecks in performance. The problem here is that jobs cannot progress to the next stage of pipeline until the job currently in the next stage has finished. This raises the question of why a pipeline is used, especially considering that each processing node is running the same application container; the GIMP. In addition, the communications model can be either MPI or PVM, which do not really fit into a pipelined architecture. Consequently, performance is degraded by high communication and synchronization overheads stemming from the pipelined approach. This is culminated in the high overheads experienced from data transfer and retransfer down the pipeline.

Resource-Aware Visualization Environment

The Resource-Aware Visualization Environment (RAVE) [144–148] project (Cardiff University) aims to facilitate the visualization of graphical environments, specifically for the support of teaching medical students of posthumous human anatomy. The specific use cases and applications which RAVE has explored are in the areas of visualizing medical imagery on lite devices such as PDAs and other portable devices. RAVE is introspective in that is can identify whether a device is capable of rendering the required viewpoint(s) or not, in the case of the latter it adopts a fine-grained parallel model and farms out the processing to available resources. The architecture of RAVE is a built upon web services, and resources are discovered through the use of UDDI registries. RAVE also detects when a rendering engine struggles to meet the demands placed upon it and will perform load balancing as required. Typical use of the RAVE system is to render a scene in response to user steering in real time, and therefore can place high demands upon the resources it uses. What is not clear, however, is how the service directories are instantiated, i.e. manually or dynamically in response to an observed demand. From the literature it appears that the latter is not the case, and therefore the assumption of RAVE is that the necessary resources will always be available to meet the demands placed upon it. This is perhaps not the best of solutions and therefore limits the reusability of the RAVE software in other scenarios. Research to reduce the need for statically bound resources is a current research aim of the RAVE project. Similarly, is the aim to move away from the currently adopted dedicated resources (for example HPC machines of a SGI origin), toward more opportunistic sources such as a Condor pool.

Recent work in RAVE of the most relevance is presented in [148], where processing engines are deployed over a collection of open access Linux workstations. Here, the processing engines are pseudo-manually initialized through the establishment of multiple SSH sessions (one for each workstation). However, there is the incentive to automate this process using Cardiff's Condor pool. Yet there are challenges in preparing the processing engine for such a scenario aside from those presented in Chapter 2. Namely that the processing engine is built upon Trolltech's Qt, which under *nix operating systems is freeware. Under Windows, however, this is not the case and the cost of individual licenses is comparable to the cost of Matlab licenses. There is only really one feasible solution to this problem, which is the use of Condor's vm (virtual machine) universe, however this is currently not an option for reasons that will be explained in Section 3.2.5.

Myriad

Myriad [149] is a proof of concept framework for building networked Machine Vision systems [150] and is orientated around the question: What happens when different sections of Machine Vision systems are networked together, possibly over large distances? A Myriad network consists of one or more components, e.g. a cameras, device controllers, databases etc. The only requirement of these components is that they can accept HTTP requests and output HTTP responses. Should this not be possible, a Myriad plug-in can be used. The model of Myriad is that every component acts autonomously in response to a request on the premise that components will interact if needed, e.g. if component A receives a request it may in turn become a client of component B, who may then become a client of C etc. In this way large command chains can be established. However, should one of these components fail, it would be difficult to determine where this failure occurred. Similarly, fault tolerance in such an approach would be difficult. An important feature of a Myriad network is that it is decentralised allowing for a potentially powerful approach, but one that is difficult to program effectively [150].

There are several benefits of Myriad: **(1)** Its communication is HTTP-based, which is very firewall friendly. It also means that other forms of security e.g. SSH tunnels, encryption and PKI are easily adopted. **(2)** It is possible to have a front end to the system in practically any programming language. This was demonstrated with an interface to prolog [151].

The disadvantages of Myriad stem mainly from its proof of concept nature. **(1)** Myriad is based on the idea of zero config networks, but there is no method currently available to advertise or discover Myriad components, the user/programmer must know their locations when developing an application. **(2)** Dependency chains can be long and therefore fault tolerance is difficult to implement. **(3)** To include a new component for which no Myriad plug-in exists one must be written. **(4)** As said above, Myriad is difficult to program effectively. (5) No method to automate the initialisation process of a Myriad network exists.

There are also a few unknowns in Myriad which would need further research in order to move from a proof of concept implementation to a usable software solution: (1) No performance testing has been done to determine where the capabilities of a Myriad network lie. (2) No experiments have been performed to test the scalability of the system.

Summary

In this Section a discussion of several approaches for distributed image processing which are not based upon Matlab have been presented. The primary difference between these approaches and the Matlab-based approaches is that they are not confined by licensing restrictions, and therefore should be able to define more flexible models and invocations of image processing libraries. However, for these approaches to be useful in the context of Matlab-based applications, they would need to be capable of consuming Matlab resources, either in a licensed or preferably a license free context. In addition, they also need to be able to answer the general challenges of the Matlab-based approaches. In Table 3.2 they are assessed and summarized using the following criteria:

C1: Could potentially run an engine on a Campus-Grid workstation
C2: Could potentially use Matlab
C3: Could potentially use compiled Matlab
C4: Transparency of code distribution
C5: Ability to include new user-defined functionality
C6: Ability/orientation to run short jobs
C7: Granularity
C8: Resource utilization model
C9: Node acquistion model

It is very difficult to always ascertain exactly whether an approach could facilitate Matlab directly or not. In some cases it is simply a case of swapping one or more libraries for Matlab. However, for the approaches that use an interactive resource utilization model it is also a case of providing an interface to Matlab, which could be provided by something such as JMatLink [152]. It is even less straightforward to ascertain whether these approaches could support compiled Matlab, as there are no documented approaches in the literature that report the ability to interactively control a compiled instance of Matlab. For this reason the approaches with an interactive model are noted as not being capable of executing compiled Matlab jobs.

Table 3.2: A summary of the attibutes of each approach, which are important for the application context of this research. Acronyms: Y - Yes, N - No, Y? - Unclear from publications, but is likely, N? - Unclear from publications, but is unlikely, ??? not determinable from available literature, B - Batch, I - Interactive, A - Automatic, S - Static, D - Dynamic, U - Performed by user.

Approach	Criteria								
Name	C1	C2	C3	C4	C5	C6	C7	C8	C9
DIPS	N	Y[19]	N	Fair	Poor	Fair	Coarse	B	AD[20]
CDIP	N?[21]	Y[19]	N?	Poor	Poor	Fair	Fine	B	AD
vBrowser	Y?[22]	Y	Y	V. Good	Good	Fair	Coarse	B	AD[23]
PIMA(GE)^2	N	Y?	N?	???	???	Fair	Fine	I	AS[24]
Cai et al.	Y[22]	Y?	Y?	Poor	Fair	Fair	Coarse	B	AD[23]
ProteomeGRID	Y[22]	Y?	N?	Good	V. Good	Good	Coarse	I	AD[24]
IP4G	N	Y?	Y?	Poor	Fair	Poor	Coarse	B	AD
IPGP	N	Y?	Y?	Good	Fair	Poor	Coarse	B	AD
Petcu & Iordan	Y?[22]	Y?	Y?	Fair	Poor	Good	Coarse	I	AD[23]
Czarnul et al.	Y[22]	N	N	Good	Fair	Poor	Coarse	I	U
RAVE	Y[22]	Y?	N?	Good	Fair	Good	Any	B/I[25]	AD
Myriad	N[26]	Y	N	Fair	Poor	n/a	n/a	I	U
Ideal	Y[27]	Y	Y	Good	Good	Y	Coarse	I	AD

It is evident that there is no single approach discussed in this section that meets the minimum criteria needed for running compiled Matlab jobs in a volatile Campus-Grid environment. ProteomeGRID, however, provides much of the basic functionality, but connecting its CORBA-based communication model to an interactively used compiled Matlab instance would be challenging and may prove to be too much of an architectural change to be feasible. The vBrowser also caters on many fronts, however, it has not yet been experimented within a Campus-Grid context and the shift to an interactive utilization model would require large architectural and middleware changes.

3.1.3 Summary

The foremost observation is that performing distributed image processing well is a significant challenge [29, 31, 32, 45, 68]. This is made all the more challenging through the implications of the analysis presented in Chapter 2. No approach covered thus far would be able to cope in these

[19]Due to NetSolve's ability to run Matlab jobs.

[20]1 per job.

[21]Intented to use Campus-Grid-like resources, but the approach was never employed in this context, and there are issues whether it is possible.

[22]No volatility-related awareness or tolerance.

[23]1 per job.

[24]User defined quantity.

[25]Later work has enabled an interactive resource utilization model.

[26]No means of resource discovery or automated deployment.

[27]Tolerance for resource volatility.

conditions, many would not even be able to handle the general opportunistic environment. Two approaches, however, stand out above the rest: JavaPorts and ProteomeGrid. The abilities of these two approaches compliment each other, but even when combined do not offer a complete solution. Neither can tolerate a volatile resource context, as both have been deployed in domains where administration is centralized and therefore do not need to be concerned with such issues. In addition, neither architectural model could support the efficient use of short running compiled Matlab jobs; JavaPorts due to its compilation strategy, and ProteomeGrid due to its approach of transferring functionality at runtime. Currently, the literature presents no implemented solution for the interactive control of a compiled instance of Matlab, which would yield a much more effective and efficient use of resources. This would be made possible by avoiding the potentially lengthy initialization process (relative to a short job's runtime) being repeated for each job. If a solution to the interactive control of a compiled Matlab instance were developed, it is still questionable whether it could be simply included in either approach, without significant architectural changes: firstly, JavaPorts uses a batch programming model, and secondly, due to ProteomeGRID's encapsulation of functionality in the job object.

3.2 Efforts in Volunteer, Opportunistic and Distributed Computing

In this Section a discussion of the approaches for the facilitation of general applications is presented. Here, approaches do not cater directly for image processing, but scientific computing in general. Whilst this means that approaches are more general, this leaves more room for custom approaches to build upon a basic infrastructure. In Section 3.2.1 approaches that offer the ability to use remote resources in an opportunistic context are presented. In addition, approaches that are less orientated around opportunistic computing but have similar goals are also presented. Section 3.2.2 presents a brief overview of some of the efforts in Condor-related Campus-Grids. Section 3.2.4 presents a few choice approaches to customized scheduling routines specifically for application-level scheduling. Finally, in Section 3.2.5 a discussion of why resource virtualization is not a feasible option for the context of this research is presented.

3.2.1 Resource Management

Condor

The Condor project [12, 13] started at the University of Wisconsin in 1984. Since then development has continued and Condor is probably the most widely used (and supported) system for the leverage of idle workstations. Many clones of Condor exist and are still being created, however, typically they fall short of Condor in terms of ability, performance and flexibility.

Workstations become resources in Condor when a set of pre-defined and customizable conditions are true, and, similarly, cease to be a resources when these conditions no longer hold. The core idea behind Condor is that workstation owners voluntarily provide their resources when they are not needed, i.e. idle. The primary indicator utilised by Condor to determine the idleness of a workstation is mouse and keyboard movement. The administrator(s) of the workstations can also impose other metrics, but the default set also considers CPU activity and memory utilization. ClassAds [153], a schema free abstraction for the representation of state and attributes, enable custom user-defined attributes that can be used for workstation selection and indicators of state.

Jobs in Condor are defined using submit scripts, which encapsulate the necessary information to run a job on a remote workstation and also provide the scheduler with metrics for workstation selection. Examples include job specific requirements, which can also be unique to only a few workstations, such as the availability of the MCR as used in Chapter 2. Alternatively, more general attributes such as the amount of memory or CPU speed can be elicited as preferences. However, as the complexity of job requirements increases, so too does the time needed by the scheduler to perform the matchmaking required for workstation allocation. Once a job has started on a workstation, it will either run to completion, fail or be evicted. Should a better workstation become available in the future, a job cannot choose to migrate to it.

The clear advantage of Condor is that users do not need to significantly modify their application to run it on Condor. However, when an application consists of many sub-jobs, the process of generating multiple submit scripts by hand is tedious. For this reason many Condor administrators write customized applications for their users to automate script generation (this process is carried out at both Cardiff and Oxford [17] Universities). In addition, other approaches such as Pegasus add to the Condor model by further abstracting the job submission procedures. Other systems, such as NetSolve [28] and Nimrod [154] can use Condor for a back-end source of computational power. Condor-G [27] is the amalgamation of technologies from the Condor and Globus [50] projects. From Globus: the use of protocols for secure inter-domain communications and standardized access to a variety of remote batch systems. From Condor: the user concerns of job submission, job allocation and error recovery [39].

There are several different modes, defined as universes, for job execution. The most relevant universe to this work is the vanilla universe, which is the most commonly used universe on Windows workstations. As Condor was initially designed for Linux, many of the advanced, and most useful, features, which are part of the standard universe do not work under Windows; checkpointing and job preemption[28] as well as migration being the foremost examples. These abilities are incorporated by recompiling the application code against Linux-specific Condor libraries. Given the ubiquity of Windows, this is rather inconvenient. Should a user wish to utilise checkpointing (and the other features), they must manually alter their application code and use a custom job manager to reflect their additions. As with many other systems (for example XtremWeb [155, 156], Entropia and BOINC), Condor has no explicit support for short [36] or interactive [157] jobs.

There are some issues in fault tolerance for Condor, as the fault tolerant mechanisms of Condor are relatively simple. Condor works on the assumption that workstations are inherently unreliable

[28]Preemption is the removal of a remote job from a workstation. If migration is available, the job will be checkpointed and moved to another workstation, if not it will be aborted [5].

[39], and uses a monitoring process (named a shadow) to monitor changes in workstation state. One of the core failings of this approach is the implicit assumption that if a job fails for reasons other than a change in workstation state - such as preemption - it is the fault of the job and not the workstation. In such cases Condor would not reschedule the job because it has reached a complete state, leaving the user to remedy the situation. The Condor job monitoring system provides little additional information and will only resubmit a job if it has been evicted or failed because of an observed workstation fault, such as connectivity failure. In all other cases the user must determine the cause and decide what to do.

DAGMan and Pegasus can be used to remedy such an eventuality, to some extent, in two ways: (1) by allowing a user to specify a predetermined number of retries and (2) executing a job specific (i.e. user defined) pre and post execution script. Here, the user can identify whether the workstation is correctly initialised and if the job has successfully completed. Yet these two systems are not always available to the user,[29] and there is no built-in mechanism to prevent the job from being submitted to the same workstation again. The post mortum analysis of a job also becomes more difficult when other middleware is in use. If a job is executed using Condor-G, but through a foreign batch system (such as PBS), detail beyond "job failed" may not be available, and the job will appear to have failed of its own accord [39]. The Java universe utilises a wrapper program to execute a Java job and performs post execution analysis to determine the meaning of any error in the execution of the job [158]. However, this assumes the knowledge and understanding of the possible outcomes, which is quite a significant assumption.

However, despite Condor's issues and the observations of Chapter 2 there are no serious short comings in Condor for simple applications and in tightly administrated scenarios. Condor also runs on most operating systems. Even when the observations of Chapter 2 are considered, Condor is still one of the most widely used (if not the most widely used) resource managers for the leverage of idle workstations and other computational resources.

Netsolve

NetSolve [28], or as has been called more recently GridSolve [159] aims to create a middleware that provides a seamless bridge between standard programming interfaces and desktop workstations that are used by many researchers and scientists and the rich supply of services provided by a Grid architecture [160]. NetSolve is built on NetLib and provides access to numerical software (e.g. Matlab) from various client systems. Like Condor resources are benchmarked when included in the system and a monitoring daemon is started as a background process. The benchmark results are then used in order to identify the best available match for a given resource request.

A major advantage of NetSolve is the number of different client interfaces available, making it accessible from many different programming languages. NetSolve also enables the use of resources on-demand directly from within an application. The system essentially allows the user (or an application) to create jobs at runtime and use the results at an application layer. The NetSolve scheduler

[29]Neither are available on the Cardiff University Condor pool.

can also directly access a Condor pool as a source of resources. Oberhuber [115] highlighted many practical limitations with the NetSolve job representation and scheduling model, which were discussed in Section 3.1.2. It is also unfortunate that, according to the NetSolve developers in [160], it is difficult to integrate third party code and libraries into the main infrastructure. This is a fairly restrictive issue, and when combined with the observations of Oberhuber the feasibility of NetSolve is questionable.

Entropia

Entropia [9], is essentially another version of Condor, but with subtle differences. Both use a form of matchmaking, harvest idle cycles, use node managers, and use job suspension and eviction/migration. They differ in that whilst Condor uses one unified scheduling strategy, Entropia can use many, and where Condor uses code modification to facilitate checkpointing, Entropia uses it for sandboxing an application. The use of sandboxing is really where Entropia and Condor differ. Entropia uses a virtual machine to run an application: code is modified to ensure that all basic Windows functionality is redirected to protect both the host workstation from malicious applications and applications from malicious hosts and owners. In this sense Entropia is better than Condor, but Entropia supports a smaller application range than Condor. Applications in Entropia have no dependences between tasks, i.e. embarrassing parallel. The only requirement for Entropia is that an application is a Window executable.

An advantage that Entropia has over Condor is that its scheduling overhead is lower, approximately 40 seconds per job [36] (this does not include the time for data transfer). Realistically this means that a job should not be less than 5 minutes. The main limitation with Entropia is that it is not Condor, and for this reason it is not as widely supported or deployed. Outside of Entropia's home University (Berkeley) there are very few Entropia installations. Entropia is also much more focused on a specific application area, whereas the Condor universes cover a much wider range of applications, for example the vm universe for virtual machines, a parallel universe specifically for MPI-based jobs and a globus universe specifically for the submission of Globus-based jobs and the Java universe specifically for Java-based applications. It is likely that this flexibility of Condor is another reason for its more widespread adoption.

Alchemi

Alchemi [15] is a .NET-based framework that provides the capability and programming environment to develop and distribute applications in a Grid scenario. Resources in Alchemi are either Linux or Windows based and harnessed opportunistically from an volunteered pool of resources. It also adopts a coarse grained abstraction and has support for the inclusion of legacy systems. Jobs are described using a bespoke XML representation, or through the definition of a job object. Any external application dependencies must be manually highlighted in the description, and are later encoded fully into the description, on receipt of a description a worker node decodes the dependencies and writes them to disk. Consequently, such dependencies may be issued to a resource multiple times.

One of the aims of the Alchemi project is to demonstrate that Windows resources can be integrated with other resource managers and Grid middleware, such as Globus and Nimrod-G. However, there is little documentation in the literature of what fault tolerance is supplied to counter the challenges of the volunteer nature of the resources in use. Similarly, there is no performance data relating to the efficiency of the approach.

Ninf

Ninf [161][30] is a client-server information library for high performance computing. Its aim is to exploit high performance in networked parallel computing and provide high quality numerical computation services. Ninf is an RPC implementation for Grid Computing similar to NetSolve [37]. Ninf servers are used interactively by an application and when a Ninf job is created, the underlying architecture (a metaserver) can choose an appropriate server and delegate the request transparently.[31] Load balancing is employed in a similar manner.

Ninf remote libraries are implemented as executable programs containing a network stub as the main routine, and are managed at the server level. When the library is called by a client program, a Ninf server searches its known libraries and executes the request if the necessary library is available. Appropriate communication with the client is also set up at this point. If the library is not available, the server propagates the request to the local metaservers. Libraries can be written in any existing scientific languages, provided that the host can execute it. Interfaces to libraries are defined using an IDL, which also describes how to compile and link the library. Unfortunately, Ninf does not allow a user to export their program or library to a server. Instead, Ninf remote libraries are installed and registered only by a responsible library provider. In this sense Ninf is more a index of remotely available software libraries, rather than a platform for facilitating computation and this limits its usability in a wider context.

Nimrod

Nimrod [154] is a tool for performing parametised simulations over networks of loosely coupled workstations. Using Nimrod the user can interactively generate a parametised experiment and leave Nimrod to control the distribution of jobs to machines and the collection of results. A commercial version of Nimrod is also available see: [162]. Nimrod is similar to Condor in many respects [37] most notably in that a user doesn't have to change code in order to run it as part of a Nimrod application and in the use of idle or under used resources. The application code is treated as a stand alone program that is executed via a remote execution server [32]. At the heart of Nimrod is a scripting language, which is used to define an experiment and input arguments. Note the potential issues given in the discussion of DIPS in Section 3.1.2, which raised concerns about how arguments are passed to applications. Assuming the resolution of such issues, Nimrod's scripting language shields the user from the intricacies of the system allowing them to concentrate on their application

[30]See: `http://phase.etl.go.jp/Ninf/` – last accessed 2007.

[31]The user can alternatively do this manually.

[154], rather than job administration. In a similar fashion to Condor, Nimrod also has a Globus add-on: Nimrod/G. There are, however, concerns over the ease of administrating Nimrod. Olabarriaga et al. [163] noted that Nimrod can intuitively facilitate the management of experiments, but, that it requires a significant amount of time and expertise to successfully set-up the environment to run the experiments. The preparations included: (1) parametrising the application, (2) adapting the application code to run on Grid nodes (with remote data resources), (3) manual discovery and configuration of Grid resources, and (4) error handling. Nimrod is also strongly orientated toward parameter sweeps, and therefore not capable of servicing applications of other paradigms.

Distributed Analysis Environment (DIANE)

The DIstributed ANalysis Environment (DIANE) [120, 121, 164–167] is a framework based on a master-worker processing model, which is used on top of regular Grid middleware and resource managers such as LSF, PBS and Condor in a transparent way. In addition, DIANE enables the interchanging of the underlying middleware without significant impact upon its implementation. The DIANE framework is implemented in a variety of programming languages to enable greater support for legacy applications. However, this introduces the requirement for inter-language support, which is provided by a communications transport mechanism implemented using CORBA ORBs, the messaging protocols in use are then either SOAP messages or XDR streams. This approach increases the overhead for message transfer, as the encoding and decoding of messages must be performed at least twice between components; firstly, in the representation of the message content, and, secondly, the functional calls between the CORBA transport layer and the components' native implementation. DIANE performs these processes transparently, however, application developers must define the interfaces for in and out going messages. The trade off which DIANE must consider occurs between the conflicting aims of generality and performance. The need for a CORBA transport layer is symptomatic of the first aim, but can significantly detriment the latter aim.

DIANE's execution model is significantly different to all the other approaches presented so far in this Chapter, with the exception of ProteomeGRID. Worker agents are sent to the underlying resources as regular jobs, but by opening a TCP/IP connection, they register to a master agent, which runs on the user's desktop computer and acts as the coordination point for the virtual worker pool. Workers are then given work through the use of a pull scheduling model in which workers request work units when they have depleted their current tasks. This has lead to efficiency of resource utilization values of between 75 and 84%, which is impressive considering the use of CORBA as a transport layer and multiple programming languages in DIANE's implementation.

Workers may dynamically join and leave the resource pool, without disrupting the processing as a whole, as typically the units of computation are a large number of short tasks. Work is distributed by the master to workers directly, therefore bypassing the middleware-scheduling layer. This enables the application makespan to be reduced, but also leads to faster job processing times, reaction to errors and improvements in the resource utilization efficiency. In terms of error identification, DIANE does not consider the full spectrum of errors that were demonstrated in Chapter 2, and only considers a more traditional set of job-related errors, i.e. revoked/crashed workers, and job-related failures such as programmer error. It does, however, enable the developer to introduce new job-based error handling mechanisms, but the error model is applied during runtime, i.e. after the worker has started. Capturing the volatility-related errors of the Campus-Grid may therefore not be possible.

DIANE typically runs in a single-user mode where a worker represents a single user and application. The master schedules jobs in an application-driven and interactive manner, therefore the resources are used for as long as they are available or needed. This is the reason that DIANE can achieve efficiencies between 75 and 84%. The agents do not live indefinitely, but terminate when the user has completed the current processing requests, as issues such as authentication and accounting policies dictate that another user cannot reuse an agent session. However, note that these values do not include any overhead involved in the authentication process, as these experiements were run in scenarios where authentication was not required. System performance when authentication is switched on is not presented in the DIANE literature.

The worker execution model is outside of the application engine itself, the system is application-neutral in this sense. Instead, the worker interacts with the application (Matlab for example) via an external API, provided that one is available. The applications presented in [164] and [165] used a python-based API for this process. However, it cannot be assumed that all legacy applications provide a python-based API for external communication. Matlab for example provides only a FORTRAN and C API for external control, and a Java API for internal control. However, correspondence with the DIANE authors has revealed that in more advanced cases a C or C++ API (for instance via a shared library) can be used, which it then exposed to a python script. However, as the worker is external to the application engine, it is not clear whether the engine remains alive in periods of idleness such as the time between one job completing and the next starting. For compiled Matlab applications this would be problematic, as when an application enters an idle state it shuts down. In regard to the capture of results, external control and a CORBA-based transport layer presents another issue, namely how to represent and transport the results back to the master or head node. To provide DIANE with the necessary information, the user must write a description which defines how to construct and represent results for transportation back to their workstation. The problem here is that such definitions may be application and user specific. In addition, they may require an intimate knowledge of DIANE, the resources used or the application environment.

Whilst DIANE does not provide explicit support for short jobs, performance results presented in [167], illustrate that the DIANE model is well equipped to provide good performance for short jobs (10s of seconds). This stems from the application-level scheduling model and the interactive use of workers. DIANE is the most relevant approach in literature to the work presented in this thesis. However, it is not a complete solution for the challenges and application areas presented in the first two Chapters of this thesis. Firstly, because it focuses mainly on dedicated Grid resources and not volunteer resources, which is a significant difference providing different challenges. Secondly, there is no intrinsic support for the management of software licenses across the different sites, or the ability to identify application-level requirements such as the availability of software, as all software libraries needed are transferred with the worker. Transfer of libraries such as a MCR installation would be cumbersome prone to error and inefficient, as transfer bottlenecks would quickly emerge. However, as a similar approach DIANE provides an additional benchmark in terms of performance of resource utilization efficiency to which this reseach can be compared.

Ibis

Ibis [168] is a Java-based Grid programing environment with several specific aims. Firstly, to optimize communication channels between potentially unreliable, heterogeneous and dynamically changing sites, and, secondly, to provide a streamlined approach for job management across multiple sites, each of which may use different resource managers and middlewares.

Communication in Ibis is provided by the Ibis Portability Layer (IPL), which encapsulates many different communication protocols for different application- and hardware-based scenarios. For example MPI-based communication may be best performed on tightly coupled resources, whereas a TCP/IP connection is preferable between remote sites. The mechanisms which cater for protocol selection are based upon user preferences and monitoring systems that identify various hardware configurations. Message encoding is performed using an optimized version of Java serialization, which enables noticeably higher data throughput. Ibis also uses a SmartSockets [169] system, which enables message routing through firewalls, NATs, private/non-routed networks and to multi-homed machines.

To execute jobs, Ibis defines the Java Grid Application Toolkit (JavaGAT) [170]. The JavaGAT is a streamlined toolkit which incorporates many different middlewares into a single system to improve transparency, fault tolerance and portability. In order to submit a job, the JavaGAT chooses for the user which middleware is needed in order to submit, steer and monitor the job. Should a middleware-related error occur, another middleware option (if available) will be employed until all options have been exhausted. In addition, errors are interpreted by the system to determine what options are available to resolve the situation. The fault tolerance systems are capable of handling software unavailability and implementation-specific failures, which means that potentially the toolkit could handle the scenario in Chapter 2 where Matlab capability is announced but not actually available. This stems from an assumption in the toolkit's implementation that things can and will go wrong in a complex distributed system. However, there are two aspects of the JavaGAT functionality which prevent it from being able to answer the challenges outlined by Chapter 2. Firstly, that it is dependent on the information provided by the middleware. Therefore if the middleware cannot identify why a failure occurred neither can JavaGAT. Secondly, JavaGAT assumes that all errors and failures occur within a reasonable period of time and that libraries do not hang indefinitely. Whilst the experiments in Chapter 2 did not exhibit indefinite hanging of libraries, libraries did hang for inexorable periods of time, which from the user's perspective is comparable. In this respect Ibis would not be able to steer the synthesized applications (third study) used to benchmark the Condor installation in Chapter 2 to completion.

InteGrade

InteGrade [8] is another cycle stealing infrastructure. However, the main difference between InteGrade and systems such as Entropia and Condor is that it performs cycle stealing on dedicated clusters rather than workstations within an institution. For this reason it cannot be adopted for this research. However, there are some interesting features of InteGrade. Firstly, is the intention to provide the ability to use a natural language job definition for submission to InteGrade. Secondly, InteGrade performs predictive modelling to determine when a resource is likely to revoke its voluntary status and therefore chooses resources based upon estimates of availability rather than their current status. Finally, is the ability of resource owners to rigidly define when their resources should not be considered for havesting even when they are idle. This therefore enables InteGrade to be sensitive to the advance reservations made by other users for resources.

MOSIX

MOSIX [171, 172][32] is a system for the management of Linux-based nodes. The main aim of MOSIX is to provide access to multiple resources, but give the impression that only a single node is in use. MOSIX achieves this by providing a runtime environment identical to that provided by standard Unix, and therefore application developers do not need to change their applications in any way, recompile against, or link to any specific libraries in order to run their applications over MOSIX. MOSIX is process-orientated, where a job has one or more processes that can be run on one or more nodes, and migrated if required. The adoption of a single system image means that processes are either transparently migrated from one node to another. When system load is high, a better node becomes available, or in response to user request. The runtime environment also captures any system calls and redirects them back to the host workstation. This also means that any files or software libraries needed by a job are not needed on the remote workstation and can be sent when needed. However, a consequence of this is an increased communication overhead. As MOSIX uses many remote nodes transparently, the system does in effect establish a virtual cluster for the user, which gives the impression of being locally available. The general drawback of MOSIX is that it is bound to Linux and cannot run natively on Windows. However, MOSIX can be run on virtualization tools, but as will be explained in Section 3.2.5, this is not an option on the Cardiff Campus-Grid.

The Globus Toolkit

The Globus Toolkit [50, 173] is a collection of software components designed to support the development of applications for high-performance distributed computing environments, or Grids [26]. The aim of the Globus Toolkit is not to provide application developers with a monolithic system, but rather a collection of standardised and standalone services. Each service provides a basic func-

[32]`http://www.mosix.org` – last accessed June 2009.

tion such as authentication, resource allocation, information, communication, fault detection,[33] and remote data access. The toolkit forms the heart of many approaches, but it is not a silver bullet. In [157] Lu et al. discuss the shortcomings and inefficiencies of Globus. Here, inefficiency comes on many levels, but stem mainly from the GRAM protocol. **(1)** File transfer is inefficient, as jobs cannot share file instances, which are subsequently transferred multiple times. **(2)** Authentication must be performed for each submission and file transfer, even when the same action is repeated multiple times. **(3)** Job descriptions must be modified for use with Globus. Condor-G for example requires a significant amount of extra meta-data in comparison to a standard job submission. This description is then translated into RSL, which regularly shaves off semantic information encapsulated within the submit script. The script may later be translated back into the original description language, and the translation process is not lossless. Attributes such as rank and surprisingly job requirements from Condor submit scripts are lost. **(4)** Once the job is submitted to Globus it may then also be converted into a multi-request RSL, e.g. if in a Condor submit script `queue = 10` (submit 10 instances of this job) this will create 10 Globus jobs rather than a job with an arity of 10. **(5)** As there is no resource broker, there are no means to perform load balancing. **(6)** Submit machines become performance bottlenecks because of the file replication process. **(7)** All clusters are seen as homogeneous. The Globus implementation is ever driving forward, and improving, however, new versions are commonly not backward compatible and require significant changes to the core implementation of an adopting approach. From the perspective of running very domain specific jobs with well defined and rigid requirements of the operating environment, Globus is not a good candidate.

Summary

In this Section a discussion of several distributed computing resource managers and associated systems has been presented. Here, the key attributes which are addressed is the ability to provide access to remote resources, which are predominantly provided on a voluntary basis, but in some cases are also more general Grid-based approaches. The challenge for these approaches is to provide access to resources to the general application, which, typically, is not short running. Therefore, very few of these approaches explicitly cater for short running jobs. In addition, as these system spread across multiple administrative domains, it is increasingly important for them to be able cater for a compiled Matlab approach due to the challenges of license administration. These approaches are summarized in Table 3.3, using the following criteria:

- **C1:** Can be deployed in a Campus-Grid environment
- **C2:** Could support a compiled Matlab application
- **C3:** Transparency of code distribution
- **C4:** Ability/orientation to run short jobs
- **C5:** Granularity

[33]Note: not tolerance

C6: Resource utilization model

C7: Job fault management

C8: Workstation fault management

Table 3.3: A summary of the attributes of each approach, which are important for the application context of this research. Acronyms: Y - Yes, N - No, Y? - Unclear from publications, but is likely, N? - Unclear from publications, ??? not determinable from reviewed literature, ns - not specified, but is unlikely, S - Some, B - Batch, I - Interactive, L - Limited.

Approach	Criteria							
Name	C1	C2	C3	C4	C5	C6	C7	C8
Condor	Y	Y	Poor	N	Any	B	L[34]	N
NetSolve	Y	N[35]	V. Poor	N	Any	B	L	N
Entropia	Y	Y	Poor	S[36]	Coarse	B	L	N
Alchemi	Y	???	Fair	???	Fine	B	N	N
Ninf	N	N[35]	Fair	S	Any	I	N	N
Nimrod	Y	Y?	Good	Good	Coarse	B	S	N
DIANE	Y?	N[37][38]	Y	Good	Coarse	I	S	N
IBIS	N[39]	Y	Y	Fair	Any	B	N	N
InteGrade	Y	Y?	ns	Fair	Coarse	B	N	N
MOSIX	N[40]	N[38]	Excellent	Y	Any	I	???	n/a
Ideal	Y	Y	Y	Good	Coarse	I	Y	Y

It is evident that there is no single approach discussed in this section that meets the minimum criteria needed for running compiled Matlab jobs in a volatile Campus-Grid environment. DIANE, however, is suitable on many fronts, and should an interface and strategy for the execution of compiled Matlab jobs be made available, the only challenge would be how the MCR is made available on the remote nodes.

3.2.2 Campus-Grids

A Campus-Grid enables the collaboration of multiple departments, labs, and centres within an institution. These groups are often in their own administrative domains but can share some common infrastructure [11].

In [174] BLAST (Basic Logic Alignment Search Tool) is run over the UABGrid[41] (a Campus-Grid at the University of Alabama) to execute parallel queries. Here, some resources are provided

[34] A job is rescheduled if it fails due to a monitorable resource event, for example power down.

[35] It is very difficult/impossible to add new functionality and libraries to a NetSolve/Ninf server.

[36] Shortest recommended job length is 5 minutes.

[37] Requires a plug-in for the control of a compiled Matlab session, however, as previously mentioned, none are presented in the current literature.

[38] Could be supported in a batch utilization model.

[39] Ibis uses only dedicated resources such as DAS-3 and relies on the stability this provides.

[40] Cannot run on Windows without virtualization tools which are not supported on the Cardiff Campus-Grid.

[41] http://uabgrid.uab.edu/ – last visited May 2009

by a Condor pool. However, the assumption that software on the nodes is persistently and accurately installed is made. They also assume that the databases are preinstalled on the nodes. In [175] the "my_condor_submit" system is presented, which is essentially a wrapper for the submission of Condor-G jobs using a Storage Resource Broker onto the CamGrid [176] and NGS. Here, the emphasis is on the discovery and utilization of resources, for example dedicated clusters for speeding up applications rather than havesting idle cycles of campus resources. There is however, one substantial Condor pool in place, but little information is given of how this is utilized and managed. The same can be said for Dartmouth's GreenGrid [11]. The Campus-Grids at the University of Bristol (UoBGrid [18] and Oxford University (OxGrid [17]), are also based on Condor, but again provide little information concerning how the challenges of Chapter 2 are or could be handled. One common theme does emerge in that many approaches utilize a Grid Information System such as the Globus MDS [177]. The flaw in such an adoption is that the information in use is still based upon what the workstations or other entities in the Campus-Grid believe to be true about themselves, which is not necessarily up to date or accurate.

Harmony [16] is a system used to harvest idle desktop resources in an enterprise environment, and is perhaps one of the most advanced Campus-Grid-like system reported in the literature. Its basis in the enterprise realm incites the requirement for guarantees related to performance and qualities of service, and therefore SLAs are introduced. The details of what constitutes the SLA are unclear, and therefore the aspects of the mechanisms employed to ensure adherence are also unclear. Essentially, the approach is to launch VMs onto the Windows workstations, in a similar manner to Condor's VM universe. Predictive models are then employed to identify when a state of unavailability may be enterred, the VM can then be simply paused and migrated. Through this approach many of the problems outlined in Chapter 2 can be avoided, however, employing this approach on the Cardiff Campus-Grid is, as will be discussed in Section 3.2.5, not an option.

3.2.3 Fault Tolerance

It is not possible for an approach to be completely fault free given the context and volatility of resources in a Campus-Grid domain. Typically, there are four main strategies employed for fault tolerance and management which are discussed below.

Workstation monitoring – Here workstations are monitored to detect any changes in availability. The most simple monitoring mechanism is the heart-beat monitor, where a service ping or equivalent is sent to verify that the worker daemon is still residing and operational at its known endpoint. In [46] progress certificates (checkpoints) are used to indicate to a central server that a worker daemon is still active. Should a workstation not check in its checkpoint within a certain period of time it is assumed that this workstation has failed or revoked it voluntary status. Lost work units are then restarted from the last checkpoint held by the server. Other examples include the Condor shadow process which monitors workstation activity to detect periods of idleness. Many other approaches such as NetSolve, Entropia and InteGrade also employ similar strategies. Should a workstation fail any of these tests the job is rescheduled. All distributed systems have to perform this process to some extent. Examples of available software include: Hawkeye [70], Ganglia [123], NWS [178] and GridRM [179]. The challenge with some of these approaches is in their setup and administration. For example Hawkeye as a part of the Condor family typically runs under Linux, and has stability issues under Windows. Ganglia must be installed on all nodes creating similar issues to the installation of the MCR. NWS is also predominately Linux-based and mainly monitors network stability rather than workstation stability. GridRM on the other hand, is quite powerful as it is capable of combining and utilizing the information of many different monitoring systems. It can also be hot deployed, which makes it extremely useful. However, many of these tools deal with monitoring at the hardware and process level, rather than at the software level. Ibis' [168] Java-GAT [170] can disseminate and interpret software-related failures, but it relies on the information provided at the resource management layer, and assumes that failures will occur in a timely manner, neither of which can be guaranteed. In this sense monitoring needs to be performed at the job level rather than in the resource fabrics.

Checkpointing is the saving of program state so that a job can be restarted without loosing previous computational effort or time. Checkpointing essentially reduces the probability of wasting computational time and resources as a consequence of failure or the revocation of voluntary status. Checkpointing can also be used as a tool for migrating one or more processes to another workstation. However, it is most useful for long running jobs and therefore is not useful for this research.

Predictive modelling – Here the aim is to predict either changes in workstation state which affect the ability of a job to run to completion, the time required for a job to complete, or provide a probabilistic indicator of job/workstation failure. The predictive models presented in the literature are commonly based upon key assumptions of the state of the resource pool and how it changes over time. The key factor in many approaches is the process by which workstations join and leave the network. Deng et al. [180] described the process through which resources enter and leave the system as an independent stochastic process. In contrast, Naik et al. [16] modelled availability as a Markov chain to predict changes in state, and therefore availability. The InteGrade system [8] collects and analyses usage patterns to construct behavioural categories of use. At scheduling time this information is used to try and avoid checkpointing situations. Mutka [5] also noted that resource managers can predict workstation availability with considerable accuracy. Whilst the consideration of the process by which workstations change is useful, it does not capture the bigger picture of workstation reliability. Here, actions such as maintenance, failures and administration by technicians are perhaps more detrimental to performance than the transience of workstation availability.

Persistent Job Storage – Here job descriptions are kept in persistent storage so that if head node failure occurs or other systems fail, the application or workflow execution can continue rather than be restarted. Kim et al. highlighted in [181] that robustness and reliability come from the retention of state of all jobs running on potentially unreliable resources. Traditionally, a centralised server holds this state in a relational database or equivalent. However, should this point fail, not only can new jobs not be entered into the system, but those that are currently being executed are susceptible to loss.

Self-*: specifically self-healing autonomic systems are in many scenarios the key to the challenge of workstation reliability. It would also be quite straightforward to fix many of the issues presented in Chapter 2, as the pilot part of the job setup quickly identified whether a workstation is capable of executing the job or not. An XOR of available files can then be performed to identify those that are missing and checksum analysis could identify any that are corrupt. Then a reinstallation process or simple file transfer could be performed. However, unfortunately it is not that simple, as administrative access is required to modify the configuration of a workstation. Typically in autonomic systems, administrative rights are given to the manager application, but the political barriers that would have to be crossed in a University context to reach such a point are significant. This is exacerbated, as scenarios such as those presented in Chapter 2 are not commonly published or appreciatively received.

It is not possible for an approach to be completely fault free, given the context and volatility of jobs in a Campus-Grid domain. For this reason much of the work in this area concentrates on prediction of state changes and the reaction to changes in worker state, but do not necessarily consider inaccuracies in the descriptive meta-data of a resource. The core goal of the former is to try and prevent job failure as a consequence of preemption and eviction or that a job needs to be checkpointed. Other approaches attempt to determine when checkpointing is most effective based upon similar prediction models. The latter defines what should happen when such events arise, but these measures are not as necessary for the execution of short jobs, as the overhead of checkpointing and migrating of a job may be less effective than simply restarting it.

There are two primary types of fault tolerance discussed in this Section. Firstly, proactive fault tolerance, which tries to minimise loss. This occurs either as time lost when a job rescheduling scenario occurs (checkpointing), or in the reduction of the risk/probability that a negative runtime state change occurs. Secondly, reactive fault tolerance, which defines the means with which to identify this outcome (monitoring) and determines what happens should this situation arise (recovery). Yet, these solutions often blur the causative entity responsible for this eventuality to arise, which ultimately yields an incomplete solution. Not all failures are caused by the job itself or changes in runtime state, but can also occur from other less obvious background and independent processes. The monitoring solutions are generic and hardware-orientated and are thus not able to detect the provenance of all failures. Similarly, not all job abortions are a consequence of preemption or changes in state. In this sense there is a requirement for fault management rather than fault tolerance, which can be a proactive process.

3.2.4 Application Level and Campus-Grid Scheduling

AppLeS

The AppLeS project [119, 182, 183] focuses on the design and development of application-level schedulers for distributed applications, specifically parameter sweeps. There are three aims of AppLeS: **(1)** to enable the user to tune the scheduling strategy to meet their requirements, **(2)** for the scheduling algorithm to adapt and tune its own parameters in order to minimize an application's makespan, and (3) to perform automatic and dynamic scheduling algorithm selection without user intervention. The scheduling model is event-driven and therefore the scheduling plan can be tuned autonomously to reflect changes in context and resource availability. The AppLeS approach does also have one major assumption: it assumes that all knowledge of the resources available is present and correct. It is also specifically orientated for parameter sweeps and therefore cannot be used for other job models.

MARS

The Michigan Advanced Resource Scheduler (MARS) [184] is a meta-scheduler for distributed resources on Campus-Grids. MARS aims to offer efficient scheduling of tasks within a Grid environment, as resource managers such as Globus provide utilities such as resource discovery, security and authentication, but not resource selection, load balancing and fault tolerance. Therefore meta-schedulers in general are fundamentally different to batch resource-level schedulers such as Condor in their capabilities. As the intended application scope of MARS is widespread it allows different scheduling algorithms to be incorporated based upon user preferences and application requirements.

MARS maintains a resources table, which is used to identify viable resources for job execution and implemented using a series of Globus components and Condor's ClassAd representation. Resource forecasting is also performed to identify resource-level metrics such as CPU utilization and queue length. These metrics can then be used to rank candidate resources for job submission. However, the resources table does not consider any metrics which are related to the volatility of a resource. In this sense the main aim of MARS is to move toward an optimal schedule for a set of jobs, but a core resource attribute: reliability is not included in the resource model, which, given the potential for volatile resources in a Campus-Grid context is a significant factor that is missing from the scheduling strategy.

3.2.5 Resource Virtualization

Resource virtualization is steadily becoming a widely adopted option for scientific analysis and Grid Computing in general, and there are two core reasons for this. Firstly, it enables a fully customizable operating environment, but in a secure and sand boxed way. This enables the consumer of the resources to get exactly what they require and have administrative access to the resource [185]. On the other hand, the resource owner is protected against any malicious actions performed by the consumer, as the virtual machine acts as a sand box and therefore protects the physical resource [186,187]. Secondly, it enables any resource regardless of operating system to be used, and therefore removes any issues of resource and, more specifically, heterogeneity with respect to the operating system and application environment.

On the outset the use of virtual machines (VMs) looks (and is) extremely attractive. In [71] Barham et al. identify that the direct effect of resource virtualization upon the performance of an application is minimal; around 5% slowdown, which is in fact very good. The challenge in this domain is not in developing resource containers to host the VM, this has already been achieved (see: VMWare [72], Virtual Box [188] and Xen [71]). Instead, the challenge is in the creation and distribution of the VMs themselves. In many approaches (e.g. [185, 187, 189, 190]) it is assumed that a generic all purpose VM already exists and can be simply distributed onto physical resources, where any additional customization can be performed at runtime. Such an approach increases the physical size of the VM and therefore increases the cost of transferring it. This would appear to be a limiting factor in the paradigm, and it was, but there are some operating system choices that can be made to reduce this problem significantly. One example here is Damn Small Linux [191], which offers a Linux implementation reportedly for as little as 6Mb. In a similar fashion regular Linux implementations can be dramatically reduced to only what is needed for the application to run. This is the approach taken by Nishimura et al. [192], who create virtualized clusters on the fly in a rapid fashion (as little as 39 seconds for a VM image). Note, however, that this approach was not applied to libraries of a significant size such as Matlab, and it is therefore not clear how the addition of a 500MB (compiled Matlab, the normal Matlab installation is in the order of 1.5GB) library would affect their reported performance. This is also true for the paradigm in general.

This paradigm is very well suited to high throughput computing, and consequently the VM universe was added to Condor. There are many research papers published that report the issues and effectiveness of this universe. However, the big challenge and limiting factor of this approach is that for Condor-based approaches each VM needs to have a reachable endpoint to enable the relevant Condor daemons to intercommunicate. There are two options for this, however: note that some institutions, Cardiff University for example, support neither and can therefore not use this universe at all. The first option is to register the VM on the network as a new "workstation". The challenge here is that in order to deploy many instances of a VM, the management software must be able to instruct the VM which MAC address it must have, but this is quite simple. The bigger challenge is deciding how many to register on the network and also the means to keep track of which MAC addresses are currently in use, which is further complicated when multiple submission points exist. The second option is to deploy the VM as an "application" and NAT a specific port range from the host machine to the VM. This is a perfectly adequate solution, but the consequence of this action is that the Network Management teams cannot differentiate between an operation performed by a VM, and one performed by the host workstation. This is specifically important for the detection of malicious software or viruses. For this reason, VMs must typically be registered on the network.

The concluding remarks for this paradigm in regard to the challenges outlined in Sections 1.3 and 2, is that it would remove many of the challenges surrounding the inaccuracies of ClassAds and would also enable a fully customizable workstation image. However, there are currently too many questions surrounding how the Campus-Grid infrastructure would respond to the presence of a VM and what the impact upon job execution times would be as a consequence of transferring a VM to a Campus-Grid workstation. It is very likely that a solution to the first issue could be found, so it can be ignored presently for this discussion. However, the issue of overheads with respect to the transfer of the VM to the remote workstation would still remain. Even when compressed and with a minimal operating system installation, the transfer costs for Matlab jobs would be significant. Optimization on the VM image would be the key to a successful implementation in this area. However, performing this optimization in a way that does not affect the performance and heterogeneity of the jobs would be challenging and may in fact not yield any improvements. When physical resources are in use, the impact of software dependencies to job submission can be largely avoided through the use of network management tools. These can install the necessary software on the workstations and therefore make it available to Campus-Grid users. This is also currently adopted by many institutions, including Cardiff University, and is arguably the better approach, even when the accuracy (or lack) of these procedures is considered.

3.3 Summary and Outstanding Issues

In this Chapter a review of the current research in the areas of distributed and parallel Matlab (Section 3.1.1) and image processing (Section 3.1.2) has been conducted. Here, research is based upon either making toolkits available in a distributed or parallel context, making multiple resources available to an application developer transparently or handling licensing issues. All approaches in these two areas are based on scenarios where no consideration is needed of the resource stability and reliability. Therefore, all would need extension in order to run in a volatile and opportunistic environment. Two approaches (JavaPorts and ProteomeGRID) stood out as the best candidates, but both would require significant changes in their conceptual and architectural models to be able to handle the requirements the necessary changes would place upon them. The review then focused on approaches that are more general. Here, DIANE and to some extent Ibis provide the most promising results and conceptual models. However, both lack the necessary fault management capabilities to enable the steering of even a simple application to completion.

In summary the following areas are not adequately addressed in the literature:

1. Interactive control of compiled Matlab applications. JMatLink [152] is the forerunner in this area for licensed Matlab, but it is not reported whether it can be used in a compiled context. It is also only able to control single instances and not multiple instances. Interactive use is needed to enable efficient resource utilization for short running jobs.

2. The ability to compile a general Matlab application instance such that it can cater for heterogeneous applications without recompilation. Current approaches in Matlab compilation typically compile small and specific applications. Then should a change in focus be required simply compile again. Such a compilation strategy prevents the adoption of interactive use and dramatically reduces reusability.

3. The necessary techniques to manage faults and inaccuracies in a workstation's descriptive meta-data. Such scenarios often occur in a dynamic and rapidly changing environment, which was characterized in Section 2.3, by a workstations's Matlab ability changing from consistently working to consistently not working and back again within 18 hours. The means to proactively respond to the associated consequences of such actions are also needed.

4. Description languages are in many cases too general, with the consequence that representing a simple sequence of image processing commands requires an entire XML schema to be defined. Applications such as Matlab already have their own functional languages that can contain all of the necessary information needed to describe a job. Moving away from such languages creates new problems and challenges unnecessarily. The incentives behind many of the general term languages are to provide language independence. However, in consequence they often introduce definition problems, mapping errors, and lack the constraints of the inherent domain-specific semantics and syntax. In other words, by attempting to hide complexity they incite a lack of understanding and precision [193]. They also increase overhead through the additional effort needed in parsing and conceptual mapping.

The prototype that has been developed for the research conducted in this thesis addresses each of these areas. Interactive control of compiled applications is made made possible in two ways. Firstly, through the creation and compilation of a general compiled Matlab applications, which is performed using an autonomic Matlab compiler (Section 5.2.1). This also addresses the second area. Secondly, through the insertion of a worker daemon into the compiled Matlab application that has direct access to the Matlab engine (Section 5.2). The management of inaccurate resource meta-data is approached through the establishment of an adaptive performance model and the differentiation between resource allocation and job scheduling. This will be introduced in Chapter 4. The final area of work description is straightforward. Due to the insertion of a worker daemon directly within the Matlab session, and the differentation of resource allocation and job scheduling, units of work can be described using regular Matlab code.

Chapter 4

Methodology

The approaches evaluated in Chapter 3 lacked the necessary levels of fault tolerance and management to enable the successful steering of jobs with strong requirements with respect to their operating enviroment. In addition, many lacked the ability to efficiently run short jobs which constitute of and cater for user defined Matlab code in a license free context. The approaches evaluated in Section 3.2 are also predominately designed with different intentions and trade offs in mind to the applications considered in this research (Sections 1.2 and 1.3.9). These differences are with respect to application generality, neutrality and orientated toward longer job execution times, where overheads are much easier to control. This has meant that in order to improve performance, feasibility and stability these approaches need additional help to become usable. In contrast, many of the approaches evaluated in Section 3.1 are very focused on this particular application type, but ironically are not general enough and lack the abilities to answer the challenges introduced in Chapters 1 and 2.

A traditional approach to utilizing Campus-Grids is to supply a set of input data and one or more executables to perform some analysis on one or more workstations, as illustrated in Section 3.2.2. The potential consequences and some short comings of a traditional approach were highlighted in Chapter 2. The approach introduced in this Chapter, however, remodels these ideas, through the deployment of infinitely running, but steerable (pilot) jobs. This execution strategy allows a workstation to be used for as long as it is available, rather than for the, potentially, short life of a job. Therefore, a workstation can be used for: **(1)** a much greater period of continuous time, and **(2)** more efficiently. It also means that multiple and potentially disparate jobs can be executed with only one initialization phase, which for short jobs relates to the opportunity to significantly reduce overheads and therefore turnaround time. This is a significant improvement over a traditional approach where a new working session is required for every job. During each working session a job must go through the scheduling process, retransfer any data (even if the same workstation and data are used) and any initialization procedures. As, Chapter 2 demonstrated this can be a significant period of time. The potential improvement this enables was demonstrated by both ProteomeGRID and DIANE.

A pilot job scheme on the other hand enables the proactive use of reactive fault tolerance measures. At the heart of this is one core assumption: a pilot job will **not** fail unless the workstation is itself the cause of the failure. Failure in this sense is every eventuality other than observable success. For example, a job that has runaway is deemed a failure. Such a job is unequivocally within a long and drawn out state transition, where there are no attributes that can be used to indicate success. In addition, it cannot (quickly) reach a specific state. Whether this state (should it ever be reached) is indicative of success or failure is inconsequential. The proactive measure is that the pilot job's primary purpose is to determine the capability of its host workstation. Once this has been achieved it will then serve as a platform for the execution of a user's jobs. The reactive measure is the avoidance of workstations that have not been able to consistently bring the pilot job to a state that is indicative of success within the constraints of the models presented in this Chapter. These workstations, therefore, are avoided because they have been unable to prove their capability and utility. This is one of the core differences to all other approaches in the last Chapter, as none provided any means of fault management such as this. Instead, they either used dedicated resources, which are not prone to such errors or assumed that are uncommon/non-existent. This was even the case for other approaches working with Condor pools. However, typically the Condor pools were not of the magnitude as that used throughout this thesis.

This approach produces a novel, autonomous, but transient distributed environment for scientific analysis. Despite an obvious leaning toward image processing applications, any Matlab-based application can be used and Matlab itself can be replaced. This system is layered on top of Condor, but the philosophy behind the approach is in no means Condor specific and can be applied to any other Campus-Grid architecture or middleware. Therefore, Condor is interchangeable with other resource managers with little conceptual modification. A discussion of how this can be achieved is presented in Section 5.7.1. The novelty in this approach stems from the way in which the application environment in controlled and interacted with and built (this will be presented in Chapter 5) and the strategies for fault and resource management.

4.1 Abstract Model

The abstract model that has been developed to answer the challenges of the previous Chapters and to facilitate the application domain has four main areas: (1) the separation of resource allocation and job scheduling, (2) a new resource management model, (3) a modified, but isolated job model, and (4) a modified users' model.

In Section 4.2 the motives behind the separation of resource allocation and job scheduling are presented. In doing this three core abilities are extracted: (1) the ability to isolate the control of job management, (2) the ability to enable workstations to be used more productively, and (3) to provide the opportunity to control more explicitly what resources of those available are used, and how.

In Section 4.3 the modifications to the traditional Job Model are presented. The traditional models are abstracted and replaced by a split model of two tiers. The first, is an infinitely running pilot job that relates to a general steerable worker daemon (Section 4.3.1). The second, is a Task

Model (Section 4.3.2), where a Task is a control sequence that represents a single unit of work, which is passed to worker daemons for evaluation.

In Sections 4.4, 4.5, 4.6, 4.7 and 4.8 the new resource management model is presented. These models describe the core of the approach by defining how resources are acquired and utilized. They are also responsible for the performance of an implemented solution, as they provide the means for fault tolerance, management and prevention. Finally, they detail the benefits which are now achievable. In Section 4.4 the methodology for the acquisition of workstations is presented, and how the volatility of the workstations in use is addressed. In Section 4.5 the methodology for encapsulating the user's requirements is described. Users will not have to specify their resource requirements, this model will represent their requirements on their behalf. In Section 4.6 the method for allocating resources is described. In Section 4.7 the approach for scheduling is described and how other scheduling algorithms can be incorporated into the model. In Section 4.8 the means to provide runtime fault tolerance is presented.

In Section 4.9 the discussion of how users can interact with a system built using these models is presented. It also contains the definitions of granularity used throughout this research.

4.2 Separation of Resource Allocation and Job Scheduling

In Chapter 2 two final conclusions were drawn: **(1)** the need to clarify a workstation's abilities and state prior to job execution, and **(2)** the need to instigate a higher level of control over what workstations do. Only once this has been achieved can other research questions be attempted. In essence, these two requirements are similar, but have different motives. Both relate to the first research question presented in Section 1.5: What measures are needed to retain (or achieve) stability in a volatile and adaptive environment?

Verifying workstation state and ability is trivial when using superficial test jobs, like those used in Chapter 2. In a realistic job space such a strategy cannot work, as any previous success cannot guarantee future success. Firstly, a job can runaway or fail, even immediately after a successful job. Secondly, in other situations, testing strategies cannot always determine if a workstation is usable or not, leaving the workstation in a hung state until it is interrupted. Each of these scenarios was demonstrated in the empirical studies undertaken in Chapter 2. Therefore, when a workstation is identified as capable, it is essential that it is retained for future jobs to prevent any changes in the state of the operational environment to occur. Currently, it is not possible to instantiate an interactive session with a workstation when a job succeeds. Instead, the best case that many resource management middlewares can offer is back-to-back scheduling.[1] However, the consequences of such an approach is that a new working session is typically created for each job. There are three points of concern with this approach: **(1)** there is no guarantee of the workstation remaining available to the user. The resource management software may identify another user with a higher (discernable)

[1]Back-to-back scheduling is when a resource is allocated to a scheduler. Then the scheduler can schedule jobs one after the other to this resource, without needing to have the resource reallocated.

priority or one whose requirements specifically identify this workstation. In this case the workstation will be reallocated. **(2)** The overhead involved in this process, a significantly limiting factor on performance and throughput, is repeated for every job, regardless of length and affinity.[2] Short jobs only exacerbate the detrimental effect on performance this evokes. **(3)** If a workstation does not function correctly, it can, and as Chapter 2 demonstrated often does, become a black hole and devour jobs greatly distorting the job space. In these last two points, resources are being unnecessarily wasted and no answers in the literature for the last point were presented.

To combat this, a new approach is needed. Only by isolating (and therefore separating) the control mechanisms which differentiate between scheduling and resource allocation can improvement be achieved. This separation introduces a conceptual limitation of the terminology in use. Traditionally, allocation and scheduling go hand-in-hand to produce an abstract plan for the execution of a job space. By removing the ability of the resource manager to schedule the user's jobs, a third term must be introduced. This term is needed to differentiate between a resource manager providing a resource (running a daemon) and that resource being used to evaluate a user's jobs. When a resource is provided by the resource manager to run a pilot daemon job it will be termed as **acquisition**. This process is described by the **acquisition model** (Section 4.4). When an acquired resource successfully initializes a daemon and is therefore capable of executing a user's jobs it will be **allocated** to a user. This will be performed by the **allocation model** (Section 4.6) in response to their resource requirements.

The hypothesis behind this separation is: if the ability to designate (i.e. schedule) work units is isolated from the local resource manager, then performance, reliability and stability can be increased. In addition, resource reusability, and therefore efficiency, is greatly increased. The control mechanisms are a cause of, and a partial solution to, the poor performance of the system. By establishing a vertical model, and building upon their current abilities, many of the challenges of job administration can be answered. For the user, this means improvements in job turnaround, and therefore research progress. For the resource administrator it means that improvements in performance can be achieved without increasing the number of resources made available. For the computer scientist, new areas of research are opened, because limitations on performance that obstruct other interesting models from being conceived and researched have been removed.

4.3 Job Model

The job model consists of two semi-independent sub-models: **(1)** the Daemon Model (Section 4.3.1), which represents a steerable worker daemon residing on a Campus-Grid workstation and is capable of performing multiple users' Tasks, and **(2)** the Task Model (Section 4.3.2), which encapsulate all the necessary information to describe a single unit of work that a user wishes to perform. To avoid confusion, from now on, **Job** and **Task** will not be used synonymously. A **Job** refers to an instance of a worker daemon (a pilot job) to execute upon a Campus-Grid workstation. A **Task** on the other

[2]See definition in Section 4.7.2

hand refers to a user's control sequence, which acts upon one or more data sets to perform some analysis.

4.3.1 Daemon Model

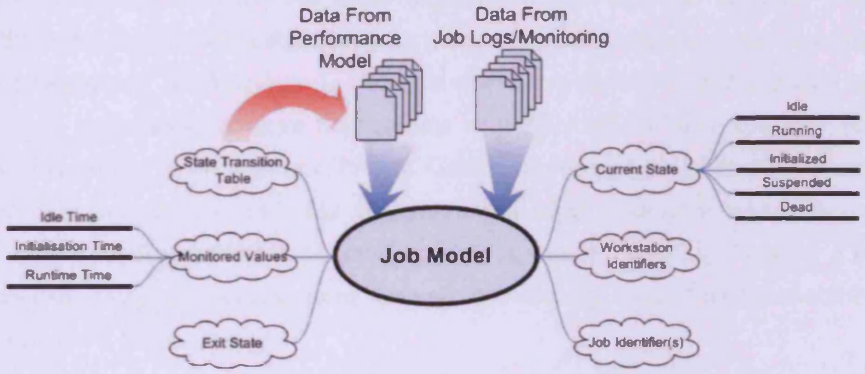
A daemon is the primary component of the pilot job. It initializes to stress test the workstation and demonstrate that the workstation is capable of executing the user's Tasks. It attempts to run infinitely, only terminating either when forced to by the Campus-Grid resource manager or upon explicit instruction from a system controller. A daemon is fully steerable, which means that it has no predefined Tasks or role before it initializes. All Tasks that a daemon performs are given to it during runtime. In addition, it is not application specific, but general, which means that it serves heterogeneous application contexts.

The crucial difference between this model and the equivalent model of other approaches is the integration into an application environment, such as Matlab, R, Octave, Mathematica, LabVIEW, etc. The two closest approaches in the literature (ProteomeGRID and DIANE) both sit outside of the application environment. The steerable nature of this model means that it is inherently an interactive service. To facilitate interactive use, a daemon does not shut the application environment down, but keeps it alive until it is instructed otherwise or the host workstation is revoked. In practice this can be extremely challenging to implement it is also not clear whether the approach untaken in DIANE actually manages to keep the engine alive or if it must reinvoke it for each request. The advantage of such an approach is that an application environment would be infinitely available, if the workstation was not reclaimed by its owner, or if the daemon was not instructed to shutdown. The daemons' aspiration for self preservation completely rewrites the assumptions of regular job models. A job is no longer a finite entity executing on a workstation, but a perpetual entity that can serve multiple, independent users without (job) resubmission. This is the case because one daemon instance can evaluate multiple distinct Tasks with only one initialisation phase. These two abilities reduce the impact on performance of unpredictable queueing times, which in a Campus-Grid context are naturally hereditary. It also minimizes setup time, and, ultimately yields higher Task throughput. In addition, all external libraries remain active in memory, further improving performance. Furthermore, because the daemon has direct access to the processing engine there is no need to use external term languages for the expression and description of work units. Instead, the application programming language can be used as it is perfectly suited for this purpose. This also means that in order to change the application environment only the means to control the application engine requires modification, as the term language is already inherently linked to the user's code.

This approach permits a far more powerful paradigm than a batch processing equivalent. Firstly, there is only one initialisation penalty per workstation rather than per Task. This penalty includes the time to create and schedule a job, transfer data to the workstation, job pre-staging, and startup overheads. This of course assumes that the job is successful and does not fail or runaway. Once a daemon has started, it is capable of executing a user's Tasks, and can be allocated to a scheduler by the allocation model (Section 4.6). Secondly, the workstation is used for the time that it is available, not just the Task's lifetime. Thirdly, because one session is used for multiple Tasks, future Tasks will run faster.[3] Finally, system stability is created and fault tolerance is provided transparently. Stability is provided by the reduced need to consider failure at the scheduling level. This is because most workstation failures will occur prior to a scheduler becoming aware of a workstation. Those that do not, are significantly easier to accommodate and rectify. They are either a consequence of a Campus-Grid event (such as suspension or revocation), programmer error or some other runtime error. Similarly, workstations are in use for the duration of their availability, therefore only revocation and intermittent failure can disrupt the system. Fault tolerance is provided through the pilot scheme. The user does not have to be concerned with how this approach works or what constraints are implemented to identify either success or failure. As a result, the complexity of Task management is reduced. In addition, the approach can guarantee that any Task related error can be differentiated from a workstation related error. This is an important advantage, as without this ability, deep and potentially tedious and time consuming, retrospective analysis is required due to the indistinct and varied nature that failures are identified (see Chapter 2). This final ability is another core difference to the work documented in the literature, as none of the approaches reviewed could make this differentiation so easily.

A distinct advantage of the external control of jobs and the differentiation of acquisition, allocation and scheduling, is that the job model can represent a new, and very useful state: *Initialized*. When a job reaches this state, it signifies that it has achieved daemon initialization, can be steered, and therefore be given Tasks to perform. It also demonstrates the stability of the workstation as any contributing factors to jobs failing or running away have not prevented the daemon from initializing. Figure 4.1 depicts the job model. In this Figure (and any following Figures in this Chapter) blue arrows denote input and red arrows denote output.

Figure 4.1: Job Model



The initialization of daemons that are directly available for interaction, gives a new view of the resources in use. Instead of workstations being invoked in an ad hoc manner for each Task, a resource is acquired, then held on to until it is no longer needed or available. This can be modelled

[3]The first execution of any software entity is the slowest because the operating system must load the required binaries into memory or cache. Future calls of these binaries are faster because they are already present in cache and do not need to be loaded into memory again.

as a Virtual Cluster of *dedicated* resources, which is built at runtime for the duration of an application's execution, but is a direct consequence of its modelled requirements in relation to workstation availability (see: Section 4.5). The workstations are likely to be in several different sub-domains of the network, but for a scheduler they appear as a single cohesive unit.

The concept of infinitely running jobs does not necessarily sit well with a resource provider or administrator. In a sense it breaks the unwritten rules of Campus-Grid and opportunistic computing etiquette.[4] When new Tasks are not put into the system, a daemon could be running, but at the same time be idle. In this state, the workstation is potentially blocked, preventing other users from using it. Naturally, an autonomic manager could simply terminate any daemon in this state, but such scenarios present interesting questions. For example, how long should a manager wait before terminating a daemon in this state, given that it does not know when new users or Tasks will arrive? What if the previous user is waiting for a synchronization point, or is currently calculating previous results before new Tasks can be added to the system. The benefit of keeping the daemon alive as opposed to terminating it is obvious in a clairvoyant system or when resources can be reserved in advance. Neither of these privileges are in place in the current context, but keeping a daemon alive, even when there is no obvious gain, does have its benefits. The environment in which a daemon exists is unpredictable and defined by external, and uncontrollable, factors. Namely, the retraction of the volunteered resources, either through workstation reclamation, shutdown or other miscellaneous failures.

