The Design and Evaluation of MPI-Style Web Services

Ian Cooper and Yan Huang

School of Computer Science, Cardiff University, United Kingdom
{i.m.cooper, yan.huang}@cs.cardiff.ac.uk

Abstract. This paper describes how Message Passing Web Services (MPWS) can be used as a message passing tool to enable parallel processing between WS-based processes in a web services oriented computing environment. We describe the evaluation tests performed to assess the point-to-point communications performance of MPWS compared to mpiJava wrapping MPICH. Following these evaluations we conclude that: using web services to enable parallel processing is a practical solution in coarse grained parallel applications; and that due to inter message pipelining, the MPWS system can, under certain conditions, improve on the communication times of mpiJava.

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. BPEL4WS is commonly used for web service based scientific workflow compositions [1], but users are limited to applications with non-interdependent processes. Furthermore, issues relating to the unsatisfactory performance of SOAP messaging have tended to inhibit the wide adoption of web service technologies for high performance distributed scientific computing. In spite of the performance concerns, the use of web service architectures to build distributed computing systems for scientific applications has become an area of much active research. Recently developed workflow languages have started addressing the problem of intercommunicating processes, Grid Services Flow Language (GSFL) [2] is one example; it provides the functionality for one currently executing Grid service to communicate directly with another concurrently executing Grid service. Another example is Message Passing Flow Language (MPFL) [3], this specifies an XML based language that enables web service based workflows using MPI-style send and receive commands, to be described. Neither of the examples mentioned above have presented a workflow engine and currently there is no workflow engine that supports MPI

style direct message passing; the GSFL paper describes an implementation using OGSA notification ports in a subscriber producer methodology, but the MPFL remains a draft language with no implementation details.

In this paper, we investigate the potential and suitability of using a web services infrastructure to support parallel applications that require MPI-like message passing. We look at various methods and tools that can be used to implement these message exchange patterns (MEPs) and assess the suitability of previous work, within the web service framework, for this emerging workflow use. We then propose an implementation for Message Passing Web Services (MPWS) and present performance results comparing MPWS against mpiJava [4]; a leading hpc Java implementation [5]. We have used mpiJava as it is a tool for distributed computing rather than for use within a cluster environment; MPWS combines distributed, loosely coupled services to form a temporary, tightly coupled application with a similar goal. There has also been much research to compare mpiJava to other HPC systems [6].

2 Background and Related Research

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 [7]. 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, this can be translated to the act of sending data from one executing service to another, concurrently executing, service. The problem here is that a service can be concurrently invoked many times; once a service is invoked, there must be a way of determining which instance of the service needs to receive the message.

SOAP based web services communicate via SOAP messages, and these messages are exchanged in a variety of patterns. Within the WS framework there is normally a simple Message Exchange Pattern (MEP) that involves either a request only, or a request and response message. The normal invocation of a service during the execution of a workflow is for the workflow manager to request a service and then, when the service has completed, a response is returned to the workflow manager. It can be seen that this requires mediation by the workflow manager at every step of the workflow process.

Kut and Birant [8] 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 and can be extended (as shown in Fig. 1) to allow these services to exchange data directly, this removes the need for the workflow manager to intervene every time a process transfers data [2].

Currently there is no standard for directly passing data from one service to another running service. Alternative MEPs are in various stages of research; in-only patterns are in common usage in most web service platforms, and research

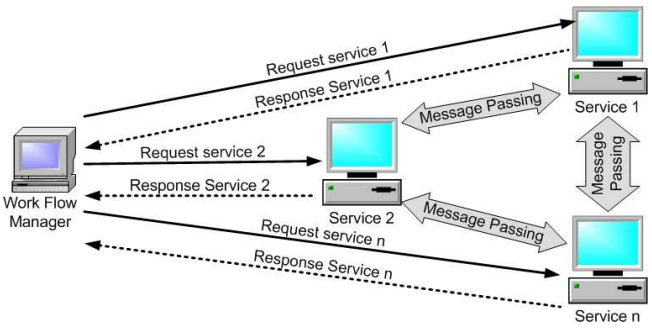

Fig. 1. Extending the use of parallel executing services to perform message passing.

has been undertaken into a single request multiple response (SRMR) MEP [9]. In this framework for SRMR an agent is used to relay the service call, and a centralized web service collects the responses.

