MPI-Style Web Services:
An Investigation into the Potential of Using Web Services for MPI-Style Applications

A thesis submitted in partial fulfilment
of the requirement for the degree of Doctor of Philosophy

Ian Michael Cooper

Cardiff University
School of Computer Science

September 2009

UMI Number: U585368

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U585368
Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

This research investigates the potential of the Web services architecture to act as a platform for the execution of MPI-style applications. The work in this thesis is based upon extending current Web service methodologies and merging them with ideas from other research domains, such as high performance computing. MPIWS, an API to extend the functionality of standard Web services is introduced. MPIWS provides MPI-style message passing functionality to facilitate the execution of MPI-style applications using Web service based communication protocols. The thesis then presents a large selection of experiments that perform a comprehensive evaluation of MPIWS's performance. This performance is compared with an existing MPI implementation that has the option of transmitting data either via Java serialised objects, or via the Java native interface to an underlying C implementation of MPI. From the results obtained from these experiments, it can be concluded that using MPIWS for applications requiring MPI-style message passing between services is potentially a practical and efficient way of distributing coarse-grained parallel applications. The results also show that the use of collective communication techniques within the Web services architecture can significantly improve the efficiency of suitable applications such as molecular dynamics simulation.

MPI-style communication can also be used to enhance the performance of Web service based workflow execution. Tests conducted have evaluated a range of functionality that can be provided by the MPIWS tool. This evaluation shows that *direct messaging* between services, without sending data via the workflow manager, can improve the efficiency of Web service based workflow execution.

Acknowledgements

During the process of completing this thesis I have had a great deal of help and support: technically from my supervisors, Dr Coral Walker and Professor David Walker; financially, from the School of Computer Science and the Engineering and Physical Sciences Research Council; and emotionally from my family and friends. I thank them all for the encouragement and enthusiasm (and also the proof reading) throughout the past few years.

I am also specifically grateful to Ahmed Alazzawi, Simon Caton, Martin Chorley, Bast Greede, Mark Hall, Mathew Morgan, Ian Wootten and the rest of the post graduate research students at the School of Computer Science, Cardiff. Without those random discussions that frequently make ideas go click, I would never have got this far...(and that proof reading again).

I also especially thank my wife, Alison, who has encouraged and supported me through the whole process and made this possible; and my children, Oliver and Sophie, whose faces lovingly look at me from their photos on my desk and say ... "GET IT FINISHED".

Table of Contents

List of Figures

Chapter 1

Introduction

Chapter Overview:

This chapter introduces the work presented in this thesis and discusses its relevance and uses. There are three main motivations behind this work: to support recently developed workflow languages to implement an MPI-style of message passing, to provide a platform for High Performance style applications to run over a service-oriented architecture, and to allow loosely-coupled service-oriented applications to communicate directly between component services. Each of these motivations will be covered in more detail in this chapter. The main contributions and hypothesis of the thesis are presented in this chapter in order to clarify the aims and objectives of the work. Finally the remainder of the thesis is summarised on a chapter by chapter basis.

1.1 Introduction

A workflow is a series of processing tasks, each of which operates on a particular data set and is mapped to a particular processor for execution. In a loosely-coupled Web service environment, a workflow can itself be presented as a Web service, and invoked by other workflows. Web service standards and technologies provide an easy and flexible way for building workflow-based applications, encouraging the re-use of existing applications, and creating large and complex applications from composite workflows.

The Web service infrastructure, as will be discussed extensively in Section 2.2, offers services the conformity to open communication standards. This enables services to communicate over the Internet with other services, deployed on any Web server, written in any programming language, and under the control of any administrative domain. The main problem with using Web services for applications that require both high performance and interoperability, is the speed of the Web service's messaging protocol [86].

In spite of the performance concerns of the Web service's messaging protocol (SOAP), the use of Web service architectures to build distributed computing workflows for scientific applications, has become an area of much active research. Recently developed workflow languages, such as Grid Services Flow Language (GSFL) [69] and Message Passing Flow Language [60], have started addressing the problem of intercommunicating processes. These languages provide the functionality to describe the act of one executing service communicating directly with another concurrently executing service.

Business Process Execution Language for Web Services (BPEL4WS) is commonly used for composing Web service based scientific workflows [1], but users are limited to applications with independent processes. In the case of a workflow with loops containing multiple independent tasks, the overhead in invoking these sub-tasks is incurred every iteration. In addition, any iterative data that is to be shared by these tasks must be passed to the service by a mediator. Figure 1.1 shows a workflow implementing a loop of three independent sub-task services. These services are connected by a mediator service to control the number of loop iterations and to control the data sharing between the services.

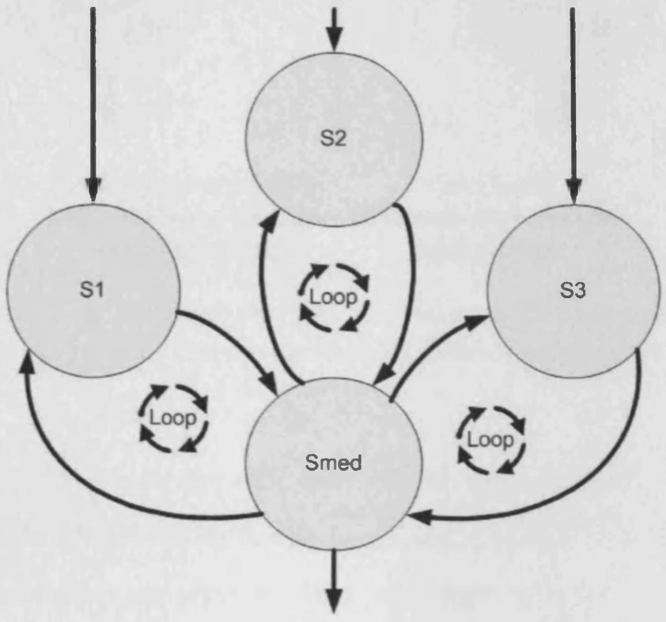


Figure 1.1: A workflow showing services S1, S2 and S3 concurrently performing an iterative task by looping via a mediator service Smed.

As an alternative to this scenario, Figure 1.2 shows the loop implemented using MPI-style message passing communication between the three services, which enables the services to be written in such a way that they can process their own loop constraints. The services can also perform data sharing through loosely synchronous communication at each iteration.

This alternative, as well as eliminating the need for the mediator service and re-invocation at every iteration, allows the use of MPI-style collective communication

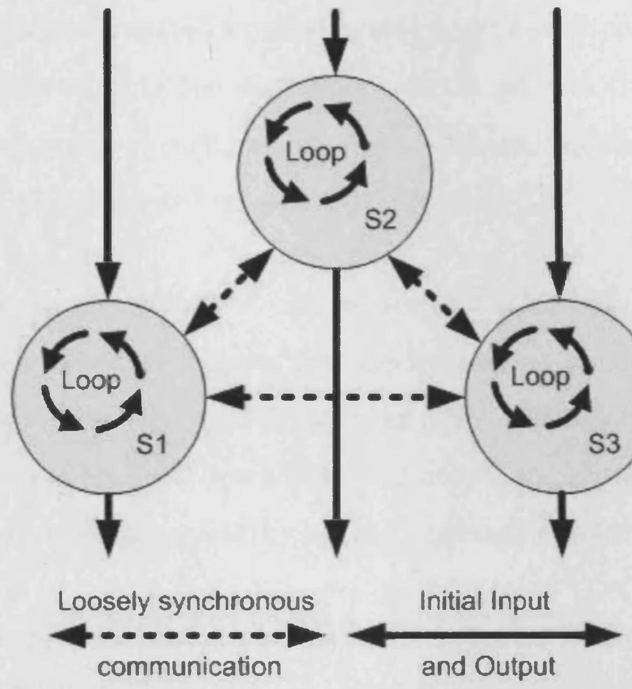


Figure 1.2: A workflow showing services S1, S2 and S3 concurrently performing an iterative task by looping internally sharing data directly with each other.

techniques [107] to improve the efficiency of the data transfer. For example, if there were eight parallel services in the loop, and the data to be shared was sent from all services to all other services, then each service could *Broadcast* its data.

MPI-style applications also have a tendency to employ this loop functionality, in that they typically perform a round of calculation followed by a round of communication between the processing elements.

One example of this style of application is described in Mu and Rice [79], where a set of Partial Differential Equation solvers are used to model an automotive engine heat flow problem. Each service is initialised to model a separate constituent part, constructed from a different material and possessing different thermal characteristics. At each time iteration, the boundary conditions between the component parts must be passed to the neighbouring service. Another example is a distributed molecular dynamics model, where a number of particles are divided between services involved in the simulation. Again, at each time interval in the simulation, the velocities of each particle must be shared between all the services. This example will be discussed extensively in Section 7.3.

From the arguments presented, it can be seen that MPI-style communication offers Web service based workflows the opportunity to expand their available functionality. But conversely, Web services offer MPI-style applications the flexibility of connectivity, interoperability and ease of deployment. This provides a very strong motivation for research into the combination of the two approaches.

This thesis presents work that investigates the potential and suitability of using a Web service infrastructure to support parallel applications and workflows that require MPI-style message passing. The thesis presents a detailed review of the current state of play in the fields of: Web service architectures; Workflow languages and managers; and MPI techniques and implementations. The thesis then discusses in depth the motivations and the problems involved in combining MPI-based applications with Web service oriented applications by examining the related work that has achieved progress in this area (Chapter 3). In the contribution chapters (Chapters 4, 5 and 6), the design of MPIWS (MPI over Web services) is discussed, along with the presentation of evaluation results. These results compare MPIWS against mpiJava [19], a leading high performance Java implementation [7]. To allow the MPIWS tool to be assessed on a realistic problem, in Chapter 7 a molecular dynamics simulation that has been adapted to use MPIWS is presented, and performance results are discussed. The final chapters of the thesis (Chapters 8 and 9) critically evaluate the work carried out and the contributions that have been made, comparing them with the related work presented in Chapter 3. These chapters also detail further work that could lead on from this thesis and the final conclusions that can be made following this work.

1.2 Hypothesis

The research hypothesis is :

> Web service component processes can communicate directly with each other, using Web service based communication protocols, to enable efficient parallel processing for MPI-style scientific applications, and to improve Web service based workflow throughput.

In this hypothesis, the term "Web service component processes" is defined to be Web services that are combined within a workflow to create a larger application. The hypothesis states that these processes can "communicate directly with each other", i.e., a service can be invoked by a workflow manager and then, while that service instance is running, send and receive messages to and from other running service instances. The phrase "using Web service based communication protocols" states that the messages sent between the services will be sent over standardised and *open* protocols used within the Web services framework published by the Organization for the Advancement of Structured Information Standards (OASIS).

One aim of this hypothesis is to "enable efficient parallel processing for MPI-style scientific applications". Within the work carried out in this thesis, a tool set will be designed, implemented and then tested to show that this Web service methodology can be used to run MPI-style applications efficiently. MPI-style applications are parallel applications that utilise the communication techniques described in the MPI Specification [78]. This includes point-to-point and collective communication operations.

The second aim of the hypothesis is to "improve Web service based workflow throughput" i.e. to reduce the latency of data communications, through the workflow's hardware infrastructure, from one service to the next. Data can be sent directly from service to service without the need to go via the workflow manager, which reduces a potential bottleneck in the system. The collective communication techniques can also be used within the workflow to enhance the performance of the data distribution.

1.3 Contributions

This section lists the major contributions that are achieved by this work. The three main contributions are:

1. The demonstration that MPI-style point-to-point communication can be efficiently executed over the Web services framework.

2. The demonstration of efficient collective communication techniques over the Web services framework.

3. The demonstration that *direct messaging* can improve the efficiency of certain Web service based workflows.

These contributions have been made possible by the development of MPIWS, a message passing tool for Web services. MPIWS facilitates the point-to-point communication of data from one service to a concurrently executing service. It also facilitates a range of MPI-style collective communication operations, which use the processing and networking resources of a distributed set of computers in order to increase the speed of distributing or collecting data within that set of computers. The contributions have been demonstrated by a comprehensive set of tests that are detailed in this thesis (Sections 4.3, 5.3 and 6.2).

The majority of the work covered in this thesis has been published. The initial work on MPI-style point-to-point communication is available in the proceedings of ICCS 2008.

> Ian Cooper and Yan Huang. The Design and Evaluation of MPI-style Web Services. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, ICCS (1), volume 5101 of Lecture Notes in Computer Science, pages 184-193. Springer, 2008.

A further publication is due to appear in the IEEE Transactions on Services Computing. This details the collective communication functionality and evaluates MPI-style applications being executed using MPIWS.

> Ian Cooper and Coral Walker. The Design and Evaluation of MPI-style Web Services. IEEE Transactions on Services Computing, Volume 2, No.1, pages 197-209, 2009.

1.4 Thesis Summary

Chapter 2: Background

Chapter 2 introduces the general areas of Web services, workflow and MPI. It discusses the various architectures associated with Web services, such as Remote Procedural Call (RPC) and REpresentational State Transfer (REST), and looks at some of the current methods of increasing the efficiency of SOAP messaging in order to justify some of the design decisions in the contribution chapters. Workflow languages are discussed as well as the development of languages that support use of MPI-style communication operations. These languages are analysed so that they can easily be referred to in the motivation section (Section 3.2). Then MPI is discussed, including a brief overview of some of the more influential implementations, along with a brief synopsis of the development of the collective communication operations.

Chapter 3: Combining Web services with MPI

Chapter 3 discusses the motivations and problems associated with the combination of Web services and MPI. The chapter contrasts the two approaches and provides a qualified argument for the motivation to research this area. There are differences in terminology between the two styles, especially in the area of blocking and non-blocking communication. One very important aspect of this research is the evaluation. This chapter discusses the objectives of the evaluation and the proposed evaluation methodology. This chapter also details the related work and gives a brief outline of the work presented in this thesis within the context of this related work.

Chapter 4: Point-to-Point Communication

The implementation of MPIWS can be neatly separated into point-to-point communication and collective operations. Chapter 4 is the first of the contribution chapters and discusses the design of the point-to-point functionality and the methods by which it was evaluated. This chapter also presents results of tests to evaluate MPIWS against mpiJava by performing data transfers with both; Java Objects, and primitive data types. These results show that under certain constraints, MPIWS can perform comparably with existing Java-based MPI implementations.

Chapter 5: Collective Operations Communication

Chapter 5 is the second of the contribution chapters. It describes the design of the MPIWS collective operations, including; Broadcast, SendReceive, Gather, Reduce, and Barrier. These operations have been implemented and evaluated. The results of running these operations on both MPIWS and mpiJava (transferring Java Objects and also primitive data types) are presented. These results show that the collective communication functionality within the Web services architecture is a viable objective.

Chapter 6: Enhancements to Workflow Communication Structures

Chapter 6 contains a discussion on designing a tool to provide a high speed communication architecture for MPI-style applications. The chapter also contains a discussion on allowing generalised Web services to use the functionality provided by MPIWS to enhance the efficiency of workflow communication. These two ideas are compared and the differences between them are contrasted. Chapter 6 describes how MPIWS is used to provide *direct messaging* between the services within a workflow. When tested against standard workflow techniques, *direct messaging* is shown to enhance the communication performance of certain workflow applications.

Chapter 7: Applications

Chapter 7 describes the evaluation of the MPIWS tool in a real environment. There is discussion and presentation of two applications that have been adapted to use the MPIWS tool. These applications show that the MPIWS API can do what was hoped of it, and the performance results from comparisons with mpiJava executions of the same applications are discussed. The first application is a parallel one dimensionally blocked matrix multiplication, and the second application is a molecular dynamics simulation code called MolDyn [76].

Chapter 8: Conclusions

In Chapter 8 the final comments on the work that has been undertaken are made. The completed work is discussed in relation to other similar work and the distinctions and similarities are detailed. Throughout the work presented in this thesis, one of the main methods of evaluation has been to compare the performance of the MPIWS tool with the performance of mpiJava. This evaluation strategy is appraised and critically discussed. Finally the conclusions are drawn and the contributions that this thesis has made are justified.

Chapter 9: Further Work

In Chapter 9 ideas for future development of the research into MPI-style communication using the Web services infrastructure are presented. Some suggestions for the further development of the MPIWS tool are also made.

Chapter 2

Background

Chapter Overview:

This chapter introduces the key elements involved in this research, namely: Web services, MPI and workflow. The aim is not to be a reference manual but to outline some of the more involved aspects of these architectures so that in the following chapter, the issues surrounding the combination of Web services and MPI coding styles can be addressed.

2.1 Introduction

This chapter provides a review of the topics associated with this research and links these topics to the work presented in the thesis. The background has been separated into three sections: Web services, Workflows and MPI.

The work in this thesis is based upon extending current Web service methodologies and merging them with ideas from other research domains, such as high performance computing.

In this chapter, the Web services section introduces some of the relevant aspects of WS-* standards such as WS-Notification and WS-Resources, in order to explore the current state of play. The section also compares different styles of Web services such as SOAP based services and the REpresentational State Transfer (REST) architecture to allow the design choices in the contribution chapters (Chapters 4 and 5) to be justified.

Workflows, and the languages that describe them, have been briefly discussed in the introduction (Chapter 1). They are an important part of the motivation for this research and are therefore covered in more detail in the background section. A brief history of their evolution and an analysis of their limitations is given, in order for the enhancements that can be provided by MPIWS to be clearly defined and evaluated.

The final key area in this research is MPI, including the techniques involved in the MPI implementations and collective communication algorithms. In the MPI section, there is an overview of the MPI architecture and its usage. This allows the MPI architecture to be critically contrasted with the Web services architecture, highlighting both the problems that need to be overcome and the benefits that will be obtained by merging the two areas. Also MPIWS has implemented a number of collective communication operations. The optimisation of these collective communication algorithms is constantly being researched and updated. A review of the applicable research is presented and a performance model is outlined in order to aid the evaluation of the collective algorithms.

2.2 Web Services

2.2.1 Overview

The research in this thesis uses Web services to perform an MPI-style of communication. In order to define the contribution made by this work, it is important to understand the limits of current Web service operations.

The World Wide Web (WWW) has been around in its most basic form since 1990, and is designed to convey information in human readable form to users via a system of servers and Web browsers. More recently the WWW community has turned its attention to the Semantic Web, an extension of the current WWW in which information or data is given well-defined meaning [13] in order for it to be processable by machines. The Web services software architecture is designed to support interoperable machine to machine interaction over a network [105]. It is designed around a client server architecture transmitting messages in open standards format.

The fundamental idea behind Web services is that the implementation of the client operation is totally abstracted from the implementation of the service. In fact, this idea is taken further in REST services where the services adhere more specifically to a Client Stateless-Server architecture. This means that each request that the client makes to the service *must* contain *all* information to process that request and must not rely on any stored information on the server side of the communication boundary [36]. Both Web services and REST services allow and encourage the use of very loosely-coupled services that can be combined into composite applications. The idea behind the Stateless-Server approach in the REST architecture is that a session can be moved from one server to another server during the course of that session with no loss of data or accuracy.

If this is related to the MPI-style of programming then it can be seen that the whole ethos of the REST architecture cannot be applied to MPI applications, as the MPI service *must* retain state throughout the whole session to allow the communication to be directed to the correct service instance.

Whilst Web services are effectively stateless, there is a set of specifications that define the Web Services Resource Framework (WSRF). These standards describe how the concept of *state* can be achieved within the Web services architecture and are reviewed in Section 2.2.2.

Web services communicate by sending messages between themselves. The messages are the important part of a Web service composition. A service may be described in the service's Web Services Description Language (WSDL) document that describes the messages to and from a service. This WSDL document outlines the interface to the service in an open standards format that all Web services can understand. WSDL documents can be published alongside the service they describe, so that clients can readily obtain the information needed to access the service. This combination of the accessible interface and common communication structure ensure that any client, written in any language and executing on any platform, can access any service, written in any other language and deployed on any other platform. In this manner, Web services enable distributed applications to work as a network of intercommunicating subtasks that transfer requests and responses as a simple exchange of messages. These messages take the form of an extensible Markup Language (XML) [16] document. XML is to data what HTML is to text [98] – it is a self-describing document that can be processed by a machine [104]. There are two aspects of the Web service messages that are relevant to the design of MPIWS: the message exchange architecture, which is discussed in Section 2.2.3; and the message encoding, which is discussed in Section 2.2.4.

2.2.2 State within Web Services

Web services in themselves can be very simple, but to allow them to be more versatile, the WSRF technical committee have defined a set of open standards that define enhanced functionality. These specifications are referred to as the WS-* specifications and are published by the Organization for the Advancement of Structured Information Standards (OASIS). Some of these specifications are very relevant to this research.

A simple Web service is inherently stateless. This means that there is no continuity between consecutive or independent invocations of the service. Each service invocation is totally independent of any other service invocation. For this research it is required to transfer data from one service invocation to another concurrently running service invocation, so the concept of state is required. The WSRF specifications outline stateful Web services which are, in turn, an extension of Web services. These specifications show that the use of a static data structure, or document can be used to transfer data from one service invocation to another service invocation. Figure 2.1 [96] shows this relationship in the context of the Globus toolkit, a grid middleware tool, and extends this to the Open Grid Services

Architecture (OGSA).

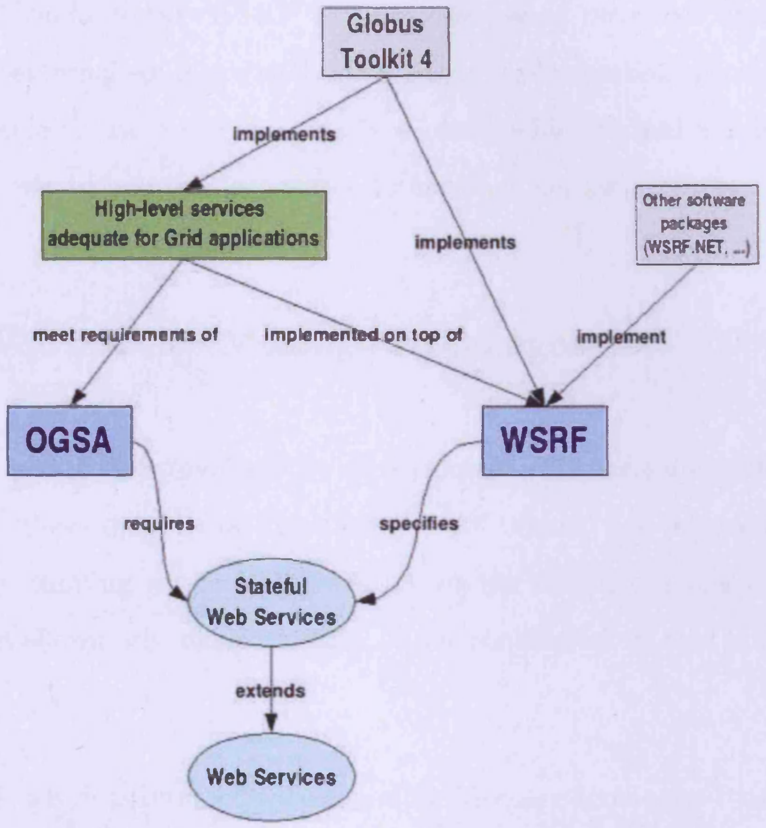


Figure 2.1: The relationship of Web services to the Web Service Resource Framework

The WSRF specifications use resources to handle all stateful data. In the WS-Resource [47] specification, a resource is defined as a set of properties that can be accessed via a Web service. This resource must be uniquely identifiable by the Web service, as different invocations of the Web service will access different versions of the resource.

The WS-Resource specification also defines a WS-Resource. This is a much more strictly controlled entity which comprises a set of sub elements, referred to as resource properties. The WS-Resource needs to be accessed and addressed via a unique address which addresses an individual resource. The WS-Resource *must* also support accessing elements of that resource via the WS-ResourceProperties specification. In the context of MPIWS and the work presented in this thesis, it is argued that: if the WSRF accepts the use of resources available to be accessed by external sources via the WS-ResourcesProperties specification, then it is reasonable to use a resource to allow identifiable internal access to stateful data that is not to be made accessible to external sources.

2.2.3 Web Service Message Exchanges

The design of MPIWS involves the development of a message exchange mechanism that allows data to be transferred from one service instance to another concurrently running service instance. A review of current message exchange mechanisms allows any developments to be considered in the MPIWS design process.

Typical Web services interact with a simple Message Exchange Pattern (MEP), the most common of which is the Request-Response MEP. It consists of the client sending a *request* to the service and then the service returning its *response* on completion of the service. Another common MEP is the Request-Only MEP, which allows the client to send its *request* and not receive a *response*. Both of these patterns are used to invoke the service, therefore they cannot be used to transfer data directly into a running service.

Another message exchange mechanism is notification. WS-Notification is a group of documents that describe a Publish-Subscribe-Notify system for Web services. This is an exchange of messages that enables a service to asynchronously receive data that it has requested about a certain topic. This is, at its simplest, achieved by the data consumer Web service subscribing to a *Topic* within a data producer. The producer then stores a reference to that consumer in its database and then, any time that the *Topic* data is updated, the producer will *Notify* the consumer. Figure 2.2 shows the scenario for the Subscribe-Notification that is presented by Graham et al. [48], in this scenario the Notification Consumer can be the same service as the Subscriber.

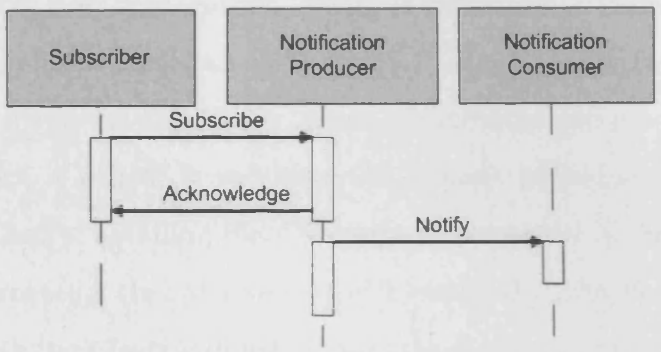


Figure 2.2: Messages in a Subscribe-Notification scenario

When looking at WS-Notification there are two main papers of interest: the first is the WS-Base Notification [49] which is the specification on which all the other specifications for WS-notification are based. The second is Publish-Subscribe-Notification for Web services [48], a white paper describing WS-Notification and how it is used.

WS-Notification *recommends* that all messages are secured using the mechanisms described in the WS-Security specification. These are a set of recommendations designed to enhance SOAP messaging and to provide message integrity and confidentiality [71]. WS-Notification allows messages to be sent to service endpoints but it doesn't support integrating the data into the running service invocation.

Another WS-* specification is WS-Addressing [106]. This defines a configuration element, *Reply-to*, in the SOAP header that redirects the output of one service to another location other than the initialising client. This *Reply-to* element is good for a single hop, but for multiple hops and MPI-style communication patterns it would not suffice.

Further research into the extension of standard message exchange mechanisms is provided by Ruth et al. [90]. They describe an implementation of a Single Request - Multiple Response (SRMR) MEP. In this message exchange mechanism for SRMR, an application uses an agent to relay the service call to the service, and the agent responds to the application when the initialisation process is complete. When the service is called, it validates the request information and returns a response to the agent, detailing the *Correlation ID* (the ID of the operation) and the number of requests that the service will eventually provide. The agent can then register with the *ClearingHouse* (a centralised service that collects responses from many services). When the initialisation stage completes, the agent then responds to the application. When all the responses have been collected by the Clearing House, they will be returned to the agent, via Socket communication. When the agent has all the responses, it then notifies the application, and the application then polls the agent for specific responses. This research is not directly relevant to the research undertaken in this thesis, but the similarities are that it is trying to allow the application to receive data that does not come directly via the standard Web services response mechanism.

Following this review of the mechanisms currently available for exchanging data between Web services, it can be established that there are no standardised Web service technologies that are designed with MPI-style messaging in mind. Further research into this area of data exchange is examined in the related work section (Section 3.4).

2.2.4 Message Transmission and Encoding

The Web services architecture is designed to abstract the implementation of distributed applications from the communication between them. One protocol commonly used in the Web services architecture is SOAP [53]. SOAP (originally, but no longer, an acronym for Simple Object Access Protocol), now on version 1.2, is an open protocol published by the World Wide Web consortium (W3C). W3C defines a mechanism for communicating XML-based documents over a transport layer. Part 2 of the W3C specification defines a SOAP binding to HTTP, but it also states that SOAP can be transported over bindings other than to HTTP, or other transmission protocols such as Simple Mail Transfer Protocol (SMTP).

The SOAP standard describes the XML-based format of the request, response and fault messages. It does not concern itself with the MEPs associated with these messages. The SOAP message comprises: a header part, which is used in the directing of the message to the correct service; and a body part, which is used in the service application. SOAP specifies that any SOAP container or server containing SOAP services should process the children of the body element but does not have to process their children (apart from fault messages). The structure of a SOAP message can be seen in Figure 2.3 [109].

SOAP was originally designed as a Remote Procedure Call (RPC) protocol that could be transmitted through firewalls, due to the SOAP messages being sent over HTTP or SMTP. For use in MPIWS, the greatest disadvantage of SOAP messages is that they are a verbose method of transferring data. This is due to both the extra header information and each item of data having to be labelled in an element or attribute. This makes them very inefficient for performance computing. This inefficiency is discussed in the next section.