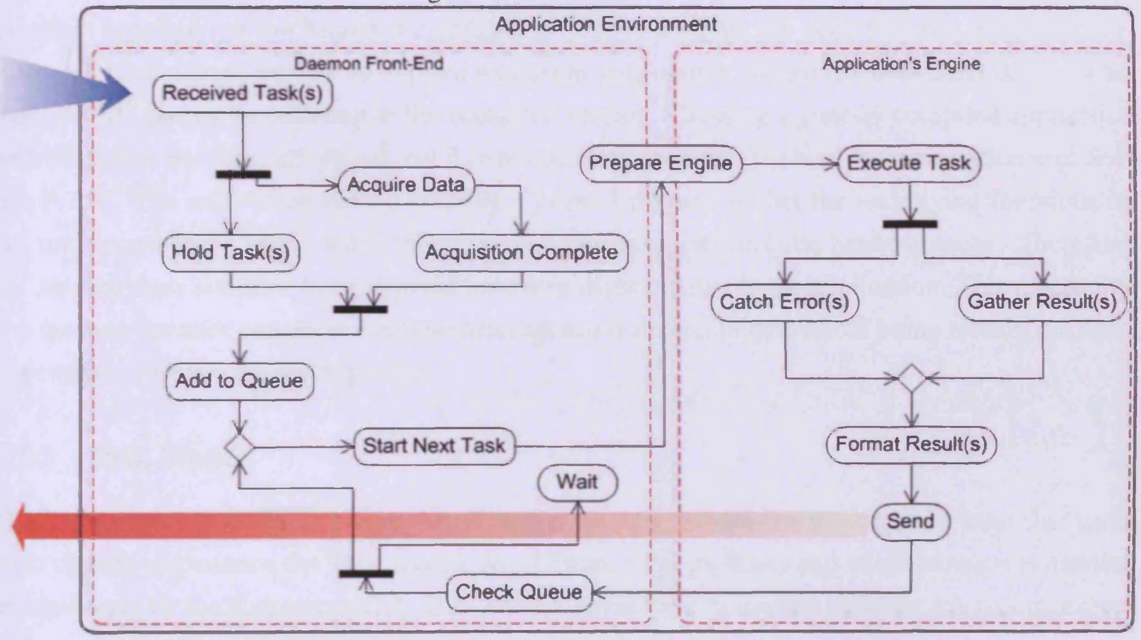
Having spare capacity provides on three fronts. Firstly, when fault management is required to remedy the consequences of resource transience. In this case, when a workstation is lost, all Tasks should be rescheduled to another daemon. If some spare capacity (idle daemons) is available, the cost of rescheduling a Task set is reduced, in comparison to initialising a new daemon instance. Secondly, in the context of a Campus-Grid, workstation heterogeneity is an inherent property and therefore load balancing strategies are often required to take this into account. Here, daemons residing on the most capable workstations, which complete Tasks quicker, can consume work from daemons on less capable workstations to balance workload. Finally, spare capacity is inseparable from the need to accommodate large fluctuations in work load. Whilst this may seem wasteful of resources, this ability is the raison d'être of Grid Computing [194]. In other words, the value of the system is not necessarily in being able to simply supply compute resources over time, but to have that supply readily available on short notice. In this last case, by keeping a few daemons running, users can always access resources with no forewarning, even if resource contention within the Campus-Grid as a whole is high.

The daemon model is presented in Figure 4.2. On the outset it looks just like any other approach. Condor, Entropia, NetSolve as well as many others use a daemon model for job execution, however, the daemon in these approaches is a general worker daemon which simply runs an executable within

[4]Note that it could also potentially raise security issues, as intrinsically it is nothing more than a service permanently listening to a specific port from where it receives instructions. Whilst it is unlikely that this could enable malicious activity, it must nevertheless be noted. However, the application of security mechanisms and what would be required in a production system is beyond the scope of this thesis.

some potentially virtualized or sandboxed environment for one working session. Approaches such as ProteomeGRID and DIANE have a less general model but nonetheless they reside outside of the application environment and therefore cannot guarantee either that previously used functionality is usable again without retransmission (ProteomeGRID) or that the system remains available when performing other processes such as staging data (DIANE). The difference in the approach presented in this Chapter is the direct integration of the daemon within the application environment. This allows it to the keep the application's engine running, which facilitates interactive use and provides a platform to improve performance. In addition, if a specific data set is in needed for every Task, it may need to be reloaded for every request, which is not the case if the application environment is kept alive, as the data set remains in memory.

Figure 4.2: Daemon Model



An additional measure to accommodate shorter Tasks is the daemon's Task queue. Creating a queue enables a daemon to transfer any data needed for pending Tasks while another Task is executing. This is not a novel concept, as it stems from paired gang scheduling [195], also known as co-allocation, but it provides schedulers with the ability to perform dynamic Task clustering (Section 4.7.1), and thus reduce the impact of data transfers and other inhibitors of performance. The advantage of a Task queue is that a scheduler has all the advantages of Task clustering, but with the additional advantage of changing its plans, should it need to. The principles of the queue are simple. When a Task is received, its data requirements are analyzed to ascertain what data is already available (transferred in the past) and what is not, therefore requiring a new transfer session. If the latter is true, the Task is placed into a held state, until all data is available.[5] At this point, or if

[5]No data is transferred twice, so even if a Task has two data requirements and one is already available, only the second will be transferred.

the data was already available, it is placed into the queue, ready for execution. There are questions relating to the size of a daemon's queue. For example: what is the optimal size of a daemon queue? When is a daemon's queue too large? What performance impacts does queue size have? These questions are, in part, application dependent, but also dependent on the assumptions and objectives of the scheduler. This discussion is presented in Section 4.7 (Scheduling Model) and an application specific discussion is presented in Section 6.10.4. Also note that the other approaches reviewed in Chapter 3 do not make this consideration.

In order to create a general execution environment a daemon should not be restricted to only one application or context. The difficulty in achieving this state is dependent upon the execution environment. For example applications built with tools such as R can achieve this state much easier than for example an application built using Matlab. Firstly, R is license free and its method of including functionality is similar to a unix-like installation. On the other hand, Matlab is licensed, and it is therefore beneficial to use Matlab's compilation tools to compile the code to make it license free. However, compilation evokes an explicit exclusion mechanism, where the user must decide what functionality should be included in the compiled version. Creating a general compiled application with Matlab is not straightforward, but it is nonetheless possible (for this implementation see: Section 5.2.1). The motivation behind creating a general daemon is that the underlying functionality can remain relatively static, while the physical application can be quite heterogeneous. Therefore, the daemon does not need to be updated for every slight change in the application. This means that one daemon instance can serve multiple heterogeneous applications without being rebuilt, but more importantly without job resubmission.

4.3.2 Task Model

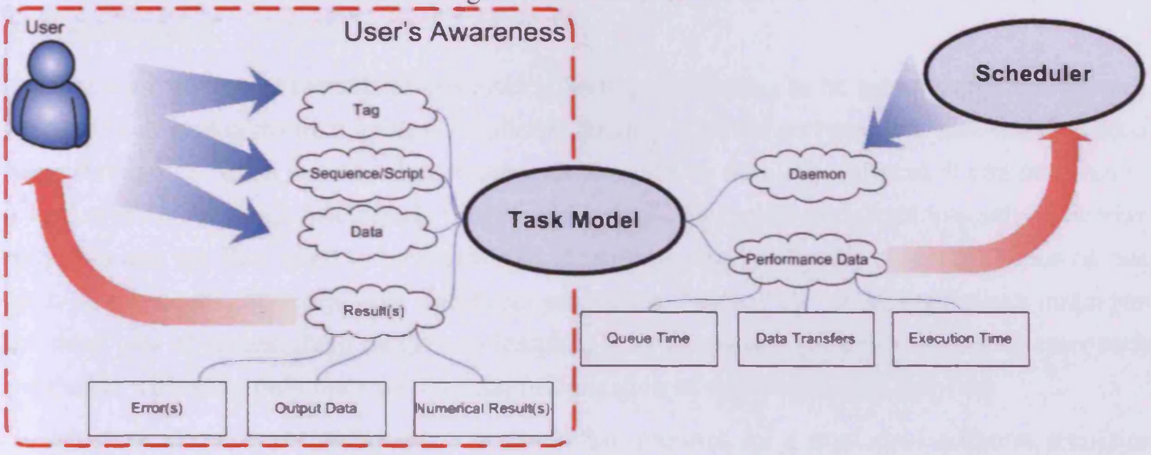
The Task model is much less complex than the Daemon Model for the simple reason that users must directly experience the Task model. In addition, all complexity and administration is handled transparently by the daemon model. They should never have to interact with the Job model, except during the selection of daemon functionality (see Section 5.2.1).

The information that a Task encapsulates is: **(1)** the function, function sequence or script that is to be evaluated, **(2)** the data to be processed, and **(3)** an identifier for the result (optional).[6] This is a very simple approach to Task definition, however, with this simplicity comes substantial descriptive power. Descriptive prowess is captured by the existing programming language in use, such as Matlab's m-code. It also removes the need for more heavyweight description languages such as the OGF's JSDL, Condor's ClassAds, Globus' RSL, the GLUE schema etc., which are themselves both ineffective with respect to the description required, and inefficient in that they introduce the need for parsing between a general language and one which may be very precise and contain many semantic synonyms, which cannot be accurately represented by a general specification. This approach also enables native programming approaches such as RMI and Java Serialization to be introduced. These three pieces of information are sufficient to create an entity that can be scheduled to a workstation

[6]An identifier can also be provided by an Application Manager (see: Section 4.9) and is therefore optional.

where it is (later) evaluated to create a useable result. A usable result is one that any application, which may require interactive use of the resources provided by the Campus-Grid, can use explicitly. Useable in this context therefore means that the result's format is sufficient to be directly incorporated into the user's application or algorithm. Figure 4.3 illustrates the Task Model and also shows how Task performance data is captured and used.

Figure 4.3: Task Model



4.3.3 Discussion

By employing these two models and separating acquisition, allocation and scheduling job granularity is decreased and abstracted. Whereas before the user provided an executable, data and the means to begin execution (i.e. arguments or a control sequence), now only an executable is provided with a generic control sequence to initialize it. This is only a subtle difference, an executable is still provided, as is some initialization sequence or pre-staging, but such a job will not actually do anything without steering. By initializing a general worker daemon, and therefore isolating any scheduling strategies, it is possible to guarantee workstation stability whilst the daemon is available. There are several reasons for this. Firstly, a job cannot runaway, because the factors that cause runaway jobs to occur have not prevented the daemon from initializing. Thus, once the daemon is available, it demonstrates that a workstation can be used. If the workstation's voluntary status changes, the daemon will either shutdown, or the resource management software will employ its rules, with the most likely outcome being eviction. In both cases any Tasks can be rescheduled to another daemon. Secondly, scenarios like incorrectly configured workstations cannot interfere with Task execution. If a workstation is not correctly configured, the daemon will simply not start, and therefore cannot receive Tasks, which prevents the Task blackholes experienced during the empirical study of Matlab jobs on the Campus-Grid in Section 2.3 from emerging. Finally, once the daemon has been initialized, tracing failure becomes a lot easier. Either an event which can be captured by the Campus-Grid resource management software caused termination or failure of the daemon, or the Task itself caused the failure, for example through programmer error. In the first case the fault would be documented in the relevant logs and can be collected by autonomic managers. In the second, the scheduler can be interrogated to determine what was sent to the daemon, and this can be presented to the user. This means that external factors, which provide significant challenges can be side-stepped. The method of job abstraction and the inherent isolation of control from the resource management software are the sources of this ability. The acquisition model (Section 4.4) defines how a software entity that can replace the user and perform job administration on their behalf. Therefore, the user is left to perform and perfect their analysis rather than debug and administrate a volatile and complex distributed system.

There are also other benefits of this model. Startup overheads can be significantly reduced as a result of each worker daemon being controlled externally. This occurs because a Task does not need to go through the same initialization routines as it would do normally. Instead, it can be given to a worker daemon, which has already progressed through the typical initialization routines, leaving only data and the Task itself to be transferred. Similarly, research by Cai et al. [138] showed that queuing overheads could consume significant proportions (60%–70%) of the application makespan for short jobs of around three minutes in length. Using the models presented in this Chapter such overheads will occur only once per daemon initialization as opposed to once per Task.

Much work has been undertaken on scheduling routines for a myriad of different scenarios and contexts. It is a shame that little (if any) of this knowledge can be put into action across a vast collection of resources such as a Campus-Grid, which typically uses a centralized and general scheduling model. Using the daemon model, scheduling strategies can be chosen to fit the context of the Task space, as opposed to a generic scheduling strategy. The latter must consider the different priorities and agendas within the Campus-Grid, but also for the myriad of applications that the Campus-Grid facilitates. A further discussion of scheduling strategies is presented in Section 4.7. The resource pool, modelled under the analogy of a Virtual Dedicated Cluster, enables scheduling strategies normally used for dedicated clusters to be adopted. In addition, any scheduling strategy which fits into the cluster model can be brought forward into this context, provided that it considers the assumptions of the scheduling model (Section 4.7), both conceptually and in its implementation.

Execution times can also be closer to their expected values, because the workstation must only initialize once for many Tasks, instead of once per Task. Considering the results of Section 2.3, a removal of the startup overhead would lead to an average reduction of 2 minutes per Task. This may not sound considerable, but a job space of only 720 Tasks could be completed 24 CPU hours[7] earlier with this reduction. In Section 2.3 5,000 Tasks were executed, through this simple approach would have yielded an improvement of almost 167 CPU hours (almost 7 CPU days). Note that this does not include any time that can be saved through a reduction in the scheduling overhead, which occurs due to the interactive and infinite nature of the worker daemons.

[7]CPU time is not conceptually the same as standard wall clock time. In fact, its exact definition is somewhat difficult to timelessly provide. In this context it is a sum of wall clock times provided by one or more CPUs. In this manner a unit of CPU time can be delivered in much less time than that which it relates to. For instance, a CPU day can be delivered in a wall clock hour, if 24 nodes each deliver one hour of CPU concurrently.

4.4 Resource Acquisition

The representation of a resource still remains the same. A resource is a single Campus-Grid workstation, acquired on the behalf of the user, using the available resource management mechanisms. It is also still subject to normal rule set and restrictions i.e. sandboxing, suspension and eviction. The difference is that the control over **what** the workstation does has been isolated from the Campus-Grid resource manager. This means that the original resource management software is simply used as a source of raw computational power and only has limited control over the resources it has provided.

4.4.1 Resource Model

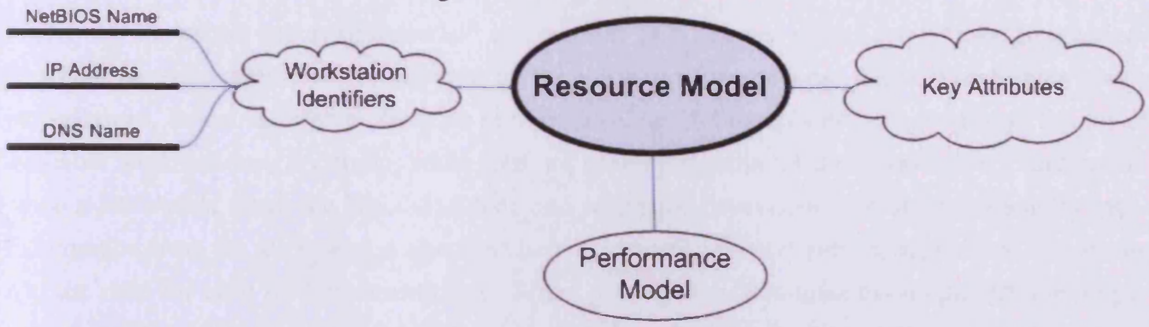
Before a detailed definition of an acquisition model is presented, the data which such a model uses must be defined. The definition above, which whilst true, does not contain sufficient detail to enable an acquisition model to consider the volatility of the resources it attempts to acquire.

An unequivocal attribute that must be considered is past performance over time. This is because intermittent resource testing cannot verify stability, but can create a general picture over time. For example, if a workstation has experienced 100 consecutive job failures, it is more likely that its next job will fail, as opposed to a workstation which has experienced one or two intermittent failures. Given the difficulty in differentiating between a runaway job and an incorrectly configured software installation, it must be assumed that workstations which experience large quantities of runaway jobs are also more likely to experience malfunctioning jobs in the future. For these reasons the resource model considers only past performance.

Figure 4.4 shows the resource model which physically identifies workstations and their key attributes. The key attributes that are an intrinsic part of the model are environment specific, and can include: **(1)** claims of software availability, **(2)** the owner(s) of the resource, **(3)** benchmark scores of capability, and **(4)** physical workstation attributes such as memory, processor speed etc. The key attributes are necessary because the construction of a model, which encapsulates the anticipated performance of both Task execution and a workstation, is defined by these attributes. For example if an attribute denotes a specific capability it is expected that this is true. Therefore, the inclusion, and monitoring, of these key attributes is important for the system as a whole, as any inaccuracies directly affect stability and performance. The results and discussion in Chapter 2 also act as an additional motivational force for their inclusion. In this case workstations reported specific abilities, namely the ability to execute compiled Matlab jobs, but when jobs were scheduled to these workstations they failed, due to incomplete or missing libraries. By capturing these key attributes and monitoring and reassessing in the future whether they are in fact true, will reduce the number of failed jobs, which will inevitably reduce resource waste.

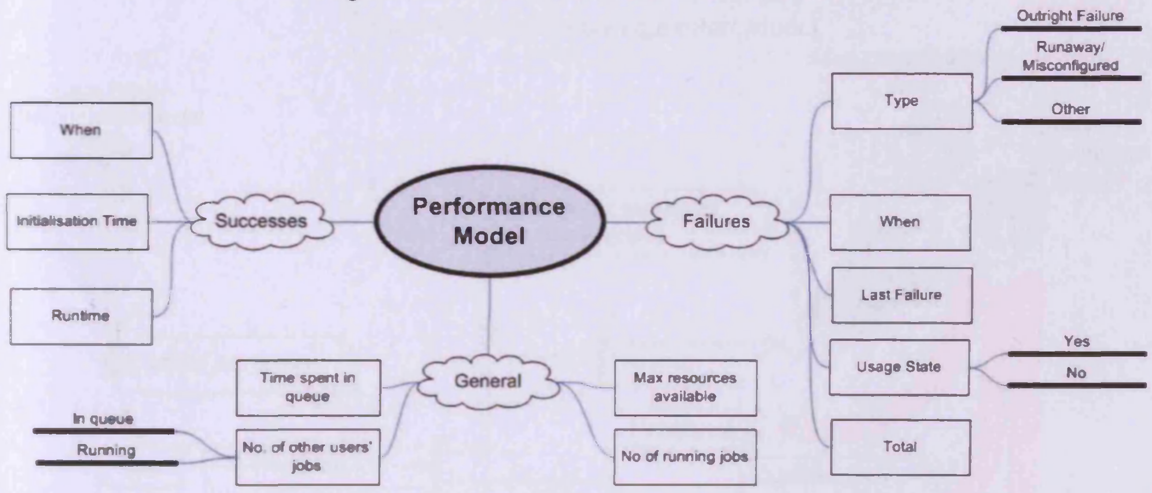
Figure 4.5 shows the performance model which is used to identify workstations that are not fulfilling the expectations placed upon them. This information can then be fed back into the resource acquisition process, which effectively prunes the set of possible workstations upon which a job can

Figure 4.4: Resource Model



be run.

Figure 4.5: Workstation Performance Model

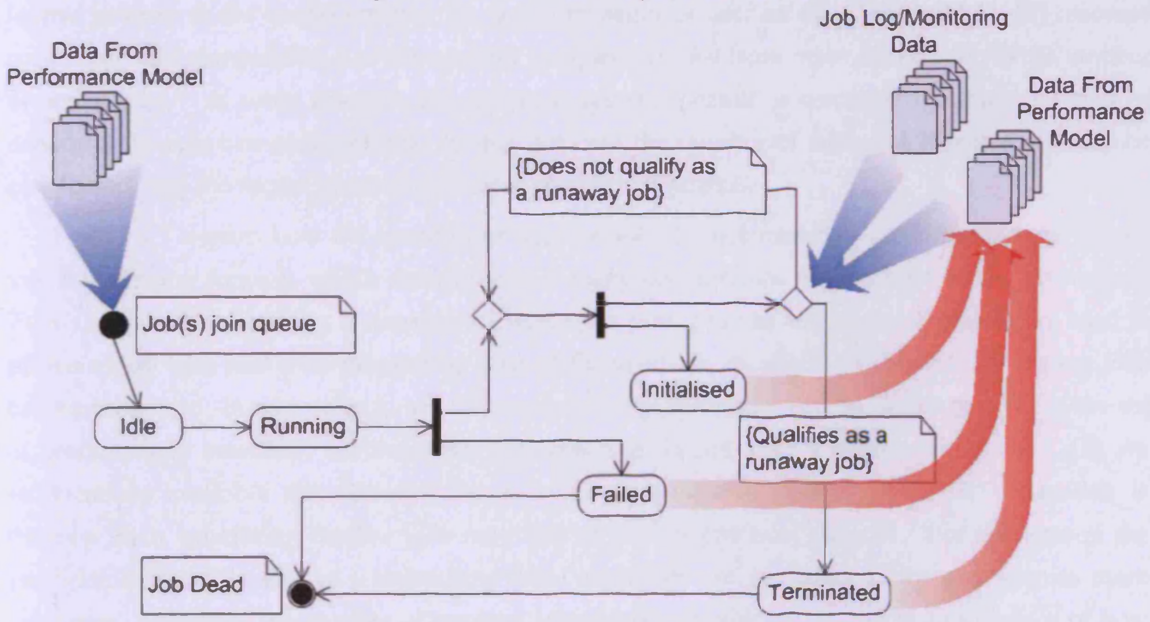


The performance model captures information about how jobs perform, e.g. how long a job ran, how long it needed to reach a specific state, the time between state transitions etc. One of the advantages of the job model is the ability to determine when a daemon's initialisation phase has completed. This data can also be used in real-time to help diagnose runaway jobs. Furthermore, it enables decisions to be made using data specific to an individual workstation when previous usage statistics exist. Therefore, a performance model can be built over time for each available workstation. The information that this model contains can be fed back into an autonomous manager, to enable decision making processes for job termination when anomalous behaviour is identified. It also permits the ability to prevent the use of specific workstations if they fall below threshold values.

4.4.2 Resource Acquisition Model

Figure 4.6 shows the acquisition model[8] and how the performance model can be used in practice. Firstly, it provides additional information for the job submission process. Here, it highlights which workstations, based upon previous poor performance, are to be explicitly removed from the set of available workstations. Secondly, when jobs are running it provides the information to determine when a job *should* reach the initialized state and where the bounds are that identify runaway jobs. Information from the job model is also used here to identify a job's current state, its state transitions and the time for each of these transitions. When put together, this information provides an upper bound and if a job's runtime exceeds this bound before the initialization state is reached, it is terminated. All eventualities are then fed back into the performance model for future decision making processes. An element of evolution is assumed in this approach. Over time the profiles of workstations develop and threshold values derived from this data set change. This means that the criteria for identifying anomalous job behaviours is not static and therefore captures the dynamic nature of the resources in use. For example this approach could easily catch the changes in a workstation's announced capabilities which were noted in Section 2.3.

Figure 4.6: Resource Acquisition Model


Job Log/Monitoring Data
Data From Performance Model
Data From Performance Model
{Does not qualify as a runaway job}
Job(s) join queue
Initialised
Idle
Running
{Qualifies as a runaway job}
Failed
Job Dead
Terminated

The approach for terminating jobs suspected of running away can make mistakes. The results presented in Section 2.3 demonstrated the variable nature in a workstation's execution even for identical jobs. However, despite this variability and occasional dramatic changes in performance, such eventualities should be only seldom occurrences. For example 99 - 100% of successful Matlab test jobs in the analysis carried out in Section 2.3 were within four or five standard deviations of

[8]Note that states which can occur after daemon initialization (for example: executing Task, suspended, evicted) are not shown.

the average runtime at the workstation level.[9] In contrast, 95% of all jobs which ran away either as a result of incorrectly configured software or architecture-related overheads could have been identified, terminated and resubmitted using the same threshold values. By using historical data in the way illustrated in Figure 4.6, some mistakes in prediction will occur and cause daemon jobs to be both incorrectly terminated and allowed to continue for longer than necessary. However, this should not be a frequent occurrence. Using such an approach the prediction of runaway jobs should not only be accurate but also prevent a workstation with an incorrectly configured software component from being blocked any longer than is necessary to identify a fault.

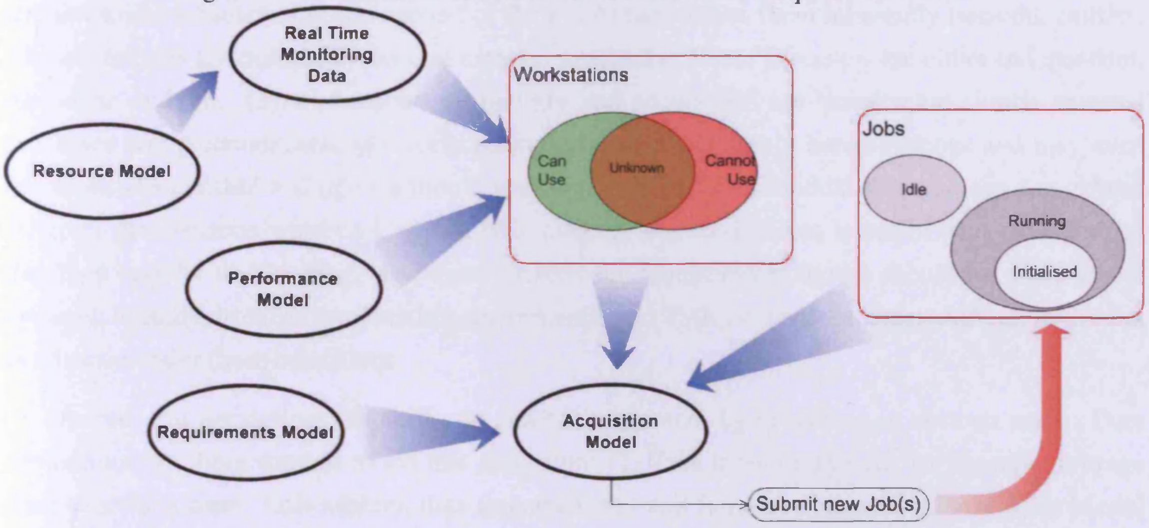
This model describes only how resources are acquired and what happens when anomalies are detected or predicted to occur. How successfully initialized resources are later used is inconsequential for this model. It aims only to initialize daemons, and does not specify how they should be used after this point. Yet, one piece of information is missing from Figure 4.6: the number of daemons that should (ideally) be in the initialized state. The overall aim of the acquisition model is to reach a requested minimum number of available (initialized) daemons, which can be given to one or more schedulers (see Section 4.6). There are a number of different factors which influence the model's ability to achieve its goal. **(1)** the number of available workstations, **(2)** the rate of change in workstation availability, **(3)** the ratio of failures and runaway jobs to successful jobs, **(4)** the effective priority of the model enactor, i.e. a job manager or user on the Campus-Grid, **(5)** resource contention and competition, i.e. the number of (queued) jobs from other users, and **(6)** the number of active jobs,[10] as some resource management systems permit or recommend a finite number of concurrently active or running jobs. In this last case the number of submission points can also be considered, and the model enactor replicated at multiple points.

Figure 4.7 depicts how the quantity of daemon jobs is determined. Essentially there are three key contributing sources, which determine how many daemon jobs are in effect, at any given time. Firstly, physical availability is considered. Here, the performance and resource models are used in unison along with real-time monitoring data of the available resources to determine if further jobs can be supported. In other words, are the answers to the following two questions yes: **(1)** of the set of workstations available, are there workstations that do not have a history of failures? **(2)** are workstations available that advertise the necessary capabilities to host a daemon? If neither is the case then submitting further jobs may not serve any practical purpose. For this reason the acquisition model is always (re)checking what resources are available, to try and acquire more resources. Secondly, the number of running jobs is also considered, as this is an indicator of how many daemons are likely to materialize. Prediction of the likelihood of successful initialization could also be performed here to provide a more accurate estimation. Finally, the requirements model (Section 4.5), which encapsulates a user's demand for Campus-Grid resources, provides an abstract numerical target of workstations that the acquisition model should aim to initialize.

[9] Using the data of all successful jobs, instead of at the workstation level, an upper boundary derived from five standard deviations of the average runtime would include 99.7% of all successful jobs.

[10] Active here, refers to all jobs that are not dead.

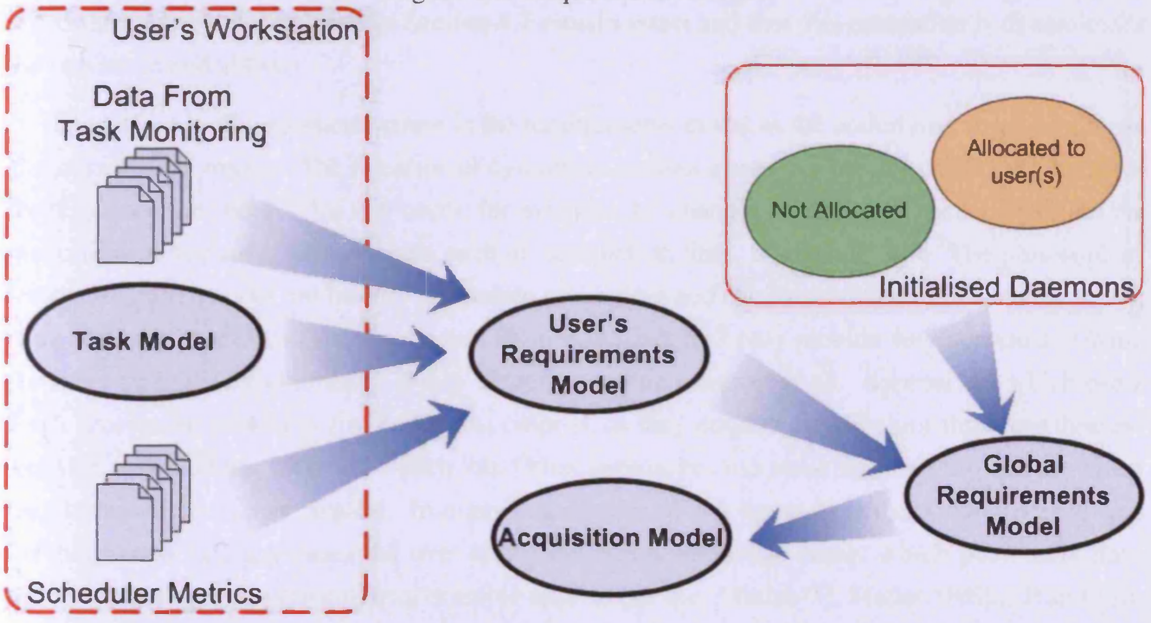
Figure 4.7: Flow of Information for the Resource Acquisition Model



4.5 Requirements Model

The requirements model (Figure 4.8) has two tiers: **(1)** a user's resources requirements, and **(2)** the global resource requirements. Even if there is only one active user these two tiers will not produce the same output, as their aims and objectives are significantly different.

Figure 4.8: Requirements Model

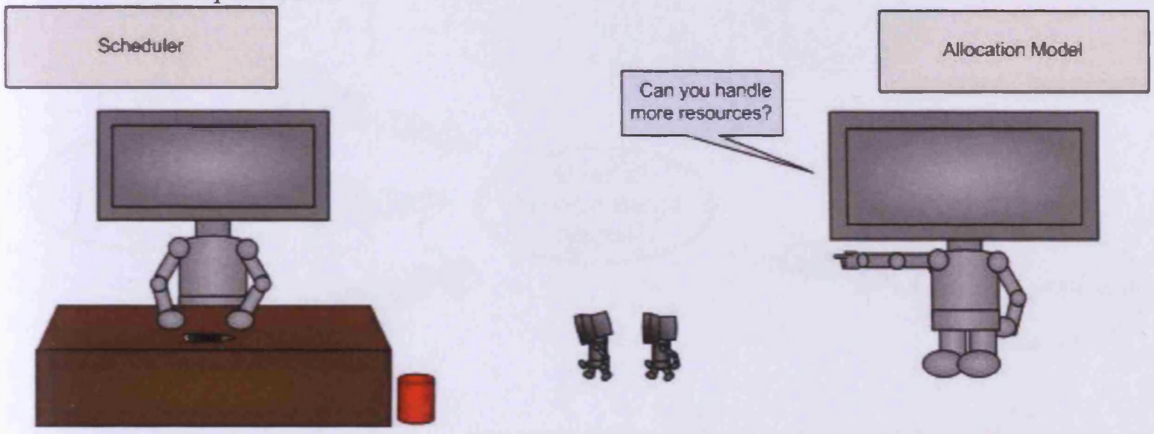


The user's requirements model is not in place to identify an exact or static numerical estimation of resources needed to (optimally) execute a Task space. Such an approach would be ill-conceived as the probability that this number of resources could be delivered for the duration of the execution is low. There are several reasons for this: **(1)** daemons do not initialize synchronously, **(2)** daemons attempt to run infinitely, but the context of their existence makes them inherently transient entities, whose lifetimes are defined by several external processes. These processes are either independent, stochastic or both. **(3)** workstation availability and acquisition are based upon similar external processes and preconditions, **(4)** workstation performance is highly heterogeneous and may even be inconsistent within a single daemon's working session. Two identical workstations can exhibit different performance within a Campus-Grid context, due to different levels of background work that they may be undertaking. For these reasons the requirements model should be reactive and dynamic initially, because workstation performance and Task demands are very difficult to predict in advance under these conditions.

Instead, this tier defines the ability to consume resources by providing an abstract range. Data can come from three sources to aid this definition: **(1)** Task monitoring data, for example average Task execution time, Task affinity, data transfer times and failure/error rates. **(2)** the Task Model itself, for example, the number of Tasks still awaiting execution, have finished or are currently executing. **(3)** scheduler specific metrics, for example the minimum and maximum number of daemons that are required or can be realistically used given previous Tasks and the current sizes of daemon queues. In addition, reactive measures to current performance can be employed. This information is provided by the scheduling model (see: Section 4.7). The simplest implementation of this model is the identification of the minimum and maximum number of daemons that could be used, given this input data. Other implementations could identify *"ideal"* ranges or rank values within the abstract range to aid in resource sharing or to prevent over allocation of resources provided that the scheduling strategies presented in Section 4.7 remain intact and that this estimation is dynamic, for the reasons stated above.

Dynamism is an important feature in the requirements model as the underlying resources in use are themselves dynamic. The retention of dynamism enables a user to represent (dynamic) demands for resources over time. This can occur, for example, by changes in the Task space, or through the invocation of semantic requirements such as completion time, throughput etc. The philosophies surrounding this model are heavily dependent on context and can be modified and extended to suit many different scenarios. The approaches defined in Chapter 3 only provide very general and static means of defining requirements, if they actually provide a means at all. Approaches which use a batch processing paradigm are the biggest culprits, as they employ greedy algorithms and then establish a new working session for each job. Other approaches use static resource acquisition which may be based upon user request. In many cases users do not know their exact requirements and for this reason in many cases the over allocation of resources can occur, which potentially have detrimental effects on performance, example approaches are: Matlab*G, Matlab*P/Star-P and ProteomeGRID. Using the model presented in this Section a user, intelligent or autonomic controller can create flexible and dynamic or feedback orientated requirements during runtime. Ultimately, these requirements may not be met, given the context of the execution environment. The number of daemons that a scheduler will receive is inevitably dependent on the number of available workstations, which successfully initialize a worker daemon. However, the intrinsic idea of illustrating the ability to consume resources, fits well into the paradigm of opportunistic computing. For the reason that as new resources become available, the requirements model captures the necessary information to inform the acquisition model whether these additional resources would be useful. This idea is depicted in Figure 4.9.

Figure 4.9: An illustration of the dynamic nature of the requirements model and its ability to capture a scheduler's ability to consume resources, rather than pin down an exact numerical representation of the resource requirements.

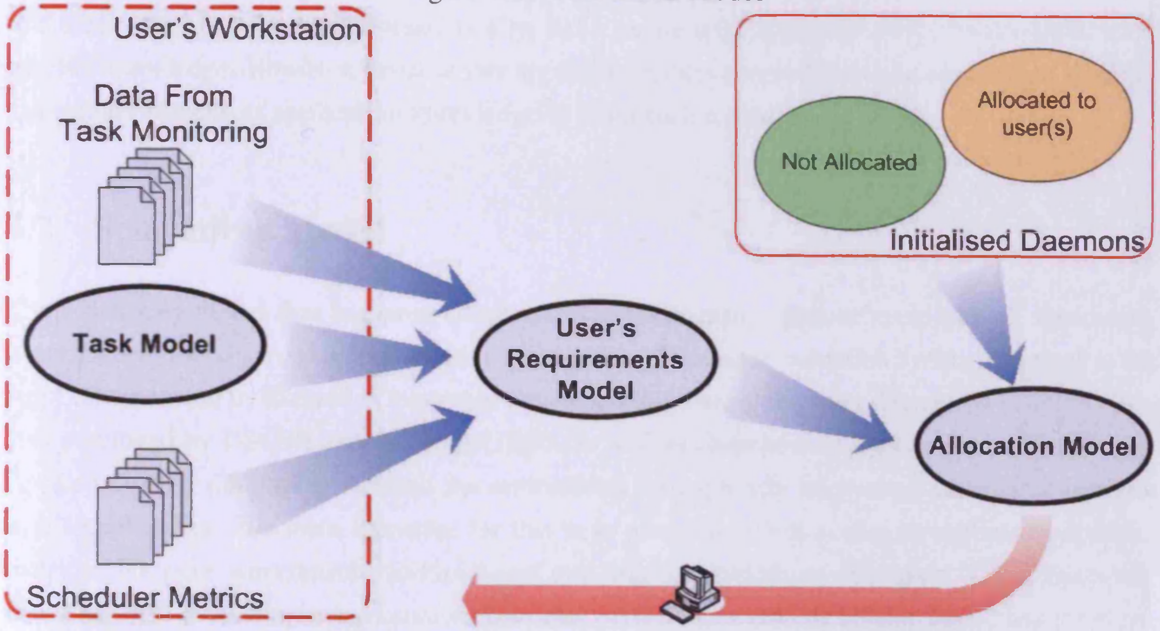


The global requirements model is an aggregation of all the users' requirements models, but with an additional buffer. This buffer represents the intention to acquire spare capacity, which, in turn, relates to the ability to provide additional resources on short or no notice. The provision of a concrete numerical target for the acquisition model is of little use. In doing so, the requirements model would ignore the dynamic nature of the resources it asks the acquisition model to acquire. Instead, the global requirements model provides a dynamic horizon for the acquisition model to aim for, but which may not be reached. The horizon represents the maximum number of consumable resource plus an additional buffer (for spare capacity acquisition). Only when the announced capability to consume resources begins to reduce could this horizon ever be reached. The most likely scenario where this would occur is during the final stages of an application's execution, where the number of outstanding Tasks is beginning to approach zero. Whilst users are online the horizon is constantly reevaluated, as users' requirements models are explicitly linked to their execution contexts and are therefore dynamic. Any additional resources which become available but cannot be consumed are put aside as spare capacity.

4.6 Requirements-based Allocation Model

Once the requirements model has illustrated the ability of a user to consume resources and some resources have materialized into daemons ready to execute users' Tasks, they must then be allocated to the user. In a single user context, whatever resources are made available through the acquisition model is what will be allocated, as illustrated in Figure 4.10. This will continue until the requirements model prevents any further resource allocation, i.e. the upper bound has been reached. At this point the acquisition model can begin to acquire spare capacity.

Figure 4.10: Allocation Model



Should another user materialize, a redistribution of resources can be employed, with spare capacity allocated first. Here, the requirements model will also facilitate resource sharing in multiuser contexts. In such scenarios the juxtaposition of users' requirements models can be used to quantify the *need* that a user has for resources. Comparative analysis can then be performed to determine the share of the available resources each user receives. Here, the assumption is that as users progress through their Task spaces, their needs for resources change.

Resource requirements can change either as a result of Task spaces decreasing or increasing in size, but also as the heterogeneity of users' Task spaces will become more apparent over time. The same is true for the workstations they have been allocated. The philosophy behind this method shares similar motives with the requirements model as a whole, where the ability to consume resources is captured. In a multiuser context this ability can be modelled in parallel with need, because the metrics provided by a scheduler are also intrinsically indicative of need. Therefore, the requirements model can also be used to describe a user's need for resources and in turn facilitate decisions concerning the number of resources that are allocated to different users.

This is a natural progression of the requirements model, which is built upon retrospective event analysis, to try and model future requirements. The need of a user will also change over time, and thus the allocation of resources should change as well. Through this approach users can gain and loose resources over time, as their requirements models balance resource distribution. Similarly, as new daemons become available, the requirements models provide the ability to decide who receives additional resources. The result is a dynamic allocation model, which is based upon feedback from localized schedulers, Task monitoring and a representation of the ability to consume resources. The question which occurs at this point is: what methods can be used to choose which workstations are given up by a user to be allocated to another? This research is not orientated to answer such a question, but its answer could come from the scheduling model itself. The benefit of a workstation to a scheduler can be modelled heuristically, but a meaningful approach must contain application specific knowledge. However, because this approach enables a scheduler to be application specific, a scheduler can use its application knowledge to build such a model.

4.7 Scheduling Model

The scheduling model that has been adopted differs from many approaches in that all scheduling is performed directly from within a user's application. The major exception to this statement is the AppLeS extension to Nimrod. Oberhuber suggested this during the final discussion of DIPS. It is also employed by DIANE and ProteomeGRID, as well as elsewhere in the literature (for example AppLeS [119]). Oberhuber stressed the motivations for explicitly integrating scheduling into the user's application. His main incentive for this is to give schedulers access to application specific metrics. He gave workstation workload and available bandwidth as examples. Other examples could be: data available on workstation, previous performance data of certain Tasks, and previous experiences with a workstation such as the duration of its availability.

The models which are defined in this Chapter are capable of enriching this idea with a significantly increased level of detail. However, the volatility of the resources in use means that the scheduling model is significantly different from the one Oberhuber suggested. The primary reason for this is context in which DIPS was designed: dedicated (stable) clusters of resources at different computational sites, as is DIANE, rather than volunteer (volatile) resources of a Campus-Grid. ProteomeGRID is potentially the only approach where this could be considered, however, it is not. This makes an entirely different context and therefore leads to a very different approach, but nevertheless has similar motives.

There are two primary categories of scheduler that can be defined: (1) a global scheduler, which serves multiple users/applications, for example: Condor, PBS [126], and other queuing/batch schedulers, and (2) a localized scheduler, which is specific to one application/user for example: AppLeS. Oberhuber only touched upon the motives for an integrated scheduling model. Many additional reasons can be raised to further motivate this approach: (1) A local scheduler can be written specifically for an application domain or context. (2) A localized scheduler has direct access to real-time information of the Task space, allowing it to schedule Tasks considering attributes such as Task affinity, requirements, and previous performance of similar Tasks. A global scheduler must consider many different scenarios, making it a general approach bound by potentially incompatible assumptions and agendas. It cannot guarantee any levels of quality in relation to its scheduling strategy to the disparate application scenarios it is trying to serve. Inevitably, it becomes a performance bottleneck.

(3) If all Tasks do not share the same requirements set and, specifically, if custom requirements are in place a localized scheduler can handle such scenarios more intuitively than a global scheduler. **(4)** Feedback loops, such as those illustrated in the Task-Model (Figure 4.3), are easier to instantiate and can consider an application's context. For example, a local scheduler can consider the question: given some previous analysis, what is the minimum number of workstations needed to perform some new (but similar) analysis in time T? **(5)** Localized scheduling can accommodate changes in an applications context, agenda, requirements and Task space more easily because it can consider semantic and application specific information. **(6)** Localized scheduling permits advanced users to create their own scheduling rules and decision making processes. It also permits users or application managers to choose between different scheduling strategies, should multiple options be available. This was demonstrated by the AppLeS project.

Scheduling is not a primary focus of this research, and therefore the scheduling model has been conceived in a way that it enables the adoption of many different scheduling strategies, which can either be sourced from the literature or specifically designed for an application. The novelty of this approach is the ability to draw upon a wide range of scheduling algorithms developed for domains other than those of opportunistic and volunteer computing. This ability stems from the model of a virtual (dedicated) cluster of workstations. Instead of providing an implementable framework, a series of assumptions and guidelines are presented in Section 4.7.3 and should be considered before any scheduling strategy is integrated into the system presented in Chapter 5.

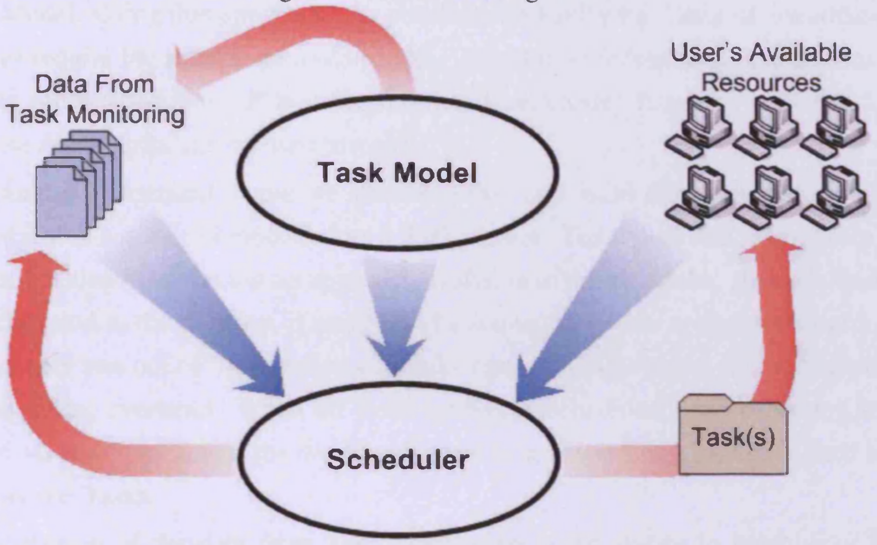
4.7.1 Dynamic Task Clustering

As mentioned in the daemon model (Section 4.3.1) and above, worker daemons have local queues to help accommodate short jobs and reduce the impact of data transfers. Yet, a daemon's queue does not need to have a finite or static size. Therefore, a scheduler can vary the number of Tasks a daemon has in its remote queue. It can base this decision upon three input sources, each of which are shown in Figure 4.11: **(1)** the Task Model, **(2)** the data from Task monitoring, and **(3)** the number of workstations currently available to the scheduler.

The main contribution of the Task Model is the number of Tasks which are still held locally, i.e. unscheduled. As this number reduces toward zero, queue sizes should in turn tend toward one.[11] If this were not the case, the user's requirements model (Section 4.5) could not describe the user's ability to consume resources as accurately. This is because fewer Tasks would be held locally and additional (new) workstations would have less work to do and eventually evoke the need for load balancing. This, in turn, reduces the demonstrable ability to consume resources and would make it less likely for additional workstations to be allocated. In addition, if the daemon queues are too large, adding additional resources would force Task reorganization, which is also undesirable. Consider this example: if the user has 100 homogeneous Tasks and 10 homogenous workstations (for simplicity), a scheduler should not use a queue size of 10 (the maximum value in this case). If

[11]Queue sizes are defined by the number of queued and executing Tasks i.e. 0 queued and 1 executing is a queue of size one as is 1 queued and 0 executing.

Figure 4.11: Scheduling Model



an eleventh daemon became available and the queue size across all daemons was 10, up to nine Task reallocations would occur initially. This would occur because the maximum queue size that could be supported (assuming all workstations have the same queue size) for eleven workstations and 100 Tasks is nine. However, had the scheduler adopted a queue size of five, for example, the addition of an eleventh workstation would not have evoked any Task reallocations upon receipt of the new workstation, as no more than 50 Tasks could be buffered at any time. Therefore, a scheduler should autonomously decrease the size of daemon queues as the number of unscheduled Tasks decreases, so that the allocation of any new workstations does not cause unnecessary rescheduling.

Load balancing (or Task reallocation) is the rescheduling of Task(s) from one or more daemons to another. It occurs when either of the following is true: **(1)** a new daemon becomes available and no Tasks are currently unscheduled, or **(2)** a daemon has completed all Tasks scheduled to it, and no other Tasks remain unscheduled. This can occur toward the end of an application's execution cycle as a consequence of the heterogeneity of workstations, previous Tasks (which can benefit from caching), Task heterogeneity and the number of Tasks not being a factor of the number of workstations. Ultimately, this means some daemons run out of Tasks before others and so load balancing is required. When either of these scenarios arises, the scheduler must decide firstly, from which daemon(s) to take Tasks and then, how many should be taken. Some solace is preserved in that such scenarios do not occur often, and usually only as the size of a Task space approaches zero.

There are several ways to decide which Task should be taken from which workstation. The actual strategy which will facilitate this decision is specific to the scheduler, and therefore the application. Yet, there is one basic heuristic that can be used and built upon:

$$w' = \forall w \in W min(AvgExecutionTime(T_{sim}) - RunningTime(T_e)) \qquad (4.1)$$

Where w' is the selected workstation, W is the set of all workstations, T_e is the currently executing Task for a given workstation, and T_{sim} is a *similar*[12] Task which has been observed in the past by the Task Model. Using this approach it is possible to identify the Tasks of a workstation that are most likely to require the most time to complete. Then the scheduler can take the Task scheduled most recently for rescheduling. If no affinity-related monitoring has been performed the average execution time of all Tasks can be used instead.

The dynamic management of remote queues in this way, is, in effect, dynamic Task clustering, where each daemon's queue is modelled as a Task cluster. The dynamism stems from the resizing of the cluster, but also from the *top up* approach available to the scheduler. As each Task completes, another is scheduled to the daemon, if and only if the queue size has not been reduced. This means that daemons only run out of Tasks when no Tasks remain unscheduled, increasing utilization and reducing scheduling overhead. When no Tasks remain unscheduled, load balancing strategies can be employed. It also means that the workload given to a workstation can be related to its current ability to consume Tasks.

The contribution of the data from Task monitoring is the ability to increase or decrease the maximum queue size, based upon past performance. For example, if Tasks have a low average execution time it may be beneficial to increase the queue size. In doing this, the requirements model would highlight a reduced capability to consume resources and the granularity of Task clusters decreases (i.e. it becomes coarser). The result of increasing queue size is that more time is allocated to perform any data transfers, which is needed due to a low execution time. The obvious penalty that could be paid is the relatively high cost that a workstation revocation could have.

The queue size does present a definite trade off (illustrated in Figure 4.12). As queue size increases the potential to reduce the effective overhead also increases as data transfers can begin earlier, have more time to complete and run in parallel. However, as queue size increases, so too does the probability that newly acquired daemons will cause Tasks to be rescheduled. In addition, workstation revocations will become more expensive. Such a scenario can make any reductions in overhead from a larger queue completely redundant, as data may need to be retransferred.

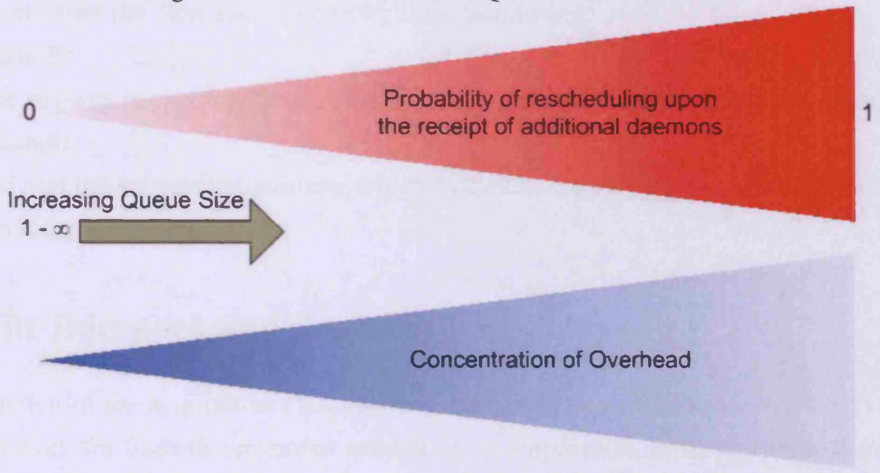
4.7.2 Task Affinity

Affinity is a measure of similarity, between two or more Tasks. Similarity has two main sources, each coming from the Task Model: (1) input data, and (2) Task Sequence. The impact of the latter is less significant and more challenging to quantify.[13] However, Task affinity derived from similar data input yields a much greater benefit. This stems from the common input data, as scheduling such Tasks to the same daemon results in a reduction in data transfers. Scheduling of this type is only possible (with the exception of traditional clustering) due to the interactive nature of daemons, as a new working session would mean any data already held by a workstation is lost.

[12]For similarity see definition of Task affinity in Section 4.7.2.

[13]It is less significant because only caching benefits will be received. It is harder to quantify because of the need for detailed knowledge of what execution will occur in relation to input arguments and data.

Figure 4.12: The Daemon's Queue Size Trade Off


0
Probability of rescheduling upon the receipt of additional daemons
1
Increasing Queue Size
1 - ∞
Concentration of Overhead

4.7.3 Scheduler Considerations and Assumptions

Figure 4.11 depicts the abstract scheduling model and shows how information flows between the different components. Essentially, the idea is that the Task model provides all the necessary information to enable a scheduler to schedule a Task to a workstation on the Campus-Grid. The scheduler and therefore the scheduling plan will be updated every time that either: **(1)** a Task finishes, or **(2)** a workstation is revoked/allocated, i.e. it is event driven. The challenge of scheduling within this context is not knowing how workstation availability will change with time. On the one hand, more resources may suddenly become available, but on the other hand many could be revoked. For this reason the following considerations and assumptions are provided.

Consideration 1:

The scheduler must be conscious of the potential for change in its Virtual Cluster of workstations. Therefore, it cannot plan too far ahead, but must, instead, be opportunistic and when planning, consider the implications of having to change its plans. This can occur either as a result of Task failure or increases/decreases in a scheduler's cluster of resources.

Consideration 2:

The model assumes that Tasks will be buffered in the remote queues of daemons. It is at the scheduler's discretion what size these queues are. Their size is related to how the requirements model will view the scheduler's ability to consume resources. However, note that queueing Tasks in this manner is, for all intents and purposes planning and, therefore, the larger a queue, the further ahead the scheduler is planning. On the other hand, not queuing Tasks adequately can increase the concentration of over data overheads (see Figure 4.12). Further discussion of Task buffering is given in Section 4.7.1.

Consideration 3:

Monitoring data is made available to the scheduler in order to aid the scheduling algorithms. It is assumed that the scheduling strategy will use this data, but it is not essential.

Consideration 4:

It is assumed that the scheduling strategy will capture the historical data it needs if this data is not available from either the Task model or from Task monitoring.

Consideration 5:

A scheduling strategy is expected to provide the data needed by the requirements model.

Consideration 6:

It is assumed that the scheduling strategy will provide the means to balance load between workstations as required.

4.8 Fault Tolerance and Recovery

The combination of the acquisition (Section 4.4) and job/daemon (Sections 4.3/4.3.1) models provides the methods for fault management needed for this approach, as they provide the capability to identify workstations that cannot contribute their resources. To facilitate fault tolerance at runtime and after the allocation of resources, both proactive and reactive measures are employed. These guarantee that once a workstation is part of the virtual cluster of resources, its contribution is subject only to the rules of the Campus-Grid architecture,[14] intermittent faults[15] and user error.[16] It also means that the additional fault tolerance mechanisms required for this approach are quite ordinary and can be classified into two domains: (1) infrastructural, and (2) Task-orientated fault tolerance.

4.8.1 Infrastructural Fault Tolerance

Infrastructural faults are events which affect the stability of the running system (daemons). Of the three categories given above, infrastructural faults relate to the rules of the Campus-Grid architecture and intermittent failures. Here, the predominant aim is to identify daemon failures. In this case failure is defined as the end of, or a pause in,[17] a daemon's life, regardless of its cause. The reason for this choice is that the acquisition model will catch any untoward architectural faults of interest, and that the primary objective is to alert the scheduling model of the loss as quickly as possible, so that any affected Tasks can be rescheduled.

Identifying daemon failure can be performed using a combination of two standard approaches: (1) heart beat monitoring, which is effectively a periodic ping to ensure that a daemon is still available at its endpoint, but also able to respond, and (2) state awareness, which is collected from the Job Model and enhances the information provided by the heartbeat monitoring. For example, if the job model detected that the daemon's job had been terminated, suspended or experienced other state transitions, but the daemon was itself still alive according to its heartbeat, this knowledge can be used to instruct the scheduler that this daemon may soon fail.[18] On the other hand, if the daemon's

[14] For example revocation, suspension and workstation failure or power down.

[15] For example lapses in network connectivity, insufficient memory on a workstation and other runtime errors.

[16] For example programmer error.

[17] For example through suspension.

[18] Such a scenario was demonstrated in Section 2.3, where the reported job runtime in the host workstation's log was significantly longer than in the job log on the submit (user's) workstation. If such a scenario arose during the execution of a daemon, it would mean that the daemon itself would remain alive, even though the job model reports it as dead.

heartbeat ceases, state information can be used to diagnose the cause of the failure. For example a temporary network fault or a few dropped heartbeat messages would indicate that the daemon had failed, but using the Job Model, the daemon's current state may not have changed, therefore other means of communication can be used to restart the daemon's heartbeat. These two fronts enable a tight monitoring approach, as both can be used together to ensure that a daemon has in fact shutdown/failed or is still active and capable of executing Tasks. Neither of these ideas is novel, but together they provide a strong foundation for failure identification.

4.8.2 Task-orientated Fault Tolerance

Task-orientated fault tolerance is concerned solely with the handling of Tasks and the implications that other faults have upon the execution of the Task space. There are two sides to Task-orientated fault tolerance: (1) the recovery from daemon failure and the identification of potential causes, for example a Task causing the daemon to terminate. (2) The recovery from user-based Task errors, for example calling non existent functions, using too many/few input parameters, and miscellaneous runtime errors such as class loader error.

Typically, recovering from daemon failure is the resubmission of the Tasks that the daemon had in its queue. However, it may also be useful to identify which Task was running and flag that it was being executed when the daemon failed. If the job model cannot identify a reason for the failure, this Task can be observed in future submissions, in order to try and identify it as one which compromises the stability of the system so that the user can be notified.

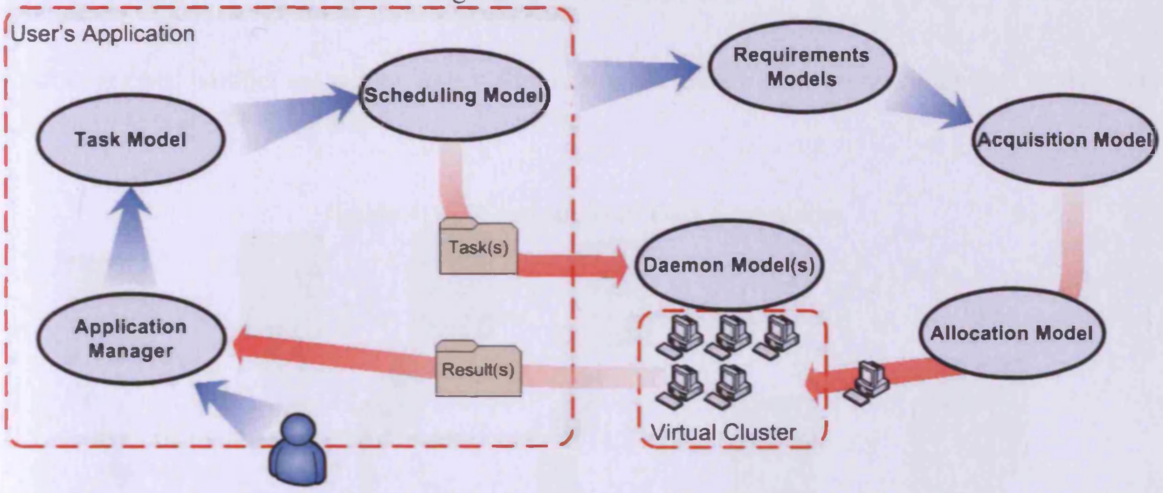
Task errors are, however, much more difficult to react to. The simplest approach is to prompt the user for a decision. Alternatives can be to attempt the Task a specific number of times before reporting it as inexecutable. Ultimately, this is an implementation issue, rather than a research question. Yet it is important to note that because of the other fault tolerance mechanisms in place, handling strategies for Task errors are now possible. In addition, they are now also clearly distinguishable from a workstation failure. These are two abilities that are not possible with the traditional Campus-Grid software used in the analysis of Chapter 2, or in any of the approaches in Chapter 3 with the possible exception of Ibis' JavaGAT.

4.9 Application Integration – The User's Model

Figure 4.13 depicts how the user's model relates to, and interacts with, the other models presented in this Chapter. The main idea in the user's model is the use of an Application Manager to reduce the effort of the user where the Application Manager transparently handles the intercommunication between the different models. This is slightly different to the interactions depicted in the Task Model (Figure 4.3). Here, an Application Manager represents the user in the system. An Application Manager could be anything from a simple application, where the user specifies their Tasks, to an instance of a PSE or an intelligent application or agent, which is capable of producing Tasks for execution. To the user it is simply an application where they describe their Task space. It can be made as complex as the user wishes, as it is a client front-end to the system. Two example Application Managers are documented in Chapter 6, which show the use of this model in practice.

From a development perspective the Task and Scheduling Models are to be considered as commodity components, which an Application Manager uses. In Section 5.5 the Client Interface (CI) is presented along with a default scheduler, which is used transparently by an Application Manager. However, new scheduling algorithms may be desirable, therefore, the CI defines a scheduling interface (Section 5.5.3) to enable new scheduling algorithms to be incorporated into the system.

Figure 4.13: User's Model



4.9.1 Granularity

Task granularity can have many different tiers each relating to fuzzy, overlapping and context- and domain-specific definitions. Therefore, to provide some perspective to this approach a definition of this term is provided to highlight the targeted application domain. In principal, there are two classes of granularity: **(1)** fine-grained, and **(2)** coarse-grained. Moreover, these two classes can be further categorized into two domains: **(1)** data granularity, and **(2)** execution granularity.

Data Granularity

In some cases fine-grained is working on a single cell of a matrix [56], it can be acting upon a single row or column of a matrix, as is the case with MPI-based approaches. It can be one slice of a three-dimensional matrix, such as in the study of MRI data etc. Whereas coarse grained predominately encapsulates data sets which have not been broken down into smaller units. When it comes to definitions of granularity, the border between fine and coarse grained is difficult to define, as many different and overlapping definitions of the two classes exist. In many cases, definitions of granularity are built upon comparative premises and in relative contexts, which are usually expressed through the use of adjectives. These definitions are sometimes also implicit in the context. For example: *A* has a coarser abstraction than *B*. For these reasons the following two definitions are presented for use throughout this thesis and the continuum of granularity abstractions for data is presented in Figure 4.14. The type of granularity that the user's model facilitates is also shown.

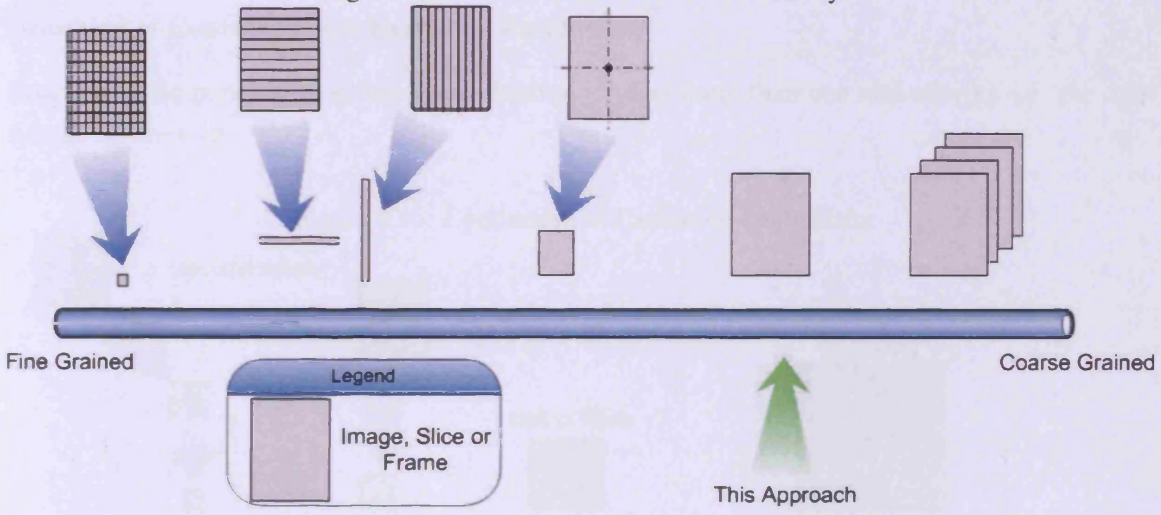
Definition of Fine-grained Data Parallelism

Fine-grained parallelism is any Task which acts upon one or more sections of a single image or data set i.e. the left side of Figure 4.14.

Definition of Coarse-grained Data Parallelism

Coarse-grained parallelism is any Task which acts upon one or more complete images or data sets i.e. the right side of Figure 4.14

Figure 4.14: Continuum of Data Granularity



Execution Granularity

Due to a more intuitive scale, definitions of execution granularity are often less difficult to pin down. For all intents and purposes, regular execution of an application on one workstation is *normal* execution and should therefore be in the middle of a continuum of granularity. In a similar manner to data granularity, execution granularity is often also expressed relatively. Fine-grained parallelism is generally when code is broken down into smaller pieces, loops and independent units of computation are the predominant examples for this. Increasing the granularity of an application is when functional units are combined to create larger units of work. However, a unit of work is much more difficult to define as its definition is application dependent and also developer dependent. For this research a definition of work is provided along with definitions of fine- and coarse-grained execution granularity. The continuum of granularity abstractions for execution is presented in Figure 4.15.

Definition of a Unit of Work

Scientific analysis is typically organized using functional components, where the lowest form of component is a base component, as most (but not all) data can be reduced to a matrix representation it is: a single matrix transformation, calculation or manipulation. Therefore, a functional component is a collection of base components combined in order to perform a specific piece of data analysis. A work unit is one or more functional components specifically combined to perform a piece of scientific analysis. Therefore, a unit of work can be one iteration of a loop as well as the entire loop, but this level of definition is application dependent.

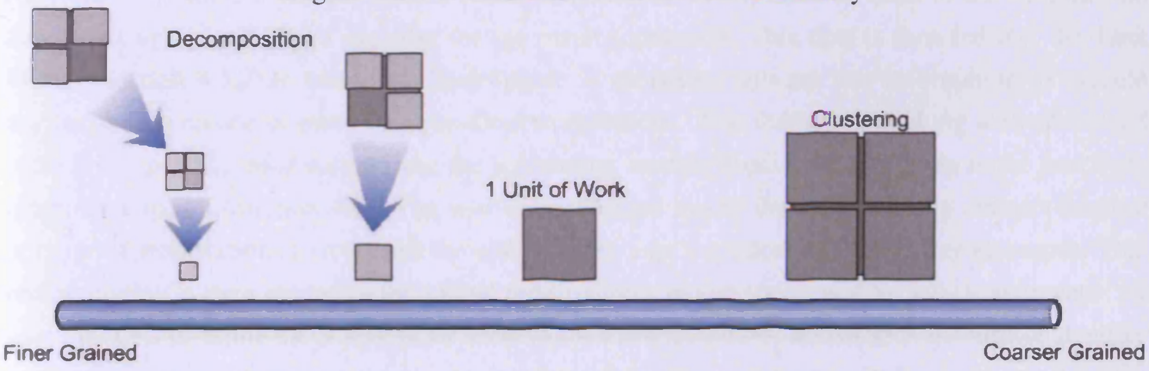
Definition of Fine-grained Execution Parallelism

Fine-grained parallelism is any Task which represents less than a single unit of work i.e. the left side of Figure 4.15.

Definition of Coarse-grained Execution Parallelism

Coarse-grained parallelism is any Task which represents more than one unit of work i.e. the right side of Figure 4.15

Figure 4.15: Continuum of Execution Granularity



The approach adopted throughout this research and for the application of the software solutions it serves, is a coarse-grained data abstraction, as indicated in Figure 4.14. Execution granularity can be facilitated across the entire continuum, but the middle and right side (coarse-grained) of the continuum will be handled transparently for the user through dynamic clustering of the daemon queues (see Section 4.7.1). The user's model does not intend to provide the ability to generate a fine grained approach transparently. Here, granularity is orientated around a data set, where one Task relates to one or more data sets.

Even in a fine-grained execution approach the data granularity will remain coarse. The cost of decomposing the data set into subcomponents is assumed to be either more expensive than processing it whole, or provide no real benefits in terms of performance. This coincides with Moler's observations in [55]. For users this is a beneficial system attribute, as it means that their original code does not need major revisions in order to use the system. In addition, it is possible to cater for rapid application development and deployment, which is facilitated by the user's model and its enactor, the Client Interface (Section 5.5). This model takes many of the intricacies of distributed computing and handles them transparently for the user. Operating at this level allows the user to work in the application domain, rather than in distributed or parallel computing. Ultimately, this enables the user to concentrate on *what* their application does and *how* it carries out their research objectives, instead of venturing into the administration of distributed resources and tools.

