1

The Design and Evaluation of MPI-Style Web Services

Ian Cooper, Coral Walker

Abstract—In this paper we introduce MPI-style web service (MPIWS), a novel service presented as a standard web service but integrated with MPI programming technologies to allow web applications to run in parallel over a loosely-coupled distributed environment. MPIWS takes advantage of the SOAP communication protocol, and allows direct MPI-style communication among loosely-coupled services. The MPI-style communication supported by MPIWS includes both point-to-point and collective communications. In this paper, point-to-point and collective communication operations are evaluated in comparison with mpiJava. The evaluation results demonstrate that, although the overhead of SOAP messaging takes a toll on performance, MPIWS is generally comparable with mpiJava sending Object datatypes, especially for coarse-grain applications, and outperforms mpiJava in some cases.

Index Terms—Web Services, MPI, Message Passing, Collective Communication.

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.

In spite of the performance concerns of the SOAP messaging protocol, 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) [1], have started addressing the problem of intercommunicating processes. GSFL provides the functionality for one executing Grid service to communicate directly with another concurrently executing Grid service. Although implementation details using OGSA notification ports in a subscriber/producer methodology are discussed in this paper, there is no enactment environment available to support GSFL. Another example is Message Passing Flow Language (MPFL) [2], which allows web service communications to be described in XML. However, no enactment engine has been implemented so far.

BPEL4WS is commonly used for composing WS-based scientific workflows [3], 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. Fig 1 shows a workflow implementing a loop of two 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 two services.

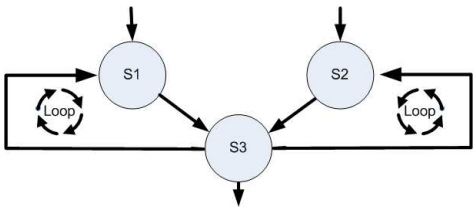

Fig. 1. A workflow showing parallel services S1 and S2 performing an iterative task by looping via a mediator service S3

As an alternative to this scenario, Fig 2 shows the loop implemented using MPI-Style message passing communication between the two services, this enables the services to be written in a way that they can process their own loop constraints and data sharing through loosely synchronous communication at each iteration.

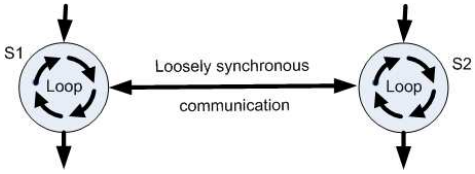

Fig. 2. A workflow showing parallel services S1 and S2 performing an iterative task by looping internally sharing data directly with each other.

This alternative, as well as eliminating the need for

• *Ian Cooper and Coral Walker are with the School of Computer Science, Cardiff University, UK.*
E-mail: {i.m.Cooper, yan.huang}@cs.cf.ac.uk

the mediator service and re-invocation at every iteration, allows the use of collective communication techniques to improve the efficiency of the data transfer; i.e. 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.

One example of this style of application is described in [4] 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 with 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 Sec 4.5.

In this paper, we extend and update the work of our previous paper [5] to investigate the potential and suitability of using a web service infrastructure to support parallel applications that require MPI-style 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 MPI-style web services (MPIWS) and present performance results comparing MPIWS against mpiJava [6], a leading high performance Java implementation [7]. Finally we examine a molecular dynamics simulation that has been adapted to use MPIWS and discuss its performance.

2 Background and Related Research

In the context of parallel computing, MPI message passing is referred to as the act of cooperatively passing data between two or more separate processes [8], each of which performs one of the subtasks of the application. 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 is not limited to one application. It can be invoked many times and work for multiple applications at the same time. The problem with allowing multiple invocation instances of a web service is the lack of a mechanism for maintaining the state of each invoked instance within a web service, so that when a communication takes place, without such a mechanism, it is difficult to determine which invoked instance should receive the message.

A Web service framework commonly uses a simple Message Exchange Pattern (MEP) which involves a request-only or a request-and-response message, providing no support for direct communication between concurrently running services. A service-composite application may involve invocations of many web services. The return data of a service invocation could be the direct input to another service. Without a direct communication mechanism, the data has to be returned to the client first before sending to the next service, which significantly affects the overall performance of executing a service-composite application.

Currently, there is no standard for passing data from one service to another, concurrently running, service. Kut and Birant [9] have suggested that web services could become a tool for parallel processing and present a model, which uses threads to call web services in parallel to allow web services to perform parallel processing tasks. This model can be extended to allow web services to exchange data directly, this removes the need for the client to intervene every time a process transfers data [1]. The scenario of direct communication between web services is shown in Fig. 3.

Research into the use of web services in parallel computation is presented by Puppin et al. [10], who suggest an approach for mapping MPI code into a WS-based communication scenario to allow MPI applications to run in a web service architecture. Their evaluation results shows that the performance of their WS-based MPI applications improves with the number of processors, and by using plenty of processing resources, WS-based MPI applications are able to run nearly as fast as MPI, with about 50% overhead. However, it appears that a fast SOAP mechanism has not been used in the implementation, suggesting the potential of further improvement in performance if one of the more efficient SOAP messaging approaches were used.

Queiroz et al. [11] presented a tool to distribute a parallel application over distributed web services, using sockets to support direct message passing between services.

mpiJava [13] is a non WS-based version of MPI that uses Java Native Interface (JNI) to provide a Java interface to MPICH which allows MPI applications to work in a more loosely-coupled distributed environment and communicate data in the form of Objects. Although it is not WS-based, it runs in a distributed environment, and is thus broadly similar to the MPI-style web services we have implemented. There has also been much research into the performance of mpiJava [12], so by evaluating against mpiJava, we can get an idea of how the web services architecture will perform against other approaches. These arguments make mpiJava a good choice to compare against MPIWS.

The web service standard WS-Notification [13] allows for a subscriber - notification protocol to be followed but this adds an extra layer of functionality on top of WS-Resources whilst still needing the Message passing protocol implementation.