Research into the use of web services in parallel computations is presented by Puppin et al [10], who developed an approach for wrapping MPI nodes within web services. Their paper shows that the performance of wrapped MPI nodes can be comparable with MPI running in a cluster environment, although, many more computers are required for the wrapped MPI version.

In our research, we focus on developing and evaluating web services that are capable of MPI–like communication with other services; the performance of SOAP messaging is a key issue in determining if MPWS can be made comparable in performance with other distributed message-passing systems.

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 space overhead for the message is potentially very large. The most efficient method of encoding data is to serialize 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 we cannot translate a binary file directly to string format, as there are not enough characters available. There are four solutions available to this problem; binary-to-character encoding [11], packaging, binary XML encoding [12], and linking [11].

Packaging, such as SOAP with Attachments (SwA) [13], or Message Transmission Optimization Mechanism (MTOM) [14] allows data to be transmitted externally to the SOAP envelope. A comparison of transmission speeds using SOAP with Attachments and true XML formatting is given in [15]. MTOM also stores the data within the object model.

MTOM has been chosen as the transmission protocol for these messages as it is SOAP based; yet it increases the speed of the data by allowing attachments, while keeping the data accessible in the object model. MTOM does not have the

coding overheads of either the binary to character or the binary XML encoding, and it stays within the SOAP communication protocols, unlike linking.

3 The Design of a Message-Passing Web Service

The challenge is to design a tool which will combine the tightly coupled programming concept like MPI and the distributed, loosely coupled architecture of SOAP web services; to do this we need to adhere to WS and SOAP messaging standards whilst providing an efficient form of communication between services. MPWS is designed to address three areas; the creation of a set of services, the initialisation of those services so they are aware of each other, and the communication between the services.

The creation of a set of services is achieved by the workflow manager, its role is to accept jobs, normally specified using an XML-based workflow language such as MPFL, then find a collection of suitable services for those jobs and invoke them all within a unique communication domain. A communication domain is a collection of service instances which are involved in the same composite application, and can communicate directly with each other; this means that each service instance must be aware of all other service instances in the domain. Based on the job definition, the workflow manager will discover and select a group of suitable Message Passing (MP) web services using standard WS techniques, then generate a communication domain ID for the workflow application. The workflow manager can then specify the rank number and invoke a *run* method for each MP service involved.

The initialisation of the service is performed in the invocation of the *run* method; the input data for the application as well as the binding information for the services to work together, is passed to each individual service that is involved in the same workflow application. The binding information includes: communication domain ID; the rank number for the service; and a list of service endpoint references, each associated with a particular rank ID. Knowing the rank number as well as the service endpoint references, will allow the service to perform point-to-point message passing with all other services in the same communication domain.

An MP web service can participate in multiple applications concurrently, so in order to solve the problem of identifying which service invocation is to be addressed, there is one communication domain established for each application instance; this is associated with a unique identifying number - the Communication Domain ID. Each MP web service instance belongs to a communication domain, and each service instance has an associated resource; this resource is identified by the Communication Domain ID, is initiated for the particular communication domain, and stores the binding information and messages sent to that service instance. WS-Resources are defined in the WSRF specifications [16], they allow for the concept of state within web services. A resource is uniquely identifiable and accessible via the web service [17]. The use of resources provides message buffers for an MP web service. Instead of sending

and receiving the messages synchronously, the message is sent to the resource associated with the receiving web service instance, then the receiving web service can retrieve a particular message from the corresponding resource. A message is associated with a communication domain ID and a message tag; this will ensure that the message can be identified within a communication domain.

MPWS has been designed to conform to WS Standards and to SOAP messaging standards, to allow the use of loosely coupled services in a traditionally tightly coupled MPI coding style. To this end we have designed MPWS to support multi-layer interfaces; the upper layer as a WS layer, and the lower layer as a message-passing (MP) layer. With the web service layer, an MP web service supports WSDL standards, providing loosely coupled services which can be easily published, discovered and reused. There are two main methods exposed via the web services interface:

- *Run* method - this mainly consists of a sequence of instructions so that it performs one or more particular tasks. Since an MP web service normally involves cooperation with other MP web services for a particular application, setting up communication domains is the first task when the run method is invoked
- *Store* method - this receives messages sent from other MP web services and stores them to the resource associated with the MP web service instance.

With the message passing layer, an MP web service is able to conduct message-passing communication with other MP web services by supporting message-passing interfaces, including *send, receive, broadcast, and sendReceive.* The message-passing interfaces are not exposed via WSDL, but are low-level interfaces that can only be invoked via the WSDL-level methods. For example, inside a run method body, there may be instructions such as sending data to a particular MP web service or receiving data from a particular MP web service, and these can be carried out by directly invoking the methods provided within the message-passing programming package, MTOM is used as the transmission protocol in this layer.