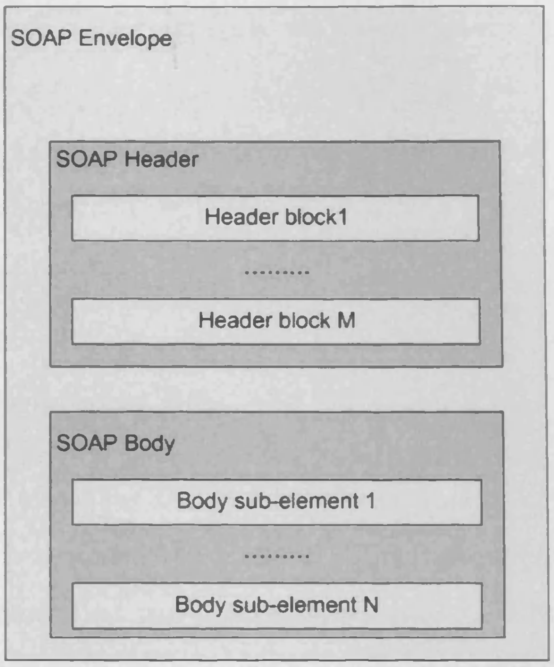


Figure 2.3: Structure of a SOAP message

Data Transmission in SOAP messages

There is a problem when it comes to sending the data within a SOAP message. SOAP uses XML and if true XML formatting is to be used, i.e. listing each entity of the data within a tagged element, the meta-data overhead within the message is potentially massive. This is due to the insertion of XML tags, and the need to represent the data as a series of characters. The most efficient method of encoding data is to serialise it into a binary representation. In the Java language there is an in-built function to transform Objects to their binary encoded representation. This is the mechanism that mpiJava uses to encode its objects before sending them to a socket. The problem is that a binary file cannot translate directly to string format, as there are not enough characters available. ASCII defines 94 printable characters, however XML reserves '<', '>' and '&' [57]. There are several standard ways that SOAP messages can deal with this problem [57]:

– Binary to character encoding such as Base64 encoding [40], or ASCII85 encoding

– Packaging such as SOAP with Attachments (SwA) [10], or Message Transmission Optimization Mechanism (MTOM) [4]

– Binary XML encoding [11]

– Linking [57]

– SOAP Message Compression [46]

Binary to character encoding translates the raw binary digits into a series of characters that can be transmitted in XML. This means that, because there is a number of binary values that can not be represented by XML viable characters, the size of the translated message could be increased. For Base64 encoding the message size increases by approximately 33% [40]. For ASCII85 encoding which allows 4 bytes to be represented by 5 characters, a space overhead of 25% [57] is produced.

There has also been recent research into improved methods of binary encoding for use within XML documents such as [57] who propose a flexible coding format that reduces the space overhead to under 1% on text files and under 2% on scientific data files. This coding format takes approximately the same time to process data as Base64.

Binary XML encodings [11] reduce the size of the transmitted data by applying a compression algorithm to the whole XML file before sending it.

Linking places a link to the binary file in the XML document so the receiving application can then retrieve this file via another protocol such as FTP.

SOAP Message Compression is an extra step that can be used to compress the whole SOAP message, gzip [42] is one form of compression that is supported by many Web servers [46]. The problem with this type of compression is that there still needs to be a method of incorporating the data into the SOAP message in order for it to be compressed. Another point about the message compression is that all the data and the meta-data is compressed together so in order to access the meta-data for message forwarding, the whole message needs to be uncompressed.

SwA is a method of attaching files to SOAP messages externally to the SOAP envelope. As file sizes start to increase, the problems with BaseX style encoding start to increase. A 1Gb file takes a lot of time and memory to encode and decode [72]. If this 1Gb file can be sent externally to the XML SOAP envelope, the raw binary file could be sent within the Multipurpose Internet Mail Extensions (MIME) envelope with binary Content Type Encoding. The problem with SwA is, due to SOAP prohibiting "Document Type Definitions" within messages, it does not describe the contents of the attachment so the receiving tool can not automatically know what it is [4] (See Ying et al. [110] for a comparison of transmission speeds using SOAP with Attachments and true XML formatting). Apache's AXis Object Model (AXIOM), which supports MTOM, is the object model for Apache's Axis2 [3]. It allows data to be stored in binary format within the object model, then either encoded in Base64, or optimised and sent as an attachment at the time of sending. This means that the binary data can be processed as if it is within the XML object model, even though it is being sent externally to the SOAP envelope [54]. Due to the flexibility of the MTOM approach and the simplicity of parsing the received messages, the AXIOM / MTOM approach is to be used in the design of MPIWS. The use of attachments within the MTOM enables the data to be encoded at a different layer to the XML generation, this choice of encoding mechanism is described in the Design section (Section 4.2.3).

2.3 Workflow

There is a large amount of current research into workflow languages and the execution of these languages. One reason for this is that each language and its execution environment are often tied to the technologies that the project is built for [97]. Due to there being so many workflow tools undergoing such active research, it follows that each one excels in a slightly different area than its competitors. The areas of interest to this thesis are: message exchange patterns within the workflow, and communication of data between the services in the workflow. In this section the relevant work carried out on current workflow languages and their execution engines (or workflow managers) is reviewed. This review allows subsequent chapters to discuss the motivations for this research (Chapters 3 and 6) and to critically analyse the contributions made by this thesis (Chapter 8).

The initial impetus for this research came from the recent development of Web service based workflow tools. In relation to the work presented in this thesis, the development of workflow languages and the research surrounding their execution can be separated into two areas. The first area treats Web services as standard, self-contained processes; once the service has been invoked, there is no further communication with the service until its completion, when it *may* return a result. The second area is where the services are treated as processes that require and / or provide intermediate communication or data during the process of their execution. These services require an MPI-style of communication functionality. The remainder of this section will outline the research in these two areas.

2.3.1 Standard Web Service Based Workflow Descriptions

Most current research into workflow languages that can support Web services treat the services as self-contained processes. Once the Web service has been invoked, it will require no further unprompted input. The basic functionality of a Web service based workflow language is to describe the execution order of the component processes. These processes are deployed as Web services and are combined to create a larger composite application. Each of these processes is a separately published service with its own WSDL [22] definition. If the workflow is designed as a directed graph, with the nodes being processes and the directed edges representing the transfer of control or data, then the workflow language can model that graph in a way that can be read by humans and machines [94]. At the time of the workflow's execution, usually, these processes are controlled by a workflow manager which calls each service in turn as and when it is required. The workflow manager then passes the relevant data from the output of one service to the input of the next service.

The way that modern workflow systems deal with supporting processes deployed as Web services has a common theme across many of the different implementations. This theme is to represent the Web service based process as a local entity, this entity collates required data before invoking a service instance. Examples of this are the Kepler scientific workflow system [2], Triana [23] and Taverna [62]. The Kepler workflow system represents the services as *actors* [15]; the *actors* are responsible for acting as a service client and invoking the service. Kepler allows multiple input ports to be defined for each service, as do Triana and Taverna. When the workflow is executed, the actors in the Kepler workflow system collate all the input messages that are required for the service invocation. The *actor* then bundles these input messages into a single service request message that can be sent from an Axis client [111]. Figure 2.4 shows a Kepler Web service *actor* with multiple input ports. Triana and Taverna have similar methodologies; Triana calls these local entities *tasks* and Taverna calls them *processors*.

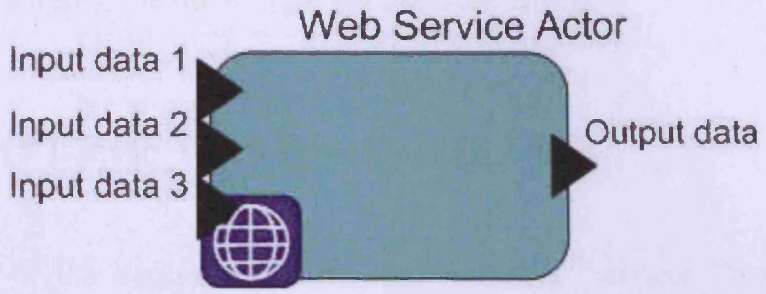


Figure 2.4: A Kepler actor with multiple inputs.

This idea of multiple input ports has been taken a step further in Glatard et al. [45] and Montagnat et. al. [77]. They suggest that the inputs and outputs of processes in the workflow could be subject to "complex data composition patterns". An example of this can be thought of as replicating the pattern of MPI's *Scatter*, where a set of output data set is split into data subsets and scattered across a number of services. Another example is MPI's *Gather*, where the input consists of a set of inputs, each from a different service. Glatard et al. [44] have implemented the MOTEUR workflow enactor, a workflow system that supports both Web services and tasks defined by executable code. One of the contributions of MOTEUR is the ability to provide data parallelism. This is where each of the data subsets (as discussed above) is used to invoke a separate instance of a service. In this scenario, the entity that represents the Web service in the MOTEUR workflow enactor has parametric ports. These parametric ports are used to represent "simultaneous processable instances of an input string" [97, pp 296]. This phrase means that the parametric input ports take the whole set of data from the complex data composition pattern and assign each of the subsets to a separate, yet potentially concurrent, invocation of the Web service.

Whilst there is recently published research expanding the communication patterns between self-contained Web services supported by workflow systems, there is no current research that explores the potential to pass data from one service directly to the next without the use of local entities such as *actors*.

2.3.2 MPI-style Workflow Descriptions

The scope of the earlier languages, such as Web Services Flow Language (WSFL) [63], was limited to the execution of one service after another, in a linear time domain. An important development in Web service flow languages was the development of flow control. This supports the ability to perform conditional processing within the realms of the workflow language. Control statements, such as *if* and *while*, allow the flow manager to choose which process should be executed next, based on the outcome of a prior service. Whilst WSFL had limited support for flow control, languages such as Business Process Execution Language for Web Services (BPEL4WS) [29] and Service Workflow Language (SWFL) [61] improve on this functionality, and Triana [23] uses additional components to control the conditional behaviour and loop constructs.

BPEL4WS stems from WSFL and XLANG [101], and defines flows in terms of Partners, Service Links, Service References, and Activities. The BPEL4WS constructs that are defined in Section 4.2 of the BPEL4WS specification [29] are the foundation of flow control.

Loops are a common construct used commonly through most coding tools. Loops that require no data interaction between iterations of the loop can be said to have no data interdependencies. For example, a loop to calculate the payroll information for each employee in a company is shown in Listing 2.1.

It can be reasoned that there is no data dependency between the required

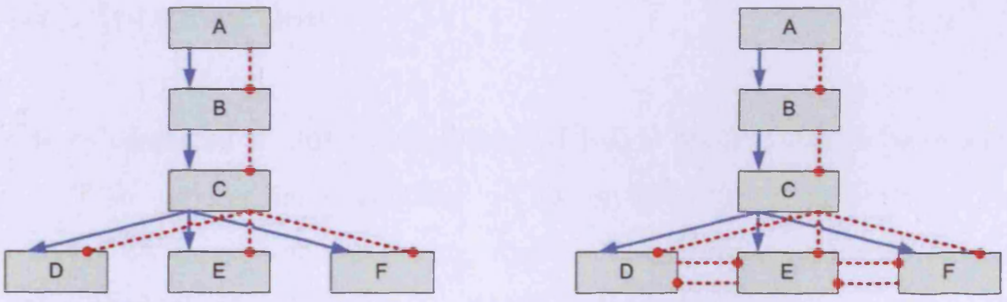
Listing 2.1: A loop example to calculate the payroll information for each employee in a company

```
For (i = 0 to number_of_employees - 1) {
  access database to get hours at basic rate for
                                            Employee i
  access database to get basic pay grade for Employee i
  calculate total basic pay
  access database to get no. of hours at overtime rate for
                                            Employee i
  access database to get overtime pay grade for
                                            Employee i
  calculate total overtime pay
  TotalPay = basic + overtime
}
```

processing in loop iteration 0 and any other iteration. This means that if a service returns the pay for a given employee number, a workflow could invoke multiple services in parallel to process the iterations of the loop. Loops are supported within BPEL4WS but the parallel invocations of services that perform loop iterations cannot be automatically managed. In order to allow for parallel invocations of loop iterations, the loop construct must be coded explicitly as a parallel invocation of separate service instances.

SWFL has been designed to allow this parallel functionality to be coded as loop statements; these loop iterations can be invoked in parallel if a *setParallel* tag is true. Abstract Grid Workflow Language is a language that is compiled by ASKALON, an enactment engine [35]. It is an XML-based workflow language that has a similar functionality to SWFL, i.e. it supports Grid workflow through a set of activities and control flow mechanisms. It also allows for the parallel processing of activities with pre and post conditions [34]. With the parallel execution of loop iterations in this manner, it is important to note that each loop iteration must be independent.

Where interdependencies exist between concurrently running service instances, the data that forms the dependencies needs to be transferred between the services involved. This data is transmitted and received during the execution time of the services. To achieve this data transfer within the workflow environment, an MPI-style message passing capability needs to be described. To illustrate parallel service interdependencies, Figure 2.5 compares a workflow with and without interdependencies. Figure 2.5 (a) shows a service composition that has used three services: D, E, F in parallel. In this composition the flow is structured, and output from a Web service process can only occur at the end of that process, and the system control passes from one service to another in an orderly manner. Figure 2.5 (b), shows additional communication between the services in the final layer of the composition. This communication transfers any interdependent data between the concurrently running Web services. It should be noted that it is only a data communication flow and not the control flow that is added.



(a) Without message passing (b) With message passing

Figure 2.5: Flow compositions *without* and *with* Message Passing. Solid lines represent control flow and dashed lines represent data flow.

Both Grid Services Flow Language (GSFL) [69] and Message Passing Flow Language (MPFL) [60] provide the functionality to describe MPI-style communication, but neither have published a workflow engine to implement the functionality. MPFL has been designed to emulate the core functionality of MPI, such as: *Send*, *Receive*, and *SendReceive*. It also emulates some of the collective communication functions: *Broadcast*, *Gather*, *Scatter*, *Reduction*, and

Barrier [60].

As will be discussed in Section 3.4, there is little published research into using Web service based communication to transfer data from one service to another concurrently executing service and what there is has no mention of collective communication functionality. MPIWS allows the functionality of compositions described by languages such as MPFL and GSFL to be implemented within a Web service infrastructure.

2.4 The Message Passing Interface and its Implementations

2.4.1 Introduction

The work presented in this thesis allows MPI-style applications to be executed over a Web services infrastructure. This involves the combination of two approaches to distributed computing, namely Web services and MPI. In order to fully appreciate the requirements of MPI-style applications and to ensure that the essential functionality is provided, an understanding of MPI and leading MPI implementations is required.

In this section, the MPI programming philosophy is introduced and current methodologies used in the implementation of leading MPI tools are reviewed. Research into improving the efficiency of collective communication algorithms and how these algorithms can be modelled is also examined.

2.4.2 Overview of MPI

Message passing communication is commonly thought of as sending data from one process to another. SOAP-based Web services do this all the time, but Web service communication is very structured; the communication generally invokes a service or returns a result from a service. In its simplest point-to-point context, MPI-style message passing is the transferral of data from one operational process to another, concurrently operational process. Generally this style of message passing is used to increase the size of the calculation that can be held in one machine, or to increase the speed of the calculation being performed. MPI's basic message passing capability is extended through the use of its collective communications functionality which allows the structured distribution or combination of distributed data to form ordered datasets. These datasets can allow distributed applications to efficiently process data in a logical order.

The Message Passing Interface (MPI) itself, is a standardised definition of a programming paradigm used to enhance the efficiency of distributed parallel computations. MPI defines the commands used to achieve various combinations of messages. These commands include: *Send*, *Receive* and *Broadcast*. MPI also defines *datatypes* that can be used to represent the information sent over these message combinations. What MPI does not define is the implementation method for any of these messages; nor does it describe the required algorithms for the collective communication operations. This is left to the MPI tool developers to choose.

2.4.3 Point-to-Point Communications

The programming framework used with MPI tools is to assign each processor an ID referred to as its rank so that functionality can then be programmed on a rank by rank basis. To give a basic understanding of the MPI-style of programming, six standard MPI commands will be described. First the set-up commands: MPI_INIT() which initialises the MPI platform and MPI_FINALISE() which shuts down the MPI platform. These commands do exactly what they suggest. The initialise command must be the first reference to MPI in a piece of code and the finalise command cleans up all MPI state within the processor. Following a finalise command, no further MPI commands can be executed. Next there are two important commands that extract information about the MPI platform that are used in most programs. They are: MPI_COMM_SIZE(MPI_COMM_WORLD), which returns the number of processors in the MPI platform, and MPI_COMM_RANK(MPI_COMM _WORLD), which returns the rank or ID of the processor that is running the program.

The *Send* command is - "SEND(object, count, datatype, destination, tag)", where: *object* is the object or message to send; *count* is the number of elements to send; *datatype* is the datatype of the *object*; *destination* is the rank of the destination processor; *tag* is a communication tag so both send and receive processors can be sure that the message is the expected one; and *comm* is the communicator object that is the current MPI communication context [37]. All these arguments allow the *Send* command to be a very versatile function.

The *Receive* command is very similar to the *Send* command: "RECV(object, count, datatype, source, tag)". The only unknown in this method call is *source*, which is the rank of the sending processor. These commands form the basics of an MPI system. Foster [37] wrote in a technical note that "With these six commands we can produce solutions to a wide range of problems".

Listing 2.2 gives a brief example of MPI style programming and shows a trivial program that sends a random number from each non-zero rank to rank zero which then prints each remotely generated random number.

Listing 2.2: A Simple MPI program

```
import mpi.*
class HelloAll{
 static public void main(String[] args){
  MPI.Init(args) ;
  int myrank = MPI.COMM_WORLD.Rank();
  int nprocs = MPI.COMM_WORLD.Size();
  int[] irecv = new int[1];
  if(myrank == 0){
   for(int toRank=1;i<nprocs;i++){
    MPI.COMM_WORLD.Recv(irecv ,irecv.length ,MPI.INT,
                                             toRank, 99);
    System.out.println("Number from process:" + toRank
                                          +" = "+irecv[0]);
   }
  }
  else{
   int toRank = 0;
   irecv[0] = (int)(10.0 * Math.random()) + 1;
   MPI.COMM_WORLD.Send(irecv , 1, toRank, MPI.INT,0,99);
  }
  MPI.Finalize();
 }
}
```

This simple program demonstrates a lot of the important concepts of basic MPI program design, including: the use of *myRank* as a rank identifier within the communication domain, and the use of the *to* and *from* ranks within the message configuration to specify the source and destination of the message.

An extension to the *Send* and *Receive* point-to-point functionality is the

SendReceive operation. This operation utilises the duplexity of the network communications in order to perform a send and a receive operation simultaneously. In the MPI standard the send and receive do not have to be to / from the same rank.

2.4.4 Collective Communications

One of the more powerful contributions of MPI to the efficiency of high performance computing is the collective communications operations. Collective communication operations transmit data throughout the communication domain, either from one to many processors, many to one processors, or even many to many processors. The two main contributions of these operations are: they can be used to create ordered data structures from distributed data, and they can improve the efficiency of the distributed communication by using the networking capabilities of the whole communication domain.

In order for MPIWS to facilitate MPI-style applications run over the Web services infrastructure, it needs to provide collective communications functionality. In this section, literature on collective communication operations, and algorithms to perform those operations is reviewed and assessed for its suitability within the MPIWS design.

Analysis of Collective Communication Operations

The analysis of the collective communications performance is of vital importance in deciding the worth of each algorithm. Pjesivac-Grbovic et al. [84] review three methodologies for theoretically analysing collective communications performance;

the Hockney method [59], the Logp method [28], and the PLogP method [67]. They conclude that all the methods have a valuable input to analysing MPI collective communications. For the purposes of this research, the Hockney method will be used due to it being the simplest and the added complexity of the other methods not being required.

The Hockney model [59] uses p – the number of processors, α – the time taken to set up the message transfer and β – the time taken to transfer each byte of the m byte long message. Using this model, the time taken to transmit a single message of m bytes from one rank to another rank is $\alpha + m\beta$.

The Broadcast Operation

The *Broadcast* operation distributes the data in the root node to all other nodes in the communication domain, so that data x at processor P_{root} becomes x at all P_j where $0 \leq j <$ *communication domain size* [8]. The simplest method of achieving this goal is to sequentially send the data from the root rank to each of the other ranks in the communication domain. The cost of this operation using the Hockney model [8] is:

$$(p-1) \times (\alpha + m\beta) \tag{2.1}$$

Alternatively, a traditional approach to the *Broadcast* operation is the binomial tree distribution [8, 103]. If the broadcast is to distribute the data to a communication domain, then the nodes in the domain can be thought of as a linear array. This array can be divided in two, the root node, the node that has the data, can then chose a node in the opposite half of the array to receive the data. This process is then recursively iterated (see Figure 2.6), in this figure the root node is node 0 and when the node array is split in two the root rank chooses node 3 to receive the data. There are now effectively two node arrays, each with a root node that contains the data, node 0 and node 3. These new arrays are then split in two again and the root ranks then choose a node in their opposite halves to recieve the data, node 0 chooses node 2 and node 3 chooses node 5. This process is now repeated with the 4 new arrays. Due to there being no contention in the sends, the cost for this *Broadcast* operation [8] is:

$$\lceil logp \rceil \times (\alpha + m\beta) \tag{2.2}$$

This method uses the processing and network capabilities of other ranks within the communication domain to increase the speed of the collective operation. The amount of data transmitted does not change but the use of the available resources has increased. There have been further improvements reported. Barnett et al. [8] describe a *Broadcast* method that uses a *Scatter* followed by a *Collect*. This method splits the message into sub messages and scatters them around sub domains then collects the data so the messages are complete at all nodes. The cost of this operation using the Hockney model [8] is:

$$(\lceil logp \rceil \times \alpha + \frac{p-1}{p} m\beta) + (p-1)\alpha + \frac{p-1}{p} m\beta \tag{2.3}$$

which can be reduced to [8]:

$$(\lceil logp \rceil + p - 1)\alpha + 2\frac{p-1}{p} m\beta \tag{2.4}$$

This algorithm is used in implementations of MPI such as MPICH [50], but for transferring messages using object serialisation, as is done in MPIWS, it introduces problems with dividing the serialised byte array for the *Scatter*.

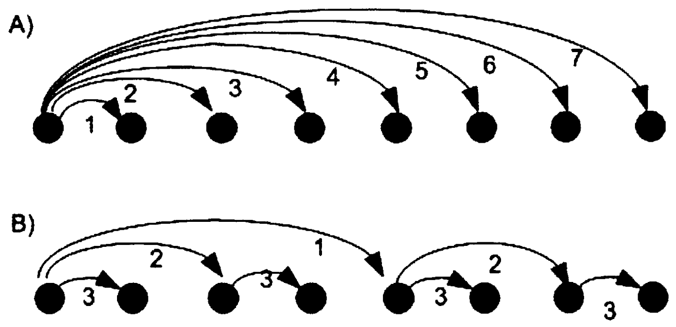


Figure 2.6: Algorithms for the Broadcast Operation: A) Serial Broadcast; B) Binomial Broadcast.

There are well known tools that use broadcast techniques for the distribution of data, for example BitTorrent [24]. BitTorrent allows clients to download a file, in a sequence of parts, from a server. Once a part has been downloaded, the client's system then makes it available for upload to other clients, this is very much in the style of the *Scatter* followed by a *Collect.* The differences in this approach is the unrestricted and unknown number of clients, and also the requirement for the clients to be actively seeking to download the content before any data transfer to that client begins. In terms of this method being used by MPIWS, the second difference mentioned means that a node could not partake in the broadcast operation until the code has reached the point where the broadcast is being executed.

The Gather Operation

The *Gather* operation gathers a set amount of data from each non-root rank to a receive buffer in the root rank. The composition of the data is arranged so that data x_j at processor P_j becomes x at P_{root} where $1 \leq j \leq$ *communication domain size* [8]. If the root rank's receive buffer is thought of as an array and each node is sending 100 integers with a root rank of 0, then after a completed operation, the first 100 integers in the root rank's buffer will be from rank 0, the second 100 integers will be from rank 1 and so on. It can be seen that the integer set from rank 0 does not have to be transmitted, yet it is collected in the final result.

Again the simplistic approach to the gather operation is to serially receive the data from the ranks in turn, giving a cost of this operation using the Hockney model as:

$$(p-1) \times (\alpha + \frac{m}{p}\beta) \tag{2.5}$$

In Equation 2.5, m is the total number of bytes collected. It has been proposed that if the combination of the data was gathered using a reversed adaption of the algorithm used for the *binomial broadcast*, then cost could be minimised [8].

$$\lceil logp \rceil \times \alpha + \frac{p-1}{p} m\beta \tag{2.6}$$

This *binomial gather* algorithm uses the intermediate ranks in the communication chain to transfer an accumulation of data to the root rank.

The Reduce Operation

The *Reduce* operation is one of the collective communication operations that uses the distributed processing capabilities of the communication domain. It adds all the values held within the ranks to the root rank, so that x_j at P_j becomes $\sum_{i=0}^{p-1} x_i$ at P_{root}. The communication structure of the *Reduce* operation is the same as the *Gather*, but in each step in the communication chain, only the combination of the data is forwarded to the onward rank. The communication cost for this binomial *Reduce* operation is:

$$\lceil logp \rceil \times (\alpha + m\beta) \tag{2.7}$$

To add the calculation cost of this operation, Rabensiefner [88] uses γ - the cost per message unit of combining 2 messages on a local processor. This gives a total time of:

$$\lceil logp \rceil \times (\alpha + m\beta + m\gamma) \tag{2.8}$$

An extension of the *Reduce* operation is the *AllReduce.* This operation concludes with the combined result at all processor ranks. This operation can be achieved by a *reduce* followed by a *Broadcast*, at a cost of:

$$\lceil logp \rceil \times (2\alpha + 2m\beta + m\gamma) \tag{2.9}$$

An alternative to this algorithm is proposed by Rabensiefner [88], called recursive doubling. In this algorithm, each service node pairs with another service node and swaps data (using a *SendReceive*), then each pair of nodes, pairs with another pair of nodes and swaps data. This process is repeated as shown in Figure 2.7. The cost given for this algorithm in Rabensiefner [88] makes the assumption that the *SendReceive* operation takes the same time as a send operation:

$$\lceil logp \rceil \times (\alpha + m\beta + m\gamma) \tag{2.10}$$

More recently the research emphasis in the field of collective operation algorithms has been to dynamically tune the message passing implementation to use different algorithms depending on the network's connection speed and the size of the messages [100, 88, 12]. This dynamic style of optimisation has not been integrated into the design of MPIWS.

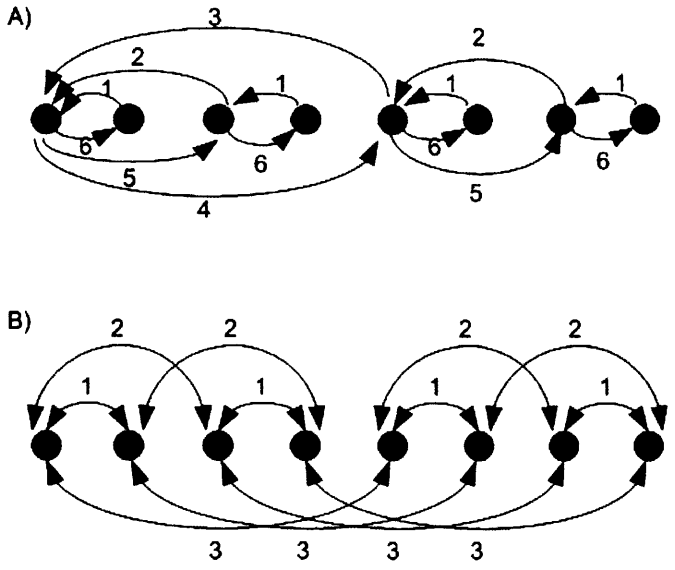


Figure 2.7: Algorithms for the AllReduce operation: A) binomial Reduce/Broadcast, B) recursive doubling.

2.4.5 Implementations of Message Passing Systems

There are many implementations of MPI [5, 41, 17]. One of the most commonly known is MPICH [50] which has been developed over the years and is now MPICH2which adheres to the MPI2 standards. The architecture of MPICH is layered [50]. The MPI layer runs on top of an Abstract Device Interface (ADI). The ADI is in essence a device driver that provides a minimally functional interface to the hardware of the underlying computer system. The MPI implementation can then build on the ADI layer to produce the complex commands necessary for the collective communications functionality.

Java based MPI

There has been a lot of research into providing a Java based implementation of MPI. These can be divided into two approaches, the pure Java implementation of the MPI standard and the Java wrapper to an underlying C implementation of the MPI standard. MPJ [21] and PJMPI [102] are two examples of a pure Java approach, both of which define a set of datatypes that can be sent over Java sockets, and use Java serialisation to create a byte array in order to send the derived datatypes. Both mpiJava [18] and Java-MPI [75] are versions of Java based MPI that use the Java Native Interface (JNI) to couple a Java implementation of MPI to an underlying C version of MPI. The main difference in the two implementations is that Java-MPI creates the wrapper to the underlying C implementation automatically using the *Java-to-C interface* (JCI), whereas the mpiJava has explicitly written wrapper code. MpiJava passes the message data to the underlying MPI in one of two ways. If the message type is a defined datatype, then the message data is transferred directly to the underlying C-based MPI. If the message type is a Java Object, then the Object is serialised to a byte array. In the MPI standard, the receiving rank needs to specify the length of buffer required to receive a message. If the message is a byte array, serialised from an array of objects, then the message length cannot be derived at the receive rank. In this case, the message length is sent as a separate message before the data message. This needs to quantify the message size, and thus limits mpiJava's ability to implement some of the collective communication algorithms that are used in the underlying C-based MPI implementation.