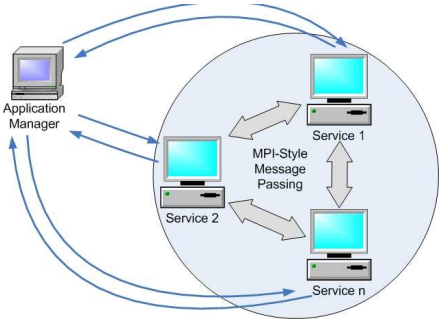

Fig. 3. Extending the use of parallel executing services to allow MPI-Style direct message passing between concurrently executing service invocations. The thin arrows indicate the request-and-response service invocations and the thick arrows indicate direct communication between web services

3 Design of 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 we need to adhere to web service and SOAP messaging standards whilst providing an efficient form of communication between services. MPIWS are designed to allow for direct communication between concurrently executing web services.

3.1 MPI-Style Web Services

An MPIWS is a service with the ability to perform direct point-to-point and collective communication with other concurrently executing MPIWS services.

Executing a particular WS-based application requiring MPI-style message passing involves a group of MPIWS services working together within a particular communication domain, at this time in the implementation these services are fixed when the domain is initialised. It is possible for a MPIWS service to have multiple invocation instances at the same time, with each instance working for a particular application and belonging to a particular communication domain. Since a service may have multiple service invocation instances, each working for a different communication domain, a domain ID is required in order to identify different 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.

Currently, MPIWS is provided as an API to be used in the development of MPIWS services, this means that it is deployed as part of the application's deployment file. A MPIWS service supports a three-layer interface: a SOAP-based application layer, an internal MPI-operation layer, and a SOAP-based direct communication layer.

The interface at the application layer is a web service interface to allow MPIWS services to be invoked in the same way as any other web service. It includes only one method, *init()*, which initiates a service invocation instance for executing the subtask coded within the *init()* method.

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 internally within the *init()* method in a similar style to a MPI application.

The methods provided by the internal MPI-operation layer do not perform direct communications themselves – this is done through the interface provided by the direct communication layer. The direct communication interface provides methods to allow the direct communication among services. Similar to the application layer interface, the methods at the direct communication layer conform to web service standards so that SOAP messaging is used in the direct communication among services. There are two direct communication methods currently supported:

- *store()* receives message data and stores it locally.
- *bstore()*, similar to *store()*, but to support binomial broadcast communications.

3.2 Communication Domains

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, or service invocation instance, is an invocation instance of the *init()* method of an MPIWS service. It is always associated with a particular communication domain and can be identified by its rank value stored in *myRank*. The invocation of the *init()* method initializes a service instance. The input data for the *init()* 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, it allows 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 as 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 [14], to provide 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 [15]. In our case, a resource is used to store local data and data received from other service instances. It is created when a service instance is initiated 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.

Fig. 4 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).

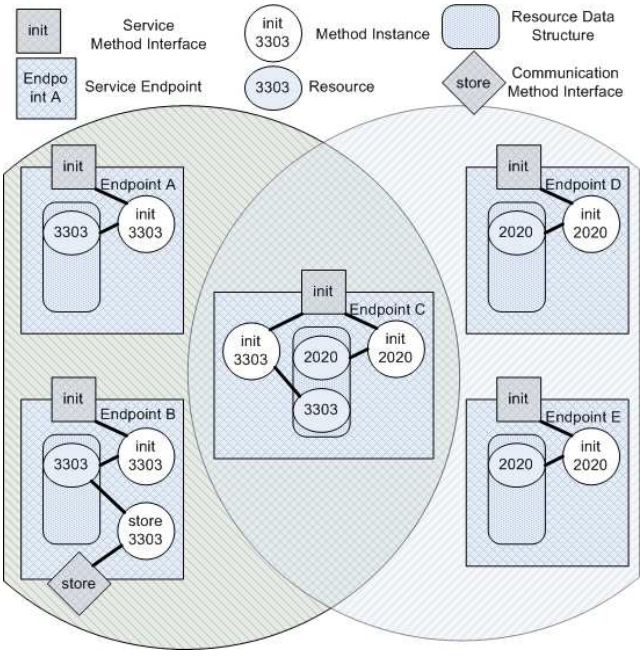

Fig. 4. Example of services working for multiple communication domains.

3.3 Communication

In an MPIWS service, invoking the *init()* method initializes a service instance that executes the application subtask; this subtask may require MPI-style 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 doesn't stop the same or other methods from being invoked. This gives the idea that, if a running method instance needs to receive data from another running method instance, 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 running method instance. So the solution to this is to devise methods that work separately from the *init()* method, and provide direct communication support for the *init()* method by receiving and storing data locally. In order to provide support for point-to-point communication between service method 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 *init()* method instance associated with the same domain ID.

Within a resource, there is a complex message buffer structure that consists of a sub-layer of buffers. Each buffer in this sub-layer is associated with a rank value of a service instance from which the current service instance is expecting to receive a message. The arriving messages are stored, in the order in which they were sent, into the particular buffer allocated to the particular service instance from which the message is sent.

Fig. 5 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.

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 method, in the rank 3 instance at endpoint B.

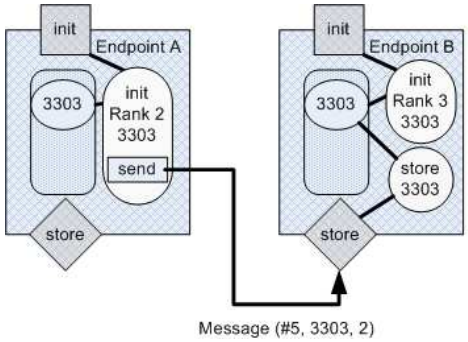

Fig. 5. MPI-style web services point-to-point send architecture.

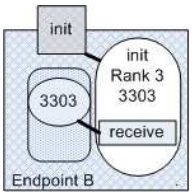

Fig. 6. MPI-style web services point-to-point receive architecture.

In the web service implementation there are several factors that may affect the sequence in which messages are received: the messages may arrive in an order different from the order in which they are sent. In our 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 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 sequence number of the message in the particular sending service. Each time when a message is sent, the sequence number is incremented and the new value is attached and sent with the next 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 lower order message has completed storage in order to retrieve the correct message.

3.3.1 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 Sec. 3.1, and 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.