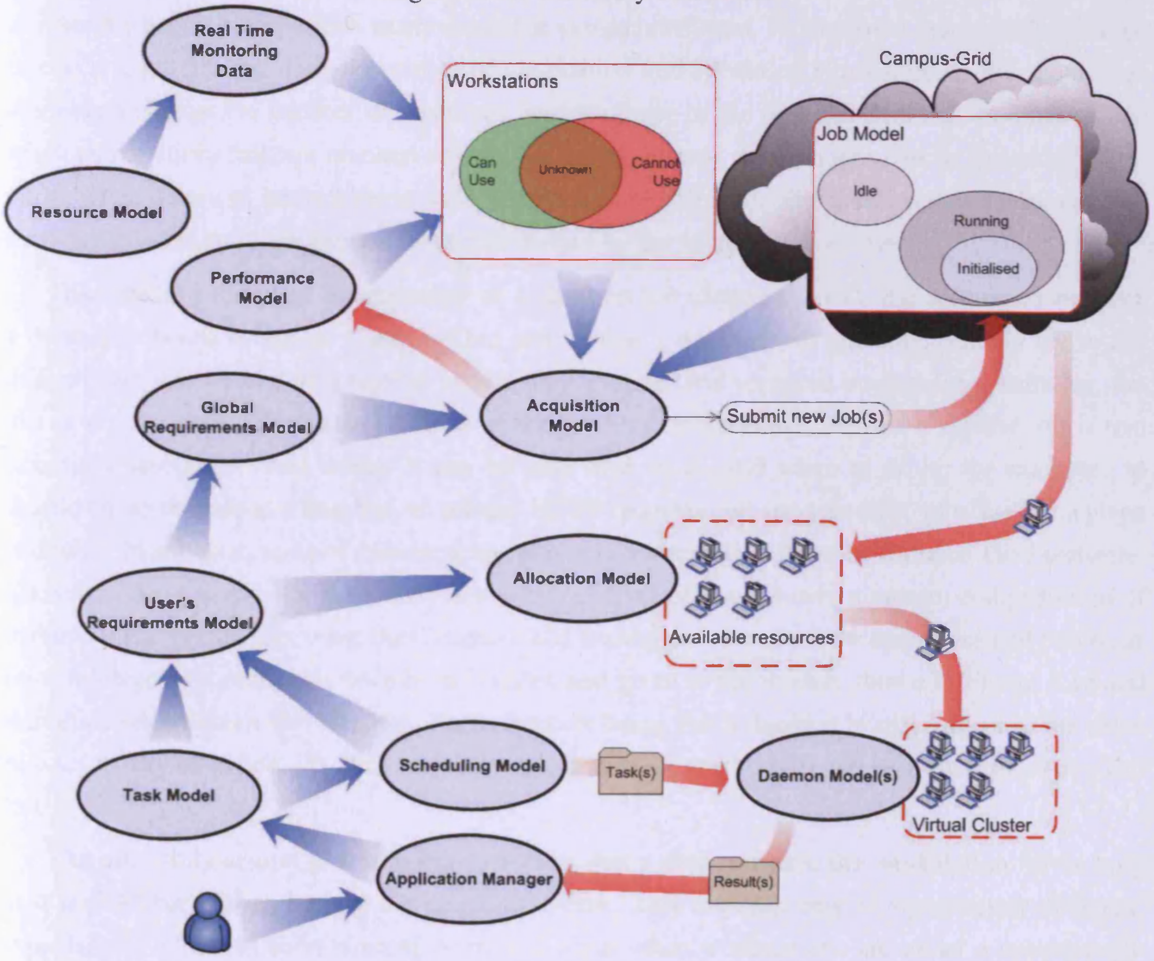
4.10 Summary of Approach

Figure 4.16 shows an overview of all the models described in this Chapter, along with the main interactions and flows of information. It is clear that there are two distinct sections in this illustration, the top and bottom, which represent the separation of resource acquisition (top) and allocation/scheduling (bottom). The summary of this approach will start bottom-up as this is how a user would perceive the approach. Incidently, as the distance[19] of a model from the user increases, the ability of the user to perceive it decreases exponentially toward zero.

The only mandatory action that a user must take is to create a new, or use an existing, Application Manager to describe their Task space. Here, they use three pieces of information to define each Task: **(1)** some control sequence which describes the analysis to be done to the data, **(2)** the data to act upon, and **(3)** an identifer for the result (optional). This data is then fed into the Task Model (Section 4.3.2) to create the Task space. A scheduler will use this information to execute the Task space on one or more Campus-Grid workstations. This information, along with additional (scheduler specific) information from the scheduling model (Section 4.7), is given to the user's requirements model (Section 4.5). The user's requirement model then generates an abstract number or range of workstations to represent the ability of the user's application to consume resources. This representation is then passed to the global requirements model (Section 4.5), which aggregates all user's models to define the ability of all users to consume resources. It also adds an additional buffer so that the system can acquire spare capacity if possible. This information is also passed to the allocation model (Section 4.6). The acquisition model uses the global ability to consume resources as a dynamic horizon relating to the number of workstations which it should acquire. However, several factors influence its ability to do this: **(1)** the number of available workstations, **(2)** the rate of change in workstation availability, **(3)** resource contention and competition, i.e. the number of queued jobs from other Campus-Grid users. This information is provided by the resource model (Section 4.4.1) and real time monitoring data. Yet, one other significant factor remains: the ratio of failures and runaway jobs to successful ones. This information, provided by the job and performance models (Sections 4.3 and 4.4.1), governs whether the acquisition model (Section 4.4) will choose to reject an available workstation or not. If it deems that there are only unreliable workstations available no

[19]Here, distance is the minimum number of hops from the user to a model. For example the user is 5 hops from the Acquisition Model. An inherent property of this concept is that a blue hop is significantly closer than a red one.

Figure 4.16: Summary of Models



daemon jobs will be submitted to the Campus-Grid.

When jobs are submitted to the Campus-Grid, they are different in that they are not normal jobs, but pilot jobs. This means that they are testing the stability of a workstation prior to Task execution. The ability to check a workstation is an hereditary property of the job, as it should never fail. Therefore, if the job fails it can only be the fault of the workstation. This information is then relayed to the performance model for future workstation-based decisions by the acquisition model regarding job submission. Should a job succeed, and therefore demonstrate a workstation's capability, it becomes a resource which can be given to a user by the allocation model. The allocation model determines whether a workstation is given to a user by using their requirements model. The autonomic nature of resource acquisition permits many desirable system attributes. In scenarios that exhibit dynamic resource requirements, demand-orientated acquisition and allocation is possible, as the acquisition model can change the number of resources in accordance to the reported demand. In other words, when workstations fail, are revoked or demand increases, new daemon jobs can be generated. Similarly, when daemons are not being used, they can be terminated. The result is that autonomic and dynamic growth, in respect to the resources *owned* by the system, is possible.

Disregarding the pilot functionality of a daemon job (Section 4.3.1), the difference between a daemon job and a regular Campus-Grid job, is that a daemon job will intentionally try to run infinitively. It is treated as a regular job by the Campus-Grid resource management software, but this is not the case. The main attribute of the job that differentiates it from a regular job is that it is fully steerable. This means it can be told what to do and when to do it; for example: to shutdown, to download a data file, to service HTTP requests and send result(s) of a Task to a given endpoint. In addition, none of these actions require any interaction from the Campus-Grid software. Therefore, the Campus-Grid has only a superficial level of control over a system compromised of daemon instances. In this sense the Campus-Grid is simply a source of raw computational power, as most management processes have been isolated and given to the models shown in Figure 4.16 and introduced throughout this Chapter. The exception being that a daemon is still subject to the same rule set as any other job, i.e. it can be suspended, evicted, sand boxed and its host workstation can fail.

The aim of a daemon to run infinitely means that a user can have the workstation for as long as it is available, rather than for the length of a Task. This duration may be significantly different, especially if a Task is short running or runs at night when workstations are under a substantially reduced demand. The steerable nature of the job also becomes beneficial, as a scheduler can submit Tasks directly to a workstation and sidestep the initialization routines that a regular job must forego (the daemon has already been through this process). The result is that a higher throughput can be achieved, without using more workstations. This also means that the requirements model can begin to tailor its requests for resources to the current context, and that the other models can try to autonomously adapt the number of workstations a user is allocated. The collection of workstations that a user amasses over time can be modelled as a virtual voluntary cluster of dedicated resources. The voluntary is self-evident in the nature of the Campus-Grid, but it must be noted that the architecture is not philanthropic: workstations can, and will eventually, be revoked. However, until such a time the workstations are dedicated to the system, and therefore its user(s). More traditional Campus-Grid approaches can employ a back-to-back scheduling strategy in a similar manner to the scheduling model, but they cannot side-step the initialization routines in the same way as this approach can. Other appraches which perform an interactive use of resources through similar means cannot do so in such a way that would enable the simple experiments run in Chapter 2 to be sucessfully completed. They also do not have the same control over the application environment and require more complex and inefficient communication strategies.

The autonomy of the resource pool provides a platform to address the challenges of having library dependencies, which were highlighted in (Section 2.3). Autonomy in this respect provides a stable platform, using fault identification and management to enable error handling models to counter undesirable workstation behaviour(s). This is achieved by the monitoring processes (Job Model), which relays information back to acquisition and performance models. These models then generate a global picture of the Campus-Grid as it is was and is currently. Here, workstation errors, such as faulty or non existent installations and other infrastructural faults can be detected both reactively and proactively. Reactive detection stems from the monitoring of a job's state transitions (Job Model). Proactive detection is the identification of faults early in a job's execution in order to minimize the number of wasted cycles, which could be used by another Campus-Grid user. The result is the ability to guarantee that, save for any runtime errors, once a workstation has initialized a daemon it will provide computation resources for the entire duration of its availability. In addition, none of these tasks must be performed by the user, enabling them to focus on their research and not how their research is facilitated.

Chapter 5

Implementation

In this Chapter the implementation of the models introduced in Chapter 4 is presented, along with a discussion of the intricacies needed to create a practical software solution for distributed image processing within a volatile Campus-Grid architecture. Four key components will be introduced, each of which enacts one or more of the models which constitute this research. In addition, one utility component, which was conceived only to ease the management of distributed components, will also be presented. Figure 5.1 illustrates these components and also depicts how they interrelate.

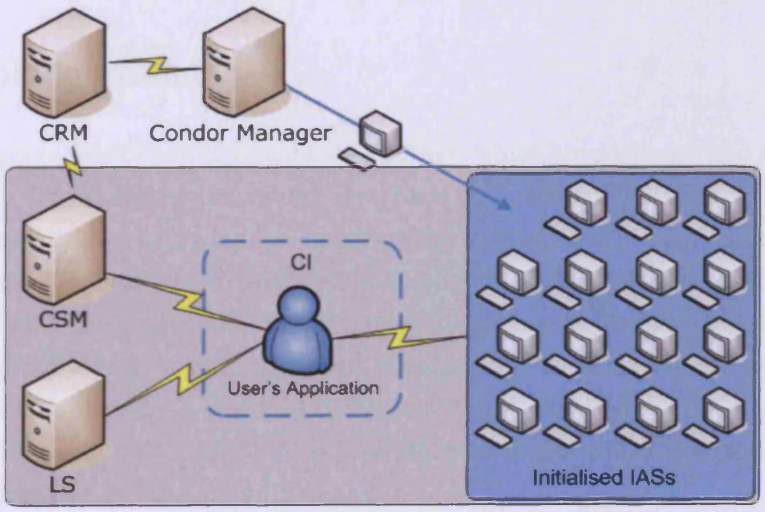


Figure 5.1: Conceptual model, showing the system components. The grey box illustrates the realm of the user's awareness and the blue boxes signifies the user's concept of IAS instances as a single computational unit.

The key components shown in Figure 5.1 are: **(1)** the CRM (Condor Resource Manager, Section 5.3), which represents the Job, Acquisition, Resource and Performance Models (Sections 4.3 – 4.4), **(2)** the CSM (Central Service Manager, Section 5.4), which represents the Global Requirements and Allocation Models (Sections 4.5 and 4.6), **(3)** the IAS (Image Analysis Service, Section 5.2), which represents the Daemon Model (Section 4.3.1), and **(4)** the CI (Client Interface, Section 5.5), which represents the Task, (User's) Requirements, Scheduling and User's Models (Sections 4.3.2, 4.5, 4.7 and 4.9). The utility component, the LS (Lookup Service, Section 5.6), is also shown. Its sole purpose is to withhold a database to enable the key components to locate each other, and system communications to be restored upon failure. However, before introducing these components, an overview of how Matlab is controlled and fulfills the requirements for a daemon back end will be given in Section 5.1.

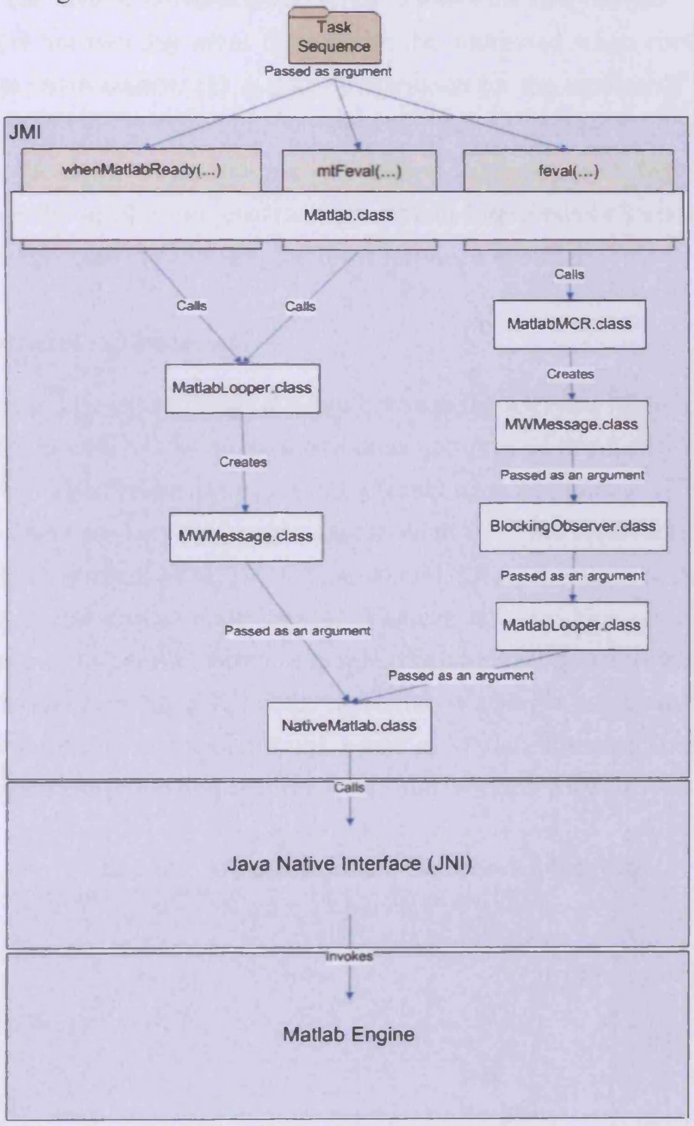
Many different implementation models could be used to build this system: client-server, peer-to-peer, web services and mobile agents are perhaps the main stream possibilities. There is no real reason for choosing one over another, apart perhaps for familiarity of paradigm or a specific advantage with respect to the application domain or research objectives. The implementation documented here has adopted a version of the client-server model with some slight changes to the traditional model, which will become apparent later in this Chapter. In [196] a simplified version of the implementation documented here was demonstrated as a JADE [197] multi-agent system, where user agents could interact and collaborate with Matlab agents to perform image analysis. A web service topology is also possible if a tool such as WS-Peer [198] is used as a container to host the services. Cooper et al. [199] also demonstrated a distributed system for MPI using Web Services; here the only challenge would be to make a servlet container available on the Campus-Grid workstations to host the services.

5.1 Controlling Matlab

Matlab is built upon a foundation of C and C++ libraries, which are packaged into operating system specific binaries. The Matlab graphical user interface is, however, written in Java, and therefore it requires a specific bridge to access the native Matlab engine. This means that all the necessary functionality to control Matlab externally is an intrinsic part of Matlab, it only needs to be harnessed correctly. The Matlab UI components interact with the Matlab engine using the JMI (Java to Matlab Interface) package. The architectural aspects of Java and the JMI mean that all of the foundational building blocks are in place for a complete software solution. Therefore this API can be harnessed by exploiting the hereditary properties of the Java object model to control Matlab from a custom or third party application.

The design of Matlab does, however, elicit one requirement: any Java application that wishes to control Matlab must run on Matlab's JVM. There is also one implementation challenge: the JMI is neither consistent across various operating systems (it is not platform independent) nor through different Matlab versions (it is not backward or forward compatible). This means that every time Matlab is upgraded, any Java-based controller may also need updating. The JMI is also mostly undocumented, which adds to the challenge of harnessing it. Figure 5.2 shows the basic structure and flow of information within the JMI architecture. This is not a complete representation, but aims to give an overview of what occurs. Essentially, the key class is the `Matlab` class (`com.mathworks.jmi.Matlab`). The three method calls shown in Figure 5.2 are the methods that have been used and worked for Windows and/or Linux versions of the JMI. To physically pass Matlab an instruction, a user defined Java application should create an instance of the `Matlab` class and call either one of the methods shown in Figure 5.2 or any of the many other relevant methods.

Figure 5.2: The structure of the Matlab JMI Interface.



To get a result back from Matlab is more challenging. Remember that the JMI is written for graphical output. Therefore results are not needed in a useable format, as they are held within Matlab's memory, enabling the UI to access data by reference (variable name). For this reason results are returned as Strings in the format that they are displayed in the Matlab workspace. When using Matlab from a Java application, such a result format is undesirable, impractical and would require unnecessary parsing and String manipulation to be made useable. Here, the interpreted functional language of Matlab is very useful, as a Task Sequence can also be executed from within

Matlab using an interpreter. Matlab provides several interpreters, but the most common is `eval`, which simply executes a Matlab expression.[1] A Task Sequence can come from a text file, which can be either local or accessed via a URL. Otherwise regular Matlab code can call an interpreter, passing a Task Sequence as an argument. These two basic abilities mean that it is possible to use Matlab to control itself. It also means that an interpreter can be used programmatically, and therefore the output from Matlab can also be controlled and tailored into a useable format.

In summary, there are two key areas that need to be addressed when controlling Matlab, and any other application environment: **(1)** A general approach for the income of Tasks and how they transpire into a call to the application environment's engine. In Section 5.1.1 a control application is defined for this purpose. **(2)** A general but still customisable approach for result generation and error handling within the application environment. Meta-Interpreters (Section 5.1.2) are used to define how a Task is executed once the engine itself has been invoked.

5.1.1 Matlab Control Application

A control application is a means to bridge the gap between the user and Matlab; its main purpose to the make the interaction with Matlab as transparent as possible by displacing the idiosyncracies of Matlab interaction. For all intensive purposes it is a front end to Matlab just like the Matlab desktop and command line. There are very few restrictions to what a control application can be or do, but it must be run on Matlab's internal JVM. Here, it can exploit the open nature of the JMI, and facilitate external calls to the Matlab engine upon request. There is no restriction on the number of control applications that can run in parallel within a single Matlab session. However, only one can have access to the Matlab engine at any given time, as Matlab is a single threaded environment. Failure to adhere to this implementation constraint will result in Matlab throwing a fatal error, which will result in an inelegant thread dump and termination of the working session.

Figure 5.3: Architecture of a Control Application

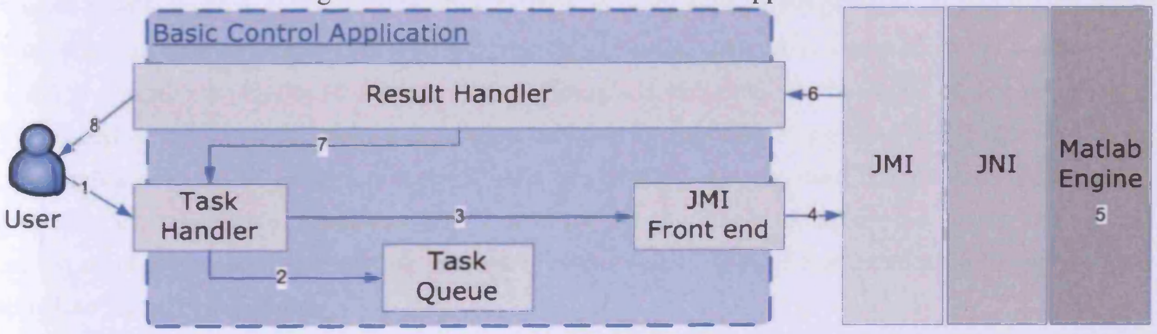


Figure 5.3 illustrates the architecture of a control application. The actions defined in this Figure are: **(1)** The user calls the control application, passing one or more Task Sequences as input. This action can occur at any time during the execution of a Task Sequence. **(2)** If the user has passed more than one Task Sequence, or if the control application is currently executing another Task Sequence,

[1]It can however, instantiate and assign values to variables as part of a sequence.

the new Task Sequence(s) are placed into a queue. **(3)** When Matlab is available, a Task Sequence is passed to the JMI front end for execution. **(4)** The Task Sequence call is generated (this process is explained below) and the engine is invoked using the JMI. **(5)** The Task Sequence is executed by a Meta-Interpreter and the result generated (see Section 5.1.2). **(6)** The result from the Meta-Interpreter is returned to the control application. **(7)** The Task handler is informed that execution has completed, it now checks the queue to determine if another Task Sequence is available for execution. **(8)** The result is returned to the user.

Generating the JMI call is the most error prone aspect of implementing this interface. This is due to the difficulty in determining the correct route of invocation, which is a consequence of the undocumented and inconsistent nature of the JMI. The easiest way to determine what options are available is to use the Java Reflection API, or other reverse engineering tools. Essentially, if the wrong method or input arguments are used Matlab will crash. Once the correct route through the JMI has been determined, Matlab control is very intuitive.

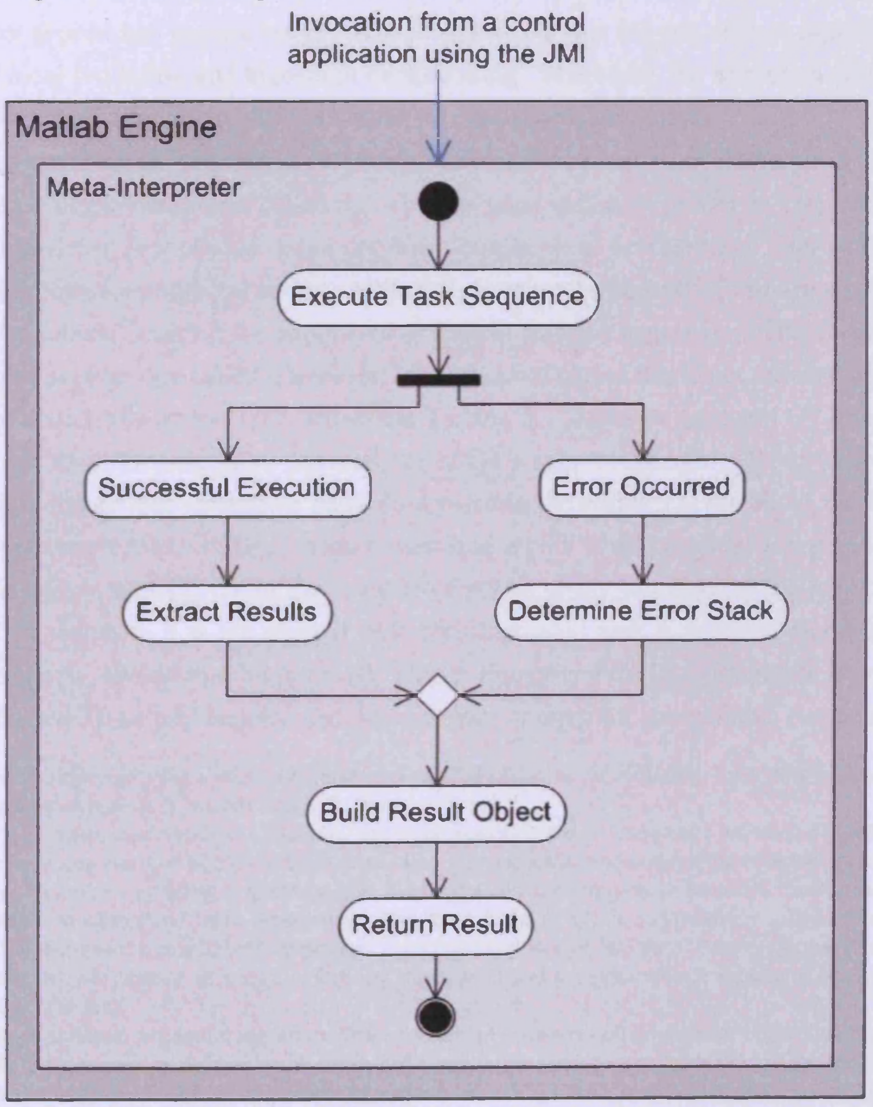
The control application calls a Meta-Interpreter (MI) (a means to standardize Task Sequence execution; Meta-Interpreters will be discussed in greater detail in Section 5.1.2) directly and the Task Sequence is passed as an argument. At this point the means for the MI to return the result is also specified, because the basic JMI functionality is inadequate. The simplest method to use is an event queue within the control application, passing the means to invoke and create an event to the MI as input. This approach also fits nicely into the distributed computing context, because if more than just the control application require the result multiple event handlers could be provided at this point. In addition, the differences in the JMI do not interfere with this approach, which means that this approach improves the platform independence of the interface.

5.1.2 Meta-Interpreters

For an interface such as this, it is desirable that every interaction is treated in the same fashion. This makes results more usable and accessible further on in an application pipeline. It also means that a user does not have to be concerned with how interaction with Matlab is achieved. A meta-interpreter (MI) is a means to standardize how a Task Sequence is executed. In the scope of this interface, it is defined as a Matlab function that takes a function or function sequence (Task Sequence) as an argument, executes it, generates a result in a pre-defined and standard format and, if necessary, performs error handling. Should an error occur, the MI also needs to capture the source and context of the error event, so that it can be presented to the user or used elsewhere within the application pipeline for error handling.

The meta-interpreter is one of the most customizable components of the interface and this customisability stems from its two main roles. Namely, how the Task Sequence is executed and how results are extracted and represented. Such customisability can arise from different sources, e.g. different application scenarios may require results in different formats, others may produce Task Sequences in varied formats etc. However, all meta-interpreters follow the same basic structure and data flow, which is shown in Figure 5.4. There are two important aspects of an MI that are required to enable it to handle a wide range of Task Sequences: (1) the MI is given the necessary means to relay the result back to the control application, (2) the Task Sequence is a correct representation of the analysis to be performed. This may be obvious, but aside from standard lexicographical mistakes, which all users can make, there is another area where more significant mistakes can occur: data handling. How access to data is represented is very important, for example, passing a URL to an MI that is unreachable is an unnecessary error. This is especially important in a distributed computing context, when data may be staged in, and possibly renamed, before the MI is invoked. In such cases the Task Sequence itself would need to be updated prior to an MI invocation. Such eventualities should, in the simple case, be handled by the control application (Section 5.1.1), however, in this approach it is handled by the implementation of the Task Model as a part of the Client Interface (Section 5.5).

Figure 5.4: Overview of a Meta-Interpreter. This Figure is actually a more detailed version of the right side of Figure 4.2, illustrating its connection to the daemon model (Section 4.3.1).



A core assumption of an MI is that regardless of the outcome of a Task Sequence, a result or error will be produced. The reason for this is due to Matlab's single threaded execution environment.[2] If a control application has more than one Task to evaluate, it needs to know when it can perform the next engine invocation. By always returning a result, even an empty one, the control application is made aware that Matlab is ready to execute the next Task. This approach also lets the user know that the Task has executed without error.

Once an MI is invoked, it can begin to execute the Task Sequence, as shown in Figure 5.4. Ideally, this would be a single Matlab function call, but it is unrealistic to assume that this will be the case. Application context is also an issue that must be considered, as different Matlab toolboxes and third party libraries evoke different styles of programming. Similarly, different applications have different levels of execution granularity (Section 4.9.1), which can evoke porting large sections of Matlab code to an MI. QT[3] for example, uses three letter mnemonics to represent an image processing function, where an algorithm consists of a chain of interchangeable function calls. Encapsulated within these functions is all the necessary functionality to render any necessary graphical output. Other approaches require more effort on the user's part for non critical aspects of Matlab, such as graphical rendering and algorithm backtracking. Therefore, the execution strategy that an MI needs to be applicable to the different styles of Matlab programming.

A general approach to Task Sequence execution does pose a low-level challenge in the capture of results. When a single function is called it is obvious what variables should be captured to generate a result. It is however, less obvious what graphical output should be captured. Here, QT provides a useful methodology for graphical output. All functions store the algorithm's image state in the same two Matlab variables,[4] making the capture of graphical output much easier. This is also persistent when multiple functions are called. However, for numerical output this is not the case when multiple functions are called within one Task Sequence. Listing 5.1 shows an example QT Task Sequence, which performs the following steps: **(1)** load one of QT's example images, **(2)** calculate the average intensity of the image and instantiate variable `a` to store the result, **(3)** threshold the image at the average, `a`, to convert it to a binary image rather than a grey scale or colour image, **(4)** isolate the largest white region, and **(5)** count the number of white pixels to calculate the area of the white region. In this sequence it is not clear if only the calculated area is required as output, or if the average intensity, `a`, should also be returned. This is also only a trivial example; in Chapter 6 much more complicated Tasks will be executed, where many more than one variable can be instantiated.

[2] At the time of implementation a multi-threaded version of Matlab was not available. Even now in the multi-threaded versions of Matlab the number of threads is limited to one per core.

[3] QT [200] is an augmented version of Matlab's image processing toolkit. The image processing paradigm of QT allows the user to side step many of Matlab's image processing idiosyncrasies and concentrate on application development. Algorithms are constructed by using interchangeable sequences of function calls in Matlab's functional programming language, and derived from three letter mnemonics. The main focus of QT is to intuitively support interactive image processing and to assist in the development processes for the prototyping of Machine Vision systems. In doing this, QT allows rapid selection and testing of image processing algorithms and provides a large repertoire of easily executable image processing functions.

[4] Two variables are used to capture algorithm progression and enable a certain amount of backtracking. When one function is called, the *current* image (the result of the last function) is placed into a scratch variable, and the function to be called will write its output to the current image. Newer versions of QT have also implemented an image stack.

If this Task Sequence was entered at the Matlab command line exactly as it is displayed, only the results of `cwp` would be displayed. However, if the semi-colons were replaced by commas, the value of `a` would also be presented to the user. Basing the decision over what is returned cannot be orientated around the use of separators. There are many different ways of writing this Task Sequence, using different combinations of commas, semi-colons, carriage returns and even embedding functions as arguments of other functions. Therefore, an MI will use the assumption that only the final functional entity in a Task Sequence will generate results that should be returned to the user.

Listing 5.1: Example QT Task Sequence

```
rni(3); a = avg; thr(a); big; cwp
```

Listing 5.2: Ideal code for getting results for a Matlab command sequence

```
result = eval('rni(3); a = avg; thr(a); big; cwp');
```

In order to execute and collect the result from this Task Sequence it is passed through the `eval` interpreter. Ideally this would be performed as shown in Listing 5.2. Yet this is not possible, for the same reason as mentioned above: Matlab does not known what output value to instantiate `results` with. The solution to this problem is straightforward with the assumption that only the last function will be used to generate a result. Simple string manipulation can be used to identify the last functional entity of a Task Sequence to enable a split execution model, as shown in Listing 5.3. The work around here is that the Task Sequence is executed in two parts, and the result correctly returned, in the second part. However, there is also a limitation with this approach: the MI would need to know the cardinality of the output prior to execution or all functions would need to have the same output cardinality. Such a requirement would make creating a general approach very difficult. Listing 5.4 shows a conceptually identical solution, with the exception that variable instantiation is performed within the `eval` call. This is a useful ability, as the contents of the `eval` call are string-based, which means that they can be created on the fly and at runtime. Therefore, it is possible to generate an `eval` call which can be tailored specifically to the cardinality of a function's output. Matlab also provides the functionality to do this.[5] Yet the basic approach provided by Matlab does not work for all functions[6] and is inadequate in some situations.[7] The simplest way to side step this problem is to inspect the function's declaration, which explicitly states the number of output arguments and their names. It also identifies if a function uses a variable number of output arguments. It is then

[5]See Matlab function: `nargout`.

[6]Functions which produce a variable number of output arguments cannot be inspected in this manner, as the number of arguments is not determined until the function is executed.

[7]When the results of a function are to be saved to disk in Matlab's `.mat` format the variable names are important for two reasons: (1) when the results are saved, Matlab does not retain their order, and (2) when the results are loaded back into Matlab, they are assigned the names given to them when they were saved. Therefore generating random variable names or names with no clear re-mapping are of little use.

just a matter of keeping track of previously read function declarations, which can be facilitated by a lookup table. The benefit of this approach is that the necessary variables can be created on the fly at runtime, and that by concatenating the Task Sequence with the necessary code to capture its output, the right cardinality can also be ensured. The consequence of this approach is that the Matlab interpreter is called twice. Nevertheless, significant performance degradation is not experienced. Listing 5.4 shows only a hard-coded example for one output variable, Listing 5.5 shows (with some pseudo code for ease of presentation) a dynamic programming solution which considers the output cardinality and could also adopt the correct variable name space if this was required.

Listing 5.3: The work around for Listing 5.2

```
eval('rni(3); a = avg; thr(a); big');
result = eval('cwp');
```

Listing 5.4: Syntactically the same as Listing 5.3 but enabling a far more powerful approach

```
eval('rni(3); a = avg; thr(a); big; r = cwp');
result = eval('r');
```

Listing 5.5: An example of extracting multiple result variables by applying a general dynamic programming approach

```
[fs_start, fs_end] = split(function_sequence); %identify the last functional entity
args = outputVariables(fs_end); %returns an array of variable names

if (size(args) == 0)
    %there are no results
    eval([fs_start, ',', fs_end]);
    result = 'no_results';
else
    resultStr = generateResultString(args) %output format - [a,b,c]
    eval([fs_start, ',', resultStr, '=', fs_end]); %concatenate all components and execute
    result = eval(resultStr); %instantiate result as an array of the variables generated
        by the eval invocation.
end

createResultObject(result);
```

To capture visual output, QT's methodology is adopted. If the programmer wishes to return an image as a result, they must store it in either of QT's image variables. The MI will then save these images to disk, using a unique name[8] and return a file handle.

Once the results have been gathered, they must then be represented in a standard format. Matlab transparently assigns types to its variables, but when they are to be converted into Java objects, all transparency is lost. Instead, the Java type to be used must be explicitly constructed, however, this is a straightforward procedure. There are only two types which any application can be concerned

[8]The `rand` function cannot be used here, as it is a seeded random function, which will always produce the same output sequence. Instead, all output files names are based upon the current time in nanoseconds and the function which created the result(s).

with: strings and numbers, both of which are easily identifiable. When it is necessary to return multidimensional data sets it is simply a case of creating a multidimensional array, which is elementary. The output data is then placed into an Object wrapper, along with the relevant file handles for saved images, and returned to the control application, by using the provided means.

When Matlab throws an error, it does so in a very pragmatic manner, by showing the cause of the error, which function was called and where specifically the error was generated (a line number). It then builds an execution stack to illustrate how this function was called. To capture this information, an MI firstly needs to ensure that an error can be caught, and secondly, to represent the error in a useable format. This is achieved through a similar representation to that of regular Java exceptions, a stack trace is created from the Matlab error, and this is then sent back to the control application as the result.

5.1.3 Controlling Compiled Matlab

Compiled Matlab is a very different kettle of fish to regular Matlab. Firstly, and most importantly, it is license free; the licensing agreement even allows compiled Matlab products to be sold without reparation to TheMathworks, with some terms and conditions. Secondly, controlling a compiled Matlab application is not trivial. There are two key reasons for this: **(1)** When a compiled Matlab application runs out of things to do it shuts down, unlike regular Matlab. This is a very undesirable feature, as it would mean that if the application was deployed as a service or as a job on a distributed infrastructure such as a Campus-Grid, it would have to be restarted for each invocation. **(2)** The JMI for compiled applications is even harder to interact with than for regular Matlab. An approach that works for regular Matlab may not work when Matlab is compiled.

Keeping the Matlab engine alive is actually quite trivial, albeit somewhat mischievous. To ensure that the Matlab Component Runtime (MCR) does not shutdown, the initialization routine should open a Matlab figure. When a figure is open, the engine can be left idle indefinitely, solving what could have been a critical flaw in an implementation of the daemon model (Section 4.3.1).

Solving the challenge of the JMI interaction is less straightforward. The cause of this complication is that few of the Matlab components which use the JMI are needed by compiled applications. Nevertheless, the MCR still retains the JMI package, and therefore the functionality is available and operational. The only implementation challenge is to determine which route through the JMI is operational within a compiled context. Windows implementations of the JMI are a lot easier to interact with than those for Linux derivatives and Macintoshes, even for compiled applications. However, in many cases a Windows version of the JMI will work on other platforms, due to the JMI's foundational base of Sun's JNI.

5.1.4 Summary

This interface uses the Java to Matlab Interface (JMI), which is a core part of the Matlab package, to control Matlab. Control is facilitated through two key components: **(1)** a control application, which regulates the invocation of the Matlab engine and provides the bridge for the transportation of results out of the Matlab engine, and (2) a Meta-Interpreter, which provides a standardized approach for the execution of Task Sequences, result generation and error handling. This interface provides the foundation basis for the exposition of Matlab as a computational service. The Image Analysis Service (IAS) presented in Section 5.2 will introduce how this interface can be used to create a remote service capable of interactive Matlab capability, but within a compiled, and therefore license free context.

There are other approaches for controlling Matlab from Java in the literature and the most popular is JMatLink [152]. However it is not known whether JMatLink can control compiled applications. In fact, no other solution has been found throughout the literature that has documented this ability. JMatLink has lessened the difficulty of extracting useable results from Matlab, but it is not yet perfected.

This interface was built for Matlab R2006a for Windows XP and a basic overview has been presented to detail how this interface could be recreated. It cannot be guaranteed that newer versions of Matlab will be controllable with the specification provided here. As mentioned previously, Matlab is neither backward compatible nor platform independent with respect to its JMI architecture. Every effort has been made to side-step this questionable implementation detail of Matlab, but ultimately every new version of Matlab provides new challenges in regard to its control from Java. Typically, the point of failure in a Matlab interface is the choice of method called in the `Matlab` class as shown in Figure 5.2. Once the correct call has been established, the remainder of the implementation is intrinsic. It is also worth noting that older versions of the JMI may work with newer versions of Matlab.

5.2 The Image Analysis Service – IAS

The Image Analysis Service (IAS) is the implementation of the daemon model (Section 4.3.1). The main aim of a daemon, apart from its pilot nature to determine workstation capability, is to be a general computational service for heterogeneous applications. In addition, because software licenses (in this case Matlab) are expensive, it should also be license free. In the context of Matlab, satisfying these two requirements in one software solution is not trivial. This is because typical compiled Matlab applications can only cater for one or few homogeneous application scenarios, can only be used in batch mode and must consequently be initialized for every Task as a new job.

An implementation of the daemon model differs in several ways. Firstly, it caters for many disparate application scenarios, by including entire toolkits and not just the snippets of typical approaches. Secondly, one instance serves multiple distinct Tasks with only one initialization phase. In other words, a control application front end enables interactive steering at runtime. This reduces the impact on performance of unpredictable queueing times within the Campus-Grid and also minimizes Task setup time. It therefore results in a higher job throughput and more consistent Task execution times.

The IAS will be described in two sections, each of which present two different areas of functionality. Firstly, in Section 5.2.1 the process of incorporating Matlab functionality into the IAS and the novelty of this approach is described. In Section 5.2.2 the functionality required to make the control application (Section 5.1.1) capable of running in a remote context is presented.

5.2.1 Incorporating Matlab Functionality

The key to facilitating heterogeneous applications is the inclusion of functionality. The obvious solution in a Matlab context is to include entire toolboxes to increase the range of available functionality. However, compiling entire Matlab toolkits requires a very different approach to compiling a typical application. Normally the compiler only needs to be told which function to compile and in the majority of cases all dependencies are handled transparently by Matlab for the user. The result is a very intuitive and useable interface to the Matlab compiler; Listing 5.6 shows an example. The compiler cannot handle all dependencies transparently: if a function relies on precompiled Windows Dynamic Loading Libraries (DLL), a static library (LIB) must be available to enable the compiler to create the necessary bindings. If one is not available it must be generated before compilation is performed.

Listing 5.6: Example call to compiler for the compilation of a typical Matlab application

```
mcc -m myFunction
```

Compiling an entire toolbox or multiple toolboxes is not as straightforward, as a "*head*" function to pass to the compiler, functions such as "`myFunction`" in Listing 5.6 seldom exist.[9] Without such a function the compiler cannot build the list of dependencies needed to compile the toolbox. Consequently, the list of functions to include must be determined prior to the compiler call. The compiler will handle most dependencies beyond this point. In addition, a head function must be generated to: **(1)** initialize the control application, and **(2)** open a Matlab figure to keep the engine alive in idle periods.

The consequence of compiling entire toolboxes to create a general daemon is that the length of the compiler call soon becomes too long to be managed by hand. For example, in Chapter 6 QT [200] is combined with PRTools [201] and DIPLib [202] to support the neurological analysis of Magnetic Resonance Diffusion Tensor Imaging (MR-DTI) data. The call to the compiler for such a daemon is in excess of 60,000 characters in length. In addition, the compilation also required over 120 LIB files to be created, for the compiler to build the necessary bindings for the DLLs of DIPLib. Naturally, the generation of the compiler call cannot be performed by hand. In addition, manually performing the LIB generation is tedious and is therefore automated using a freely available tool DLL to Lib is available from Binary Soft.[10] This is achieved by creating a project file and calling

[9]Even when a head function does exist for a toolbox, it cannot be guaranteed that it will encapsulate the sufficient dependencies to identify the complete toolbox functionality.

[10]`http://www.binary-soft.com/dll2lib/dll2lib.htm` last accessed May, 2008

DLL to LIB from the command line, using the project file as input. Essentially, the project file contains a standard HEX header, followed by the length of the filename, two default startup arguments, the name of the file to produce (which must be the same as the input DLL only with a different extension) and finally a standard footer. The same process is then repeated for each DLL where a LIB is not yet available. Once a LIB is available, the compiler can then use it to link against the DLL it represents at compile-time; the LIB is not needed after compilation.

The solution for the compilation process has been to create an Autonomic Compiler (AC), which can be implemented in any language that Matlab can run. The AC in this instance has been written in Java with a Matlab wrapper function to call it. Essentially, the user supplies a list of directories which contain the toolboxes they wish to incorporate into the daemon. They also indicate whether the directory tree should be recursed or not, provide a list of functions to exclude and finally the name of their head function. Exclusions can be represented in different formats, for example, a regular expression, canonical path name or simply the name of the function to exclude.[11] Finally, the AC will then traverse all directories and search for Matlab functions, which pass the exclusion rules. It will also identify any DLLs and create a list of those which do not have an associated LIB. The AC will then generate any required LIBs using the above method and generate the compiler call. The `eval` interpreter is used to execute the compiler call, which is structured as shown in Listing 5.7.

Listing 5.7: Structure of the automatically generated compiler call

```
mcc -mv headFunction list of functions and DLLs
```

When dealing with this magnitude of functions, it is not uncommon to encounter functions, which are synonyms and could confuse the compiler. To combat such events all functions are referred to explicitly using their absolute path. Similarly, the AC will discover functions which contain erroneous functionality, illegal characters or that do not have valid function descriptions, all of which cause the compiler to fail. These are typically test scripts, which are not required within the IAS. For this reason compilation inevitably requires some supervision. When multiple functions cause the compiler to fail supervision can become tedious, as Matlab only reports one error per compilation attempt. The compilation process is also time consuming and for an IAS of the magnitude described in Chapter 6 requires about five minutes per attempt. To prevent the user from having to upgrade their exclusion list by hand after each attempt, the AC employs a dynamic exclusion list (self configuration) based upon faults it has experienced whilst trying to execute the compiler call.[12] This list represents every function that has caused the compiler to fail, so that

[11]Note that if only a function name is supplied for exclusion, all functions with this name will be excluded.

[12]To pin down the cause of a compilation error is not as straightforward as it would be expected. The output error from the compiler defines only that it has failed and the source of originating call, but not which function caused the failure. This is displayed on the Matlab command line as an error message, but it is not encapsulated into the error event. To capture this information, the AC uses the compilation flag v (verbose) in conjunction with the Matlab `diary` function, which records all output displayed at the command line in a text file. Regular expressions are then used to identify the most recent compiler error and its cause. The cause is stored to inform the user of the reason for the failure and enable

it can regenerate the compiler call after each failure to remove the problematic function. It also means, the AC can continue trying to perform the compilation and present all faults to the user at once, who can then make the necessary corrections to all functions, or simply delete the function if it is superfluous. The AC will stop the compilation process to present the code faults to the user either: if a successful compilation has occurred, or if a function cannot be excluded, for instance if another function is dependent on it.[13] After a successful (or failed) compilation the AC will update the Matlab wrapper function which called it[14] to include the exclusions it needed to perform the compilation. By employing this approach, compilation can become a background process enabling the user to perform other tasks either until the compilation process has completed, or until the AC requires user support. Thereafter they can perform all the necessary editions to important functions and invoke the AC again for the final compilation. Using this approach even the most challenging toolboxes have been compiled with little or no user effort.[15]

Other useful tasks can also be performed during the generation of the compiler call. During the introduction of the MI architecture (Section 5.1.2) the importance of knowing both the cardinality of a function's input and output as well as the names ascribed to the output variables was presented. The compiler call is built by identifying every function of a toolbox that the user could conceivably call. Therefore, by inspecting the function declaration of each function, a complete database of function cardinality and mapping of output argument names can be generated for use by the MIs. After the compilation process has been completed the AC can also be instructed to upload the generated application, with all the required daemon code, to the Campus-Grid submission node (CRM Section 5.3) for transparent distribution across the Campus-Grid.

5.2.2 Daemon Functionality

Once the necessary functionality has been selected by the user and incorporated into a compiled stand-alone Matlab application, a front end is required, so that at runtime Tasks can be received, executed on the Matlab engine and the result returned. For this purpose the general control application introduced in Section 5.1.1 needs to be extended, so that when it resides on a Campus-Grid workstation the core components can interact with it. To achieve this the basic control application is extended to encapsulate the necessary daemon characteristics: **(1)** a heart beat for monitoring, **(2)** a front end for remote steering, **(3)** the ability to disseminate Tasks to identify data dependencies, and **(4)** the ability to host result data.

Figure 5.5 shows an overview of the constitutional IAS modules and the basic flow of information for Task execution. The sequence of events are as follows: **(1)** A Task is received by the IAS. **(2)** The Task is passed to the Task handler (control application) where it is analyzed for data dependencies. **(3)** Data (if necessary) is downloaded and the Task is placed into the daemon's queue.

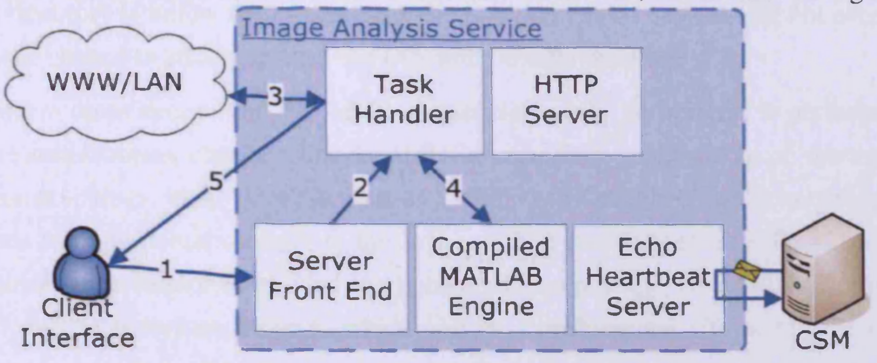
them to suitably modify the function in question.

[13]In this case the AC will open the function in the Matlab editor and inform the user of the cause of the failure. At this point all other functions causing the compiler to fail can also be displayed.

[14]This is detected by accessing Matlab's function invocation history.

[15]This also includes the SPM [203] toolbox, which the end user believed to be uncompilable.

Figure 5.5: The architecture and information flow of the IAS, the left side of Figure 4.2



(4) When the Matlab engine is available the Task handler generates the Matlab call and invokes the engine through the JMI (as described in Section 5.1.1). After execution the result is received and modified to include the endpoint of the HTTP server so that result data can be downloaded. **(5)** The result is returned to the CI, which may then interact with the HTTP module directly to download results. There are two basic result formats provided by the IAS: **(1)** an array (as defined by the control application in Section 5.1.1), and **(2)** a Matlab `.mat` format of results. Here, the lookup table of output cardinality and variable names generated by the AC is used by an IAS MI to define the result representation for the `.mat` file, as shown in Listing 5.5. In the latter case the result is a URL to the `.mat` file, hosted by the IAS instance's HTTP server.

The daemon front end is a basic server implementation, which uses a singleton Task handler to process and queue received Tasks. In a regular client-server model the client retains an active connection until a response is received. This is not the case in this implementation, due to the potential that multiple communication channels (one for each Task in the daemon's queue) be open for long periods of time. Such an implementation could lead to high communications overhead and the daemon front end can become a performance bottleneck. Similarly, on the client workstation, if multiple communication channels are open to all IAS instances bandwidth is wasted unnecessarily, and will limit performance as the number of IAS instances *"owned"* by a scheduler increases. In addition, having more open sockets relates to a higher memory consumption (from the required buffer space). It is also more difficult for the operating system to identify a free port number for the communicative action. For this reason, the Task model was extended to encapsulate the necessary information to enable a daemon to reestablish the connection with the CI to return the result once it is available (see: the CI's results server in Section 5.5.2).

Perhaps the key area of functionality for a daemon is its pilot routine. A daemon job needs to demonstrate the capability of a workstation to perform Matlab jobs. The issues of resource volatility are the cause of this necessity and were identified in Section 1.3.1 and Chapter 2. In order to run a Matlab job, the MCR needs to be correctly installed. In a large number of cases all or part of this library was missing. Therefore, the quickest way to (initially) test a workstation is to quantify the size of the MCR directory. All MCR-ready workstations share the same installation process. Consequently, the MCR should always be in the same place, save for unannounced changes to the

Campus-Grid installation structure, which are not always seldom occurrences. If the measured size of the MCR directory is below the known size, the daemon's pilot routine will not even permit the workstation the chance to attempt to run the IAS initialization routine.

If the pilot routine deems that the MCR installation *could* be correct, it performs any other necessary pre-initialization checks. The Cardiff Campus-Grid is incapable of running Java jobs under the default settings, in fact workstations as a whole can struggle to run Java applications. This inability stems from perpetual changes to the Java run time environment installation, and therefore the Condor installation cannot be pointed at a persistent Java runtime. As a fallback a spare daemon initialization routine is written in Java, which can be a problematic. Typically the initialization routine uses a pre-installed utility to unpack (unzip) the IAS, yet sometimes this tool is not available and a backup approach is needed, which is written in Java. The lack of a trusted Java availability also prevents the daemon pilot routine and initialization from being written in Java, and the result is a 200 line windows batch script. To alleviate the unpredictable nature of Java's availability, the daemon's pilot routine locates a version of the Java runtime environment and adds its location to its session Windows path. To locate a Java installation, the pilot routine first searches the Windows Registry for the latest installation of Java. If one is not found, it then probes the most likely locations (`C :\Java` and `C:\Program Files\Java` for example) of the workstation's hard drive for Java installations. Finally, if it still cannot find an installation it will use the MCR's Java installation. The selection of the MCR runtime is not made first because it is likely to be older than those available on the workstation and specifically modified for Matlab. The IAS can now be unpacked and the Matlab stand-alone executable invoked regardless of the state of the workstation.

The startup process of the daemon involves instantiating the server front end (and a HTTP server) and then the discovery of the CSM (Section 5.4) by using the LS (Section 5.6). As soon as an IAS registers its presence on the system (i.e. it achieves the *initialized* state) it demonstrates a workstation's ability to perform Compiled Matlab jobs. If this state is not reached, the CRM management strategies (Section 5.3.3) will terminate the IAS job. To announce its successful initialization the IAS daemon, after it has located the CSM (using the LS Section 5.6), will also inform the CRM that it has successfully initialized (again using the LS). When such a request is received, the CRM updates the job model (Section 5.3.1) and records the Job ID, workstation ID and elapsed time in the performance model's database (Section 5.3.3).

When an IAS is allocated to a user by the CSM (Section 5.4.2), it commences internal monitoring of activity, to identify when it is not being used sufficiently. The idea behind such an approach is to inform the CSM of improper use of its services, which may stem from too many IAS instances for the user's scheduler to handle or some error may have occurred causing the CI or its scheduler (Section 5.5.3) to crash. Currently, this process is orientated around idle time, where if an IAS is idle (has no Tasks), but is allocated, it will inform the CSM after a predefined period of "*idleness*" has elapsed. The CSM will then retract the allocation of the IAS and the user's scheduler is informed. This process is in place primarily for two reasons. Firstly, to identify discrepancies between the claims of a scheduler in its ability to consume resources (the user's requirements model: Section 4.5) and secondly, in future work where multi-user scenarios are considered (see: Section 5.4.2). If such cases were to arise the IAS instance could be allocated to another user.

5.2.3 Summary

The IAS is a remotely steerable interactive computational service, capable of servicing heterogeneous Matlab applications through the incorporation of one or more entire toolboxes. It is the only known implementation of an interactive compiled Matlab-based service which is built from Matlab itself and not a third party or open source foundational context. As a result a user can retain much of their "*normal*" operational context and therefore reduce the challenge of portability (Section 1.3.7). Similarly, due to its context within the Campus-Grid environment and architecture, the user does not need to be concerned about how it is controlled or communicated with (see: Section 5.5). Therefore, virtual clusters of IAS instances can be modelled as a black box.

5.3 Condor Resource Manager – CRM

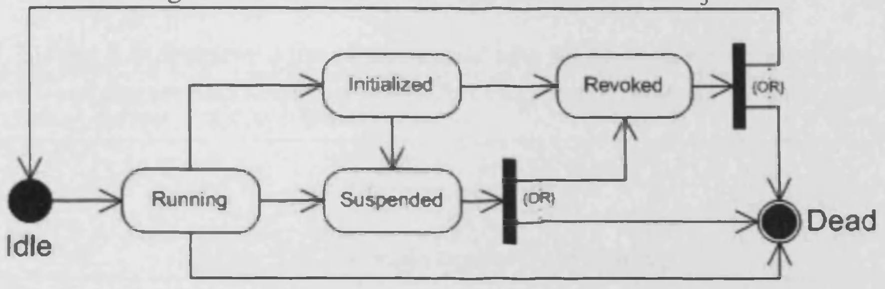
The CRM is an implementation of the Acquisition, Job and Resource Models (Sections 4.4, 4.3, and 4.4.1) or the top half of Figure 4.16. Its key roles are: **(1)** the submission of the IAS daemons introduced in Section 5.2 (the Acquisition Model: Section 4.4) to construct a virtual cluster of dedicated, but voluntarily supplied Matlab-enabled resources. This process is presented in: Section 5.3.5, and involves the monitoring of the Campus-Grid as a whole (Section 5.3.2) to determine whether more IAS jobs could be supported. **(2)** The management and monitoring of IAS jobs (the Job Model: Section 4.3). This process is presented in: Section 5.3.1. **(3)** Workstation profiling based upon previous performance (the Resource and Performance Models: Section 4.4.1). This process is presented in: Section 5.3.3.

5.3.1 Job Model

To build the Job model the CRM uses the Condor logs as a primary source of input. The logs encompass which events occurred in a job's life and when. For example when the job was submitted, started running, changed its memory consumption, if it was suspended or if Condor lost communication with the host workstation and if it terminated what the exit code was. This is the same approach as that used in Chapter 2, however, for the CRM the Condor logs by themselves are insufficient, as they cannot identify the new states introduced for the IAS and therefore this additional information is stored externally. This information is stored in a MySQL database for speed, but could be any other suitable format. The combination of the CRM custom monitoring and the logs enables the CRM to construct a state space for every job that has been submitted to the Campus-Grid by the CRM. Figure 5.6 shows the possible states for an IAS and their transitions.

The job model is a primary source of input for the performance model (Performance Model: Sections 4.4.1 and 5.3.2), where it provides real time information concerning the time consumed

Figure 5.6: Possible state transitions of an IAS job



for the state transitions of an IAS instance for each workstation acquired by the CRM. Naturally as a state space, data mining tools can also be employed to establish how long on average a job needs to commence execution once submitted, the average job life time and the time required to progress from running to initialized, all of which are possible at both an individual workstation and global level. Therefore, the CRM can estimate how long state transitions should take, and terminate jobs (as specified in Section 4.4.1) when these expectations are not met. Similarly, more complicated statistical models could be employed.

To extract data from the Condor logs, the first three characters of each line are processed to identify a Condor state transition. If a line begins with a three digit number a new state transition is documented, where the number identifies the type of transition. For each transition the Job ID[16] and event time are recorded, in some cases other additional information is also recorded. Regular expressions are used to extract the exact contents of the event.

5.3.2 Resource Model

In order to prevent the CRM from submitting too many jobs it monitors the Campus-Grid as a whole, which enables it to build up a general picture of the current state of resource utilization and availability.

Campus-Grid Monitoring

Campus-Grid monitoring identifies the current state of the Condor pool in real time. For example how many jobs (from all users) are idle or running, how many workstations are in use, how many are idle and of those which are idle how many are reliable MCR-enabled workstations. This monitoring information allows the CRM to determine how many IAS jobs can currently be supported by the Campus-Grid. The data for this analysis is collected by parsing the command line output from `condor_status`. Listing 5.8 shows an example of compiling a list of workstations that are both unclaimed[17] and MCR-enabled, suppressing the output to only the name of the workstation and its RAM. Listing 5.9 shows example output of this. Note the use of different naming standards in this

[16]In other implementations the programmer should consider that a Job ID is **not** globally unique and is also repeatable. The Condor command `condor_clean` will reset the ID counter.

[17]A workstation which is both idle (a Condor node) and available for a Condor user to use.

Listing, preventing the construction of an intuitive workstation name space for the CRM.

Listing 5.8: Retrieve a list of unclaimed idle MCR-enabled workstations

```
condor_status -constraint HAS_MATLAB_V74==TRUE -constraint State==\''Unclaimed\'' -format
    ''%s,'' Name -format ''%s\n'' Memory
```

Listing 5.9: Example output from Listing 5.8

```
...
ENROL20,2038
ENROL22,2038
ENROL25,2038
ENROL28,2038
X0007E9DED939,1022
X000CF1FD5B85,494
X000CF1FD5BEF,494
X000CF1FD5CD5,494
X000CF1FD5D4C,494
X000CF1FD5DF0,494
...
```

By simply counting the number of lines, the CRM can determine the number of available workstations. However, by itself this information is quite meaningless due to potential mistakes in a workstation's ClassAd. Therefore, the CRM can query the performance model for each available workstation to ascertain the number of workstations that could realistically host an IAS job. By employing the performance model in this way, the CRM can enrich the information provided by Condor through the use of past experiences at the workstation level. This ability enables the CRM to decide if there is any purpose in submitting new IAS jobs, based upon its analysis of whether one or more workstations are available that have a history of successfully hosting an IAS job.

Workstation Database

The CRM maintains an active database of every workstation to have ever connected to the Condor pool during a CRM status poll, or to have run an IAS job. Given the transient nature of a workstation's membership to the Condor pool, the size of this database does not adhere to the actual size of the Condor pool. It, in fact, contains between two and three times the number of entries than the official Cardiff University figures concerning the size of the Condor pool. Essentially, this shows how much the day to day administration of the Campus-Grid affects each workstation's status within the Campus-Grid creating the dynamic and volatile environment which was outlined in Chapters 1 and 2.

To build the base of the resource model, the ClassAd for each encountered workstation is parsed to extract the necessary data and key attributes outlined in Section 4.4.1. Namely, the workstation identifiers: IP address and DNS name and the NetBIOS name map, which is provided by output such as that shown in Listing 5.9. When the CRM discovers a NetBIOS name that is not present in its database, it attempts to acquire the ClassAd for this workstation. It is not always possible to receive the ClassAd upon request due to the workstation leaving the pool, or simply refusing to transfer it. Therefore, gaps in the CRM's Campus-Grid knowledge are always present. The database can encompass any number of the ClassAd attributes, but currently nothing more than the workstation owner[18] and identifiers are necessary.[19] The database does not retain whether the workstation announces MCR capability or not, as this feature is too dynamic and unreliable. Instead the performance model captures all successful Matlab jobs and failures,[20] in order to depict a workstation's current state (or at least its last known state). It also shows a workstation's stability with respect to state switching. This is a much more powerful approach, as a profile of each workstation can develop over time. In addition, the last recorded interaction can be modelled as a measure of confidence with respect to the accuracy of the CRM's knowledge of each workstation. For example if a workstation has a solid history of success or failure it only needs a pilot job once a day to confirm its capability. However, other workstations will require many more pilot jobs for the performance model to capture their state and its variability. In this respect the performance model provides the foundation for (simple) learning techniques, which aid the decision making processes of the CRM.

The CRM polls the Campus-Grid hourly, to ensure that any unknown workstations are recorded. The basic individual workstation record is updated daily, in order to confirm that the current information is correct.[21] This process for adding and confirming a workstation's base record is the same, with the exception that where a new record is added, an existing one is updated if a discrepancy or change is detected. This process is shown in Figure 5.7 and can be summarized as follows: firstly, the CRM identifies all machines currently connected to the Condor System (using a `condor_status` call like that shown in Listing 5.8) this returns a list formatted like that shown in Listing 5.9. Secondly, for each workstation the CRM downloads the ClassAd to local disk. This is performed by directing the output of `condor_status -long WORKSTATION_NAME` to a text file. The naming standard for the output is in accordance to the NetBIOS name of the workstation, as this provides a basis for identification and ClassAd validation. Finally, each attribute needed for the database is extracted from the ClassAd and the database is updated accordingly. Remember that the NetBIOS name is needed as Condor identifies a workstation for blacklisting using this name, but identifies a workstation in the job logs using its IP address.

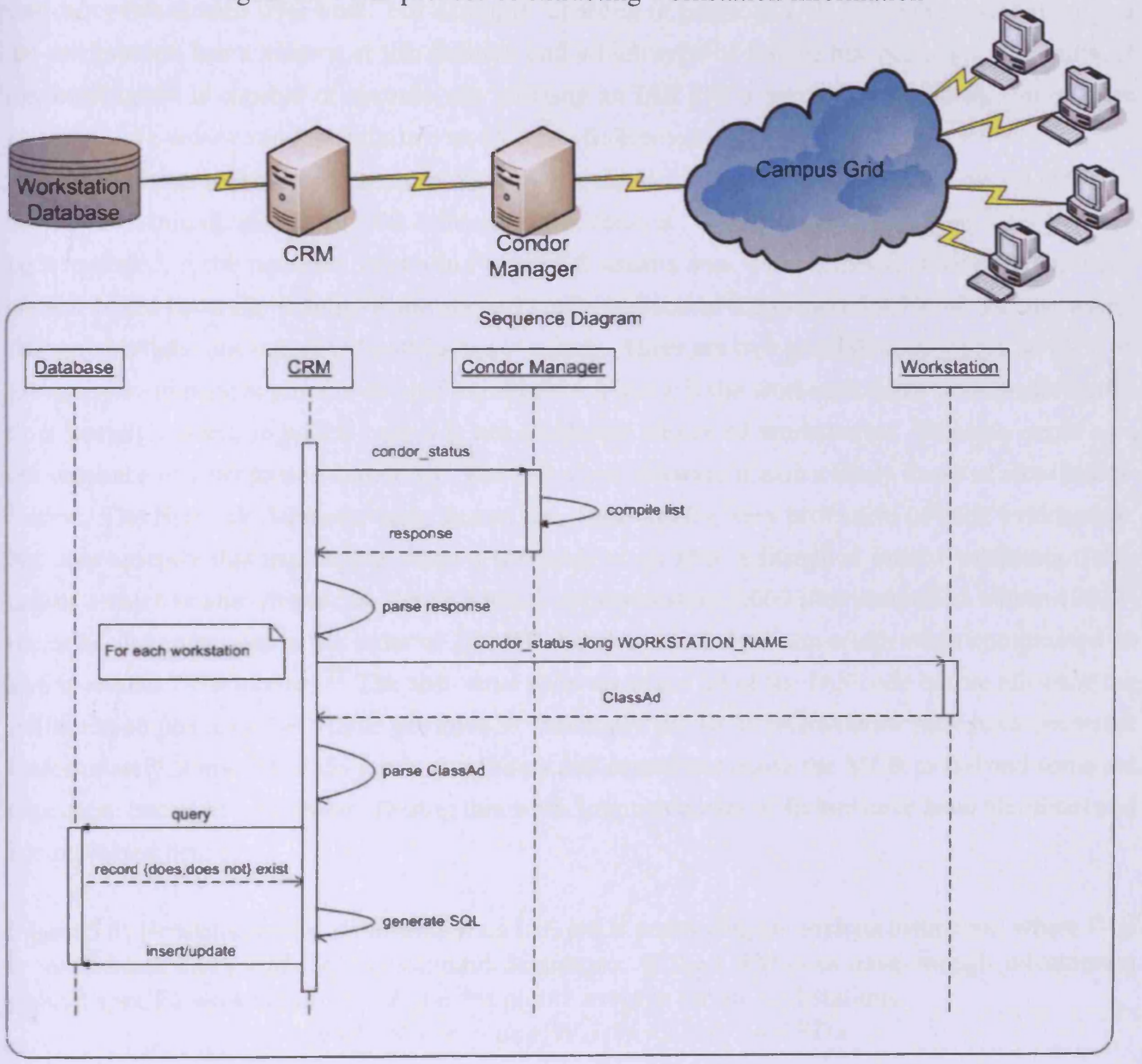
[18]E.g. COMSC: School of Computer Science, OAWS: Open Access Workstations, which are typically in libraries and open computer suites. OAWS can be suffixed with extensions relating to which building the workstations are in. OAWS_CARBS, for example, are workstations in the Cardiff School of Business, BIOSC: School of Biosciences etc.

[19]In future work this could be extended to encapsulate all or larger portions of the ClassAd and other observed behaviours, so that the CRM could act as a Campus-Grid Information Service [177]. The benefit of this would be that when no other service such as Condor's Quill or HawkEye is present, the CRM can take their place. The benefit of a GIS is that it can supply schedulers with additional information when they are in the planning stages. It could for example indicate, how long on average a given workstation is available, its last benchmark scores etc.

[20]For example the data from Section 2.3 could be used as a training set for the performance model.

[21]Workstations can change on a daily basis, as observed in Chapter 2, with respect to their state and performance benchmarks. However, it is also not uncommon for IP addresses to be reassigned and therefore the CRM must capture this information for consistency. For example the analysis performed in Chapter 7 was interrupted by an overhaul of the network topology as the as the network was migrated to a layer three network. Here, the IP address of most workstations where migrated from a 131.251.*.* address to a 10.*.*.* address. Consequently, the CRM's database was overnight drastically inaccurate as the NetBIOS name to IP address map was made redundant. It required three months for the CRM to rebuild the map to same level of extent.

Figure 5.7: The process of extracting workstation information



5.3.3 Performance Model

The necessity for strong fault management in the domain of this research has been motivated for in Chapters 1 and 2. The aim of the performance model (Section 4.4.1) is to provide a foundation for the CRM to avoid and identify the different scenarios where resource volatility affects job management. The Job and Resource Models (above and defined in Sections 4.3 and 4.4.1) provide a foundational basis for adaptive and dynamic fault management and self-configuration of the system as a whole. These two models provide all the necessary information to construct a model of a workstation's performance over time. For example, analysis of previous IAS jobs can indicate firstly, if the workstation has a history of job failures and which type of failure has occurred. Secondly, if the workstation is capable of successfully running an IAS job, a prediction of the maximum time required for a workstation to initialize an IAS can be created.