2.5 Chapter Summary

This chapter has reviewed the three main areas associated with this research, namely: Web services, MPI and workflow. In this chapter, the various architectures associated with Web services such as RPC and REST have been discussed and some of the current methods of increasing the efficiency of SOAP messaging have been examined. It has been found that there is no current standard for passing messages from one Web service invocation to another concurrently executing Web service invocation. Workflow languages are discussed and the development of languages that support use of MPI-style communication operations has been detailed. This provides a motivation for the development of MPIWS. These languages are analysed so that they can easily be referred to in the motivation section (Section 3.2). Finally, MPI has been discussed, including a brief overview of some of the more influential implementations, along with a brief synopsis of the development and analysis of the collective communication operations.

Chapter 3

Combining Web Services with MPI

Chapter Overview:

This chapter presents some of the important issues concerning the combination of the inherently different coding paradigms of MPI and Web Services. These issues include the coupling of the distributed MPIWS services and the question of how to evaluate the MPIWS tool. This chapter also details the related work that has attempted to achieve similar goals, including a number of Grid related MPI projects and other Web service based message passing tools.

3.1 Introduction

As seen from the previous chapter, the two programming paradigms of Web services and MPI do not easily fit together. On the one hand there is the Web service directive that each service must be decoupled to such an extent that the only view the client has of the service is the WSDL interface. On the other hand the distributed MPI ranks are so tightly-coupled that the ranks typically run the same application code. There is however current research into the development of workflow languages such as MPFL [60] and GSFL [69] which have the ability to describe the flow of data in an MPI style. Both these languages are in draft form and currently have no tools to implement them. The motivation behind this research is to provide a tool that offers Web service based workflows the opportunity to expand their available functionality and at the same time enable MPI-style applications the flexibility of connectivity, interoperability and ease of deployment.

This chapter discusses where the two programming styles can be combined, how to evaluate MPIWS, and related work that has been undertaken in this field.

3.2 The Problems Associated with Combining Web Services and MPI

The biggest problem that this research faces is the combination of such different and opposing programming paradigms. One of the defining purposes of the Web services architecture is to provide an environment for applications to work together in a loosely-coupled manner with no concern of how the other services are managed, written, or deployed. The coding style for MPI is totally opposite in its approach to distributed processes. MPI processes are tightly-coupled distributions of code which are written very much with the knowledge of how all other processes are to behave.

There are two main scenarios that benefit from the combination of MPI and Web services: the porting of MPI-style applications to run over Web services, and the use of MPIWS to improve the efficiency of Web service based workflow executions. One example of this is a service oriented implementation of the Oceans/Atmosphere model, as described in Walker and Huang [108]. This implementation uses a service to model the Ocean and a service to model the Atmosphere, at each time step of the model there must be an interchange of data between these services, therefore at each time step, there needs to be an invocation of each service. If however, the data interchange could be managed in a coordinated manner, such as by using MPI-Style messaging, the overheads of the service invocations and data persistence could be avoided [108].

The issue of the coupling can be approached in a different way for each of these scenarios.

For the case of running MPI-style applications over a set of Web services, the Web services are expected to be deployed in a more tightly-coupled fashion so that they can work together to run an MPI-style application. These deployed services become more tightly-coupled when they are invoked as part of a single MPI-style application. The benefit of running the application over the Web services architecture is to allow the use of a simplified interface to span administrative domains, firewalls, and locations by utilising the HTTP protocol. The same is true for Web service based workflows that are specifically designed to use the MPI-style of communication operations between its services.

Alternatively, for the case of using MPIWS to improve the efficiency of Web service based workflow executions, *direct messaging* between workflow services is introduced. This technique allows a service to communicate its output directly to the input of another service without it being sent via the workflow manager.

For the case of *direct messaging* within workflow management, the need for the tight coupling of services is significantly reduced. This is because the MPI-style communication only occurs at the beginning and end of the services. The data sent in this MPI-style communication replaces the application data sent in the standard Web service *Request* and *Response* messages. There is one important difference between standard Web services and the MPIWS services for *direct messaging* workflow execution. This difference is that the *direct messaging* services are unaware, at the time of deployment, where their input data will come from, and where to send the output data. The standard workflow managed service knows that the data will come from and be returned to the workflow manager (the service's client). Whereas, in the *direct messaging* services, there needs to be a certain amount of configuration at the time of invocation in order to make the service instance aware of where it fits into the whole workflow picture. This configuration will be discussed in more detail in Chapter 6.

3.3 Evaluating MPIWS for MPI-Style Applications

The evaluation of MPIWS for use in MPI-style applications is a problematic task because, at present, there is no competing Web based tool to compare its performance against. This fact in itself provides a limited evaluation as this shows that novel functionality has been achieved by providing a tool that facilitates MPI-style communication functionality over the Web service framework. There is however a need to quantify the performance of MPIWS in terms of similar existing tools that are used in this application environment.

One method of evaluating MPIWS that is proposed in this research, is to compare the performance of an existing implementation of MPI with the performance of MPIWS. Depending on the implementation chosen, this will give a relative indication of the potential of Web service based MPI implementations compared with implementations designed with optimal performance in mind.

There are many problems associated with this approach. To time a MPI application running over MPIWS and evaluate its performance against a leading implementation of MPI, such as MPICH, and conclude that the Web services approach is inferior is both obvious and of little use. Therefore it is essential to gain a useful conclusion from these experiments. The comparison of an implementation with more similar objectives can give more relevance to the results. MPI implementations such as mpiJava or MPJ could be more suitable, as the application code is written in Java to allow more platform independence and they provide a functionality that enables both Objects and the MPI defined data types to be transmitted as the data message. The performance evaluation of MPIWS against both types of data transfer in such an implementation allows a more significant conclusion to be made about the efficiency of the Web service approach.

As discussed in the background chapter (Section 2.4.5), mpiJava works by providing a Java Native Interface to an underlying implementation of MPI. Comparing MPIWS against mpiJava wrapping MPICH, allows both the evaluation of a message passing tool designed for a distributed computing environment, and also gives a fair indication of MPIWS' performance when compared to a high performance message passing implementation.

The evaluation of MPIWS against an MPI implementation gives a comparative performance evaluation, but it does not give any indication of whether the application benefits from being distributed across multiple machines. Increase in speed of an application is not the only criteria for distribution over a larger number of processors / machines. Other reasons include distributing the storage of data to allow larger calculations to be performed and allowing processing to be done at a site local to data collection. It is however still essential to ascertain that the MPIWS tool can, in respectable circumstances, provide a speed-up in the application's execution when it is provided with a greater number of resources on which to run. Therefore tests must be conducted to measure the run times of MPIWS applications for a varying number of machines that the applications are distributed across.

3.3.1 Styles of data transfer: blocking and non-blocking

Message passing systems use a variety of methodologies to send and receive messages. These are easy to describe in their simple form but the actual functionality is very implementation dependant. Additionally MPI specifications and Web service tool documentation talk about blocking / non-blocking and synchronous / asynchronous communication in slightly different manners. The aim of this section is to review the documentation from both approaches and to provide a justification of the choices of data transfer style made in the evaluation of this work.

The MPI standard [95] discuss blocking and non-blocking in terms of whether or not the data buffer in the sending task is free to be modified. Tutorials on Web services [65, 81] on the other hand, discuss the blocking functionality as waiting until communication has been completed.

Synchronous communication in both MPI and Web services relate to the sending and receiving tasks operating at the same time. However, MPI refers to the receiving task as the actual application, whereas within the MPIWS utilisation of Web service infrastructure, the receiving task refers to the buffering service on the remote server which is running in a different thread to the application task. Whilst the MPI's synchronous *send* can be posted at any time, it will only complete successfully when a matching *receive* operation has started [95]. This section assimilates the approaches so tests can be devised to evaluate MPIWS.

The MPI *send* and *receive* communication operations come in either blocking or non-blocking variants. The base functionality is non-blocking and the blocking varieties are built on top of these [58].

The MPI standard non-blocking send, *MPI_ISend*, is implementation dependant but it returns immediately with a status object that can be examined at any time to check on the progress of the send. After the *MPI_ISend* has been called, it MAY use a system allocated buffer [9] to free the application data buffer for modification as soon as the application data has been copied to the system buffer. This means that the status object reports that the *MPI_ISend* is still active until the application data is free to be modified. If there is insufficient system buffering available then the *MPI_ISend* method will use the synchronous *MPI_SSend* strategy. MPI's non-blocking buffered send *MPI_IBSend* is very similar to the standard non-blocking send except the buffer is not system allocated, it must be defined and allocated by the application programmer.

MPI non-blocking synchronous send, *MPI_ISSend*, as with *MPI_ISend*, returns immediately with a status object. However checking the progress of the *MPI_ISSend* will not reveal completion until both the application data buffer is free for modification, and the corresponding receive operation has started [9].

Blocking communication within MPI is defined by the same style of communication functions: *MPI_Send, MPI_BSend, and MPI_SSend.* MPI_Send is the standard blocking send. It does the same as *MPI_ISend*, but it does not return until the application data buffer is free to be modified. This is the same as the blocking buffered send *MPI_BSend*, the difference is, as with their non-blocking counterparts, *MPI_Send* MAY use a system allocated buffer and *MPI_BSend* uses a user allocated buffer.

These non-blocking buffered sends are analogous to the Axis2 fire-and-forget [65] method. If an object is serialised and stored within a OMElement, which is the local buffering, it can then be passed to the *fire-and-forget* method. The *fire-and-forget* method then returns to the application task after the send has been initialised. There is a slight difference in that Axis2 will ensure that a receiving host exists before returning, but it does not ensure that the receiving service exists.

MPI's synchronous send *MPI_SSend* is a blocking operation which will block until the application buffer is able to be modified and the non-local receive has been started. Axis2 provides a *sendRobust* method which, when sent a OMElement, will send this data and report any problems with the server side processing [65]. This method is similar to the *MPI_SSend* in the sense that it blocks until the remote service is actively receiving data. The difference is that the *sendRobust* method uses local buffering and, in the context of the MPIWS architecture, it is only waiting for the message to reach the service's remote-message-buffer method, NOT the service's application method.

Having reviewed the literature for both the MPI and Web service tools that are to be used in the evaluations, the mpiJava applications will use the non-blocking Send methods which use a system allocated buffer, and the MPIWS will use the *fire-and-forget* service client, which returns after contact with the receiving host address has been made. These are the most similar and most well used forms of communication within the two approaches.

3.4 Related Work

This section reviews other work that shares some of the same objectives as MPIWS, to allow Web service support for MPI-style messaging.

In the context of parallel computing and MPI, message passing is referred to as the act of cooperatively passing data between two or more separate workers or processes [51]. Thus, message passing is used in parallel scientific applications to share data between cooperating processes. It enables applications to be split into concurrently running subtasks that have data interdependencies. In a service-oriented scenario where each service runs one of the subtasks, this can be translated to the act of sending data from one executing service to another concurrently executing service. The service may be used in many applications, and therefore will be invoked many times. The problem is that when messages are being sent to a service, there must be a way of determining which invocation of the service needs to receive the message.

Currently, there is no standard for passing data from one service to another running service. Kut and Birant [70] have suggested that Web services could become a tool for parallel processing and present a model, using threads to call Web services in parallel, to allow Web services to perform parallel processing tasks. This model can be extended to allow these services to exchange data directly, which removes the need for the client to intervene every time a process transfers data [69]. This is shown in Figure 3.1. In this figure, the thin arrows indicate the request and response service client calls from the application manager

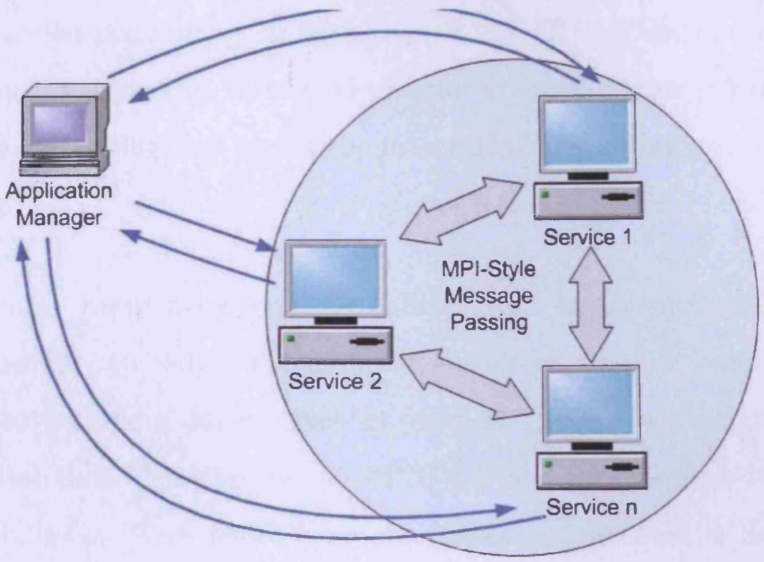


Figure 3.1: Extending the use of parallel executing services to allow MPI-Style direct message passing between concurrently executing service invocations.

and the thick arrows indicate the extension to this concept to allow message passing between the services.

Research into the use of Web services in parallel computations is also presented by Puppin et al. [86]. The results presented in this paper are upgraded results of a test from a previous paper [85] which evaluates a processor farm application. These tests are implemented both on a local cluster and across the Internet, accessing machines that MPI applications could not reach due to the network's firewall configurations. The processor farm uses a client application to invoke the computation services and collect the responses. Puppin et al. report that the Web services approach induces a 50% overhead on the MPI version of the application. It is likely that this implementation could be improved with more efficient data representation within the SOAP messaging. It is worth noting that the processor farm application is not a typical architecture of an MPI application as it does not contain direct message passing between Web service nodes.

There has been much work researching the use of Web services as a portal to

MPI and parallel computing clusters [83, 68, 33, 91]. These works recognise the advantage of using the Web service architecture as a user interface to high performance computing, but retain the use of dedicated resources to provide the computation.

There has also been work done to allow MPI to operate over the Grid infrastructure [38, 66, 80]. These implementations use the layered approach of MPICH to provide a device level interface to the Globus tool kit: "Globus communication device" in the case of MPICH-G and the "globus2 device" in the case of MPICH-G2. The device layer interfaces (as discussed in Section 2.4.5) provide the point-to-point data communication layer for MPICH. MPICH-G and G2 also provide the start-up functionality for the MPI processes. The difference between these Grid enabled MPI versions is that the communication functionality is defined by multimethod communication libraries such as Nexus [39]. This library was used in the MPICH-G version and performed the MPI communication using TCP/IP based sockets. In the MPICH-G2 implementation a bespoke communications library is used but this research concentrated more on intra-cluster communications than inter cluster communications. The Teragrid [99] project uses the MPICH-G2 implementation.

Coti et al. [27] provides another implementation of MPI operating over the Grid infrastructure. In this paper they present a framework where Grid services are used to facilitate the configuration of OpenMPI [41] nodes to work within and between administrative domains. These services act as either centralised brokers to aid in the configuration of the communication channels or as proxies to allow the forwarding of data from within one firewall to within another. Another technique that they present is the use of Traversing-TCP [89]. But in all cases, the data transmitted between the OpenMPI nodes is in the form of the native OpenMPI standard format.

There is a difference between these Grid enabled MPI implementations and the service based portals. The difference is that in the Grid enabled versions the computation can be distributed over administrative domains and locations. It is not restricted to a single cluster as it is in the portal based solutions.

In another paper, Queiroz et al. [87] presented a tool to distribute a message passing application using the Windows based desktop Grid middleware, Alchemi [74]. This approach enables the MPI based application to use the resources of idle Windows desktop machines. This approach uses the services provided by Alchemi to set up the message passing application but then they use sockets to directly implement MPI message passing.

These Grid implementations of MPI use the Grid functionality to initiate the MPI nodes, where as Krishnan et al., in their 2002 paper [69], suggest the use of Grid standards to perform direct messaging between the services. This suggestion was to use the OGSA notification ports. Neither results from this suggestion nor a tool have been published since the suggestion was made in the paper, but this could equate to the more modern use of WSRF's WS-Notification [49].

The most relevant related work to the work presented in this thesis is an additional proposal in Puppin et al. [86] which suggests an approach for mapping MPI code to be run within a Web services architecture. At the time of writing the paper the proposal was work in progress as they report:-

> `In this paper we upgrade the results of our experiments, which we presented in [sic [85]]. While we work on our MPI mapping to WS, we manually ported a MPI application (a farm-like computation) to a WS-based solution.`

Although their proposed architecture is undeveloped, it is very significant to this work, they have proposed 3 mappings to MPI primitives: *MPI_Init, MPI_Send* and *MPI_Receive.*

MPI-init invokes each Web service in the application giving "a unique ID to each of them" [86].

MPI-send is proposed to use one way communication to provide non-blocking messaging. The receiving Web service can "receive the message as soon as it is available for listening" [86].

MPI-receive "is performed simply by accepting requests from other entities" [86]. It is also proposed to enable one service to force another service to send messages by "using a blocking communication that asks for data" [86].

In Section 8.2, the paper of Puppin et al. [86] is revisited to assess the differences in their work and the work presented in this thesis.

3.5 Chapter Summary

This chapter has addressed the problems associated with combining the two opposing coding styles associated with Web services and MPI. These are mainly the coupling of the distributed computing tasks and the evaluation of these two different architectures. The chapter has presented an argument for dealing with the coupling of tasks in two different ways: retaining the tight coupling for MPI-style applications, or providing a configurable loosely-coupled service environment for the execution of service-based workflows using *direct messaging.*

The chapter then reviews previous related work that covers the integration of Web services and MPI applications and details what is considered the most important relevant work, Puppin et al. [86].

Chapter 4

Point-to-Point Communications

Chapter Overview:

In order to prove that the Web services platform is a practical and efficient environment on which to run parallel scientific applications, a tool must be developed that will facilitate message passing over Web service protocols. This tool is MPI-style Web Services (MPIWS). The first part of this research is to develop and evaluate the point-to-point communication tool that would send data from one executing service to another executing service. This chapter outlines the design and discusses the major design choices, and then provides an in-depth performance evaluation using both standard benchmark tests and internal timings analysis. These tests compare the performance of MPIWS against mpiJava [18], a leading Java implementation of MPI.

4.1 Introduction

MPIWS has been developed in order to facilitate the execution of MPI-style applications over the Web services framework. An MPIWS service is a Web service with the ability to perform direct point-to-point and collective communication with other concurrently executing MPIWS services. These services use the MPIWS tool to achieve this communication. The functionality required by MPI-style applications can be separated into two sections: point-to-point communication and collective communication. The point-to-point communication involves the communication of data from one executing service to another concurrently executing service. This chapter outlines the design and evaluation of the point-to-point functionality included in the MPIWS tool.

4.2 Point-to-Point Design

4.2.1 MPI-Style Web Services

The challenge is to design a tool that combines the tightly-coupled programming approach of MPI with the distributed, loosely-coupled architecture of SOAP based Web services. To do this, there is a need to adhere to Web service and SOAP messaging standards, whilst providing an efficient form of communication between services. MPIWS services are designed to allow for direct communication between concurrently executing Web services.

Currently, MPIWS is provided as an API to be used in the development of MPIWS services, which means that it is deployed as part of the applications deployment file. MPIWS services are deployed and invoked in much the same way as a standard Web service. At any particular endpoint, a service is deployed within the Web service container (the work presented in this thesis uses Axis2). This service can then allow access to its various methods via the Web service's SOAP interface. The deployed service is identified by its service endpoint reference, which takes the form:

```
''http://name1.cf.ac.uk:8080/axis2/services/Benchmark''
```

While one of the service's methods is invoked, an instance of that method is running. If, at the same endpoint, that method is invoked again, then there will be another instance of that method running. These instances will be referred to as the services' *method instances.*

A MPIWS service will have a method called *run()*, which is the main application method, and contains the MPI-style code. The *run()* method also initialises the service instance. A service instance is a collection of associated method instances and resources as seen in Figure 4.1. This will be discussed further in Section 4.2.2 and 4.2.3.

When MPIWS services are involved in a MPI-style distributed application, each service endpoint involved will have a service instance. The *run()* method initialises and executes the MPI-style application for that service instance. In many cases, the application within one *run()* method instance will need to communicate with other *run()* method instances that are involved with this application. To this end, the initialisation of the service must provide the details of all service endpoints involved. This collection of service instances is called the communication domain. Within this domain, the service instances are assigned a rank so they can be easily identified. The ranks are usually 0 to $(n-1)$ where n is the number of service instances in the communication domain. The rank of the

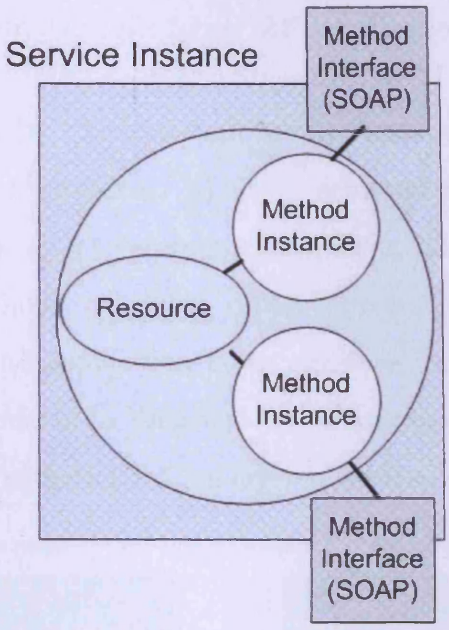


Figure 4.1: A collection of associated method instances and resources forming a service instance.

local service instance is referred to as *myRank*. Point-to-point communication is the act of sending data from one rank to another rank, and in MPI, this is achieved with the commands *Send* and *Receive*.

An MPIWS service supports a three-layer interface: a SOAP-based application layer, an internal MPI-operation layer, and a SOAP-based direct communication layer (see Figure 4.2).

The interface at the application layer is a Web service interface to allow MPIWS services to be invoked in much the same way as any other Web service. It includes only one method, *run()*, which initiates a service instance and performs the subtask that this service provides for the distributed application.

The internal MPI-operation layer provides an interface to a collection of MPI communication methods, including *Send*, *Receive* and collective communication operations such as *Broadcast*, *Gather* and *Barrier*. These methods are used within the *run()* method in a similar style to an MPI application.

The methods provided by the internal MPI-operation layer do not perform direct communications themselves. This is achieved through the use of the interface provided by the direct communication layer. The direct communication interface provides methods to allow direct communication between service endpoints. Similar to the application layer interface, the methods at the direct communication layer conform to Web service standards so that SOAP messaging is used between service endpoints. This layer is discussed in Section 4.2.3.

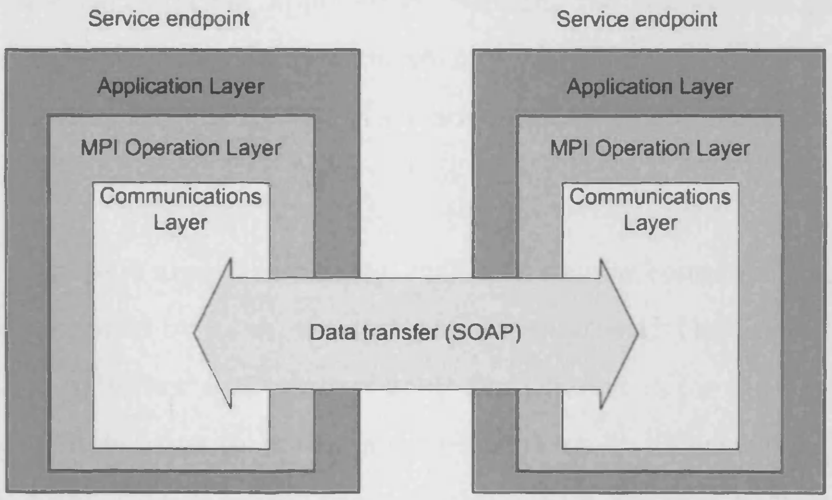


Figure 4.2: The three layer communications diagram for the MPIWS design.

4.2.2 Communication Domains

Executing a particular Web service based application requiring MPI-Style message passing involves a group of MPIWS services working together within a particular communication domain. It is possible for a MPIWS service endpoint to have multiple service instances at the same time, with each instance working for a different application and therefore, belonging to a separate communication domain. Since a service's method may have multiple instances, each working for different communication domains, a domain ID is required in order to differentiate between these communication domains. A communication domain is initialized by sending its domain ID to each service involved and assigning a rank number to each service instance to identify the particular instance within the domain. A local variable *myRank* is used to store the rank value of the service instance. Domain ID and *myRank* are used together to identify a particular service instance within a communication domain.

A communication domain is a collection of service instances working for a particular service-composite application. Within the communication domain, service instances can be identified by their *myRank* values, and communicate directly with each other by using the service endpoint references associated with the rank values.

A service instance is always associated with a particular communication domain and can be identified by its rank value stored in *myRank*. The invocation of the *run()* method initializes a service instance. The input data for the *run()* method includes: the input data to the application subtask to be executed within the method, and the binding information for the service instance to work together with other service instances within a communication domain. The binding information includes:

- A communication domain ID.

- The rank value for the particular service instance.

- A list of service endpoint references.

Each of the service endpoint references is associated with a particular rank value to allow the service to perform direct message passing with other services in the same communication domain.

An MPI-style Web service can participate in multiple applications concurrently, which means that at each service endpoint, there may be one or more service instances. Each service instance has its own data including the local data variables as well the data messages received. WS-Resource is used to provide a storage mechanism for each service instance invoked within a service. WS-Resources are defined in the WSRF specifications [30]. It provides the ability to access, maintain and manipulate persistent data values or state within a Web service. Within the WS-Resource framework, a resource is uniquely identifiable and accessible via the Web service [47]. In the case of MPIWS, a resource is used to store local data and data received from other service instances. It is created when a service instance is initiated within the *run()* method, and is associated with a communication domain ID so that only the service instance associated with the same domain ID can access and manipulate the data stored within the resource structure.

Figure 4.3 illustrates an example of a service participating in multiple communication domains. In this example there are five services deployed at endpoints A-E. Services A and B work solely for communication domain 3303 and services D and E work solely for communication domain 2020. At each of these service endpoints there is only one service instance associated with its respective communication domain ID, and one single resource associated with the service instance. The service at endpoint C has been invoked by both communication domains 3303 and 2020, so there are two service instances invoked (one for each communication domain) and two resources generated (one for each service instance).

4.2.3 Communication

In an MPIWS service, invoking the *run()* method initializes a service instance that executes the application subtask. This subtask may require MPI-style

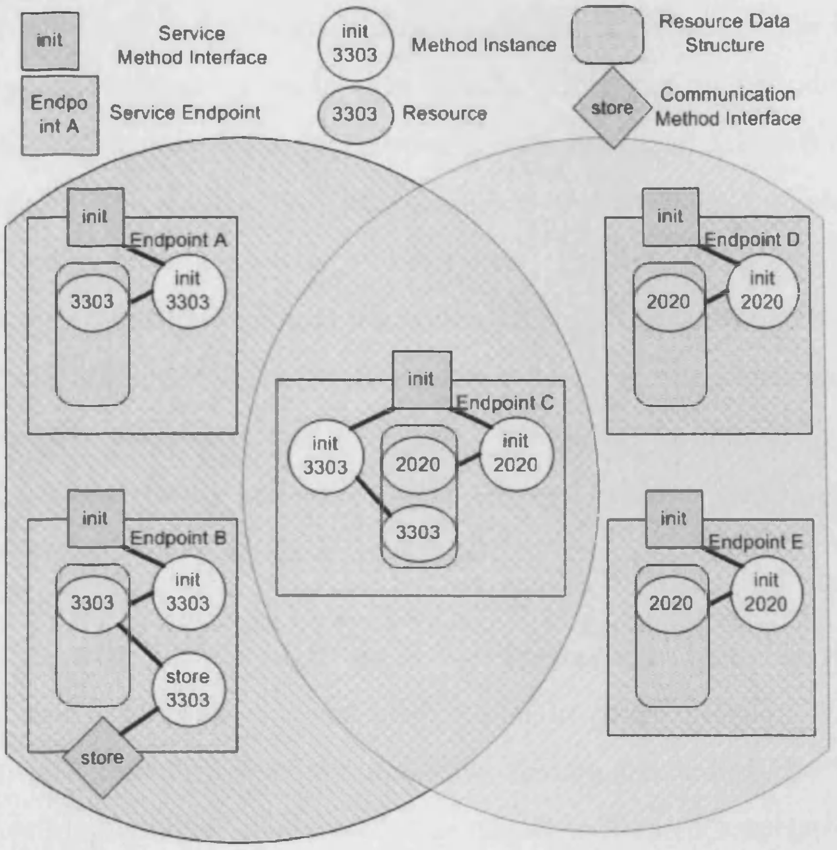


Figure 4.3: Example of services working for multiple communication domains.

communication with other service instances. The difficulty with allowing message passing between service instances is that data is normally passed into a service when a method of the service is invoked, and there is no conventional way to pass data into the method after it is invoked. However, when one service method is invoked and running, it does not stop the same or other methods from being invoked at the same service endpoint. This gives the idea that, if a *run()* method instance at one service endpoint needs to receive data from a *run()* method instance at another service endpoint, it can use a different method to receive the data and store it locally. This data must be stored in a way that it can be identified later and retrieved by the local *run()* method instance, thus creating an architecture where the sending service instance “pushes” the data to the receiving service instance. So the solution to this is to devise methods that work separately from the *run()* method, and provide direct communication support for the *run()* method by receiving and storing data locally. In order to provide support for point-to-point communication between service instances, MPIWS offers the *store()* method. This method performs the function of receiving data and storing it in a local data structure within the resource. The data messages are always associated with a particular communication domain ID and can be identified by the sender's rank value as well as its sequential order. A received message is stored into the resource associated with the same communication domain ID that the message is associated with, and can only be retrieved by the service *run()* method instance associated with the same domain ID.