There is performance issues relating to sending the data within a SOAP message. SOAP message uses the XML language. If strict XML formatting is used, i.e., listing each entity of the data within a tagged element, the space overhead for the message is potentially very large. A more efficient method of encoding data is to serialize it into a binary representation. In Java there is an built-in function to transform objects to their binary encoded representation, which is used in mpiJava to encode objects before calling the native MPI layer. However, there is a problem when putting a binary file into a SOAP message: a binary file cannot be translated directly to a string format, as there are not enough characters available. Four solutions are available to this problem: binary-to-character encoding [16], packaging [17] [18], binary XML encoding [19], and linking [16].

Packaging, an approach that is used by both SOAP with Attachments (SwA) [17] and SOAP Message Transmission Optimization Mechanism (MTOM) [18], allows data to be transmitted externally to the SOAP envelope. A comparison of transmission speeds using SwA and strict XML formatting is given in [20], which shows that for a 256×256 matrix of *doubles*, the XML encoded SOAP messages take over 10 times as long to transmit as the same data serialized and sent using SwA. MTOM also uses the SOAP with attachment approach and is supported by Apache Axis 2, and can be used in MPIWS services. It transports data as attachments to the SOAP message with no coding overheads of either the binary to character or the binary XML encoding. It also stays within the SOAP communication protocols, unlike linking. In addition, it decreases the message data size by allowing attachments containing serialised objects, while keeping the data accessible in the object model.

MTOM requires the transmitted data to be bundled into a neat, binary coded package so that it can be attached to the SOAP message. As the transmitted data is of type *Object*, Java serialization is used to convert the object to the required binary format. Keeping data in its own object format at both sending and receiving ends saves overhead from XML encoding and decoding and significantly improves the performance of SOAP

messaging. This style of data transmission has been used in many other related works, such as Queiroz [11] where the serialisation of data sent over sockets is used, and mpiJava [6] where the Java wrapper transfers the Objects as byte arrays within the MPICH implementation.

3.3.2 Fire-and-Forget Invocation Approach

The use of a WS-Resource to provide a message buffering service for message passing encourages the adoption of the asynchronous *fire-and-forget* service client model [21] which is supported in Apache Axis 2, 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 [21], which expects a response message before the method returns, and the *sendRobust* client model [21], which sends data and returns when the processing at the server is complete with either the results or information to report any problems [21].

3.4 Collective Communications Design

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. The inclusion of supporting collective communications in MPIWS is essential to demonstrate the potential efficiency of a WS-based approach for scientific computing. To this end, we have implemented a number of collective communication operations including: *Broadcast*, *Gather*, and *Barrier*.

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.

3.4.1 SendRecieve

The *sendRecieve* 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* form that second service node is taking place. This operation utilises the duplexity of the communications network. MPIWS implements a sendReceive operation by using threads to perform each of the basic point to point operations.

3.4.2 Broadcast

The easiest example of true collective communication to envisage is the broadcast operation. The simplest way to perform a broadcast operation is for the broadcasting rank, commonly called the root, to repeatedly send the message to all other ranks in the communication domain. *SerialBroadcast* is a straight forward approach in which the root creates the message and sends it serially to each rank in turn.

The implementation of the *serialBroadcast* operation requires multiple *sends*. The XML message is created via an *Element* object, which uses a data stream to attach the serialized data to 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, this adds extra latency to the *broadcast*.

The serial version of the *broadcast* has poor load balance because the communication relies on the root repeatedly sending the message to other ranks. It also suffers inefficiency because it performs one *send* operation at a time and so does not fully utilize the bandwidth capabilities of the network. A better efficiency, with better network utilization and load balance, can be gained if the *send* operations are distributed among multiple ranks to allow multiple *send* operations to be performed concurrently.

The binomial distribution of the data message[22] is a more efficient method of performing the broadcast and has been widely used in MPI implementations. 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 approach, both the re-transmission of the data, and the calculation of which rank to re-transmit to, must happen in the methods at the direct-communication layer. To this end MPI-style web services provide a *bStore* method, distinct from the *store* method, with the additional re-transmission functionality required for the 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 re-transmission of the data within the direct-communication layer methods which need to be taken care of.

Firstly, the *fromRank* of the message must remain set to the rank value of the root that initiated the broadcast. So it is necessary to copy the *fromRank* value of the received message to the re-transmitted message during the binomial broadcast.

Secondly, 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 broadcasts' root rank, to the ultimate receiving rank by other intermediary

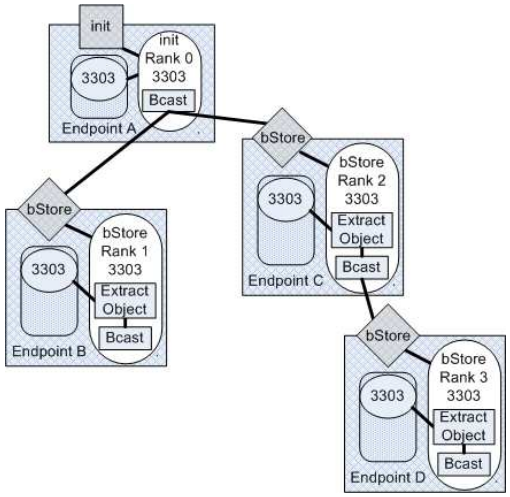

Fig. 7. Architecture for the *broadcast* operation.

ranks within the communication domain. The messages' sequence number is different for each receiving rank, but the sequence number for that receiving rank is only accessible at the broadcast root rank. 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 messages' overhead data, but allows each ultimate receiving *bStore* method instance to extract the correct sequence number for its rank.

Fig. 7 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 rank 1 and 3, the object is extracted and the *Bcast()* method calculates that there are no further transmissions needed and the broadcast completes.

3.4.3 Gather

The *Gather* collective operation *retrieves* data from all non-root service nodes and arranges it in an array at the root service. 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 to the index value of the cell [23]. 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 [22]. Both versions are implemented by using the point-to-point primitive operations: *send* and *receive*.

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 nodes. 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, 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 searching through the received array, and copying each non-null object into its own array. It then sends the merged array to the node above in the binomial tree.