The decision making process employed by the CRM is based on the data from over 5,000 successful IAS initializations for 600 different workstations. Based upon this data an IAS job will be terminated, if the heuristic shown in Figure 5.8 returns true. Five standard deviations has been chosen based upon the results of the second study in Section 2.3, where 99.7% of the successful Dummy Matlab Jobs initialized within these bounds. There are two possible reasons for an IAS job to require an excessive amount of time to initialize. Firstly, if the workstation has poor performance or is heavily loaded, in which case it is not a suitable choice of workstation. This can occur as a consequence of workstation autonomy. The anti-virus software is also a likely cause of slow initialization. The Network Administrative Teams are, quite rightly, very protective of each workstation, but consequently this means that when a job such as an IAS is launched onto a workstation this causes a reactive anti-virus scan. An IAS job is approximately 3,000 files embedded within 100 directories and consumes in the order of 20 MB of disk space (when unpacked, when compressed an IAS is around 7MB in size).[22] The anti-virus software scans all of the IAS code before allowing the initialization phase of the IAS to commence. Secondly, an MCR related error may have occurred. Unfortunately, a myriad of disparate conditions and events can cause the MCR to fail and some are a lot more tractable than others. During this work four categories of failure have been identified and are explained below.

Figure 5.8: Heuristic used to determine if an IAS job is portraying anomalous behaviour, where WS is workstation name and SDs is standard deviations. If the CRM does have enough information about a specific workstation, it will use the global average for all workstations.

$$JobRuntime > avg(WS, initTime) + 5SDs$$

Straight MCR failure

Straight MCR failure is the easiest to diagnose and detect. The name is derived from the nature of the failure; the effect is instantaneous and happens as soon as initialization commences. Straight

[22]These of course increase as the range of Matlab functionality increases. In this is the IAS instance that is used in Chapter 6.

failure occurs when a critical MCR component is missing or corrupt (typically `mclmcrrt74.dll`), the MCR has not been installed (ClassAd error) or the Windows search path is incorrect. The effect of such an error is that a compiled Matlab application will terminate with the exit code: `-1073741515`. This exit code along with any others is caught by the CRM's Job Model.

Discrete MCR failure

Discrete MCR failure is more difficult to diagnose; here failure occurs because a subsidiary MCR component is missing or corrupt. In this case an error message is displayed as a Windows dialog. This is not a good situation for a job running on a remote virtual desktop, as the job will not naturally terminate until a user clicks `ok`. Until this occurs the CRM has no indication that the job has failed, only that the initialization state has not been reached. To identify such a failure the heuristic shown in Figure 5.8 is used to decide when to terminate the job.

These first two scenarios are mostly captured and prevented by the pilot routine of the IAS (Section 5.2.2), which performs an MCR integrity check to identify straight and discrete MCR failure prior to the launch of the compiled application. This is typically a more efficient approach than allowing the MCR to fail and reduces the number of wasted cycles on a faulty workstation. The integrity check requires between 2 and 222 seconds (as will be shown in Section 7.6), depending on workstation load, but on average requires 20 seconds. When the pilot routine fails, the job will terminate with the same exit code as above. The worst case scenario is that the MCR check succeeds, but the installation is faulty or corrupt. In such scenarios the failure heuristic (Figure 5.8) identifies a discrete failure. In such a case the workstation would be blocked unnecessarily most five standard deviations, which is seldom more than eleven minutes. This can occur as the pilot routine only identifies variations in size of the MCR as a whole and a corrupt MCR installation does not necessarily have a different total size or number of components. This is effectively a trade off between the time required to analyze the MCR and the accuracy of the analysis.

If the job model discovers this or another an unexpected error code, the performance model assigns 1 fail point to the workstation. If a workstation receives more than a predefined number of consecutive fail points it is blacklisted for 24 hours for every subsequent point. During this period of time it is assumed that the workstation will be restarted in accordance to Cardiff University's green computing initiative. Upon restart the network management software will have the chance to remedy any software abnormalities, but in reality it may not. Therefore the CRM will give the workstation the chance to demonstrate a change in state, but only once per day, when it has a history of failure. When a workstation is blacklisted, the IAS submission script generator (Section 5.3.5) will signal that the Condor matchmaker should not consider it for IAS jobs. Listing 5.10 shows an example of this.

Abstract/Feigned MCR failure

Abstract or feigned MCR failure derives its name from its cloudy nature. Currently, no definite reason or cause has been identified for the errors encapsulated by this category. Here, the MCR

Listing 5.10: An example of a black listed machine

```
requirements    = $(owners) && $(blacklist) && $(IAS)
owners          = IS_OWNED_BY == OAWS || ...
blacklist       = machine != ENROL20 && machine != X000CF1FD5DF0 ...
IAS             = HAS_MATLAB_VXX==TRUE
```

initializes for inexorable amounts of time. In Chapter 2 this failure type was identified as a runaway job, and the longest recorded time for a runaway IAS job required to reach the initialized state was over 3 days.[23] Errors in this category have the symptoms of discrete failure, but the key difference is that the MCR has **not** terminated. During extensive testing it was not possible to recreate a runaway job within a controlled environment. Such failures are dealt with in the same way as discrete failures, i.e. through the application of the termination heuristic (Figure 5.8).

Post suspension failure

Post suspension failure is a reasonably well documented issue with compiled Matlab jobs on Condor. When Condor suspends a Matlab job the Matlab engine is interrupted and paused, which is fine. The issue, however, is that when Condor `unsuspends` the job, one of two things will happen. Either the Matlab engine will experience a critical thread dump and crash silently or it will terminate cleanly, providing a stack trace for debugging. Regardless of which route is taken the same job exit code is returned: -143. The exact same error will occur if a control application (Section 5.1.1) attempts to multithread Matlab.

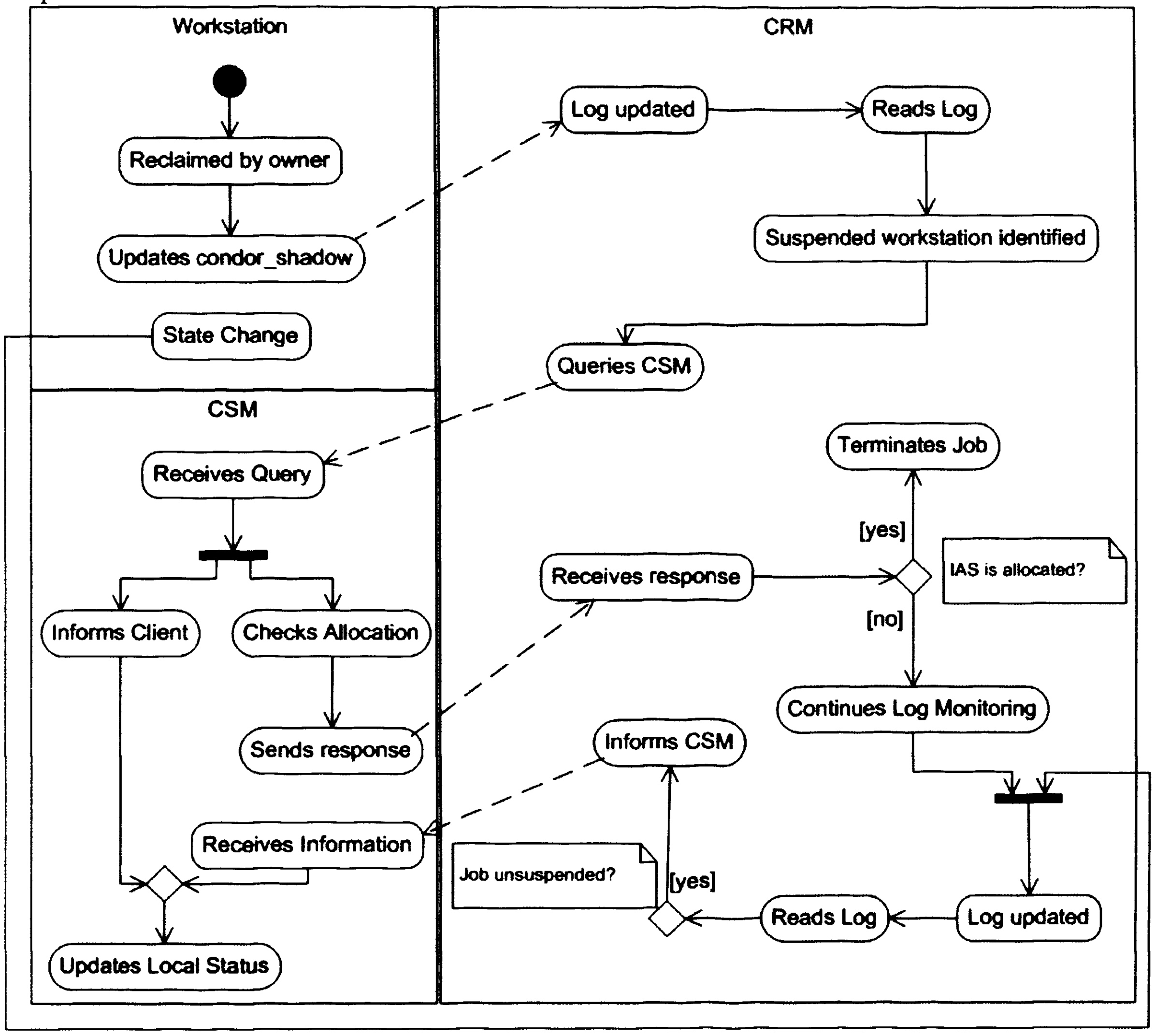
During the design and implementation of this system another eventuality has presented itself, and only occurs under exceptionally specific conditions. The nature of this system is that IAS instances are initialized preemptively and this means that IAS jobs can be idle. If an IAS is in the initialized state, but has no Tasks to perform and is suspended, it can (sometimes) be successfully `unsuspended`. It is believed that this is because the Matlab engine is not reinvoked upon `unsuspension`. During this time the IAS heart beat thread is usually halted, but not always, and it falls out of communication with the CSM. When the IAS is `unsuspended`, its heart beat restarts and it can rejoin the cluster. No additional functionality was built into the IAS to facilitate this, it happened as a matter of chance.

The Job Model enables the CRM to identify when a job enters the suspended state. When this occurs the CRM questions the CSM in regard to whether the IAS is in use. If it is, the CRM will terminate it because it is assumed that the IAS is active and executing a Task. Therefore, upon `unsuspension` it will fail anyway and to remove the chance that a failed `unsuspension` has a knock on effect to the workstation owner or Condor, the CRM requests that the job is terminated. If the IAS is not allocated (it is assumed to also be idle) the CRM will monitor the job until it is either evicted or been `unsuspended` and the CSM will not allocate the IAS to a user. Figure 5.9 shows this process. This is also the reason for the two logical ORs in Figure 5.6, as it is conceivable that

[23]However, remember that in Section 2.3 a time of 16 days was reached.

the CRM may not be able to determine if the job has been suspended before Condor evicts the IAS job.

Figure 5.9: The interactions between the CRM and CSM to identify the course of action for a suspended IAS



The adaptive nature of the performance model stems from the monitoring data, which is reflected in the termination heuristic (Figure 5.8) for workstations which experience difficulty in guiding an IAS to the initialization state. Similarly, jobs are suspended in an ad hoc manner and the way in which this is addressed is dependent on the current context. By employing these dynamic and context-based identification measures, it is possible to guarantee that by the time an IAS becomes available it will function correctly. In Chapter 2 an overview of the issues surrounding the management of compiled Matlab jobs on the Condor Pool was given. In response to these challenges the Job, Resource and Performance models (Sections 4.3 and 4.4.1) were defined and the Condor Resource Manager (CRM) was developed to enact these models.

5.3.4 Acquisition Model

The Acquisition Model (Section 4.4 and illustrated in Figure 4.6) takes input from three sources: **(1)** real time Campus-Grid monitoring and the Resource Model, **(2)** the Performance Model, and **(3)** the CSM's Global Requirements Model (Section 5.4.3) in order to determine whether action should be taken. The first two inputs define the potential to gain resources (physical availability), and the latter the current demand.

The actual implementation of the Acquisition Model is quite simple, as most of the foundations for decision making are provided by the three input sources. The acquisition model has only to answer the questions: **(1)** are more resources needed (initialized IAS count is less than demand) and if so: **(2)** are workstations available which are either likely to produce an initialized IAS, i.e. of those available which are not guaranteed to fail[24] or currently have an unknown state (for example a previously unknown workstation)? Then if necessary and possible, the CRM will inject new IAS jobs using its automated job submission module (Section 5.3.5).

The main consideration that the CRM must take when decided whether to inject new IAS jobs or not, is in regard to the number of IAS jobs running but not yet initialized. The CRM could predict the outcome of a running job, but because of the myriad of known scenarios and unpredictable events which govern job success, this is a very difficult process. Currently, the CRM considers only jobs that have started recently (for example those running for less then 2 standard deviations of the average initialization time) as those likely to succeed and builds its projected resource count around this assumption. The CRM must also consider the current number of IAS jobs residing in its job queue, as presented in Chapter 2, because as the number of jobs increased beyond a specific point undesirable behaviour was observed. This is also the case for the CRM, however, the threshold is lower. Once the number of concurrently running IAS jobs approaches 80 issues with socket buffer space arise and communication between the CRM and CSM can become unreliable. This issue arises due to Condor opening multiple sockets for each job in the execution state, and consequently consuming a large portion of the available buffer space for communication.

In future work a more concrete mathematical model could be implemented to improve the accuracy of the CRM's estimation. In real terms a poor estimation does not have serious ramifications upon the system's performance as a whole in the current Cardiff University Campus-Grid context, where the impact of workstation volatility is significant. However, in porting this approach to other contexts and environments this may not be the case and more established methods of estimating the projected resource count should be considered. Some relevant examples of other work within this domain were given in Section 3.2.

[24]This is an important distinction, as the CRM will consider submitting an IAS to a workstation which may in fact fail. The point is that the CRM provides the workstation the chance to demonstrate either its ability to initialize a job if it is not sure of the outcome.

5.3.5 Automated IAS Submission

An IAS is a single executable, which is accompanied by a small number of code archives and does not require any special considerations for submission to a Campus-Grid workstation. Therefore, the normal submission procedures supplied by Condor are more than adequate for the submission of an IAS. It is the limitations of Condor's management processes that are the unequivocal causes of the complexities of job management. When submitting IAS jobs the CRM automatically generates a Condor submission script, which contains all of the necessary definitions for an IAS to run on a Campus-Grid workstation[25] and the log file for Condor, which is used to report on progress and events. The most important advantage of an automated submission facility is that this process enables the CRM to define which Campus-Grid workstations should not be considered as a candidate workstation in the matchmaking process. This decision is based upon the juxtaposition of the CRM's job and performance models (Section 5.3.1 and 5.3.3 respectively) and available in real time. The generated submit script is then executed using the supplied Condor command line tools.

A consequence of the volatility of resources is that large numbers of jobs are launched by the CRM. This means that over time the job log becomes too large and difficult to manage. Therefore, the CRM limits the scope of a log to one day. The purpose of this approach is to reduce the size of individual logs, but also to enable multi-threaded approaches for the log parsing required by the CRM monitoring processes.

5.4 Central Service Manager – CSM

The Central Service Manager (CSM) is the enactment of the Global Requirements and Allocation Models (Sections 4.5 and 4.6). Put simply, it is the component with which IAS instances (Section 5.2) register their availability and where the CI (Section 5.5) places the user's demand for resources. The role of the CSM is to aggregate the requirements of one or more users, and along with an additional buffer for spare capacity, to inform the CRM of the current demand for resources. This is presented in Section 5.4.3. Once IAS instances have registered with the CSM they are allocated to a user for Task execution. The mechanisms that the CRM adopts for this process are presented in: Section 5.4.2. The CSM also performs some IAS monitoring, which is presented in Section 5.4.1.

However, first a discussion with regard to the motivation for another component, as all functionality of the CSM could conceivable be carried out by the CRM. The distinction is in fact based upon a set of simple premises. (1) There is no need for the Client Interface (Section 5.5) to be aware of the CRM. If a GIS was integrated into a future version of this implementation it would be a new component in the CI, which directly interacts with the CRM database, but not the CRM itself. (2) There may in fact be more than one CRM (one for each access point to or submission node for the Campus-Grid), which would increase the maximum number of IAS instances the system

[25]For example: the IAS components, which are stored in a configuration file, the number of jobs to submit, and the workstation requirements for an IAS, i.e. MCR availability

could run concurrently, and prevent the CRM becoming a single point of failure. **(3)** The conceptual spaces and responsibilities of the CRM and CSM are logically distinct. This also means that the CSM can run as a Campus-Grid job and therefore be managed by the CRM. **(4)** The CSM can fail and be restarted with negligible detriment to the performance of the system as a whole, whereas the CRM cannot.

5.4.1 Resource Monitoring

During the life time of an IAS it is monitored by the CRM's job model (Section 5.3.1) to identify state transitions, but the CSM also performs some additional heart beat monitoring. The reasons for this distinction were given in Section 4.8, but this process could also be carried out by the CRM as a part of the Campus-Grid monitoring process (Section 5.3.2). The reason why this is not the case, is due to the awareness of the CRM. Only the CSM has knowledge of both users and IAS instances, and is therefore best placed to inform a user's CI (Section 5.5) when an IAS failure is detected. In addition, if a user leaves the system, the CSM can inform an IAS to empty its queue of that user's Tasks.

To perform heart beat monitoring is trivial, however, when an IAS is performing Tasks which place the workstation under heavy load, its heart beat can sometimes fail. The difficulty here, is in identifying whether a stopped heartbeat is in fact an indicator of failure or if the IAS is just experiencing a high load. The CSM deals with a failed heartbeat by immediately messaging the IAS front end to inform it that its heartbeat has ceased. If the communication succeeds, the IAS can reinitialize its heartbeat thread, and the CSM knows that an intermittent error occurred, but that the IAS is in fact still alive. If the communication fails, the CSM can expect that the IAS has in fact terminated. It can then question the CRM, which can in turn check its job model (Section 5.3.1) to provide a response.

It is possible that the IAS has in fact failed but that the CRM is unaware that the termination has occurred, as the log is not always updated in real time. Such a scenario opens a debate concerning what action should be taken. Ultimately, the CSM strikes the IAS from it allocatable resources and informs the user of a failure. However, if the workstation has for example lost network temporarily, the IAS job will sit idle until Condor terminates the job as part of a job's natural life cycle. During the time that an IAS is perceived to have failed, it may still be executing user's Tasks. Therefore, if the CRM were to terminate the IAS in response to the CSM's query, useful work may be lost. Note that an IAS does not need the CSM or CRM to communicate with a user, once it has been allocated. Consequently, if an IAS falls out of communication with the CSM, it does not lose the connection with its user. For this reason the CRM will not terminate the job. If the job has failed Condor is likely to be in the process of handling the event. On the other hand, if the IAS is still alive and returns a result to its user, the CI (Section 5.5) will inform the IAS that the CSM believes it to have died and the IAS can reregister with the CSM.

5.4.2 Allocation Model

IAS allocation is based entirely upon the data that the user's requirements model produces, and the number of IAS instances currently available to the CSM. The user's requirements model is enacted by the scheduling interface (Section 5.5.3), and the level of detail provided is dependent upon the implementation of the scheduler. The bare minimum that the scheduler should supply is some notion of the maximum number of IAS instances it can feasibly use. If no minimum is supplied the CSM will assume that a minimum of one IAS is required. Additional information is only required when the CSM must balance the allocation of IAS instances between two or more users.

The allocation model has three possible states: (1) No or too few IAS instances are available to meet the requirements of the users connected to the system. (2) Neither too few nor too many IAS instances are available for allocation. (3) The CSM has spare IAS instances available, which can be used if demand changes. Naturally, the first scenario is not a desirable one, and is most difficult for the CSM and scheduler to resolve. There are essentially two options: (1) do nothing and wait for more IAS instances to become available, or (2) use the IAS resources that are available on the assumption that further IAS instances will become available. Neither option is guaranteed to resolve the problem, which is due to an inherent attribute of opportunistic computing in general: there can be no guarantees with respect to resource availability. Therefore, it is ultimately the scheduler that must decide which action should be taken. If IAS instances are available, the scheduler can reduce its minimum requirement to enable the CSM provide some computational resources. On the other hand if no IAS instances are available or the scheduler cannot reduce its minimum requirements, there is little that the scheduler can do other than wait, or terminate and try again later. Some solace is provided due to the observation that for the CRM to be unable to initialize a single IAS is quite uncommon. The CSM could also query the CRM to provide the scheduler with some indication of the state of the Campus-Grid, but currently this facility is not available to the CSM. In addition, what combination of information the CRM can use to build a model that can answer this query is a research question in its own right, and beyond the current scope of this research, as it would require the adoption of predictive and statistical models which are already present (to some extent) in the literature, some examples were given in Section 3.2.3.

There are of course some simple indicators available to the CRM that can be used to identify when it is unlikely for resources to be available in the near future: (1) when the number of idle jobs waiting for resources is several orders of magnitude greater than the number of available workstations. (2) When no workstations are currently available which meet the requirements to run IAS or have a history of success. (3) When the average time for a job to experience the state transition from idle to running is high. However, each of these scenarios could only be indicative of a negative outcome, where the user may wish to run their application at a later date. Identifying when it is a "*good*" idea for a scheduler to wait requires a much deeper appreciation of the current state of the Campus-Grid. For example learning or pattern recognition frameworks have been used in the literature to identify common traits in order to model volunteer computing environments and Campus-Grids.

The second allocation scenario is the most common for the allocation model and an acceptable position to be in. However, any changes in IAS availability could lead to an insufficient number of IAS instances being available, and therefore result in the issues discussed above.

When IAS instances are available, the CSM will allocate either as many as are available or the maximum number requested for by the scheduler. When an IAS is allocated, the CI (Section 5.5) receives the endpoint of the IAS's front end and the IAS is informed of the user that it has been allocated to. The CSM also logs the allocation so that it can update either party if one falls out of communication with the CSM. All communication between the two parties (CI and IAS) is then direct point to point communication in which the CSM plays no part. In this respect the CSM is only the means for the CI to discover resources, but has no control over what an IAS does. For this reason if the CSM fails users can still use their allocated resources, but not discover (be allocated) new resources until a new instance of the CSM is spawned.

Currently, the CSM can only reliably handle one active user at any given time. However, the allocation model was conceived to cater for multiple active users and the foundations for such an approach are in place. The approach for IAS allocation within a multiple user context is associative averaging, where the need of each user is compared relative to all others based upon the data gathered by the scheduler to determine the allocation given to a user. Here, the sets of attributes can be used to iteratively quantify the relative need for all connected users for the duration of their runtime. The aim of quantifying need relative to other users is to perform reactive fair share allocation, based upon the observations of the scheduler and Task model. However, as many aspects of the CI are fully customizable the question of trust is soon raised: Can the CSM trust the values supplied to it by a user? Ultimately such an issue is beyond the scope of this research, but it nevertheless is an important issue. Similarly, a scheduler can overwrite the monitoring framework of the CI (Section 5.5.2) and consequently, may no longer produce accurate data. Both of which are concerns that any future work on this particular feature of the CSM should consider. This feature is currently not actively supported in the CSM prototype. However, some initial work was presented in [33].

5.4.3 Global Requirements Model

When the CSM initializes, one of its first actions is to contact the CRM and request a predefined number of resources, which denote the CSM's initial spare capacity. These resources will be allocated once a user materializes, based upon the user's demand and will start the cycle of the CSM and CRM communication for setting a dynamic target horizon for the CRM to aim for.

When a user does materialize, the CSM's implementation of the Global Requirements Model (Section 4.5) analyzes the estimation for the ability to consume resources from the CI (Section 5.5) and compares it to the current number of resources provided by the CRM. The CSM will initially allocate what resources it has increasing its request to the CRM to gain spare capacity. However, should the estimate from the user be higher than the CSM's current request, the CSM will incrementally increase its request to the CRM, until the two begin to converge. Note that full convergence will not occur due to the CSM's ambition to acquire spare capacity. The reason for an incremental approach is that the CSM initially assumes that the user's estimation is inaccurate, but improves over time. Improvements will stem from the generated performance data of Tasks as they begin to complete giving the CI's implementation of the user's requirements model (Section 5.5.3) real data upon which it can build its estimate.

5.5 The Client Interface

The Client Interface (CI) is enactment of the Task, (User's) Requirements, Scheduling and User's Models (Sections 4.3.2, 4.5, 4.7 and 4.9). It provides a generic toolkit for interacting with two of the other three core system components, namely the CSM (Section 5.4) and IAS instances (Section 5.2). In addition, it also provides the necessary functionality for the interaction with an instance of the LS (Section 5.6). Using the CI users can build autonomous applications, simply describe a Task space or farm out Tasks to one or more IAS instances from some application environment in a transparent and interactive manner. For most users the CI will simply be a transparent platform upon which to build their applications, however, it is fully customizable and can cater for many disparate application scenarios.

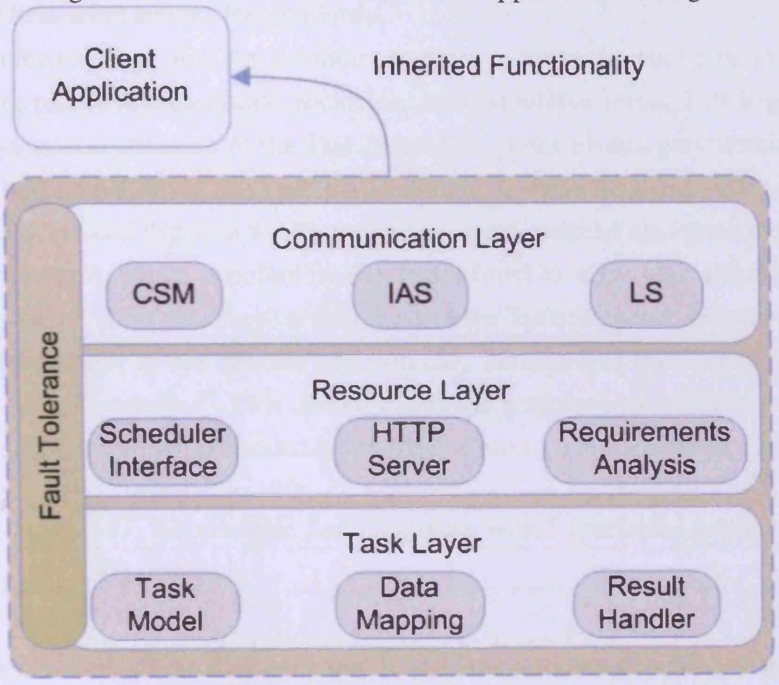
The CI has three key aims: **(1)** To isolate the administrative procedures involved in distributed image processing from the developer. Thus, providing a pragmatic approach and allowing the user to stay within their application domain. **(2)** To enable the developer to specify no more than what they want to do, on what data it should be performed and some way of identifying the result(s). **(3)** To support applications for which batch processing is not an option.

The CI is a multi-layered component and consists of three layers: **(1)** communications, **(2)** resources, and **(3)** Tasks. Figure 5.10 shows each of these layers, their basic modules and what functionality a user's application inherits from the CI API. Each layer is incorporated into an Application Manager, which manages the execution of a Task space on behalf of the user (Section 5.5.1). After the outline of the Application Manager, the layers depicted in Figure 5.10 will be presented bottom up. In Section 5.5.2 the Task Layer will be presented, this will by followed in Section 5.5.3 by the Resource Layer and in Section 5.5.4 by the Communication Layer. Note that aspects of fault tolerance are integrated into all layers.

5.5.1 The Application Manager

An Application Manager is written in Java and by default inherits the necessary functionality for each layer transparently, as illustrated in Figure 5.10. Should a developer wish to alter the way in which a layer is implemented they can simply overload the default implementation either through polymorphism or by using the provided API to define new modules. This enables them to include new behaviours specific to their application or to create other customized front ends to the system as a whole. Throughout this work several different front ends have been created and are reported in the literature. A web-based front end was presented in [150], which was in fact the first version of the Application Manager and a much simpler version of the system that is presented in this Chapter.

Figure 5.10: General structure of an Application Manager



During the conference it was also presented as a live demonstration, where the speaker in Boston interacted with workstations on the Cardiff University Campus-Grid. In parallel a Prolog-based front end was also created and is documented in [151]. In Chapter 6, a Matlab-based front end is presented for parallel image processing from within a Matlab session.

An Application Manager is self regulating and configuring and therefore capable of autonomic behaviour, which stems from the three layers of the CI. There are many areas where the Application Manager can be extended either to improve the basic functionality and robustness of the CI or to consider new research questions (see: Section 8.3).

5.5.2 The Task Layer

The Task Layer is responsible for the client side execution cycle of a user's Tasks, which includes definition (the Task Model), data management and the receipt of results.

The Task Model and Mapping of Data

In accordance with the Task Model defined in Section 4.3.2, the user supplies only the following information: **(1)** A Matlab Task Sequence, which describes what is to be done. **(2)** Zero or more data sets, i.e. what to act upon **(3)** An ID Tag, which is a method to identify the result(s) or an error, should one occur. It is not always necessary for a user to supply an ID Tag, as an Application Manager can also automatically generate one. However, if a user uses a general Application Manager, rather than the specific Application Managers mentioned elsewhere in this thesis, then they are often required to supply an ID Tag. A Task is encapsulated by a Java Object, which handles many of the tedious Task administrative actions transparently.

The first administrative action that is undertaken stems from the background procedures that transform the three pieces of information above into an schedulable entity. This begins with the verification of the syntactical structure of the Task Sequence, to ensure that parenthesizes are balanced and that all functions constituting the Task Sequence are actually included in the IAS. Remember that when the IAS is created (Section 5.2.1), the Autonomic Compiler also generates a complete list of the included functions, which is polled by the Task Model to verify that a function is in fact included within the IAS.[26] This process also decomposes the Task Sequence into functional particles, where each particle relates to one Matlab function call, for example the Task Sequence shown in Listing 5.11 contains 5 particles.[27] This decomposition is performed to identify the last functional entity of a Task Sequence which is needed by an MI (Section 5.1.2) to correctly generate a result.

Listing 5.11: An example Task Sequence with 5 functional particles

```
rni(3).a = avg.thr(a).big.cwp
```

The decomposition of a Task Sequence into functional particles also enables the Task Model to identify data dependencies. There are two ways in which to identify a data dependency: **(1)** The particle is a data load function e.g. `load`, `gwi`[28] and `rni`,[29] or **(2)** A particle's argument is a subordinate data load e.g. `myFunction(load('c:\myData.mat'))`. The identification of data dependencies in this manner means that the user can specify their code without significant modification.[30] In both of the presented cases some data needs to be transferred to the remote node. To accommodate this need the Task Model will signal to the CI that a HTTP Server (Section 5.5.3) is required for data hosting. The reason to transfer the data in this way is because it allows an IAS to decide when and if it requires the data, and therefore creates a pull rather than a push model.

All data sets that are identified within functional particles are copied into the root directory of the HTTP server.[31] During this process each data set is renamed to ensure that two data sets with the same name do not overwrite each other. The name is also globally unique; a one way hash of the absolute file name and canonical host name of the HTTP server's endpoint. This ensures that an IAS does not download the same data set twice. The particle is then edited to reflect this by converting the local data load into the remote equivalent. The Task is then flagged with a data dependency to enable the IAS's Task Handler (Section 5.2.2) to identify data dependencies and whether it needs to perform a transfer. The Task Handler will always download data to the same

[26]There are many other checks which could be performed at this point. For example: **(1)** the identification of Matlab functions that could be used maliciously e.g. `delete`, `cd`, `quit`, which would provide some basic levels of security, and **(2)** further syntactical analysis to ensure that the correct number of input parameters are used.

[27]The five particles are: **(1)** `rni(3)`, **(2)** `a = avg`, **(3)** `thr(a)`, **(4)** `big`, and **(5)** `cwp`.

[28]QT Command Get Web Image

[29]QT Command Read Numbered Image

[30]The main modification is the need for absolute path names rather than the use of relative path names typically used in Matlab. However, some Matlab functions exist which can perform this task for the user.

[31]An alternative would be to generate a map within the HTTP server to identify the local copy.

relative location on the Campus-Grid workstation. Therefore, the Task Model can edit the data load to reflect where the data will be on the Campus-Grid workstation. Similarly, because the name of the data set is determined by the CI, a complete particle transformation is possible. For example `load('c:\a_folder\a_sub_folder\data\image.png')` would be changed to `load ('Downloaded_Files/some_hash.png')`.

By default an IAS will not transfer the same file twice. Therefore, if live data is in use (for example a webcam feed), the Task Model provides a flag to indicate that this data should always overwrite any copy that an IAS has transferred in the past. Currently, there is no mechanism for issuing a time to live for a Task or for deciding when data should ideally be downloaded. However, the output from a live fed can be captured by the Task Model for later processing, whenever this is necessary.

Receipt of Results

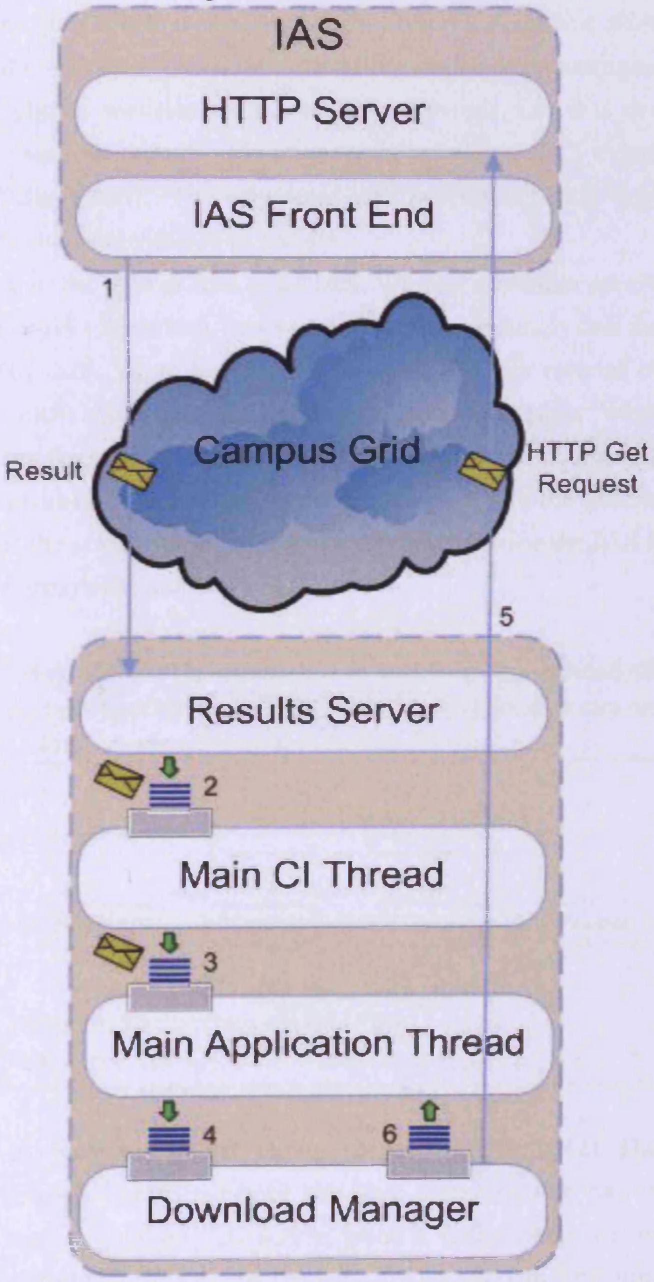
The Result Handler is the solution to the potential problem of the client-server model when multiple IAS instances are "*owned*" by an application, which was outlined in Section 5.2.2. Essentially, it allows the IAS to close a connection to the CI once a Task has been received, as the implementation of the Task model adds an endpoint of a results server to each Task. Typically, this is running on the same physical machine as the Application Manager, and is therefore initialized transparently by the CI. However, it could also be a different physical machine, for instance a database front end or result consumer within a workflow management system.[32] The process which then occurs is illustrated in Figure 5.11 and is as follows: **(1)** the IAS produces a result, extracts the endpoint from the Task, reestablishes connection with the Application Manager and sends it to the results server, **(2)** the results server forwards the result to the main CI thread using an event queue. Here performance data is extracted by the scheduling interface (see Section 5.5.3). **(3)** The raw result is then passed on to the user's application through an event queue. **(4)** Depending on the implementation of the application and a user's preferences the download manager can be invoked to download any data still in the possession of the IAS, for example graphical output or `.mat` files. **(5)** The download manager contacts the IAS's HTTP server and downloads the results using the ID Tag as an prefix for the file names. **(6)** The download manager informs the user's application that the download is complete and of the file names attributed to the downloaded result files, using a callback event queue. The download manager can also be replaced by a customized user module to perform similar actions.

5.5.3 The Resource Layer

The resource layer is responsible for all interactions between a user and the IAS instances which have been allocated to a user, which includes: scheduling, requirements modelling and the hosting of Task data for the transfer to IAS instances.

[32]Note that in these cases the Application Manager would also have to be informed so that it can invoke the scheduler.

Figure 5.11: Result Stack

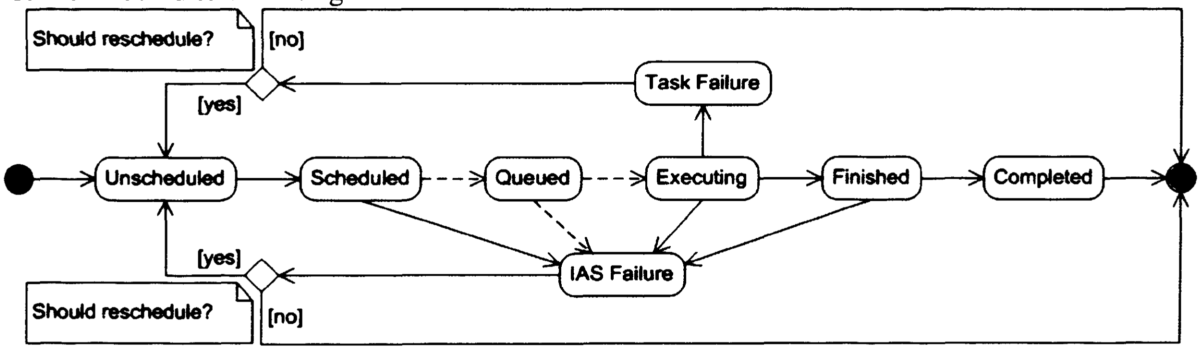


Scheduling Interface

The scheduling considerations outlined in Section 4.7 provide the template for all schedulers to adhere to. In order to turn these considerations into a practical template for programmers, the scheduling interface provides a superclass template for all scheduling algorithms to follow. Whether an algorithm is taken from the literature or written specifically for an application, it must still follow the conceptual template prescribed by the scheduling interface. In line with other scheduling approaches in the literature (Section 3.2.4), the scheduling interface encourages an iterative algorithm where the scheduling plan is reevaluated in response to events, i.e. it is event-driven. All events come from one of two sources namely: **(1)** changes in resources (IAS events), and **(2)** changes to the state of a Task (Task events). The scheduler API propagates each event through its abstract skeleton for a scheduler implementation to handle.

An IAS event is either the gain or loss of an IAS. When a scheduler receives a new IAS instance to interact with, it can begin scheduling Tasks to that IAS immediately and therefore must readdress its scheduling plan. Similarly, when a scheduler loses an IAS, for reasons other than IAS idleness (see: Section 5.2.2) it must also readdress its current scheduling plan. When a scheduler loses an IAS, it is told the reason for the loss, to enable it to determine the action required. For example, if an IAS has failed the Tasks in its queue need to be placed back into the unscheduled state. However, this does not mean that the scheduler should not immediately strike the IAS from its registry, as the failure may only be intermittent (see: Section 5.4.1).

Figure 5.12: Possible Task States. The transition to and from the queued state are not visible to a scheduler, as this is a process specific to an IAS instance. A scheduler can only observe a transition from scheduled to executing.



Task events relate to changes in a Task's state (shown in Figure 5.12). The action that the scheduler needs to take is strongly dependent upon the state transition that has occurred. For example, a scheduler does not need to perform any action when a Task transfers from the scheduled to the queued state, and unless the IAS inform mechanisms are updated, this information will not be available to the scheduler. However, it is useful for a scheduler to know when a Task has moved to the executing state. Naturally, the most useful updates stem from the *completed* and two *failure* states. The distinction between the states *finished* and *completed* is that in the *finished* state results may still need to be collected. In the *completed* state, however, any results that need to have been explicitly downloaded (as described in Section 5.5.2 and Figure 5.11) and are fully available to the user. This distinction is necessary because an IAS can be lost at any time in the life cycle of a Task, and therefore a scheduler may even need to reschedule a Task after its execution has finished successfully, if the results have not yet been accessed.

The areas of core functionality that the scheduling interface provides are: **(1)** event listening and delivery to the abstract methods shown in Listings 5.12 – 5.14, **(2)** Task dispatch using the Communication Layer (Section 5.5.4), **(3)** basic Task performance monitoring for example: average queuing and executing time, **(4)** basic IAS monitoring for example: average and maximum count, **(5)** basic requirements modelling for example: determining minimum and maximum IAS allocation, based upon current state of all Tasks, **(6)** basic prediction for example: estimated completion time with the current allocation, average Task execution and queue times, **(7)** data collation for the requirements model, **(8)** autonomous handling of IAS queue sizes, and **(9)** user preference elicitation with respect to runtime attributes such as: the minimum and maximum queue size or number of IAS instances. Note that subclasses of the scheduling interface are free to overload any of these areas of functionality.

These areas of functionality are catered for through the definition of an abstract scheduler superclass, which all other schedulers must extend. Here, the method signatures required are defined, and the necessary event queues and message handlers established. Listings 5.12 – 5.14 shows a breif outline of the key methods.

Listing 5.12: Task Methods for a scheduler

```
//Add new Task (encapsulated by the Command Object) to group of unscheduled Tasks
protected abstract void _add(Command c);

//Changes in command state
protected abstract void failed(Command c, ErrorResult r);
protected abstract void finished(Command c, Date d, IAS ias);
protected abstract void resubmit(Object tag);
protected abstract void resultDownloadFailed(Object Tag);
protected abstract void started(Command c, Date d, IAS ias);

//Access all Tasks in each state, here buffered is queue at an IAS
//queued is a Task which has not been scheduled
protected abstract CommandCollection getBuffered();
protected abstract CommandCollection getComplete();
protected abstract CommandCollection getError();
protected abstract CommandCollection getExecuting();
protected abstract CommandCollection getQueued();
//Get a Task regardless of state
protected abstract Command getCommand(Result r);

//Determine is a Task is in a specific state based upon its ID Tag
protected abstract boolean isBuffered(Object tag);
protected abstract boolean isComplete(Object tag);
protected abstract boolean isError(Object tag);
protected abstract boolean isExecuting(Object tag);
protected abstract boolean isQueued(Object tag);
```

The scheduler superclass does not provide any support for load balancing as this is a scheduler dependent issue.

Listing 5.13: IAS Methods to be defined a scheduler implementation

```
//Modify plan due to loss
protected abstract void lostIAS(IAS ias);
//Schedule a Task to this IAS
protected abstract void schedule(IAS ias);
//Create initial scheduling plan
protected abstract void schedule();
```

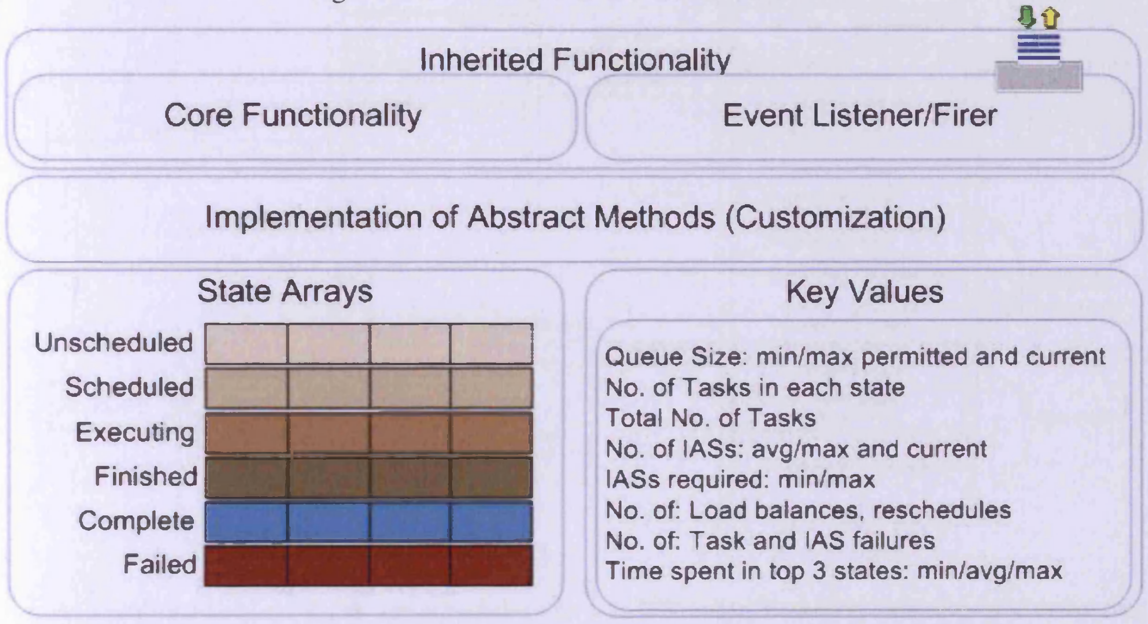
Listing 5.14: Administration methods for the all schedulers provided by the abstract scheduler superclass

```
//Add new Task(s) to scheduling plan, also checks here for duplicity in ID Tags
public void add(Command c);
public void add(CommandCollection commands);

//Get key values either for requirements model of post execution analysis
public Value getValue(int type);
//Set key values such as maximum queue size, number of IAS instances etc.
public void setValue(Value v);

//Initial scheduler plan is ready
public boolean isReady();
//Whether the scheduler is currently active
public boolean isRunning();

//Set IAS collection object
public void setIASPool(ClientIASPool IASs);
//Set the listener of the user's Application Manager
public void setResultListener(ResultListener rListener);
//Receive an event which may affect the scheduling plan
public void receiveInformation(InformEvent event);

//Send a Task to the provided IAS
protected void send(Command command, IAS ias);
```

The Default Scheduler

The default implementation of a scheduler is a simple FCFS[33] opportunistic scheduler and an overview of its structure is shown in Figure 5.13. Its opportunistic status (a consideration required by the scheduling model) stems from its ability to consume new IAS instances in an ad hoc manner provided that Tasks remain that are neither in the *complete* nor *failed* states.

Figure 5.13: Structure of the default scheduler

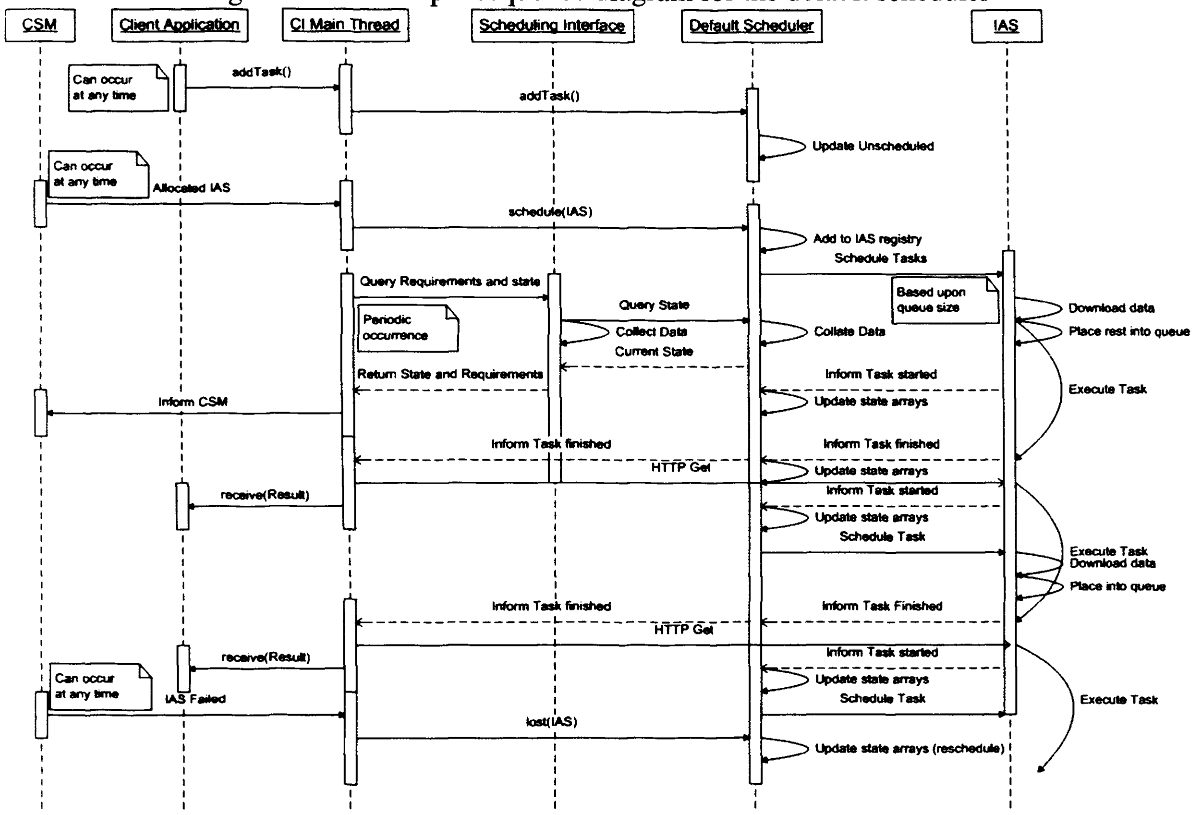

Inherited Functionality
Core Functionality
Event Listener/Firer
Implementation of Abstract Methods (Customization)
State Arrays
Unscheduled
Scheduled
Executing
Finished
Complete
Failed
Key Values
Queue Size: min/max permitted and current
No. of Tasks in each state
Total No. of Tasks
No. of IASs: avg/max and current
IASs required: min/max
No. of: Load balances, reschedules
No. of: Task and IAS failures
Time spent in top 3 states: min/avg/max

Every Task event and therefore state transition is captured by a migration of a Task from one state array to another in accordance with the possible state transitions illustrated in Figure 5.12. Here, the *failed* state array contains Tasks which will not be rescheduled, the decision making processes which determine whether a Task will be rescheduled or not are presented in Section 5.5.5. The key values for the requirements analysis are mostly implemented as part of the scheduling superclass, or extracted through abstract method invocations.

Figure 5.14 shows a sequence diagram of an example runtime for the default scheduler. Notice that there are several actions which can occur in an ad hoc manner during runtime, which other scheduler implementations need to consider. The scheduler is essentially an asynchronous and event driven multi-threaded scheduling module. This means that it is not possible to draw a single sequence diagram for all scenarios, which will be both readable and complete. Note also that Figure 5.14 shows only one IAS instance. The focus of this research is not on scheduling and therefore no significant effort has been made within this domain. Therefore, much of this approach's runtime potential is yet to be achieved. In Chapters 6 and 7 this approach is evaluated using both real world and synthetic applications, the performance of which can be improved through the use of better scheduling algorithms to the default scheduler presented here.

[33]First Come First Served

Figure 5.14: Example sequence diagram for the default scheduler



As mentioned above Tasks are scheduling in a FCFS order, however, this does not mean that they are scheduled in the order in which they are defined. Instead, they are scheduled in the order in which they became a fully schedulable entity. Here, the requirements are that any data has been placed into the web root of the HTTP module and that the syntactic checks of the Task Sequence (Section 5.5.2) have been completed. When a Task is rescheduled for any reason, it is placed at the end of the queue. The scheduler also provides some simple load balancing and redistribution. Here, a redistribution of the IAS queues will occur when no Tasks remain in the unscheduled state. In order to determine which Task should be taken for redistibution the scheduler uses the heuristic presented in Section 4.7:

$$\begin{gathered} IAS[]iass = Max(QueueSize_{IAS}), \qquad (5.1) \\ forall(iass) \\ Task_t = lastSubmitted(Min(AvgExecutionTime - iass_i(R_t))) \end{gathered}$$

In other words, it firstly identifies the IAS instances with the largest queue sizes to create a list of potential candidates. Secondly, it identifies which IAS has the most work remaining on its current Task, and takes the last Task submitted to that IAS for redistribution. This Task is then flagged as one which has been redistributed to prevent it from being redistributed again for the k times the average Task execution time, where k is the current queue size for all IAS instances. Load balancing is performed when a call to the `void schedule(IAS ias)` method cannot identify a Task to schedule due to the depletion of Tasks in the unscheduled state.

To determine the current queue size the following heuristic is used:

$$Min(Q_{max}, Max(Q_{min}, \lceil NoTasksUnscheduled + NoTasksInIASQueues \times 0.05\rceil)) \quad (5.2)$$

In other words, 5% of the total number of Tasks which have not yet been scheduled, provided that this value falls between the bounds of the predefined minimum and maximum values for queue size. Where, the default values are 2 and 10 respectively. These values can also be modified by the user if they so wish. 5% is used, as in preliminary tests it was most likely to result in the desired behaviour; namely the reaction of the queue size to the number of unscheduled Tasks. This heuristic is employed after every scheduling event, and therefore changes dynamically over time or in response to new Tasks being given to the scheduler for execution. In addition, it also decreases toward the minimum value over time. In this way it reduces any negative impact from the scheduler receiving new IAS instances near the end of the execution of the Task space.

The implementation does not perform any form of redundency scheduling, where toward the end of the Task space's execution Tasks are replicated across IAS instances. This is an area which could improve the performance of the system as a whole. The reason for not adding this feature is to prevent the scheduling algorithm from getting overly complicated.

The Requirements Model

The Requirements Model (Section 4.5) is implemented as a collection of monitored metrics and captures three different categories for the CSM to consider: **(1)** Task-orientated metrics, which describe the Task space of the application. Examples include: average execution time, the average time spent in an IAS' queue, the number of Tasks in each possible state, i.e. unscheduled, executing, completed and failed, and an estimated completion time based upon the current allocation and the previously mentioned Task metrics. **(2)** Resource-orientated metrics, which describe the demand for resources in relation to the current Task space. Examples include: the number of IAS instances available to the scheduler, the average number "owned", excluding any time when no IAS instances were available, the current queue size used by the scheduler, and the minimum and maximum number of IAS instances requested and allocated. **(3)** Usage-orientated metrics, which describe how a scheduler is using the IAS instances allocated to it. They can also be used to report back to the user to illustrate how the system performed when executing their application. Examples include: the number of load balances due to new IAS instances becoming available, the number of resubmissions due to IAS and Task errors and the number of times that an IAS was left idle. In a multi-user context (see Section 5.4.2) this category could be used for tie breaking, when determining IAS allocation.

To capture the Task-orientated metrics, the scheduling interface provides some basic functionality and a mandatory skeleton, which all schedulers must implement. This skeleton enables the scheduling interface to capture the number of Tasks presently in each of the Task states (Figure 5.12) without any knowledge of a scheduler's implementation. For other metrics such as the average time spent in the *scheduled* and *executing* states, the scheduling interface flags when a Task is sent to an IAS. Then when an IAS begins Task execution, it informs the Results Server identified in the Task as the endpoint for result consumption. A result server will propagate this information as a Task event to the scheduling interface, where the total time spent in queue is recorded. Naturally, this is subject to network latency, but in the grand scheme of things this is not necessarily a negative attribute. The time spent in an IAS queue is part of the time a Task is in a pending state. Therefore, the time required to transfer a Task object is not that semantically different, as the Task's state remains the same during this time. To calculate the estimated remaining time (as described in Figure 5.15) is useful for the CSM, because it identifies whether there is any purpose in requesting further resources from the CRM. This estimate does not consider Tasks that are currently running for the simple reason that this level of accuracy in the estimate is not needed. In addition, the host workstation that the Task is running may be revoked during its execution.

Figure 5.15: Heuristic for estimating the remaining runtime of an application, where N is the current number of workstations, T_{exe} is Task execution time, S_{all} is Tasks in all states and S_{comp} and S_{fail} are all Tasks in the *completed* and *failed* states respectively.

$$\frac{average(T_{exe}) * (count(S_{all}) - count(S_{comp}) - count(S_{fail}))}{N}$$

To capture resource- and usage-orientated metrics some internal *house keeping* is performed, as many are simple observations. For example the current queue size and maximum number of IAS instances "owned" and the number of failures or load balancing occurrences. However to calculate the average number of IAS instances, the total number available is sampled once per second to generate an average. The minimum number that the scheduler requests is always set to one, but scheduler implementations are given the scope to determine this value for themselves. On the other hand, the maximum allocation is calculated by considering the number of unscheduled and scheduled Tasks (but not those currently executing) divided by the minimum queue size. Again scheduling implementations are free to modify and build upon this functionality, users can also set a finite limit.

The scheduling interface provides the data collation mechanisms which then transparently provide the necessary information to the CI upon request, as shown in Figure 5.13.

HTTP Server

The HTTP server is a Java implementation of an HTTP server for data distribution using a pull model. Despite its documentation as a part of the CI, all other components have the option of integrating an HTTP server into their core infrastructure. The exception here is the IAS (Section 5.2) for which under the current implementation it is a mandatory prerequisite. In addition, as a fully customized implementation, it can also feature other useful pieces of functionality other than data transfer. For instance it is exceptionally useful for hosting debugging messages and the current state of an component should it run on a Campus-Grid workstation. As, the HTTP server is an implementation specific and customized for this work it has no software dependencies and is very concise: less than 8KB of compiled code.

5.5.4 The Communications Layer

All components have access to this foundational layer of functionality. The adopted approach to enable components to communicate with each other is message-based, where a message is a single serialized Java Object. The reason for this choice is the lightweight nature of the protocol, as even a large message (a Task for example) is typically less than a kilobyte in size. In addition, extending a protocol based upon messages is straightforward.

To receive a message, all components have a server front end, which is discoverable by all other components. However, the location and discovery of a component's endpoint is based upon the components initiating and receiving the communication. The CRM and LS are in static and known locations and therefore, their endpoints are placed into a configuration file. The CSM has no fixed location or port to use for communication, and therefore must register with the LS to be discoverable (see Section 5.6). Similarly, IAS instances are not fixed to specific locations or port numbers. They discover the CSM using the LS, and their CSM registration process divulges their service endpoint. Finally, a user has no locality requirements only that they are able to communicate with other components on the Campus-Grid. They also use the LS to discover the CSM and divulge their messaging server endpoint during their CSM registration. The CI discovers IAS instances as a result of the CSM's allocation of an IAS.

When a message is received, internal logic disseminates the message context and initializes a suitable handler (if necessary) for the required action to produce a response. In most cases the communication model is a standard client-server approach. However, as previously mentioned (see Sections 5.2.2 and 5.5.2) this is not the case for Task execution where the message (a Task) contains an endpoint for a result server (Section 5.5.2).

The natural alternative to a message-based approach would have been the use of Java's Remote Method Invocation (RMI) API. However, this is a developer dependent implementation issue that does not need discussion here.

5.5.5 Fault Tolerance

The CI needs fault tolerance in two key areas in order to retain stability. Namely, in the communication layer, especially in regard to IAS instances, and at the scheduling level in regard to Task errors and IAS failures.

When communication failures occur there is little that the CI can do other than inform another component of the failure. If the CSM has failed, it will inform the LS, as will IAS instances if they detect that the CM is unreachable. An LS can then start a local instance of the CSM or make a request to the CRM to schedule one on the Campus-Grid. This is a simple implementation issue and is therefore not documented in this thesis. If an IAS becomes unreachable, the IAS communication module will fire an IAS event representing the failure to be consumed and dealt with by the scheduling interface. This event will also be propagated to the CSM.

There are two types of Task failure: (1) IAS failure, where an IAS fails, and (2) Matlab failure, where the Task failed during its execution. The decision making process for deciding whether a Task should be rescheduled after an IAS failure is firstly, dependent upon the state in which the Task was at the time of the failure. If the Task was in any state other than *executing* it is safe to assume that the Task was not responsible for the failure and it is placed back into the *unscheduled* state. However, if the Task was *executing* when the failure occurred the scheduling interface will flag the Task as a potential cause of the IAS's failure, and place it back into the *unscheduled* state for observation. If a flagged Task is identified as a possible cause for further IAS failures the scheduler will place the Task into the *failed* state and alert the user. The number of failures which a Task must cause before it is assumed to be a cause of failure is a subjective issue. However, the default scheduler (Section 5.5.3) will place a Task into the *failed* state after it is identified as a potential cause three times. Matlab failures are runtime failures of a Task and are therefore treated differently. Simply, placing the Task into the *failed* state immediately makes sense for most failures of this type. Yet, some failures are not the fault of the Task such as a class loader error. Therefore, the scheduler will prompt the user for a decision concerning what action should be taken for such failures and others with the same Matlab error message. This approach enables a scheduler to build an exclusion list over time to decide when to reschedule specific Matlab failures. Future implementations could also consider providing the user with the ability to modify the Task or create more elaborate rules to define what action should be taken for a given scenario.

5.5.6 Summary

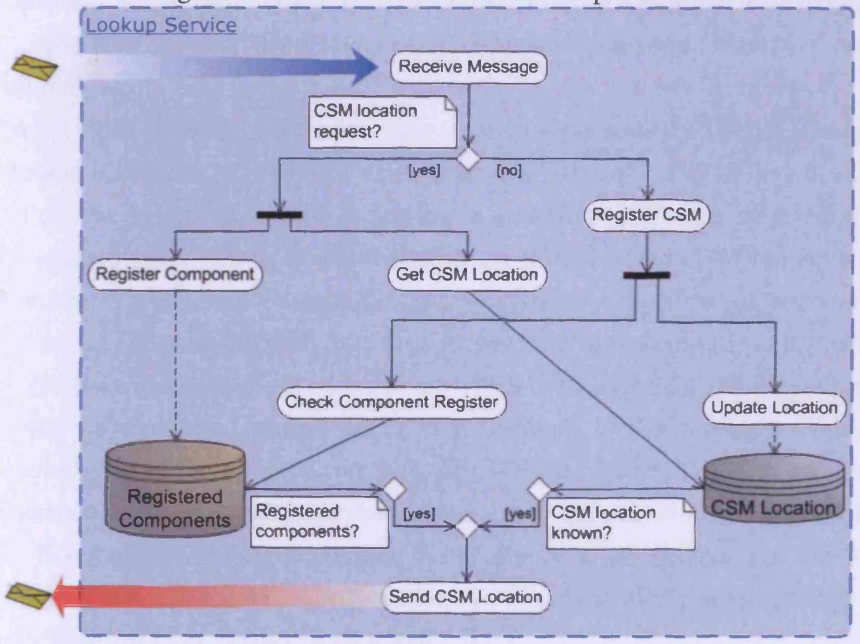
Ultimately the aim of the CI is to remove as many of the administrative issues of Campus-Grid utilization as possible from the user. This is achieved by providing a layered architecture which handles many of these issues transparently. The key features of the CI that enable this are: (1) simple Task specification: a Task Sequence, data and (sometimes) an identifier, (2) automated resource utilization and acquisition through Task remapping, opportunistic dynamic scheduling and autonomous requirements modelling, and (3) intuitive fault handling, which enables the identification of Task-related errors and the autonomic and context-related handling of Task errors; a feat that was previously not possible.

The CI (Section 5.5) has only two requirements for a user: (1) A Java Virtual Machine (JVM) is required to run an Application Manager (Section 5.5.1), and (2) A suitable Internet or Network connection in order to communicate with the LS (Section 5.6), CSM (Section 5.4) and any allocated IAS instances (Section 5.2). This first requirement can also be alleviated through the use of a customized Application Manager such as those presented in [151] (an SWI-Prolog based front end), [150] a web based front end and Chapter 6 a Matlab based front end will be presented.

5.6 The Lookup Service – LS

The nature of the CSM (Section 5.4) means that it can be submitted as a Campus-Grid job, and therefore it has no fixed location. Similarly, it means that it can be evicted and resubmitted elsewhere on the Campus-Grid during the runtime of a component. A consequence of this ability is that when new components join the system they need a point of reference through which they can locate the CSM. In addition, when a new CSM instance joins the system all components need to be informed of the new CSM's location. A Lookup Service (LS) is a static point in the system, whose role is specifically to act as the point through which all components in the system can discover the CSM, and be informed of a new CSM location when a new CSM instance joins the system. This process is illustrated in Figure 5.16. There is no novelty in the use of a lookup service, it is purely a utility to ease the administration of transient and nomadic system entities.

Figure 5.16: The data flow of a Lookup Service.



5.7 Summary

In this Chapter the implementation and enaction of the models introduced in Chapter 4 was presented. This began with how Matlab is controlled (Section 5.1) and fulfills the requirements for the IAS (Image Analysis Service, Section 5.2) daemon back end. Here, Matlab is controlled interactively and in a compiled context, something which is not presented anywhere else in the literature. In addition, the IAS daemon's Matlab functionality can be made fully heterogeneous with respect to its application context. This is achievable through the Autonomic Compiler (Section 5.2.1), which can capture and compile entire toolboxes at the request of the user. It can also tune the compilation and identify multiple errors to enable the user to supervise the compilation and correct the errors that the AC identified. Using this approach even the most complicated toolboxes can be compiled with little user effort. This translates into a compiled Matlab stand-alone application which can serve completely different applications, without recompilation or resubmission to the Campus-Grid.

In Section 5.3 the Condor Resource Manager was presented, which represents the Job, Acquisition, Resource and Performance Models (Section 4.3 – 4.4). It acquires Campus-Grid workstations transparently for the user, and absorbs all of the effort required to manage resources in a volatile and dynamic context. The CRM captures the capabilities of a workstation through the retention of observed performance data with respect to the pilot job aspects of an IAS. Here, the IAS initialization routine tests a workstation before attempting to launch the Matlab Component Runtime (MCR) and join the pool of other IAS daemons which have been initialized in response to a user's request. Should this test pass, but the IAS still not be able to start, the CRM employs a termination heuristic which is orientated around observed performance data. The end result is that users can concentrate on their research and not on the debugging and administration of a complex distributed system. In addition, to a scheduler the initialized IAS instances appear as a single cluster of dedicated, but transient, resources.

In Section 5.4 CSM (Central Service Manager), which represents the Global Requirements and Allocation Models (Chapters 4.5 and 4.6) was presented. The role of the CSM is simple: it is a single point of resource (IAS) discovery. However, as a single point it may be prone to failure and thus cause the entire system to fail. To combat such eventualities, the CSM has been designed so that it could be run as a job on a Campus-Grid workstation and therefore, if necessary, managed by the CRM. The LS (Lookup Service, Section 5.6) was also introduced to act as a static point in the system. It retains a database of all components which have requested the location of the CSM. Then when a new CSM instance informs the LS of its location, the LS propagates this information to all components so that they can reregister with the CSM and update it of their current state. This process enables the CSM to fail and the system state to be completely restored when a new instance is initialized. Note that should the CSM fail, the system as a whole does not fail with it. IAS instances previously allocated remain available for use. However, until a new instance of the CSM is spawned new IAS instances cannot be allocated.

Finally, in Section 5.5 the CI (Client Interface) was presented. The CI aims to make the definition and management of Tasks in a distributed system as transparent as possible. From the user's perspective the minimum requirement is only the definition of a Task, which captures all of the information necessary to create a schedulable entity. An example of the Java code needed to create a Task and capture its result, is given in Listing 5.15 to illustrate the ease with which this can be defined. In the next Chapter this will be presented in a real application scenario.