Within a resource, there is a message buffer structure, where each received message is stored to await retrieval from the main *run()* method. Within this buffer, for each rank in the communication domain (excluding the *myRank*), there is a sub-layer buffer which stores the messages from its associated rank in the order in which they were sent.

Figure 4.4 shows an example of a *Send()* operation scenario between two MPIWS services: A and B. A communication domain has been initiated with the communication domain ID equal to 3303. Service A is to send a message to service B within the communication domain. In this example, two service instances have been invoked within communication domain 3303: rank 2 instance and rank 3 instance. The rank 2 instance, running at service endpoint A, is sending a message to rank 3 instance which is running at service endpoint B. To do this, the rank 2 instance invokes the *Send()* method, which is an internal MPI-communication method, with the message data as the input. The *Send()* method calls the *store()* method at endpoint B and passes the message data as its input data. Since the *store()* method is a standard Web service operation, the messages it receives are standard SOAP messages. Each SOAP message received includes,

– the message data required by the receiving service instance, rank 3.

– the message sequence number, #5.

– the communication domain ID, 3303.

– the *fromRank*, the rank value of the sending service instance, rank 2.

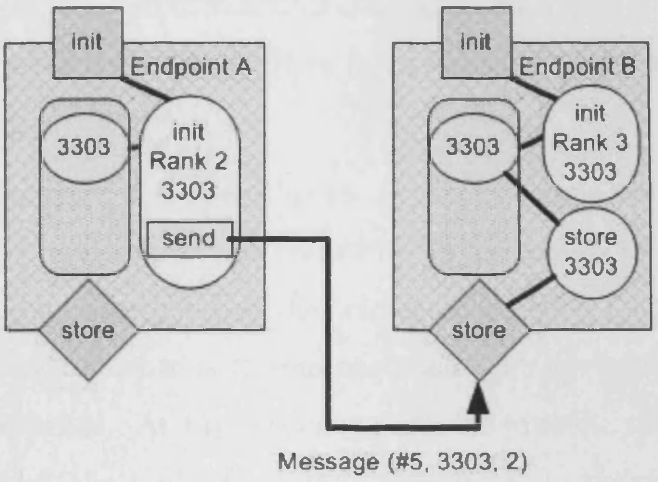


Figure 4.4: MPI-style Web services point-to-point send architecture.

The *store()* method at endpoint B receives the SOAP message, and stores the message data into the particular buffer that is associated with rank 2 and located in the resource associated with domain ID 3303. The stored message data can be retrieved later by invoking the *receive()* method, an internal MPI-communication layer method, in the rank 3 instance at endpoint B, as illustrated in Figure 4.5.

In the Web service implementation there are several factors that may affect the sequence in which messages are received, including the multithreading of send and receive mechanisms within the SOAP container. The messages may arrive in a different order from the order in which they are sent. In the MPIWS implementation the message that the receiving service instance requests next depends on the order in which the messages were sent. Thus, it is necessary to

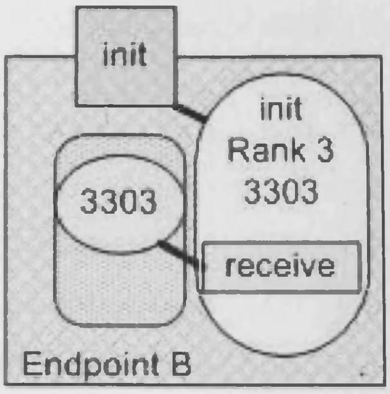


Figure 4.5: MPI-style Web services point-to-point receive architecture.

record the sending order of the messages so that they can be identified later when they arrive at the receiving service endpoint. To this end, a sequence number is attached to each message to record the transmission order. Each time a message is sent, the sequence number is incremented and the new value is attached and sent with that message. At the receiving service endpoint, the *store()* method uses the *fromRank*, the rank of the sending service, to decide which message buffer the message should be stored in, and the sequence number attached to the message to decide the order of the message to be stored in the message buffer. The service instance on the receiving endpoint can retrieve the message from the corresponding message buffer. In the case that a message has not been stored yet but a subsequent message has been stored, the service instance has to wait until the prior message has completed storage in order to retrieve the correct message.

Message Encoding

The communication between MPI-Style Web services is designed with a two-layer protocol stack: an upper layer that has been described as the direct-communication layer in Section 4.2.1, which allows the use of communication methods via the standard SOAP communication protocols, and a lower layer that deals with the encoding of the message data during its transmission.

In order to evaluate the possibilities for the lower data encoding layer, the data type must first be discussed. In related work, there have been various methods of encoding for the transmitted data. In MPICH, the transmitted data is defined as a specific *datatype* and according to the MPI specifications the implementing language sends the contents of the memory from a pointer marking the beginning of the data array, to the number of items sent times, the *datatype* size. This is a very efficient method as there is minimal data stored, and minimal time spent encoding the data. The mpiJava allows both: a native interface to the MPICH transmission methods for the primitive data types; and a method of encoding Java Objects to a byte array and then allowing the native MPICH to transmit that data. This Object to byte array conversion is also used in Queiroz [87] as their method of transmitting objects. The tool they have developed has chosen not to provide a direct *datatype* transmission mechanism. Pupin [86] has used the XML structure to transmit elements of the data array, which creates a large data size overhead as all elements need to be converted to XML compatible format.

In order to avoid the large overhead that would be created by the conversion of the message data to XML format, MPIWS allows the message data to be serialised to a byte array and added to the SOAP message as an attachment Message Transmission Optimisation Mechanism provided in the Axis2 tool set. The MTOM allows the data to be extracted as an element as the SOAP message is parsed. Currently the Java serialisation mechanism is used to serialise the messages which are stored as Java objects. Although this method does not provide the language independence that MPIWS strives for, it allows the tool to be evaluated fairly against competing MPI implementations such as mpiJava or MPJ. As further work this serialisation could be modified to be more in line with the MPI standard where simple data types are defined by the MPIWS and complex data types are predefined by the application programmer.

Fire-and-Forget Invocation Approach

There has been discussion in Section 3.3.1 about the available styles of Web service client. The use of a WS-Resource to provide a message buffering service for message passing encourages MPIWS to adopt the asynchronous *fire-and-forget*[65] service client model which is supported in Apache Axis2, to send the SOAP messages. The *fire-and-forget* client method returns immediately after the existence of the receiving host is confirmed. It can provide increased performance over the *sendReceive* client model [65], which expects a response message before the method returns, and the *sendRobust* client model [65], which sends data and returns when the processing at the server is complete with either the results or information to report any problems [65].

The use of the *fire-and-forget* service client model means that the MPIWS needs to rely on the network protocols to provide its reliability. This is due to there being no form of acknowledgement that the data has been received. There are additional standards to enhance the reliability of Web service messaging but this issue is discussed in the further work chapter, Chapter 9.

4.2.4 MPIWS Messages

Having discussed the architecture of MPIWS, this section can now look at the SOAP messages that will be needed for the invocation of the *run()* method and the *store()* method. As discussed in Section 2.2.1, messages define the Web service application's interface. The initialisation of the main application task is achieved by invoking the *run()* method, as discussed in Section 4.2.1 and the *run()* method is also used to initialise the MPIWS service. To this end, the SOAP message to invoke the service's *run()* method can logically be separated into two main elements: the MPIWS data, and the application data. The MPIWS data schema remains the same for all MPIWS services, no matter what the main application is, but the schema for the application data is designed for each different application service. Listing 4.1 shows an example of the body of a *run()* method invocation for the matrix multiplication service.

Listing 4.1: An outline example of the SOAP message body to call a MPIWS matrix multiplication service

```
<soapenv:Body>
 <mpi_ws:run xmlns:mpi_ws=....">
  <mpi_ws:mpi_wsData>
   <mpi_ws:eprList mpi_ws:eprLength="8">
    <mpi_ws:epr>http://cslx01..../CollectiveCommsTest
                                       </mpws:epr>
    <mpi_ws:epr>http://cslx02..../CollectiveCommsTest
                                       </mpws:epr>
    .
    .
    <mpi_ws:epr>http://cslx08..../CollectiveCommsTest
                                       </mpws:epr>
   </mpi_ws:eprList>
   <mpi_ws:rank>0</mpws:rank>
   <mpi_ws:iD>445</mpws:iD>
   <mpi_ws:reportingMode>INFO</mpws:reportingMode>
  </mpi_ws:mpi_wsData>
  <mpi_ws:appData>
   <app:matrixSize xmlns:app= http://...>200
                                  </app:matrixSize>
   <app:loopIterations>5</app:loopIterations>
  </mpi_ws:appData>
 </mpi_ws:run>
</soapenv:Body>
```

The two main elements are *mpi_wsData* and *appData*. In the *mpi_wsData* element the endpoint references are shown, along with the service invocation rank and the communication domain ID. In the *appData* element, the matrix size is given to enable the service to initialise the matrix and the number of loop iterations is given so the service knows how many times to run the calculation.

Listing 4.2 shows the relevant parts of a store SOAP message from the same matrix multiplication service. The message is sending a serialised object that contains part of a matrix that is to be multiplied.