3.4.4 Barrier

The *barrier* operation provides a synchronization 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 [24]. The method we choose 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 consecutive sends compared to other methods, such as the Double Ring [24].

A *barrier* operation involves very small or null data transmission. Compared with the small data size that are 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.

3.4.5 Reduce

The *reduce* operation is similar to the *gather* but instead of arranging the data from all nodes in a sequential array, it merges the data from the service nodes with an operation such as SUM or PRODUCT [23]. The design and the evaluation is similar to the *gather*, although there must be specific datatypes transmitted as the Object, such as an array of *doubles*, so as to allow the merge operator to function correctly.

The *allReduce* operation is an extension of the *reduce* operation, in that, 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 [23].

There are a number of different methods to achieve the *allReduce* operation, one of which is a *scatter* followed by a *reduce* [22]. 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 is difficult to be split into chunks and distributed over multiple nodes. Thus MPIWS adopted two different approaches: the *reduce* operation followed by a *broadcast* operation, and the recursive doubling approach [25]. 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, this process is repeated as shown in Fig 8

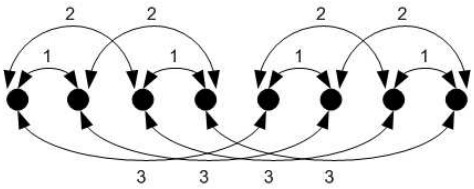

Fig. 8. The recursive doubling communication for the *allReduce* Algorithm with three steps (1, 2 and 3).

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; this is discussed in [25].

4 Evaluation

Through the evaluation of MPIWS, we would like to show that the WS-based architecture that enables MPI-like point-to-point and collective communications among web services can perform well compared with other message-passing implementations, such as mpiJava, over a loosely-coupled, distributed network. MpiJava also has the ability to transfer data as Objects, allowing a more object oriented approach to MPI-Style message passing, this allows like for like comparison of the two systems. The evaluation has been split into two parts: evaluation of point-to-point operations, and evaluation of collective communication operations.

4.1 Point-to-Point Communication

The evaluation tests focus mainly on the speed aspect of the communication implementations and MPIWS services are tested against mpiJava. Many benchmark suites have been devised and put forward as definitive parallel computing benchmarks [26] [27], and many of these are designed to test the underlying hardware or the collective communications features of the message-passing tools. We have chosen tests that specifically target the performance of the message passing tools.

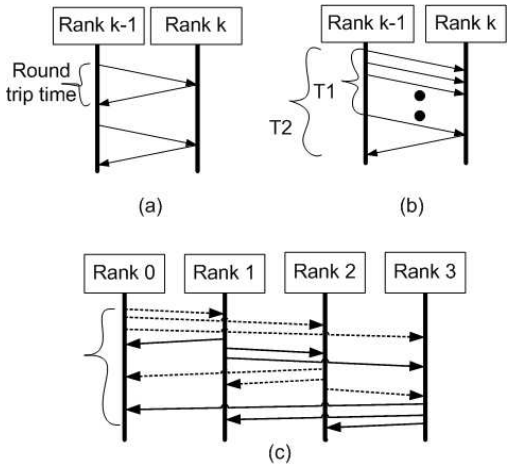

Fig. 9. Scenarios of PingPong, Ping*Pong and Matrix Multiplication tests. An arrow represents a portion of the matrix being sent from one processor to another.

The PingPong test is one of the most popular tests that are used to provide a simple bandwidth and latency test for point-to-point communications. Getov et al. [28] used a number of variations of the PingPong test to compare the performance of MPI and java-MPI. Foster and Karonis [29] also used the test to evaluate MPICH-G, a grid-enabled version of MPI. Here, we have chosen two variations of the PingPong tests: the standard PingPong test, and the Ping*Pong test.

The standard PingPong test requires an even number of processors within a communication domain, with each of the processors paired with another. Within each pair of processors a message is sent from one processor to the other, and is then sent back again. The scenario of the PingPong test is illustrated in 9(a). In this test, the round-trip time of the message traveling from one processor to another is measured.

The Ping*Pong test [28] is a variation of the PingPong test. Similar to the PingPong test, it involves an even number of processors each of which is paired with another. In each pair group, one processor sends multiple messages to the other processor in the same pair group, and the receiving processor returns each message it receives. Fig. 9(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 [28]. 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).

A further test is performed based on a real application, a one dimensionally blocked parallel matrix multiplication, which is a simple parallelized version of the matrix multiplication problem. The communications for the

matrix multiplication application are shown in Fig. 9(c). It is important to note that although the sequence of the *send* operations are 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' utilization at the time of testing, could dilute the performance of the communications. We have therefore omitted the calculation part of the application and presented only the communication results of the application.

4.2 Collective Communication

For the evaluation of collective communication operations, we have tested both serial and binomial versions of the *broadcast*, *gather* and *allReduce* operations against mpiJava.

In the *broadcast* and *allReduce* tests, a *barrier* operation is performed before the start of the operation. The time calculation starts after the *barrier* operation is completed. The *broadcast* operation ends when all the service nodes have received the broadcasted message and the broadcasting service was notified. The notification is performed by a *report-to-root* operation which is effectively a minimal data gather. In the *allReduce* tests this report to root must be done with the *reduce - broadcast* method and the recursive doubling method to ensure the consistency.

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

4.3 Evaluation Environment

In the evaluation tests, all the MPIWS services are deployed using Apache AXIS 2.1.2 and are hosted in a Tomcat 5.5.20 application server. The mpiJava API we 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 to reduce the impact of the network on the results. 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 should be especially noted for the threaded *sendRecieve* tests.

4.4 Evaluation Results and Discussion

All tests are done using an Object comprising of an $N \times N$ matrix of *doubles* as the message. In the graphs of the test results, MPIWS indicates the MPIWS implementation, mpiJava indicates the mpiJava implementation, S MPIWS indicates the serial version of the MPIWS implementation, and B MPIWS indicates the binomial version of the MPIWS implementation.