Fig. 2(a) gives an example which shows a send operation scenario between two MP web services, A and B. A communication domain was initiated with the communication domain ID equal to 3303. Service A sends a message to service B within the communication domain. The send method from the MP service is called to send the message to service B. This is done by invoking the store method provided by service B. When the store method is called, it stores the message it received into the resource associated with the domain ID 3303. Although service B has received the message and stored it within one of its associated resources, the message cannot be used unless a receive method is called. The receive method retrieves this message from the resource (ID = 3303) associated with the service instance, the tag name associated with the message is used to identify the particular message within the communication domain (Fig. 2(b)).

The use of the resource to provide a buffering service for message passing encourages the adoption of the asynchronous *fire-and-forget* style [18] of message

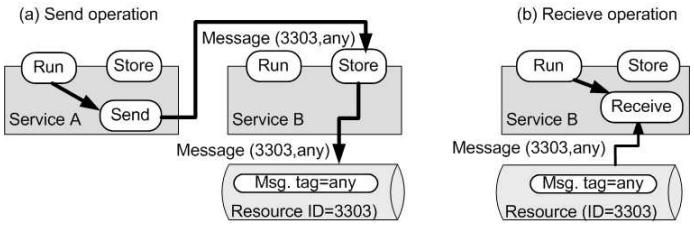

Fig. 2. An example of sending a message from Service A to Service B.

sending which is supported in AXIS 2.1.1. The fire-and-forget send method returns immediately after the existence of the receiving host is confirmed providing increased performance over the sendRecieve or sendRobust style .

4 The Evaluation

4.1 Testing

Many benchmark suites have been devised and put forward as the definitive parallel computing benchmark tests ([19],[20]), many of these are designed to test the underlying hardware or the collective communications features of the message passing tools. The purposes of the tests that are to be performed on MPWS and mpiJava are to find the speed of the communication implementations and not the capabilities of the network.

The ping pong test is used in most of the bench mark suites as a simple bandwidth and latency test. Getov et. al. [21] used a number of variations of the ping pong test to compare the performance of MPI and java-MPI, also Foster and Karonis [22] use the ping pong test to evaluate MPICH-G, a grid enabled MPI. It has been decided to use two variations of the ping pong tests. The first, PingPong, transfers data from one process to another and then back again. In this test, there are an even number of processors within the communication domain that are paired up to concurrently pass data to and from each other, see Fig. 3(a). In this figure the messages are represented by the solid arrows, the time taken for the message to be sent from one service to a second service and then back again is measured as the round trip time.

The second test is the Ping*Pong test [21], this test involves sending multiple messages from one service to a second service before the second service returns a message, this is also seen in Fig. 3(b). This test will differentiate 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 processing the next message [21]. The ping*pong test may show more a realistic view of the systems performance, as it emulates many real applications of message passing (such as a matrix multiplication).

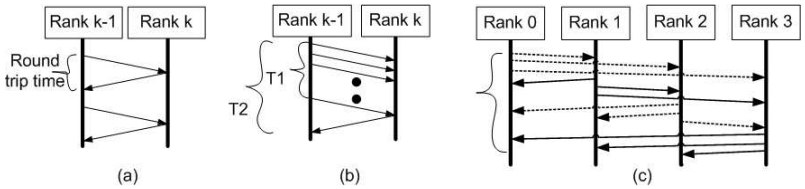

Fig. 3. Communication Diagram for PingPong, Ping*Pong and matrix multiplication tests.

As a further test that has a more real life application to it, a one dimensionally blocked parallel matrix multiplication application is used. This application is based on a simple parallelisation of the matrix multiplication problem. The communications for the matrix multiplication application are shown in Fig. 3(c), each arrow represents a portion of the matrix being sent from rank(i) to another processor. It is important to note that while the order of the sends for each rank are fixed, a rank can start sending its data as soon as it has received data from the preceding rank.

For the matrix multiplication application, the actual multiplication calculations are extremely time consuming and dilute the performance of the communications with variances in processor utilisation at the time of testing. We have, therefore omitted the calculation part of the application and only presented the communication part.