Listing 4.2: An example SOAP message to send a message from one rank to another

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope ................... >
 <soapenv:Header>
     .
     .
  </soapenv:Header>
  <soapenv:Body>
   <mpi_ws:store xmlns:mpi_ws=...." >
    <mpi_ws:data>
     <mpi_ws:msgNo>5</MPWS:msgNo>
     <xop:Include href="cid:1.urn:uuid:3855...69321
                                    @apache.org"xmlns:xop=
                                   "http://www.w3.org/2004
                                       /08/xop/include" />
    </mpi_ws:data>
    <mpi_ws:id>445</MPWS:id>
    <MPWS:msgTag>1</MPWS:msgTag>
   </mpi_ws:store>
  </soapenv:Body>
  .
</soapenv:Envelope>
--MIMEBoundaryurn_uuid_3855...69319 content-type:
     application/octet-streamcontent-transfer-encoding:
                                          binarycontent-id:
<1.urn:uuid:3855...69321@apache.org>??
```

In Listing 4.2 the *data* element contains a reference to the attached message part. This attachment will be processed by the SOAP engine upon arrival at the destination endpoint reference and the *store()* method will be able to treat it as a standard element. The *msgTag* element within the *store* element is used to indicate the *fromRank.*

4.2.5 SendReceive

The *SendReceive* operation is a very simple combination of a *Send* from one service node to a second service node, whilst at the same time a *Receive* from that second service node is taking place. This operation uses the duplexity of the communications network. MPIWS implements a *SendReceive* operation by using threads to perform each of the basic point-to-point operations.

4.3 The Evaluation

4.3.1 The Purpose of the Evaluation

The purpose of evaluating MPI-style services is to show that the Web service based architecture can perform acceptably when compared to other message passing tools when running over a non-dedicated network. To this end the results of a collection of tests using the MPIWS tool are evaluated against a leading Java implementation of MPI, mpiJava [18]. The MPIWS application management is controlled by bespoke code for each individual application as MPIWS's integration with workflow management tools is beyond the scope of this thesis. MpiJava is a non-Web-service-based version of MPI that uses Java Native Interface (JNI) to provide a Java interface to MPICH, and allows MPI applications to work in a more loosely-coupled distributed environment. Although it is not Web service based, mpiJava runs in a more heterogeneous distributed environment than some other implementations, due to the Java application programs being platform independent, and is thus broadly similar to the MPI-style Web services presented in this thesis. There has also been research into the performance of mpiJava [6, 20], so by evaluating against mpiJava, an idea of how the Web services architecture will perform against other approaches can be gained. These arguments make mpiJava a good choice to compare against MPIWS.

As discussed in Section 2.4.5, there are two methods of transmitting data in mpiJava. The first method is to use the primitive MPI datatypes and directly send them via the native interface to MPICH. The second method uses Java Objects and converts the Object to a Byte array, and then sends this array with the datatype *Byte* via the native interface to MPICH. Although there is research providing performance evaluation of the two mpiJava transmission styles [20], this research is from 1999 and it suggests the performance is hindered by the Java serialisation mechanism provided within the JDK. The Java serialisation mechanism has been revised since that time so it is important that where possible, the results of both these mpiJava data transmission methods are presented. These results will be presented along with the results of the MPIWS tool. It is also important to present both sets of mpiJava results in order to directly compare the MPIWS tool with both an implementation of MPI running at optimum performance and an Object passing tool using a leading underlying MPI implementation.

The evaluation tests focus mainly on the speed aspect of the communication implementations; MPIWS services are tested against mpiJava. Many benchmark suites have been devised and put forward as the definitive parallel computing benchmarks [73, 64], and many of these are designed to test the underlying hardware or the collective communications features of the message-passing tools. In this work, tests have been specifically chosen that target the performance of the message passing tools.

In the evaluation tests, all the MPIWS services use Apache AXIS 2.1.2 and are hosted in a Tomcat 5.5.20 application server. The mpiJava API that has been used is mpiJava V1.2 which wraps MPICH 1.2.6. All code was written in Java 1.6.0. The evaluation tests are undertaken on a public network of university machines, all of which are prone to unforeseen activities. The tests were done during low usage hours to reduce inconsistencies. All graphs show minimum timings gained from repeated tests for each message size to reduce the impact of the network on the results. This technique is recommended in Gropp and Lusk's paper on Reproducible Measurements of MPI Performance Measurements [52]. In their paper it also suggests that the tests for each message size are carried out non-consecutively as any perturbations in the timings caused by network or processor inconsistencies may last many milliseconds [52].

The Linux machines used for the tests have twin Intel Pentium 4, 2.8GHz processors. In order to eliminate the possible discrepancies in thread handling within mpiJava and the Tomcat deployment, only one processor is used on each machine. This is achieved with processor affiliation settings.

4.3.2 The PingPong Test

The PingPong test is one of the most popular tests that is used to provide a simple bandwidth and latency test for point-to-point communications. Getov et al. [43] used a number of variations of the PingPong test to compare the

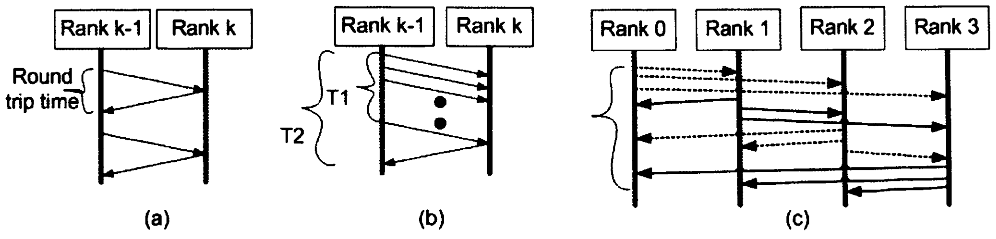


Figure 4.6: Scenarios of PingPong, Ping*Pong and Matrix Multiplication tests. An arrow represents a portion of the matrix being sent from one processor to another.

performance of MPI and java-MPI. Foster and Karonis [38] also used the test to evaluate MPICH-G, a grid-enabled version of MPI.

The standard PingPong test requires an even number (n) of service instances within a communication domain, with each of the instances paired with another. For this implementation of the test, where $0 \leq rank \leq n-1$, if the service instance's rank i is even, then its partner rank is $i+1$ and if it is odd then the partner rank is $i-1$. Within each pair of service instances, a message is sent from one rank to the other, and is then sent back again. The scenario of the PingPong test is illustrated in Figure 4.6(a). In this test, the round-trip time of the message travelling from one processor to another and back again is measured. The data transmitted in these tests consists of an array of Java *doubles* which is treated as an object for both the Object transmission tests and as a raw MPI.DOUBLE array for the datatype transmission test. The size of the array is varied and plotted against transmission time.

The results of the PingPong test are displayed graphically in Figure 4.7 with the message size in the range of 0 to 5Mbytes. To be able to see clearly the difference between the two MPI implementations when the size of message is small, the results of the PingPong test with message size in the range of 0 to 400Kbytes are displayed in Figure 4.8 with a larger scale.

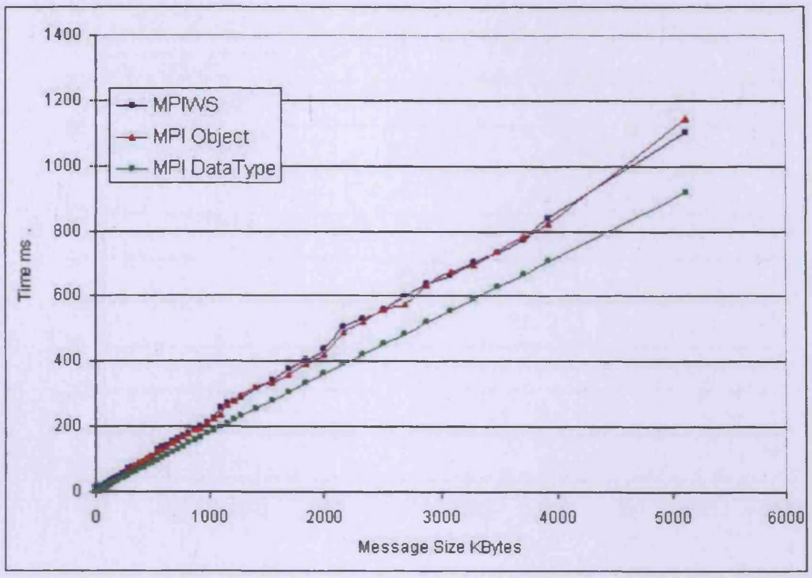


Figure 4.7: PingPong test results (Message size 0 - 5MByte).

These graphs show that there is a substantial difference in performance between the two mpiJava transmission methods: the datatype transmission that directly passes the message to the underlying MPICH clearly outperforms the Object transmission. Again, at a superficial glance, the difference between the mpiJava Object PingPong test and the MPIWS PingPong test, at higher message sizes, appears to be minimal. To make a more accurate judgement on this data, previous authors [43] have used linear regression techniques to statistically analyse the data. Using the difference of least squares approach, both the latency and the bandwidth of the systems can be estimated using the mean message size $\bar{M}$, the standard deviation of the message size SD_M, mean timings $\bar{T}$ and the standard deviation of the timings SD_T. If the predicted line equation for the graph is of the form:

$$Y = a + bX$$

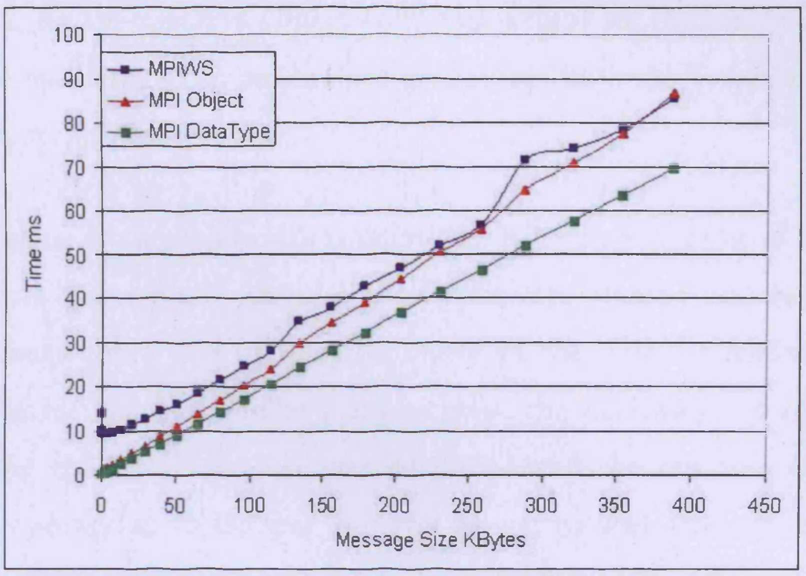


Figure 4.8: PingPong test results (Message size 0 - 400KByte).

The slope of the of the graph is b:

$$b = r\frac{SD_T}{SD_M} \tag{4.1}$$

where r is the Pearson product-moment correlation coefficient between message size and time, this can be calculated by:

$$r = \frac{1}{n-1}\sum_{i=1}^{n}(\frac{M_i - \bar{M}}{SD_X})(\frac{T_i - \bar{T}}{SD_T}) \tag{4.2}$$

and the intersection a of the line with the Y axis can be obtained by the equation:

$$a = \bar{T} - \bar{M} \tag{4.3}$$

The PingPong graphs show the data for two message transmissions, the Ping and the Pong. This means the calculated latency times can be halved. This gives the latencies (using data from the smaller message sizes) of 3.94ms for the MPIWS test, 0.27ms for the mpiJava Object test, and 0.14ms for the mpiJava datatype test. These estimations can be obtained graphically from the Y-axis intersections of the data graphs in Figure 4.8.

The calculation of the bandwidth estimations(defined an amount of application message data per second) has also been split into smaller and larger values of the message sizes, due to the irregularity of the object transfer data sets above 1Mbytes. For the smaller message sizes, the bandwidth of the MPIWS tool running the PingPong test was 80.1Mbps and the mpiJava Object test was slightly slower at 75.4Mbps. However as can be seen from the graph, the mpiJava datatypes approach is the faster method for the PingPong test with its bandwidth estimated at 89.7Mbps. For the larger message sizes, the bandwidth of the MPIWS tests is 75.3Mbps, and the mpiJava Objects test only just slower at 74.1Mbps, whilst again the mpiJava datatypes approach is appreciably faster as expected (88.8Mbps). The calculated estimates can be seen in table form in Table 4.1.

	Small Message Sizes		large Message Sizes	
Test	Latency	Bandwidth	Latency	Bandwidth
PingPong Test				
MPIWS	3.94ms	80.2Mbps	-	75.3Mbps
mpiJava Objects	0.27ms	75.4Mbps	-	74.1Mbps
mpiJava Datatypes	0.14ms	89.7Mbps	-	88.8Mbps
Ping*Pong Test				
MPIWS	2.09ms	91.1Mbps	-	77.1Mbps
mpiJava Objects	-0.05ms	82.7Mbps	-	78.1Mbps
mpiJava Datatypes	0.00ms	90.4Mbps	-	90.6Mbps

Table 4.1: Table of Latencies and Bandwidths for Ping Pong and Ping*Pong tests (Note:- These are statistical estimations)

4.3.3 The Ping*Pong Test

The Ping*Pong test [43] is a variation of the PingPong test, which involves an even number of service instances each of which are paired with another. In this case, within each pair group, one service instance sends multiple messages to the other service instance in the same pair group, and then the receiving instance returns a single message. Figure 4.6(b) shows the scenario of the Ping*Pong test. This test differentiates between the *intra* message pipeline effect, where the message is broken into smaller parts by the system and processed through a pipeline to speed up the communication, and the *inter* message pipeline effect, where the system does not have to wait for one message to complete its transfer before starting to process the next message [43]. The Ping*Pong test shows a more realistic view of the system's performance, as it emulates many real applications of message passing (such as matrix multiplication).

In this test the ping message is sent 10 times and then the same message is returned as the Pong. This means that there are 11 messages for the bandwidth calculations.

The results for the Ping*Pong test are shown in Figures 4.9 and 4.10. Figure 4.9 shows the results when message size is in the range of 0 to 41Mbits (approximately 5.1Mbytes) and Figure 4.10 shows the results on a larger scale when the message size is in the range of 0 to 1.4Mbits (approximately 175Kytes).

The graph in Figure 4.10 distinctly shows the difference in performance for the smaller message sizes. The MPIWS test can be seen to have a high latency, but the bandwidth does not seem to be that different to the mpiJava approaches. When statistical analysis is performed on the data set using the least squares approach (as in Section 4.3.2) the estimated latency for the MPIWS Ping*Pong

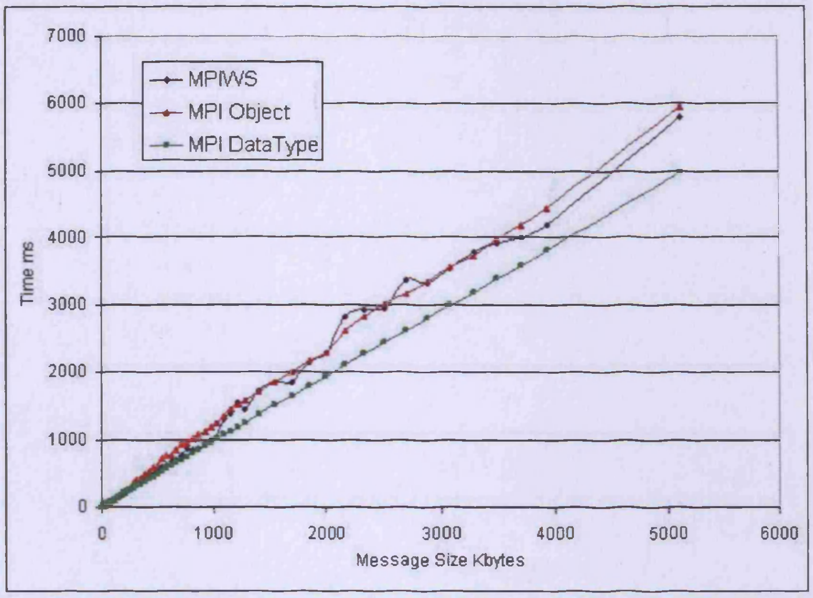


Figure 4.9: Ping*Pong test results (Message size 0 - 5MByte).

test is 23ms and the bandwidth is estimated at 91.2Mbps (for these results the message size values under 1Kbyte were ignored because the overhead of the SOAP messages will skew the latency). For the mpiJava *object* test, both the graph and the statistical analysis of the smaller message sizes show that this method provides a slower bandwidth than the MPIWS that provided 82.7Mbps, but the latency is much smaller at -0.5ms (remember that this is a statistical estimation). An unexpected result was that, mpiJava's test sending datatypes provided a reduced bandwidth of 90.1Mbps for small message sizes but due to the negligible latency (-0.023ms), proved to be faster throughout the whole range of message sizes. As the message sizes increase, the relationship between the bandwidths becomes more in line with the PingPong tests with the MPIWS and mpiJava object tests providing bandwidths of 77.1Mbps and 78.0Mbps, respectively, and the mpiJava datatypes test providing a bandwidth of 90.6Mbps. A table of bandwidth and latency estimations can be seen in Table 4.1.

The results of the PingPong and Ping*Pong tests show that the MPIWS tool

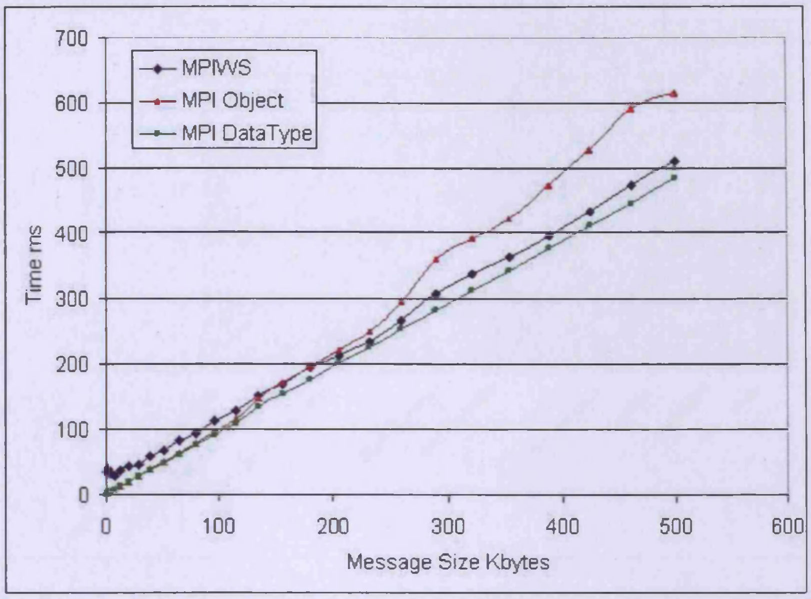


Figure 4.10: Ping*Pong test results (Message size 0 - 500KByte).

does have a high latency when transmitting point-to-point data, but they also show that the bandwidth of the MPIWS tool is quite comparable with the mpiJava tool. This is not that surprising as the underlying transmission protocols are the same and only the HTTP packet headers are added for the MPIWS data transmissions. The latency is somewhat more of a problem. The high latency creates a significant performance constraint when transmitting the smaller messages, but as the message sizes increase, the proportion of latency overhead to data transmission decreases so that for message sizes at around 150Kbytes for the Ping*Pong and 250Kbytes for the PingPong tests, the latency is absorbed in the running time to allow the MPIWS and the mpiJava passing serialised objects to run with equivalent timings.

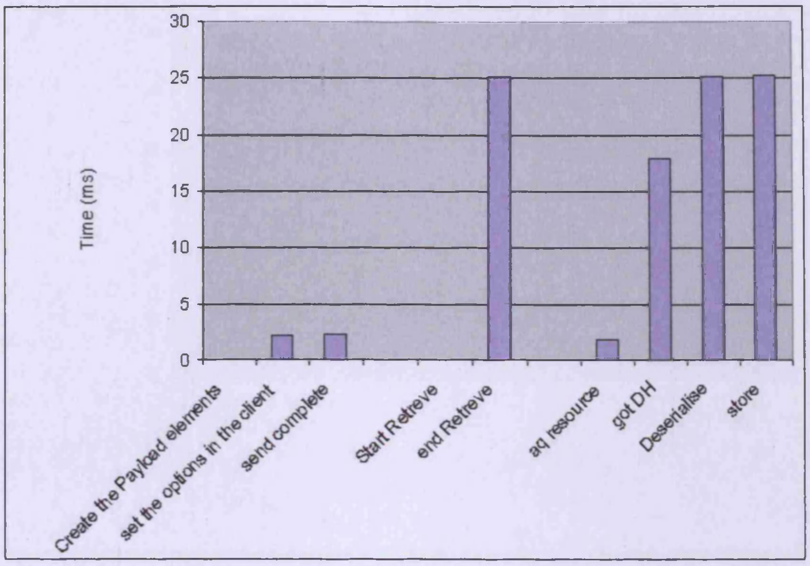


Figure 4.11: Internal timings from the *Send*, *Receive* and *store* operations using the *fire-and-forget* Service Client.

4.3.4 The Internal Timings

The evaluation of the MPIWS tool can be enhanced by analysing the internal timings of the message passing data transfer. To this end a simple test, similar to the PingPong test, has been performed that sends data from one service instance to another and then back again. In this test, modified versions of the *Send*, *store* and *Receive* methods are used, and timing data at various points in the execution of the operations are captured. By using these methods an understanding of the transfer process can be gained and theoretical design decisions can be justified.

The two sets of results presented show the internal timings from the *Send*, *store* and *Receive* operations. Figure 4.11 shows the results of the data communication using a *fire-and-forget* service client and Figure 4.12 shows the results of the communication using the *sendRobust* service client.

The *Send* and *Receive* operations are performed on separate computers. It is not

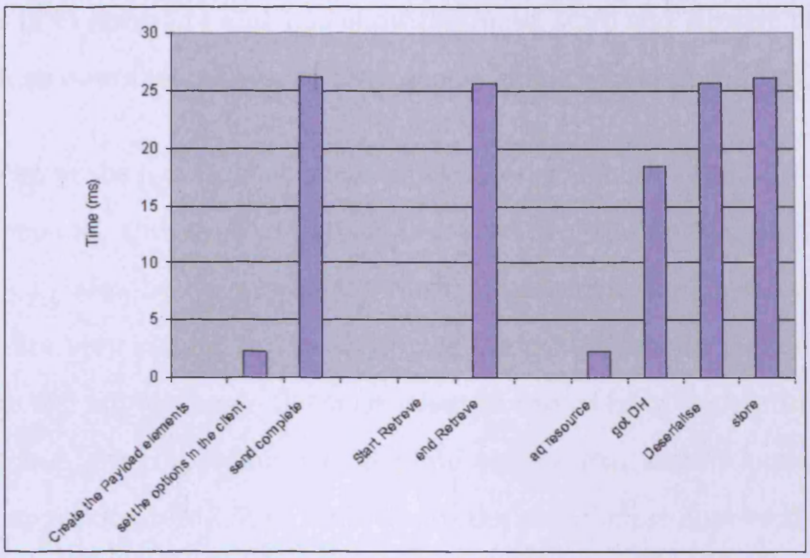


Figure 4.12: Internal timings from the *Send*, *Receive* and *store* operations using the *sendRobust* Service Client.

possible to precisely time both operations with a view to giving timing information of the one operation relative to the other. This is because the system clocks will differ slightly on each machine. The internal timings test performs a PingPong communication with all timing data being collected at the initiating rank. This means that the data communication that is timed for the *Send* operation is not the same data communication that is timed for the *Receive* operation. The *Send* timing is for the *Ping* and the *Receive* timing is for the *Pong*.

In order to present the data with relative timing information an assumption has been made: the *Send* and *Receive* in one direction of the PingPong will have the same timings as the *Send* and *Receive* in the other direction. It follows from this assumption, that the data communication in one direction will take half the round trip time. If the *Receive* timing data from the *Pong* communication is shown with half the round trip time removed, the data can be used to represent the *Ping's Receive* timing data.

The graphs in Figures 4.11 and 4.12 show the *Send*, *store* and *Receive* timing data normalised, as described above, to represent a single communication operation.

It can be seen in the figures that both approaches spend little time setting up the payload elements, and approximately 2.5ms setting the options in the service client. It can also be seen that for both approaches, the *Receive* and *store* operations are very similar in the timing of internal operations. The difference between the two approaches is the time taken to return from the sending method. In the *fire-and-forget* approach the sending service can start completing other tasks after approximately 2.5ms; however, in the *sendRobust* approach the service must wait for 25ms. These tests show that the choice of using the *fire-and-forget* service client will provide a benefit for the performance of MPIWS applications.

4.3.5 SendReceive

Using the MPI standard, the exchange of data between two ranks must be controlled very carefully. If two ranks exchange data with each other, both using a *Send* followed by a *Receive*, then a deadlock situation could arise [95, pp62]. This deadlock arises because both ranks are waiting for the other to receive its data. The MPI standard allows for two solutions to this problem. Firstly, one rank sends the data then receives the data, and the other rank receives the data then sends the data. Secondly, both ranks use the *SendReceive* operation. Using the MPIWS implementation, this deadlock will not arise, because the *store* method is executed in a separately invoked service thread to the *Receive* method. This means that the *Send* from a local rank can be completed without the remote *Receive* being called, thus allowing the local rank to continue with its *Receive* operation.

The *SendReceive* operation in the MPIWS API is implemented using the standard point-to-point operations, however, the two tasks are executed in separate threads. Although the operations already run using threads, the extra layer of threading at the MPI layer provides a slightly better performance. This allows the operating system to interleave the processing and data transfer of the two tasks to achieve a better overall performance. The point to note is that although the tasks are threaded, there will only be one processor available in the evaluation tests. In the context of a network of work stations connected together with Ethernet networking (100Base-T) then the connection to the ranks is full duplex, i.e. there is the potential to both send and receive 100Mbits of information at the same time. This makes the *SendReceive* operation very economical when compared to the *Send* followed by a *Receive.*

SendReceive Evaluation

The MPIWS *SendReceive* operation has been evaluated against the alternative MPI standard approach of one rank sending then receiving, and the other rank receiving then sending the data.

Figures 4.13 and 4.14 show the timings of the MPIWS *SendReceive* operation and the MPIWS *Send* followed by *Receive.* From these graphs it can be seen that the combined *SendReceive* operation runs faster than the serial *Send* followed by *Receive.* This is especially true for the smaller message sizes ($<$ 1Mbyte). This is put down to the caching capabilities within the network cards, and use of only one processor core to execute the multiple threads.

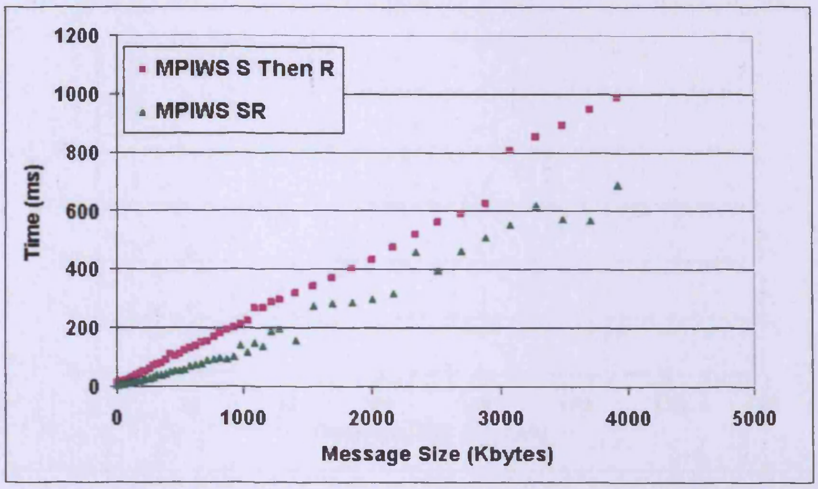


Figure 4.13: Timings of the MPIWS *SendReceive* operation and the MPIWS *Send* followed by *Receive* for message sizes between 0 and 4Mbytes

4.3.6 The Matrix Multiplication Test

A further test is performed based on a real application, a one dimensionally blocked parallel matrix multiplication, multiplying 2 N by N matrices of *doubles*. This application is a simple parallelised version of the matrix multiplication problem. The communications for the matrix multiplication application are shown in Figure 4.6(c). It is important to note that although the sequence of the *Send* operations is fixed, both the sending and the receiving processors do not have to wait until the *Send* or *Receive* operations complete before they process the next message.

In the matrix multiplication application test, the multiplication calculations are extremely time-consuming. Together with the variances in the processors' utilisation at the time of testing, it could dilute the performance of the communications. The calculation part of the application has therefore been omitted, and only the communication results of the application have been presented.

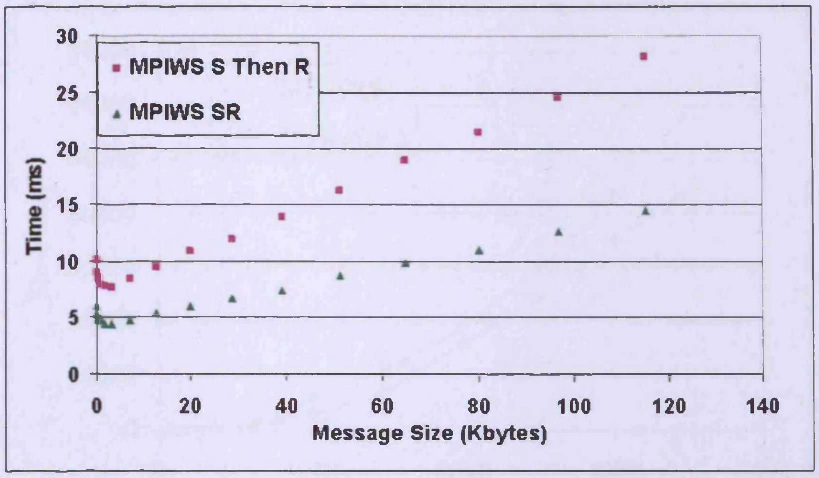


Figure 4.14: Timings of the MPIWS *SendReceive* operation and the MPIWS *Send* followed by *Receive* for message sizes between 0 and 120Kbytes

The results of the matrix multiplication test running over 8 processors and using point-to-point communication operations are shown in Figures. 4.15 and 4.16.

Figure 4.15 shows the results when the matrix size N is in the larger range of 0 to 3500 while Figure 4.16 shows the results when N is in the smaller range of 0 to 600. According to the results, when the size of the matrix is large enough, in this case 270×270, the application runs faster using MPIWS than using mpiJava. Tests over different numbers of processors have also been conducted and all the results came out consistently. The results shows clear agreement with the Ping*Pong test. The matrix multiplication requires consecutive *Sends* to distribute the matrix over processors. The combination of *fire-and-forget* sends with message buffering at the receiving processor have a good inter-message pipeline effect on the MPIWS which is demonstrated in the Ping*Pong Test, and explains the test results showed in Figures 4.15 and 4.16.

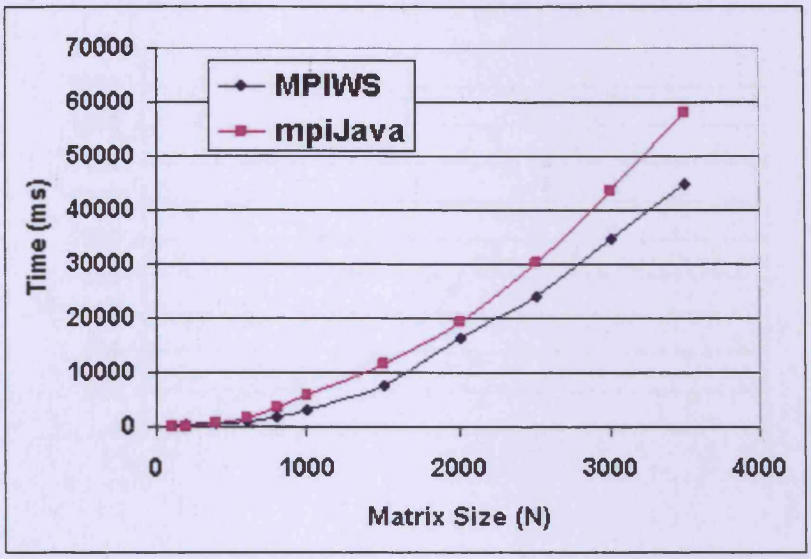


Figure 4.15: Results of matrix multiplication test using point-to-point communication with 8 processors (N = 0 - 3500).

4.4 Chapter Summary

This chapter presents the design of MPIWS for point-to-point communications. It gives a detailed picture of the tool's architecture and use of resources to provide state and session within the application's communication domain. This chapter addresses how MPIWS is to be deployed as an extension to the application's deployment files, and how the application designer is to use the MPIWS tool. To evaluate the point-to-point functionality of MPIWS, tests are presented that compare the performance of MPIWS to the performance of a leading Java based MPI implementation, mpiJava. These tests show that the MPIWS tool performs comparably with the mpiJava passing Objects but has a relatively large latency.

The tests also show that when MPIWS is compared with mpiJava passing defined datatypes, the mpiJava bandwidth is 15% faster than the MPIWS bandwidth for large message sizes (over 1Mb) and is similar for smaller message sizes. The

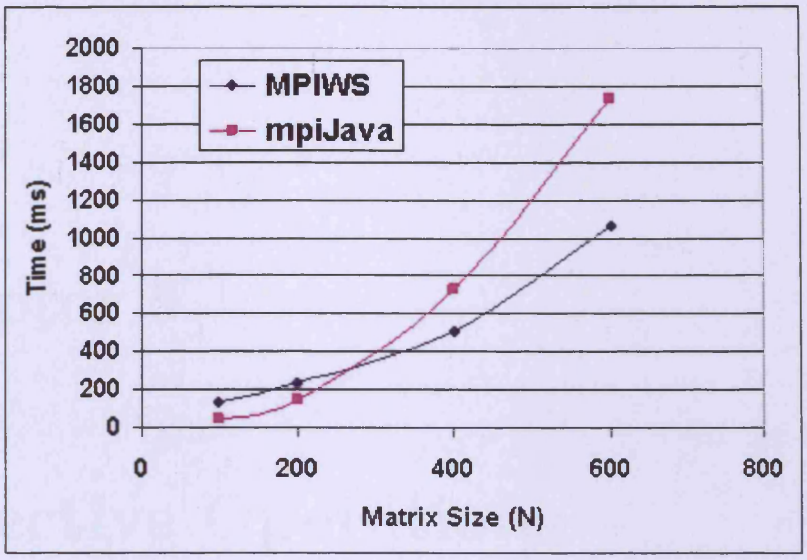


Figure 4.16: Results of matrix multiplication test using point-to-point communication with 8 processors (N = 0 - 600).

latency using MPIWS is significantly larger, approximately 2ms compared with very minimal latency for the mpiJava.

Chapter 5

Collective Operations

Chapter Overview:

One of the powerful features of MPI implementations is their ability to perform collective communication operations; the details of which have been discussed at length in Chapter 2.4.4. This chapter describes the design and implementation of the collective communication functionality within the MPIWS tool, which includes: *Broadcast*, *Gather*, *Barrier*, *Reduce* and *AllReduce*. The implementation is then evaluated by performance testing against a leading Java based MPI implementation, mpiJava, and the results are then analysed to enable a discussion about whether MPIWS has a practical use within the distributed computing tool set.

5.1 Introduction

Collective communication is used within the distributed computing environment to enhance the performance of message passing on a domain level. It provides faster communication for applications that require domain level systematic communication operations. Supporting collective communications in MPIWS is essential to demonstrate the potential efficiency of a Web service based approach for scientific computing. To this end, a number of collective communication operations, including *Broadcast*, *Gather*, *Barrier*, *Reduce*, and *AllReduce* have been implemented.

Collective operations are more complex than point-to-point communication and require extra processing such as retransmitting messages, combining data into a larger data set or appending data to existing data. In our design, the collective operations are built by extending the implemented point-to-point operations and adding the extra processing required for collective communication. These additions are implemented in both the MPI operations layer, and in the direct communications layer[1].

5.1.1 The Purpose of the Evaluation

The evaluation of the MPIWS collective communication functionality serves two purposes. The first is to assess whether the collective techniques, when used within a Web services environment, are more efficient and timely than conventional serial communication. To this end the evaluation must test implementations of both the serial and algorithmic versions of each collective operation. The second purpose of the collective communications evaluation is to

[1]For an explanation of the layer structure within MPIWS see Section 4.2.1 and Figure 4.2

aid in assessing the suitability of the MPIWS tool for MPI-style communication, as the collective communications functionality forms a large backbone of the MPI efficiency. To give a comparison with existing tools, the research presents direct evaluation of the performance of MPIWS against the performance of our test case, mpiJava. This will be a vital part of the overall evaluation of MPIWS.

For the evaluation of collective communication operations, both serial and binomial versions of the *Broadcast*, *Gather*, *Barrier*, *Reduce* and *AllReduce* operations have been evaluated against mpiJava. In addition there are two mpiJava versions of each test; one transmits the data as defined data types and the other uses the object transmission provided within the mpiJava tool.

5.2 Collective Operation Functionality

5.2.1 Broadcast