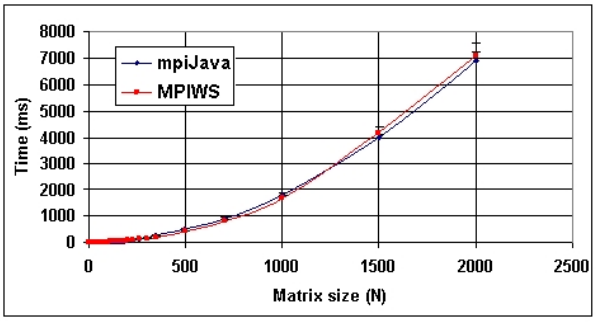

Fig. 10. PingPong test results (N = 0 - 2000).

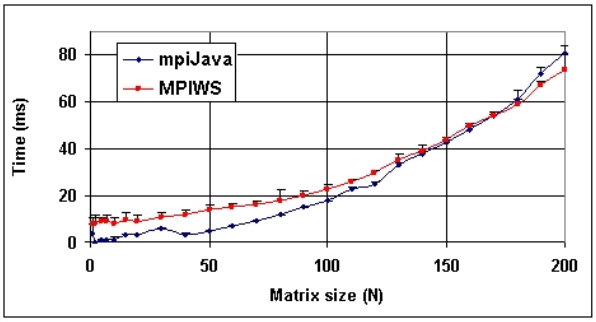

Fig. 11. PingPong test results (N = 0 - 200).

The results of the PingPong Test are displayed in Fig. 10 with N in the range of 0 to 2000. 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 N in the range of 0 to 200 are displayed in Fig. 11 with a larger scale. The results clearly show the expected communications overhead of the SOAP messaging, which degrades the performance of MPIWS when the transmitted data is small. However, when the transmitted data is large enough(in this example, over the 200 KB for when $N = 160$), the SOAP communication overhead becomes relatively small enough to be absorbed by the total communication time to make MPIWS services run at a speed that is similar to mpiJava.

The results for the Ping*Pong test are shown in Figs. 12 and 13. Fig. 12 shows the results when N is in the range of 0 to 2000 and Fig. 13 shows the results on a larger scale when N is in the range of 0 to 200. It can be seen clearly from the figures that the MPIWS implementation outperforms mpiJava when the transmitting data is larger than 130 KB (when $N = 130$). The positive results of Ping*Pong test can be explained by the inter message pipeline effect caused by the *fire-and-forget* approach and message buffering used in MPIWS.

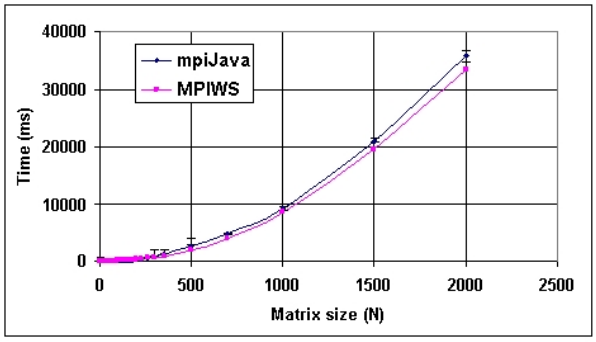

Fig. 12. Ping*Pong test results (N = 0 - 2000).

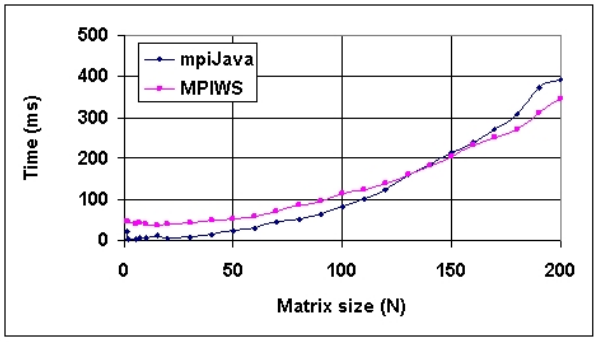

Fig. 13. Ping*Pong test results (N = 0 - 200).

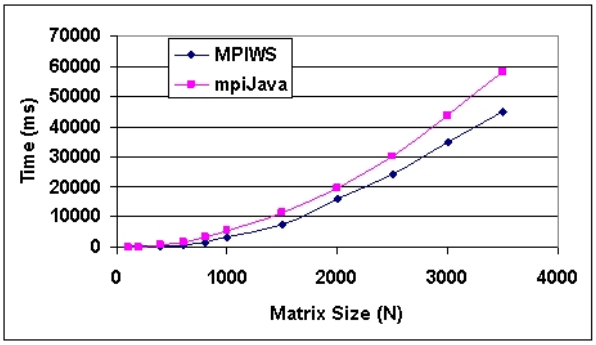

Fig. 14. Results of Matrix multiplication test using point-to-point Communication with 8 Processors (N = 0 - 2500).

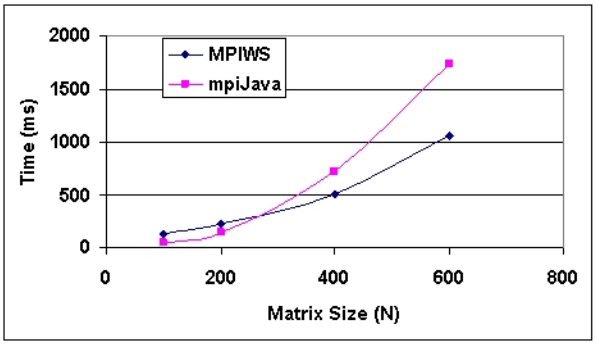

Fig. 15. Results Matrix multiplication test using point-to-point communication with 8 processors (N = 0 - 600).

The results of the matrix multiplication test running over 8 processors and using point-to-point communication operations are shown in Figs. 14 and 15. Fig. 14 shows the results when N is in the larger range of 0 to 2500 while Fig. 15 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. We have also done the tests over different numbers of processors and all the results came out consistently. The results shows clear agreement with the Ping*Pong test. The matrix multiplication requires consecutive *send*s to distribute the matrix over processors. The combination of *fire-and-forget* sends with message buffering at the receiving processor have a good inter pipeline effect on the MPIWS which is demonstrated in the Ping*Pong Test, and explains the test results showed in Figs. 14 and 15.