5.7.1 Modifying the Implementation for Other Infrastructures

This implementation has been designed around the aim to not be explicitly linked to only one specific resource management middleware. There are two areas of this implementation which in the

Listing 5.15: A sample Task definition and result collection

```
import code.command.Command;
import code.ClientInterface.*;
import code.event.*;

public class myClient extends ClientApplication {

    public myClient() {
        super();
    }

    public void init() {
        //First, define the data set:
        File data = new File(``c:\\Data\\input.mat'');
        //Secondly, define the Task
        Command command = new Command(data, ``Matlab sequence'', ``id tag'');
        //Thirdly give the Task to the scheduler
        sendCommand(command);
    }

    public void receiveResult(Result r) {
        //when result is received download it
        new ResultDownloader(r.getResult().toString(),r.Tag.toString(),r.Tag,this);
        //or extract numerical results
        Vector v = r.getResults();
    }

    public void receiveError(Command c) {
        //if an error is received display it
        System.err.println(c);
    }

    public void receiveInformation(InformEvent event) {
        //This method is called when the result downloader is finished
        //Allow the super class to handle the event first
        super.receiveInformation(event);
        //Then process it
        switch(event.getType()) {
            case InformEvent.FINISHED:
            //Get file data
                File[] resultFiles = (File[]) event.getSource();
                break;
        }
    }
}
```

event of a migration to another infrastructure would need to be modified. Firstly, when no Condor installation is in place, the CRM would require a new method to disseminate job information to construct its job model (Section 5.3.1) and the infrastructure as a whole (Section 5.3.2). In addition, it would require a new IAS submission module (Section 5.3.5). Secondly, regardless of infrastructure the IAS's Matlab Control Daemon (Sections 5.1.1, 5.1.3 and 5.2.2) may need revision if the Matlab version is altered. Similarly, if the move involves a change in operating system from Windows to Linux or Macintosh the same updates are required. In addition, the Autonomic Matlab Compiler (Section 5.2.1) will need to consider different library types when performing compilation, however, this is not a significant challenge.

Chapter 6

Analysis of Neurological Diffusion Tensor Images

This Chapter documents some collaborative work with the Academic Medical Centre (AMC),[1] The University of Amsterdam,[2] and Delft University of Technology[3] and is an extension of the following paper: Ref [59]. Many of the domain-specific topics covered in this Chapter are summarized from Caan et al.'s work: Ref [204], and for further information the reader is referred to this article.

The purpose of this Chapter is to demonstrate that the approach presented in Chapters 4 and 5 can be employed in a real application scenario. Here, the application scenario is a specific form of medical image processing: the analysis of neurological images to determine and identify neurological characteristics that can be indicative of neurodegenerative diseases.

This Chapter is organized as follows: in Section 6.1 a high-level introduction into the application area is presented. In Section 6.2 the analytical context of this Chapter is presented: a study into the classification of Schizophrenia. Section 6.3 will show how this analysis was previously performed to provide additional motivation for this work. In Section 6.4 the general challenges that are faced by researchers in this domain are presented. In Section 6.5 a discussion of how the research undertaken in this thesis is applicable in this domain, and the specific benefits that can be realized to research scientists through the construction of a Matlab-enabled Virtual Cluster of Campus-Grid workstations. In Section 6.6 a new approach for this analysis using the Virtual Cluster is presented. In Sections 6.7, 6.8 and 6.9 the new approach is performance tested and discussed. Finally, in Section 6.10 the future work of this collaboration is presented and the outstanding challenges are discussed.

[1] `http://www.amc.nl` – last accessed August 2009.
[2] `http://www.uva.nl` – last accessed August 2009.
[3] `http://www.tudelft.nl/` – last accessed August 2009.

6.1 Introduction to MR-DTI Analysis

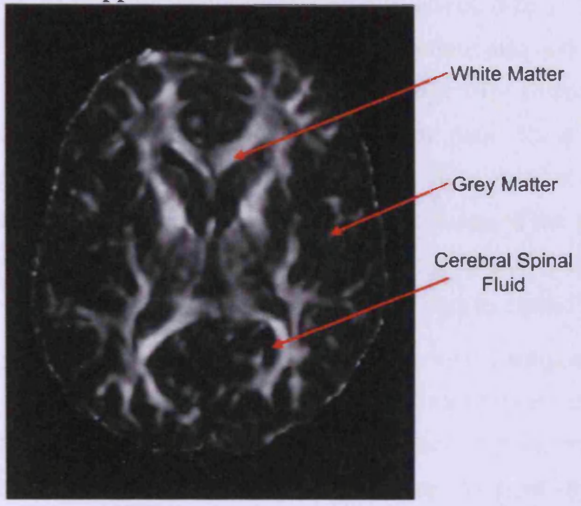
Magnetic Resonance Imaging (MRI) enables digital measurements to be taken of internal body structures, and generate multi-dimensional visualizations of areas of specific interest. Specifically, it is the hydrogen atoms from the human body's abundant supply of water that are identified. The use and availability of MRI technology enables clinicians to perform in vivo examination upon a patient. For the patient, this means they can avoid inquisitive and diagnostic surgery, and surgeons can create more detailed surgical plans and identify potential complications before the operation begins. The availability of a digital representation also allows research scientists to develop computational algorithms for the comparative analysis of specific regions within the body between patient and control groups.

There are several different flavours of MRI data, each of which are orientated toward the type of analysis or diagnosis to be performed. In this Chapter MR-DTI (Magnetic Resonance Diffusion Tensor Imaging) is used, which over the past few years has provided important insights into the orientation and structure of the brain's white matter (see Figure 6.1).[4] This is reflected in a growing number of studies adopting DTI to determine changes in brain structure and to measure the diffusional motion of water molecules [205]. Typically, the properties derived from the DTI data are harnessed to analyse the differences between diseased and healthy subjects. One such property is the Fractional Anisotropy (FA) value [206], which quantifies the local tissue integrity in proportion to the degree that nerve fiber bundles are aligned. The FA is used because it is sensitive to subtle neurological differences in the white matter. However, note that any changes in the observed FA-value of a patient can only be interpreted in comparison to a healthy control. The control must also match to the patient with regard to age, education level and handedness, as each of these attributes influence the FA value. A more detailed overview of the criteria needed for subject selection can be found in [204]. Changes in the white matter are important to identify, as they are typically indicative of neurodegenerative diseases, such as Alzheimer's, Schizophrenia and Multiple Sclerosis.

There are many areas for data analysis within a neurological context; in this Chapter, however, the main focus is on the developement of analytical models for comparative analysis. Here, example application scenarios are: the calcualtion of the risk of developing a disease or the identification of a region of the brain that could be indicative of a disease. These analytical tools and techniques can also be used across a range of conditions such as the affects of drug use (see: [205]) or the analysis of diseases such as dementia (see: [207]) or Alzheimer's (see: [208]). This means that many of the tools and techniques that are presented in this Chapter can be generalized to facilitate other areas of research in medical image processing.

[4]White matter is one of the three main components constituting the brain. In short, it is composed of bundles of fibres (axons), which facilitate the transfer of messages between areas of grey matter within the central nervous system. Grey matter on the other hand, are (in board terms) the processors of sensory messages in order to create a response to a stimulus.

Figure 6.1: Anatomy of the brain, depicted using an axial cross section and created by a visualization of the FA. White matter fibers are highly aligned, and have a bright intensity. Grey matter is more randomly oriented, and therefore has a lower intensity (FA is nearly zero). Cerebral spinal fluid has no structure at all and therefore appears black.



6.2 Analytical Context: The Classification of Schizophrenia

In this Chapter the main focus is in the classification of Schizophrenia, which is a cognitive disorder occurring in about 1% of the world's population. The first symptoms usually occur in early adolescence, with signs of abnormal social behaviour and hallucinations. Research into Schizophrenia is interested in determining firstly, whether the disease affects the brain tissue, and secondly, if so, which regions are affected. It is generally assumed that Schizophrenia affects the white matter structure in the brain [209]. The main medical findings have identified specific regions of the brain which are suspected to deteriorate or change anatomically as a result of the disease and are measured as reductions in the observed FA-value [210–217]. Currently, exactly which regions are most relevant is still unknown.

Computerized image processing of brain imagery has led to improvements in the understanding of many neurodegenerative diseases such as Schizophrenia. However, disseminating this information into usable data is a complex process, because the brain's structure is highly inter-connected and interacting. Typically, image processing approaches that investigate Schizophrenia use region of interest (ROI)-analysis or voxel-based analysis (VBA) [214] on data which has been annotated by an expert. These methods treat regions of the brain independently and thus disregard the connectivity of the brain. Research by Caan et al. [204] has resulted in an algorithm that models correlations between different brain regions using a pattern recognition approach. In the adopted pattern recognition framework, a classification error is computed, which indicates the ability to distinguish between the schizophrenics and healthy controls. A robust estimate of the classification error is computed in a series of independent Tasks that can run in parallel. In this research, parallel execution was hindered by the high costs of Matlab licenses and the unavailability of dedicated resources that are capable of running licensed Matlab sessions. Therefore, computational analysis had to be performed sequentially, which resulted in long running times.

The goal of this analysis is to locate regions of the brain that clearly differentiate between schizophrenic patients and healthy controls. In the case of affected brain tissue, signal changes in DTI scans are to be expected. As these changes are subtle, patient and control groups are compared to gain statistical power. Imaging artifacts, noise and heterogeneity within the patient and control groups hamper the detection of pathological processes in the data. Caan et al.'s method adopts a pattern classification approach using supervised techniques. The approach is designed to robustly answer the questions *if* and *where* a significant change in the scans of the patient and control group exist. Note that the relevance of the results obtained with the proposed method resides in analyzing the given population for clinical research purposes, and not (yet) to classify new incoming patients.

The classification process consists of two steps: (1) Principal Component Analysis (PCA) and (2) Linear Discriminant Analysis (LDA). PCA is used to extract representative features, known as Principal Components (PCs), from the data. Here, the number of principal components (*nPCs*) to be used needs to be optimized. If too few are selected, relevant information, i.e. specific anatomical regions of interest (those indicative of the disease) will be lost, and a biased estimation will be produced. On the other hand, should too many be used, then small variations (for example, noise in the data) will be over emphasized and therefore incorrect regions will be highlighted as significant. LDA is then used as a cost function to minimize classification error by assessing whether, and if so where, a significant difference can be found between the patients and controls.

To perform the analysis, a training set of a subseries of DTI-volumes with known class-labelling is used to train the LDA-classifier. This classifier is subsequently applied to samples in an unseen test set. The classification error is defined as the relative part of the test set that is incorrectly classified. Random classification would yield an error of 50%. Based on the group sizes, an upper boundary on the error of 35% is concluded to indicate a significant difference between patients and healthy controls. The estimated classification error should be independent of the composition of training and test sets, therefore a cross-validation strategy is adopted. Essentially, this involves splitting the total dataset (all DTI-volumes) into different groups that are partially used as test set, and the remainder of the volumes as the training set. Different compositions of groups lead to repeated cross-validation, and the computed classification errors are averaged for all. Typically, a ten-times repeated five-fold cross-validation is performed. Such an approach creates multiple coarse-grained Tasks that can be executed independently. In addition, data sets are required for each cross-validation step. Figure 6.2 illustrates the entire workflow from image acquisition through to the evaluation of the results. Here step *c* is where the classification is performed and is also the part of the workflow which is to be parallelized. In fact, many MRI analytical workflows have a similar structure: acquire images, spatially normalize the data, process and interpret the results. Therefore, the common use case is also captured in this Figure, and consequently, the applied implementation discussed in this Chapter is applicable to any other area of analysis with a similar use case. A discussion of the parallelization of step *b* is presented in Section 6.10.

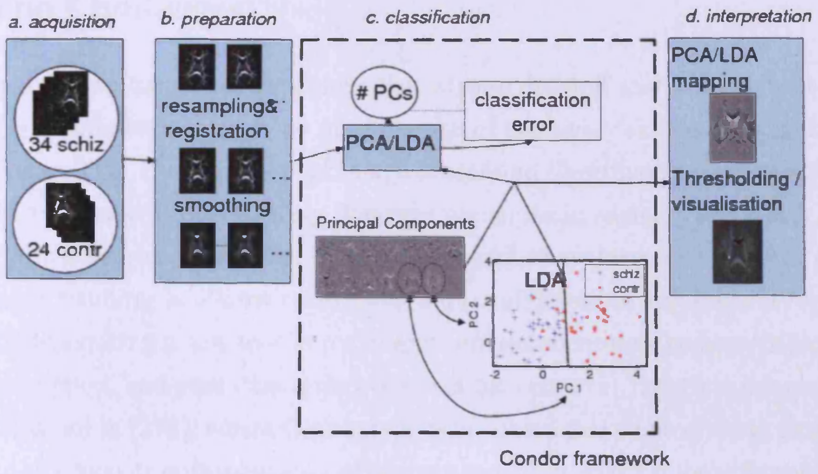


Figure 6.2: Workflow of the analysis indicating the parallelized steps.

6.3 Previous Approach

The previous approach for this work was sequential execution within a licensed Matlab session. Only one parameter needs to be tuned: *nPCs*, which gives rise to a parameter sweep. This means that the cross-validation step has to be performed repetitively for a range of parameter values. The execution of the analysis is CPU intensive, and therefore the user's workstation is blocked for the duration of the analysis, preventing them from performing any other work during this time. Normally, a high-performance infrastructure such as that provided by the VL-e project [133], which is available at the AMC, could be used to run the computations. However, running (licensed) Matlab sessions is not possible within this infrastructure, which holds for public Grids in general.

There were challenges in the compilation of the two third party Matlab toolboxes (PRTools: Pattern Recognition Tools [201] and DipLib: the Delft Image Processing Library [202]) that are used for this analysis, as neither enables a simple compilation. Both have initialization scripts, which must be called to prepare and setup the toolbox. In addition, DipLib uses approximately 120 dynamic loading libraries, which must be converted to statically bound libraries in order for the compiler to incorporate them. The complexity of the compilation was sufficient to mean that a compiled version was too difficult to produce.

When executed sequentially, this analysis required 90 minutes on an Intel Centrino Duo 1.83GHz, with 1GB of RAM and Windows XP Pro SP3. The version of Matlab in use was R2006a. Note that as an inherently single threaded environment the Matlab could only make use of one of the two available cores. However, despite this Matlab itself was unusable for this period of time.

6.4 Domain Challenges

There are three core challenges to researchers that wish to develop analytical algorithms for neurological data: **(1)** It is difficult to recreate the expertise of the highly skilled clinicians within image processing software.[5] **(2)** The validation of image processing algorithms requires a substantial effort due to the rigid requirement for reliability. Inherent variations in anatomy and a lack of, or limited, ground truths only exacerbate this situation. Consequently, to evaluate an algorithm, extensive data sets must be used resulting in a large consumption of, and therefore requirement for, computation capabilities. **(3)** Integrating a new tool into a clinical environment may be impossible without third party or vendor support, and even then it may obstruct the operator. These challenges are discussed in much greater detail in [218], where Olabarriaga et al. stated that some of these challenges can be appeased when an adequate computational infrastructure is available. Of these three core challenges the first is inherent to the domain, however, the remaining two overlap considerably to those discussed in Section 1.3. The overlap is in regard to issues of user ability (discussed in Section 1.3.3), achieving parallelism (discussed Section 1.3.6), determining the resource requirements of an application Section 1.3.5) and the restrictions and requirements placed upon the operating environment (Section 1.3.7).

The challenge of user ability is perhaps one of the more complex issues to resolve, as the background of users and researchers is largely clinical as opposed to computer science or informatics. Therefore, a critical component within this domain is the collaboration between many scientific areas, for example Radiology, Neurology, Psychology, Physics and Computing [219]. Even when these collaborations are formally defined, it does not guarantee that co-workers are adequately competent with the necessary distributed computing tools to perform the analysis. It is here that the challenge of how parallelism is achieved comes to the fore and is also coupled with several trade offs. Firstly, a fine grained abstraction is likely to inhibit progress, as the user cannot be expected to use the techniques needed to achieve smooth execution. In addition, a fine-grained architecture may not be available. Secondly, a computer scientist cannot be expected to understand and break down the computational models into units of work that can be executed in parallel. Here, a coarse-grained approach is much easier to facilitate, but can lead to an inefficient use of resources, as demonstrated in Chapter 2. Finally, medical researchers are not always willing to, or, more often than not, cannot allow their data to leave their work place because of patient confidentiality and the sensitive nature that the data can have. This can even be the case after the data has been anonymized. Such a requirement quickly reduces the options for sources of computational power for a parallel solution.

Once these almost administrative challenges have been addressed, the identification of resource requirements of the application as a whole must be addressed. All too often in the literature approaches document scenarios where adding more resources actually reduces performance. Anecdotally, end users are known to have a biased or inaccurate estimation of their resource requirements, which could therefore lead to a wasteful utilization of resources. This scenario is made yet more dif-

[5]Note that this is after the image data has been rendered, as image processing techniques are essential to the rendering of the raw data generated by the MRI scanners.

ficult by the size and quantities of data that are in use. In addition, the amount of time the data needs to be processed in relation to the time needed to transfer it can be strongly unbalanced. Furthermore, and as mentioned above, data can seldom be transferred to a remote location.

The requirement for in-house processing raises the challenge of performing the analysis within the confines of the researchers' normal working environment. Here, software licensing also has a large part to play in the adopted application solution. Many researchers rely on Matlab due to its abilities in rapid prototyping and the wide availability of relevant third party toolkits. For example, SPM (Statistical Parameter Modelling) [203], which is one of the more popular toolkits for the analysis of functional MRI (fMRI), is made freely available by the University College London to the [neuro]imaging community, to promote collaboration and a common basis for analysis across laboratories. DIPLib and PRTools are provided free of charge for academic institutions for the same reasons. The problem is that whilst these toolboxes are license free, they are built upon and therefore dependent upon a Matlab foundation. Consequently, a Matlab license must be available for each node that partakes in the execution of an application, which quickly increases the costs of parallelizing existing applications. An individual license costs, as previously mentioned in Section 2.3, between £525 and £6300, but cluster licenses can be several orders of magnitude higher. Here, Matlab compilation is the solution, however, as discussed in the critique of JavaPorts in Section 3.1.1, this is not a straightforward process to perform well.

There is also the question of application steering and how Tasks are to be injected into the system. There is no direct requirement for Matlab to act as the head application in this particular scenario, but it would be a desirable attribute. In addition, the ability to steer the Virtual Cluster directly from Matlab would have a high utility for other applications, specifically the application that will be presented in Section 6.10. On the outset such a requirement does not appear that difficult, but experience in the use of Matlab, an in-depth knowledge of its architecture and the potential requirement for low response times from Matlab (to prevent bottlenecking) present additional challenges. Firstly, Matlab is built upon an interpreted language, which slows it down considerably. Secondly, its graphical and console output is inefficient; Matlab can perform most simple image processing functions several orders of magnitude quicker than it can display their results. Finally, Matlab's memory management and Java handling can be abhorrent, and if an application pushes Matlab too hard it will lock up, crash and file a useless thread dump.

6.5 Relevance of this Research to the Neuroimaging Domain

The research documented in this thesis and the implemented software solution – the transparent construction of a Virtual Cluster of Matlab-enabled workstations (Chapter 5) – are significantly beneficial to neuroimaging domain. Below is a discussion to support this claim in relation to the challenges introduced in the previous Section.

The CI (Client Interface Section 5.5) enables a user to specify no more than what analysis they wish to perform and upon what data it should be performed. The CI will take this information and generate a set of schedulable Tasks for the user (see: Section 5.5.2) and execute it transparently. The CI can even be incorporated into a Matlab session (see: Section 6.6.1) for added transparency. Ultimately, this means that the user's knowledge of what resources they are using or how they are used is minimal. In addition, the granularity that this research encourages is a coarse-grained data abstraction (see: Section 4.9.1). This means that complex and low level approaches for parallelizing existing code are not required. Instead, users can work at the function level, i.e. one Task relates to a chain or sequence of function calls upon the specified data (see: Section 5.5.2). As a result, the call for remote execution can be made almost[6] identical to what would be entered within a local Matlab session.

On a similar note, one of the limiting factors mentioned in Section 6.4 is in regard to software licensing, specifically Matlab, and the complexities of creating a license free (compiled) solution. The Virtual Cluster as a whole, has no reliance upon Matlab licenses, despite one hundred percent of an IAS's (Section 5.2) functionality stemming from Matlab. In fact, should it be necessary even an Application Manager based within Matlab could be compiled to remove all licensing restrictions. The IAS is also a general and extensible software component that caters for diverse and heterogeneous applications through its novel Autonomic Matlab Compiler, which captures entire Matlab toolboxes (see: Section 5.2.1). This ability completely alleviates the high costs of running a parallel software solution and most of the effort needed to perform the compilation. In addition, by removing the prerequisite for licenses, the number of processing nodes is no longer dependent upon the number of licenses.

To determine the number of resources that should be used is also not an issue for the user. Many approaches in the literature place this requirement upon the user. However, the User's Requirements Model (Section 4.5) models the ability of the user's application to consume resources. This abstract estimation is then turned into a acquisition of resources based upon the current resource availability and a fault-aware acquisition model (Sections 4.4, 4.4.1 and 5.3.3) on the user's behalf.

When the system does acquire resources for the user, it does so using the daemon model (Section 4.3.1), which facilitates the construction of a Virtual Cluster and its interactive use. In short, this means that a Task experiences a much reduced overhead as a consequence of remote execution. Therefore, a Task can be minutes or even seconds in length and its makespan near its actual runtime and not be distorted by the disproportionate overhead traditional approaches can apply.[7] In addition, all fault management, which is the consequence of resource volatility (Section 1.3.1 and Chapter 2), is handled transparently for the user by the CRM (Section 5.3). This provides the user with two key abilities: **(1)** the ability to concentrate on their research and not administrate a potentially complex and volatile distributed system, and **(2)** the ability to differentiate between a fault in their code and workstation error. Note that the current Campus-Grid infrastructure could not guarantee this distinction (see: Section 2.3). The result is a strong orientation toward resource reliability as once an IAS instance becomes available, it is guaranteed to function correctly for its lifetime. If, due to its

[6]It is almost identical because a small modification to the way in which data is presented is necessary. This will be further explained in Section 6.6.

[7]This will be demonstrated in Chapter 7.

volunteer nature, an IAS's host workstation is revoked and the IAS is terminated, the system will autonomously reconfigure and leave the user unaware of the failure.

Many clinical researchers are bound by laws governing the transfer of data, and are therefore either unwilling or prevented from using external sources of computational power. This approach enables them to utilize their local resources by building upon the traditional model of opportunistic computing.

6.6 Implementation of a New Approach

In order to port any application into a parallel context it is inevitable that some changes to its code are required. This was also true for this analysis, albeit only slight. All data loads were separated from the execution code[8] and all console and visual output was also suppressed for improved performance. No further modifications were required. To capture results the MI (Section 5.1.2) encapsulates results in Matlab's `.mat` format so that they can be easily imported into a Matlab session for analysis. When results are being generated, the MI uses variable names ascribed by the function's output declaration, using the lookup table generated during compilation (see: Section 5.2.1) and therefore remains consistent with the original code. This approach also provides a naming standard that is both predictable and intuitive.

The two external toolboxes used in this analysis also present their own individual challenges. However, in order to create a general solution, the entire toolboxes have been added to the IAS. This means that a single instance of the IAS can accommodate Tasks, which use any combination of both these toolboxes without recompilation. No other compiled Matlab approach in the literature has documented this ability.

PRTools

Compiling the entire PRTools toolbox was relatively simple, the IAS's Autonomic Compiler (Section 5.2.1) was directed toward the PRTools directory. Consequently, it identified several problematic Matlab scripts containing illegal characters and unbalanced parenthesis. However, once these were removed or corrected[9] PRTools was available.

One other accommodation was required, as PRTools contains a setup script which is called transparently in regular Matlab. However, when PRTools is compiled, this process is not performed, and therefore PRTools was modified to take this into account. To determine if the setup process has occurred, PRTools uses a `global` boolean variable. In a compiled version of Matlab the workspace containing this variable is reset for every call to the engine. To accommodate this, the variable was,

[8]Data loads need to be separated because an explicit data load within the application's code may fail. In order to guarantee that data dependencies are retained and that the same file is not transferred twice, filenames are converted into unique identifiers. Therefore, any explicit data load that the Client Interface (Section 5.5) could not identify, for example a function which is called by the user's code and expects a specific name, would fail.

[9]Approximately 30 of the 500 PRTools functions were diagnosed with errors and remedied as a result of the compilation process.

firstly, made `persistent`, and secondly, its scope was changed so it can be controlled by an MI.

DIPLib

The addition of DIPLib was just as straightforward as that of PRTools. DIPLib uses 120 precompiled dynamic loading libraries (`DLLs`), which increase speed and enables some of the package to be used outside of Matlab. Regular Matlab accesses these libraries transparently, but the Matlab compiler requires a static library (`LIB`) in order to link against the DLL. Therefore the Autonomic Compiler generated all the necessary LIB files and then proceeded to compile DIPLib.

In a similar fashion to PRTools, DIPLib also requires setup to function correctly. However, unlike PRTools this process is not transparent. The function `dip_initialise` must be called prior to the use of any DIPLib functionality. It is not possible to predict what an IAS will do or when during its life time. Similarly, checking each Task Sequence involves unnecessary string manipulation. For these reasons `dip_initialise` is called the first time that the MI is invoked. Once these two steps were taken, DIPLib had been incorporated into the IAS.

6.6.1 Interacting with the Virtual Cluster

This application is essentially a coarse-grained embarrassingly parallel problem. Once the necessary Matlab functionality had been made available to the IAS, no core extensions to the architecture were required to run it, with the exception of an Application Manager. Here, there are two options: **(1)** a simple Java program that defines the Tasks needed for execution and invokes the CI (Section 5.5) and inherits all the necessary functionality to do so, or **(2)** a Matlab-based Application Manager, which hides most of the administration of initializing the CI and passing Tasks to it for execution.

A Java-based Application Manager

In [59] the first scenario was used to perform the analysis. To demonstrate the simplicity of interacting with the system, Listing 6.1 shows the Application Manager developed for this analysis. Note that the Application Manager actually uses less lines of Java than the number of lines required to define the same application using a Condor submit script. It also has the benefits provided by the use of the Virtual Cluster (which will be discussed in Section 8.1.3) rather than the challenges of administration as outlined in Chapter 2.

Matlab Integration

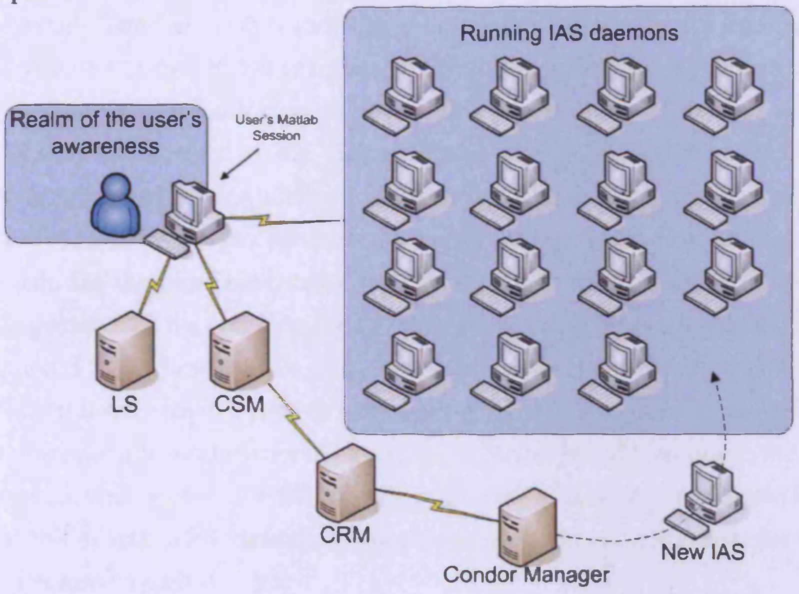
In order to integrate an Application Manager into Matlab, a slightly different conceptual model (see Figure 6.3) to that presented in Chapter 4 is created. In this case Matlab is in control of when Tasks are created and given to the system for execution as opposed to a user.

The Java-based Application Manager fell under the category of explicit parallelism, which is characterized by the presence of explicit constructs in the architecture to define parallel calls [93, 220]. In this case the Matlab Application Manager describes the way in which parallel computation

Listing 6.1: Java Application Manager

```
// import statements removed for presentation

public class MRIClient extends ClientApplication {
 pubic MRIClient () {
   super();
 }

 public void init() {
  // Define the data set and sequence to perform:
  String data = ``c:\\Data\\schiz_study_fa_reg1mm_int16.mat'';
  String sequence = ``pcaldacrossval_cardiff(load(''+data+``),iterations)'';

  //Create a parameter sweep using the Task sequence
  VariableSequencer vs = new VariableSequencer(sequence);

  //Identify the sweeping parameter: iterations
  Variable iterations = new Variable(iterations);

  //set parameter value range from 1 to 20 (inclusive) with a step of 1
  iterations.addValueRange(1,20,1);

  //add sweep variable and compute the combinations
  vs.addVariable(``iterations'');

  //schedule all combinations as new Tasks denoted here as a MRICommand
  while(vs.hasNext()) {
   sendCommand(new MRICommand(vs.getCurrentSequence(),vs.getTag()));
   //note an automated Task identifier denoted here as Tag
   vs.next();
  }
 }

 public void receiveResult(Result r) {
  //when result is received download it
  new ResultDownloader(r.getResult().toString(),r.Tag.toString(),r.Tag,this);
 }

 public void receiveError(Command c) {
  //if an error is received display it
  System.err.println(c);
 }

 public static void main(String [] args) {
  new MRIClient();
 }
}
```

Figure 6.3: New conceptual model. Here, the user is no longer aware of the core components outlined in Chapter 5



takes place, and therefore provides the basis for flexibility in the approach. However, the cost of flexibility is the additional work required to parallelize the code and write the Application Manager itself. Writing an Application Manager is not considerable effort, it is arguably easier than writing the equivalent Task description as a collection of submit scripts, and then managing the application. However, it could be seen as an unnecessary administrative task for the user to perform. For this reason, the Matlab-based Application Manager was created.

Implicit parallelism allows programmers to write their applications without any concern for how the parallelism is achieved or the parallel calls made. Exploitation of parallelism is, instead, automatically performed by a compiler or the runtime system. The latter is the case here. The advantage is that the parallelism is transparent to the programmer. However, extracting parallelism implicitly requires a great effort by the system developer and it is very difficult to create a general approach. This becomes all the more challenging when multiple data dependencies exist.

The Matlab-based Application Manager adopts a hybrid of the two models and aims to find a balance between the two. There are three factors that prevent this approach from having a true implicitly parallel model. **(1)** The user flags a function as one which should be executed on the Campus-Grid. **(2)** A handle for a result is returned instead of a result. **(3)** Data loads need to be converted to strings and have full file paths. This enables the underpinning architecture to handle explicit data dependencies. For example `'load(c:\myData.dat)'` instead of `load(myData.dat)`. Ultimately, this means that users must change their code to use this approach; however, changes are only minor. A hybrid approach was chosen over one or the other because it enables a generalizable solution much more easily than an implicitly parallel approach, whilst retaining some of its benefits. By drawing on an implicitly parallel approach, it also makes parallelism easier to achieve for the user than an explicitly parallel equivalent.

In order to flag a Matlab function as one which should be executed on the Campus-Grid instead of locally, the Matlab function `parallelize` was created specifically for this purpose. An example call is: `parallelize myFunction`. When this is performed, all the necessary hooks to the underlying architecture are built transparently. This process turns all future calls of the user's function into a Task to be executed on the Virtual Cluster with immediate effect. In order to achieve this, the following processes are undertaken: **(1)** start the Matlab Application Manager if it is not already running. **(2)** Create a polymorphic equivalent of the user's function, which is higher in the Matlab search path, and therefore executed instead of the user's function for all future calls. **(3)** Extract the input arguments of the function for the new overloaded version, using the hashtable built by the AC (Section 5.2.1),[10] and build a map, which translates input arguments into an output file in preparation for their transfer to a remote workstation. This only applies to numerical arguments, as they require more complicated means of storage than String-based arguments; the argument type is determined at runtime. **(4)** Add the functionality to create a Task object and give it to the Application Manager. **(5)** Include the functionality to query the Application Manager for the handle and return it as the function's result.

Results are collected using a different approach to regular Matlab, which occurs as a consequence of using remote resources. When a *parallelized* function is called and a Task is created to reflect this, a handle to the result is returned instead of the result itself. If the result was returned at this point, the call would be a blocking call, therefore making the use of multiple remote resources redundant. By returning a handle, further Tasks can be injected, making parallel execution more feasible and beneficial. From a programming perspective the only difference is that results are collected when they are needed, but not necessarily when they are available. The idea here is to enable Matlab to give the scheduler (via the Application Manager) all Tasks for a processing block in bulk and then later collect the results only when they are needed for generating new Tasks. A blocking call is only generated when a result is requested, as it is assumed that without the result the application cannot progress. This also presents the potential for future development work to enable an application to move on without a result if it is either no longer required, or if without it the application as a whole is not largely affected. The latter could be true in optimisation problems, where one dropped iteration is not largely detrimental to the performance of the algorithm as a whole. Another example would be in genetic algorithms or parameter sweeps, where a solution is presented that makes others redundant.

When a result is requested, the handle is used to access the result file transparently, which will have been downloaded from the IAS by the Application Manager when the result was received. At this point the function that was executed, and therefore created the result, is identified. It is necessary to know exactly which function was called in order to determine the output arguments, and, more importantly, their order. Once this has been identified the `.mat` results file is loaded,

[10]If the function has not previously been incorporated into the IAS, the user is alerted. The compilation process can be performed at a later date. In such cases the function is inspected by the AC so that the process of generating a polymorphic equivalent of the user's function can continue.

the necessary variables are created, instantiated and returned. The process of calling a parallelized function and receiving a result is summarized in Figure 6.4.

The Application Manager extends the basic structure of that used by the Java-based Application Manager to enable the functionality presented above. There are two additions to the basic Application Manager required to achieve this: **(1)** a method to create a unique handle for a result. **(2)** The means to map the handle to a result, or if an error occurred, to inform Matlab so that the necessary action can be taken.

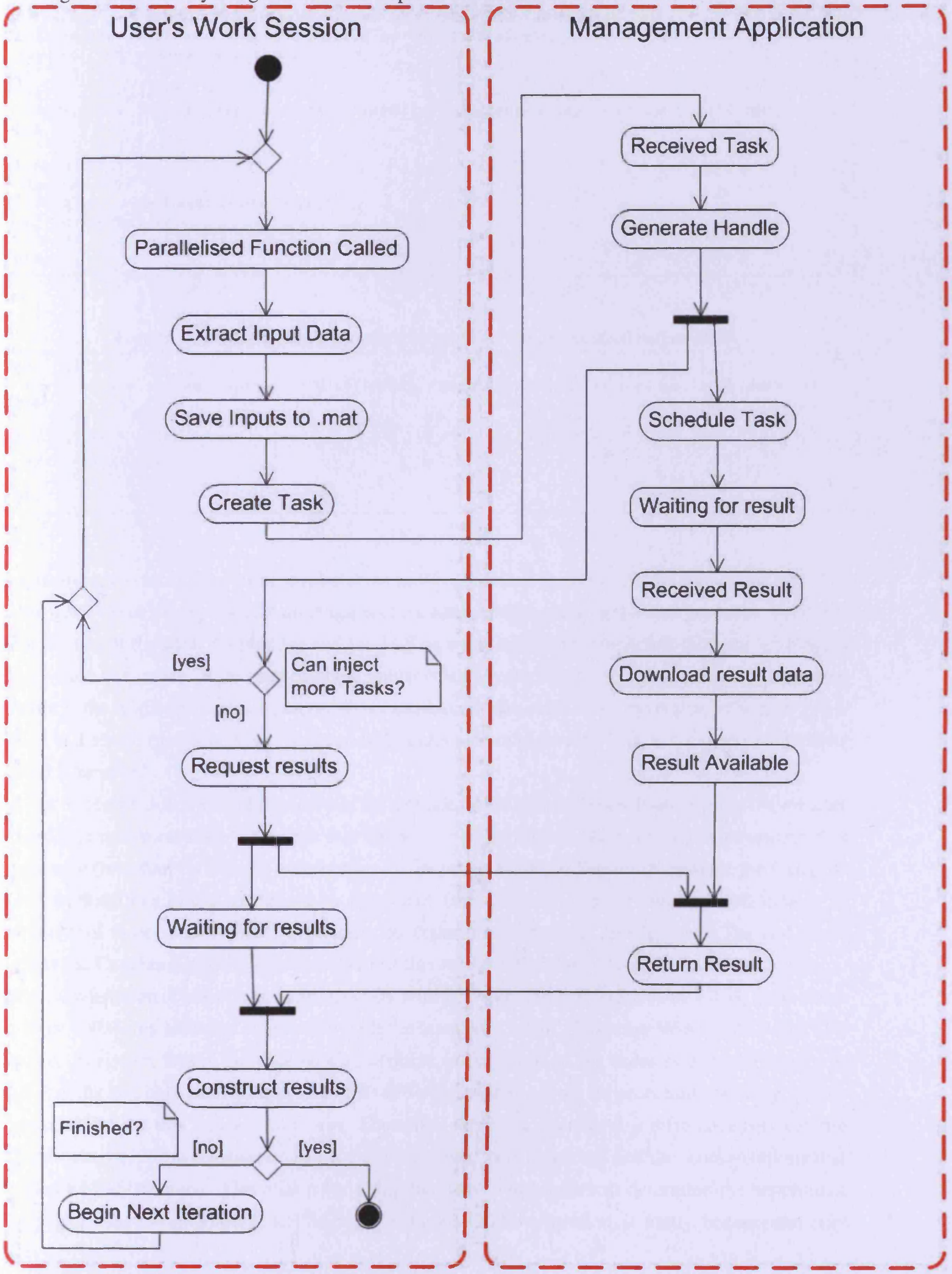
The key difference between the Java-based Application Manager and the Matlab-based Application Manager is in how the user describes their application. To use the Matlab-based Application Manager the user would perform steps similar to that shown in Listing 6.2. For a comparison, the code required to perform the analysis sequentially within the local Matlab session, i.e. the previous approach, is presented in Listing 6.3.

6.7 Evaluation Methodology

The PCA/LDA analysis constitutes of 20 independent iterations (one per Task) and the same data set is used for each iteration. The data set contains the 58 DTI scans, which have been compressed to reduce transfer costs and is 20MB in size. The results of the analysis are not needed immediately after completion and were later aggregated and analyzed offline to the determine their accuracy. During this work three scenarios are examined for the analysis of the DTI data. Each scenario has been chosen to illustrate a specific set of qualities of the Virtual Cluster and the improvement in the AM that was achievable through its use. The three scenarios are: **(S1)** The Virtual Cluster is built from scratch, i.e. no IAS jobs have been submitted by the CRM before the Application Manager began. This scenario will demonstrate the rate at which the system can be initialized, and that despite using significantly older and lower specification workstations, a considerable improvement in performance can be achieved with little effort on the user's part. **(S2)** Some IAS instances have been previously used and are still available. Here, the value of being able to recycle previously used resources, as well as the adaptive qualities of the approach, will be demonstrated. This could occur as a result of either previously undertaken work where the system had not been shutdown, or through the preinitialization if IAS instances to enable spare capacity on demand. **(S3)** More than enough IAS instances are available for use by the application. When enough Campus-Grid workstations are available, this scenario occurs naturally after a few runs of the analysis. This occurs as the CRM (Section 5.3) responds to the demand placed upon it and also attempts to acquire spare capacity for the CSM (Section 5.4). This final scenario will demonstrate the maximum reduction in AM achievable and serve as a reminder to the heterogeneity of the workstations in use.

In this analysis the two performance metrics of interest are: **(1)** the accuracy of the analysis in comparison to the previous approach, and **(2)** the improvement in application makespan as a consequence of using the Virtual Cluster. Ensuring the accuracy of the results is very important because a different execution model and result representation is in use. Similarly, as this is a medical

Figure 6.4: Activity Diagram for calls of parallelized functions and the use of their results



Listing 6.2: Matlab calls required to perform the application on the Virtual Cluster

```
%create a polymorphic equivalent of the pcaldacrossval_cardiff function,
%all future calls will now be executed on the Virtual Cluster as a Task
parallelize('pcaldacrossval_cardiff');
for i=1:20
 %get result handle for each call
 handle[i] = pcaldacrossval_cardiff('load(c:/Data/schiz_study_fa_reg1mm_int16.mat)', i);
end

for i=1:20
 %get results
 [results] = getResults(handle[i]);
 %process results...
 ...
end
```

Listing 6.3: Matlab calls required to perform the application sequentially

```
for i=1:20
 results[i] = pcaldacrossval_cardiff(load(c:/Data/schiz_study_fa_reg1mm_int16.mat), i);
end

for i=1:20
 %process results...
 ...
end
```

application, the ramifications of inaccuracies in the generated results could be significant were this approach ever to be employed in an diagnostic scenario as opposed to a research scenario. To ensure that the use of the Matlab compiler and the IAS have not negatively impacted upon the application as a whole, the results of the analysis being performed are already known. The second performance metric is the Application makespan (AM) is calculated in the exact same manner as in Section 2.4.1, i.e. it is the time from when the Application Manager submits the first Task to the time that the final result is received.

In Section 6.3 it was highlighted that on a single workstation, the analysis requires 90 minutes to produce usable results. At the time that this work was performed, the workstation in question was quite new (less than six months old) and top of the range when purchased. In contrast, the Campus-Grid workstations in use can be up to four years old, as the University replaces workstations in rotations of three to four years, depending on departmental policies and funding. The end result is that the Campus-Grid workstations used in this analysis had a considerably lower specification. Here, workstations could be Intel Pentium 3s with between 256 and 512 MB of RAM, have either a 10 or 100Mbit/s Ethernet connection with the standard Cardiff University Windows XP Pro SP2 image. However, it was possible to also acquire workstations in the order of a Intel Pentium 4, 1.8–2.4Ghz and between 512 and 1024 MB of RAM, with the same Ethernet and operating system composition, but this was not common. Therefore, there is a considerable difference between the specification of the workstation used for the sequential benchmarking and the workerstations that hosted an IAS instance. The reason for using the user's workstation to determine the benchmark for performance as opposed to an "*average*" Campus-Grid workstation, is firstly, because the only way to accurately demonstrate any improvement and therefore benefit of the approach, is to make this comparison relative to the user's normal working environment. Secondly, determining what the "*average*" Campus-Grid workstation is relates to a significant research challenge in itself.

Estimating an accurate speed up value for this analysis is difficult to estimate objectively (and fairly), as there are too many complexities to consider. For example, the heterogeneity of Tasks and workstations, workstations' previous work (giving rise to the potential for caching) and background processing all make quantifying speed up quite challenging and potentially subjective. For these reasons reductions in application makespan are considered as opposed to speed up values.

6.8 Results

Figure 6.5 shows the mean classification error plotted against the number of principal components for the ten-times repeated five-fold cross-validation (yielding 50 runs in total). For this particular analysis with 58 subjects, an error below 35% was considered significant. Mean errors decrease from 28% for 1 PC to 22% for 15 PC's, after which it stays approximately constant; for larger *nPCs*, error reduction is not significant, so 15 PCs can be regarded as optimal for further classification. The corresponding thresholded map is overlaid to the FA volume and displayed in Figure 6.6, which is what would be presented to a radiologist for interpretation. These results are consistent with Caan et al.'s previous findings in [204], which demonstrates that despite changing the execution model, the result representation and moving to a compiled version of Matlab, accuracy has not been compromised.

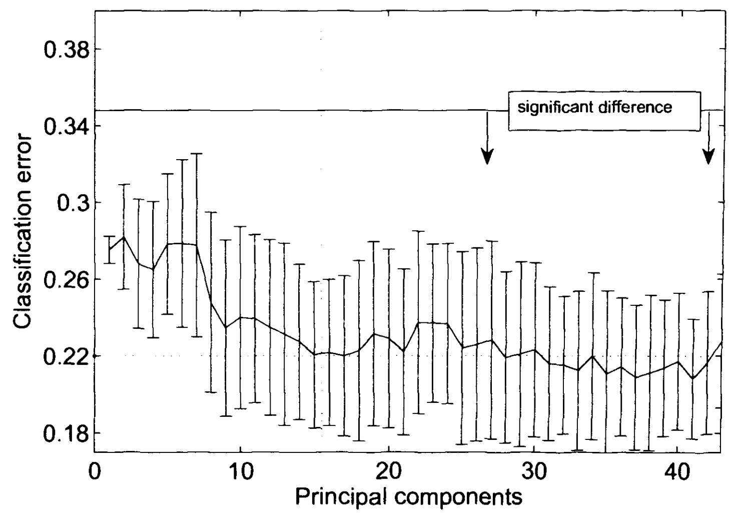


Figure 6.5: Mean classification error as function of *nPCs*, with error bars denoting the standard deviation respective to the mean.

Figure 6.7 shows a visualization of this analysis for execution scenario one (S1), where no

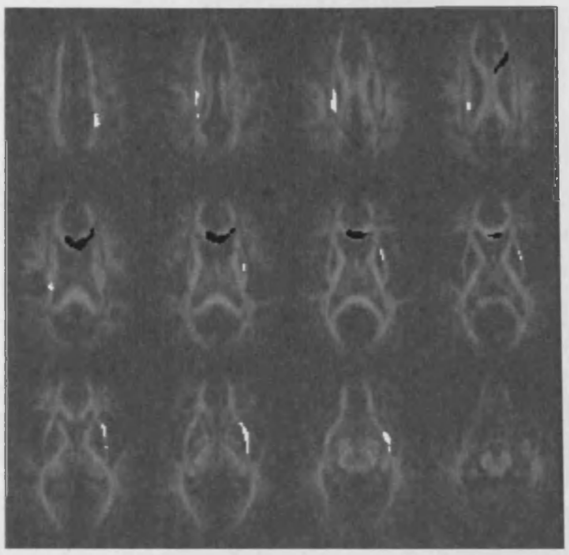


Figure 6.6: The thresholded map for *nPCs*=15, displayed as white and black regions for increased and decreased FA for patients, respectively, with the FA on the background. Slices are displayed from top to bottom.

IAS instances were available when execution began. In this scenario the system, and by extension the user, must wait for the CRM to schedule some IAS jobs and then for these IAS instances to successfully initialize before Task execution can begin. Figure 6.7 shows that two IAS instances were initialized, and therefore useable, within the first 60 seconds of the experiment's runtime, and further instances became available in the following 120 seconds. This demonstrates the speed with which the construction of a Virtual Cluster of Matlab-enabled workstations can begin, despite the difficulties in running compiled Matlab jobs that was documented in Section 2.3. It is also clear from the visualization that IAS instances are acquired sporadically, which emphasizes the heterogeneity of the workstations upon which they are running, the variability in which workstations are available and the nonuniform distribution of jobs by Condor. As the number of IAS instances reaches 10, the scheduler employs Task reallocation (or load balancing) to take advantage of the new resources, as no Tasks remain unscheduled, but are either complete, executing or waiting in an IAS queue (denoted as buffered in Figure 6.7).

The user waited 15 minutes and 35 seconds for their analysis to complete. Considering that a maximum of 13 IAS instances were used (the 14th, acquired around 480 seconds, was not used), this does not initially show good performance. However, consider the following: **(1)** that number of IAS instances was not a factor of the number of Tasks, **(2)** that during runtime new resources were gained (an average 8 IAS instances were used)[11] **(3)** that the workstations had to be allocated by Condor and then successfully initialized before servicing a Task, **(4)** that no optimised allocation strategy was in place during this work, **(5)** the heterogeneity of the Tasks and workstations in use: the average Task

[11]To determine the average number of resources, the underlying monitoring framework samples the number of IAS instances in use once per second while Tasks are still to be executed and the number of IAS instances is not zero.

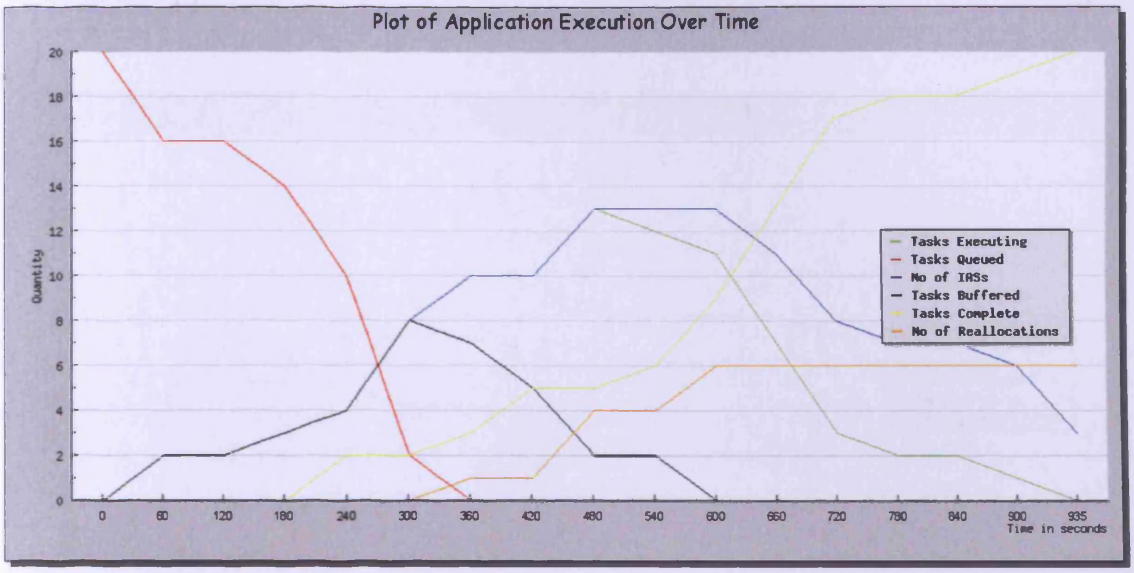


Figure 6.7: Plot of application execution when no resources were available when the application began.

completion time was in the order of 4 minutes, but minimum and maximum execution times are: 3 and 7 minutes respectively, and **(6)** that the 90 minute benchmark for sequential computation was performed using a workstation that was potentially far more than the average Condor node. Given these conditions, an 83% reduction in the AM, with marginal effort on the user's part, is respectable.

Figure 6.8 shows the analysis for scenario two (S2), where 11 IAS instances were available prior to execution and could therefore be recycled. The scheduler initially queues Tasks, but consequently reallocation occurred once to accommodate the eleventh IAS. Between 300 and 330 seconds of the runtime two further IAS instances became available. Here, the requirements model (Section 4.5) had illustrated the potential for yet further resources and the combination of the CSM (Section 5.4) and CRM (Section 5.3) yielded further IAS instances. This caused further load balancing to provide work for the new IAS instances. However, reallocation occurred three times, which indicates that one of the original eleven IAS instances had also run out of work, and was therefore provided with additional work from a slower IAS. On average 8 IAS instances were used, as in the first scenario, but the overall execution time has now dropped to 12 minutes and 45 seconds. This reduction illustrates the additional benefit of spare capacity within the system, and how the recycling of a set of resources established some time in the past (either from previous runs of an experiment or simply through instructing the CRM to have some resources always available) can further improve performance. This scenario reduced the overall application makespan by 86% and has improved upon scenario one by 18%.

Figure 6.9 shows analysis for scenario three (S3), where more than 20 IAS instances were available prior to execution. In this case, Task reallocation also occurs 10 times as the scheduler does not expect to receive such a quantity of resources. However, this time enough IAS instances are available to service one Task each; the reallocation occurred within the first 30 seconds of

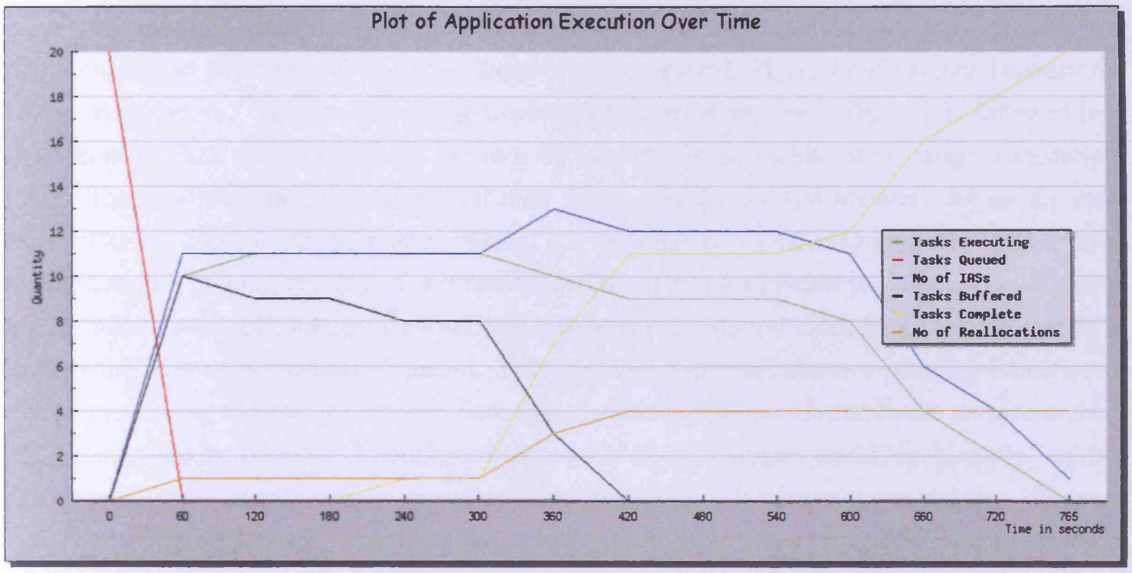


Figure 6.8: Plot of application execution when some resources were available when the application began.

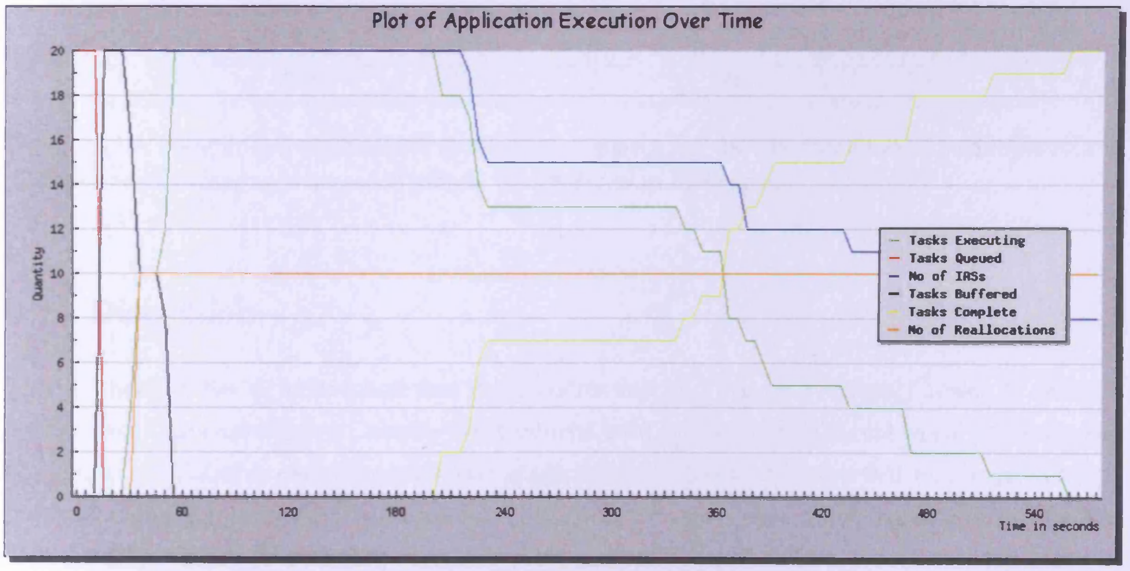


Figure 6.9: Plot of application execution when enough resources were available to cover all Tasks when the application began.

execution. Note that the data set was also transferred in this time. The effects of Task and resource heterogeneity are also most apparent in this scenario. Homogeneity at both levels would result in a more consistent and uniform Task completion rate. Instead, Figure 6.9 shows a large gap between 230 seconds and 340 seconds where no Tasks were completed. Here, the difference between the fastest workstations,[12] and the remaining workstations is most apparent. This gap is followed by a quick burst of Task completions over the next 60 seconds, as the middle of the range workstations finish. The slowest workstations finish off their Tasks over the last few minutes, and are the main cause of a less efficient utilization of resources. It is clear that here load balancing was not employed to duplicate the Tasks of the slowest workstations. The current load balancing strategy is not geared toward redundancy scheduling of a Task which has already started execution. Instead, it opts to balance the sizes of the daemons' queues. This scenario, therefore, shows a potential limitation of the load balancing strategy for smaller Task spaces. This is also manifested in the average number of IAS instances in use: 14. Therefore, better scheduling strategies could significantly improve performance in the third execution scenario, and potentially also in scenarios one and two. However, in the third scenario the user's wait time has now been reduced to 9 minutes and 50 seconds, a overall application makespan reduction of 89%, an improvement of 37% and 23% against scenarios one and two respectively.

Table 6.1 shows a juxtaposition of the completion times for the three scenarios in comparison to the benchmark application makespan for sequential execution.

Scenario	AM	Reduction in AM	Initial IAS Count	Avg No. of IASs
Benchmark	90:00	N/A	1	1
S1	15:35	83%	0	8
S2	12:45	86%	11	8
S3	09:50	89%	≥ 20	14

Table 6.1: A comparison of the three execution scenarios for the classification of schizophrenia in diffusion tensor images, where AM stands for application makespan.

6.9 Discussion

This application has demonstrated that the construction and use of a Virtual Cluster of Matlab-enabled workstations within a Campus-Grid environment can be applied to real world problems and not just to simulated or carefully contrived applications. It has also shown that real improvements in performance are achievable with minimal effort on the user's part. This began with the method for capturing Matlab functionality that was previously too complicated a process for the end user to perform. The use of the Autonomic (Matlab) Compiler (Section 5.2.1) not only facilitated this process and made it possible for the toolboxes PRTools and DipLib to be compiled, but did so in a general manner by capturing and compiling the entire toolbox. The end result is that all future

[12]Those most likely with a history of performing a similar Task as a result of a previous experimental run.

applications which need either of these toolboxes can be supported by the IAS without recompilation. In addition, the AC also highlighted areas of the toolboxes which contained errors. These were promptly relayed back to the developers for future improvements in the toolbox. In this scenario the user's function was also added to the IAS for convenience. However, this process is not essential to the use of an IAS, as any Matlab sequence that can be entered at the Matlab command line can be transferred to an IAS and executed.

The analysis performed in this Chapter and the three execution scenarios were used demonstrate two key aspects of the models discussed in Chapter 4 and implemented in Chapter 5; namely: that this approach does not compromise the accuracy of the algorithms in use, despite a different execution model, the use of the Matlab compiler and a different result representation. It also demonstrated that the Virtual Cluster can provide significant improvements in application makespan even when the workstations in use are of a much lower specification to the workstation used to benchmark the application for sequential execution. The reason for this difference was because improvements can only be measured relative to the user's normal execution scenario to accurately illustrate the benefit to the user of the approach documented in this thesis. In addition, significant improvements in performance were achieved despite the challenges of the Campus-Grid environment documented in Chapter 2. Note also that the best application makespan observed, using execution scenario three, enabled the entire analysis to be completed in less time than the recommended minimum job length for the Condor system.

The three execution scenarios also demonstrated some key aspects of the approach that are noteworthy. The first scenario, where no IAS instances were available when the Application Manager was started, the speed with which the system can be set up was demonstrated. Ordinarily, this time would be lost as scheduling overhead, but because each IAS instance can consume multiple Tasks, by scheduling in a just-in-time fashion, this time can be modelled as an investment or down payment on the resource itself. This "investment" will yield a reduction in the runtime of Tasks submitted to that IAS following this initialization period. An impression of the speed with which an IAS initializes was also demonstrated in this scenario; the first IAS instance that was ready for Task consumption and execution appeared after just 30 seconds.

In the second scenario, 11 IAS instances were available before the Application Manager was started. This demonstrated firstly, the utility of the ability to recycle previously initialized IAS instances, which was seen as a further reduction in application makespan. It also demonstrated the adaptive nature of the approach as a whole. The DTI analysis consists of 20 Tasks in total, but 11 workstations are available, which is not a factor of 20. The adaptive nature of the opportunistic scheduler capitalized upon the workstation heterogeneity by scheduling the fastest workstations, with more Tasks balancing load from the slower workstations. Essentially, when the fastest workstations finished and had no remaining work, Tasks were removed from the queues of other, slower, workstations to improve throughput. The adaptive nature of the system was also demonstrated in that during runtime the CRM (Section 5.3) managed to initialize a further three IAS instances. This occurred as a consequence of the global requirements model (Section 4.5), identifying that further

IAS instances could be consumed. Consequently, the CSM's (Section 5.4) demand upon the CRM was increased, with the effect that further IAS jobs were injected into the Campus-Grid.

In the third scenario, more than 20 IAS instances were available before the Application Manager began. This scenario demonstrated the variability of the Campus-Grid workstations. 20 IAS instances were available and used, but the total wall clock execution time did not drop to 4.5 minutes. Why? There are several answers to this question. Firstly, the workstation used to generate the performance benchmark was far more powerful than a large portion of the Campus-Grid workstations. Secondly, each Task is not homogeneous even on the same workstation, which is due to subtle differences in the way in which each Task acts upon the data set. Finally, IAS instances that have performed work in the past for other applications or previous runs of this analysis benefit from operating system caching of the libraries used by Matlab and thus can perform Tasks with a reduced runtime. This last reason further demonstrates the utility of being able to reuse IAS instances for subsequent and even disparate Tasks, as all Tasks share the same base Matlab functionality.

The ultimate goal of this analysis is to help a clinical researcher identify regions of the brain that are indicative of a neurodegenerative disease, enabling evidence-based decisions. By using the prototype documented in Chapter 5, the Matlab-based application was ported to a Campus-Grid infrastructure with minor modification. The application workload, consisting basically of independent Tasks, was transparently distributed among the Campus-Grid workstations. By using a general pre-compiled Matlab daemon, Matlab licenses are no longer required to run the application in parallel, which yields a saving of £1,500 per workstation in use. From the user's perspective this is very important since it enables the running of any Matlab application on similar Grid systems without the high cost of cluster licenses. In addition, the approach was generalized to facilitate any other combination of the tools used without requiring the user to recompile the Matlab-based toolboxes PRTools and DIPLib. The Campus-Grid implementation provided a reduction in application makespan from 90 minutes to less than 10 minutes, enabling the realization of complex and time consuming analysis within a reasonable time. The reduction in application makespan was obtained through: **(1)** the compilation of Matlab code, as licensed sessions cannot be supported, **(2)** parallel execution, and **(3)** an improved Task submission and management scheme.

Had this analysis been performed on the Condor system without the use of the Virtual Cluster it would not have been possible to complete this analysis as quickly. This claim can be substantiated with the following observations. Firstly, the average runtime for a Task was in the order of four minutes, where the minimum and maximum runtimes were three and seven minutes respectively. Secondly, the average initialization time for the IAS was approximately three minutes. Remember that in Section 4.4.2 it was identified that 95% of all runaway Matlab jobs could be identified using five standard deviations of the the average initialization time of a Matlab job, and therefore also of the IAS, which corresponds to approximately eleven minutes. In order to identify that a job had in fact runaway, a Job Manager like that defined in Section 2.4.2 would not be able to identify a runaway job before 18 minutes (11 for runaway classification plus maximum recorded Task runtime of 7 minutes). Thirdly, in Section 7.6 it will be demonstrated that a Campus-Grid job waits in the

Condor queue for nearly three minutes on average before being matched to a workstation, and a minimum of nine seconds. When these times are put together, they identify that the Campus-Grid would perform one Task in a minimum of 3:39 minutes. However, in Section 2.4 it was identified that 45% of all Matlab jobs fail and therefore nine Tasks would statistically need to be submitted at least twice before completing successfully. In reality it is more likely that between three and five rounds of submissions would be required in order to complete all twenty Tasks. This was determined by identifying the number of iterations, n, needed to make $Complete_n = Tasks$. To determine how many Tasks were completed in each iteration the following function was used:

$$Complete_n = Tasks - ROUND(FailureRate * (Tasks - Complete_{n-1})) \tag{6.1}$$

where

$$Complete_0 = 0 \tag{6.2}$$

Here, Tasks = 20 and FailureRate = 45%. This translates into a best Task completion time of between 3:39 and 4:23 minutes. However, they may need to be submitted up to five times,[13] therefore this is the best case scenario, where Tasks are submitted to the most powerful workstations in the Campus-Grid and do not fail, which is unlikely. Note also that the application makespan is defined by the last, i.e. slowest, Task to complete and not the first. A summary of the projected Task completion times for the best, average and worst case scenarios for Task runtime are presented in Table 6.2 with a variety of scheduling attempts required to complete a Task in each category. The best case is calculated using the shortest observed time required for: Task execution (3 minutes), IAS initialization (30 seconds), a job to be matched to a workstation by Condor (9 seconds) and the pilot test to complete (2 seconds).[14] The average case is calculated using the average values observed for these four aspects of a job: 4 minutes, 3 minutes, 177 seconds, and 22 seconds respectively. The worst case is calculated using the maximum observed time required for: Task execution (7 minutes), 5 standard deviations of the IAS initialization plus the average initialization time (11 minutes), the pilot test to complete (222 seconds). However, the time spent in the Condor queue is the average time to retain a more realistic spectrum of runtimes.

Table 6.2: Projected completion times of a single Task (in minutes) on the Campus-Grid using a traditional approach.

	Submission Attempts				
	1	2	3	4	5
Best Case	3:39	3:50	4:01	4:12	4:23
Average Case	9:57	13:14	16:30	19:48	23:05
Worst Case	20:57	27:36	34:15	40:54	54:12

[13]The number of Tasks completed in each iteration are: **(1)** 11, **(2)** 16, **(3)** 18, **(4)** 19, **(5)** 20

[14]See Section 7.6.

Using the projected runtimes in Table 6.2, it would require only one Task with average performance to fail once to prevent the application makespan being comparable to that demonstrated in Section 6.8. Also consider that failures are commonly experienced consecutively due to the emergence of job blackholes. Therefore, this is not an unlikely scenario to occur and would make it very difficult for a traditional use of Condor to outperform the Virtual Cluster. In addition, the effort required to create an autonomic job manager capable of managing the application would be a considerable effort.

6.10 Future Work: Spatial Registration of Data

Prior to a group-wise comparison of feature values, such as the Schizophrenia analysis, spatial registration (also known as normalization) of the data is required (step b in the workflow illustrated in Figure 6.2). The registration process is the transformation of three-dimensional data, so that anatomical variations can be accounted for. Without this process the heterogeneity inherent in anatomical structures makes comparative analysis impossible. Registration of this type is not only used in neurological analysis but in any comparative study on anatomical structure(s). There are two approaches for the registration of neurological data. It is dependent upon the analytical context which is adopted. The two approaches are: (1) The use of the FA-value. This is adopted when no hypothesis of the disease exists and therefore analysis is performed on the entire brain.[15] (2) Correspondence between specific fibre tracts is identified. Conventionally this process is performed by using a predefined set of features from the data set [221]. This is adopted when a specific region of interest exists.

The future work of this collaboration is a parallel solution for the second approach. However, there are many challenges involved in performing this process both seqentially on a local workstation and in parallel. In Section 6.10.1 a high-level overview of the analytical process will be presented, before a discussion of the challenges and the current approach are presented in Sections 6.10.3 and 6.10.2 respectively. Finally, in Section 6.10.4 the partial solution currently developed is presented along with a discussion that outlines why this solution could not be completed, and the new research required in order to create a working solution. The analytical content of this Section is a high-level summary of the following article: [221] and for further information the reader is directed to this article.

6.10.1 Registration Process

Registration is performed using an iterative pipeline, which is shown in Figure 6.10. The analytical scenario which is given as an example in this Section to illustrate the challenges and example computation times is adopted from a study into Amyotrophic Lateral Sclerosis (ALS).[16] It is hy-

[15]This was the case for the analysis of the Schizophrenia data.

[16]ALS is a progressive motor neuron disease, which is usually fatal. It affects the nerve cells in the central nervous system that control voluntary muscle movement, which over time degenerate and waste away.

pothesized that mainly the corticospinal tract (CST)[17] is affected by the disease, and therefore this is the target region of the brain for spatial alignment and matching. There are sixty subjects included in the study. Fibre tracking is performed by using, DTIStudio [222],[18] which uses regions of interest that when combined by logical operators, define the CST. This requires some new Matlab functionality to be incorporated into the IAS: namely DTIStudio, however, this is facilitated completely by the Autonomic (Matlab) Compiler (Section 5.2.1).