The easiest example of a collective communication operation to envisage is the broadcast operation, where data stored at one rank is sent to all other ranks in the communication domain. The simplest way to perform a broadcast operation is for the broadcasting rank, commonly called the *rootRank*, to serially send the data to each of the other ranks in the communication domain in turn.

Ideally within MPIWS the broadcast operation could be achieved by creating one XML-based SOAP message and consecutively sending this message to all other ranks. This technique would save the time involved in serialising the data on multiple occasions. Unfortunately the creation of the XML element involves the use of data streams to pass the data into the SOAP message. If the message is sent twice or more, the data stream has to be split between the multiple *Send*s which corrupts the message. To solve this problem, multiple message elements that use separate data streams for the object data are used, with one for each *Send* operation. However, whilst this method adds extra latency to the serial *Broadcast* operation, it does allow the reuse of the standard point-to-point *Send* operation.

There are a number of disadvantages in using the serial broadcast method which will be detailed. The serial version of the *Broadcast* has poor load balancing because the communication relies on the root repeatedly sending the message to other ranks.

A superficial view of this serial operation would suggest that there is no concurrent sending of data to different recipient ranks. However, since the send method uses the *fire-and-forget* Service Client, the Service Client returns after the existence of the receiving host has been confirmed. This means that if there are messages which still need to be transmitted from the root node, then it is likely that these messages will be sent concurrently by another Service Client. Although data can be transmitted to multiple recipients concurrently, there is a limitation on the utilisation of the network bandwidth; a single rank can only provide data to match the capabilities of the network card of the host machine. In the case that the network bandwidth is greater than that of the network cards in individual hosts, this broadcast mechanism can never utilise the full potential of the network bandwidth.

A better algorithm, with better network utilisation and load balance, can be achieved if the *Send* operations are distributed among multiple ranks. This allows multiple *Send* operations to be performed concurrently and utilises the bandwidth of multiple network cards. One such algorithm is binomial distribution.

The binomial distribution of the data message [8] is a more efficient method of performing the broadcast and has been widely used in MPI implementations for smaller message sizes (eg. MPICH). This method uses the receiving ranks within the communication domain to take part in the collective operation by forwarding on the message to further ranks. The implemented system uses a standard power of two binomial distribution to broadcast the message.

With this binomial approach, both the retransmission of the data, and the calculation of which rank to retransmit to, must happen in the methods at the direct-communication layer. The justification for this is that: if the recipient service instance needs to wait until the application layer is ready to receive the message data, then this could hold up the entire broadcast operation when only one receiving rank is not ready. Whereas, if the retransmission is achieved at the direct communication layer, there is no requirement for the ranks to wait until the application is synchronised before the retransmission is carried out. This is because the retransmission is independent of the service's application.

To this end MPI-style Web services provide a *bStore* method, distinct from the *store* method, with the additional retransmission functionality required for the binomial broadcast. This method primarily stores the data within the message data structure as with the standard *store* method, but then re-accesses the resource to recalculate the ranks that it is to send to, and performs the *Send* operation.

There are two issues associated with the retransmission of the data within the direct-communication layer methods which can be discussed in order of complexity.

1. The *fromRank* element of the message must remain set to the value of the *rootRank* that initiates the broadcast. So it is necessary to copy the *fromRank*

CARDIFF UNIVERSITY

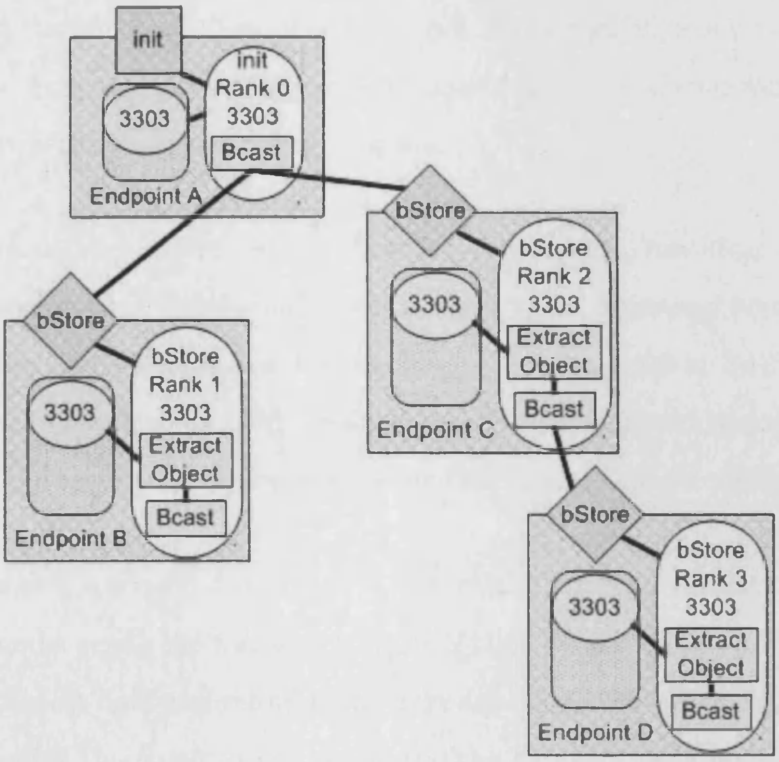


Figure 5.1: Architecture for the *Broadcast* operation.

value of the received message into the retransmitted message during the *bStore* method execution.

2. The sequential ordering of the messages is achieved by the use of a sequence number which separately sequences each message from one rank to any other rank. Within the binomial *Broadcast* operation, messages are forwarded from the broadcast's *rootRank*, to the ultimate receiving rank by other intermediary ranks within the communication domain.

As described in the point-to-point design (Section 4.2), within the scope of a pair of ranks, the message's sequence number is essentially unique to each message between those ranks, but the sequence number for that receiving rank is only accessible at the broadcast *rootRank*. If the forwarding rank was to use its repository to generate a sequence number, then the sequence number for a message sent from the forwarding rank to the ultimate receiving rank would be associated with the incorrect point of origin.

This problem can be solved by the *Broadcast* root rank including an array of sequence numbers that correspond to each rank in the *Broadcast* communication domain. This slightly increases the message's overhead data, but allows each ultimate receiving *bStore* method instance to extract the correct sequence number for its rank and use it to file the message within the resource's message buffer.

Figure 5.1 shows a simple scenario of a binomial *Broadcast* operation. Rank 0, as the root node, sends the message to rank 2 then rank 1. The *bStore()* method extracts the object data and rebuilds the message element for each retransmission. At ranks 1 and 3, the object is extracted and the *bStore()* method calculates that there are no further transmissions needed and the broadcast completes.

5.2.2 Gather

The *Gather* collective operation retrieves data from all non-root service nodes and arranges it in an array at the root service so that data d_j at rank R_j becomes an array of data d_{0ton} at R_{root}, where $j = 0$ to communication domain size $n - 1$ [8]. The resulting array is of size equal to the number of service nodes available in the communication domain, and each cell of the array contains the data sent from the service node with rank that equals the index value of the cell [107]. In MPIWS, two implementations of the *Gather* method have been implemented and tested: the serial version of the *Gather* method and the binomial version of the *Gather* method [8]. Both versions are implemented by using the point-to-point primitive operations *Send* and *Receive.* These operations are in the

MPI-operations layer and, unlike the *Broadcast* operation's *bStore* method, so is the *Gather* functionality. This is because the *Gather* is a synchronisation method and requires data from the application layer before it can proceed.

In the serial implementation of the *Gather* method, each non-root node within the communication domain sends its chunk of data directly to the root, and the root receives and collates the data into an array in their rank order.

The binomial implementation of the *Gather* method uses the same binomial tree used in the binomial *Broadcast* for the root service node to gather data from each non-root service node. In the execution of the binomial *Gather* operation, for each service node, an array of size equal to the number of nodes in the domain and initially occupied with null objects, is generated. The data generated by the service node is stored into the array corresponding to the rank value of the node. The service node may serve as an intermediary node that receives data from other nodes and then sends the received data, as well as its own data, to the node at a higher level of the binomial tree. The received data is in the form of an array with all the data stored in the corresponding cells. Each intermediate node needs to merge the received array with its own array by copying each non-null object into its own array. It then sends the merged array to the node above in the binomial tree.

5.2.3 Barrier

The *Barrier* operation provides a synchronisation mechanism for MPI applications. It involves no data transmission, but provides a guarantee that each service node in the communication domain has reached a particular point during its execution. There are many ways of implementing the *Barrier* operation, and a good reference to many of these methods can be found in Pjesivac-Grbovic et al. [84]. The method chosen in the design of MPIWS uses the collective operations that have already been implemented: a *Gather* operation followed by a *Broadcast* operation. The method was chosen because it involves the least number of back to back sends when compared to other methods, such as the Double Ring [84].

A *Barrier* operation involves very small or null data transmission. Compared with the small data size that is transmitted, the overhead of sending an empty or near empty SOAP message is high and this causes the poor performance of a *Barrier* operation. However, this problem can be overlooked if the MPIWS services are to be used in a coarse-grained application with large data transmission, in which the transmission times of large data transfers make the overheads of the barrier negligible.

5.2.4 Reduce

The *Reduce* operation is briefly described in the MPI background chapter (Chapter 2). It combines data values held within the ranks and transfers the combined result to the root rank so that the data x_j at rank R_j becomes $\sum_{i=0}^{R-1} x_i$ at R_{root} [107]. In the MPI specification there is the ability to define different operations as well as *summation*, The MPIWS tool provides the *summation* operation as a proof of concept. The design of the *Reduce* operation presents a few problems that have not yet been dealt with so far, namely, the use of Objects as the transmission data. The practice of summation requires a very specific datatype, i.e. the system cannot be expected to add two objects together unless a method is provided to enable this. MPIWS has provided support for the reduction of arrays of all Java types that can use the standard Java arithmetic operators. As an enhancement, MPIWS could offer the reduction of any object that implemented a *Reduction* interface containing a set of user defined combination operators.

The Communication structure for the *Reduce* operation is effectively the same as the communication structure for the *Gather*. Also, as with the *Gather* operation, the functionality for MPIWS's *Reduce* is executed within the MPI-operation layer. This is because the data cannot be transmitted until the local application layer has reached the gather point in its code, and must not be allowed to alter the data until the gather's transmission has been completed.

5.2.5 AllReduce

The *AllReduce* operation is an extension of the *Reduce* operation because the data is reduced to all nodes instead of just the root node. This means the resulting merged data is transferred to all the service nodes [107]. As mentioned in the background sections there are a number of different methods to achieve the *AllReduce* operation, one of which is a *Scatter* followed by a *Gather* [8]. This method is the method by which MPICH achieves the *AllReduce.* The problem with using this method in the MPIWS architecture is that the data is transmitted in the form of Objects, which are difficult to split into chunks and distribute over multiple nodes. Thus MPIWS adopted two different approaches: the *Reduce* operation followed by a *Broadcast* operation, and the recursive doubling approach [88]. Both approaches have been implemented and evaluated in MPIWS.

The method of recursive doubling utilises the efficiencies gained from the *SendReceive* operation. Each service node pairs with another service node and swaps data, then each pair of nodes pair with another pair and swap data, and this process is repeated as shown in Figure 5.2

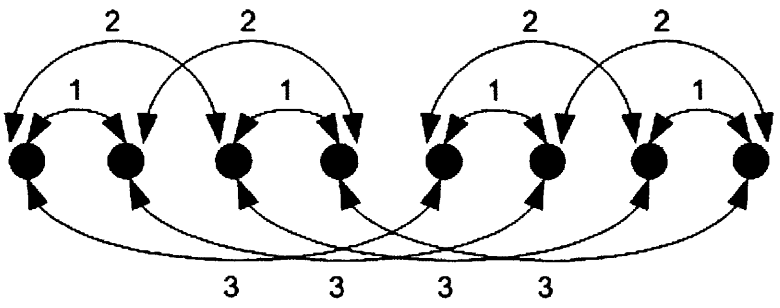


Figure 5.2: The recursive doubling communication for the *AllReduce* algorithm with three steps (1, 2 and 3).

This method does not involve splitting the data into chunks but is not as efficient as the *Scatter/Reduce* method. This method is very simple for communication domain sizes of a power of 2, but harder to implement for non power of 2 domains; as discussed in [88].

5.3 Collective Communication Evaluation

5.3.1 Broadcast Evaluation

In the broadcast test, a *Barrier* operation is performed before the start of the operation, in order to synchronise the services in the communication domain. The timing, conducted at the broadcast's *rootRank*, starts after the *Barrier* operation is completed. The *Broadcast* operation is then performed, which ends when all the service nodes have received the broadcasted message. In order to synchronise the communication domain at the end of the operation, the broadcasting service, i.e. the *rootRank*, is then notified by all services. The notification is performed by a *report-to-root* operation which is effectively a minimal data gather.

The results of the broadcast tests are shown in Figures 5.3 and 5.4. Figure 5.3

shows the results when the message size is in the larger range of 0 to 4Mbytes, while Figure 5.4 shows the results when the message size is in the smaller range of 0 to 500Kbytes. Six implementations of *Broadcast* have been tested: serial and binomial versions of MPIWS *Broadcast* (labelled on the graphs as MPIWS SBcast and MPIWS Bcast); serial and binomial versions of mpiJava Object data type *Broadcast* (labelled on the graphs as mpiJava Obj SBcast and mpiJava Obj Bcast); and serial and binomial versions of mpiJava defined data-type *Broadcast* (labelled on the graphs as mpiJava DT SBcast and mpiJava DT Bcast). All tests are carried out with a communication domain size of eight services.

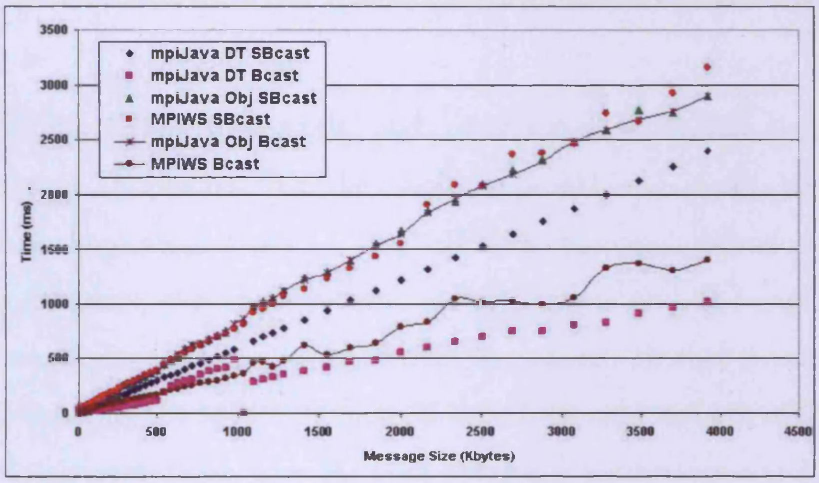


Figure 5.3: Broadcast test results (Message size = 0 - 4Mbytes).

It can be seen in Figures 5.3 and 5.4 that the mpiJava Object data type broadcasts of both the serial and *Bcast* perform in the same manner. This is because the mpiJava *Broadcast* doesn't utilise the underlying MPICH functionality, but transfers the data serially. It is not surprising to see that the MPIWS serial broadcast performs with a similar efficiency to these mpiJava Object broadcasts. Nor is it surprising to see that the mpiJava *SerialBroadcast* using defined data type transmission, is faster than all three other serial algorithms, which is effectively the same as the Ping*Pong scenario examined in the

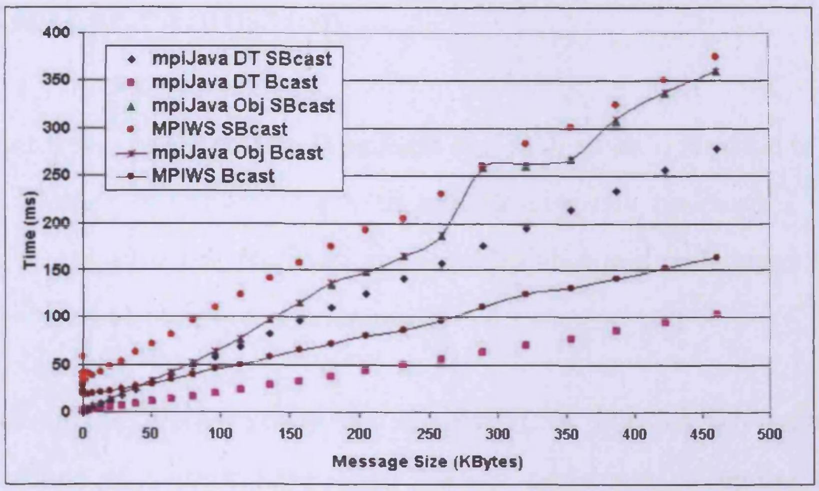


Figure 5.4: Broadcast test results (Message size = 0 - 500Kbytes).

previous chapter. The interesting part of the results, is the comparison of the algorithmic approaches, to the serial approaches in both Web services and mpiJava implementations. The MPIWS binomial *Broadcast* runs at approximately twice the speed of the serial version and the mpiJava data-type *Broadcast* runs in just over twice the speed of the serial data-type transfer. Comparing the MPIWS binomial broadcast against both of the mpiJava data-type implementations, the MPIWS version's performance is in between, completing in just under half the time of the mpiJava serial data-type version and approximately one and a half times the running time for the mpiJava's binomial data-type version.

The results are expected because using a binomial tree is a more efficient approach in implementing *Broadcast* than using a serial approach [8], and *Broadcast* in mpiJava using Object transfer is a serial version of broadcast due to there being no mapping to the native MPICH broadcast for broadcasts of the type Object. The MPICH algorithmic broadcast uses the binomial *Scatter/AllGather* approach for all messages over 12Kbytes, and this contributes to its improved efficiency.

5.3.2 Gather Evaluation

In the gather test, similar to the *Broadcast* test, a *Barrier* operation is performed before the *Gather* operation starts to synchronise the processors. The time calculation starts after the *Barrier* operation finishes and ends when the *Gather* operation returns at the root service node.

The results of the *Gather* tests are displayed in Figures 5.5 and 5.6. Six implementations of *Gather* have been tested: serial and binomial versions of MPIWS *Gather*; serial and algorithmic versions of mpiJava Object data type *Gather*; and serial and algorithmic versions of mpiJava defined data type *Gather*. All tests are carried out with a communication domain size of eight services.

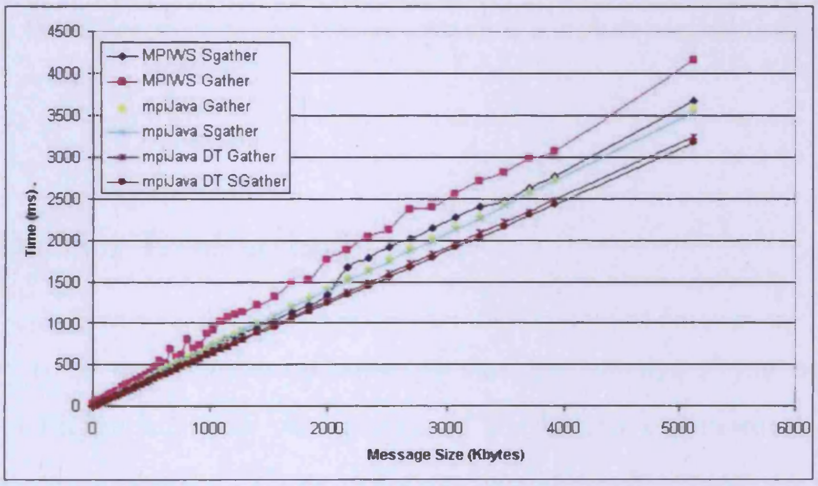


Figure 5.5: Gather test results (Message size = 0 - 5Mbytes).

In contrast to the *Broadcast* operation, where using a binomial tree significantly improves performance, using a binomial tree degrades the performance of a *Gather* operation because of the overhead that arises from repeatedly transmitting the cumulative data. According to the results, the mpiJava *Gather* performs better than the MPIWS serial *Gather* when the message size is small ($N = 150$), but

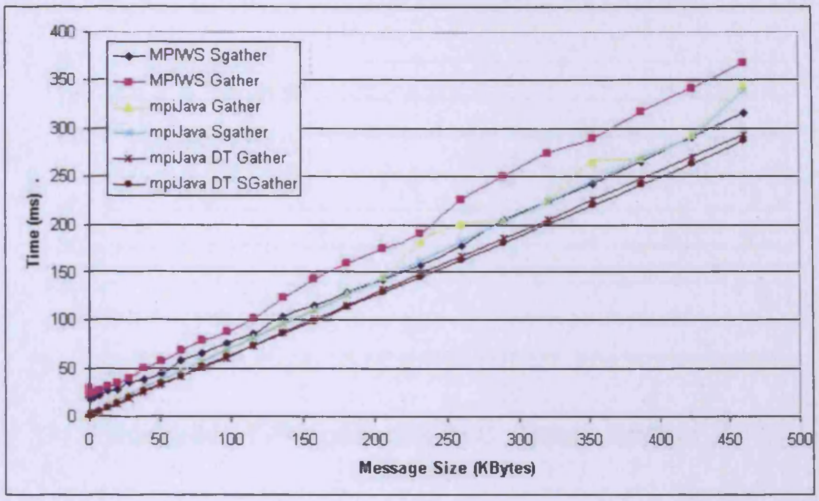


Figure 5.6: Gather test results (Message size = 0 - 450Kbytes).

as the message size increases, the graphs show the overheads of the MPIWS are diluted and the differences in the two approaches is not dependent on the message size.

5.3.3 Barrier Evaluation

Since there is no dependence on message size, the results of the barrier tests displayed in Figure 5.7 show the timings of the barrier communication against the number of processors. Three different barrier implementations are tested: the serial version of the MPIWS *Barrier* operation, the binomial version of the MPIWS *Barrier* operation, and the mpiJava *Barrier* operation. The serial version of the MPIWS *Barrier* operation is implemented by a serial MPIWS *Gather* followed by a serial MPIWS *Broadcast*. The binomial version of the MPIWS *Barrier* operation is a binomial MPIWS *Gather* followed by a binomial MPIWS *Broadcast*. When the message size is small, the overhead of SOAP messaging becomes significant and this is clearly shown in the results: both serial

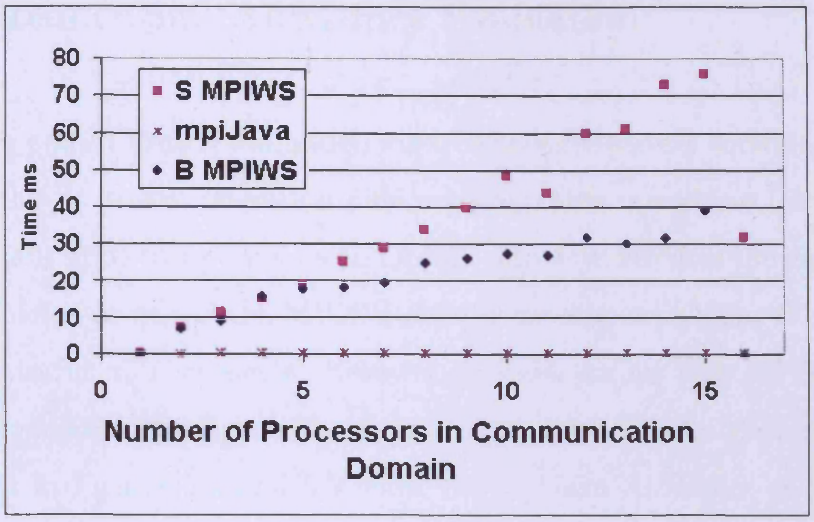


Figure 5.7: Barrier test results.

and binomial versions of the MPIWS *Barrier* operation are much slower than the mpiJava *Barrier*. Comparing between the serial and the binomial versions of the MPIWS *Barrier* operations, the binomial implementation works better than the serial implementation when the number of processors is greater than 5.

This operation is the worst case scenario for the MPI-style services due to the minimal size of the data transmitted and the need to send a comprehensive SOAP message to achieve the communication: the whole of the SOAP message is overhead. Although this result on its own is not a very positive argument for the MPI-style Web services architecture, the *Barrier* is a very short operation compared to coarse-grained data transmission operations. In most application scenarios, the poor performance of the *Barrier* will become unnoticeable due to the longer transmission times of communications of larger quantities of data.

5.3.4 Reduce and AllReduce Evaluation

The *Reduce* and *AllReduce* evaluation must be considered very carefully. MpiJava processes the *AllReduce* operation differently to other operations, as it will not allow the data to be transferred as an Object. MpiJava requires the data transfer to be conducted as one of the MPICH defined datatypes to allow the reduction operations to function properly. However this also means that MPIWS can no longer be evaluated against a message passing tool which is transferring Objects. The graphs in Figures 5.8 and 5.9 show the mpiJava *AllReduce* as well as the two MPIWS implementations of the *AllReduce*: the *Reduce/Broadcast*, and the recursive doubling methods.

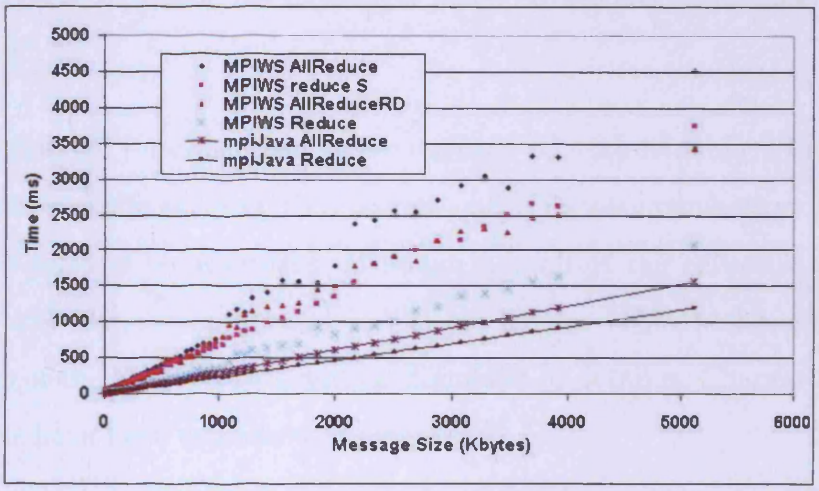


Figure 5.8: Reduce and AllReduce test results (message size 0 - 5Mbytes).

With the *Reduce* evaluation, it can be seen that the collective communications approach is consistently beneficial. When it is compared to the transmission of datatypes within mpiJava the impact of the extra serialisation step can be seen. Again for the *AllReduce* evaluation it can be seen MPIWS does not fare too well, but, as has been discussed, this is not a surprise. What is important though, is the comparison to the serial operations in the *Broadcast* and *Reduce* experiments.

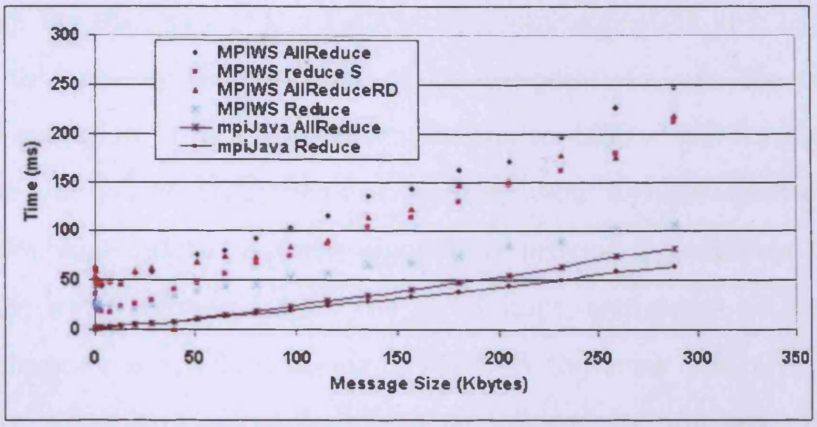


Figure 5.9: Reduce and AllReduce test results (message size 0 - 300Kbytes).

These comparisons show the ability of the Web services architecture to use the collective communications operations in order to increase the efficiency of the data transfer.

The test results for the collective communication operations confirm that MPIWS is a practical and efficient way to integrate collective communications techniques into a Web services environment, although not all of the collective operations (especially the *Barrier* operation) are as efficient as could be hoped. The full conclusions of the MPIWS tool will be discussed in detail in Chapter 8 after the applications have been examined in Chapter 7.

5.4 Chapter Summary

This chapter has introduced the collective communication functionality within the MPIWS tool set. The performance of the *Broadcast*, *Gather*, *Barrier*, *Reduce* and *AllReduce* have been evaluated against serial implementations using MPIWS and mpiJava implementations. The results have shown that collective communication techniques similar to those used in the MPI implementations can improve the efficiency of the Web service communication architecture when compared to serial implementations of these operations. In the case of the *Broadcast* operation, this improvement is up to 50%. Within the MPIWS architecture two collective algorithms have been used; binomial distribution, and recursive doubling. Both of these algorithms provide a justifiable increase in performance when compared with the serial implementations within MPIWS. One notable point is the limitations of MPIWS to utilise collective techniques that require the split of a complete data set followed by processing of elements within that subset. It should also be noted that the performance of the barrier operation is unsurprisingly slow due to the ratio of overhead to the message data.

Chapter 6

Enhancements to Workflow Communication Structures Using MPIWS

Chapter Overview:

In this chapter, "*direct messaging* within workflow executions" is discussed. This discussion demonstrates the potential of using MPIWS to enhance the efficiency of data communication within a Web service based workflow environment. A typical workflow is executed by an application called a manager, which is responsible for the invocation of all services in the workflow. Once a service has been invoked by the manager, the output data from that service is transmitted, via the manager, to the input of the next service defined in the workflow. This data transmission is sent via the Web service's standard response and request messages. By using the *direct message* passing functionality provided in the MPIWS tool, it is proposed that passing the data directly from one service to the next, without relaying it via the manager, will increase the speed of the workflow's communication.

6.1 Introduction

In the previous chapters, MPIWS has been proposed and evaluated against mpiJava, a leading Java implementation of MPI. MPIWS enables MPI-style applications to be executed within the Web services framework by facilitating the transfer of data from one executing service to another, concurrently executing service. This data transfer can occur at any time during the execution of the MPIWS service. The original motivation for this research came from the development of workflow languages such as MPFL [60] and GSFL [69] with the ability to describe MPI-style communication. In this chapter, the use of MPIWS to provide enhancements to workflow communications is examined.

Within a standard workflow scenario the data communication consists of input to, and output from the service. This style of communication is undertaken at the beginning and end of each service. In a Web service based workflow execution, these communications are the request to invoke a service, and the response from the termination of that service.

Web service based workflows are typically controlled by managers that centralise the flow of data from one service to the next. This is achieved by the response data being returned to the manager from one service before that data is then forwarded, by the manager, as an invocation request to the next service. In this research it has been proposed that the decentralisation of this data communication, by allowing the services to transfer the data directly from one service to the next, could enhance the overall performance of service based workflows, especially ones that process large data sets.

To achieve this decentralisation, MPIWS is used to enable the direct transfer of data between services. To execute a workflow application, a communication domain is initiated, and the initial data is passed from the manager to the first service in the workflow. All other services in the workflow are concurrently initialised within the communication domain with their respective ranks, and then wait for the service input data. The communication of the data from one service to the next is performed by a *Send* and *Receive* operation between the two service ranks within the communication domain. Once a service has sent its output data on to the next service in the workflow, it responds to the manager with any metadata required for the workflow management. Once a receiving service has received its input data, it proceeds with the execution of its task.

For the services to know where the input data is to come from, or where the output data needs to be sent, there needs to be an extra level of understanding between the services within the communication domain. Each service must have knowledge of how it fits into the workflow definition. This issue can be addressed by including an XML based configuration description element (workflowCFG) within the initialisation request to the service from the manager. The configuration element specifies the rank from which the current service should expect to receive its input data, and the rank to which the current service should send output data. A simple XML schema for the configuration of services involved in direct message passing has been devised to demonstrate the concept. This schema is highly extensible and allows for the description of multiple input and output sources. It could be extended to allow for collective communication operations.

Listing 6.1 shows a simple example of the configuration element. It prescribes that the receiving service receives its input data from *rank 1* and sends its output data to *rank 3* (A more comprehensive description for the service configuration elements can be found in Appendix A.1).

Listing 6.1: Configuration element prescribing that Rank 2 receives input data from Rank 1 and passes the output data to Rank 3.