The results of the broadcast tests are shown in Figures. 16 and 17. Fig. 16 shows the results when N is in the larger range of 0 to 700, while Fig. 17 shows the results when N is in the smaller range of 0 to 300. Three implementations of broadcast have been tested: serial version of MPIWS *broadcast*, binomial version of MPIWS *broadcast* , and mpiJava *broadcast*. The MPIWS serial version and mpiJava *broadcasts* perform comparably with mpiJava *broadcast* doing slightly better, but the MPIWS binomial version of *broadcast* performs much better than the either of them. The results are expected because using a binomial tree is a more efficient approach in implementing *broadcast* than using a serial approach [22], and *broadcast* in mpiJava is a serial version of broadcast due to there being no mapping to the native MPICH broadcast for broadcasts of the type *Object*.

The results of the gather tests are displayed in Fig. 18. Three gather tests are performed: the serial version of MPIWS *gather*, the binomial version of MPIWS *gather*, and the javaMPI *gather*. In contrast to the *broadcast* operation, where using a binomial tree significantly improves performance, using a binomial tree works degrades

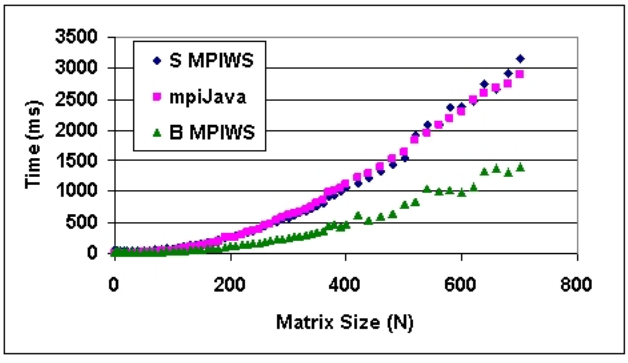

Fig. 16. Broadcast test results (N = 0 - 700).

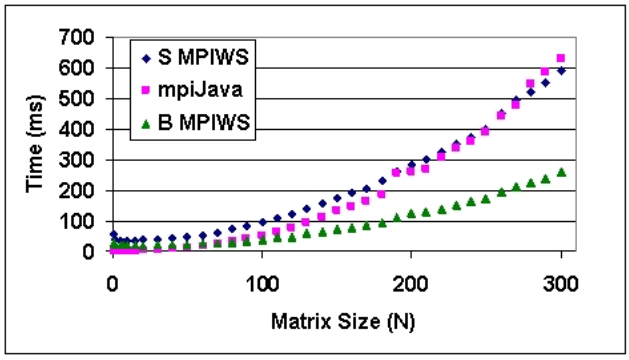

Fig. 17. Broadcast test results (N = 0 - 300).

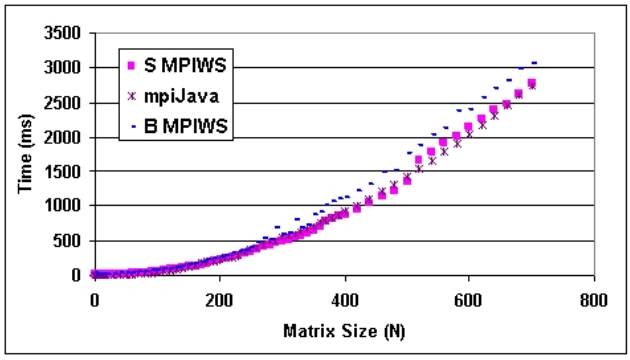

Fig. 18. Gather test results (N = 0 - 700).

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 ($N = 150$) and slightly worse than the MPIWS serial *gather* when the message size is large than this value.

Since there is no dependence on message size, the results of the barrier tests displayed in Fig. 20 show the timings of the barrier communication against the number of processors. Three different barrier implementations are tested: the serial version of MPIWS *barrier*

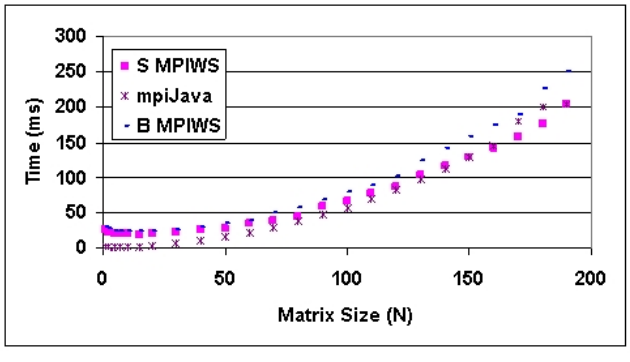

Fig. 19. Gather test results (N = 0 - 200).

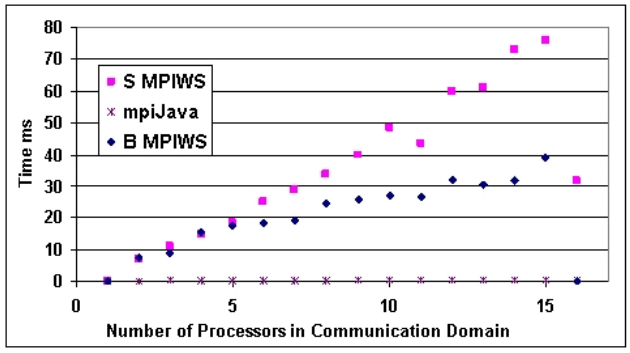

Fig. 20. Barrier test results.

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 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 are 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.

The *reduce* and *allReduce* evaluation must be considered very carefully, they are included in the evaluation specifically due to their use in the molecular dynamics application presented in sec 4.5. 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. This allows the reduction operations to function properly, but also means that the MPIWS is no longer being evaluated against a message passing tool which is transferring Objects. The graphs (fig. 21 and 22) show the mpiJava *reduce* and *allReduce* as well as the MPIWS *serial reduce* and *binomial reduce* and two implementations of the MPIWS *allReduce*; the *reduce* - *broadcast*, and the recursive doubling (RD) methods.