4.2 Evaluation Results and Discussion

Versions of each test have been written and evaluated as both a web service, running on Tomcat 5.5.20 using AXIS 2.1.2, and in Java using the mpiJava API (V1.2 wrapping MPICH 1.2.6); all code was written in Java 1.6.0. The MPWS evaluation tests are undertaken on a public network of university machines, all of which are prone to unforeseen activity. The tests were done during low usage hours to reduce inconsistencies and all graphs show minimum timings to reduce the impact of the network on the results; the error bars show maximum timings over the set of tests. The Linux machines used for the testing have twin Intel pentium 4, 2.8GHz processors; in order to eliminate the discrepancy's between the different handling of threads with the MPWS and mpiJava systems, both systems were restrained to using only one processor on each machine.

The graphs in Fig. 4 and Fig. 5 show the timings of MPWS and mpiJava running the ping pong tests. The results show the expected communications overhead of the SOAP message, that degrades the performance for smaller messages, but they also show that over a message data size threshold of approximately 200Kbytes (or n = 160) the extra communication overhead has been absorbed by the total MPWS communication time to make the MPWS and MPI systems run at a relatively similar speed.

The graph in Fig. 5 concentrates on the timings for smaller message sizes, allowing the reader to easily compare the two systems. The ping pong test shows

that for large message sizes the MP web services are an acceptable alternative to mpiJava, but below the data sizes of around 125Kbytes, the systems overheads are very noticeable. This is not really unexpected, as the there are the overheads of the SOAP headers and the HTTP protocol to consider.

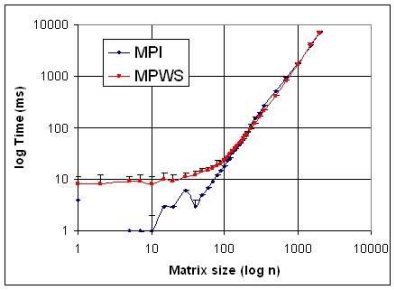

Fig. 4. Times of Ping Pong test MPWS and mpiJava.

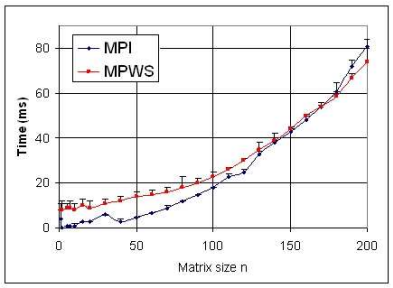

Fig. 5. Times of Ping Pong test MPWS and mpiJava; small message sizes.

The results for the ping*pong test are shown in Fig. 6, it is noticed that the threshold (n=130) for MPWS absorbing the overhead of the SOAP messages is slightly lower than with the PingPong test. More significant, is the tenancy for MPWS to outperform the version using mpiJava's standard send; we put this down to the inter message pipeline effect and the buffer handling of the two different systems.

The parallel matrix multiplication communication results are shown in Fig. 7, they consistently show that the MPWS performs the communications faster than mpiJava at matrix sizes above the overhead threshold. We again put the results of the matrix test down to the application of the system buffers in the MPWS and mpiJava implementations, and the inter message pipeline effect. In the ping*pong test, both the inter message pipeline of the send and receive were being tested, but in the matrix multiplication test, each of the consecutive sends from every processor are being received by a different processor. In MPWS, the main message buffering occurs in the receiving processor. This distributes the message buffering process at the time of high utilisation.

5 Conclusion and Further Work

From the tests we have discovered that despite using MTOM, The overhead of SOAP messaging is still a problem which affects the performance of MPWS when message sizes are small. However, when the message sizes reach a threshold, MPWS and mpiJava systems run at a relatively similar speed. We also found that the inter message pipe effect, is a noticeable feature in MPWS applications that use consecutive sends; it is even more so in those applications who's consecutive sends are received by a distributed selection of processors.

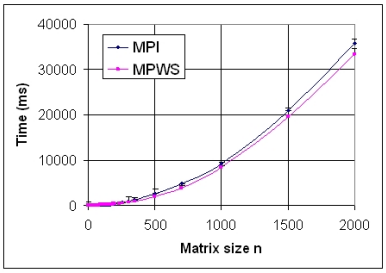

Fig. 6. Times for the Ping*Pong test MPWS and mpiJava.