Essentially, the registration process brings the nerve fibres of the CST into spatial correspondence. Here a point set representation of the white matter fibre tracts is used for point set matching [223]. A combination of clustering and matching of the data is also employed to enable both representative and corresponding features to be obtained. This allows fibre tracts to be incorporated into the clustering process to enable an adequate definition. The result is a common frame of reference, denoted as an atlas, to which all data sets can be matched. To perform this three key steps are undertaken: **(1)** the points in each point set are clustered, **(2)** a transformation is estimated to warp the cluster centre points, and **(3)** an atlas is determined, (the average for each of the cluster centre points in all the data sets), which is a common frame of reference representing the mean shape of the point sets. These steps are then repeated in an iterative manner where the cluster size is reduced in a step wise manner. In other words, the algorithm is progressively moving from global matching to local matching, and therefore refining the matching process with each iteration, until the cluster size reaches a predefined minimum.

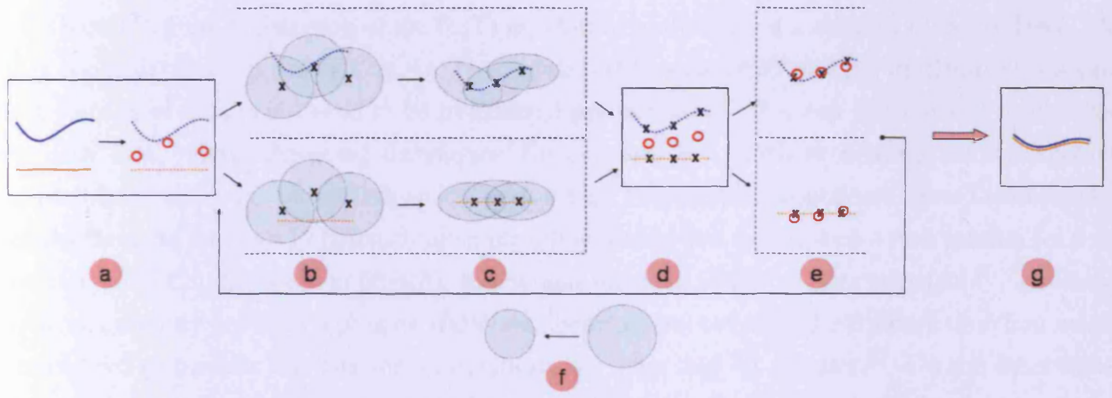


Figure 6.10: Pipeline of the registration process indicating the parallelizable steps. (a) Fibre tracts are sampled and atlas points are randomly initialised. (b-c) Each data set is clustered and cluster centre points are repositioned. (d) Atlas points are updated. (e) A (regularised) transformation is computed from the atlas point to the subsequent cluster centre point sets. (f) The cluster size is reduced and the next iteration begins. (g) When the minimum cluster size is reached, the individual data sets are warped to the atlas frame. Practically, steps *b*, *c* and *e* can be computed in parallel and *d* on a head node, which can be the user's Matlab session.

[17]The CST is a large bundle of nerve fibres located between the cerebral cortex and the top of the spinal cord. Its main function is the delivery of signals that enable voluntary movements throughout the body.

[18]Specifically the FACT algorithm, which enables a full brain tractography to be performed.

6.10.2 Current Approach and Motivation for a Parallel Solution

Computationally, this process is not demanding, but the memory requirements can quickly overwhelm the capacity of a standard workstation. In this particular case only 60 data sets are to be registered, but in future studies there could be a far greater number of data sets. Note also that this data set only contains one bundle of fibres. If other studies such as the Schizophrenia analysis were to be analyzed on a tract basis, multiple bundles would be involved. Consequently, the memory required to register the same number of data sets could increase dramatically. The effect this has on the researcher is that their algorithm cannot keep all data sets resident in memory simultaneously. Swapping data sets back and forth to disk would not be a solution, as it would significantly increase the execution time. Currently, this type of analysis is performed using a dedicated server with 8GB of RAM. However, this type of machine has a finite availability and capacity. The 60 data sets used in this analysis require approximately 5GBs of memory, making this approach infeasible if the data set was doubled in size. For this reason a parallel approach is desirable. In addition, because the computation time is so low batch processing is also not an option because the overhead involved in job submission is too excessive. Therefore, it was proposed to attempt this analysis using the Virtual Cluster due to the success of the previous study.

6.10.3 Challenges in Parallelizing Data Registration

Performing the registration process in parallel is not trivial due to the very short execution times (3 – 7 seconds for the registration of the CST) in relation to the size of a data set (25MB). Here, one Task is one iteration upon a data set, i.e. for 60 data sets there are 60 Tasks per iteration. This means that 1.46GB of data could need to be transferred per iteration,[19] if a new session was established for each Task. Here, clustering techniques, for example in a batch processing scenario, cannot be performed upon the entire Task space, because each iteration is dependent upon the combined results from the previous iteration. If an approach was used that established a new session for each job (a typical Campus-Grid approach), this would result in 75GB of data transfers.[20] The local network currently provides a peak of 100Mbps, therefore the theoretical minimum time that would be required to transfer the data for all iterations is 1 hour and 42 minutes.[21] On the other hand, because the IAS (Section 5.2) is used interactively, it means that a data set need to be transferred only once per workstation. With the incorporation of a new and data-aware scheduling routine it is possible to attempt to try and optimise data transfers by always sending Tasks relating to the same data set to the same IAS instance. This enables a best case data transfer rate of 1.46GB, which over a 100Mbps LAN would require 120 seconds.[22]

Unfortunately, this is an unlikely scenario, as few workstations will achieve peak performance and not all will share the same connection speeds. Some may, for example, have only a 10Mbps

[19] $\frac{60 \times 25MB}{1024} = 1.46GB$

[20] There are around 51 iterations in this registration process, giving rise data of: $51 \times 60 \times 25MB = 74.7GB$.

[21] $\frac{74.7 \times 1024 \times 8}{100} = 6119.424\ seconds = 1 : 41 : 59.424$

[22] $\frac{60 \times 25MB \times 8}{100} = 120\ seconds.$

connection. The head node is also likely to become a performance bottleneck. The variability in connection speeds presents an additional challenge, as the system allocates work based upon the perceived speed of a workstation. However, for the first iteration the fastest workstations will be those with the fastest network connections, and not the fastest CPUs. This means that in the second iteration the fastest CPUs will compute their Tasks much quicker and therefore run out of work. This can potentially result in a large load imbalance, which will have significant ramifications upon performance as the analysis progresses. Identifying a workstation's connection speed is also difficult to determine without prior interaction, and this data may also not be reliable over time. Some simple performance testing of the registration algorithm has identified that a Task will require on average between 3[23] and 27[24] seconds to execute, depending upon a workstation's LAN and CPU speeds as well as its previous data transfers. The registration analysis consists of 3060 Tasks[25] so any discrepancies in load will cause long tail scheduling scenarios at the end of each iteration, which will drastically detriment the application makespan.

One further challenge remains: for this pipeline to run it needs to be controlled from a Matlab instance to enable steps d and f in Figure 6.10. This was addressed in Section 6.6.1 through the development of a Matlab-based Application Manager. However, it also means that the input parameters and output data for each Task must be transferred to an IAS. There is approximately 1.3MB of unique input and output data corresponding to 7.8GB of data,[26] which further motivates the requirement for minimizing the transfer of the CST data.

An application of this type would normally need dedicated and stable resources in order to retain throughput, which is demonstrated in the current approach (Section 6.10.2). A Campus-Grid cannot guarantee that either of these attributes are withheld, making an already difficult problem even more challenging. Yet, it can provide the computational power and (distributed) memory required. In addition, it may also be able to provide a much greater capacity than a dedicated machine. Here, the challenge is not having enough resources for the execution, but harnessing them with adequate accuracy and performance.

A parallel solution to this process is not only beneficial in other neurological studies, but also within other areas of anatomical study and image registration in general. Similarly, the reliance on Matlab at the head of the computational steering is relevant to many other areas of scientific analysis with similar requirements. Therefore, the benefit of a solution, even a partial one, is clear.

6.10.4 Core Components for a New Approach

There are two key components that determine the success of an approach to facilitate a registration pipeline: **(1)** Matlab integration, which was presented in Section 6.6.1, and **(2)** an adequate scheduling strategy. Both of these areas have their challenges, which were discussed above, but they are not

[23]The minimum observed execution time of a Task without data transfer.

[24]The maximum observed execution time of 7 seconds plus the theoretical minimum time to transfer a data set over a 10Mbps LAN connection.

[25]60 $datasets \times 51\ iterations = 3060$.

[26] $\frac{3060 \times 1.3 \times 2}{1024} = 7.77GB$

mutually exclusive and must both work together to create a solution. The integration into Matlab needs to be unobtrusive and familiar to the user, but also reliable and intricately balanced to ensure a smooth integration between Matlab and the underlying software solution. The scheduling strategy must be considerate of the data requirements of the domain and consider the possibility that workstations are heterogeneous in both performance and network connectivity. In addition, they are also dynamic with respect to their availability.

This specific scheduling problem has many similarities to challenges which have been researched in the past, most notably in approaches for data Grids [224]. Here, the general challenge is to create scheduling strategies to include resources that are both computationally capable but also do not generate high transfer costs. More specifically, it is the consideration of a variety of additional metrics and constraints that make developing such scheduling strategies challenging [225, 226]. Therefore, a solution that separates data and Task scheduling is likely to be inefficient [225,226]. A common approach is to explicitly address both data and Task requirements in the scheduling strategy, which is the case in: [119, 225–228].

Several data-aware scheduling strategies are presented in [225] and [226], along with a discussion and identification of two distinct strategies, which in theory should work very well. However, when interpreting the presented results, it is imperative to consider that their results are based upon simulations. The results of this work was criticized by Park and Kim in [227] because one of their primary observations was that network bandwidth is the ultimate factor which will determine firstly, which scheduling strategy should be used and secondly, how effective it will be. Simulated environments therefore, cannot fully capture the nuances of real deployment. Park and Kim [227] undertook similar analysis but within a real context, however, their resource scenario is significantly more reliable than the conditions of a Campus-Grid. The forerunner of this domain is probably the AppLeS scheduler [119] (an extension of Nimrod) for parameter sweeps within a computational Grid environment. AppLeS focuses on data reuse in order to minimize the cost associated to the (re)transfer of data. In [228] a data-aware scheduler that was integrated in the Gridbus [97] broker is described. Here, as in many other approaches in the literature and those mentioned above, it is the identification of resource information prior to scheduling that is paramount to the accuracy, cost and performance of the scheduling strategy. To facilitate this, Grid Information Services (GIS) [177] or some monitoring system such as NWS [178], Ganglia [123], Hawkeye [70] or GridRM [179] is used to provide all the necessary data to aid the resource allocation process within the scheduling strategy.

Many approaches simply assume that such a service is available, maintained and accurate. AppLeS, for example, assumes that full knowledge of the system, i.e. computational loads, network conditions and topologies are available to the scheduler. In the context of a Campus-Grid this is simply not the case: consider the results of Section 2.3 for the accuracy of Campus-Grid metadata. Therefore, scheduling strategies in a Campus-Grid environment have to overcome a significant hurdle. For example, a Campus-Grid managed by Condor can be integrated with many different monitoring systems which could provide this information. Monitoring information can be directly encapsulated into a workstation's ClassAd, which can then be used by the Condor matchmaking processes to select resources based upon more detailed information than the general Condor ClassAd provides. NWS, Ganglia and GridRM are the primary examples here, as to some extent is Hawkeye. The experiences gained through the work for this thesis gives rise to the opinion that the necessary information being available is a utopian and unrealistic expectation in the general case of Campus-Grids. This is not to say that no Campus-Grids will have such a service, they should, as this would improve many aspects of their performance. It is more a question of whether the administration will have the necessary foresight to introduce and manage such a service, and then enrich the depth of the elicitation of technical preferences to enable its use. The Cardiff University Campus-Grid does not enable such a depth of technical preferences, and it is unlikely that it will do so in the near future.

Scheduling is paramount for a successful approach to parallelizing the registration of anatomical data. Essentially, there is a trade off between the number of workstations that can act upon a data set and the time required for these workstations to receive a data set. Ideally, more than one workstation should be able to act upon a given data set, as this would make load balancing much easier and reduce the effect of workstation revocation. However, replicating the data set unnecessarily is a waste of bandwidth. Finding a solution to this trade off is not too taxing when resources are static, dependable and when the necessary information is available to accurately cherry pick resources based upon a set of constraints. In a Campus-Grid context none of these states are guaranteed as explained in Section 1.3.1, Chapter 2 and above. This means that when a change in workstation availability occurs, not only must the balancing of workload be performed, but also of data allocation. The latter is more detrimental to performance. In addition, new workstations may not meet the requirements needed, such as good network performance, as these preferences cannot be considered by the Condor matchmaker. Therefore, when a new workstation becomes available, its utility may not become apparent immediately due to the penalty of data reallocation. The use of an IAS's local queue appeases any data penalty somewhat, as multiple transfers can run concurrently. IAS instances could also be instructed by the scheduler to acquire replicas of data as background processes when they have no other Tasks to perform. In addition, an opportunistic scheduling scenario could be derived to accommodate these challenges.

However, there is one core question that arises due to the inability to instruct Condor to select a workstation based upon the requirement for network connectivity: can a workstation that is computational inferior, but has a 100Mbps Ethernet connection be more beneficial than a computationally superior workstation with 10Mpbs Ethernet connection? Unfortunately, but perhaps not surprisingly, the bandwidth is by far the dominating factor, which can be validated by the observations in [225–227]. Figure 6.11 depicts this comparison graphically, using two projected and estimated scenarios. Here, an optimal scheduling strategy with all workstations having a 10Mbps Ethernet connection is projected against a worst case random allocation scenario, where all workstations have a 100Mbps Ethernet connection. To make this comparison more interesting and competitive, the optimal scenario has also been enhanced by using the minimum observed Task runtimes, i.e. in other words, all workstations are very high specification with respect to computational performance. Therefore, this scenario is best case for computational resources and optimal in terms of scheduling. In contrast, the random scheduling scenario has been coupled with the longest observed Task completion time, i.e. the slowest workstations. In addition, it has been tweaked to ensure that data is transferred as much as possible, i.e. each data set is transferred N times, where N is the number of IAS instances.

It can be clearly seen that a Virtual Cluster formed from workstations with higher network bandwidth and lower processing power, even with a worst case data scheduling algorithm possible, will always yield a more timely execution, even though it can clearly be seen that as N increases so too does execution time.[27] This means that, in terms of developing a solution for a registration pipeline, there is little purpose in pursuing an optimal or even good scheduling strategy. At least until there is the ability to either: **(1)** enable Condor to consider this requirement, or **(2)** introduce a new component into the system that can performance test network connectivity, and produce the necessary meta-data for consideration by a scheduler or to enable a whitelist of preferential workstations for use by the CRM (Section 5.3). It is for this reason that this work remains a future persuit, as following this route would be straying from the core research questions posed in Section 1.5. The Virtual Cluster has been used to perform the analysis on a reduced scale using the scheduler presented in Section 5.5.3 with promising results. However, the level of variability with respect to networking performance does not enable an adequate solution when all data sets are included without either of the necessary aforementioned extensions. Therefore, the lesson that has been learnt from this study is that not only are the sources of resource meta-data typically inaccurate (Section 2.4), but they are also inadequate with respect to the range of resource metrics that they consider. For this reason the full application scope that could be utilized by a Campus-Grid is not yet fully exploited.

[27]This occurs because the time to transfer data begins to outweigh the time needed to process it, and also because the head node reaches peak performance and its network connection becomes saturated.

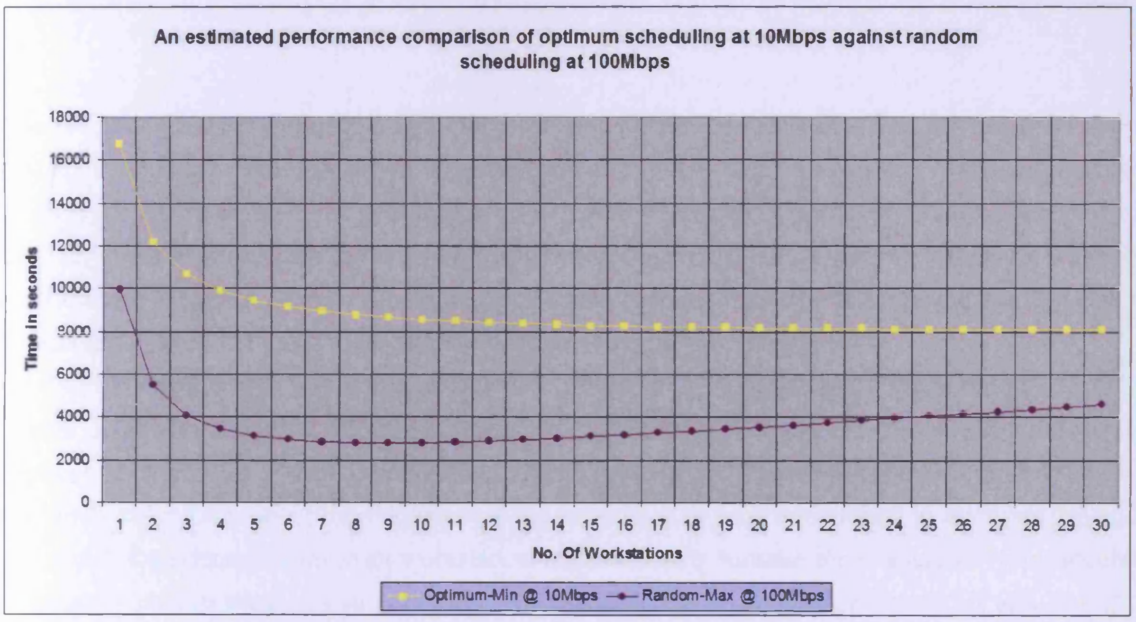


Figure 6.11: A comparison of the minimum predicted execution with optimum scheduling at 10Mbps against the maximum predicted execution with random scheduling at 100Mbps. The minimum estimations are calculated using the minimum daemon setup time and Task time recorded and an optimal data-aware scheduler (no data set is transferred more than once). The maximum estimations are calculated using the highest permitted daemon start time of 10 minutes (see Section 5.3), the longest recorded Task time and the worst case for random scheduling (every data set is transferred to every workstation).

Chapter 7

Evaluation

The previous Chapter demonstrated the application of the research undertaken for this thesis in a real application scenario. In Chapter 2 the performance of the Campus-Grid was evaluated in order to motivate the need for more research. In this Chapter a rerun of the experiments presented in Chapter 2 will be performed using the Virtual Cluster. This will enable a direct comparison with the performance of Campus-Grid as quantified in Chapter 2 and other approaches presented in Chapter 3 to validate this research and illustrate where improvements have been made.

The structure of this Chapter is as follows: In Section 7.1 the methodology behind the experiments documented in this Chapter is presented. In Sections 7.2 and 7.3, the same synthesized applications that were performed in Section 2.4 are presented using two different execution scenarios. In Section 7.4 an overview of some of the experimental runs undertaken in Sections 7.2 and 7.3, which experienced significant workstation failures during runtime are discussed. Here, special attention is paid to scenarios such as the CRM (Section 5.3) loosing all workstations and how the system dealt with such scenarios, i.e. the system's fault tolerance. In Section 7.5 the experiment undertaken in Section 2.2 is repeated in a Matlab context to quantify and compare the overhead involved in the use of the Virtual Cluster. In Section 7.6 the data held in the Condor logs that were generated by IAS jobs submitted through the CRM is analyzed to illustrate what the system was doing behind the scenes when the experiments run in this Chapter were performed. Finally, in Section 7.7 the results are summarised and discussed.

7.1 Evaluation Methodology

There are three primary incentives behind the experiments conducted in this Chapter. Firstly, to illustrate that performance has been improved upon through the introduction of the models described in Chapter 4. Secondly, that Tasks of even the shortest runtimes can feasibly be executed in an opportunistic and volatile environment. Finally, they are to provide a foundation upon which answers to the research questions posed in Section 1.5 can be built. This discussion will be addressed in Chapter 8.

There are three topics discussed in this Section, which are: how each of the experiments covered in this Chapter fulfil the above incentives, what data is captured in order to illustrate the results of an experiment, and the way in which the experiments are orchestrated. These three topics are discussed in Sections 7.1.1 – 7.1.3 respectively.

7.1.1 Philosophy of Experimentation

The experiments undertaken in this Chapter are based around those performed in Chapter 2. The reason for this is that it enables a direct comparison with the performance of the Campus-Grid middleware (Condor) to the prototype documented in Chapter 5 by using the same Task definitions and (synthesized) applications. The motivation for the use of synthesized applications and the definition of what constitutes a synthesized application was given in Section 2.4.1. However, the two main points to remember are: firstly, that they create a fairer platform for experimentation, as they are not affected by the prolific resource heterogeneity of a Campus-Grid infrastructure, and secondly, that the Task runtime can be explicitly and stringently controlled to enable the performance of a system to be studied for a variety of different Task runtimes.

Sections 7.2 and 7.3 present a rerun of the analysis conducted in Section 2.4.3. However, instead of Condor scheduling the Tasks of the synthesized application, this time Condor will be used to build a Virtual Cluster of Matlab-enabled nodes (IAS instances see: Section 5.2), which are steered interactively at runtime. Here, each workstation is initialized only once by the CRM (see: Section 5.3) in response to the demand modelled by the CI (see: Section 5.5) and established by the CSM (see: Section 5.4). Each IAS performs Tasks in a back-to-back manner where the CI submits each Task directly to the IAS in a custom fashion and not through the normal Condor means, as described in Section 5.5.2. The end result is that the initialization penalty, which was high in the traditional Campus-Grid approach (Section 2.4.3) is experienced only once per workstation and not once per Task. In addition, the challenges of managing resource volatility are answered by the CRM and therefore do not need to be considered in the scheduler's plan.

The difference between this analysis and that conducted in Section 2.4.3 is that in Sections 7.2 and 7.3 the focus is on Tasks with short runtimes rather than a larger range of runtimes. This is because the main application focus of this research is the ability to provision resources specifically for short Tasks. In Section 2.4.3 longer Task runtimes were included to validate the experimental methodology. The synthesized applications executed in Sections 7.2 and 7.3 are: 100 Tasks with a predefined runtime of: 1, 5, 10, 15, 30, 45 and 60 seconds. An upper bound of sixty seconds was chosen because once Task runtime goes beyond this point only negligible improvements in performance are achievable. This will be discussed in further detail and demonstrated in Section 7.3. The results of these experiments will be discussed using the same performance metrics as those presented in Section 2.4.1; namely, Job Efficiency (now denoted as Task Efficiency: TE), Job Runtime (now denoted as Task Runtime: TR), Application Makespan (AM) and the Efficiency of Resource Utilization (ERU). The means to determine the values of each metric also remains the same.

The single difference between the experimental analysis presented in Sections 7.2 and 7.3 is that in Section 7.2 the Virtual Cluster is restarted manually after each experimental run. Consequentally, the aims of this experiment are to demonstrate several improvements in performance over the traditional Campus-Grid approaches, which include: speed of set up, reduced application make span, Task runtimes nearer to their expected runtimes, improved efficiency of resource utilization, and transparency with respect to the management of resource volatility issues, which were experienced in Chapter 2. In this case, all performance values include the time required to initialize and build the Virtual Cluster, thereby capturing all the potential aspects that have a negative impact on performance, such as: Campus-Grid queue times, no available workstations etc.

In contrast, the experimental analysis presented in Section 7.3 differs in that the Virtual Cluster will not be shutdown between experimental runs, but instead be recycled.[1] This is a direct consequence of the Autonomic Compiler (Section 5.2.1) and its ability to compile entire Matlab toolboxes into a single stand-alone application. In addition, the produced code can be used, steered and controlled interactively through the IAS daemon. Therefore, this will demonstrate the further improvements in performance which can be achieved due to the heterogeneity in application scope of the IAS daemon. The results presented in this Section demonstrate the further improvements in performance that can be realized even for the shortest of Tasks of 1 second in duration, by not shutting the system down between experimental runs.

The approach of not shutting down the Virtual Cluster is much more feasible in a real context. From a theoretical perspective it is perfectly acceptable behaviour to restart the Virtual Cluster after every experimental run. However, it is not a behaviour that would be employed in practice, as it is wasteful of both time and resources. For this reason, Section 7.3 aims to quantify the benefits of not shutting down the Virtual Cluster between experiments. In practice this would occur between runs of an application, analysis of different data sets, or after the tuning of an algorithm or research hypothesis. Here, the time needed to acquire and initialize the Campus-Grid workstations that constitute the Virtual Cluster is not experienced by the user, as it is in Section 7.2. This is achieved by using a Virtual Cluster that was initialized at some point in the past. The benefit of this action is that the potentially time consuming initialization period can be avoided and execution can begin immediately, even if only one workstation is available. In this context the CRM still acts as it would normally, the only difference is that it will attempt to increase the number of IAS instances already available rather than attempt to build the Virtual Cluster from scratch.

In Section 7.5 the experiment undertaken in Section 2.2 is repeated, but in a Matlab context. This is a study of the impact that queue length and communications overhead has on the performance of the system. The key difference to the analysis undertaken in Section 2.2 is that instead of running the Task directly on the workstation and then allowing Condor to clean up afterwards, the Task is executed by an IAS instance. Here, the overheads experienced by the Task relates to the costs of scheduling and transferring the Task to the IAS, invoking the Matlab engine, generating a result and transferring the results back. The Task is `pause(0)`, which pauses the Matlab engine for zero seconds. This is comparable to the Dummy Job used in Section 2.2 (the issuing of the `exit` command), and will be referred to as a Dummy Task. By performing such a Task the overall runtime

[1]Here, recycling relates to the reuse of a Virtual Cluster that has been created at some point in the past.

of the Task is:

$$k + R \tag{7.1}$$

Where k is the overhead of submitting a Task and R the predetermined runtime. By making R as close to zero as possible it is possible to measure k more accurately. k is the time required to schedule a Task and also for the IAS to execute it, gather any results and return the result. The generated results for the Dummy Task are: the string "[no results]" and two 256×256 binary images of a few kilobytes in size. The aim of this experiment is to demonstrate firstly, that the number of Tasks in the scheduler's queue is no longer detrimental to the achieved performance, as observed in Section 2.2, and secondly, that the communication implementation is sufficiently light weight and unobtrusive that the overhead in executing a Task is negligible. It will also demonstrate that load balancing techniques could be employed with out high communication costs, when the Task message is retransferred.

The third study undertaken in Section 2.3 is not explicitly repeated here, as the solutions to these challenges are transparently handled by the CRM and therefore inherent in the results of Sections 7.2, 7.3 and 7.5. These results and how they answer the challenges of supporting compiled Matlab jobs on the Campus-Grid are discussed in Section 7.6.

7.1.2 Capture of Experimental Data

All performance data captured in these experiments is provided by the introspective monitoring of the CI (Section 5.5) across its three layers (Task, Resource and Communication). The data which is captured is done so in order to determine the values of the four performance metrics, as defined in Section 2.4.1. In addition, in order to visualize the performance of the system during its execution of a synthesized application, several performance indicators are captured at one second intervals to provide a snapshot of current progress. The snapshots are then ordered chronologically to visualize the experimental run.

The data captured for the visualization process is:

1. The current number of resources (IAS instances)[1] that contribute towards the Virtual Cluster, which will show how the provisioning of resources changes over time, but also the rate at which resources are gained and lost over time. In addition, this will illustrate how the scheduler releases unnecessary resources as the number of unscheduled Tasks approaches zero.

2. The number of Tasks complete, which will show the progress of Task completion over time so that this can be related to the other performance indicators mentioned here.

3. The number of Tasks in remote IAS queues, which will show how the scheduler utilizes the IAS queues and dynamically modifies the queue size over time with respect to the number of IAS instances and Tasks remaining.

4. The number of Tasks rescheduled as a consequence of workstation autonomy,[2] which will outline the number of failures due to workstation autonomy.

5. The number of Tasks that have been rescheduled due to a balance in work load,[2] which will demonstrate the opportunistic nature of the scheduler when changes to its scheduling plan become necessary. This can happen through the receipt of new resources or the identification of workstations by the CRM that have been suspended.

6. The average completion time of all Tasks,[3] which will give an indication of consistency with respect to Task execution times.

This data will be used to visualize the system's performance and handling of the Task space as well as the autonomy of the Campus-Grid. This data is inserted into a MySQL database for persistent storage. In order to minimise the cost of data entry into the database, the MySQL server is located on the same physical machine as the testing Application Manager (described in Section 7.1.3). Once an experimental run has completed some general performance data is also collected and submitted to the database to provide the summary values needed for the calculation and analysis of the performance metrics. These are as follows:

1. The total number of Tasks submitted by the scheduler, which is needed for the calculation of the TE performance metric.

2. The number of Tasks which were rescheduled due to autonomy failures and balancing actions, again needed for the TE performance metric.

3. The TE value for the experiment is then calculated using the above data and entered into the database.

4. The minimum, average and maximum execution times for a Task, which is needed for the TR metric.

5. The AM of the experiment, which is needed for the AM and ERU metrics.

6. The average and maximum number of IAS instances used in the experiment, as the former is needed for the ERU metric. The latter is also collected for discussion purposes later in the Sections 7.2.3.

The data gathering process is not greatly obtrusive on the system's performance due to the multithreaded nature of the CI. The gathering process is a thread which accesses the above values and inserts them into the database in the background. It is rare for an error to occur in the data gathering process which has either a detrimental or beneficial effect on the performance data collected. In cases where an error is detected the results are thrown away. Should an error occur, it is most likely to be because the Task runtime was very short and this is specifically the case for the `pause(0)` Tasks, which constitute the experiments conducted in Section 7.5. If a more serious error occurs, the system is intentionally halted automatically by the Application Manager (Section 7.1.3), the database cleaned and the experimental run restarted.

[1]This contributes toward the calculation of the average number of IAS instances used, which is needed for the ERU performance metric.

[2]The differentiation between Tasks resubmitted due to autonomy failures and balancing actions is needed for the TE performance metric.

[3]The average Task completion time is needed for the TR performance metric.

7.1.3 Experiment Orchestration

The experiments performed in this Chapter were performed using a specially designed and built Application Manager (Section 5.5.1). The reasoning for this was threefold. Firstly, running these experiments multiple times is tedious and therefore the probability of human error increases. The definition and use of an Application Manager means that all experiments are treated in a uniform manner and decreases the likelihood of human error. One example, is ensuring that a minimum of five minutes pass between each experimental run. A period of five minutes was chosen, as this is the time required by Condor to perform a negotiation cycle and place any released Campus-Grid workstations back into the list of available workstations. Secondly, the Application Manager is used to create the necessary tables in the database for each experiment and also access the necessary data from the CI and insert it into the database. It is also introspective of the results it collects and is used to validate the results with respect to the potential errors that can occur, as mentioned in Section 7.1.2. Finally, this Application Manager is used to illustrate the capabilities that can be injected into an Application Manager by a user.

The administrative tasks that the Application Manager performs are (in this order): **(1)** probe the database to identify which experiment is to be run next, based upon the provided Task runtimes and number of repetitions for each Task runtime, **(2)** create a table in the database for the snapshot entries, using a supplied naming standard, **(3)** create 100 Tasks of the current predefined runtime, **(4)** invoke the CI for the discovery of and connection to the CSM, **(5)** start and perform the snapshot monitoring, **(6)** pass the Tasks to the scheduler, **(7)** wait for all 100 Tasks to complete, **(8)** calculate and insert the summary values into an "overview" table in the database, **(9)** shutdown the CI, and disconnect from the system as a whole, **(10)** alert the user that the experiment has finished, so that any system administration can be performed, for example in Section 7.2 all components need to be restarted, **(11)** wait for five minutes, **(12)** display a message box to allow the user to indicate that the system is ready for the next experiment to start, and finally **(13)** start the next experiment, until all experiments have been performed.

All of the experiments presented in this Chapter were run during "*normal*" working hours, i.e. between 8am and 6pm, on weekdays during the teaching semester. This relates to the time when resource autonomy is highest, but not necessarily when resource contention is highest, as many Condor users submit their jobs to run overnight. Ironically, this is when the fewest resources are available. Every combination of the experiments (of which there are 20) performed in this Chapter have been executed at least five times. However, many have been run ten times, to enable variation with respect to the workstations in use and the context in which they are used. In total over 31,000 Tasks were executed by the Virtual Cluster, consuming approximately 65 CPU hours.

7.1.4 Resources Used in the Evaluation

To ensure as fair a comparison as possible, the CRM (Section 5.3) resides on the same submit node that was used in Chapter 2, which is an Intel Pentium 4, 3GHz Dual Core, with 1GB of RAM and a 100Mbit Ethernet connection on the Condor subdomain of the Cardiff University network. The OS is the "Standard Image" Windows XP SP2 used for all workstations managed by the Novell network management software. The LS (Section 5.6) and CSM (Section 5.4) are co-located on the same physical machine for convenience, which is a 2.6GHz Intel Pentium 4 with 512MB RAM, and a 100Mbit Ethernet connection on the School of Computer Science subdomain of the Cardiff University network and the OS is Windows XP Home SP2. The Application Manager and MySQL server are run on an Intel Centrino Duo 1.83GHz, with 1GB of RAM and a 100Mbit Ethernet connection on the School of Computer Science subdomain of the Cardiff University network, the OS is Windows XP Pro SP3. The IAS instance used in the evaluation is the same as the one used in Chapters 2 and 6 to remain as fair as possible with respect to the produced performance data, but also to be representative of a real application scenario with respect to IAS initialization times. It is 7MB in size when compressed, and 20MB after it has been uncompressed by the Matlab Component Runtime.

7.2 Building and Using a Virtual Cluster

In this Section the experiments performed are the execution of synthesized applications comprising of 100 identical Tasks of 1, 5, 10, 15, 30, 45 and 60 seconds in length. The execution scenario used here is to build a Virtual Cluster of Matlab-enabled nodes (IAS instances see: Section 5.2) as an intrinsic part of the execution of the synthesized application. Therefore after each experimental run all system components: i.e. the CRM (Section 5.3), CSM (Section 5.4), CI (Section 5.5), and LS (Section 5.6) are shutdown and restarted, and all IAS jobs are terminated manually. As previously mentioned in Section 7.1, a period of at least 5 minutes has elapsed before any system component is restarted. They are started in the following order:

1. LS, as it is needed by all other components to locate the CSM and by the CSM to register its location.

2. CSM, to then register with the LS

3. CI (started by the Application Manager, see Section 7.1.3) and CRM are started at the same time, to ensure that IAS jobs are not submitted to the Campus-Grid before the Application Manager commences.

4. IAS instances are submitted by the CRM in response to the demand for resources provided by the CSM.

Table 7.1 shows the performance of the Virtual Cluster for this part of the evaluation and the four performance metrics are discussed below using the results illustrated in Table 7.1.

7.2.1 TE Performance

The TE performance is largely improved and mostly around 100% for the simple reason that autonomy related failures (i.e. the revocation of a workstation's voluntary status) are not included

Table 7.1: Table of the performance of 100 synthesized Tasks using the approach documented in this thesis. Here IAS jobs were started by the CRM after the client joined the system. Key: AM1 - AM for one workstation when run locally, VF - Volatility Failures, AF - Autonomy Failures, BA - Balancing Actions.

	Job Time in Seconds						
	1	5	10	15	30	45	60
Min Tasks	100	100	100	100	100	100	100
Avg Tasks	100.6	100.4	100.4	100.4	107.2	100.6	110.6
Max Tasks	102	101	102	101	127	102	134
Min Tasks	100	100	100	100	100	100	100
Avg Tasks	100.6	100.4	100.4	100.4	107.2	100.6	110.6
Max Tasks	102	101	102	101	127	102	134
Min TE	98.04%	99.01%	99.01%	99.01%	98.04%	100%	99.01%
Avg TE	99.41%	99.6%	99.8%	99.6%	99.41%	100%	99.9%
Max TE	100%	100%	100%	100%	100%	100%	100%
VF	0%	0%	0%	0%	0%	0%	0%
AF	0%	0%	50%	0%	91.67%	100%	99.06%
BA	100%	100%	50%	100%	8.33%	0%	0.94%
Min TR	1.02	5.01	10.02	15.03	30.04	45.03	60.1
Avg TR	1.02	5.03	10.03	15.04	30.1	45.09	60.48
Max TR	1.03	5.04	10.04	15.05	30.26	45.23	60.99
TR STD	0.01	0.01	0.01	0.01	0.08	0.07	0.33
Min AM	132	193	210	256	359	506	454
Avg AM	170.2	224.8	261.2	295.6	483.8	552	996.5
Max AM	240	260	349	345	892	595	2043
AM1	100	500	1000	1500	3000	4500	6000
Min ERU	7.01 %	34.95 %	43.66 %	49.82 %	28.43 %	68.01 %	33.17 %
Avg ERU	13.8 %	39.9 %	50.77 %	54.47 %	59.96 %	75.05 %	74.68 %
Max ERU	18.6 %	47.37 %	56.31 %	58.26 %	71.34 %	80.94 %	91.51 %

in the calculation, as they are beyond the control of the system and inherent in the nature of opportunistic computing. The same consideration was made in Section 2.4. In addition, volatility related failures are not experienced by the scheduler, but absorbed and managed by the CRM. What this ultimately relates to will be discussed in Section 7.6. The removal of volatility related failures means that the only time that a scheduler needs to reschedule a Task is when the workstation's period of availability ends (an autonomy related failure). Note that additional Tasks may be submitted by the scheduler in an attempt to reduce the overall AM, through the balancing of IAS queues or balancing actions. However, balancing actions have a much reduced impact upon performance in comparison to autonomy failures. The reason is that a balancing action is performed with the intention to reduce the AM, whereas, an autonomy failure requires the Task to be resubmitted with the consequence that any previous computing time or data transferred is lost. The main factor in the results of this performance metric is the improvement due to the removal of volatility related failures (Section 1.3.1), for example inaccurate resource meta-data. For this reason the improvement from 60–70% in Section 2.4 to averages in the high 90%'s in Table 7.1 is experienced, which is a considerable improvement. Table 7.2 juxtaposes the values of this performance metric as delivered by the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, using the execution scenario presented in this Section. It mainly illustrates the benefit to the scheduler in that Tasks are only rescheduled when a balancing action is required, or when the volunteer status of a workstation is revoked. This is because none of the volatility-related issues that hampered performance in Section 2.4, are experienced by the scheduler, as they are handled transparently by the CRM. Consequently, the Virtual Cluster executes Tasks much more efficiently.

Table 7.2: Comparison of the TE performance between the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, for 1 and 60 second Task Runtimes.

Performance Metric	Task Runtime			
	1 second		60 seconds	
	Condor	Virtual Cluster	Condor	Virtual Cluster
Minimum TE	63.694%	98.039%	53.763%	99.010%
Average TE	75.898%	99.410%	76.500%	99.901%
Maximum TE	83.333%	100%	100%	100%

7.2.2 TR Performance

The TR of all Tasks is much nearer to the predefined times and much more consistent. Consistency is signified by the low standard deviation, which is calculated using all Task runtimes in all repetitions for experimental runs relating to the given Task runtime. This is due to the performance no longer being linked to any workstation related heterogeneity, which is experienced through variations in the time required for the initialization of the IAS. This occurs because the workstation is initialized and its capability validated before a Task is submitted to the workstation, which on average is in the order of 140 seconds; this will be discussed in Section 7.6. Therefore, when the Task arrives at the workstation it could be executed immediately. The overheads (Task generation, submission, execution, result generation and result transmission) for this approach are also very low; never more than one second and typically a matter of milliseconds (this will be shown in Section 7.5). This again is due to direct point to point communication, using small messages and the ability to queue Tasks at the remote workstation, which further decreases the Task runtime.

Table 7.3 juxtaposes the values of this performance metric as delivered by the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, using the execution scenario presented in this Section. Here it can be seen that improvement is significant, even in the minimum observed TR values. In all cases, this relates to a reduction in the TR value of between 99.99% and 99.9999%. A reduction of 100% would occur if no overhead was experienced.

Table 7.3: Comparison of the TR performance between the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, for 1 and 60 second Task Runtimes.

Performance Metric	Task Runtime			
	1 second		60 seconds	
	Condor	Virtual Cluster	Condor	Virtual Cluster
Minimum TR	32	1.02	94	60.1
Average TR	100.9	1.02	154.1	60.48
Maximum TR	574	1.03	946	60.99
TR Standard Deviation	67.3	0.01	57.3	0.33

7.2.3 AM Performance

The AM performance is largely improved, but for the one second Tasks it is still not good enough to merit parallel execution, as the required time is greater than that of local and sequential execution. This is largely due to the time required to build a Virtual Cluster before Tasks can begin execution. Tasks of 5 seconds or greater in length, however, do experience a significant improvement in the AM to demonstrate that this approach can provide a good reduction in the AM for a set of Tasks of as little as 5 seconds in length. Compared to the AM performance presented in Section 2.4, where little difference was experienced between the AM performance of 1 second and 60 second Tasks, the results presented in Table 7.1 demonstrate that a significant improvement has been achieved. The AM for 60 second Tasks has also noticeably improved. However, in order to fully quantify this improvement, the number of workstations in use must also be considered. In Section 2.4 the best AM for a set of 60 second Tasks was 946 seconds, of all the experiments run in this Section, the nearest AM was 973 seconds. To execute this Task space, Condor used 16.4 workstations on average and a maximum of 31. The Virtual Cluster, on the other hand, was comprised of an average of 6.7 IAS instances and a maximum of 8, which quite clearly shows that despite the AMs being similar, the way in which these two experimental runs were performed is quite different. This is summarized in Table 7.4.

This is also not a one off occurrence, as in general the Virtual Cluster out performs Condor. In

Table 7.4: Comparison of two experimental runs for a synthesized application of 100 sixty second Tasks, one using Condor in its traditional manner, the second using the Virtual Cluster. Both approaches had a similar AM.

Performance Metric	Performance of	
	Condor	Virtual Cluster
TE	70%	100%
Avg TR (in seconds)	108.5	60.9
AM (in seconds)	946	973
ERU	38.7%	91.5%
Av No. of Workstations	16.408	6.738

Figure 7.1 on average 3.786 workstations were used and the AM was 142 seconds. When compared to Figure 2.14 where on average nearly double (7.141) the number of workstations were used, but the AM was almost a factor of ten times longer at 1195 seconds. Similarly, in Figure 7.2, an average of 15.596 workstations were used, for comparison in Figure 2.15, 16.408 workstations were used. Here the Campus-Grid is performing better due to the longer Task length, however, its AM is almost double the AM for the Virtual Cluster, even when using a comparable number of workstations. In this case the traditional Campus-Grid approach needed 946 seconds whereas the Virtual Cluster needed 492 seconds for the same Task space. Table 7.5 summarizes these two comparisons.

Table 7.5: Further comparison between the traditional Condor usage and the Virtual Cluster, based upon the average number of workstations in use.

Performance Metric	Task Runtime			
	1 second		60 seconds	
	Condor	Virtual Cluster	Condor	Virtual Cluster
TE	82%	100%	70%	100%
Avg TR (in seconds)	69.9	1.01	108.5	60.1
AM (in seconds)	1195	142	946	492
ERU	1.17%	18.6%	38.7%	78.2%
Av No. of Workstations	7.140	3.785	16.408	15.600

Even if the Task runtime is increased to 600 seconds, the Virtual Cluster still outperforms Condor. Figure 7.3[5] graphically juxtaposes two experimental runs of 100 synthesized Tasks of 10 minutes in length. Here, an earlier version of the Virtual Cluster was deployed, the main difference being that the CRM did not yet inform the CSM when a workstation was suspended. The consequences of this can be seen around the 3000 second point of the execution, where the productivity of the Virtual Cluster flatlines. The scheduler in use was also not an eager scheduler which makes use of redundancy scheduling. Despite this the completion time of the Virtual Cluster far surpasses that of Condor. There is also one other point which can be raised from the result illustrated in this Figure, namely, the difference in consistency of Task completion. This occurs because when

[5]Presented in Ref [38].

the number of unscheduled Tasks (T) approaches zero, Condor can only allocate a maximum of T workstations, which are not guaranteed to function correctly. For this reason, the final Tasks take much longer to complete. This is not the case using the Virtual Cluster, as the workstations have already been allocated. Therefore as T approaches zero, the scheduler simply reduces the number of IAS instances in use and continues executing the remaining Tasks.

Table 7.6 juxtaposes the values of this performance metric as delivered by the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, using the execution scenario presented in this Section. This shows that for the average case the Virtual Cluster offers a reduction in AM over Condor by 92% and 47% for one and sixty second Tasks, respectively.

Figure 7.1: Graphical representation of the runtime of 100 synthesized Tasks of one second in length. In this experiment the average number of workstations used was 3.786 and the resource utilization efficiency was 18.60%.

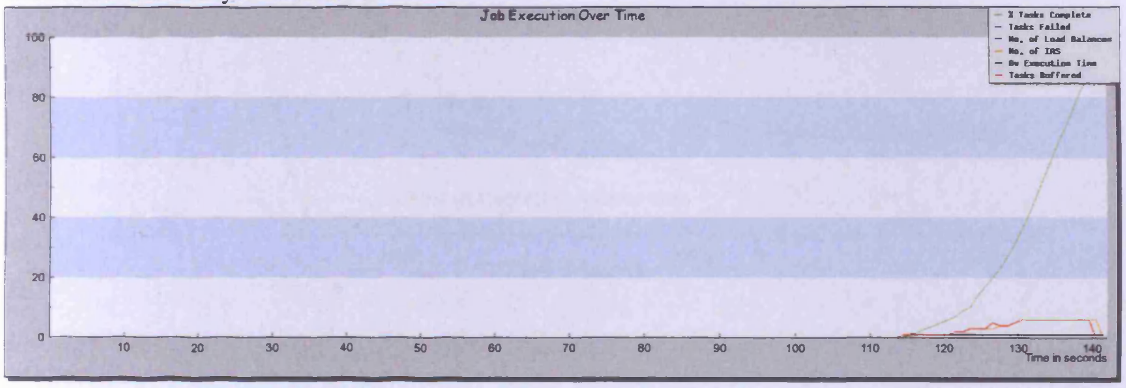
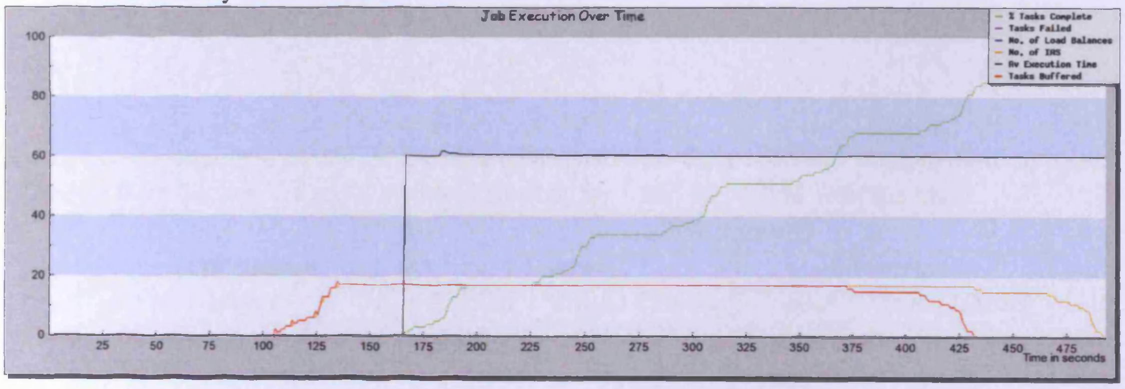


Figure 7.2: Graphical representation of the runtime of 100 synthesized Tasks of sixty seconds in length. In this experiment the average number of workstations used was 15.596 and the resource utilization efficiency was 78.19%.



7.2.4 ERU Performance

The ERU is again largely improved, but is not yet good enough for the one second Tasks. In Section 2.4 the best reported ERU was 1.172% for one second Tasks and 39.976% for sixty second

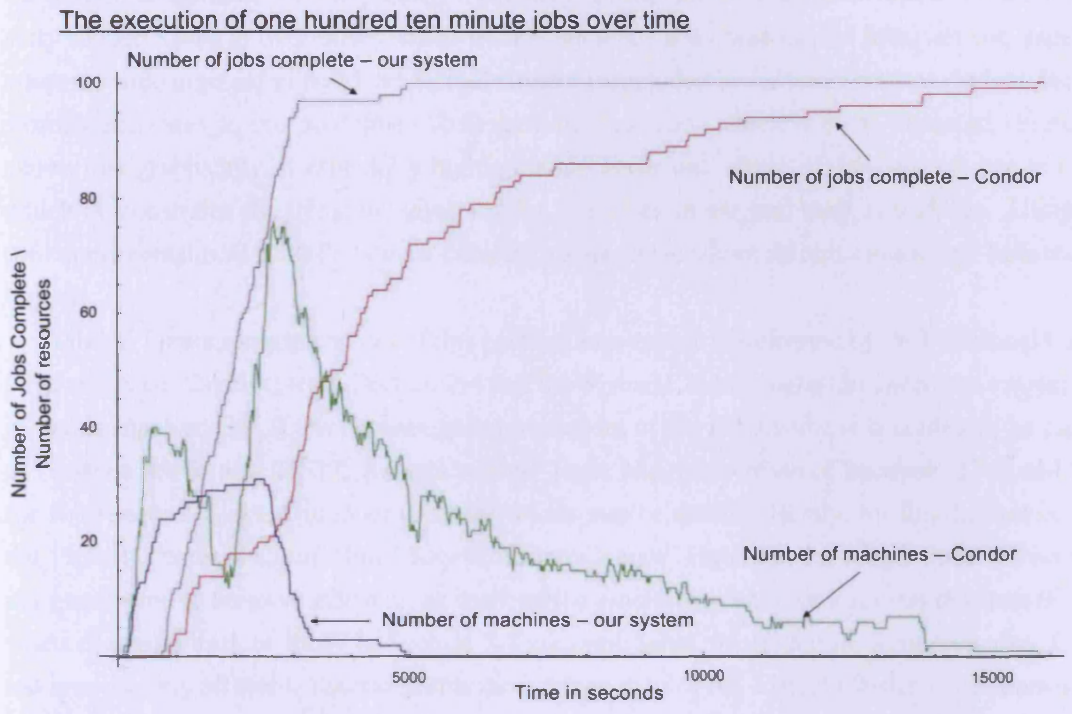


Figure 7.3: Graphical comparison of 100 synthesized ten minute Tasks with a regular Condor approach and a earlier version of the Virtual Cluster. In both cases an average number of workstations used is comparable and approximately 17.

Table 7.6: Comparison of the AM performance between the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, for 1 and 60 second Task Runtimes.

Performance Metric	Task Runtime 1 second Condor	1 second Virtual Cluster	60 seconds Condor	60 seconds Virtual Cluster
Minimum AM	642	132	946	454
Average AM	2220.9	170.2	1865.5	996.5
Maximum AM	12830	240	7345	2043
AM on 1 local workstation	100	100	6000	6000

Tasks. However, for the Virtual Cluster the minimum ERU for one second Tasks is 7% and the average and maximum are 13.8% and 18.6% respectively, which are significant improvements over the traditional Campus-Grid approach. It can also be seen that the maximum efficiency achieved for sixty second Tasks is over 90%, which by any standard is a considerable achievement, especially when the time required to build the Virtual Cluster is included in the measurement. In fact, the ERU steadily increases as the predefined Task runtime increases, which is to be expected. Figure 7.4 shows this graphically, it especially highlights the occasions where performance is not too good, which demonstrates the dynamic nature of the resources in use and their availability. Ultimately, the experimental runs with the lowest efficiencies are those where resource autonomy failures were highest.

Table 7.7 juxtaposes the values of this performance metric as delivered by the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, using the execution scenario presented in this Section. If the percentage improvement of the ERU value is considered: an increase of between 498% and 2652% for one second Tasks and an increase of between -17% and 227% for sixty second Tasks, a number of observations can be made.[6] Firstly, for the shortest of Tasks the Virtual Cluster is significantly outperforming Condor. However, for longer Task runtimes it is not guaranteed to be more efficient, as workstation autonomy can have a serious detrimental effect when examples such as those in Section 7.4 are considered. Note that in these scenarios, Condor too is negatively affected. Secondly, that the average case of the Virtual Cluster yields consistently a significant improvement over Condor. Finally, as Task runtime decreases the benefit of using the Virtual Cluster increases in regard to the ERU value.

Table 7.7: Comparison of the ERU performance between the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, for 1 and 60 second Task Runtimes.

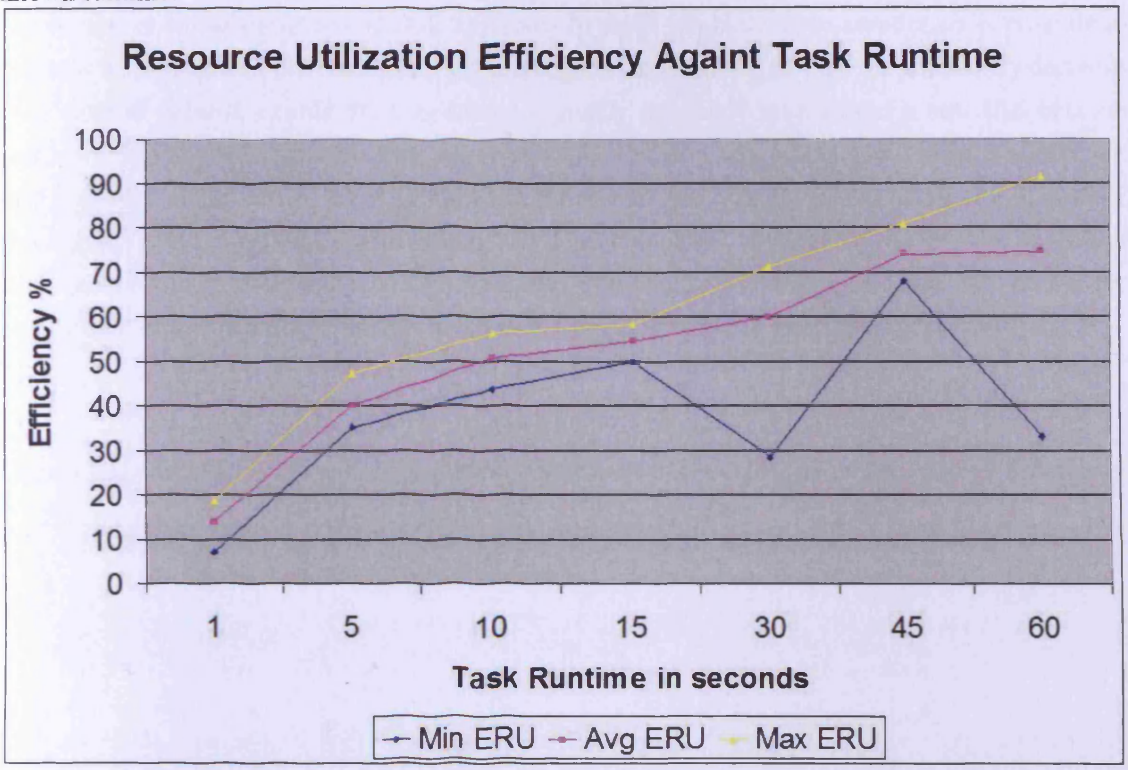
Performance Metric	Task Runtime			
	1 second		60 seconds	
	Condor	Virtual Cluster	Condor	Virtual Cluster
Minimum ERU	0.676%	7.010%	27.950%	33.169%
Average ERU	0.898%	13.8%	35.581%	74.684%
Maximum ERU	1.172%	18.602%	39.977%	91.512%

7.2.5 Summary

This set of experiments has shown that the use of a Virtual Cluster provides improvements over the standard Campus-Grid approach in each of the four performance metrics. This is achieved through two core abilities: firstly, the ability of the CRM to absorb and manage issues related to workstation volatility, and secondly, the ability to reuse a workstation for multiple Tasks without the need to reinitialize the workstation for subsequent Tasks. This also means that a scheduler can schedule

[6]The values are calculated by the following two functions: (1) $min = \frac{Max\ Condor\ ERU}{Min\ VirtualCluster\ ERU} - 1$ (2) $max = \frac{Min\ Condor\ ERU}{Max\ VirtualCluster\ ERU} - 1$

Figure 7.4: Efficiency of Resource Utilization against Task Runtime for 100 synthesized Tasks of a known runtime.



additional Tasks to a workstation at any time and as many times as required, thereby building a queue on the workstation, which further decreases overhead. The results presented in this Section have shown that a volatile and dynamic Campus-Grid can in fact support short jobs of as little as 5 seconds in length. Furthermore, it can perform this analysis faster than sequential execution of the Task space on a single local workstation. The consequence of an improvement in the TE, TR and ERU performance metrics was that the overall AM of an application could be significantly improved, even when fewer workstations were in use. In some cases the AM and the average number of workstations used were both significantly decreased in the same experimental run.

Using the setup presented in this Section, it is not yet practical to execute a Task space where the Task length is in the order of 1 second. This is due to the unpredictable nature of the Campus-Grid to provide one or more workstations and time required to initialize the workstations provided. Here, the time spent to acquire and initialize the workstations strongly outweighs the time spent executing the Tasks, which is the primary reason for the poor performance of the Virtual Cluster for Tasks of this length. However, note that if the number of Tasks was increased, the ERU value would also increase, and the AM decrease relative to the number of Tasks performed, as the Virtual Cluster would be in use for a longer period of time. Therefore, the penalty of initializing the Virtual Cluster would be more closely related to the time it was in use, making the build time less detrimental to performance.

Figures 7.1, 7.2 and 7.5 – 7.9 show example graphical representations of the experimental runs conducted in this Section when a Virtual Cluster is built and then used, to give an impression of how the system performs for different Task runtimes. In these graphical representations it is possible to see how the IAS queues are managed; they are larger at the beginning of the run and slowly decrease over time in order to enable the scheduler to quickly rebalance load should a new IAS become available. It is also clear that the time required by the CRM to acquire resources is quite significant and is often a major part of the AM for an experimental run. Another common feature in each of the Figures is the consistency with which Tasks are completed, denoted by the average execution time being a straight line, and also the linear rate with which Tasks complete. Note also that at the end of each experiment the scheduler releases IAS instances as the final Tasks are completed.

Figure 7.5: Graphical representation of the runtime of 100 synthesized Tasks of five seconds in length.

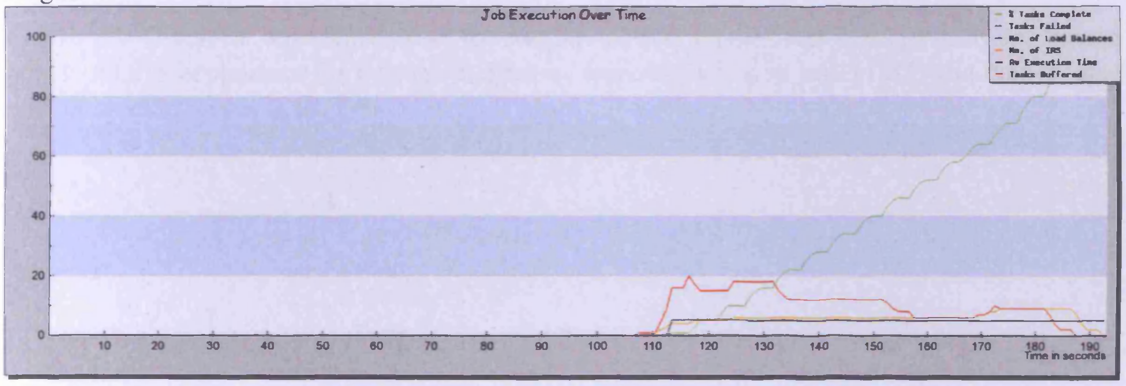


Figure 7.6: Graphical representation of the runtime of 100 synthesized Tasks of ten seconds in length. In this experiment the average number of workstations used was 9.658 and the resource utilization efficiency was 49.31%.

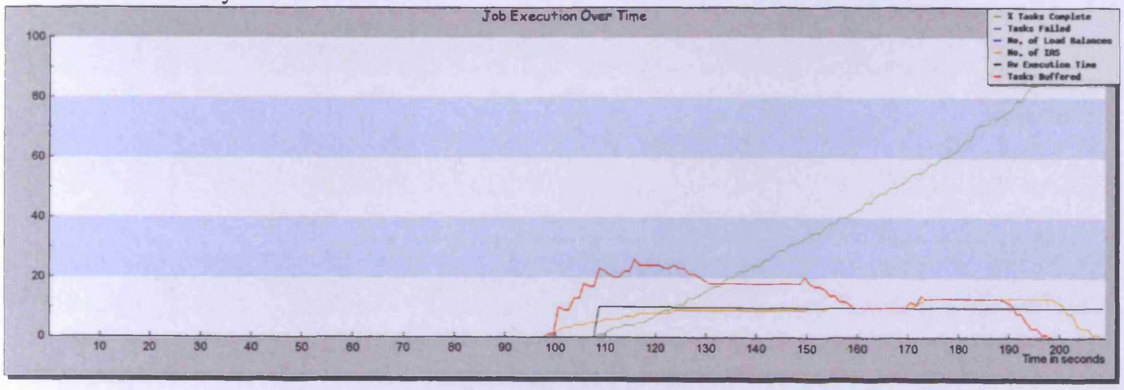


Figure 7.7: Graphical representation of the runtime of 100 synthesized Tasks of fifteen seconds in length. In this experiment the average number of workstations used was 9.331 and the resource utilization efficiency was 54.12%.

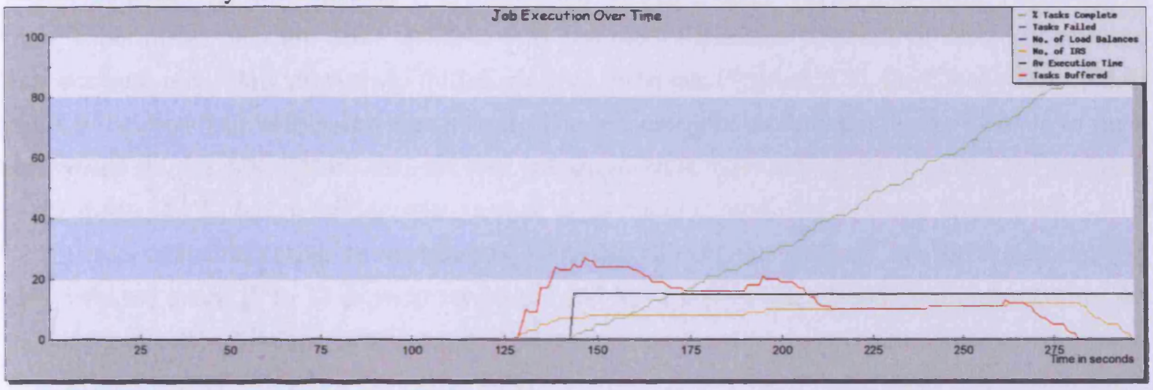


Figure 7.8: Graphical representation of the runtime of 100 synthesized Tasks of thirty seconds in length. In this experiment the average number of workstations used was 11.070 and the resource utilization efficiency was 64.53%.

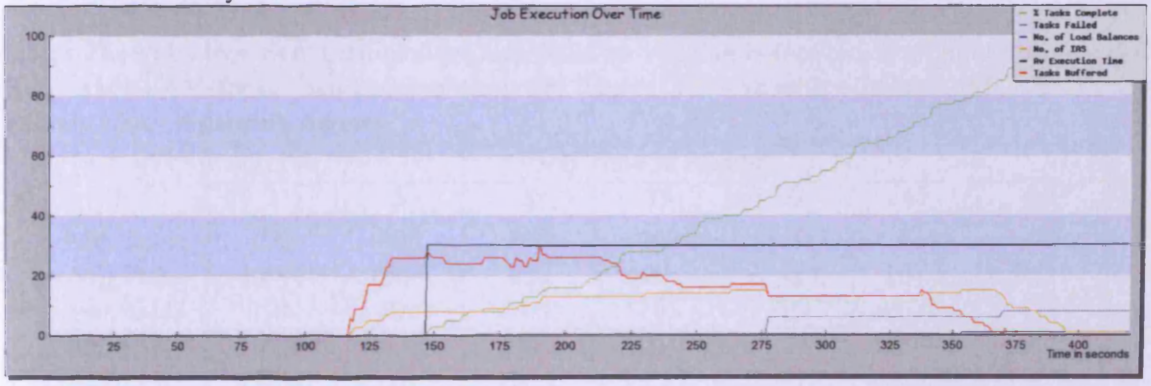
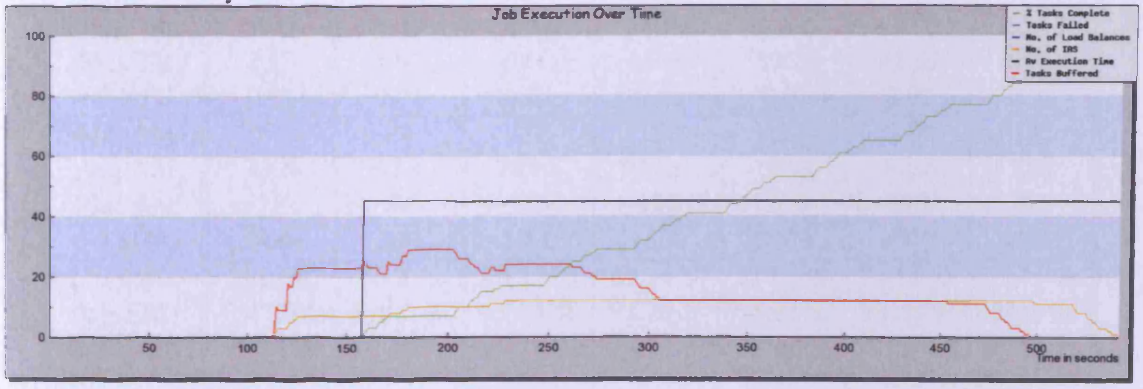


Figure 7.9: Graphical representation of the runtime of 100 synthesized Tasks of forty-five seconds in length. In this experiment the average number of workstations used was 10.732 and the resource utilization efficiency was 77.36%.



7.3 Recycling the Virtual Cluster

In the previous Section, the system was shut down and restarted after each experimental run. In this Section the benefit of the ability to recycle the Virtual Cluster will be presented. Therefore, in these experiments only the CSM (Section 5.4) and the CI (Section 5.5) will be restarted between experimental runs. Any previously initialized IAS instances (Section 5.2), the CRM (Section 5.3) and LS (Section 5.6) will be left untouched. The reasoning behind restarting the CSM is to flush the system as this will ensure that all IAS instances were released by the CI after the previous experiment. The CI has to be restarted, as each experiment is modelled as a new application.

In order to enable a range of workstations to be used during this analysis, the Virtual Cluster was also restarted every 10 to 15 experimental runs and left for between 10 and 30 minutes before the testing Application Manager was permitted to commence the next experiment. This period of time reflects that needed to ensure both that different workstations were in use, and that a reasonable number of workstations were acquired prior to the commencement of the next experiment. The results of recycling the Virtual Cluster are presented in Table 7.8, and the performance metrics discussed below.

Table 7.8: Table of the performance of 100 synthesized Tasks using the approach documented in this thesis. Here IAS jobs were preinitialized and therefore running before the client joined the system. Key: AM1 - AM for one workstation when run locally, VF - Volatility Failures, AF - Autonomy Failures, BA - Balancing Actions.