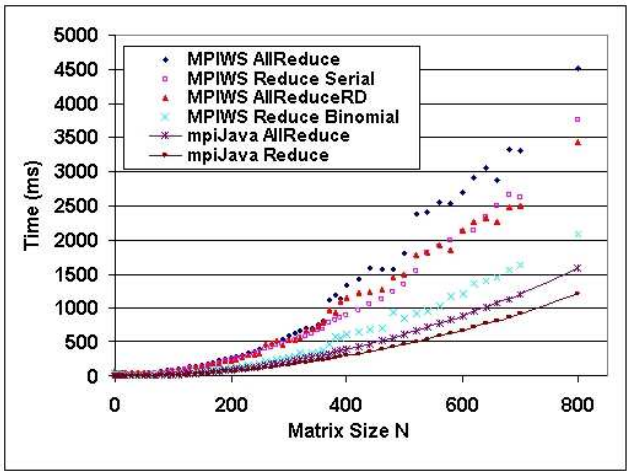

Fig. 21. AllReduce test results (message size N = 0 - 800).

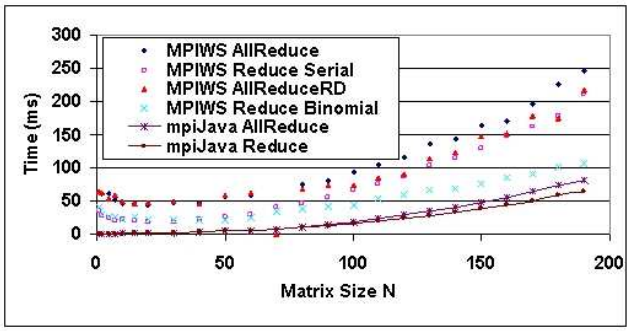

Fig. 22. AllReduce test results (message size N = 0 - 200).

With the *reduce* evaluation, we can see that the collective communications approach is consistently beneficial. When it is compared to the transmission of datatypes within the javaMPI we can see the impact of the extra serialisation step. Again for the *allReduce* evaluation it can be seen the 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. These comparisons show the ability of the web services architecture to utilise the collective communications operations in order to increase the efficiency of the data transfer.

4.5 Application Moldyn

MolDyn[30] is a piece of molecular dynamics simulation code, provided by the Java Grande Forum with the MPJ Version 1.0 source code. It is used as an evaluation benchmark.

The MolDyn simulation problem consists of an array of particles. Each particle has a position, a velocity and a force; each of these is defined in terms of its x, y, and z components. The whole particle array is present and initialised at all the participating ranks, and then there are 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 (xcoord = xcoord + xvelocity + xforce and xvelocity = xvelocity + xforce) that scales at $O(n)$ where n is the size of the particle array, so it is faster to carry these 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 this scales at $O(n^2)$. 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 in to the particle objects and then the next iteration can be performed.

The MolDyn code fits nicely in to 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.

As 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 fig 1). Then 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.

MolDyn is also a typical high performance computing application. MPI is commonly used for molecular dynamics simulation and there are many examples of production grade code available [31], [32]. 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 both 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 2K to 32K particles.

4.5.1 Evaluation of MolDyn running on MPIWS

If we look at the communication results for the *allReduce* operation we can create an estimation of the extra time that the MolDyn application should take running on the MPIWS tool compared to on mpiJava. Fig 23 shows the timings for a range of particle array sizes run on both MPIWS and mpiJava as well as the predicted and actual

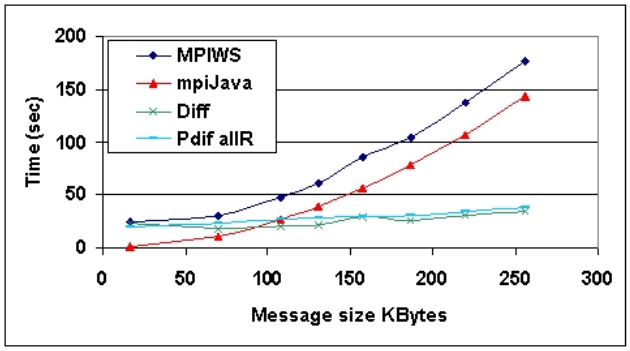

Fig. 23. The times taken for the MolDyn Application vs the individual forces message size for MPIWS and mpiJava.

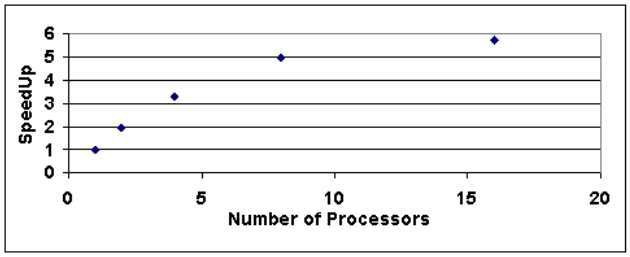

Fig. 24. The speedup for the MolDyn application over 0-8 MPIWS services running the application with 27,436 particles (individual force message size is approx. 157Kbytes).

difference in the two results. The second graph, fig 24 shows the speed up of the 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 speedup graph, there is a definite timing improvement when the application is distributed over more than one service.

These results do prove an extremely important point about the applicability of MPI-Style collective communications in the workflow environment. If MolDyn was run for 27,436 particles (message size approx 220 Kbytes) on 8 services, the time for the barriers would be $33ms$, the time for the *serial allReduce* of the double values would be $60ms$ then the time for a *serial reduce* and a *serial broadcast* is $155 + 198ms$ this gives a communication time for 50 iterations of 67 seconds, compared to the 41 seconds for the collective operations.

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. It also shows that MPIWS can be used to efficiently run scientific computing applications that are written for a traditional MPI implementation by simply replacing the MPI communication calls with the MPIWS communication calls and deploying the application as an MPIWS web service.

5 Conclusions

Direct communication support and MPI-Style message passing among web services provides the ability for MPI-style applications to fully utilise the modularity of the web services environment. It could become the 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, we have 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 similar Objects. It is also 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.

The benefits to the efficiencies within workflow communication have also been assessed and it has 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.

From the above observations, we conclude that using MPIWS for WS-workflow applications requiring MPI message passing is potentially a practical and efficient way of distributing coarse-grained parallel applications.

Further work on this research would be to increase the functionality of MPIWS to more fully implement the MPI standard, including the use of message tags, and the ability to receive from 'Any Source' as well as implementation of more of the collective communication operations. A longer term goal is to integrate MPIWS with MPFL to produce an execution environment for MPI-Style workflows.