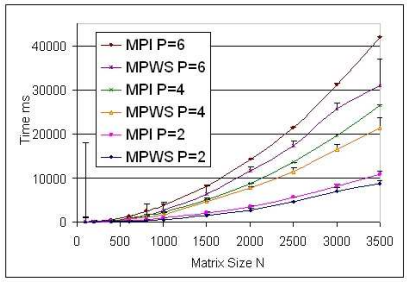

Fig. 7. Times for the Matrix Multiplication test MPWS and mpiJava.

From the above observations, we conclude that MPWS is an effective tool for coarse grained parallel applications , such as a parallel matrix multiplication, implemented in a service oriented environment.

The next steps will be to consider the design of other send styles, such as ssend (synchronous send), and evaluate MPI style collective communication functionality such as: broadcast; gather and scatter; and all reduce.

References

1. Akram, A., Meredith, D., Allan, R.: Evaluation of bpel to scientific workflows. In: CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), Washington, DC, USA, IEEE Computer Society (2006) 269–274
2. Krishnan, S., Wagstrom, P., von Laszewski, G.: Gsfl: A workflow framework for grid services (2002) In Preprint ANL/MCS-P980-0802
3. Huang, Y., Huang, Q.: Ws-based workflow description language for message passing. In: 5th IEEE International Symposium on Cluster Computing and Grid Computing, Cardiff, Wales, U. K (2005)
4. B. Carpenter, G. Fox, S. Ko, and S.Lim.: mpiJava 1.2: API Specification. `http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html` (October 1999).
5. Baker, M., Carpenter, B., Shafi, A. In: An Approach to Buffer Management in Java HPC Messaging. Volume Volume 3992/2006 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (May 2006) 953–960
6. Lee, H.K., Carpenter, B., Fox, G., Lim, S.B.: Benchmarking hpjava: Prospects for performance. In: 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers. (March 2002)
7. Gropp, W.: Tutorial on MPI: The Message-Passing Interface
8. Kut, A., Birant, D.: An approach for parallel execution of web services. In: Proceedings - IEEE International Conference on Web Services, IEEE Computer Society (June 2004) 812–813
9. Ruth, M., Lin, F., Tu, S.: Adapting single-request/multiple-response messaging to web services. In: Computer Software and Applications Conference, 29th Annual International. Volume 2. (2005) 287 – 292

10. Puppin, D., Tonellotto, N., Laforenza, D.: How to run scientific applications over web services. In: Parallel Processing. ICPP 2005 Workshops. International Conference Workshops on. (2005) 29 – 33
11. Harrington, B., Brazile, R., Swigger, K.: Ssrle: Substitution and segment-run length encoding for binary data in xml. In: Information Reuse and Integration, 2006 IEEE International Conference on. (Sept. 2006) 11–16
12. Bayardo, R.J., Gruhl, D., Josifovski, V., Myllymaki, J.: 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, New York, NY, USA, ACM Press (2004) 345–354
13. Barton, J.J., Thatte, S., Nielsen, H.F.: Soap messages with attachments. W3c note, W3C (Dec. 2000)
14. The Apache Software Foundation: MTOM Guide -Sending Binary Data with SOAP. 1.0 edn. `http://ws.apache.org/axis2/1_0/mtom-guide.html` (May 2005)
15. Ying, Y., Huang, Y., Walker, D.W.: Using soap with attachments for e-science. In: Proceedings of the UK e-Science All Hands Meeting 2004. (Aug. 2004) Poster.
16. Czajkowski, K., Ferguson, D.F., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W.: The ws-resource framework version 1.0. Technical report, Globus Alliance and IBM (2004)
17. Graham, S., Karmarkar, A., Mischkinsky, J., Robinson, I., Sedukhin, I.: Web Services Resource 1.2 (WS-Resource) Public Review Draft 01. OASIS. (10 June 2005)
18. Jayasinghe, D.: Invoking web services using apache axis2. `http://today.java.net/pub/a/today/2006/12/13/invoking-web-services-using-apache-axis2.html` (Dec 2006) Accessed Aug 2007.
19. Luszczek, P., Dongarra, J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., Takahashi, D.: Introduction to the hpc challenge benchmark suite. Technical report, icl.cs.utk.edu (march 2005 2005)
20. Intel: Intel mpi benchmarks. Technical report, Intel (June 2006)
21. Getov, V., Gray, P., Sunderam, V.: 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), New York, NY, USA, ACM Press (1999) 21
22. Foster, I., Karonis, N.: A grid-enabled mpi: Message passing in heterogeneous distributed computing systems. In: Supercomputing, 1998. SC98. IEEE/ACM Conference on, IEEE Computer Society (1998) 46 – 46