	Job Time in Seconds						
	1	5	10	15	30	45	60
Min Tasks	100	100	100	100	100	100	100
Avg Tasks	100.4	100.4	100	100.2	103.4	103.6	102.6
Max Tasks	104	102	100	101	109	112	112
Min TE	100%	98.04%	100%	99.01%	95.24%	91.74%	100%
Avg TE	100%	99.61%	100%	99.8%	97.87%	98.15%	100%
Max TE	100%	100%	100%	100%	100%	100%	100%
VF	0%	0%	0%	0%	0%	0%	0%
AF	100%	0%	0%	0%	35.29%	44.44%	100%
BA	0%	100%	0%	100%	64.71%	55.56%	0%
Min TR	1.01	5.02	10.01	15.02	30.02	45.04	60.03
Avg TR	1.05	5.04	10.05	15.03	30.05	45.22	60.14
Max TR	1.13	5.07	10.11	15.04	30.11	45.92	60.31
TR STD	0.01	0.01	0.01	0.01	0.08	0.07	0.33
Min AM	7	29	73	108	123	273	182
Avg AM	11.5	42.4	104.8	123	170.8	340.6	295
Max AM	25	54	153	138	213	610	482
AM1	100	500	1000	1500	3000	4500	6000
Min ERU	28.94 %	83.2 %	90.35 %	87.11 %	81.01 %	71.96 %	75.16 %
Avg ERU	73.73 %	87.36 %	93.15 %	92.28 %	91.71 %	92.25 %	94.2 %
Max ERU	88.14 %	90.15 %	94.71 %	95.32 %	98.11 %	98.6 %	99.55 %

7.3.1 TE Performance

No improvements in TE performance are possible through the recycling of the Virtual Cluster.

7.3.2 TR Performance

No improvements in TR performance are possible through the recycling of the Virtual Cluster.

7.3.3 AM Performance

There is yet another significant improvement in the AM due to the ability to recycle previously initialized IAS instances. The main reason being that the cost of building the Virtual Cluster can be greater than the time required to execute the set of Tasks, as was the case for the one and to some extent the five second Tasks in the previous Section. These results also demonstrate that the use of the Virtual Cluster is beneficial in regard to result latency, even for Tasks of one second in length. In the previous Section the minimum AM for 100 one second Tasks was 132 seconds, 32 seconds longer than the AM of a single workstation, which suggested that the Virtual Cluster is not feasible for Tasks of 1 second in length. However, in Table 7.8 the maximum recorded AM is 25 seconds, well below the AM for one workstation and therefore demonstrating that even Tasks of one second in length can feasibly be supported in an opportunistic, dynamic and volatile environment. These results show that a large collection of Tasks of very short runtimes can benefit from parallel execution, even in such a computational environment.

Table 7.9 juxtaposes the values of this performance metric as delivered by the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, using the execution scenario presented in Section 7.2 and this Section. Here the utility of being able to recycle the Virtual Cluster is clearly demonstrated. In comparison to when the Virtual Cluster had to also be constructed, the decrease in the AM is on average 93% for one second Tasks and 70% for sixty second Tasks. If this is compared to the use of Condor, then the decrease in AM is on average 99% for one second Tasks and 84% for sixty second Tasks. Here a decrease of 100% would be if the application was completed instantaneously.

Table 7.9: Comparison of the AM performance between the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, in both the execution scenarios: when built from scratch (Section 7.2) and recycled (this Section) for 1 and 60 second Task Runtimes.

Performance Metric	Task Runtime					
	1 second			60 seconds		
	Condor	Virtual Cluster		Condor	Virtual Cluster	
		built	recycled		built	recycled
Minimum AM	642	132	7	946	454	182
Average AM	2220.9	170.2	11.5	1865.5	996.5	295
Maximum AM	12830	240	25	7345	2043	482
AM on 1 local workstation	100	100	100	6000	6000	6000

7.3.4 ERU Performance

The ERU demonstrates another significant improvement for all Task runtimes, and efficiencies of 88.141% and 99.554% were recorded for Tasks of one and sixty seconds in length respectively. This clearly demonstrates the potential of a Virtual Cluster of resources, even for the very shortest of Tasks. The improvement in the ERU is due to one new key attribute of the Virtual Cluster. Namely that Tasks complete in distinguishable blocks, which is characterized by the rising peaks in Figures 7.12 – 7.17. This is a direct consequence of the scheduler receiving multiple IAS instances simultaneously at the very beginning of the execution of the Task space. When the Virtual Cluster is built, IAS instances are allocated to the Application Manager in an ad hoc manner, because this is the nature in which each workstation initializes. This translates into a less effective scheduling scenario, as Tasks are scheduled in "*micro-clusters*" and only one cluster is scheduled at any given moment in time. When the Virtual Cluster is recycled, multiple clusters are scheduled at the same time, which means they will also finish at or around the same time in this case. The effect is that the scheduling strategy is greatly simplified, because the number of scheduling steps becomes a function of: $\lceil \frac{T}{N} \rceil$, where T is the number of Tasks and N the initial number of workstations. Generally, this will not occur as uniformly, as resource performance is not homogeneous. This does not mean that it will not happen, only that it is less likely. In addition, the scheduler has the option to choose how many resources it receives. In these experiments the scheduler tries to maximize its resource pool (see: Section 5.5.3), rather than optimize its performance by identifying a "*good*" number of resources from those currently available to use. The direct consequence of the scheduler used in these experiments is that the average number of resources used is generally higher than the experiments conducted in Section 7.2. However, the ERU is not negatively affected, and the improvement in performance is so significant that 100 sixty second Tasks can be completed with a lower AM, when recycling resources, than 100 one second Tasks when the Virtual Cluster must also be built.

Table 7.10 juxtaposes the values of this performance metric as delivered by the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, using the execution scenario presented in Section 7.2 and this Section. By recycling the Virtual Cluster it is clear that the ERU has increased. However, if these values are compared relatively to those of the Condor system, then the significance of the improvement that has been achieved becomes clearer. For the one second Tasks, the improvement in the ERU is between 2369% and 12939% and for the sixty second Tasks, between 88% and 256%. In comparison to when the Virtual Cluster must also be built, the improvement in ERU for one second Tasks is between 56% and 1157%, and for the sixty second Tasks, between -18% and 200%. This also shows that recycling the Virtual Cluster for longer running Tasks is not guaranteed to improve efficiency, but in the average case an improvement of 26% is experienced.

In comparison to the similar approaches presented in the literature, namely: ProteomeGRID and DIANE (both discussed in Section 3.2.1), which reported ERUs of up to 81% and 84% respectively then the use of the Virtual Cluster has outperformed both approaches with a monitored

Table 7.10: Comparison of the ERU performance between the traditional Campus-Grid approach (Condor) from Section 2.4 and the Virtual Cluster, in both the execution scenarios: when built from scratch (Section 7.2) and recycled (this Section) for 1 and 60 second Task Runtimes.

Performance Metric	Task Runtime					
	1 second			60 seconds		
	Condor	Virtual Cluster		Condor	Virtual Cluster	
		built	recycled		built	recycled
Minimum ERU	0.676%	7.010%	28.939%	27.950	33.169%	75.159%
Average ERU	0.898%	13.8%	73.729%	35.581%	74.684%	94.205%
Maximum ERU	1.172%	18.602%	88.141%	39.977%	91.512%	99.554%

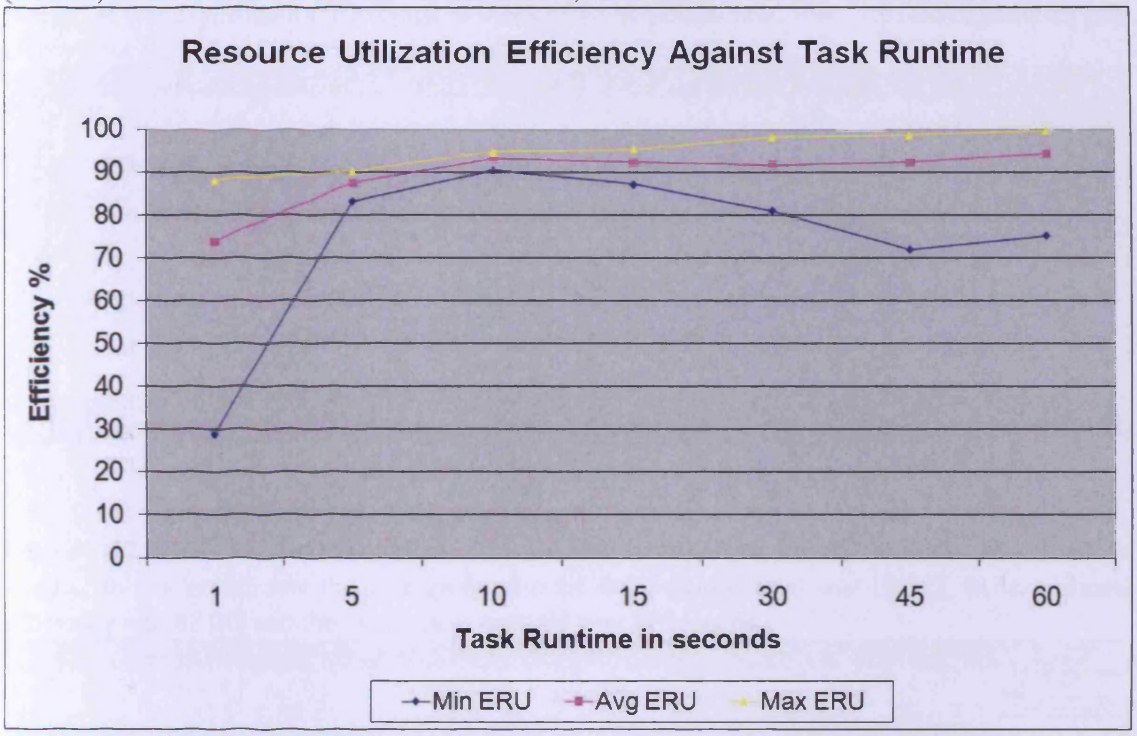
ERU of 88.1% even for a 1 second Task. Note that the experiments do not include data transfer and are therefore best case scenarios. However, an application such as a parameter sweep requires data to be sent only once to each IAS instance. Similarly, if Tasks share some similarity in data requirements, data-aware scheduling strategies can be introduced to attempt to schedule Tasks to IAS instances which have already downloaded some or all of a Task's data dependencies. Finally, the IAS instances' queues permit schedulers to increase the time that an IAS has to transfer data, as the IAS can download any data requirements whilst executing other Tasks.

In Figure 7.10 the ERU results are presented graphically to illustrate that Tasks above 60 seconds in length will be difficult to run more efficiently, as the ERU values are approaching 100%. This does not mean that longer running Tasks will be less efficient, only that the benefit of using the Virtual Cluster over a more traditional approach will diminish as Task length increases. In this sense the benefit received using the Virtual Cluster is inversely proportional to Task length. In other words, as Task length approaches zero, the benefit of using the Virtual Cluster increases. This can be substantiated by the observations in was improvements of the AM and ERU metrics. In both cases the improvement in these values relative to the performance of the Condor system, was significantly higher for one second Tasks than for sixty second Tasks.

7.3.5 Summary

The purpose of this Section is to demonstrate the significance of being able to recycle previously initialized IAS instances. This ability is only possible because the IAS is intrinsically and inherently orientated toward the support for heterogeneous Tasks. Therefore, it is feasible that this ability can dramatically increase the ERU and reduce the AM for a given Task space for real applications. In Section 6.8 the effect of recycling resources was shown for a real application as a reduction in result latency. Ultimately, the ability to recycle resources leads to an improved performance for the shortest Tasks and for small Task sets. This occurs as the time required to build the Virtual Cluster can be predominantly avoided. It cannot be completely avoided, as even in this execution scenario the CRM will initialize new IAS instances if the CSM identifies that more IAS instances could be consumed. Figures 7.11 – 7.17 graphically represent example experimental runs and illustrate how

Figure 7.10: Comparison of the ERU performance between the traditional Campus-Grid approach (Condor) from Section 2.3 and the Virtual Cluster, for 1 and 60 second Task Runtimes.



the execution of a Task space differs when the Virtual Cluster is recycled.

7.4 Fault Tolerance

Figure 7.18 illustrates an example of the autonomy of the Campus-Grid and how the CRM reacted to correct this scenario. However, to fully understand the context of this experimental run, some background information is required. Firstly, the time of day that this experiment was carried out is of great significance. The experiment was started at 10:58 am on a weekday during one of the teaching semesters. At this time of day the students have a 20 minute break from lectures between 10:50 and 11:10, as the lecture timetable shifts from starting on the hour to starting at ten past the hour. It is always exceptionally hard to predict what will occur within the Campus-Grid during this time. Sometimes there are no workstations available, as students check their email, surf the web etc., on other occasions workstations are in abundance, as students take a break. On this occasion there were between 200 and 230 workstations available for use by Condor users, which, considering the maximum recorded value of 808, demonstrates that many of the workstations were in use. Of the available workstations between 18 and 33 were reported as being Matlab enabled.

Initially, the CRM managed to initialize a maximum of 15 IAS instances. Yet it can be seen that all IAS instances were lost for a period of time (approximately 6 minutes). The point to be made here is that this occurred at 11:07, just before the next teaching session started. During this

Figure 7.11: Graphical representation of the runtime of 100 synthesized Tasks of one second in length. In this experiment the average number of workstations used was 11.6, the experiment efficiency was 78.37% and the experiment runtime was 11 seconds.

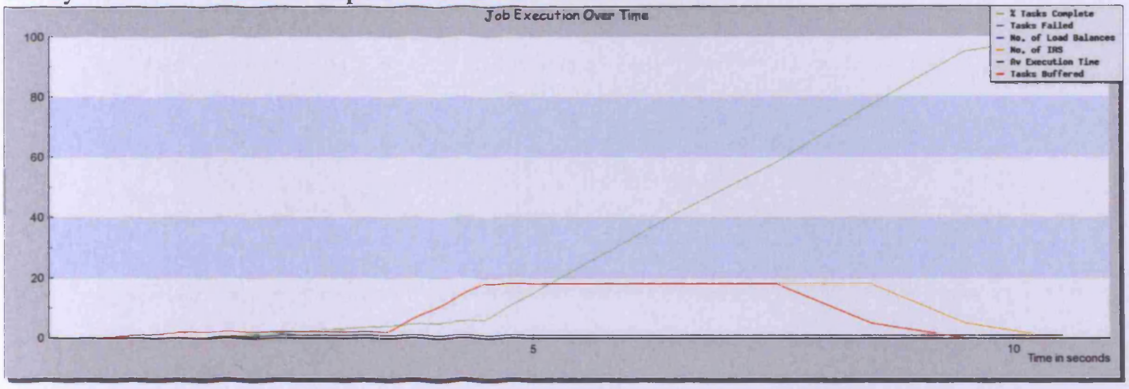


Figure 7.12: Graphical representation of the runtime of 100 synthesized Tasks of five seconds in length. In this experiment the average number of workstations used was 19.592, the experiment efficiency was 88.0% and the experiment runtime was 29 seconds.

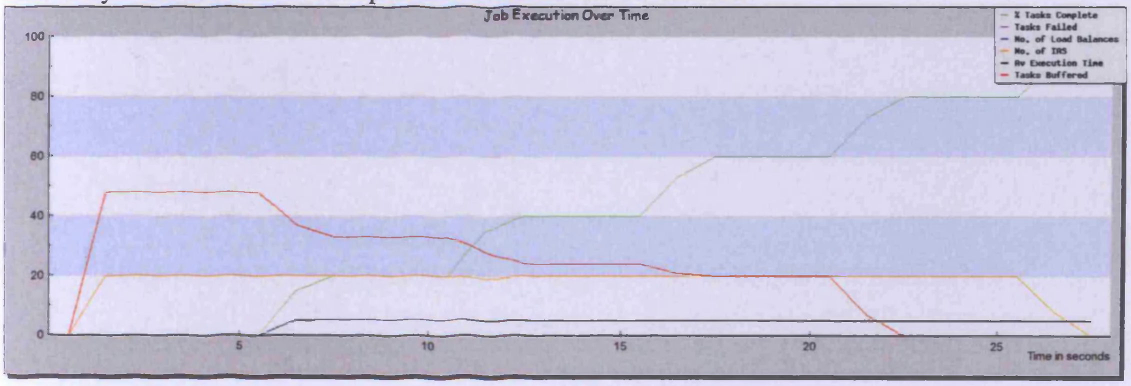


Figure 7.13: Graphical representation of the runtime of 100 synthesized Tasks of ten seconds in length. In this experiment the average number of workstations used was 14.486, the experiment efficiency was 94.56% and the experiment runtime was 73 seconds.

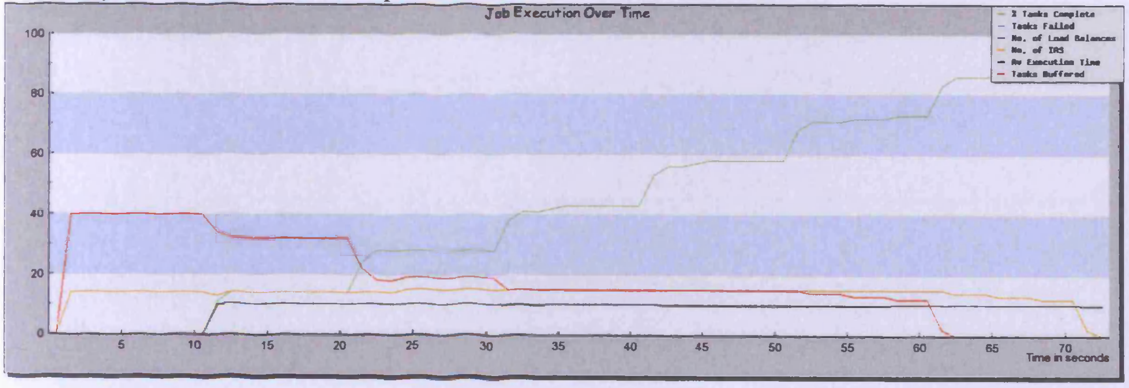


Figure 7.14: Graphical representation of the runtime of 100 synthesized Tasks of fifteen second in length. In this experiment the average number of workstations used was 12.983, the experiment efficiency was 93.93% and the experiment runtime was 123 seconds.

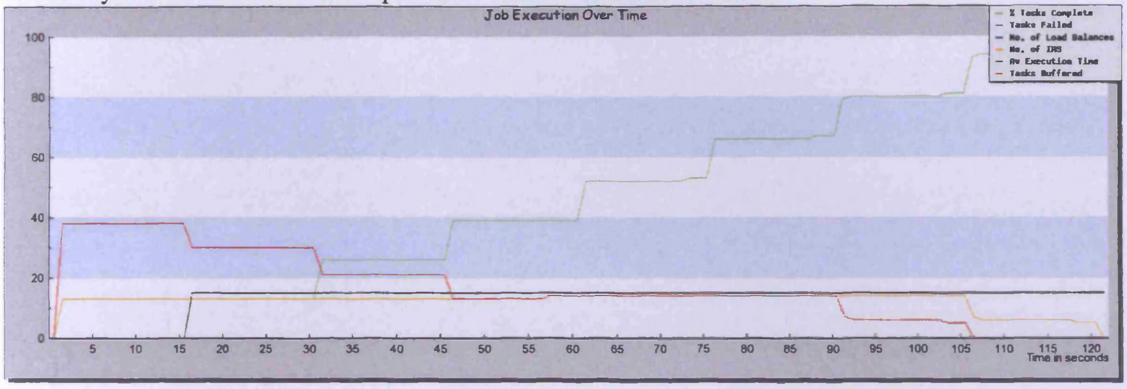


Figure 7.15: Graphical representation of the runtime of 100 synthesized Tasks of thirty seconds in length. In this experiment the average number of workstations used was 16.709, the experiment efficiency was 98.11% and the experiment runtime was 183 seconds.

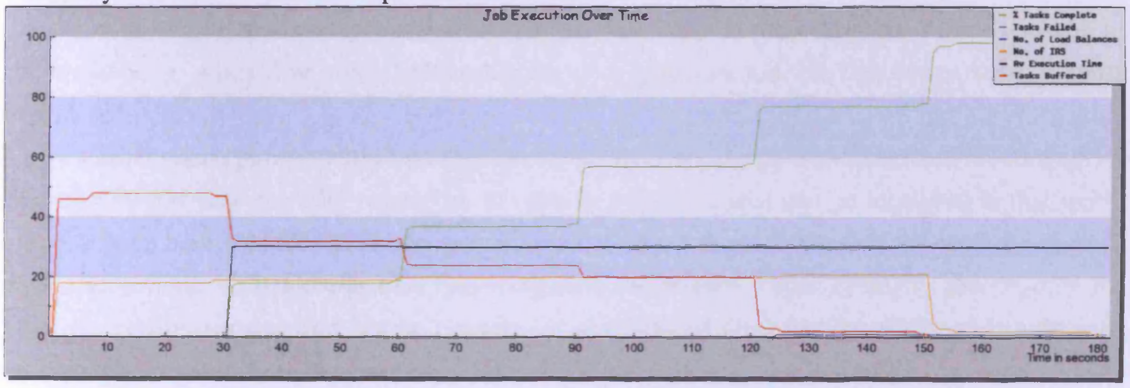


Figure 7.16: Graphical representation of the runtime of 100 synthesized Tasks of forty-five seconds in length. In this experiment the average number of workstations used was 16.717, the experiment efficiency was 98.60% and the experiment runtime was 273 seconds.

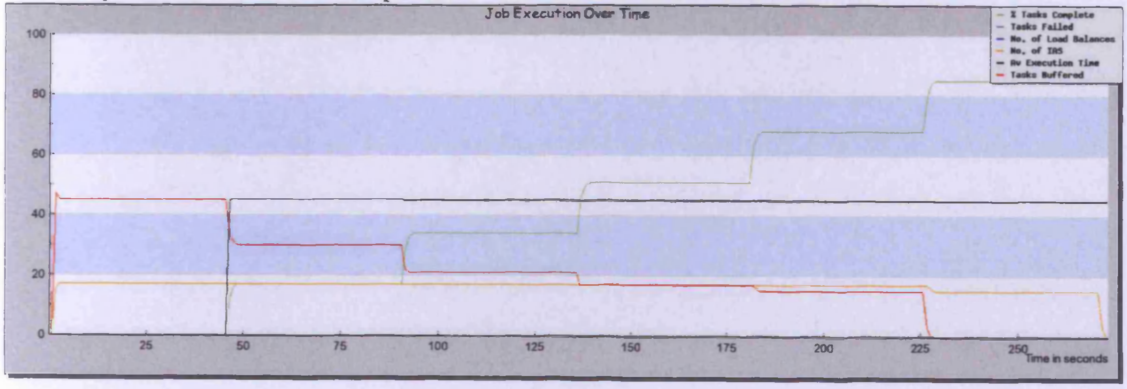
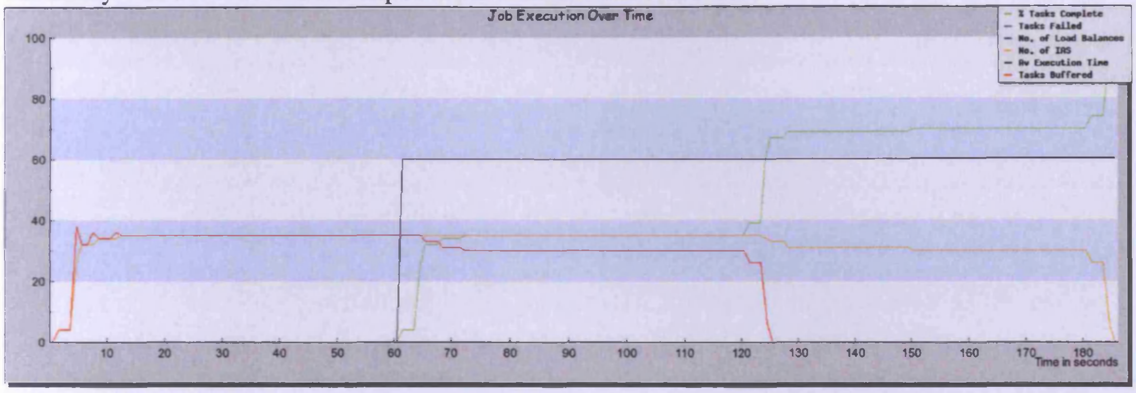


Figure 7.17: Graphical representation of the runtime of 100 synthesized Tasks of sixty seconds in length. In this experiment the average number of workstations used was 32.766, the experiment efficiency was 98.45% and the experiment runtime was 186 seconds.



time the CSM informed the scheduler that the IAS instances had failed and the scheduler identified which Tasks needed to rejoin the queue due to the workstation revocations. Meanwhile, the CRM commenced the process of acquiring new resources for the user and the first IAS instance to (re)join the system was at 11:13. This IAS was allocated to the Application Manager by the CSM, and Task execution continued despite the interruption, and the scheduler continued the execution of the application. Over time more IAS instances were gathered and the Task space was promptly completed.

A similar interruption occurred during the execution of the experiment illustrated in Figure 7.19, however, in this case the only reason for the loss in resources that can be identified is that workstations were being shutdown, as the experiment was run toward the end of the working day. In fact, after around 1220 seconds only one workstation is in use. These examples demonstrate the system's ability to react to dynamic changes in the Campus-Grid and its ability to handle such situations transparently to the user, even when only one workstation remains.

Figure 7.18: Graphical representation of the runtime for 100 synthesized Tasks of sixty seconds in length. In this experiment the average number of workstations used was 12.357 and the experiment efficiency was 41.93%. However, during this experiment it can be seen that at one point all IAS instances were lost.

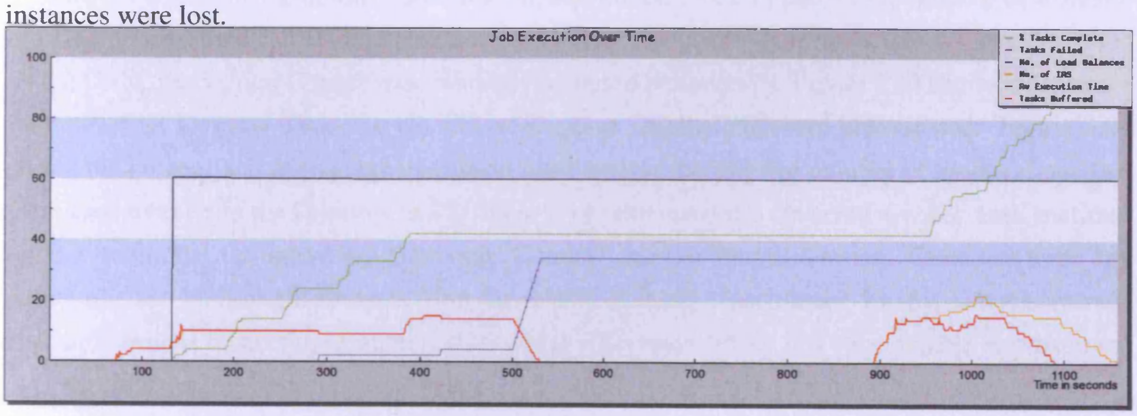
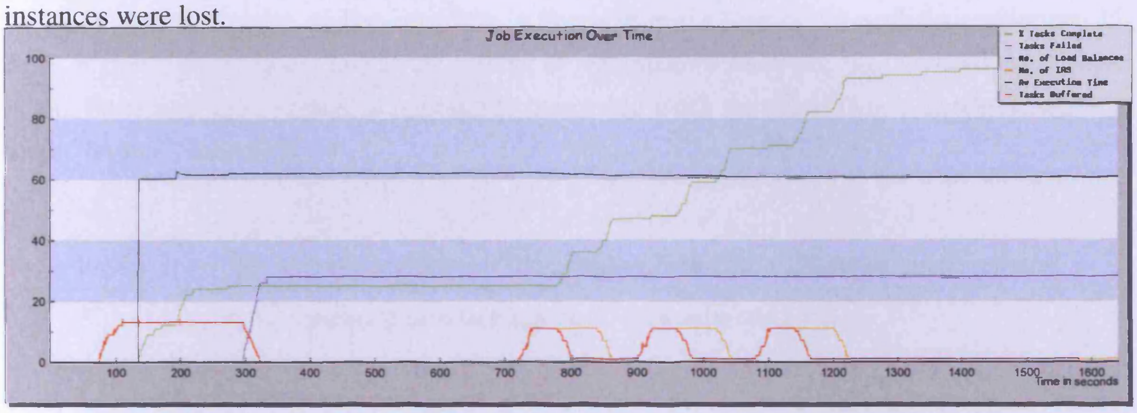


Figure 7.19: Graphical representation of the runtime for 100 synthesized Tasks of sixty seconds in length. In this experiment the average number of workstations used was 11.030 and the experiment efficiency was 33.17%. However, during this experiment it can be seen that at several points all IAS instances were lost.



7.5 Analysis of Queue Length and Overheads

In this Section an evaluation of the impact queue length and communications overhead have on the performance of the system is presented. The analysis undertaken is a replica of that undertaken in Section 2.2, with the only exception that instead of running the Dummy Task directly on the workstation and then allowing Condor to clean up afterwards, the Dummy Task is executed by an IAS instance. Here, the overheads experienced by the Dummy Task relate to the costs of scheduling and transferring it to the IAS, invoking the Matlab engine, generating a result and transferring the results back.

In order to provide a comparison to Section 2.2, the same queue lengths are in use (100, 200, 300, 400, 500, 750, 1000, 2000). It is of particular interest whether the increase in queue size impacts upon the performance of the system as was observed in Section 2.2. Each experimental run was repeated multiple times and the execution scenario is that of Section 7.3. The motivation behind this choice is that no significant differences were observed between the two execution strategies with respect to Task runtime. In addition, using this execution scenario enables a certain level of control over the number of workstations that are used, and therefore the impact of the number of workstations on performance can also be observed. To ensure that a variety of workstations were acquired by the CRM, the Virtual Cluster was manually restarted periodically. Figure 7.20 shows the average runtime of all Dummy Tasks for the different queue lengths that were investigated. Figure 7.21 shows the comparison of average execution time against the average number of workstations that were used to execute the Dummy Tasks. Table 7.11 juxtaposes the observed average Task runtimes for the traditional Campus-Grid approach (Condor) and the Virtual Cluster. There are three key points that can be brought forward from the results of these experiments. Firstly, that the average execution time is on average a matter of 10's of milliseconds, which is a considerable improvement over the experiments undertaken in Section 2.2, where the average execution time ranged between 9 and 67 seconds depending on queue size. This indicates that performance has been improved considerably. Secondly, there is no clear observation that as queue length increases, runtime also increases. Although Figure 7.20 gives the indication that smaller queue lengths are more expensive, the actual difference is quite negligible and could easily be accounted for through inaccuracies in the time keeping abilities of Java, or subtle differences in the load of the workstations in use. Finally, Figure 7.21 illustrates that as the number of workstations increases, the overhead experienced is also more likely to increase, as opposed to increasing when the queue length increases, which is more "normal" behaviour.

Figure 7.20: The average runtimes of the Dummy Tasks for a variety of queue lengths.

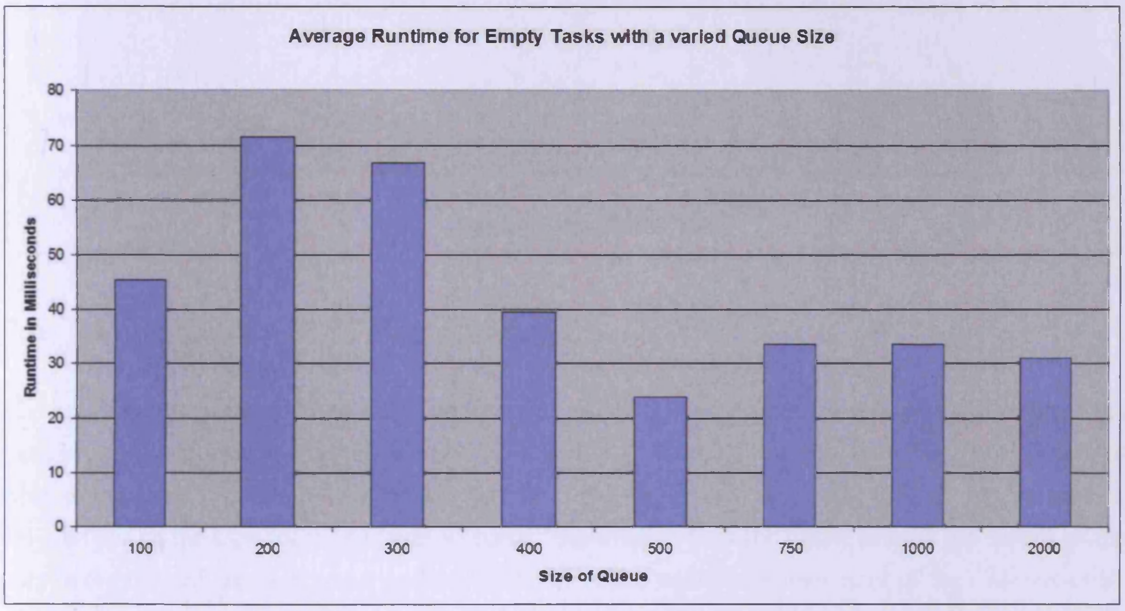
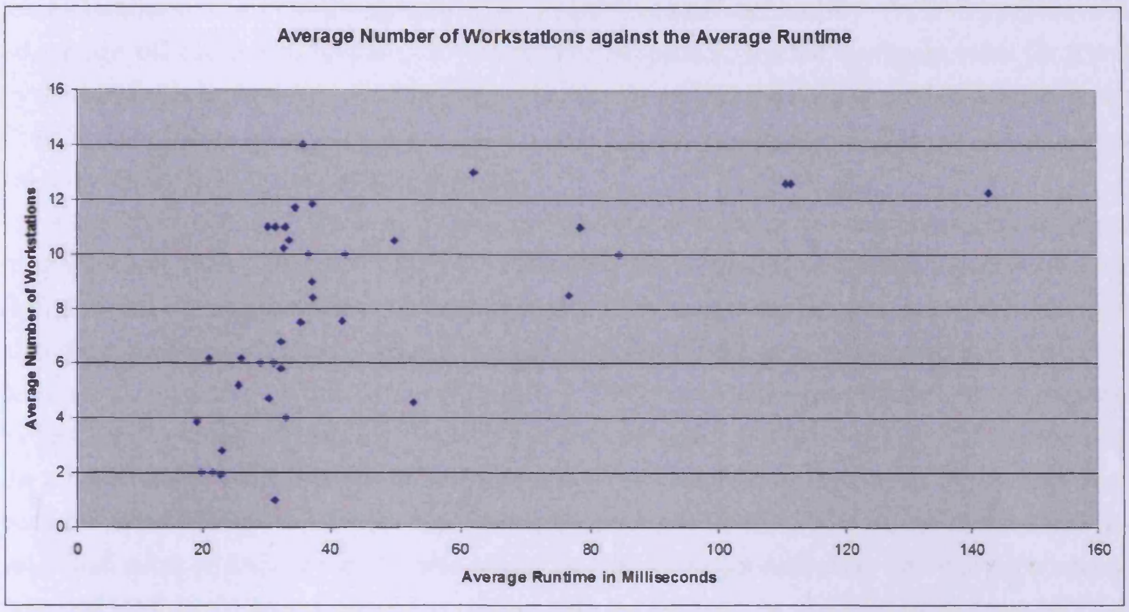


Table 7.11: Comparison of average Dummy Task runtimes using the traditional Campus-Grid (Condor) in Section 2.2 and the Virtual Cluster.

Queue Size	Average Task Runtime (in seconds)	
	Condor	Virtual Cluster
100	25.3	0.0454
200	9.8	0.0716
300	22.0	0.0667
400	32.5	0.0395
500	33.4	0.0239
750	40.5	0.0335
1000	66.9	0.0335
2000	48.9	0.0311

Figure 7.21: The maximum runtimes of the Dummy Tasks for a variety of queue lengths.



7.6 Analysis of CRM Performance

The experiments undertaken in this Chapter have shown that the performance of the Campus-Grid can be greatly improved through the use of a Virtual Cluster. However, they have not illustrated what the system, specifically the CRM (Section 5.3), was doing behind the scenes. In this Section the analysis of the Condor logs is presented to demonstrate how the CRM acts on the behalf of the user in order to acquire resources and add stability to the volatile environment of the Campus-Grid.

The experiments documented in this Chapter relate to over 31,000 Tasks executed by the Virtual Cluster. In contrast, only 1741 IAS jobs were submitted by the CRM. However, note that after each experiment undertaken in Section 7.2 all Condor jobs were manually terminated when each experiment was complete. Note also that even in the second execution scenario (Section 7.3), all IAS jobs were eventually terminated, and were also periodically terminated to enable a variation in the workstations in use. In addition, this number also captures experimental results which are not presented in this Chapter. Such examples include: scenarios where errors in the implementation of the Virtual Cluster were discovered and where glitches in the monitoring data[7] or database access occurred. In these scenarios the results were thrown away, and it is therefore impossible to determine exactly how many Tasks were performed in relation to the number of Condor jobs mentioned above. However, it would not be unrealistic to be at least 10,000 – 20,000 additional Tasks.

The autonomic aspects of the CRM in respect to the monitoring of and reaction to the pilot status of the IAS jobs meant that during these experiments 39 workstations of 112 used were blacklisted and 636 IAS instances were created and consumed by the testing Application Manager (Sec-

[7]This frequently occurred in the experiments undertaken in Section 7.5, as the Task space completed too quickly for enough accurate monitoring data to be acquired.

tion 7.1.3). In contrast, 1 of the workstations blacklisted later successfully ran an IAS job, and of those blacklisted 5 had previously initialized an IAS instance successfully. An IAS instance took on average 142 seconds to initialize, a minimum of 24 seconds, and the maximum value permitted by the CRM was in the order of 11 minutes. On average an IAS job waiting for 171 seconds in the Condor queue before being matched to a workstation by Condor, where the minimum and maximum values were 9 and 10714 seconds respectively.

A breakdown of all IAS jobs submitted to Condor and the cause of their termination or failure is presented in Table 7.12. The different reasons for job termination or failure that occurred are: **(1)** Pilot: the workstation failed the IAS pilot test, **(2)** Manual: the job was manually terminated after the completion of an experimental run, **(3)** Suspended: the job was suspended by Condor, and terminated in response by the CRM, **(4)** Runaway: the job was terminated by the CRM in response to the CRM's termination heuristic (Section 5.3.3) classifying it as a runaway job, **(5)** Unsuspended: the job was unsuspended by Condor and failed as a consequence; in these cases the unsuspension occurred either before the CRM could identify the suspension event because the IAS job had not yet initialized as an IAS, or the IAS was idle at the time, which would occur between experimental runs, and **(6)** Lost Contact: Condor lost contact with the workstation, the most likely causes being a power down, failure of the Condor sched daemon, or a networking issue.

Reason for job termination or failure	Occurrences		CPU Time (in hours)	
	Frequency	Percentage	Quantity	Percentage
Pilot	817	47%	4:30	2.6%
Manual	797	46%	120:38	68%
Suspended	58[8]	3%	25:36	14.4%
Runaway	53	3%	9:52	5.6%
Unsuspended	8	0.5%	6:12	3.5%
Lost Contact	8	0.5%	10:24	5.9%
Total	1741	100%	177:24	100%

Table 7.12: Summary of the causes of IAS Job Termination.

From Table 7.12, several conclusions can be drawn. Firstly, that the total CPU Time observed (177:24 CPU Hours) is much greater than the CPU Time consumed for all experiments (approximately 65 CPU Hours). This occurs primarily because between every experimental run there is a gap of at least five minutes. However at times this was be as much as 30 minutes, and in Sections 7.3 and 7.5 all IAS instances were left running during this time. At times there were up to 30 or 40 IAS instances running concurrently, and consequently a short five minute pause can quickly consume almost one CPU hour. When repeated between all experiments, this relates to a significant period of time, which is actually quite difficult to quantify. An estimation of this period is in the order of be-

[8]In fact 105 IAS jobs were suspended in total. However, the CRM will only terminate a suspended IAS job if it is in use, as it is guaranteed to fail should it be unsuspended. This is not always the case if it is idle. The remaining 47 jobs can be accounted for as follows: 40 were terminated manually when an experimental run was complete or the Virtual Cluster reset, 3 were later unsuspended, but failed the pilot test. Condor lost contact with the workstations of the another 3 and 1 was later unsuspended and consequently failed.

tween 16 and 23 CPU hours as a lower bound and between 30 and 37 CPU hours as an upper bound. In addition, it was not uncommon for the experiments in Section 7.5 to be repeated five or six times before a useable result was captured. This was typically due to the fact that the application would complete in less time than the monitoring periodicity. Therefore, it was not possible for the testing Application Manager to calculate the required performance metrics, as it had no data. In addition, the IAS termination process for the experiments conducted in Section 7.2 was not automated and therefore, a period of time could arise between the completion of an experiment and the termination of all IAS jobs, which is difficult to quantify accurately.

Secondly, the pilot testing process of the IAS job is very quick at identifying workstations that are incorrectly configured. On average this process required 20 seconds, the minimum and maximum times recorded were 2 and 222 seconds respectively. It is also quite successful at identifying faulty workstations, as only 3% of all IAS jobs were classified as runaway jobs and consequently terminated by the CRM.

Thirdly, the amount of time consumed by the runaway jobs appears, on the outset, to be quite high for only 53 jobs. However, the average runtime of a runaway job was in fact 11 minutes and 9 seconds, which is actually what the expected result should be.[9] The problem, however, is that there are 4 jobs which the CRM had difficulty in terminating. Here, difficulty in terminating relates to the CRM executing `condor_rm` to terminate the job, but the job not responding. These four jobs collectively consumed 6:44 CPU hours (68% of the total time consumed by runaway jobs), and this is why the total time consumed by the runaway jobs is so high. Why this occurs is difficult to say without access to the logs of the specific workstations in question. However, as this analysis was performed retrospectively, it was no longer possible to access the required log data.

Finally, the period of time that was consumed by the jobs that were terminated because they were suspended or failed, either because they were unsuspended or Condor lost contact with the workstation is perhaps surprisingly high. The main point which needs to be outlined is that these jobs also include any time that was potentially spent as an IAS. In addition, the Condor logs contain only one suspension event for the jobs that failed due to being unsuspended, which means that the CRM had no means to identify that they had in fact been suspended. The one job which was flagged as suspended by Condor was not suspended during an experimental run, therefore it would have been idle at the time and consequently not terminated by the CRM. The jobs which Condor lost contact with cannot be dealt with by the CRM. It is likely that the CRM attempted to terminate them, but this process would have failed due to Condor not being able to contact the workstation, and therefore any attempts made by the CRM are not recorded in the logs. It is not possible to reliably quantify the time that the job spent doing something useful, such as being an IAS, running the pilot test etc., because, as already mentioned with the unsuspended failures, the log data does not always identify when the event first occurred.

The key results that should be outlined through the use of the Virtual Cluster and specifically the

[9]The standard deviation of all IAS initialization times is 113.6971. If this value is inserted into the CRM's termination heuristic using only the average initialization time for all recorded IAS instances (approximately 8000) then the recommended time at which an IAS job should be terminated is after 11 minutes and 50 seconds.

CRM are: firstly, that the number of jobs submitted to the Campus-Grid is significantly lower than the number of Tasks performed, secondly, that the issues surrounding resource volatility and the negative impact this has on performance, as demonstrated in Chapter 2, are managed autonomously and transparently to the user and scheduler by the CRM. Consequently, neither the user nor the scheduler need to be made aware that they are using resources in a volatile and unreliable environment. Finally, it is clear from Table 7.12 that workstations are being invoked and utilized for as long as they are needed rather than for the length of a potentially short Task.

It must be noted that there is the potential for inaccuracies in the data presented in this Section, which stems from three sources. Firstly, is the observation made in Section 2.2, that the accuracy of the data stored in the Condor logs is questionable with respect to when a job was actually terminated. Secondly, the data for all experiments is stored in daily Condor logs, rather than one log for each experimental run. The latter would not be possible for the experiments conducted in Sections 7.3 and 7.5, as IAS jobs submitted in previous experiments are recycled. Therefore in order to collate data, the time stamps used in experiment database are used to identify which IAS jobs were submitted or running during the same time frame as an experiment. The potential for inaccuracies here is because there are three different clocks in use: (1) the clock of the workstation hosting the Application Manager and the experiment database, (2) the clock of the workstation hosting the CRM, which defines the times recorded in the Condor logs and (3) the clock of the server hosting the CRM's database, which defines when events such as the CRM terminating a job are recorded. Each of these clocks cannot be guaranteed to be perfectly synchronized, however, all are regulated by an Internet time server. Finally, all timestamps are defined by when the data was enterred into the database or log, rather than when the event occurred. Therefore, if there was a delay in enterring the data some timing inaccuracies may occur. However, the impact of any inaccuracy from these three sources should not affect the general trend that these results portray.

7.7 Summary of Results

The key aim of this Chapter has been to demonstrate that the conceptual models discussed in Chapter 4 and the prototype implementation presented in Chapter 5 provide a significant improvement in performance and stability in comparison to the middleware currently available to the Campus-Grid user, i.e. Condor. It also intended to show that Tasks with a runtime of as little as one second can be efficiently and feasibly supported in an opportunistic and volatile environment.

The management of issues relating to the volatile and dynamic nature of the workstations in use have been made much more transparent through the introduction of the CRM. The end effect of this is that the user can focus on what they are trying to achieve, i.e. their research, rather than expend unnecessary effort in the administration of the resource infrastructure that they are using. In addition, the number of Campus-Grid jobs needed to execute a user's application has been significantly reduced (Section 7.6). Furthermore, the system is able to capture and adapt to changes in workstation status making it resilient to the adaptive context incited by multiple tiers of administration. The performance model (Section 4.4.1) enables the system to identify and react to inconsistencies in workstation configuration more quickly, but also differentiate between resource errors and Task errors. This means that once an IAS has joined the Virtual Cluster, it is guaranteed to function correctly until the host workstation revokes its voluntary status from the Campus-Grid.[10] When an intermittent failure occurs, any Tasks can be simply rescheduled to another running IAS instance.

The effect of the management strategies put into effect by the CRM is that the overall reliability of the Campus-Grid is increased from the user's perspective. Here, reliability relates to the recorded Task efficiency values in Sections 7.2 and 7.3, where 100% values were not uncommon. The resource utilisation model exercised through the IAS enables the workstations in use, and by extension an instance of compiled Matlab, to be remotely steered in an interactive manner by a scheduler, which can be application specific. This translates into a Task runtime much closer to that which the Task would receive if executed locally, and also enables data transfers to be issued only once per workstation as opposed to once per Task. The impact of this ability relates to the significant improvements in application makespan observed in Sections 7.2 and 7.3.

The ability to create a compiled Matlab container capable of heterogeneous Tasks enables IAS instances to be recycled for disparate application scenarios, which in turn means that the initialization cost of the system can be avoided for subsequent applications. This ultimately meant that Tasks as short as one second in length could receive noticeable reductions in application makespan, as observed in Section 7.3. Furthermore, this ability relates to a more efficient use of resources observed in Sections 7.2 and 7.3, to the extent that overall application makespan was improved, even though the number of resources in use was sometimes decreased significantly.

The dynamic scheduling and allocation model implicitly elicits dynamic Task clustering or granularity. This again occurs due to the interactive nature of the IAS. By using the IAS interactively, Task granularity is not a significant factor on performance. In the Condor experiments (Chapter 2) performance could be radically improved by packaging more Tasks into each job. The overhead incurred for issuing an IAS with a new task is a matter of milliseconds, as observed in Section 7.5. This enables schedulers to simply issue new Tasks as a workstation completes previous Tasks, or to cheaply perform redundancy scheduling. In addition, by buffering Tasks in remote IAS queues, this cost is largely absorbed, as the IAS is already executing another Task. The user can also benefit from *co-allocation* when data is also transferred, which further reduces overhead. Outside of these controlled conditions it means that the system can utilise faster workstations more effectively.

The findings from these experiments were presented at CCGrid '07: the Seventh IEEE International Symposium on Cluster Computing and the Grid, where it was a best paper finalist and in the Journal of Concurrency and Computation: Practice and Experience. See Refs [33] and [38].

The experimentation and results that have been observed in this Chapter have been focused entirely on the comparative analysis to a campus-wide Condor pool. However, the improvements that

[10]This of course assumes that users correctly use the IAS and do not issue it with Tasks that could cause the IAS to shutdown or fail.

have been made possible by the construction, use and possible recycling of the Virtual Cluster are not unique to only a Condor pool. Many of the causes of poor performance are general issues and not specific only to Condor pools and Campus-Grids. Therefore, any infrastructure that has similar attributes to the Cardiff University Condor pool, will benefit from the adoption of this system. Many of these attributes were outlined in Section 1.3, but in summary the main characteristics are: **(1)** where multiple tiers of decentralized administration exist, but which incite a volatile resource environment, **(2)** a general purpose cycle stealing or batch processing infrastructure, as these are not orientated toward the facilitation of short running Tasks, and **(3)** an infrastructure where resource matching is based upon meta-data that is not linked to a management system. If a given infrastructure has one or more of these attributes, then the improvements in performance documented in this Chapter are attainable, by deploying either the implementation presented in Chapter 5, considering the requirements for porting to another infrastructure presented in Section 5.7.1, or one of a similar nature.

Chapter 8

Conclusions & Future Work

In this final Chapter, the focus will return to the research questions and hypothesis posed in Sections 1.5 and 1.4 respectively. Section 8.1 will present the new knowledge that can be put forward toward answers to these questions. Following this discussion, Section 8.2 will present a crictical analysis of the hypothesis and the research undertaken in this thesis to address it. Finally, in Section 8.3 a new research direction is presented which could capitalize upon this research.

8.1 Answers to the Research Questions

In this Section the four research questions posed in Section 1.5 will be discussed in relation to the research undertaken in this thesis in Sections 8.1.1 – 8.1.4. In each Section, the research question will be repeated, and the relevant models which have been implemented will be discussed. In addition, the analysis performed to demonstrate the added capabilities will also be discussed, along with the new knowledge that has consequently been delivered. Finally, any areas of work which could be performed as further research to augment this thesis will also be highlighted.

8.1.1 Feasibility

Question 1: What measures are needed to make a Campus-Grid feasible for the Task[1] definition given in Section 1.3.9?

In order to answer this question, another one had to be raised first: how do the current conceptual models (job and resource management) need to be extended? This ultimately related not only to the provisioning of Tasks using the definition given in Section 1.3.9, but also to how stability was to be established within the volatile context of the Campus-Grid, as was demonstrated in Chapter 2.

In order to approach these questions, the traditional job model was split into two parts to enable the differentiation between resource acquisition and the scheduling of work units/Tasks. This meant that the process of managing dynamically changing resources, i.e. those which exhibit volatile

[1]The original question in Section 1.5 used the word 'job' here, however, to remain consistent to the terminology introduced in Chapter 4 it has been replaced with Task.

behaviour (see Section 1.3.1), became a challenge for the system as a whole, rather than the user - therefore increasing transparency. On the other hand, it provided the means to allow the user or an Application Manager (Section 5.5.1), acting on their behalf, to be concerned with the definition and scheduling of work units and not the management of a complex and volatile distributed system.

Stability was added through this division, and it became possible to identify when a workstation was not exhibiting "*normal*" behaviour, by submitting a verified pilot job. The pilot job was guaranteed to be successful if and only if the operating environment of the host workstation was correctly set up. Therefore, if the pilot job failed for any reason, this was a clear indication that the workstation's environment was inadequate and incorrectly advertised. In some cases, the pilot job did not simply fail, but became stuck in a state transition, which was termed as a runaway job. In Chapter 2 the potential runtime for a runaway job was demonstrated to reach excessive periods of time, a maximum recorded value was sixteen days. A termination heuristic was defined and introduced (Section 5.3.3) in order to identify when this is likely to have occurred based upon a training set of 5000 Matlab test jobs and all other IAS (Section 5.2) jobs that had successfully initialized in the past, which is approximately 8000 additional jobs. In this sense, the heuristic itself became sensitive to changes in the Campus-Grid. Once the pilot job completed, a worker daemon was initialized to identify that its host workstation was fully capable of accepting work units. This enabled a new state to be introduced in the traditional state space: *initialized*. When a workstation achieved this state, it could be guaranteed that it would function correctly.

The only exception to this statement was if the job (daemon) was suspended. In this scenario, regardless of whether the daemon job was unsuspended or not, the job would fail unless it was idle when the suspension occurred. If the daemon was idle it is likely, but not guaranteed, that it would fail upon unsuspension.[2] Due to the splitting of the job model, determining whether a daemon was idle or not was straightforward. Consequently, if the daemon was not idle at the time of suspension, it was terminated and any Tasks rescheduled. If instead, it was idle, the daemon could be flagged as suspended, and not given any further work to do until it was either evicted, unsuspended or failed.

The way in which stability has been added to the volatile nature of the Campus-Grid was to differentiate between resource acquisition, the allocation of these resources and the scheduling of work units. This is quite strongly reflected in the architecture presented in Chapter 5 where the CRM (Section 5.3), CSM (Section 5.4) and CI (Section 5.5) components respectively address these areas. The outcome from the introduction of these components and the research undertaken in this thesis was that a Task would fail only when the availability of the workstation in use was revoked (termed as an autonomy failure), which, in an opportunistic computing scenario, is to be expected. If a Task were to fail during execution for reasons other than workstation autonomy, it was possible to identify exactly why this occurred, as the daemon executing the Task (the IAS Section 5.2) has error handling mechanisms built into it specifically to capture application specific error messages, and relay these back to the user. The ability to differentiate between a failure due to unreliable resources and those due to programmer or application error was not possible prior to this work.

[2]The reasoning for why this could occur was presented in Section 5.3.3

Once stability, and by extension reliability, was achieved, focus could turn to the provisioning of Tasks with the attributes outlined in Section 1.3.9. The key concept that has enabled efficient provisioning even for Tasks of as little as one second in length has been the daemon model (Section 4.3.1), and its enactor the IAS (Section 5.2). Other approaches in the literature have used daemon models that are similar, but the uniqueness of this work is the way in which its functionality is gathered and the placement of the daemon itself within the application container, Matlab in this case. Other approaches typically sit outside the application container, and therefore must rely on less effective means of application control and steering. There are no other known approaches that can even compete with the way in which the IAS gathers functionality through the use of the Autonomic (Matlab) Compiler (Section 5.2.1), and the impact on performance that this has was demonstrated in Section 7.3. The reason is that the IAS enables a completely heterogeneous application scope, and therefore previously initialized IAS daemons can be recycled without being reinitialized to perform analysis for mutually exclusive and disparate applications. In theory the IAS could be made into a fully functioning, but license free, version of Matlab, by using the Autonomic (Matlab) Compiler presented in Section 5.2.1. However, this would violate the licensing conditions of Matlab and the Matlab compiler.

The functional gathering abilities of the Autonomic Compiler, plus the aim of the IAS to run infinitely, mean that workstations that are correctly configured can be used for as long as they are available rather than for the length of a single Task or cluster of Tasks. The outcome was the ability to see noticeable improvements in the application make span of even 100 Tasks of only one second in length. The other abilities of the IAS that have contributed to this achievement were: **(1)** The ability to build a queue of multiple Tasks on the workstation itself, so that data acquisition and other administrative tasks could be performed whilst another Task was executing. This is essentially dynamic Task clustering and co-allocation. **(2)** A direct connection to and manipulation of the native Matlab engine meant that Tasks could be passed to the engine using exactly the same API as the Matlab workspace does when being used by a real user. This meant that the overhead involved in invoking the engine was negligible and comparable to "*normal*" use. An additional ability that this creates is the capture of errors as they occur, which translates into the ability to provide the user with a meaningful error event and trace of the error's provenance. **(3)** The IAS Meta-Interpreters (MIs: Section 5.1.2) make the means to gather results much more intuitive, as they are orientated to the use case of typical Matlab applications and, if required, can be tailored to new application scenarios quite easily. This means that the results contain only data that is relevant to the Task performed. Middleware, where Condor is one example, generally returns all data that has been created into the working directory of the workstation, or changed during execution. Such an approach is ill-equipped to identify what is actually required without additional explicit instructions included in the Task description. The IAS doesn't need this information, as all data is made available through its internal HTTP server. Therefore, if something is needed but was not provided by the MI, the user or an Application Manager can pull this data from the IAS.

In summary, the research and analysis performed to answer this question lies in the adoption of:

a new job model, adaptive blacklisting and pilot testing of workstations coupled with a proactive strategy for the termination of jobs exhibiting anomalous behaviour, the abstraction of jobs to an infinitely running daemon that controls the computational software (e.g. Matlab) using the internal APIs, the consolidation of entire toolboxes or packages to increase reusability and the displacement of Task scheduling from the Campus-Grid middleware to the application itself.

The research presented in this thesis has been centred around inaccuracies in the advertised capabilities of a workstation. However, only one of the two possible scenarios were considered. Namely, when a resource advertises a capability that it does not actually have. In contrast, some small scale experiments, which will not be discussed here, have hinted toward a scenario where some workstations do not advertise a Matlab capability, but can in fact successfully run Matlab jobs. Such a scenario can occur just as easily as an inaccurate advertisement discussed throughout this thesis. For example, the installation is successful, but interrupted before the workstation's ClassAd can be updated. Currently, there are no means within the defined models to discover such workstations.

8.1.2 Resource requirements

Question 2: How can the number of resources to be used for the execution of a set of Tasks[3] be determined?

This question was approached through the definition of the requirements model (Section 4.5), where an abstract target for the number of resources to be acquired is generated. This target is generated based upon three sources of information: **(1)** Task data, which describes the remaining work (Tasks) to be performed. **(2)** Resource data, which describes the current demand for resources in relation to the resources that have already been allocated, and, in addition, to the Task data. **(3)** Usage data, which describes how the scheduler is using the resources that have thus far been allocated to it. When combined together this information was used to identify an abstract horizon value for the CRM to aim for. Even in a simple form, where consideration is made only by utilizing the maximum number of IAS instances that a scheduler highlights can such a horizon be created. The whole philosophy behind this approach is not to identify an "ideal" number of resources; this could never be the case for reasons such as workstation heterogeneity and autonomy. Instead, it is to provide an indicator, which can be used to model the current demand for resources that one or more users have relative to the current supply. Or in other words, if more workstations were to become available, to enable the CRM to answer the question: Assuming that these workstations are correctly configured, would they be useful? The demand for resources is expected to be dynamic and change with time as Tasks are completed or new ones supplied to the scheduler. Therefore, the CRM's target is constantly adapting.

The actual number of workstations that CRM acquires through its aim to reach this target is based upon several key factors: **(1)** The current supply of idle workstations in the Campus-Grid that advertise the necessary capabilities. **(2)** The CRM's current knowledge of the available work-

[3]Again the original question in Section 1.5 used the word 'job' here.

stations in relation to previously observed performance, i.e. the current blacklist. **(3)** The number of available workstations that are incorrectly configured, but not yet blacklisted. **(4)** The effective priority of the user that the CRM represents within the Campus-Grid. **(5)** The number of other Campus-Grid users who also have jobs awaiting resources. Therefore, the CRM was made sensitive to the state of the Campus-Grid as a whole. For example, if it has already managed to acquire some workstations, but is incapable of acquiring further workstations, it will not increase the number of IAS jobs it has in the queue. The reason for this is that if there are currently jobs in the queue which are idle and some workstations have already been acquired, adding further jobs will not make a considerable difference. However, on the other hand, if the CRM has no registered IAS instances, and Condor is reporting no available workstations, it must place jobs into the queue to encourage the Condor matchmaker to provide one or more workstations. In the latter case, the CRM attempts to capitalize upon Condor's fair share allocation and effective priority strategies. These, in simple terms, define that a user with no or few workstations will be given a higher priority over a user with many workstations in future matchmaking decisions. Therefore, by submitting IAS jobs even though no workstations are available, the CRM is encouraging the Condor matchmaker to provide it with resources as soon as they become available.

To return to the original question, the number of resources (IAS instances) which are allocated to a scheduler is determined based upon two observations. Firstly, the number of IAS instances that the CRM is physically able to acquire based upon the five constraints mentioned above. Secondly, the demand quantified by the requirements model, using the data extracted from the scheduler and Task Model (Section 4.3.2). Here, if the CRM manages to acquire more IAS instances than the requirements model defines, the actual number that will be allocated is based upon the demand quantified by the requirements model. There are several reasons why the CRM can acquire too many IAS instances. However, this commonly occurs either when the application is nearing completion and therefore the scheduler releases IAS instances that are no longer required, or because the global requirements model (Section 4.5) has achieved its aim of acquiring spare capacity. Remember that the global requirements model aggregates the total demand from all users and increases it in order to instruct the CRM to acquire spare capacity. The acquisition of spare capacity is undertaken to enable the CSM to provide additional resources should a sudden change in demand occur.

The core idea that has been implemented is to generate a collection of resources based upon some modelled demand, which was termed a Virtual Cluster of Matlab-enabled workstations, and allocate them to a scheduler as soon as they become available. Due to the way in which this has been realized, the actual number of workstations that are used has become less influential on performance in comparison to other factors of the system as a whole. For example, in Chapter 2 using more workstations had little influence over the completion time of the set of Tasks. Instead, it was the volatility of the workstations in use that defined when the set of Tasks would complete. In the research conducted in this thesis, creating the much needed stability has enabled better performance, even when less workstations are in use. In addition, the interactive nature of how each workstation is utilized through the IAS has enabled the scheduling of the Task set to be performed dynamically, opportunistically and in a just-in-time manner, which enables a scheduler to adapt its scheduling plan and harness new workstations when they become available. The advantage of this approach is that the user does not need to be concerned either with what resources are available, how many should be used or how they can package their set of Tasks together into larger units of work in an attempt to avoid costly scheduling overheads.

8.1.3 Benefit to User

Question 3: What differences does the user experience as a result of introducing these models?

The first benefit that a user can receive, based upon the research undertaken for this thesis, is a significant increase in transparency. Transparency comes at many levels, each of which have different levels of utility in different application scenarios. The areas in which transparency has been increased are: **(1)** The management of volatile and adaptive resources within a dynamic environment. This is provided by the CRM (Section 5.3). Here, the user no longer needs to be concerned with how resources are acquired, or the way in which issues of volatility are managed. **(2)** The gathering of functionality, and the general administration and creation of compiled Matlab instances. This is provided by the Autonomic (Matlab) Compiler (AC) (Section 5.2.1). Here, entire Matlab toolboxes can be captured using a small number of inclusion statements. Any errors within the Matlab functions are highlighted by the AC in batches, which enables the user to perform compilation predominately as a background process. **(3)** The process of Task submission, preparation and administration. This is provided by the CI's (Section 5.5) Task Layer (Section 5.5.2). **(4)** The elicitation of preference with respect to the quantity of resources needed. This is provided by the CI's (Section 5.5) requirements model, which is constructed using the Task and Resource Layers (Sections 5.5.2 and 5.5.3). Here the user specifies only what processing they wish to perform, and upon what data. The CI then handles everything from the creation of a schedulable entity to the receipt of results. This process can also be further abstracted by hiding this definition stage, for example behind approaches such as polymorphic overloading of Matlab functions (see: Section 6.10.4). **(5)** The integration within an application environment, as demonstrated in Chapter 6. **(6)** The processes required for Task-based fault tolerance, i.e. the ability to differentiate between faults that occur due to errors in application code and those which occur due to an inconsistency in the operating environment. This is provided by the distinction between resource acquisition, allocation and scheduling, i.e. the CRM (Section 5.3), CSM (Section 5.4) and CI (Section 5.5) respectively. In addition, the provisioning of a daemon (the IAS: Section 5.2) within an application environment enables the capture of such faults in a more useful fashion.

The second benefit is a much improved application latency, as demonstrated in Chapters 6 and 7. This reduction is caused by several key attributes of this approach. Firstly, that workstations are initialized only once per session, as opposed to once per Task, which was made possible through the introduction of the CRM (Section 5.3) and IAS (Section 5.2). This meant that each workstation could be used interactively for as long as it was needed, rather than for the potentially short runtime of a single Task. Consequently, it also meant that Tasks could be executed much nearer to what their normal execution time on a local workstation would be. Furthermore, it also contributed to workstations being used very efficiently, for example in Chapter 7 an efficiency of resource utilization of 88% was observed for one hundred Tasks with a runtime of only one second. This was an improvement of almost 13000% over the traditional Campus-Grid approach. Secondly, the generation of adaptive workstation profiles by the CRM (Section 5.3) prevented faulty workstations from becoming job blackholes, which, as demonstrated in Section 2.2 could significantly increase the number of job submissions needed for an application to complete. Finally, because the functionality captured by the Autonomic (Matlab) Compiler (Section 5.2.1) enables completely disparate application scenarios to be catered for, the Virtual Cluster does not need to be restarted or recompiled in order to facilitate a change in application scope.

The third benefit, which is actually a consequence of the second benefit discussed above, is that performance could be improved whilst at the same time the number of workstations in use could be reduced. The reasons for this are the same as those presented for the second benefit: namely, Task runtimes being comparable to local execution, efficiency of resource utilization being very good, workstations used for as long as they are needed in an interactive fashion and job blackholes being avoided through the use of proactive pilot tests and adaptive blacklisting. This therefore benefits not only the direct user of the Campus-Grid, but other users too, as more resources remain available for them to use.

When these three benefits are combined together, the final outcome is that the user is able to rapidly deploy their applications into a parallel execution scenario, and receive their results in a timely manner. This can be achieved regardless of whether their Task granularity relates to Task runtimes of a few seconds in length or several minutes. Ultimately, this means that new research hypotheses can be rapidly tested and developed by non computer scientists, which is exactly what infrastructures such as Campus-Grids are intended for. This can be achieved despite the challenges outlined in Section 1.3 and the poor performance observed in Chapter 2.

8.1.4 Continuum of Capability

Question 4: What are the bounds of these models?

The target Tasks for this research are those with very short runtimes (seconds). For this reason, the experiments conducted in Chapter 7 were orientated toward determining when a Task is too short to feasibly be executed using the system documented in Chapter 5. It was found that when the Virtual Cluster must also be constructed, the lower limit of feasibility is in the order of five seconds in length. However, note that this was also for a small set of Tasks – 100. Therefore, if the number of Tasks was increased, the lower limit will decrease proportionally, as the cost of building the Virtual Cluster will have a reduced effect on performance. For this reason, further analysis could determine a more precise value between one and five seconds, for various application sizes. However, this is not a critical issue. The key point instead, is that even for Tasks of as little as five seconds in length, and when the number of Tasks to be performed was small, significant reductions in application makespan were observed. If multiple sets of Tasks are to be executed, then the approach can easily cater for Tasks of as little as one second in length, simply because the Virtual Cluster can be recycled without being redeployed or reinitialized. In Section 7.2 an example of running longer Tasks was also given. Here, even ten minute Tasks received an improved performance.

What this means in relation to the continuum of capability that this approach has, is that it can facilitate very short Tasks of as little as one second in length and still provide real improvements in application makespan. For longer running Tasks the Virtual Cluster is bound by the same conditions as regular Campus-Grid middleware, i.e. workstation autonomy, or, in other words, the consequences of volunteered resources. However, it will never give reduced performance compared to a more traditional Campus-Grid middleware, and further experimentation would demonstrate this, but it will be bound to the general trends of availability that are experienced within the Campus-Grid as a whole. In fact, as Task runtime increased, the performance of both systems would converge. The reason for this is that as Task runtime increases, the probability that each workstation in the Virtual Cluster can perform more than one Task before its voluntary status is revoked will tend toward zero.

Further experimentation would give a deeper answer to this question. Of particular interest would be an investigation with data intensive applications. Here, some new models or modifications to those presented in this thesis would need to be introduced, as already discussed in Section 6.10.4.

8.2 Critical Evaluation of the Research Hypothesis

In Section 1.4 the research hypothesis for this thesis was presented. The core part of the hypothesis is repeated below, to remind the reader of what it entailed:

> The hypothesis of this research is that a Campus-Grid can be made reliable, efficient and stable even for very short Tasks[4] (seconds), if the conceptual models of job and resource management are changed to directly consider the volatile nature of the resources in use.

The research conducted in this thesis and the approach documented in Chapters 4 and 5 have tested this hypothesis to the point where it is possible to say that it does indeed hold. The four research questions posed in Section 1.5 have enabled the ability to make this statement by focusing on three areas in which the hypothesis could be tested; namely: feasibility, benefit to the user and the continuum of capability (research questions 1, 3 and 4 respectively).

The first reseach question (discussed in Section 8.1.1) encapsulated the main crux of how an evaluation of the hypothesis could be approached. Therefore, it identified the core and minimum requirements needed by an approach in order to establish a useable prototype implementation, which could be evaluated and experimented with to test the hypothesis. The outcome of this research question demonstrated that a Campus-Grid infrastructure can be made both stable and reliable. This was despite the questions relating to reliability and performance that were raised in Chapter 2.

[4]Originally, in Section 1.4 the word 'job' was used here, but to remain consistent with the terminology introduced in Chapter 4, it has been replaced with 'Task'.