References

[1] S. Krishnan, P. Wagstrom, and G. V. Laszewski, "Gsfl: A workflow framework for grid services," Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, Tech. Rep., 2002.

[2] Y. Huang and Q. 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.

[3] A. Akram, D. Meredith, and R. 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)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 269–274.
[4] M. Mu and J. R. Rice, "Modeling with collaborating pde solvers–theory and practice," *Computing Systems in Engineering*, vol. 6, no. 2, pp. 87 – 95, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/B75C5-49V61XR-1K/2/0540534724f2a88fb59d15366a3a03b5
[5] I. Cooper and Y. Huang, "The design and evaluation of mpi-style web services." in *ICCS (1)*, ser. Lecture Notes in Computer Science, M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds., vol. 5101. Springer, 2008, pp. 184–193. [Online]. Available: http://www.springerlink.com/content/n74287q451wgl504
[6] B. Carpenter, "Java for high performance computing: Mpi-based approaches for java," Pervasive Technology Labs, Indiana University. internet presentation, accessed Aug 2007. [Online]. Available: http://www.hpjava.org/courses/arl/lectures/mpi.ppt
[7] M. Baker, B. Carpenter, and A. Shafi, *An Approach to Buffer Management in Java HPC Messaging*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, May 2006, vol. Volume 3992/2006, pp. 953–960.
[8] W. Gropp, *Tutorial on MPI: The Message-Passing Interface*. [Online]. Available: http://www.new-npac.org/projects/cdroms/cewes-1998-05/reports/gropp-mpi-tutorial.pdf
[9] A. Kut and D. Birant, "An approach for parallel execution of web services," in *Proceedings - IEEE International Conference on Web Services*. IEEE Computer Society, June 2004, pp. 812–813.
[10] 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*, 2005, pp. 29 – 33.
[11] C. Queiroz, M. A. S. Netto, and R. 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.
[12] H.-K. Lee, B. Carpenter, G. Fox, and S. B. Lim, "Benchmarking hpjava: Prospects for performance," in *6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, March 2002.
[13] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedukhin, D. Snelling, S. Tuecke, W. Vambenepe, and B. Weihl, "Web services base notification," 03/05/2004 2004.
[14] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe, "The ws-resource framework version 1.0," Globus Alliance and IBM, Tech. Rep., 2004.
[15] S. Graham, A. Karmarkar, J. Mischkinsky, I. Robinson, and I. Sedukhin, *Web Services Resource 1.2 (WS-Resource) Public Review Draft 01*, OASIS, 10 June 2005. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-cd-01.pdf
[16] 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*, Sept. 2006, pp. 11–16.
[17] J. J. Barton, S. Thatte, and H. F. Nielsen, "Soap messages with attachments," W3C, W3C Note, Dec. 2000. [Online]. Available: http://www.w3.org/TR/SOAP-attachments
[18] M. Gudgin, N. Mendelsohn, M. Nottingham, and H. Ruellan, "Soap message transmission optimization mechanism," W3C, Tech. Rep., January 2005. [Online]. Available: http://www.w3.org/TR/soap12-mtom/
[19] 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*. New York, NY, USA: ACM Press, 2004, pp. 345–354.
[20] Y. Ying, Y. Huang, and D. W. Walker, "Using soap with attachments for e-science," in *Proceedings of the UK e-Science All Hands Meeting 2004*, Aug. 2004, poster.
[21] D. Jayasinghe, "Invoking web services using apache axis2," Internet, Dec 2006, accessed Aug 2007. [Online]. Available: http://today.java.net/pub/a/today/2006/12/13/invoking-web-services-using-apache-axis2.html
[22] M. Barnett, L. Shuler, S. Gupta, D. G. Payne, R. van de Geijn, and J. Watts, "Building a high-performance collective communication library," in *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 107–116.
[23] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Comput.*, vol. 20, no. 4, pp. 657–673, 1994.
[24] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. Volume 10, no. Number 2, pp. 127–143, June 2007.
[25] R. Rabenseifner, "Optimization of collective reduction operations," in *Computational Science - ICCS 2004*, M. et al., Ed., vol. 3036/2004. Springer Berlin / Heidelberg, 2004, pp. 1–9. [Online]. Available: http://www.springerlink.com/content/hha38fla0p0nhp1x/
[26] 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," icl.cs.utk.edu, Tech. Rep., march 2005 2005.
[27] Intel, "Intel mpi benchmarks," Intel, Tech. Rep., June 2006. [Online]. Available: ftp://ftp.uybhm.itu.edu.tr/belgeler/sistem_kullanimi/IMB_ug-3.0.pdf
[28] V. Getov, P. Gray, and V. 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)*. New York, NY, USA: ACM Press, 1999, p. 21.
[29] I. Foster and N. Karonis, "A grid-enabled mpi: Message passing in heterogeneous distributed computing systems," in *Supercomputing, 1998. SC98. IEEE/ACM Conference on*. IEEE Computer Society, 1998, pp. 46 – 46.
[30] L. Smith, "Moldyn," Edinburgh Parallel Computing Centre, Tech. Rep., 2001.
[31] "Dl_poly." [Online]. Available: http://www.cse.scitech.ac.uk/ccg/software/DL_POLY/index.shtml
[32] "Moldy." [Online]. Available: http://www.ccp5.ac.uk/moldy/moldy.html

Ian Cooper is a PhD student and associate lecturer in the School of Computer Science at Cardiff University. He received the B.Eng in Electronics Engineering from Cardiff University in 1996 and then went on to work in industry before returning to Cardiff to attain the M.Sc in Computer Science in 2005. He is now completing his PhD at Cardiff, Ian's research interests include high performance and distributed computing.

Coral Y Walker is a lecturer in the School of Computer Science at Cardiff University. Coral was awarded the B.Eng degree from Chengdu University of Science and Technology in 1991, the M.Sc. in Computing from Nanjing University in 1998, and was awarded the PhD in Computer Science from Cardiff University in 2003. Coral's main research interests are web services and WS-based Grid architecture, workflow languages and their execution environments.