```
<workflowCFG>
  <in>
    <fromRank> 1 </fromRank>
  </in>
  <out>
    <toRank> 3 </toRank>
  </out>
</workflowCFG>
```

6.2 Example Workflows

In this section, workflows that demonstrate *direct messaging* data transfer between services are described. These workflows are executed to show both their potential, and also to show any improvements in the workflows communication efficiency when compared to standard workflow execution.

6.2.1 Chain Workflow

The first workflow to be described, is a simple chain of services which comprises a set of identical service instances. Each service instance accepts a data element as its input, and then echoes the same data element as its output. The workflow manager is in charge of passing the data from one service to the next service in the workflow. This example minimises any processing done in the service so the communication can be assessed independently. The workflow is pictured in Figure 6.1

The workflow is executed by both a workflow manager that supports standard Web service invocation, and by a workflow manager that supports *direct*

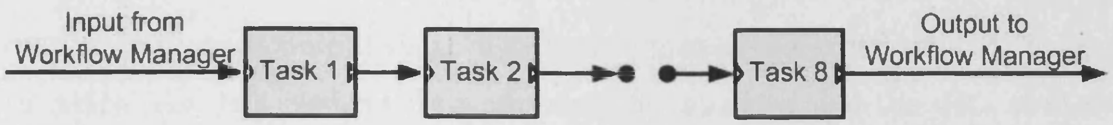


Figure 6.1: Workflow for the service chain experiment.

messaging between services using MPIWS.

In the first instance, where a standard workflow manager is used, each service receives the data element as part of the invocation request message from the workflow manager and then returns the output data to the workflow manager via the response message. The service is deployed on a number of servers to provide 8 identical services. The workflow manager then calls each of the 8 services in turn and re-transmits the output message data to the input of the next service. Figure 6.2 shows the invocation scenario of the workflow using the standard workflow manager. Each request and response message includes the whole data element. In the test, the execution of the workflow is timed for a range of message sizes.

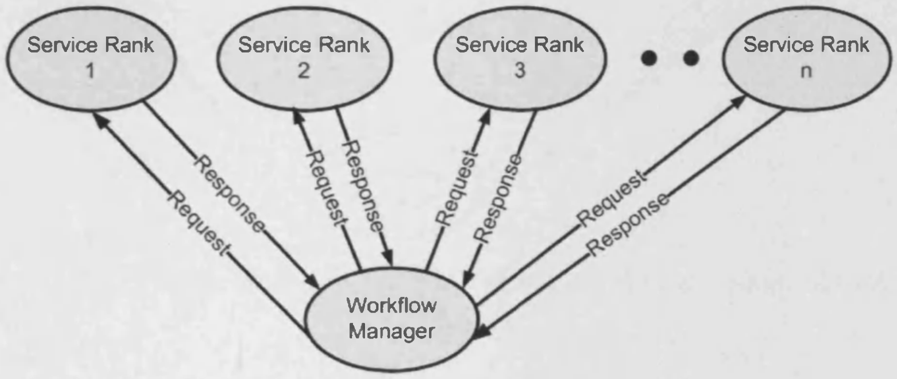


Figure 6.2: Invocation scenario of the chain workflow using the standard workflow manager.

In the workflow execution scenario using workflow managers supporting *direct messaging*, the service involved is designed as an MPIWS-style service. When invoked, the service initialises within its prescribed domain. Each service parses its configuration element, which describes how the service should receive its input data and return its output data. The first service in the workflow is configured to receive the data element from the workflow manager and the data element is part of the service invocation request. When the first service completes, it uses the MPIWS *Send* method to transmit the data element to the next service in the workflow. This next service has already been initialised and configured to *Receive* the data from the first service and to *Send* its output data to the next service in the workflow. This process continues until the final service, which is configured to return the data element to the workflow manager within the response to its initial service invocation request. Figure 6.3 shows the workflow execution scenario using MPIWS *direct messaging*. The solid arrows represent the messages containing the data set as well as MPIWS metadata and the dashed lines represent the messages containing only MPIWS metadata.

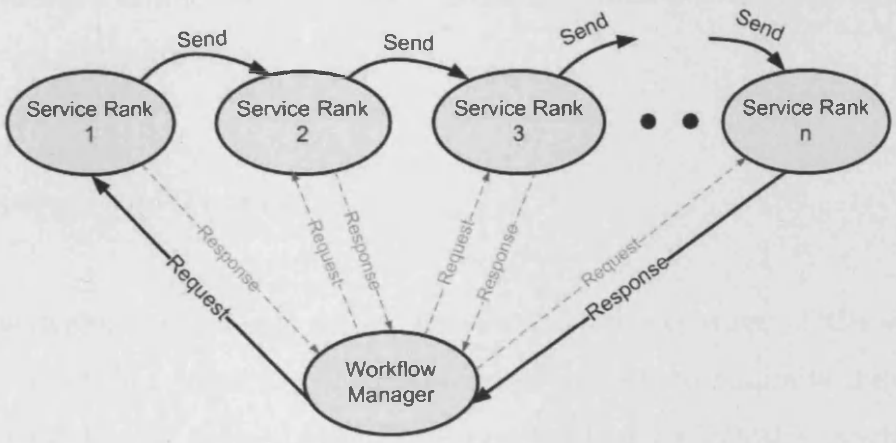


Figure 6.3: Invocation scenario of the chain workflow using MPIWS *direct messaging*.

MPIWS provides the functionality for data to be transferred as serialised Objects. This approach improves the efficiency of data communications for scientific applications that transfer large quantities of numerical data. However, for a general Web service that does not use as much numerical data, this serialisation would not provide as great a benefit. Additionally the inclusion of the serialisation to the protocol stack requires that all services in the MPIWS domain adhere to the serialisation protocol. In the case of MPI-style applications, this is not so much of a problem as the services are designed in a more tightly-coupled manner. But, for general services, it is beneficial to retain the data in XML format to allow a more loosely-coupled architecture.

The work presented in this chapter uses the MPIWS methods but retains the XML formatting for the transmitted data. This allows the communication to adhere to Web service formatting standards. Another advantage of retaining XML formatting in the *direct messaging* workflow execution is to allow a comparison to the standard workflow execution for the purposes of evaluation. This comparison is acceptable as the data transmitted in both the standard and the *direct messaging* approaches is same in both content and format. Again the whole process, including the initial set-up of the communication domain, is timed for a range of message sizes.

Assumptions and Theory

Both the deployment of this set of services and the execution of the workflow manager is within a single Local Area Network in order to minimise the affect of the differing network route times. The conditions under which this experiment is conducted allow certain assumptions to be made:

– The network bandwidth and latency for each route between services in the network is, allowing for network usage and wiring variations, the same. This can be assumed because each of the Web service servers and the workflow manager processor are all directly connected to the same physical switch.

– The processing time for the data at each service node, allowing for processor utilisation, is the same. This can be assumed because: the configuration of each of the machines is identical, the testing is done at a time of low usage, and minimum timings are taken from a number of test iterations.

Allowing for these assumptions, a simplified theoretical prediction of the best possible results can be made. The time improvement of the *direct messaging* execution of the workflow will never be greater than the ratio of the number of messages sent in each system. For this workflow, comprising a chain of eight services and a workflow manager, the standard Web service method transmits the data sixteen times and the *direct messaging* method transmits the data nine times. Hence the best possible performance for the *direct messaging* execution in this case will take 9/16ths of the standard workflow execution.

In this performance estimation the initialisation overhead of the *direct messaging* method has been completely ignored. Within the experiment, all the time taken for the MPIWS initialisation and the extra data transfer associated with it will be included in the total timings of the experiment. Although this is a simple theoretical prediction, it shows the potential improvement in the communications effectiveness using *direct messaging* supported by MPIWS.

Results

The results in Figure 6.4 show the timings of the workflow execution using the standard workflow execution approach and the *direct messaging* workflow execution approach. As a reference, the theoretical minimum timings of 9/16ths of the standard approach is also shown in the graph. It can be seen that the workflow execution using the *direct messaging* approach show a marked improvement over the standard workflow approach. However, the effectiveness of using *direct messaging* in this workflow execution becomes obvious only when the size of the data transmitted is large enough (>300Kbytes in this example). This can be put down to the MPIWS initialisation overheads.

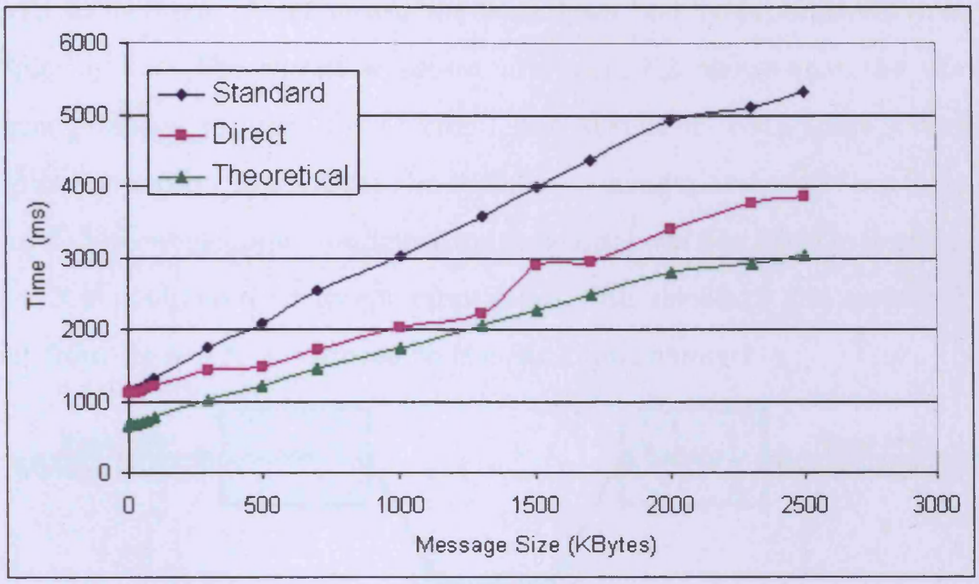


Figure 6.4: Message size vs. execution time for standard workflow execution and *direct messaging* workflow execution.

These results provide experimental evidence that enabling *direct messaging* between services involved in the execution of a workflow, improves the efficiency of the workflow execution. It is, however, essential to understand the limitations of these results. The experiments were specifically conducted on a network where all the servers are connected to the same switch. In more general scenarios, the services would be deployed on a more distributed network. This would mean the messaging times from service to service and from service to workflow manager will be different for each and every case. In the case of *direct messaging*, this must be taken into account when estimating the potential gain of any system wishing to employ this methodology.

6.2.2 Workflow Services with Multiple Inputs

The second workflow presented in this chapter demonstrates the potential of MPIWS to facilitate *direct messaging* workflows that incorporate services with multiple inputs. The workflow shown in Figure 6.5 shows that the workflow manager provides an input for *service 1* and *service 2*. Both these services are configured to receive input from the workflow manager and send their output to *service 3*. This single input configuration style has been described in Section 6.2.1. *Service 3* is configured to accept input from both *service 1* and *service 2*. The output from *service 3* is returned to the workflow manager.

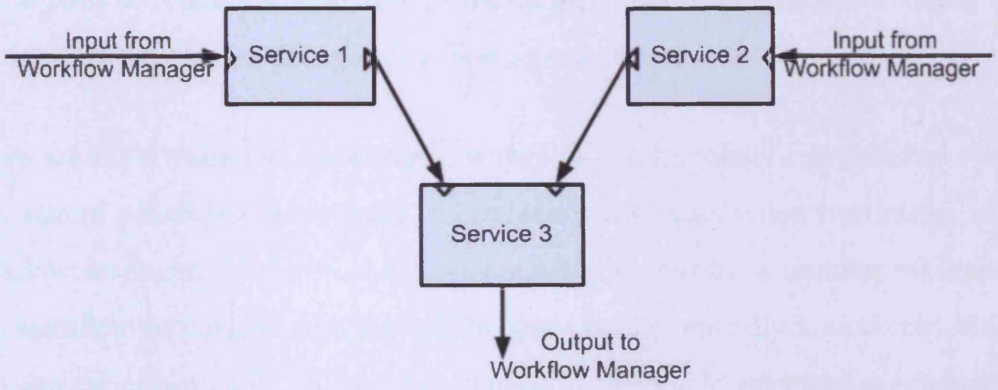


Figure 6.5: Workflow demonstrating a service with multiple inputs.

The configuration element for *service 3* is shown in Listing 6.2. The <*in*> element describes the inputs to the service, and in this example it contains two child elements. The presence of multiple child elements within the <*in*> element instructs the MPIWS initialisation method to loop through these child elements, receiving data from each specified rank using the MPIWS *Receive* method. The received data from each of the input ranks is stored in an array that is then passed to the services application.

This workflow example demonstrates that multiple input functionality can be achieved with *direct messaging* provided by MPIWS. This decentralised approach

Listing 6.2: Configuration element prescribing that the service receives input data from Rank 1 and Rank 2 and then passes its output data to the service's client (the workflow manager).