As a single research entity, this would not have been sufficient to enable an informed discussion of the hypothesis and what it entails. For this reason the third and fourth questions were also raised and were discussed in Sections 8.1.3 and 8.1.4. The research needed to answer these two questions required that the solution created for the first question could be employed within a range of application scenarios. This was performed for the simple reason that a system which is both reliable and stable, but with no utility has no purpose. In order to demonstrate utility, these two questions were posed to quantify utility. The performance metrics in use were: **(1)** the benefit to the user of the approach, and, **(2)** the continuum of capability that the approach has.

Using these two high level performance metrics, it could be identified that the benefit of the approach to the user is threefold: a significant increase in transparency, a significant reduction in application latency, and significant improvements in the efficiency of resource utilization, which translated into the ability to reduce the number of resources and still receive significant improvements in performance. However, benefit did not remain only with the direct user, but also transpires, unbeknown to them, to other Campus-Grid users. This occurs because fewer resources are now needed to accomplish better performance, which leaves more resources available to other Campus-Grid users. The analysis to determine where the borders on capability lie demonstrated that the approach can indeed cater for Tasks which are very short in length, where short relates to seconds in length. In addition, it also identified that this can be achieved feasibly and in an efficient manner.

The second research question, which has not yet been discussed in this Section, was not wholly focused on the hypothesis in an explicit manner. Instead, it looked at an underlying and implicit concept, which, rather than being essential to the evaluation of the hypothesis, fulfils a more auxillery purpose. Here, the question revolves around how a "number" of resources to be used for a given application can be determined. The idea here is that by attempting to make this process transparent, the approach can be, as a whole, more opportunistic in its core functionality. Therefore this question was really looking at how the application itself can be made opportunistic, and therefore capture, and react to, the dynamic nature of the resources in use. By making the necessary abstractions in the conceptual model to facilitate this ability, the approach as a whole also became more transparent and autonomic, which enables non computer scientists to focus on their research rather than on administration that would inhibit their research.

It is due to the answers of these four research questions as well as the work and research undertaken in order discuss their answers, that the evaluation of the hypothesis has led to the following informed opinion: When the changes to the job and resource conceptual models of a Campus-Grid, as defined in Chapter 4 are made, and this is performed with respect to the issues of volatility and unreliability that were outlined in Section 1.3 and Chapter 2, it is possible to receive excellent performance even for Tasks which are very short in length (seconds). Such Tasks may also have one or more of the other attributes identified in Section 1.3.9.

Naturally, there are questions that can be raised against the research conducted in this thesis. Perhaps the most obvious question is: Is the infinite agenda of the daemon model (Section 4.3.1) breaking the etiquette of Campus-Grids, opportunistic and volunteer computing and the general use case of systems such as Condor? The short answer to this question is "yes, but", where the "but" is crucially important to shaking off such concerns. It captures issues such as the measured improvement in the four performance metrics (introduced in Section 2.4.1 and used to evaluate the system in Chapter 7) being so significant that 100 sixty second Tasks can now be completed faster than 100 one second Tasks using a pure Campus-Grid approach. A counter argument here would be that the IAS daemons can sit idle for undisclosed periods of time. However, whilst this is true, the daemon is still subject to the same rule set and constraints as a normal Campus-Grid job, i.e. it can be suspended and evicted from the workstation when the voluntary status of the workstation is revoked. The key difference is that where a normal Campus-Grid job experienced terrible performance, the daemon model enables excellent performance. Workstations in use to were performing useful work for up to 88% of the time an application uses them, for Tasks of as little as one second in length. When compared to a normal use case, Condor for example, the time spent performing useful work was less than 1% for exactly the same set of Tasks. Therefore, in answer to this question, it is possible to raise the following question: which of these two approaches is more wasteful of resources? The work conducted in this thesis provides the informed basis to state that it is the traditional Campus-Grid approach which is far more wasteful of resources. For this reason, it is hard to justify that the application of the daemon model goes against the common use case etiquette, as it actively benefits the direct user of the Virtual Cluster as well as other users of the Campus-Grid by requiring fewer resources for the same analysis.

A similar question which can be raised is: why should a researcher wish to perform short Tasks, as by their very definition, they are short running? This question is actually quite simple to answer, and it can also be answered with a question: what if they have thousands of Tasks, and not hundreds as have been experimented with here in this thesis? In such a scenario, the utility of this work soon becomes appealing. Firstly, as the user need not be concerned with how they partition their Tasks into larger clusters. This is handled transparently by the CI (Section 5.5), and will remove the need for the user to consider what a sensible cluster size is. Secondly, how and when the Tasks are submitted to the Campus-Grid will not become an issue. This is because the Tasks are held in memory by the CI, rather than be enterred into the Campus-Grid as entities that need to be matched to workstations. In Section 2.2 the consequences of such an action was demonstrated. Finally, short Tasks require a quick turnaround, and this approach has been specifically conceived for such a purpose. The benefit that the use of a Virtual Cluster has for interactively produced Tasks is that the overhead experienced from the batch queue is only a one off cost because once resources have been allocated they are dedicated to that user for the full duration that they are available or required.

8.3 Future Work

The models and prototype that have been studied in this thesis have been possible for only one reason: namely, that the resources were sourced in-house, i.e. the Campus-Grid workstations fall within the accessible domain. Therefore, the question that will be discussed in this Section is:

what protective measures would be in place if the performance observed in Chapter 2 had been delivered when such resources were outsourced? This will lead into a discussion concerning how a Campus-Grid could be used to provision Cloud services at the lowest level, i.e. Task submission and management.

8.3.1 Current Industry Approach

Naturally, the first place to look for consumable computational resources is to identify what Industry services are available. Here, Amazon's Elastic Compute Cloud (EC2)[5] is probably the forerunner in this domain. In order to provide some "peace of mind" to the consumer many providers offer a Service Level Agreement (SLA), which, under their conceptual model, can be viewed as a legal contract or terms of service. The purpose of the SLA is to define the: what, how, cost, guarantees and conditions of the service provision (execution of Tasks in this case). Typically, the SLA is simple in definition, human readable, and contains little legal jargon to aid in its understanding and readability.

The problems with the industry options soon arise when the contents of the agreement are reviewed, as well as the way in which they are negotiated. The primary example here is that the pricing mechanisms are static, and therefore not negotiable. Typically, different price tiers relate to a better quality of service and more stringent guarantees. The levels of quality of service and the guarantees that the SLA contain are commonly quite sensible; example terms are: percentage uptime, maximum response time for a request etc. In addition, should any of the terms be violated a "compensation" will be given to the consumer, which can be modelled as a provider penalty. For example Amazon offer up to 25% off the next bill, Google offer up to 15 days free usage after the billing period has ended. Note that these guarantees are also static and not negotiable. On the outset, the guarantees look very attractive, until the exclusion list is considered. Here, providers list a set of conditions that are acceptable conditions under which the guarantees are not applicable; for example: scheduled maintenance, network/power outages, hardware failures and software trouble. The problem here is that once this list is considered in its entirety, it becomes very difficult for the provider to violate the SLA. In addition, it is also common to see a limited liability clause, which will prevent the consumer making legally grounded claims for loss of earnings or other damages should the SLA be violated. If one of the few scenarios should arise where the SLA is violated, there are no active enforcement measures to capture the violation. Instead, it is the consumer that must take reactive action. However, this may not be a straightforward procedure, consider this extract from the Amazon SLA:

> [...] **you** must submit a request by sending an e-mail message [...] include, in the body of the e-mail, the dates and times of each incident of non-zero Error Rates that **you claim** to have experienced [...] received by us within ten (10) business days after the end of the billing cycle in which the errors occurred [...] Your failure to provide the

[5]`http://aws.amazon.com/ec2/` – last accessed August 2009

request and other information as required above will **disqualify** you [...]

Note that this is not uncommon in other industry SLAs, Google for example have a very similar clause. Ultimately, it means that there is little or no introspective regulation performed by the provider to ensure that the promises made in the SLA are adhered to. In addition, most providers, when a violation is reported in this way, offer no more than service credits (free time). The problem here is that what if the consumer: **(1)** No longer wishes to use this provider? **(2)** Has reached the end of the contractual period? **(3)** Has lost profit through the violation of greater value than that of the service credits? **(4)** Does not receive enough credits to rectify the loss in useable results? In these cases service credits have no utility, as the consumer is still loosing out.

8.3.2 What Could be Different?

There are several ways in which this could be different, and this is portrayed in the research projects currently investigating this area. Essentially, there are four key areas of investigation: **(1)** Infrastructures that enable automated discovery of consumer/provider couplets. **(2)** The automated definition of machine readable SLAs, using "well formed" term languages. **(3)** Matchmaking or negotiation strategies that enable the contents of the SLA to be more customizable and tailored to what the provider can offer, and what the consumer actually wants. **(4)** The necessary mechanisms for SLA enforcement, i.e. automated violation detection, with the potential to also apply a "penalty".

There are two common trends for the discovery of both consumers and providers: **(1)** auction-based market mechanisms, and **(2)** brokering systems. Market mechanisms encourage competition, provide choice, enable preference elicitation, and can also increase the transparency and flexibility of discovery. Brokerage systems enable a consumer or provider to outsource the effort required to find a participant to trade with. Some example EU projects in both of these areas are: SORMA,[6] BREIN,[7] AssessGrid[8] and SLA@SOI.[9]

There are many approaches for the definition of SLAs in the literature. One example is the Contract Net Protocol [229, 230], which was one of the first protocols for negotiating electronic service contracts. Other languages for specifying SLAs include: WSLA [231], SLAng [232], WS-Agreement [233] and RBSLA [234]. In addition, many of these approaches define a form of negotiation protocol. However, note that such protocols are often not full bi-lateral protocols, i.e. propose/contra-propose, and at best are uni-lateral strategies where instead of an outright reject an invitation to treat is offered.[10] Of these approaches WS-Agreement is the most commonly adopted

[6]SORMA - Self-Organizing ICT Resource Management, `http://www.sorma-project.org` – last accessed August 2009.

[7]BREIN - Business objective driven reliable and intelligent grids for real business, `http://www.eu-brein.com` – last accessed August 2009.

[8]Advanced Risk Assessment & Management for Trustable Grids, `http://www.assessgrid.eu` – last accessed August 2009.

[9]Empowering the service industry with SLA-aware infrastructures, `http://sla-at-soi.eu` – last accessed August 2009.

[10]Here, an invitation to treat relates not to a full contra-propose, but a more loose concept such as: were an offer of X be made, it is possible that it would be accepted.

in current FP6 and 7 EU projects, as well as the research community as a whole. A discussion of SLA-related work in some of these projects can be found in [235], and a good general overview of the work in this area can be found in [236]. The research undertaken in matchmaking is well established; Condor, for example utilizes matchmaking algorithms. In relation to specific matchmaking strategies for SLAs, SWAPS [237] is perhaps one of the most interesting, as it considers the semantic meaning behind the concepts captured in the SLA itself. One aspect in the research of SLAs that is somewhat sparse is in the area of economic term languages for negotiation or economic matchmaking, where EJSDL (Economic JSDL) [238] is the only known example. This is specifically true in the area of penalty definition.

The research in SLA Enforcement raises many difficult questions and is associated with many outstanding challenges. These include: legal issues surrounding what constitutes enforcement and what actions can legally be undertaken, gathering unbiased and accurate data for violation detection, defining the needed protocols for the transfer of enforcement data and the associated actions, and conflict resolution, for example when none of the constitutional terms of the SLA have been violated, but one or more Tasks have failed. Many of these challenges can also be related to, and are symptomatic of, the term languages that are used to express the contents of the SLA. Some areas which are enforcement-related but yet to be pursued include: concepts around insurance against the consequences of violation, term languages that can be used to express legal issues and constraints that define when or how a SLA can be terminated or considered void.

There are essentially four different approaches which can be taken to SLA Enforcement: **(1)** An enforcement component is run on the provider's resources, which is trusted by the consumer. **(2)** An enforcement component is run on the consumer's resource, which is trusted by the provider. **(3)** An enforcement component is run by a third party, which is trusted by both the consumer and the provider. **(4)** A hybrid of the previous three, as impartiality has its benefits but it is not universal in scope, and therefore requires data from both participants of the SLA. In this case, all three entities, i.e. the consumer, the provider and the third party must trust each other. Trust is a difficult concept to fully capture here, but must be an underlying assumption in such systems, as there are no means to ensure that all the provided data is accurate. It is yet to be seen whether the need for this assumption undermines research in this area. Some possible solutions to this problem could lie in the use of pilot applications[11] and reputation systems. However, both can be easily compromised.

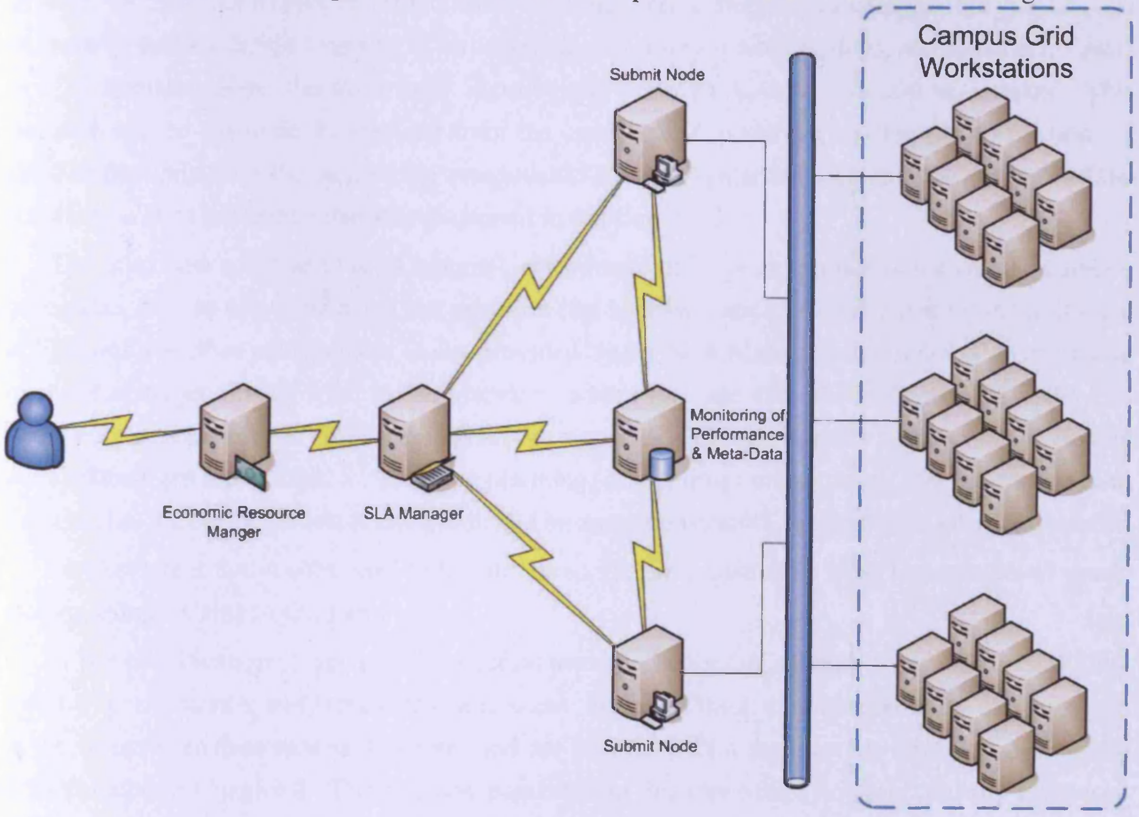
The objective of this Section has been to illustrate that there is a lot of work in this area already underway, and that the area is one that is attracting interest. In the next Section the discussion will move to how the research discussed in this Section can be brought together for a new application area using the research reported in this thesis.

[11]Here, a pilot application relates to a similar concept that surrounds the IAS (Section 5.2) pilot test. An example pilot application could be the deployment of a standalone monitoring system by an enforcement component to validate whether the enforcement data it is receiving from a provider is accurate. In contrast, it could also be the local execution of a consumer's Task or set of Tasks, by an enforcement component. Such an approach could be employed to determine whether a consumer's Tasks are in fact correctly compiled or error prone.

8.3.3 Campus-Grid Cloud Provisioning

Given that research into SLA-aware architectures are emerging but still in their infancy, and the recent advent of Cloud Computing where "services" are provided, typically for some economic gain, there is plenty of scope for Universities and other research institutions to sell spare capacity within a Cloud scenario. Campus-Grids are well established and typically contain the necessary accounting and security infrastructures. Therefore the main body of work required to enter into this area is the adoption of the necessary technologies that were briefly introduced in the previous Section. Therefore, the question arises: Can a Campus-Grid be used in Cloud provisioning? Whilst this question actually relates to many different strands of provisioning such as "Software as a Service", "Platform as a Service" and the many other "as a Service" derivatives, the main focus in this discussion will consider only the lowest level of provisioning, i.e. Job Provisioning, as this is the most closely related area to the research conducted in this thesis.

Figure 8.1: An example infrastructure for Campus-Grid-based Cloud Provisioning.



The general approach which could be taken to achieve such a provisioning scenario is depicted in Figure 8.1. Here, there are several additional components to a typical Campus-Grid infrastructure. Firstly, there are some components defined as a part of the prototype documented in Chapter 5, which have been simply renamed. For example the Monitoring of Performance and Meta-Data, could be performed in a similar way in which the CRM (Section 5.3) acted when considering advertisements of Matlab capability. In addition, a component similar to the CRM can be placed on the submit nodes and can orchestrate the provisioning through a means similar to the construction of the Virtual Cluster. Here, because of the SLA context, issues which affect performance are of great importance, as poor performance can lead to the application of penalties and/or a reduction in reputation. It is here that the research undertaken in this thesis is of the greatest benefit. This is the case because issues such as reliability, instability and poor performance were directly addressed. In addition, many of the solutions presented in this thesis can be simply mapped to a wider scope of Tasks and application scenarios. Multiple submit nodes are present in Figure 8.1 to counter the problems observed with respect to queue size in Section 2.2.

The second new addition to the Campus-Grid infrastructure is a SLA Manager component. Here, the current portfolio of SLAs that are currently active are managed. Management of a SLA portfolio, relates to more than just the monitoring of the currently active SLAs to avoid and minimize any violation-based penalties. Here aspects such as risk, capacity planning in relation to the current portfolio and future SLAs, as well as the identification of spare capacity all need to be performed as well. In this context risk can relate to several different concepts, for example the risk of: one or more violations should a new SLA be accepted, autonomy-related failures, malicious consumers or elicit conduct. Here, the main focus is on whether new SLAs can or should be accepted. This decision will be based on the demand from the current SLA portfolio, and the observed status of the Campus-Grid from the monitoring component. This could also be augmented using some of the research on state prediction that was discussed in Section 3.2.3.

The third new addition to the Campus-Grid infrastructure is an Economic Resource Manager, here issues such as negotiation and bid injection (for auctions) are managed, based upon the current obligations and other management issues provided by the SLA Manager. Examples of an economic resource manager already exist in the literature, where just one example is the Economically Enhanced Resource Manager from the SORMA project [98]. The key aspects that such a component must address are issues such as: strategic planning, competition management and revenue generation etc., i.e. the consideration of economic and business constraints, in addition to job requirements.

The key areas that would need to be addressed and investigated in order to begin cloud provisioning using a Campus-Grid are:

1. The classification of, and proactive action toward, changes in: reliability, software availability, stability, performance, and workstation autonomy. Many of these characteristics are very dynamic in the resource environment as a whole, and are prolific within the Campus-Grid domain, as was demonstrated in Chapter 2. The research presented in this thesis has, however, already addressed many of these challenges. This work could therefore be taken forward and extended for a Cloud provisioning scenario.

2. The inclusion of a legal context into the SLA. However, it is most likely that such a legal foundation will be instigated through the use of a pre-existent paper contract, upon which the SLA can be based. Issues of legal directives and other issues in contract law can then be handled in a more formal and familiar manner.

3. The identification and dissemination of risk alongside SLA-aware risk assessment and management. This risk may also be associated to some legal context of the SLA.

4. Either the methodology for the injection of offers into a market mechanism, or for the negotiation of the constitutional terms of the SLA. Here there are many challenges, not only in the scope of the representation of the offers made by potential participants, but also in the negotiation and matchmaking strategies in use. For example, should semantics be included, as in [237], should participant couplets be identified using functional means, as in [236], do matches need to be fully inclusive or can partial matches also be considered? These are just a few scenarios, and the challenges in these areas are significant. For example, the ontologies needed for semantic matchmaking must consider many different possibilities, which can be computationally expensive. Similarly, functional approaches can also be expensive should the use high order polynomial functions be adopted. In addition, they may also not be a one-way function or in fact ever resolve. The challenge in partial matchmaking is enabling this level of decision making autonomously, but also to capture the preferences of the participants accurately enough for these decisions to be made. In addition, it is common that neither participant knows their exact preferences. However, they may have beliefs of baseline estimations, which can be iteratively improved using selected methodologies of preference elicitation [239].

5. The SLA term language could prove to be an issue. SLA specification languages such as WS-Agreement and WSLA envelope one or more other term languages such as JSDL [127] in order to describe the service to be provided. However, when using general specifications for service description, especially job description, some of the inherent content of the original specification can be semantically lost, as was previously mentioned in the discussion of the Globus Toolkit in Section 3.2.1. In addition, general specification languages cannot always capture the type of an input parameter which can have other side-effects, as previously mentioned in the discussion of DIPS in Section 3.1.2.

6. The methodology for the enforcement of the SLAs will also need to be considered. However, this is more of a higher level architectural question rather than one specific only to an approach for the provisioning of Cloud services using a Campus-Grid. Here, issues such as data acquisition and validation are key. In addition, the challenge of trust remains outstanding. There are also currently no documented enforcement protocols in the literature.

In conclusion, it is likely that a Campus-Grid can be used in Cloud provisioning, however, first there are many interesting and challenging research questions to be addressed. It has been the intention of this discussion to introduce some of these questions in relation to, and based upon, the work which has been conducted in this thesis to illustrate a new research direction that could be developed in the future.

Bibliography

[1] Sudarshan Raghunathan. Making a supercompter do what you want high-level tools for parallel programming. *Computing in science and engineering*, pages 72 – 83, 2006.

[2] Ewa Deelman, Tevfik Kosar, Carl Kesselman, and Miron Livny. What makes workflows work in an opportunistic environment? *Concurrency and Computation.: Practice and Experience.*, 18:1187 – 1199, 2006.

[3] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[4] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45 (11):56 – 61, 2002.

[5] Matt W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Transactions on Software Engineering*, VOL. 18, NO. 4:319 – 328, 1992.

[6] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software Practice & Experience*, 21, 8:757 – 785, 1991.

[7] A. Acharya et al. The utility of exploiting idle workstations for parallel computation. In *Proceedings of ACM International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS '97)*, 1997.

[8] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. Integrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency and Computation.: Practice and Experience.*, 16:449 – 459, 2004.

[9] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63:579 – 592, 2003.

[10] J. Osborne and A. Hardisty. Cardiff University's Condor Pool: Background, Case Studies, and fEC. pages 361–364, Nottingham, UK, September 18th to September 21st 2006.

[11] James E. Dobson, Jeffrey B. Woodward, Susan A. Schwarz, John C. Marchesini, Hany Farid, and Sean W. Smith. The dartmouth green grid. *V.S. Sunderam et al. (Eds.): ICCS 2005*, LNCS 3515:99 - 106, 2005.

[12] Condor project website. http://www.cs.wisc.edu/condor/ – last visited August 2009.

[13] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[14] Condor@cardiff. http://www.cardiff.ac.uk/arcca/condor – last visited August 2009.

[15] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .net-based enterprise grid computing system. In *International Conference on Internet Computing*, pages 269 – 278, 2005.

[16] Vijay K. Naik, Swaminathan Sivasubramanian, David Bantz, and Sriram Krishnan. Harmony: A desktop grid for delivering enterprise computations. In *Proceedings of the Fourth International Workshop on Grid Computing (GRID03)*, 2003.

[17] D.C.H. Wallom and A.E. Trefethen. Oxgrid, a Campus Grid for the University of Oxford. In *Proceedings of UK e-Science Conference, Nottingham, UK*, 2006.

[18] David Wallom, Ian Stewart, and Jon Wakelin. The university of bristol grid and a production campus-grid. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing - HPDC-14.*, 2005.

[19] M. Humphrey and G. Wasson. The University of Virginia Campus Grid: Integrating Grid Technologies with the Campus Information Infrastructure. *Advances in Grid Computing – EGC 2005*, LNCS 3470:50–58, 2005.

[20] A Todd-Pokropek. Advanced image processing in radiology. *Imaging*, 2002:478 – 484, 14.

[21] Shannon Hastings, Tahsin Kurc, Stephen Langella, Umit Catalyurek, Tony Pan, and Joel Saltz. Image processing for the grid: A toolkit for building grid-enabled image processing applications. In *Proceedings of the 3rd IEEE/ACM International Symposium on ClusterComputing and the Grid (CCGRID03)*, pages 36 – 43, 2003.

[22] Eugene Borovikov, Alan Sussman, and Larry Davis. A high performance multi-perspective vision studio. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 348 – 357, 2003.

[23] Alois Goller and Franz Leberl. Radar image processing with clusters of computers. In *Aerospace Conference Proceedings, IEEE*, volume 3, pages 281 – 285, 2000.

[24] Dana Petcu and Victoria Iordan. Grid service based on GIMP for processing remote sensing images. In *Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'06)*, pages 251 – 258, 2006.

[25] Ian Foster. What is the grid? a three point checklist. http://www.gridtoday.com/02/0722/100136.html.

[26] I. Foster and C. Kesselman. *In The Grid: Blueprint for a Future Computing Infrastructure*, chapter Globus: A Toolkit-Based Grid Architecture, pages 259–278. Morgan Kaufmann Publishers, 1999.

[27] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7 – 9, 2001.

[28] Sudesh Agrawal, Jack Dongarra, Keith Seymour, and Sathish Vadhiyar. *Grid Computing - Making the Global Infrastructure a Reality*, chapter NetSolve: past, present, and future; a look at a Grid enabled server, pages 613 – 622. Wiley Publishing, 2003.

[29] Andreas Uhl. Parallel and distributed processing of visual content: Traditional view and new directions. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euromicro-PDP05)*, page 2, 2005.

[30] Raihan Ur Rasool and Guo Qingping. Users-grid matlab plug-in: Enabling matlab for the grid. In *IEEE International Conference on e-Business Engineering (ICEBE'06)*, pages 473 – 478, 2006.

[31] Pawel Czarnul, Andrzej Ciereszko, and Marcin Fraczak. Towards efficient parallel image processing on cluster grids using GIMP. *M. Bubak et al. (Eds.): ICCS 2004*, LNCS 3037:451 – 458, 2004.

[32] D. Abramson, R. Sosic, J. Giddy, and M. Cope. The laboratory bench: Distributed computing for parametised simulations. In *Proceedings of Parallel Computing and Transputers Conference*, pages 17 – 27, 1994.

[33] Simon Caton, Omer Rana, and Bruce Batchelor. Dynamic condor-based services for distributed image analysis. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 49 – 56, 2007.

[34] Derrick Kondo, Michela Taufer, Charles L. Brooks III, Henri Casanova, and Andrew A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[35] SungJin Choi, MaengSoon Baik, ChongSun Hwang, JoonMin Gil, and HeonChang Yu. Volunteer availability based fault tolerant scheduling mechanism in desktop grid computing environment. In *Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA04)*, 2004.

[36] Derrick Kondo, Andrew A. Chien, and Henri Casanova. Resource management for rapid application turnaround. In *Supercomputing*, 2004.

[37] Rajkumar Buyya, Manzur Murshed, David Abramson, and Srikumar Venugopal. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Software - Practice And Experience*, 35:491 – 512, 2005.

[38] Simon Caton, Omer Rana, and Bruce Batchelor. Distributed image processing over an adaptive campus-grid. *Concurrency and Computation: Practice and Experience*, 21 Issue 3:321 – 336, 2008.

[39] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17:323 – 359, 2005.

[40] Haresh S. Bhatt, V. H. Patel, and A. K. Aggarwal. Web enabled client-server model for development environment of distributed image processing. *GRID 2000 R. Buyya and M. Baker (Eds.:)*, LNCS 1971:135 - 145, 2000.

[41] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser. Using multiple feedback loops for object profiling, scheduling and migration in soft real-time distributed object systems. In *Proceedings if the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 1999. (ISORC '99)*, pages 291 – 300, 1999.

[42] Cesar A.F. De Rose, Hans-Ulrich Heiss, and Barry Linnert. Distributed dynamic processor allocation for multicomputers. *Parallel Computing*, 33:145 – 158, 2007.

[43] Cherri M. Pancake and Curtis Cook. What users need in parallel tool support: Survey results and analysis. In *Proceedings of the IEEE Scalable High-Performance Computing Conference*, pages 40 – 47, 1994.

[44] Ron Choy and Alan Edelman. Parallel Matlab: Doing it right. *Proceedings of the IEEE*, VOL. 93, NO. 2:331 - 341, 2005.

[45] Asim YarKhan, Keith Seymour, Kiran Sagi, Zhiao Shi, and Jack Dongarra. Recent developments in GridSolve. *International Journal of High Performance Computing Applications*, Vol.20 No.1:131 - 142, 2006.

[46] Wolfgang Blochinger, Wolfgang Westje, Wolfgang Kuchlin, and Sebastian Wedeniwski. ZetaSAT boolean satisfiability solving on desktop grids. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 1079 - 1086, 2005.

[47] Franz Niederl and Alois Goller. Method execution on a distributed image processing backend. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing. PDP'98.*, pages 243 – 249, 1998.

[48] Grid5000 project website. http://www.grid5000.fr/ – last accessed April 2009.

[49] http://www.cs.vu.nl/das3/ – last accessed April 2009.

[50] Ian Foster and Carl Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115 – 128, 1997.

[51] R Buyya, D Abramson, and J. Giddy. Nimrod/g: an architecture for a resource management and schedulingsystem in a global computational grid. In *The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region.*, pages 283 – 289, 2000.

[52] Srikumar Venugopal, Rajkumar Buyya, and Lyle Winton. A grid service broker for scheduling distributed data-oriented applications on global grids. In *Proceedings of the 2nd workshop on Middleware for grid computing*, pages 78 – 80, 2004.

[53] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The complete reference*. The MIT Press, 1995.

[54] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5 (1):46 – 55, 1998.

[55] Cleve Moler. Why there isn't a parallel Matlab. http://www.mathworks.com/company/newsletters/news_notes/pdf/spr95cleve.pdf.

[56] Parry Husbands, Charles L. Isbell, and Alan Edelman. Interactive supercomputing with MIT-Matlab. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing (P99)*, 1999.

[57] Roy Friedman and Erez Hadad. Distributed wisdom: Analyzing distributed-system performance latency vs. throughput. *IEEE Distributed Systems Online*, 7 (1), 2006.

[58] Frederick M. Waltz. Separated-kernel image processing using finite-state machines (skipsm). In *Machine Vision Applications, Architectures, and Systems Integration III*, pages 383 – 395, 1994.

[59] Simon Caton, Matthan Caan, Sílvia Olabarriaga, Omer Rana, and Bruce Batchelor. Using dynamic condor-based services for classifying schizophrenia in diffusion tensor images. In *CCGRID '08: Proceedings of the Eigth IEEE International Symposium on Cluster Computing and the Grid*, pages 234 – 241, 2008.

[60] Octave. http://www.gnu.org/software/octave/ – last Accessed April 2009.

[61] GIMP - GNU Image Manipulation Program. http://www.gimp.org/ – last Accessed April 2009.

[62] LabVIEW. http://www.ni.com/labview/ – last Accessed April 2009.

[63] Rajesh Raman, Miron Livny, and Marvin Solomon. Resource management through multilateral matchmaking. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 290 – 291, 2000.

[64] Daniel Bunford-Jones, Omer F. Rana, David W. Walker, Matthew Addis, Mike Surridge, and Ken Hawick. Resource discovery for dynamic clusters in computational grids. In *Proceedings of The 15th International Parallel and Distributed Processing Symposium.*, pages 759 – 767, 2001.

[65] Adam Lathers, Mei-Hui Su, Alex Kulungowski, Abel W. Lin, Gaurang Mehta, Steven T. Peltier, Ewa Deelman, and Mark H. Ellisman. Enabling parallel scientific applications with workflow tools. In *Challenges of Large Applications in Distributed Environments, IEEE*, pages 55 – 60, 2006.

[66] Jiadao Li, Oliver Waldrich, and Wolfgang Ziegler. Towards sla-based software licenses and license management in grid computing. In *CoreGRID Symposium, Las Palmas de Gran Canaria, Gran Canaria, Spain, August 2008, Springer, 2008, CoreGRID Series*, 2008.

[67] E.G.P. Bovenkamp, J. Dijkstra, J.G. Bosch, and J.H.C. Reiber. Mulit-agent segmentation of IVUS images. *The Journal of the Pattern Recognition Society*, 37(4):647 – 663, 2004.

[68] Andrew S. Grimshaw, Wm. A. Wulf, and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, Vol. 40, No. 1:39 – 45, January 1997.

[69] Jiansheng Huang, Ameet Kini, Erik Paulson, Christine Reilly, Eric Robinson, Srinath Shankar, Lakshmikant Shrinivas, and David DeWitt. An overview of quill: A passive operational data logging system for condor. Technical report, School of Computer Science, University of Wisconsin, 2007.

[70] Condor hawkeye. http://www.cs.wisc.edu/condor/hawkeye/ – last accessed June 2009.

[71] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[72] http://www.vmware.com (Last accessed Feb 2009).

[73] Rajkumar Buyya and M. Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *CoRR*, cs.DC/0203019, 2002.

[74] B S Garbow, J M Boyle, J J Dongarra, and C B Moler. *Matrix Eigensystem Routines – Eispack Guide Extension.* Springer-Verlag, 1977.

[75] J J Dongarra, J R Bunch, C B Moler, and G W Stewart. *Linpack Users' Guide.* SIAM, 1990.

[76] Albert I. Reuther, Tim Currie, Jeremy Kepner, Hahn G. Kim, Andrew McCabe, Peter Michaleas, and Nadya Travinin. Technology requirements for supporting on-demand interactive grid. In *Proceedings of the Users Group Conference (DOD-UGC'05)*, 2005.

[77] http://www.scilab.org/ (Last accessed Feb 2009).

[78] Mark Wilkinson. Using matlab and octave with condor: a case study. Technical report, 2005.

[79] S. Steinhaus. Comparison of methematical programs for data analysis. Available online at: http://www.scientificweb.de/ncrunch - last visited Feb 2009.

[80] Parry Husbands and Charles Isbell. The parallel problems server: A client-server model for interactive large scale scientific computation. *J.Palma and J.Dongarra and V.Hernandez(Eds.):VECPAR98*, LNCS 1573:156 – 169, 1999.

[81] Vijay Menon and Anne E. Trefethen. MultiMatlab: Integrating Matlab with high-performance parallel computing. In *Supercomputing*, pages 30 – 47, 1997.

[82] Elias S. Manolakos, Demetris G. Galatopoullos, and Andrew P. Funk. Distributed Matlab based signal and image processing using Javaports. In *Proceedings of IEEE International Conference on Acoustics, Speech,and Signal Processing.(ICASSP'04)*, pages 217 – 220, 2004.

[83] L. S. Blackford, A. Cleary, J. Choi, E. D'Azevedo, J. Demmel, Society for Industrial, Applied Mathematics, I. Dhillon, J. Dongarra, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, 1997.

[84] Parry Husbands and Charles L. Isbell. Interactive supercomputing with Matlab*P. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing (P99)*, 1999.

[85] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Cambridge University Press (UK)., 1996 (Revised in 2002).

[86] Long Yin Choy. Matlab*P 2.0: Interactive supercomputing made practical. Master's thesis, Massachusetts Institute of Technology, 2002.

[87] B Wong, M. Fadri, and S. Raman. Thermodynamic calculations for molecules with asymmetric internal rotors. ii. application to the 1,2-dihaloethanes. *Journal of Computational Chemistry*, Vol 29, No. 3:481 – 487, 2007.

[88] MolPro product website: http://www.molpro.net/ (last accessed June 2009).

[89] Mark D. Barnell and Brian J. Rahn. Migrating modeling and simulation applications on to high. In *SPIE Enabling Technologies for Simulation Science X Conference*, 2006.

[90] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, and David Cheng. Star-P: High productivity parallel computing. In *Workshop on High Performance Embedded Computing*, 2004.

[91] J. Kepner and N. Travinin. Parallel matlab: The next generation. In *7th High Performance Embedded Computing Workshop (HPEC)*, 2003.

[92] J. Kepner. Parallel programming with matlabmpi. In *High Performance Embedded Computing (HPEC 2001) Workshop*, 2001.

[93] Ying Chen and Suan Fong Tan. Matlab*G: A grid-based parallel Matlab. http://hdl.handle.net/1721.1/3863.

[94] Y. M. Teo and X. B. Wang. Alice: A scalable runtime infrastructure for high performance grid computing. In *In Proceedings of IFIP International Conference on Network and Parallel Computing, in Springer-Verlag, Lecture Notes in Computer Science Series 3222*, pages 101–109, 2004.

[95] Wilson Rivera, Carmen Carvajal, and Wilfredo Lugo. A service oriented architecture grid based environment for hyperspectral imaging analysis. *International Journal of Information Technology*, Vol. 11 No. 4:104 – 111, 2005.

[96] M. Li, P. van Santen, D.W.Walker, O.F.Rana, and M.A.Baker. Portallab: A web services toolkit for building semantic grid portals. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID03)*, pages 190 – 197, 2003.

[97] The Gridbus Project, University of Melbourne – http://www.gridbus.org, last accessed April 2009.

[98] T. Pueschel, N. Borissov, D. Neumann, M. Macias, and J. Guitart. Extended resource management using client classification and economic enhancements. In *eChallenges e-2007 Conference & Exhibition*, 2007.

[99] Ewa Deelman, Jim Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract workflows onto grid environments. *Journal of Grid Computing*, 1 (1), 2003.

[100] Ewa Deelman, Jim Blythe, Yolanda Gil, and Carl Kesselman. Workflow management in GridPhyN. *Grid Resource Management*, J. Nabryski, J. Schopf, and J. Weglarz (Eds), 2003.

[101] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C.

Jacob, and Daniel S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13 (3), 2005.

[102] James E. Dobson. Evaluation of biomedical applications on heterogeneous grid systems. http://dbic.dartmouth.edu/~jed/Papers/dobson.biomedical_grid.pdf – last Access August 2009.

[103] Sun Grid Engine website: http://www.sun.com/software/sge/ – last accessed August 2009.

[104] Raihan Ur Rasool and Qingping Guo. A pro-middleware for grids computing. *I. Stojmenovic et al. (Eds.): ISPA 2007*, LNCS 4742:556 – 562, 2007.

[105] R Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, R. Joisha, A. Jones, A. Kanhare A. Nayak, S. Periyacheri, M. Walkden, and D. Zuretsky. Computing systems. In *Field-Programmable Custom Computing Machines, IEEE Symposium on*, pages 39 – 48, 2000.

[106] The Geodise project. http://www.geodise.org/ - last visited Nov 2006.

[107] M. Hakki Eres, Graeme E. Pound, Zhouan Jiao, Jasmin L. Wason, Fenglian Xu, Andy J. Keane, and Simon J. Cox. Implementation and utilisation of a grid-enabled problem solving environment in matlab. *Future Generation Computer Systems*, Volume 21, Issue 6:920 – 929, 2005.

[108] Gang Xue, Matthew J. Fairman, Graeme E. Pound, and Simon J. Cox. Implementation of a grid computation toolkit for design optimisation with matlab and condor. *Euro-Par 2003 Parallel Processing*, LNCS 2790:357 – 365, 2003.

[109] Carl Q. Howard, Colin H. Hansen, and Anthony C. Zander. Optimisation of design and location of acoustic and vibration absorbers using a distributed computing network. In *Proceedings of Acoustics*, 2005.

[110] Duan H. Beckett, Ben Hiett, Ken S. Thomas, and Simon J. Cox. Applied grid computing: Optimisation of photonic devices. *Euro-Par 2003 Parallel Processing, LNCS*, 2970:533 – 536, 2003.

[111] J. Fritschy, L. Horesh, et al. Using the GRID to improve the computation speed of electrical impedance tomography (EIT) reconstruction algorithms. *Physiol. Meas.*, 26:209–215, 2005.

[112] Victor Zhorin and Tiberiu Stef-Praun. Grid-enabled estimation of structural economic models. Technical report, University Library of Munich, Germany in its series MPRA Paper with number 11384, 2008.

[113] George Teodoro, Tulio Tavares, Renato Ferreira, Tahsin Kurc, Wagner Meira Jr, Dorgival Guedes, Tony Pan, and Joel Saltz. A run-time system for efficient execution of scientific

workflows on distributed environments. *International Journal of Parallel Programming*, Volume 36, Number 2:250 – 266, 2008.

[114] R. Ferreira, W. Meira, D. Guedes Jr, D. Drummond, B. Coutinho, B. Teodoro, G. Tavares, T. Araujo, and G. Ferreira. Anthill: A scalable runtime environment for data mining applications. In *Symposium on Computer Architecture and High-Performance (SBAC-PAD)*, 2005.

[115] Martin Oberhuber. Distributed high-performance image processing on the internet. Master's thesis, Graz Techn. Univ., 1998.

[116] A. Goller. Parallel processing strategies for large sar image data sets in a distributed environment. *Computing*, 62:277 – 291, 1999.

[117] George Eckel, Jackie Neider, and Eleanor Bassler. *ImageVision Library Programming Guide*. Silicon Graphics, Inc., 1996.

[118] Image/J available from: http://rsbweb.nih.gov/ij/ (last accessed June 2009).

[119] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The apples parameter sweep template: User-level middleware for the grid. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[120] J.T. Moscicki. Distributed analysis environment for hep and interdisciplinary applications. In *Nuclear Ins. Methods Phys. Res. A 502, 426429 (2003)*.

[121] J.T. Moscicki, S. Guatelli, M. Mantero, and M. G. Pia. Distributed geant4 simulation in medical and space science applications using diane framework and the grid. In *Innovative Particle and Radiation Detectors, Nuclear Physics B (Proc. Suppl.) 123*, 2003.

[122] Martin Swany and Rich Wolski. Representing dynamic performance information in grid environments with the network weather service. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, 2002.

[123] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30:2004, 2003.

[124] S. Zhou, J.Wang, X. Zheng, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computing systems, technical report csri-257. Technical report, Computer Systems Research Institute, University of Toronto, 1992.

[125] *IBM LoadLeveler: User's Guide, Doc. No. SH26-7226-00, IBM Corporation (1993).*

[126] Robert L. Henderson. Job scheduling under the portable batch system. *LNCS*, 949:279 – 294, 1995.

[127] Anjomshoaa et al. Job Submission Description Language (JSDL) Specification, Version 1.0. Technical report, Open Grid Forum, 2005.

[128] E. Laure et al. Programming the Grid with gLite. *Computational Methods in Science and Technology*, 12(1):33–45, 2006.

[129] S. Andreozzi et al. GLUE Specification v. 2.0. GLUE WG, 2008.

[130] DMTF. *Common Information Model (CIM) v2.19.1*. Distributed Management Task Force (DMTF), 2008.

[131] William Lee, A. Stephen Mcgough, and John Darlington. Performance evaluation of the gridsam job submission and monitoring system. In *In UK e-Science All Hands Meeting*, pages 915–922, 2005.

[132] Silvia D. Olabarriaga, Piter T. de Boer, Ketan Maheshwari, and Adam Belloum. Virtual lab for fmri: Bridging the usability gap. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, 2006.

[133] Virtual laboratory for enhanced science (VL-e). http://www.vl-e.nl.

[134] Tristan Glatard, Kamel Boulebiar, and Silvia D. Olabarriaga. Workflow integration in vl-e medical. In *21st IEEE International Symposium on Computer-Based Medical Systems*, 2008.

[135] T. Glatard, Montagnat J., Pennec X., Emsellem D., and Lingrand D. Moteur: a data-intensive service-based workflow manager. Technical report, Laboratoire I3S Informatique, Signaux et Systéms de Sophia Antipolis, France, 2006.

[136] Andrea Clematis, Daniele D'Agostino, and Antonella Galizia. An object interface for interoperability of image processing parallel library in a distributed environment. *F. Roli and S. Vitulano (Eds.): ICIAP 2005*, LNCS 3617:584 – 591, 2005.

[137] Michi Henning. The rise and fall of CORBA. *ACM Queue*, Vol 4, Issue 5:28 – 34, 2006.

[138] Guoyin Cai, Yong Xue, Jiakui Tang, Jianqin Wang, Yanguang Wang, Ying Luo, Yincui Hu, Shaobo Zhong, and Xiaosong Sun. Experience of remote sensing information modelling with grid computing. *M. Bubak et al. (Eds): ICCS*, LNCS 3039:989 – 996, 2004.

[139] Andrew W. Dowsey, Michael J. Dunn, and Guang-Zhong Yang. ProteomeGRID: towards a high-throughput proteomics pipeline through opportunistic cluster image computing for two-dimensional gel electrophoresis. *Proteomics*, 4:3800 – 3812, 2004.

[140] Michael Beynon, Chialin Chang, Umit Catalyurek, Tahsin Kurcand Alan Sussman, Henrique Andrade, Renato Ferreira, and Joel Saltz. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Comput.*, 28(5):827–859, 2002.

[141] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics.* Prentice Hall, 2nd edition, 1997.

[142] National Library of Medicine. Insight segmentation and registration toolkit (itk). http://www.itk.org/.

[143] Ran Zheng, Hai Jin, Qin Zhang, Ying Li, and Jian Chen. IPGE: Image processing grid environment using components and workflow techniques. *GCC*, LNCS 3251:671 - 678, 2004.

[144] Ian J. Grimstead, Nick J. Avis, and David W. Walker. Automatic distribution of rendering workloads in a grid enabled collaborative visualization environment. In *In proceedings of Super Computing 2004, SC2004*, 2004.

[145] Gao Shu and Nick J.Avis. Workflow-based distributed visualization. In *In Proceedings of The Sixth International Conference on Grid and Cooperative Computing (GCC 2007)*, 2007.

[146] Ian J. Grimstead, Nick J. Avis, David W. Walker, and Roger N. Philp. Resource-aware visualization using web services. In *Proceedings of the UK e-Science All Hands Meeting*, 2005.

[147] Ian J. Grimstead, David W. Walker, Nick J. Avis, Frederic Kleinermann, and John McClure. 3d anatomical model visualization within a grid-enabled environment. *Computing in Science & Engineering*, pages 32 – 38, 2007.

[148] A. Al-Saidi, Nick J. Avis, Ian J. Grimstead, and Omer Rana. Distributed collaborative visualization using light field rendering. In *In Proceedings of the 9th IEEE Conference on Cluster Computing and the Grid (CCGrid), Vizualization Workshop*, 2009.

[149] Luke Chatburn. *Myriad: A framework for distributed and networked machine vision.* PhD thesis, Cardiff University, 2006.

[150] B. G. Batchelor, S. J. Caton, L. T. Chatburn, and R. A. Crowtherand J. W. Miller. Vision systems on the internet. In Kevin G. Harding, editor, *Proc. SPIE Vol. 6000 Two- and Three-Dimensional Methods for Inspectionand Metrology III*, pages 25 – 39, 2005.

[151] B. G. Batchelor, S. J. Caton, L. T. Chatburn, R. A. Crowther, and J. W. V. Miller. Networked vision system using a prolog controller. In Bhaskaran Gopalakrishnan, editor, *Proc. SPIE Vol. 5999 Intelligent Systems in Design and Manufacturing VI*, pages 151 – 162, 2005.

[152] S. Muller and H. Waller. Efficient integration of real-time hardware and web based services into Matlab. In Horton G, Moller D, and Rude U, editors, *Simulation in industry'99: 11th european simulation symposium*, pages 41–45. Simulation in Industry'99: 11th European Simulation Symposium 1999, 1999.

[153] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 28–31, 1998.

[154] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 122 - 131, 1995.

[155] Gilles Fedak Cecile, Gilles Fedak, Cecile Germain, and Vincent Neri. Xtremweb a generic global computing system. In *In Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01)*, pages 582–587, 2001.

[156] Cécile Germain, Vincent Néri, Gilles Fedak, , and Franck Cappello. Xtremweb: Building an experimental platform. *R. Buyya and M. Baker (Eds.:) GRID 2000*, LNCS 1971:91 - 101, 2000.

[157] Lici Lu. Resource management for a campus computational grid. Master's thesis, University of Adelaide, 2002.

[158] Douglas Thain and Miron Livny. Error scope on a computational grid : Theory and practice. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC)*, 2002.

[159] The NetSolve/GridSolve website - http://icl.cs.utk.edu/netsolve/index.html - last visited Nov 2006.

[160] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra. Recent developments in gridsolve. *International Journal of High Performance Computing Applications (Special Issue: Scheduling for Large-Scale Heterogeneous Platforms), Robert, Y eds. Sage Science Press*, 1, 2006.

[161] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In *HPCN Europe*, pages 491 – 502.

[162] EnFuzion product information: http://www.axceleon.com/products.html (last accessed June 2009).

[163] Sílvia D. Olabarriaga, Aart J. Nederveen, and Breanndán ÓNuállain. Parameter sweeps for functional mri research in the "virtual laboratory for e-science" project. In *Seventh IEEE International Symposium on Cluster Computing and the Grid: CCGRID*, pages 685 – 690, 2007.

[164] J.T. Moscicki, H.C.Lee, S.Guatelli, S.C. Lin, and M.G.Pia. Biomedical applications on the grid: Efficient management of parallel jobs. In *Nuclear Science Symposium Conference Record, IEEE*, 2004.

[165] J.T. Moscicki. Diane - distributed analysis environment for grid-enabled simulation and analysis of physics data. In *Nuclear Science Symposium Conference Record, IEEE*, 2003.

[166] Vladimir V. Korkhova, Jakub T. Moscickib, and Valeria V. Krzhizhanovskaya. Dynamic workload balancing of parallel applications with user-level scheduling on the grid. *Future Generation Computer Systems*, 25:28–34, 2009.

[167] Cecile Germain-Renaud, Charles Loomis, Jakub T. Moscicki, and Romain Texier. Scheduling for responsive grids. *Journal of Grid Computing*, 6:15–27, 2008.

[168] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java based grid programming environment. volume 17, pages 1079–1107, June 2005.

[169] Jason Maassen and Henri E. Bal. Smartsockets:Solving the Connectivity Problems in Grid Computing. In *Proceedings of The 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, Monterey, CA, USA, June 2007.

[170] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. User-friendly and reliable grid computing based on imperfect middleware. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07)*, nov 2007. Online at http://www.supercomp.org.

[171] A. Barak, A. Shiloh, and L. Amar. An organizational grid of federated mosix clusters. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2005.

[172] A. Barak. The mosix organization grid. In *A White Paper*. The Hebrew University of Jerusalem, Isreal, 2005.

[173] Globus: Fundamental technologies needed to build computational grids. http://www.globus.org.

[174] E. Afgan, P. Sathyanarayana, and P. Bangalore. Dynamic task distribution in the grid for BLAST. In *IEEE International Conference on Granular Computing*, 2006.

[175] R. P. Bruin, T. O. H. White, A. M. Walker, K. F. Austen, M. T. Dove, R. P. Tyer, P. A. Couch, I. T. Todorov, and M. O. Blanchard. Job submission to grid computing environments. *Concurrency and Computation: Practice and Experience*, 20:1329 – 1340, 2008.

[176] M. Calleja, B. Beckles, M. Keegan, M. Hayes, A. Parker, and M. Dove. Camgrid: Experiences in constructing a university-wide, condor-based grid at the university of cambridge. In *Proceedings of the UK e-Science All Hands Meeting*, pages 173 – 178, 2004.

[177] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, 2001.

[178] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[179] Mark Baker and Garry Smith. Gridrm: A resource monitoring architecture for the grid. *Grid Computing GRID 2002*, LNCS 2536:268 - 273, 2002.

[180] Zhiqun Deng, Zhicong Liu, Hong Luo, Guanzhong Dai, Xinjia Zhang, Dejun Mu, , and Hongji Tang. Nodes' organization mechanisms on campus-grid services environment. *H. Jin and Y. Pan and N. Xiao and and J. Sun (Eds.): GCC 2004*, LNCS 3251:464 - 471, 2004.

[181] Jik-Soo Kim, Bobby Bhattacharjee, Peter J. Keleher, and Alan Sussman. Matching jobs to resources in distributed desktop grid environments. Number CS-TR-4791, 2006.

[182] Fran Berman and Rich Wolski. Application-level scheduling on distributed heterogeneous networks (technical paper). In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC'96)*, 1996.

[183] Fran Berman and Rich Wolski. The apples project: A status report. Technical report, Department of Computer Science and Engineering, University of California, 1997.

[184] Abhijit Bose, Brian Wickman, and Cameron Wood. Mars: A metascheduler for distributed resources in campus grids. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, 2004.

[185] Katarzyna Keahey, Ian Foster, Timothy Freeman, Xuehai Zhang, and Daniel Galron. Virtual workspaces in the grid. *J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par*, LNCS 3648:421–431, 2005.

[186] David Isaac Wolinsky, Abhishek Agrawal, P. Oscar Boykin, Justin R. Davis, Arijit Ganguly, Vladimir Paramygin, Y. Peter Sheng, and Renato J. Figueiredo. On the design of virtual machine sandboxes for distributed computing in wide-area overlays of virtual workstations. In *Second International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, 2006.

[187] Sriya Santhanam, Pradheep Elango, Andrea Arpaci-Dusseau, and Miron Livny. Deploying virtual machines as sandboxes for the grid. In *Proceedings of USENIX Worlds*, 2005.

[188] http://www.virtualbox.org/ (Last accessed Feb 2009).

[189] X. Zhang, T. Freeman, K. Keahey, I. Foster, and D. Scheftner. Virtual clusters for grid communities. In *CCGRID '0g: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, 2006.

[190] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jos, A. B. Fortes, and Renato J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[191] http://damnsmalllinux.org/ (Last Accessed Feb 2009).

[192] H. Nishimura, N. Maruyama, and S. Matsuoka. Virtual clusters on the fly - fast, scalable, and flexible installation. In *CCGrid'07: Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 549 – 556, 2007.

[193] Maarten van Steen. On the complexity of simple distributed systems. *IEEE Distributed Systems Online*, Vol. 5, No. 5:1 – 3, 2004.

[194] Stephen D. Kleban and Scott H. Clearwater. Fair share on high performance computing systems: What does fair really mean? In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID03)*, pages 146 – 153, 2003.

[195] Yair Wiseman and Dror Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14 No. 3:561 – 592, 2003.

[196] Pablo Suau, Mar Pujol, Ramon Rizo, Simon Caton, Omer Rana, Bruce Batchelor, and Francisco Pujol. Agent-based recognition of facial expressions. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems - AAMAS '05*, pages 161 – 162, 2005.

[197] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE a white paper. Technical report, telecomitalialab.com, 2003.

[198] Andrew Harrison and Ian Taylor. Wspeer - an interface to web service hosting and invocation (pdf). In *HIPS Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models, IPDPS*, 2005.

[199] Ian Cooper and Coral Walker. The design and evaluation of mpi-style web services. *IEEE Transactions on Services Computing*, 99(1), 2009.

[200] Bruce Batchelor. QT, a Machine Vision Prototyping System. http://bruce.cs.cf.ac.uk/bruce/ – last accessed August 2009.

[201] Delft University Pattern Recognition Group. PRTools, a statistical pattern recognition toolbox. http://www.prtools.org/ – last accessed August 2009.

[202] Delft University Quantitative Imaging Group. DIPLib, a Matlab toolbox for scientific image processing and analysis. http://www.diplib.org/ – last accessed August 2009.

[203] SPM - Statistical Parametric Mapping. http://www.fil.ion.ucl.ac.uk/spm/ – last accessed August 2009.

[204] M.W.A. Caan, K.A. Vermeer, L.J. van Vliet, C.B.L.M. Majoie, B.D.Peters, G.J. den Heeten, and F.M. Vos. Shaving diffusion tensor images in discriminant analysis: A study into schizophrenia. *Medical Image Analysis*, 10:841 – 849, 2006.

[205] Maartje ML de Win, Liesbeth Reneman, Gerry Jager, Erik-Jan P Vlieger, Silvia D Olabarriaga, Cristina Lavini, Ivo Bisschops, Charles BLM Majoie, Jan Booij, Gerard J den Heeten, and Wim van den Brink. A prospective cohort study on sustained effects of low-dose ecstasy use on the brain in new ecstasy users. *Neuropsychopharmacology*, 32:458 – 470, 2007.

[206] P.J. Basser and C. Pierpaoli. Microstructural and physiological features of tissues elucidated by quantitative-diffusion-tensor MRI. *J Magn Reson B*, 111:209–219, 1996.

[207] Wiesje M. van der Flier et al. Accelerating regional atrophy rates in the progression from normal agiing to alzheimer's disease (ic-p2-144). In *Alzheimer's Imaging Consortium IC-P2: Poster Presentations*.

[208] S. Bagnasco, F. Beltrame, et al. Early diagnosis of alzheimer's disease using a grid implementation of statistical parametric analysis. In *Proceedings of Healthgrid*, pages 69 – 81, 2006.

[209] M.E. Shenton, M.E. Dickey, et al. A review of MRI findings in schizophrenia. *Schizophr Res*, 49:1–52, 2001.

[210] I. Agartz, J. Andersson, and S. Skare. Abnormal brain white matter in schizophrenia: a diffusion tensor imaging study. *Neuroreport*, 12:2251 – 2254, 2001.

[211] B. Ardekeni, J. Nierenburg, et al. MRI study of white matter diffusion anisotropy in schizophrenia. *Neuroreport*, 14:2025 – 2029, 2003.

[212] J. Foong, M. Maier, et al. Neuropathological abnormalities of the corpus callosum in schizophrenia: a diffusion tensor imaging study. *Journal of Neurology, Neurosurgery and Psychiatry*, 68:242 – 244, 2000.

[213] D Hubl, T. Koenig, et al. Pathways that make voices. *Arch. Gen. Psychiatry*, 61:658 – 668., 2004.

[214] R.A. Kanaan, J.S. Kim, et al. Diffusion tensor imaging in schizophrenia. *Biol Psychiatry*, 58(12):921–929, 2005.

[215] M. Kubicki, C. Westin, et al. Cingulate fasciculus integrity disruption in schizophrenia: a magnetic resonance diffusion tensor imaging study. *Biol. Psychiatry*, 54:1171 – 1180, 2003.

[216] Z. Sun, F. Wang, et al. Abnormal anterior cingulum in patients with schizophrenia: a diffusion tensor imaging study. *Neuroreport*, 14:1833 – 1836, 2003.

[217] F. Wang, Z. Sun, et al. Anterior cingulum abnormalities in male patients with schizophrenia determined through diffusion tensor imaging. *Am. J. Psychiatry*, 161:573 – 575, 2003.

[218] Silvia D. Olabarriaga, Jeroen G. Snel, Charl P. Botha, and Robert G. Belleman. Integrated support for medical image analysis methods: from development to clinical application. *Special Issue on Image Management in Healthcare Enterprises*, 2006.

[219] J.G. Snela, S.D. Olabarriaga, J. Alkemade, H. Gratama van Andel, A.J. Nederveen, C.B. Majoie, G.J. den Heeten, M. van Straten, and R.G. Belleman. A distributed workflow management system for automated medical image analysis and logistics. In *Proceedings of the 19th IEEE Symposium on Computer-Based Medical Systems (CBMS'06)*, 2006.

[220] Vincent W. Freeh. A comparison of implicit and explicit parallel programming. tr 93-30a. Technical report, University of Arizona, 1994.

[221] M.W.A Caan, L.J. van Vliet, C.B.L.M. Majoie, E.J. Aukema, C.A. Grimbergen, and F.M. Vos. Spatial consistency in 3d tract-based clustering statistics. *D. Metaxas et al. (Eds.): MICCAI 2008, Part I, LNCS 5241*, pages 535 – 542, 2008.

[222] H. Jiang and P. van Zijl et al. Dtistudio: Resource program for diffusion tensor computation and fiber bundle tracking. *Computer Methods and Programs in Biomedicine*, 81:106 – 116, 2006.

[223] H. Chui, A. Rangarajan, J. Zhang, and C. Leonard. Unsupervised learning of an atlas from unlabled point-sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:160–172, 2004.

[224] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23 no 3:187 – 200, 2000.

[225] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC02)*, 2002.

[226] Kavitha Ranganathan and Ian Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, 1:53 – 62, 2003.

[227] Sang-Min Park and Jai-Hoon Kim. Chameleon: A resource scheduler in a data grid environment. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)*, 2003.

[228] Srikumar Venugopal, Rajkumar Buyya, and Lyle Winton. A grid service broker for scheduling distributed data-oriented applications on global grids. In *2nd Workshop on Middleware in Grid Computing*, 2004.

[229] R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, 29(12):1104–1113, 1980.

[230] S. Paurobally, V. Tamma, and M. Wooldridge. A framework for web service negotiation. *ACM Transactions on Autonomous and Adaptive Systems*, 2(4), 2007.

[231] H. Ludwig, A. Keller, A. Dan, R.P. King, and R. Franck. Web service level agreement (wsla) language specification, version 1.0. *IBM Corporation, January*, 2003.

[232] D.D. Lamanna, J. Skene, and W. Emmerich. Slang: A language for defining service level agreements. *9th IEEE Workshop on Future Trends in Distributed Computing Systems-FTDCS*, pages 100–106, 2003.

[233] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Open Grid Forum's Grid Resource Allocation Agreement Protocol (GRAAP) Working Group, 2007.

[234] A. Paschke, J. Dietrich, and K. Kuhla. A logic based sla management framework. In *Semantic Web and Policy Workshop (SWPW) at ISWC 2005*, 2005.

[235] Michael Parkin, Rosa M. Badia, and Josep Martrat. A comparison of sla use in six of the european commissions fp6 projects. Technical Report TR-0129, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, April 2008.

[236] R. Sakellariou and V. Yarmolenko. Job scheduling on the grid: Towards sla-based scheduling. *High Performance Computing and Grids in Action*, 16:207–222, 2008.

[237] Nicole Oldham, Kunal Verma, Amit Sheth, and Farshad Hakimpour. Semantic ws-agreement partner selection. In *15th international conference on World Wide Web*, 2006.

[238] N. Borissov, S. J. Caton, Omer Rana, and Levine A. Market protocols for the provisioning and usage of computing services. In *6th International Workshop on Grid Economics and Business Models*, pages 160–170, 2009.

[239] J. Stoesser and D. Neumann. A model of preference elicitation for distributed market-based resource allocation. *17th European Conference on Information Systems (ECIS-2009)*, 2009.

Appendix A

Summary of Heuristics and Formulae Employed in this Research

In this Appendix a summary of the heuristics and formulae used in this thesis is provided to provide an overview of the mathematical model implemented in this thesis.

A.1 Terminating Misbehaving Jobs

A termination heuristic was first mentioned in Section 2.4.2, and derived in Section 4.4, as a means of identifying runaway jobs.[1] The impact of runaway jobs on performance was empirically demonstrated in two different contexts in Chapter 2. By observing the tendencies of runaway jobs it was discovered that if the current runtime of a job was compared to the average it was possible to accurately[2] identify runaway jobs, and consequently terminate them to reduce their negative impact upon performance and efficiency of resource utilisation.

The heuristic has been presented twice in the thesis. It was first presented in Section 2.4.2 (see: Figure A.1), here it was used during the steering of Matlab-based empirical experiments to performance test the Condor pool and identify when jobs are running away, and consequently terminated. Its second occurrence (see: Figure A.2) was in Section 5.3.3 where it was used by the CRM (Section 5.3) to identify when IAS (Section 5.2) jobs were running away. With the exception of the difference in how the threshold is computed,[3] there are two distinct differences between these two versions. Firstly, that in Figure A.2 the learning process of the CRM is included. This inclusion is made so that the computed termination threshold can be based on the historical observations of the workstation in question, i.e. its average initialization time. Secondly, a 10 minute minimum threshold is included in Figure A.1. The motivation for this is to prevent the termination heuristic

[1]A runaway job is one which runs for an inexorable period of time, but does not consequently fail. Instead, it is stuck in a state transition or blocked by the operating system.

[2]99.7% correctly classified normal jobs and 95% runaway jobs, based upon the test data from Section 2.3

[3]In Figure A.1 the job runtime is also considered: "predefined runtime", as in this experimental analysis jobs with a known and predefined runtime where used.

from being too aggressive during the learning process, as the nature of workstations used cannot be determined in advance. For example, if only workstations with good performance were initially acquired the heuristic could behave inappropriately when a less powerful workstation was acquired. The minimum runtime was also originally included in early versions of Figure A.2, for the same reason. However, as the historical basis of the CRM's performance model developed such a minimum was no longer required.

Figure A.1: The Autonomic Job Manager's job termination heuristic

IF
$Runtime > Max(10\,minutes, avg(Matlab\,Initialisation\,Time){+}5StandardDeviations) +$
$predefined\,jobruntime$
$THEN$
$terminate\,and\,resubmit$

Figure A.2: Heuristic used to determine if an IAS job is portraying anomalous behaviour, where WS is workstation name and SDs is standard deviations. If the CRM does have enough information about a specific workstation, it will use the global average for all workstations.

$$JobRuntime > avg(WS, initTime) + 5SDs$$

A.2 Rescheduling Tasks as a Consequence of New Resources

Due to the nature of the resources in use, i.e. they are provided on a voluntary basis, it is often the case that extra resources become available stochastically. In addition, as resources (IAS instances) are used interactively, Tasks are scheduled so that a queue can be established on the compute node, enabling data dependencies to be transferred as other Tasks are being executed. This approach was referred to as Dynamic Task Clustering, further details on the scheduling model employed can be found in Section 4.7. The consequence of stochastic behaviour at the resource level is that when a new resource is acquired by the CRM and allocated to an application, all incomplete Tasks may have already been scheduled to one or more IAS instances, and therefore load balancing is required to capitalize on the new IAS instance. In order to achieve this, a load balancing heuristic was introduced in Section 4.7.1 (see: Figure A.3) to enable the application scheduler to decide from which IAS instance(s) to take one or more Tasks to schedule to the newly acquired IAS instance. Note that this heuristic is also employed when one IAS completes all Tasks in its local queue and no Tasks remain unscheduled by the scheduler.

As a precursor to the application of this heuristic, IAS instances are ranked according to the size of their Task queues, i.e. a Task is taken from the IAS with the most Tasks in its queue. The heuristic is then employed as a tiebreaker to choose the IAS instance with the highest projected execution time. In other words, it selects an IAS that has the most remaining execution time for its current Task. This is performed by comparing the current Task's runtime to a 'similar' one that has been observed in the past. See Section 4.7.2 for a definition of Task similarity (affinity).

Figure A.3: Heuristic used to determine from which IAS instance to take a Task for load balancing, where w is a Workstation in the set of workstations W, T_e is the currently executing Task, and T_{sim} is a Task that is 'similar' to T_e.

$$w' = \forall w \in W min(AvgExecutionTime(T_{sim}) - RunningTime(T_e))$$

A.3 Calculating Application Makespan for Traditional Condor Approach

The final mathematical expression (see: Figure A.4) employed in this thesis was used from a purely theoretical perspective, namely to determine whether it was possible for a traditional Condor approach to outperform the Virtual Cluster implementation within the application scenario presented in Chapter 6. As a precursor to the application of this formula several runs of the application were performed by the Virtual Cluster in order to determine the runtime of an average Task, as well as the minimum and maximum runtimes. These statistics were then combined with the historical data of Matlab initialization times from Chapters 2 and 7. When these two pieces of information are combined the formula can be employed to estimate the number of scheduling iterations that would be required to complete a given application, and therefore the best case, average case and worst case time frames needed by Condor to execute all jobs. See Section 6.9, for how this was performed.

Figure A.4: Formulae to compute the number of iterations required to execute an application using a traditional Condor approach, where n is the current iteration.

$$Complete_n = Tasks - ROUND(FailureRate * (Tasks - Complete_{n-1}))$$
$$where$$
$$Complete_0 = 0$$