```
<workflowCFG>
  <in>
    <fromRank> 1 </fromRank>
    <fromRank> 2 </fromRank>
  </in>
  <out>
    <toRank> client </toRank>
  </out>
</workflowCFG>
```

can be used to replace similar functionality provided by centralised entities, such as actors in the Kepler [2] workflow management system.

There are other inherent advantages with this methodology for workflow execution, one of which is the potential to reduce a communication bottleneck at the workflow manager. If the workflow manager is executing a number of branches of a workflow in parallel, having all the data being centrally transferred via the manager processor could cause a bottleneck in the whole workflow execution. By decentralising the workflow communications, the bottleneck will be eliminated.

6.3 Chapter Summary

In this chapter *direct messaging* between Web service based workflow components has been introduced. This technique uses the MPIWS tool to facilitate the direct communication of data from the output of one service to the input of the next service, as defined by the workflow description. This avoids the extra communication required to route that data via the workflow manager. Testing has confirmed that for workflows that require large data transfers (>300Kbytes), this method improves the communications performance of certain workflow executions. *Direct messaging* could also help to avoid communication bottlenecks at the workflow manager in workflows that have large numbers of communications between services.

Chapter 7

Applications

Chapter Overview:

The previous chapters have outlined the design and implementation of MPIWS, and have evaluated it against the mpiJava tool set. This is an essential piece of work in order to prove that MPIWS is efficient in terms of the communications protocols. This chapter examines some applications that can employ MPIWS, in order to address the motivations for this work in more detail. Starting with a one dimensionally blocked matrix multiplication calculation using the *Broadcast* operation, and then a piece of molecular dynamics simulation code, this chapter will assess the application of MPIWS for high performance computing.

7.1 Introduction

In this chapter, two applications that use MPIWS are assessed. The first application is a one dimensionally blocked matrix multiplication calculation. It uses the *Broadcast* operation to distribute data to every rank in the communication domain. This application has been chosen as it is frequently part of MPI benchmark suites [73] and is therefore well known in its use for evaluation and testing. The one dimensionally blocked matrix multiplication calculation also has a simple communication pattern which simplifies any analysis that is to be done. The second application is a piece of molecular dynamics simulation code, MolDyn. This application has been chosen because the simulation of molecular dynamics is a highly active research area within high performance computing. The ability of MPIWS to run the MolDyn code demonstrates the ease of porting MPI-style code to the MPIWS platform, and the ability of MPIWS to run real high performance computing applications.

7.2 Matrix Multiplication

For the matrix multiplication test, a one-dimensional blocked matrix multiplication application using collective communication operations has been implemented. The application is run using both the MPIWS infrastructure and mpiJava. The speed-up of the applications when running in parallel over differing numbers of processing nodes is presented for analysis.

The matrix multiplication application is a common example of the use of parallel processing to perform time consuming calculations. The number of operations for the calculation scales as $O(n^3)$ for $n \times n$ matrices (it is actually $2n^3$ [31]), so for $n = 1000$ there are two billion floating point operations.

This application is based on a simple parallelisation of the matrix multiplication problem [31]. In a matrix multiplication $\bar{C} = \bar{A} \times \bar{B}$ for an n by n matrix, Demmel [31] describes a matrix that is divided into columns of size n by n/p where p is the number of processors involved in the calculation and n is divisible by p. The columns are referred to as X(i) where X denotes the full matrix, i is the number of the column, where i = 0 to p-1. This is shown in Figure 7.1. Also the dashed square at the bottom of X(1) is an n/p by n/p part of the matrix referred to as X(j,i), again j = 0 to p-1 and this sub matrix is a block taken by equally dividing the X(i) matrix into p rows.

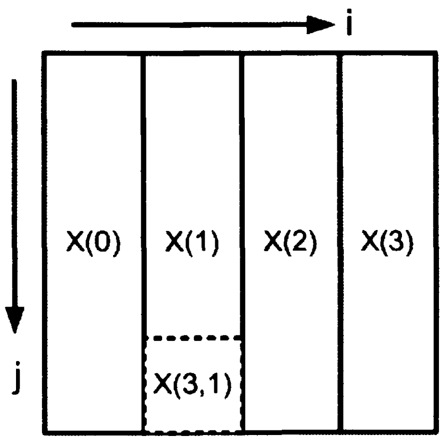


Figure 7.1: Matrix split one dimensionally into columns

Equation 7.1 [31] shows the calculation involved. What is being said in this equation is: if processor rank(i) owns matrix columns A(i), B(i), and the answer matrix column C(i), each processor will work out its section of the answer C(i). For this to happen, each processor will need its B(i) and also every A(i) in the system. This requires, the system to distribute every A(i) to every processor. Processor rank(i)'s result for C(i) is derived by accumulating the results from A(j) * B(j,i) for every value of j (where j = 0 to p-1).

$$C(i) = C(i) + A * B(i) = C(i) + \sum_{j=0}^{p-1} A(j) * B(j,i) \tag{7.1}$$

The communications for the matrix multiplication application are shown in Figure 7.2, where each horizontal block represents A(i) being broadcast from the *rootRank* of the broadcast to all the other ranks in the communication domain. In between the blocks of communication there is a set of calculations at each rank that process the matrix block that has just been received.

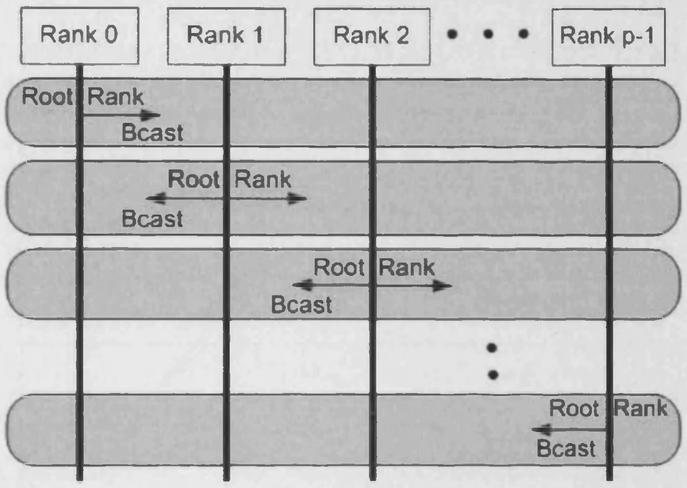


Figure 7.2: Parallel matrix multiplication communications

These calculations are extremely time consuming and the results of the higher order matrix multiplications dilute the performance impact of the communications. This means that it would be difficult to extract relevant information from the total time of the matrix calculations, so instead the speed-up of the applications have been presented using both the mpiJava and the MPIWS tools.

7.2.1 Matrix Multiplication Evaluation

Figure 7.3 shows the effect of the number of processors on the speed of a matrix multiplication application running over MPIWS services with a range of matrix sizes. From an initial perspective, it can be seen that the speed-up of the application for small problem sizes (eg. n = 160) is very poor. This is not unexpected: the number of calculations for this size of application is approximately 8 million, which on one processor takes 45ms. Yet if the broadcast graph in Figure 5.4 is referred to, the broadcast communication time for an n = 160 message (approximately. 200Kbytes) is approximately 75ms.

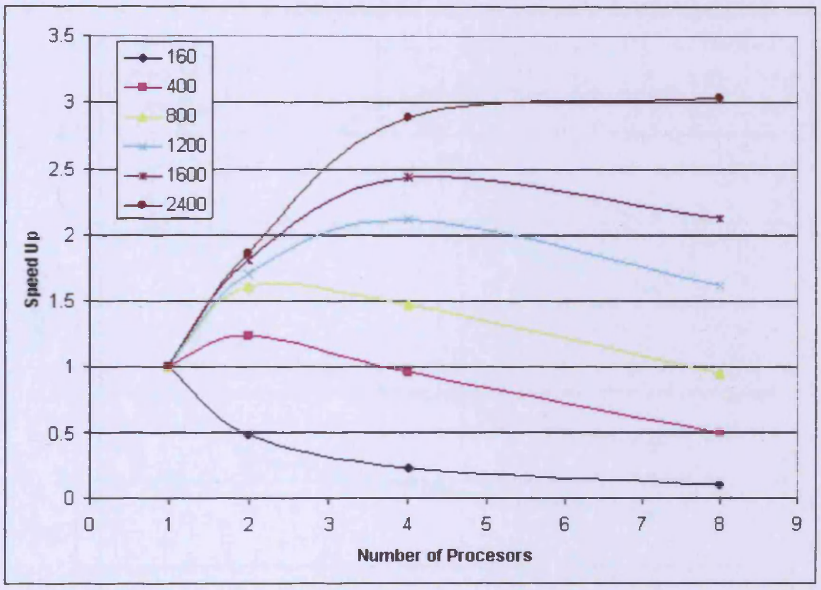


Figure 7.3: The speed-up for the matrix multiplication application over 1-8 MPIWS services.

As the problem size increases, the calculation time also increases by $O(n^3)$, yet the communication time only increases by $O(n^2)$. This means that the efficiency of the parallel application can increase. Efficiency is usually defined as:

$$Efficiency = \frac{Speed - up}{Number\ of\ processors} \tag{7.2}$$

Referring back to the graph in Figure 7.3 it can be seen that as the problem sizes get larger, the speed-up increases. For a problem size of $n = 2400$, the efficiency of the application using 2 processors is approximately 93% and when using the 4 processors it is 72% and when using 8 processors, the efficiency is 38%. Again this fall in efficiency is expected because, as the number of parallel processors (p) increases, the number of calculations per processor scales as $O(1/p)$ but the communication time scales as $O(logp/p)$.

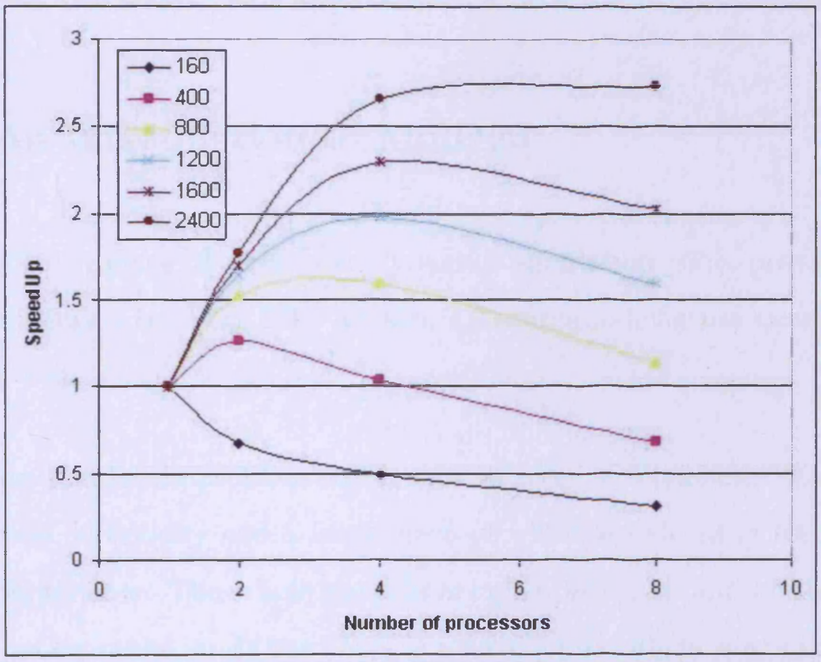


Figure 7.4: The speed-up for the matrix multiplication application over 1-8 mpiJava nodes.

It is impossible to directly compare this MPIWS version of the matrix multiplication application with an mpiJava version. This is because the MPIWS version broadcasts the matrix parts as Objects in a binomial broadcast operation. This is the most efficient method that MPIWS can use for this application. A similar mpiJava application could either broadcast the matrix serially as Objects, or broadcast the matrix binomially as defined data types. To give a loose comparison to the MPIWS matrix multiplication application and to show that the speed up obtained is not unreasonable, results of a mpiJava implementation are presented. Figure 7.4 gives the results of an mpiJava matrix multiplication using Objects being broadcast in a serial manner. This graph shows that the speed-up, is similar to the MPIWS version although no detailed analysis should be made.

7.3 Molecular Dynamics

7.3.1 An Introduction to MolDyn

MolDyn [93] is a piece of molecular dynamics simulation code, provided by the Java Grande Forum with the MPJ Version 1.0 source code for use as an evaluation benchmark test.

The MolDyn simulation problem consists of an array of n particles. Each particle has a position, a velocity and a force, each of which is defined in terms of its x, y, and z components. The whole particle array is presented and initialised at all the participating ranks, and then there is a series of iterations where the particles move and the positions, velocities and forces are recalculated. The movement and recalculation of the velocities are a relatively simple calculation that scales as $O(n)$, so it is faster to carry these calculations out for every processor locally. The main part of the calculation is the recalculation of the forces exerted on each particle. The calculations of the new particle forces are distributed amongst the contributing ranks.

The calculation of the force on particle i is a function of the distances between it, and every other particle in the problem, which scales as $O(n^2)$. The distribution of these recalculations is achieved by each rank processing one in every p particles in the particle array, where p is the number of processors in the problem domain. When these distributed calculations have been achieved, the forces are collected into an array for each dimension, and an *AllReduce* operation is performed on each of the force arrays. The force data can then be reassembled into the particle objects and then the next iteration can be performed.

The MolDyn code fits nicely into the evaluation of MPIWS as it spans two types of application. Firstly it can be thought of as an iterative workflow, and secondly it is a scientific application which sits firmly in the realm of the mpiJava application scope.

MolDyn can be thought of as an iterative workflow that repeatedly calls a set of distributed services to perform a looped iteration on a set of data (see Figure 1.1). This workflow can be optimised by enabling the distributed services to directly communicate the iteration results throughout the communication domain, saving the repeated initialisation costs associated with the loop model, and also allowing the use of collective communication techniques to increase the efficiency of the data distribution.

As a standard workflow which is looped through, the service would comprise the initialisation, the move functionality and the recalculation of the velocities and the forces. The resultant data would then need to be returned to the workflow manager (or an intermediary service) to combine the distributed force arrays. The force arrays would then need to be re-distributed to the services for the processing of the next iteration. This model assumes that the position and velocity vectors can be stored locally at the service endpoints in, say, a resource in between iterations, otherwise they too would need to be transferred.

MolDyn is also a typical high performance computing application. MPI implementations are commonly used for molecular dynamics simulation and there are many examples of production grade code available [32, 76]. These codes use a variety of communication architectures to achieve their goals but for the purposes of the evaluation of MPIWS, MolDyn will suffice. The communications architecture involves the *AllReduce* operation on the three force arrays and on three energy variables, plus three *Barrier* synchronisations per iteration. The benchmark test performs 50 iterations and the size of the particle array varies from 2 thousand to 32 thousand particles.

Evaluation of MolDyn running on MPIWS

If the communication results for the *AllReduce* operation are examined, the extra time that the MolDyn application should take running on the MPIWS tool compared to on mpiJava can be estimated. Figure 7.5 shows the timings for a range of particle array sizes run on both MPIWS and mpiJava as well as the predicted and actual difference in the two results. The second graph, Figure 7.6 shows the speed-up of the MPIWS MolDyn application whilst running on a range of service nodes.

These graphs show that the predictions are not all that dissimilar to the actual results. As expected the MPIWS version does take longer than mpiJava, but, as can be seen from the speed-up graph, there is a definite timing improvement when the application is distributed over more than one service. The MPIWS implementation of the molecular dynamics simulation gives an efficiency of 61% when a 27,437 particle simulation is split between 8 services.

These results show an important point about the applicability of MPI-style collective communications in the workflow environment. If MolDyn were run for

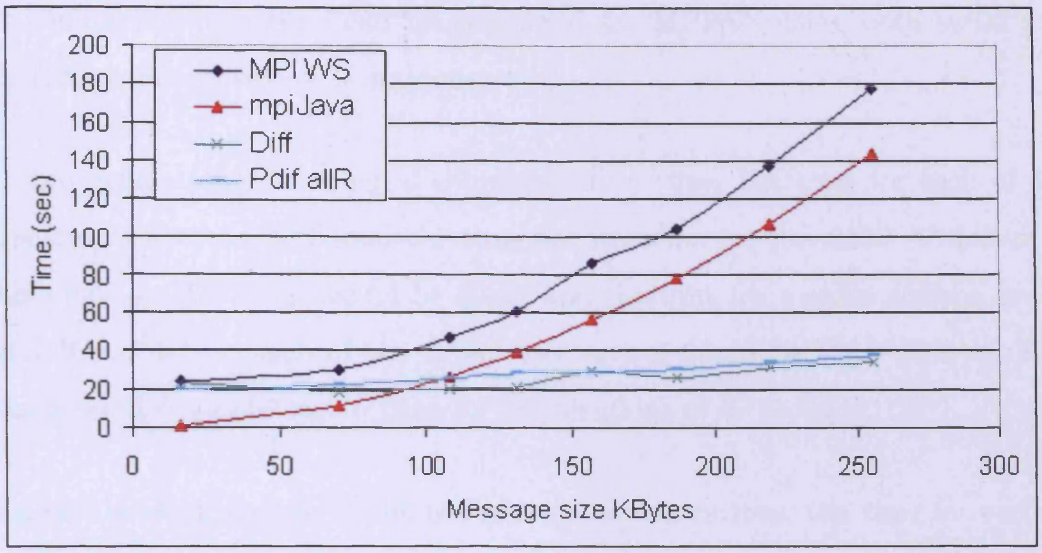


Figure 7.5: The times taken for the MolDyn Application vs the individual forces message size for MPIWS and mpiJava.

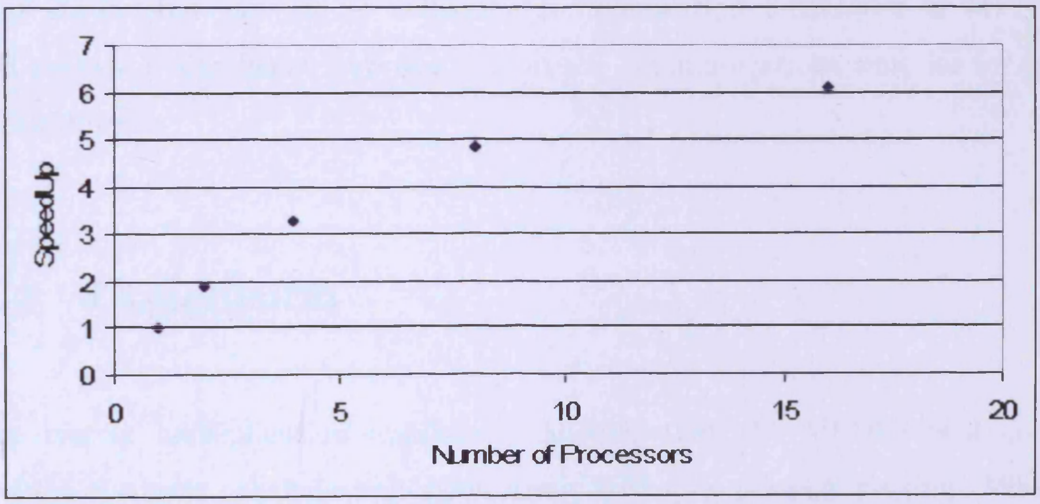


Figure 7.6: The speed-up for the MolDyn application over 0-16 MPIWS services running the application with 27,437 particles (individual force message size is approximately 220Kbytes).

27,436 particles on 8 services (force array message size approximately 220Kbytes), the communications time can be estimated for MPIWS using both serial and collective communication techniques.

For the estimation of the serial communication time, the time for each of the three barriers would be $33ms$, the time per iteration for the serial *AllReduce* of each single double value would be $60ms$ and the time for a serial *Reduce*, and a serial *Broadcast* for each of the three force arrays would be $155 + 198ms$. This gives a serial communication time for 50 iterations of 67 seconds.

This can be compared with the collective communications, the time for each of the three barriers would be $24ms$, the time per iteration for the *AllReduce* of each single double value would be $60ms$ and the time for a recursive doubling *Allreduce* for each of the three force arrays would be $190ms$. This gives a collective operations communication time for 50 iterations of 41 seconds.

This shows that the use of collective communication techniques in MPIWS will provide a significant improvement in the communications time for for real applications.

7.4 Conclusion

The matrix multiplication application showed that the MPIWS tool could perform a simple parallel application using MPI-style message passing. Whilst the comparison of performance to the mpiJava version is limited to Object type transmission, the speed-up of the MPIWS application does show that MPI-style applications can be run efficiently over a Web services architecture. This test also demonstrates the collective operation *Broadcast* in a real application.

The use of the MPIWS's *AllReduce* operation in the MolDyn application undoubtedly limits its performance when evaluated against the mpiJava approach, but still the speed-up of the MPIWS version as shown in Figure 7.6 shows that this type of application can benefit from parallelisation over the Web services infrastructure.

This evaluation of the MolDyn simulation demonstrates that MPIWS based communication can make a significant difference in the communication overheads of Web service workflows that contain parallel loop structures. If the workflow was implemented as a loop of service invocations, the data from the current service iteration would be returned to a central service for combination and re-dispersal in the next iteration. This means there would effectively be a serial *Reduce* operation followed by a serial *Broadcast* operation, whereas in the MPI-style services approach, the use of MPIWS's collective *AllReduce* operation still greatly improves the performance of the total communication stage. This is true even though the efficiency of the recursive doubling *AllReduce* operation passing Java Objects is not as good as the mpiJava's *Scatter/Gather* approach passing data types.

The evaluation also shows that MPIWS can be used to efficiently run scientific computing applications that are written for traditional MPI implementations by simply replacing the MPI communication calls with the MPIWS communication calls and deploying the application as an MPIWS Web service. This is considerably less demanding than having to re-write the application to fit into the existing workflow structure.

7.5 Chapter Summary

This chapter has presented two applications that use the MPIWS tool to perform MPI-style computing. The first is a parallel matrix multiplication application which is a common application used for evaluation purposes. The second is a molecular dynamics simulation that demonstrates the ability of MPIWS to run real life high performance computing applications.

In the matrix multiplication application the MPIWS implementation was run on a range of problem sizes using an $n \times n$ matrix. For $n = 2400$ the MPIWS implementation achieved an efficiency of 93% when the application was split between 2 services, 72% when split between 4 services, and 32% when split between 8 services. Whilst the MPIWS application has not been compared directly with an mpiJava implementation (due to the broadcast algorithms and the differences in transmitting objects and data types), an implementation of the application running on mpiJava gives similar results.

For the molecular dynamics simulation, the MPIWS implementation gives a slower, yet predictable, performance when compared with an mpiJava implementation. The MPIWS implementation of the molecular dynamics simulation still gives an efficiency of 61% when a 27,437 particle problem size is split between 8 services. This application also proves the ability of MPI applications to be simply ported to MPIWS services.

Chapter 8

Conclusions

Chapter Overview:

This chapter contains a review of the work that has been detailed in this thesis. In order to assess where this work fits in relation to current work in the area, this chapter compares MPIWS, the tool presented in this thesis, with similar tools and ideas that have been described in the related work section (Section 3.4). The chapter then goes on to critically appraise the evaluation methods used to assess the MPIWS tool, pointing out any limitations of the tests that must be considered. The conclusions are then presented and qualified.

8.1 Introduction

The hypothesis proposed in this Ph.D. thesis is:

> Web service component processes can communicate directly with each other, using Web service based communication protocols, to enable efficient parallel processing for MPI-style scientific applications, and to improve service based workflow throughput.

In order to prove this hypothesis, the work documented in this thesis examines the potential of using the Web service framework to provide support for MPI-style message passing communication. The uses of this style of communication can be separated into two sub-classes: MPI-style applications, and "direct messaging" between Web services in workflow executions. A background study, presented in Chapters 2 and 3, has shown that there is currently no complete methodology available to facilitate this style of communication, and therefore MPIWS has been designed. MPIWS is a novel tool that facilitates MPI-style communication between concurrently executing Web services. The communications between these services are transmitted over Web service protocols, and the communication operations that are provided by MPIWS include a subset of the MPI collective communication operations. Currently MPIWS is provided as an API to be used in the development of MPIWS services, which means that it is deployed as part of the applications deployment file. The following sections will relate MPIWS to current research in this area, then appraise the evaluation methods used in this research. Finally this chapter will present the final conclusions and contributions of the work undertaken.

8.2 Relation to Current Work

8.2.1 MPI-Style Applications

This thesis has described the development of MPIWS, a tool that provides the functionality for MPI-style Web services to communicate within a defined communication domain over the Web services framework. MPIWS is a tool that combines the flexibility and accessibility of the Web services architecture with the parallel processing ability of the MPI coding style. There are a lot of related methodologies that have addressed this combination of coding styles. These include the use of Web services to act as portals to MPI clusters [33, 68, 83, 91]. The motivation for MPIWS differs from this approach as the ability for services from multiple administrative domains to be included in the application is provided.

The use of Grid services to facilitate the configuration of MPI nodes across different administrative domains has also been extensively researched. MPICH-G2 [66] uses the Globus Grid middleware to set up communications channels between MPI nodes. This method requires the Globus toolkit to be present on all machines involved. Coti et al. [27] use Grid services as centralised brokers to facilitate the communications between administrative domains. This is a centralised approach whilst the MPIWS is decentralised once the invocation has been completed. Queiroz et al. [87] also use Grid services to set up MPI communications within a desktop grid environment. One of the biggest differences between the Grid services approaches and MPIWS is that in MPIWS the data is sent over Web service communication protocols, whereas in the Grid services implementations the data is sent over the underlying MPI communication protocols.

The use of Web services to provide the communication platform for the MPI message data has also been researched before. Krishnan et al. [69] suggested the use of notification standards. This idea was not directly used in the design of MPIWS as the notification was found to add a layer of complexity above using resources to store the message data. However, the idea of using statefull service methodologies has been used.

The work that is most related to this thesis was undertaken by Puppin et al. [85]. This work describes an approach for mapping MPI code to be run within a Web services architecture. Puppin et al.'s work describes a system where the send mechanism stores the data locally. The receive mechanism then invokes a service method on the sending machine which responds with the data; this is effectively a "pull" mechanism. This differs from the MPIWS methodology which is effectively a "push" mechanism: the sending service stores the message data in the remote receiving machine by invoking a store method, and the receiving machine then retrieves the data locally when it is needed. The push arrangement allows the transferral of message data before the receiving service is ready to use it. This is especially useful in collective operations such as the *Broadcast* as it avoids the necessity of waiting for all the services to synchronise before the operation can complete.

Puppin et al. have published two papers relating to this work [85, 86]. Neither of these papers mention the implementation of collective communication operations. Collective communications over the Web service framework is one of the most important distinctions of MPIWS. In this thesis the functionality provided by MPIWS to enable collective communications is designed and evaluated, and leads to a more complete message passing tool.

Another important difference between MPIWS and Puppin et al.'s work is the style of data encoding. MPIWS offers the option of encoding the data as serialised objects and then transmitting the serialised data as attachments within the MTOM mechanism. Puppin et al. mention the need for data encoding in their work but suggest XML-binary Optimized Packaging protocol (XOP) [55]. Work using this data format has not been published.

8.2.2 Execution of Workflows

Due to the current lack of tools that facilitate MPI-style direct communication, there is an absence of related work that evaluates the use of direct communication to enhance communication structures within workflow executions. Work that proposes an MPI-style of data composition is described in Montagnat et al. [77]. This work shows the potential for operations such as *Gather*, *Scatter* and *Reduce* to be used to collect or disperse data to a collection of services within a workflow. In Montagnat's description there is no mention of *direct messaging* between the services. However, in the workflow presented in Section 6.2.2 which contains services with multiple inputs, it has been shown that *direct messaging* could be used to perform this style of data composition.

8.3 Appraisal of Evaluation Procedures

The hypothesis posed in this thesis includes the phrase "Web service component processes can communicate directly with each other, using WS based communication protocols". This phrase on its own can be proved by the existence and functionality testing of MPIWS, the design of which has been extensively covered in this thesis. Within the hypothesis, the addition of the further phrase "to enable efficient parallel processing for MPI-style scientific applications", requires a greater level of evaluation regarding the performance of MPIWS's functionality.

Within the work carried out in this thesis, the method of assessing whether MPIWS is an efficient communications tool is to compare it against another MPI implementation. The performance of MPIWS's functionality has been compared against similar functionality in a competing MPI implementation. For this comparison we have decided to use mpiJava.

One of the advantages of using mpiJava is that there are two different methods of transferring the message data: the first is via serialised objects, and the second is by using the MPI defined datatypes. As the MPIWS tool uses serialisation to encode the message data, mpiJava's Object transfer provides a fair appraisal of the MPIWS performance for some of the functionality. This functionality includes the *Send*, *Receive*, *SendReceive*, and serial *Broadcast* operations. For other operations such as the binomial *Broadcast*, mpiJava does not provide a binomial distribution algorithm for the Object transfer. This is due to the complications of forwarding the buffer size message. The differences in the two systems makes the algorithmic collective communication operations difficult to compare fairly.

When the mpiJava message data is sent using the MPI defined data types, the mpiJava tool passes the data directly to the underlying MPICH implementation and the communication is handled directly by the C code. The efficiency of the C handling the primitive data types, compared with Java handling and serialising Objects, gives mpiJava a large advantage over MPIWS. Although this may be seen as an unfair test, it is very important to have a comparison with a top end MPI tool, as this method of testing does give a good indication of top MPI performance. Unfortunately there are added complications as for each collective communication operation, the underlying MPICH implementation uses different algorithms depending on message size. The MPIWS tool has not implemented some of these algorithms, for example, the *Scatter/Allgather* version of the *Broadcast*.

There are also collective operations that cannot be achieved using Object serialisation in the mpiJava implementation, such as the *Reduce* and *AllReduce* operations. The MPIWS versions of these operations can be compared only against the datatype transmission provided by mpiJava and its underlying MPICH.

It has been established that the evaluation of MPIWS against either type of data transfer is a challenging process, but the work discussed in this thesis has provided an extensive range of tests and has used many different data transfer methods. These challenges have been kept in mind while the conclusions were being formulated.

The final phrase of the hypothesis states that MPIWS will "improve Web service based workflow throughput". This claim has been investigated in Chapter 6. The style of data encoding chosen for the workflow data transfer was to retain the XML formatting of the messages. This choice enables a fair comparison with the standard workflow execution, although it could be possible to improve the performance of the *direct messaging* workflow execution by using the MPIWS option of serialising the data and sending it via MTOM.

The tests have been conducted on a local area network where the routing times between all the servers are similar. This allows a simplified analysis of the results, but the conclusions drawn from this analysis must take this into account. The potential benefit for workflows deployed in a more distributed environment will vary greatly depending on the bandwidth and latency of each communications link. The *direct messaging* approach will be more beneficial to executions where the workflow manager is located in a more remote part of the network from the majority of the services. This could be when the workflow manager is a mobile device.

8.4 Conclusions

MPIWS provides direct communication support and MPI-style message passing among Web services. This in turn provides the ability for MPI-style applications to fully exploit the modularity of the Web services environment. MPIWS could become a building block for the future development of execution environments for WS- and XML-based workflow languages, such as MPFL, that support WS-composite scientific applications.

From the tests undertaken, it has been discovered that despite using MTOM, a fast SOAP mechanism using SOAP-with-attachments, the overhead of SOAP messaging is significant enough to affect the performance of MPIWS when message sizes are small. However, when the message sizes reach a certain threshold, MPIWS runs at a similar, or even faster, speed compared with mpiJava passing serialised Objects. MPIWS tests can also run within approximately 120% of the time for mpiJava tests passing primitive data types over an underlying MPICH implementation. It has been found that the inter message pipe effect, a noticeable feature in applications that use consecutive MPIWS *Sends* as well as those with a distribution of receiving processors, contributes positively to the performance of MPIWS. The test results for the collective communication operations confirm that MPIWS is a practical and efficient way to integrate collective communications techniques into a Web services environment, although not all of the collective operations (especially the *Barrier* operation) are as efficient as could be hoped.

From the above observations, it can be concluded that using MPIWS for applications requiring MPI-style message passing between services is potentially a practical and efficient way of distributing coarse-grained parallel applications. It has also been shown that the use of collective communication techniques within the Web services architecture can significantly improve the efficiency of suitable applications such as the MolDyn simulation code.

MPI-style communication can be used to enhance the performance of Web service based workflow execution. The tests conducted have evaluated a range of the *direct messaging* functionality that could potentially be provided by the MPIWS tool. The evaluated functionality includes direct communication of data from the output of one service to the input of the next service in the workflow. This evaluation shows that *direct messaging* can improve the efficiency of Web service based workflow execution, especially if the workflow manager has a lower quality connection to the network. The direct communication of data from the output of a collection of services to a service that requires input data from multiple services has also been demonstrated. This demonstration shows that *direct messaging* workflow execution has the potential to perform MPI-style data composition communications.

The tests performed provide a proof of principal for the use of *direct messaging* to enhance the communications structure of Web service based workflow execution.

8.5 Summary

This chapter has outlined the final thoughts relating to the work undertaken in this Ph.D. thesis. It has provided evidence that the hypothesis has been proved and provides critical discussion on the evaluation methods undertaken to support this proof.

The main contributions made by this thesis are:

1. The demonstration that MPI-style applications can be executed over the Web services framework.

2. The demonstration of efficient collective communication techniques using Web services over the Web services framework.

3. The demonstration that *direct messaging* can improve the efficiency of certain Web service based workflows.

Chapter 9

Further Work

Chapter Overview:

The further work chapter briefly examines the work that this thesis presents and puts forward a view of where the research could go from this point. The future work can be separated into four sections: improvements to the MPIWS tool, support for message passing workflow languages, further research into the development of workflow execution, and research into different communication methodologies.

9.1 Introduction

The work presented in this thesis is a comprehensive investigation into the use of the Web services framework for MPI-style applications. The work also examines the potential of MPI-style communication to enhance the efficiency of Web service based workflow execution. Because the main goal for the research is not to develop a production quality MPI-style message passing tool for Web services, MPIWS has been designed primarily as a research aid. Therefore, there are a number of improvements that could be made to the MPIWS tool. These improvements are outlined in Section 9.2. The most logical step forward for this work is the integration of MPIWS with a message passing workflow language, as outlined in Section 9.3. Section 9.4 outlines suggestions, made by Glatard et al. [45], about the use of data composition patterns within the workflow structure. These suggestions provide an avenue of research into using collective communication techniques within the workflow execution environment. As a final direction that would be worth investigating, Section 9.5 looks briefly at multicast protocols that could be used to enhance the effectiveness of the collective communication operations.

9.2 The Functionality of MPIWS

The MPIWS tool is currently designed to prove the hypothesis that has been presented in this thesis. To make the tool more usable, there are a number of enhancements that could be made:

– To add more collective communication operations to MPIWS. For example the scatter operation could be achieved as long as the message data was provided in an object array. This added functionality may not conform directly to the MPI standard but could enhance the effectiveness of the MPIWS tool.

– To include the use of message tags. The inclusion of message tags would make message identification more conformant to the MPI standard.

– To add the ability to receive from "Any Source". Again, the inclusion of this functionality would increase the tools conformity to the MPI standard.

– To include collective communication operations for dynamically configured sub-domains. This would allow the enhanced communication functionality to be used for subsets of a whole workflow.

– To include the option of including a more reliable messaging protocol such as WS-reliability.

These enhancements to the MPIWS tool do not affect the conclusions of this thesis, but would provide a more usable implementation of a Web service based message passing tool.

9.3 Message Passing Workflow Languages

Message Passing Flow Language [60] is a workflow description language that can define *direct messaging* between concurrently executing Web services. A longer term research goal is to investigate the integration of MPI-style Web services with a MPI-style workflow language such as MPFL to produce an execution environment for MPI-style workflows.

9.4 Development of Workflow Execution

Point-to-point *direct messaging* has been proved to enhance the performance of certain Web service based workflow executions (see Chapter 6). Glatard et al. [45] mention the use of data composition patterns being used in data intensive workflows, and some of these patterns could be equated to MPI operations such as the *one to all* which is effectively a *broadcast*. The use of data composition patterns is beneficial in the formulation of data sets produced from a distribution of services. By definition, the use of MPIWS's collective communication functionality will allow this to be achieved. In order to explore the potential of using data composition patterns, as described in Glatard et al. [45], the full range of collective operations should be implemented and evaluated for use within workflow execution.

9.5 Communication Methodologies

Methods of network multicast have been researched for a long time, Boivie et al. [14] and Shin et al. [92] are examples of methods to multicast data around small multicast groups at the Internet Protocol level. Phan et al. [82] looks at using a SOAP multicast protocol called Similarity-Based SOAP Multicast Protocol (SMP). This protocol groups similar SOAP messages, requiring only one message to be sent from the originating client or service. This method of combining SOAP messages, which can be used to avoid excessive network traffic, could be useful in reducing the cost of collective operations. It would be especially applicable to operations such as *broadcast*, where the message data to each recipient is identical. Phan et al. [82] report that the reduction in network traffic using this approach can be up to 70% but there is a 10% loss in response time. Another problem is that the internet routers must be configured to parse the SOAP body's SMP header. This involves the standard being well recognised before adoption can be widespread.

9.6 Summary

The further work chapter outlines directions in which the research in this thesis could be taken. These directions include: the improvement of the MPIWS tool, the integration of MPIWS with workflow managers, further research into the development of workflow execution, and looking at communication methodologies that are at lower layers in the communication protocol stack.

Appendix A

XSD documents

A.1 ServiceConfig.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://http://service....imc/xsd">
 <xs:element name="workflowCFG">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="in">
     <xs:complexType>
      <xs:choice>
       <xs:element name="fromRank" type="xs:integer"
          maxOccurs="unbounded"></xs:element>
       <xs:element name="client" type="xs:string"
          maxOccurs="1"></xs:element>
       <xs:element name="operation" type="xs:string"
          maxOccurs="1"></xs:element>
      </xs:choice>
     </xs:complexType>
    </xs:element>
    <xs:element name="out">
     <xs:complexType>
      <xs:choice>
       <xs:element name="toRank" type="xs:integer"
          maxOccurs="unbounded"></xs:element>
       <xs:element name="client" type="xs:string"
          maxOccurs="1"></xs:element>
       <xs:element name="operation" type="xs:string"
          maxOccurs="1"></xs:element>
      </xs:choice>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

A.2 MPIWSRun.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://service......imc/xsd">
 <xs:element name="run">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="mpi_wsData">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="eprList">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="epr" type="xs:string"
             maxOccurs="unbounded"></xs:element>
         </xs:sequence>
         <xs:attribute name="eprLength" type="xs:integer
            "/>
        </xs:complexType>
       </xs:element>
       <xs:element name="rank" type="xs:integer"></xs:
          element>
       <xs:element name="iD" type="xs:integer"></xs:
          element>
       <xs:element name="reportingMode" type="xs:string
          "></xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="appData">
     <xs:complexType>
      <xs:sequence>
       <xs:any minOccurs="0"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

A.3 MPIWSStore.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://http://service...imc/xsd">
 <xs:element name="store">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="data">
     <xs:complexType>
      <xs:sequence>
        <xs:element name="msgNo" type="xs:integer"></xs:
           element>
        <xs:element type="xs:base64Binary"></xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
     <xs:element name="iD" type="xs:integer"></xs:element
        >
    <xs:element name="msgTag" type="xs:integer"></xs:
       element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

A.4 MPIWSBstore.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://service...imc/xsd">
 <xs:element name="bStore">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="data">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="msgNo" type="xs:integer"></xs:
          element>
        <xs:complexType>
         <xs:sequence>
          <xs:element name="No" type="xs:integer"
             maxOccurs="unbounded"></xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element type="xs:base64Binary"></xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="iD" type="xs:integer"></xs:element>
    <xs:element name="msgTag" type="xs:integer"></xs:
       element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Bibliography

[1] Asif Akram, David Meredith, and Rob Allan. Evaluation of BPEL to Scientific Workflows. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 269–274, Washington, DC, USA, 2006. IEEE Computer Society.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an Extensible System for Design and Execution of Scientific Workflows. In *Scientific and Statistical Database Management, 16th International Conference on*, pages 423–424, 2004.

[3] Apache. Apache Axis 2.0. Web site: http://ws.apache.org/axis2/. Accessed Sept 2009.

[4] The Apache Software Foundation. *MTOM Guide -Sending Binary Data with SOAP*, 1.0 edition, May 2005.

[5] M.A. Baker and D.B. Carpenter. MPJ: A Proposed Java Message-Passing API and Environment for HighPerformance Computing. In *the Proceedings of the 2nd Java Workshop at IPDPS 2000*, pages pp 552 – 559. LNCS, Springer Verlag, Heidelberg, Germany, May 2000.

[6] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpiJava: An Object-Oriented Java interface to MPI. In *International Workshop on Java for Parallel and Distributed Computing IPPS/SPDP*, April 1999.

[7] Mark Baker, Bryan Carpenter, and Aamir Shafi. *An Approach to Buffer Management in Java HPC Messaging*, volume Volume 3992/2006 of *Lecture Notes in Computer Science*, pages 953–960. Springer Berlin / Heidelberg, May 2006.

[8] Mike Barnett, Lance Shuler, Satya Gupta, David G. Payne, Robert van de Geijn, and Jerrell Watts. Building a High-Performance Collective Communication Library. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 107–116, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[9] Blaise Barney. Message Passing Interface (MPI). Internet Tutorial: www.llnl.gov/computing/tutorials/mpi/. Accessed Aug 2007.

[10] John J. Barton, Satish Thatte, and Henrik Frystyk Nielsen. SOAP Messages with Attachments. Technical report, http://www.w3.org/TR/SOAP-attachments, W3C, Dec. 2000.

[11] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki. An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 345–354, New York, NY, USA, 2004. ACM Press.

[12] Olivier Beaumont, Loris Marchal, and Yves Robert. Complexity Results for Collective Communications on Heterogeneous Platforms. *Int. Journal of High Performance Computing Applications*, 20:5–17, 2006.

[13] T Berners-Lee, J Hendler, and O Lassila. The SemanticWeb: A New Form of Web Content that is Meaningful to Computers will Unleash a Revolution of New Possibilities. *Scientific American*, May 2001.

[14] Rick Boivie, Nancy Feldman, and Christopher Metz. Small Group Multicast: A New Solution for Multicasting on the Internet. *IEEE Internet Computing*, 4(3):75–79, 2000.

[15] Shawn Bowers and Bertram Ludscher. Actor-Oriented Design of Scientific Workflows. In *24 Intl. Conference on Conceptual Modeling*. Springer, 2005.

[16] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report, http://www.w3.org/TR/REC-xml/, W3C, 2008.

[17] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[18] B Carpenter, G Fox, S Ko, and S Lim. The mpiJava Project. www.hpjava.org/mpiJava.html, October 1999.

[19] Bryan Carpenter. Java for High Performance Computing: MPI-based Approaches for Java. Pervasive Technology Labs, Indiana University. Internet presentation. http://www.hpjava.org/courses/arl/lectures/mpi.ppt, Accessed Aug 2007.

[20] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 66–71, New York, NY, USA, 1999. ACM.

[21] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12 Issue 11:1019 – 1038, 2000.

[22] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report,http://www.w3.org/TR/wsdl, W3C, March 2001.

[23] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming Scientific and Distributed Workflow with Triana Services. *Grid Workflow 2004 Special Issue of Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[24] Brian Cohen. BitTorrent Protocol Specification. First Workshop on Economics of Peer-to-Peer Systems (P2P03). 2003.

[25] Ian Cooper and Yan Huang. The Design and Evaluation of MPI-Style Web Services. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS (1)*, volume 5101 of *Lecture Notes in Computer Science*, pages 184–193. Springer, 2008.

[26] Ian Cooper and Coral Y. Walker. The Design and Evaluation of MPI-Style Web Services. *IEEE Transactions on Services Computing*, 2(3):197–209, 2009.

[27] C. Coti, T. Herault, S. Peyronnet, A. Rezmerita, and F. Cappello. Grid Services for MPI. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 417–424, May 2008.

[28] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[29] Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, Version 1.0. Technical report,http://www.ibm.com/developerworks/library/specification/ws-bpel/, IBM, July 2002.

[30] Karl Czajkowski, Donald F Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Igor Sedukhin, David Snelling, Steve Tuecke, and William Vambenepe. The WS-Resource Framework Version 1.0. Technical report, Globus Alliance and IBM, 2004.

[31] Richard C Demmel. Lecture Notes Parallel Matrix Multiplication CS267. Technical report, University of California at Berkeley, 1996. available at: www.cs.berkeley.edu/~demmel/cs267/lecture11/lecture11.html.

[32] DL_Poly. Web site: www.cse.scitech.ac.uk/ccg/software/DL_POLY/index.shtml. Accessed Sept 2009.

[33] Jan Dünnweber, Anne Benoit, Murray Cole, and Sergei Gorlatch. Integrating MPI-Skeletons with Web Services. In *Proceedings of the International Conference on Parallel Computing*, pages 787–794, 2005.

[34] T. Fahringer, S. Pllana, and A. Villazon. A-GWL: Abstract Grid Workflow Language. In *International Conference on Computational Science, Programming Paradigms for Grids and Metacomputing Systems*, Krakow, Poland, 2004. Springer-Verlag.

[35] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Jr, and Hong-Linh Truong. ASKALON: a Tool Set for Cluster and Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(7-8):143–169, 2005.

[36] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[37] I. Foster. Designing and Building Parallel Programs. Technical report, Argonne National Laboratory, 1995.

[38] I. Foster and N.T. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *SC98. IEEE/ACM Conference on Supercomputing*, page 46. IEEE Computer Society, 1998.

[39] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An Interoperability Toolkit for Parallel and Distributed Computer Systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1993.

[40] N. Freed and N. Borenstein. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, Nov. 1996.

[41] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[42] Jean-loup Gailly, Mark Adler. GZIP. http://www.gzip.org/, Accessed April 2010. 1991.

[43] Vladimir Getov, Paul Gray, and Vaidy Sunderam. MPI and Java-MPI: Contrasts and Comparisons of Low-level Communication Performance. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 21, New York, NY, USA, 1999. ACM Press.

[44] T. Glatard, J. Montagnat, and X. Pennec. Efficient Services Composition for Grid-enabled Data-intensive Applications. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 333–334, 2006.

[45] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and Efficient Workflow Deployment of Data-Intensive Applications On Grids With MOTEUR. *International Journal of High Performance Computing Applications*, 22(3):347, 2008.

[46] B.D. Goodman. Squeezing SOAP. Technical report, http://www.ibm.com/developerworks/webservices/library/ws-sqzsoap.html, accessed April 2010, IBM, 2003

[47] Steve Graham, Anish Karmarkar, Jeff Mischkinsky, Ian Robinson, and Igor Sedukhin. *Web Services Resource 1.2 (WS-Resource) Public Review Draft 01.* OASIS, June 2005.

[48] Steve Graham, Peter Niblett, Dave Chappell, Amy Lewis, Nataraj Nagaratnam, Jay Parikh, Sanjay Patil, Shivajee Samdarshi, Igor Sedukhin, David Snelling, Steve Tuecke, William Vambenepe, and Bill Weihl. Publish-Subscribe Notification for Web services. Technical report, WSRF, May 2004.

[49] Steve Graham, Peter Niblett, Dave Chappell, Amy Lewis, Nataraj Nagaratnam, Jay Parikh, Sanjay Patil, Shivajee Samdarshi, Igor Sedukhin, David Snelling, Steve Tuecke, William Vambenepe, and Bill Weihl. Web Services Base Notification, May 2004.

[50] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of the Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, Sept 1996.

[51] William Gropp. Tutorial on MPI: The Message-Passing Interface. Internet tutorial: www.new-npac.org/projects/cdroms/cewes-1998-05/reports/gropp-mpi-tutorial.pdf. Accessed 2007.

[52] William Gropp and Ewing L. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, London, UK, 1999. Springer-Verlag.

[53] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, W3C, April 2007.

[54] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Herv Ruellan. SOAP Message Transmission Optimization Mechanism. Technical report, W3C, January 2005.

[55] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Herv Ruellan. XML-binary Optimized Packaging. Technical report, W3C, 2005.

[56] John L. Gustafson. Fixed Time, Tiered Memory, and Superlinear Speedup. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5) Charleston, South Carolina*, 1990.

[57] B Harrington, R Brazile, and K Swigger. SSRLE: Substitution and Segment-Run Length Encoding for Binary Data in XML. In *Information Reuse and Integration, 2006 IEEE International Conference on*, pages 11–16, Sept. 2006.

[58] David Henty. Message Passing Programming Edinburgh Parallel Computing Centre, Lecture notes MSc HPC 2006.

[59] Roger W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.*, 20(3):389–398, 1994.

[60] Yan Huang and Qifeng Huang. WS-Based Workflow Description Language for Message Passing. In *5th IEEE International Symposium on Cluster Computing and Grid Computing*, Cardiff, Wales, U. K, 2005.

[61] Yan Huang and David. W. Walker. Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid. In *Lecture Notes in Computer Science*, volume 2659 / 2003, pages 254 – 263. Springer-Verlag GmbH, Jan 2003.

[62] Duncan Hull, Katherine Wolstencroft, Robert Stevens, Carole Goble, Matthew Pocock, Peter Li, and Thomas Oinn. Taverna: a Tool for Building and Running Workflows of Services. *Nucleic Acids Research*, 34(Web Server issue):729–732, July 2006.

[63] IBM. Web Services Flow Language (Specification). Technical report, IBM, May 2001.

[64] Intel. Intel MPI Benchmarks. Technical report, Intel, June 2006.

[65] Deepal Jayasinghe. Invoking Web Services using Apache Axis2. Web site: www.today.java.net/pub/a/today/2006/12/13/invoking-web-services-using-apache-axis2.html, Dec 2006. Accessed Aug 2007.

[66] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[67] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. *Lecture Notes in Computer Science*, 1800:1176, 2000.

[68] S. Krishnan, B. Steam, K. Bhatia, K.K. Baldridge, W.W. Li, and P. Arzberger. Opal: SimpleWeb Services Wrappers for Scientific Applications. IEEE International Conference on Web Services, pages 823–832, Sept. 2006.

[69] Sriram Krishnan, Patrick Wagstrom, and Gregor Von Laszewski. GSFL: A Workflow Framework for Grid Services. Technical report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, 2002.

[70] Alp Kut and Derya Birant. An Approach for Parallel Execution of Web Services. In *Proceedings - IEEE International Conference on Web Services*, pages 812–813. IEEE Computer Society, June 2004.

[71] Kelvin Lawrence and Chris Kaler. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). Technical report, OASIS, February 2006.

[72] Steve Loughran. Fear of Attachments. Web site: http://www.mail-archive.com/axis-user@xml.apache.org/msg08732/Fear_of_Attachments.pdf, Feb 2003. Accessed 2/05/07.

[73] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, icl.cs.utk.edu, March 2005.

[74] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-Based Enterprise Grid Computing System. In *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*, 2005.

[75] Sava Mintchev and Vladimir Getov. Towards Portable Message Passing in Java: Binding MPI. In *Proceedings of the 4th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 135–142, London, UK, 1997. Springer-Verlag.

[76] Moldy. Web site: www.ccp5.ac.uk/moldy/moldy.html. Accessed Sept 2009.

[77] Johan Montagnat, Tristan Glatard, Isabel Campos Plasencia, Francisco Castejn, Xavier Pennec, Giuliano Taffoni, Vladimir Voznesensky, and Claudio Vuerli. Workflow-Based Data Parallel Applications on the EGEE Production Grid Infrastructure. *Journal of Grid Computing*, 6:369–383, 2008.

[78] MPI-Forum. MPI : A Message-Passing Interface Specification. Technical report, Message Passing Interface Forum, 1994.

[79] Mo Mu and John R. Rice. Modeling with Collaborating PDE solvers–Theory and Practice. *Computing Systems in Engineering*, 6(2):87 – 95, 1995.

[80] K Park, S Park, O Kwon, and Park H. MPICH-GP: A private-IP-enabled MPI over Grid Environments. In *Parallel and Distributed processing and applications, Lecture Notes in Computer Science*, volume 3358, pages 469–473. Springer-Verlag, 2004.

[81] Srinath Perera and Ajith Ranabahu. Web Services Messaging with Apache Axis2: Concepts and Techniques. Web site: www.onjava.com/pub/a/onjava/2005/07/27/axis2.html, July 2005. Accessed Aug 2007.

[82] Khoi Ahn Phan, Zahir Tari, and Peter Bertok. Similarity-Based SOAP Multicast Protocol to Reduce Bandwidth and Latency in Web Services. *IEEE Transactions on Services Computing*, 1(2):88–103, 2008.

[83] Marlon Pierce and Geoffrey Fox. Scientific Applications as Web Services: A Simple Guide. Web site: http://grids.ucs.indiana.edu/ptliupages/publications/cise_WSforeScience.pdf, 2003. Accessed Sept 2009.

[84] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance Analysis of MPI Collective Operations. *Cluster Computing*, Volume 10(Number 2):127–143, June 2007.

[85] D. Puppin, N. Tonellotto, and D Laforenza. Using Web Services to run Distributed Numerical Applications. In *11th EuroPVM/MPI2004*, pages 19–22. Springer LNCS 3241, Sept 2004.

[86] D. Puppin, N. Tonellotto, and D. Laforenza. How to run Scientific Applications over Web Services. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 29 – 33, 2005.

[87] Carlos Queiroz, Marco A. S. Netto, and Rajkumar Buyya. Message Passing over .NET-based Desktop Grids. In *Proceedings of the Workshop on*

Cutting Edge Computing, in conjunction with the 13th IEEE International Conference on High Performance Computing (HiPC'06), 2006.

[88] Rolf Rabenseifner. Optimization of Collective Reduction Operations. In M.Bubak et al., editor, *Computational Science - International Conference on Computational Science 2004*, volume 3036/2004, pages 1–9. Springer Berlin / Heidelberg, 2004.

[89] Ala Rezmerita, Tangui Morlier, Vincent Neri, and Franck Cappello. Private Virtual Cluster: Infrastructure and Protocol for Instant Grids. In *Euro-Par 2006 Parallel Processing*. Springer Berlin / Heidelberg, 2006.

[90] M. Ruth, Feng Lin, and Shengru Tu. Adapting Single-request/Multiple-response Messaging to Web Services. In *Computer Software and Applications Conference, 29th Annual International*, volume 2, pages 287 – 292, 2005.

[91] P. Sajjipanon and S. Ngamsuriyaroj. Web Services for MPI-Based Parallel Applications on a Rocks Cluster. In *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pages 265–270, Dec. 2008.

[92] Myung-Ki Shin, YJ Kim, KS Park, and SH Kim. Explicit Multicast Extension (Xcast+). *ETRI journal*, 23(4):202–204, 2001.

[93] Lorna Smith. MolDyn. Technical report, Edinburgh Parallel Computing Centre, 2001.

[94] James Snell. Part 4: Introducing WSFL. Technical report, IBM, June 2001.

[95] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, 1998.

[96] B Sotomayor. The Globus Toolkit 4 Programmer's Tutorial. Web Site: www.gdp.globus.org/gt4-tutorial/multiplehtml/index.html, 2004. Accessed Sept 2009.

[97] I. Taylor, E. Deelman, D. Gannon, and M. Shields. *Workflows for e-Science.* Springer, 2008.

[98] Ian Taylor. *From P2P to Web services and Grids.* Springer, 2005.

[99] TeraGrid. Web site: www.teragrid.org. Accessed Sept 2009.

[100] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[101] Satish Thatte. XLANG Web Services for Business Process Design, 2001.

[102] WeiQin Tong, Hua Ye, and WenSheng Yao. PJMPI: Pure Java Implementation of MPI. *High-Performance Computing in the Asia-Pacific Region, International Conference on*, 1:533, 2000.

[103] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically Tuned Collective Communications. In *Supercomputing 2000: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[104] W3C. XML Tutorial. Web tutorial: www.w3schools.com/xml/default.asp. Accessed Sept2009.

[105] W3C. Web Service Architecture. Technical report, W3C Working group, February 2004.

[106] W3C. Web Services Addressing (WS-Addressing). Technical report, August 2004.

[107] David W. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Comput.*, 20(4):657–673, 1994.

[108] David W. Walker, and Yan Huang. Workflows: Representation and Semantics. *Distributed Programming Abstractions: Workshop II*, e-Science Institute, Edinburgh, UK. http://wiki.esi.ac.uk/w/files/c/c0/Talk_DWW.pdf, 2007.

[109] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[110] Ying Ying, Yan Huang, and David W Walker. Using SOAP with Attachments for e-Science. In *Proceedings of the UK e-Science All Hands Meeting 2004*, Aug. 2004. Poster.

[111] Jianting Zhang, Ilkay Altintas, Jing Tao, Xianhua Liu, Deana D. Pennington, and William K. Michener. Integrating Data Grid and Web Services for E-Science Applications: A Case Study of Exploring Species Distributions. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 31, Washington, DC, USA, 2006. IEEE Computer Society.

CARDIFF UNIVERSITY ★ PRIFYSGOL CAERDYDD ★