PMPI: A multi-platform, multi-programming language MPI using .NET

Mohammad M. El Saifi Edson Toshimi Midorikawa
Department of Computer Engineering and Digital Systems
Polytechnic School – University of São Paulo
Sao Paulo - SP - Brazil

{mohamad.saifi, edson.midorikawa}@poli.usp.br

ABSTRACT

Implementation of the MPI standard on heterogeneous platforms is desirable because it permits using resources discarded by existing MPI implementations of homogenous systems. This paper describes PMPI, as partial implementation of the MPI standard on a heterogeneous platform. Unlike other MPI implementations, PMPI permits MPI processes written in different programming languages to run on multiplatform system. PMPI is built on top of .NET framework. PMPI can span multiple administrative domains distributed geographically. To programmers, the grid looks like a local MPI computation. The model of computation is indistinguishable from that of standard MPI computation. This paper studies the implementation of PMPI with Microsoft .NET framework and MONO to provide a common layer for a multiprogramming language multiplatform MPI application. We show the obtained results using PMPI, and compare them to MPICH2. The obtained results will show that the use of .NET framework for PMPI is feasible and can be optimized for performance.

Keywords

MPI, Parallel Computing, HPC, .NET Framework, MONO

1. INTRODUCTION

For many years, parallel computation was always an attractive alternative for obtaining high-performance computing [Dongarra et al. 2003] [Foster 1995]. With the use of multiple computational nodes interconnected by a high-speed network, clusters of computers are the most common platform of parallel machines. The recent introduction of multi-core microprocessors will result in parallel computers becoming available on desktops.
MPI is perhaps the best known standard used in parallel computation allowing nodes spread across the network to collaborate to achieve a common computational goal [Andrews 2000] [MPI Forum 1994].

The limitation of MPI is two fold. On the one side, most existing MPI implementations, such as MPICH2, execute only on homogeneous platforms [MPICH2 2006]. Accordingly, idle cycles that are spread across a variety of machine architectures and operating systems across networked PCs, are discarded because of the lack of an MPI that executes on a heterogeneous platform. These idle cycles are increasingly being recognized as a huge and largely untapped source of computing power

On the other side, almost existing MPI implementations use C, C++ or FORTRAN programming language. Accordingly, researchers and programmers who collaborate on the solution of the same problem need to stick to one of the languages that supports the MPI library they intend to use.

The implementation of MPI that can tap into those idle resources on heterogeneous platforms is desirable because it allows researchers and programmers, who need high performance computing and have available heterogeneous platforms around their campus, to use all available resources [Kelly, Roe and Sumitomo 2002][Kelly and Roe 2002][Kelly and Mason 2003]. Having the ability to use MPI on heterogeneous systems maximizes computational power resources.

In addition to using MPI on a heterogeneous platform, programmers want to use a variety of programming languages in their computational program. In the same MPI computation, programmers want nodes to run applications written in different programming languages simultaneously using MPI standards. This becomes a merit when we have multiple programmers participating in the solution of a unique problem, where each programmer is writing a program that runs on a separate node such as same data multiple program solutions. This permits programmers to explore their abilities and skills in their preferred programming language, and to use the programming language that best suit the solution of the problem.

This paper studies the feasibility of implementing MPI standard on a heterogeneous platform by implementing the component PMPI. PMPI aims to provide programmers and researchers with a framework that takes care of a transparent communication infrastructure between the heterogeneous nodes in a MPI computation in a robust and secure manner. The programmer is left to concentrate only on the application specific computational aspects. We take advantage of the .NET framework to provide application programmers with a choice of the programming language, all of which can use the same PMPI framework classes.

There are different choices that can be made to implement the PMPI component. We choose the .NET framework [Ritchter and Balena 2002] for this purpose as the first tentative and used .NET Remoting [McLean 2003] [Rammer 2002] as the communication infrastructure for PMPI. In this implementation, PMPI acts as a remote-object based framework for creating MPI parallel applications. The framework is built using the extensibility features of the .NET Remoting framework.

Unlike the Java virtual machine, the .NET runtime is designed to be language independent. Accordingly, developers can create their applications using any language that targets the CLR such as: C#, Visual Basic, Visual C++ or one of many other .NET languages such as Eiffel, Perl, Cobol, Component Pascal, Smalltalk, or Fortran [Ritchter and Balena 2002]. Today there are about twenty six different programming languages that target the .NET framework [Ritchter and Balena 2002]. PMPI enables programmers to program in a normal MPI fashion, without being concerned what platform or programming language other participating nodes will run.

The main contribution of this paper is to study the feasibility of implementing MPI on a virtual machine and show performance results compared to other existing MPI implementation. This offers programmers who have heterogeneous systems with a library that can reap the available computational power on available machines.

The remainder of this paper is organized as follows. Section 2 describes the architecture of PMPI. Section 3 describes the programming model of PMPI. Section 4 explains the sample application used in the tests. Section 5 describes the results and some preliminary performance figures. Finally, section 6 concludes and discusses future and related works.

2. ARCHITECTURE

PMPI architecture follows the standard structure of a layered networking architecture. PMPI is composed of three components. The first component is PMPI which contains MPI implementation. The second one is the agent that runs on each node participating in the MPI computation. The agent is responsible for starting MPI programs on nodes, and offers administrative information about nodes, in addition to information about administrative domains. The third component is PMPI Gateway, or PIP (Platform Interface Portal). The PIP serves as a gateway to administrative domains to overcome problems raised by firewalls and NAT separating different administrative domains.

Each administrative domain has a PIP known to all agents. Inside PMPI component, there is an address resolution layer that is transparent to programmers. This layer decides on whether to direct MPI calls directly to other nodes or to their corresponding PIPs. This permits programmers the freedom to concentrate on their problem rather than communication implementation.

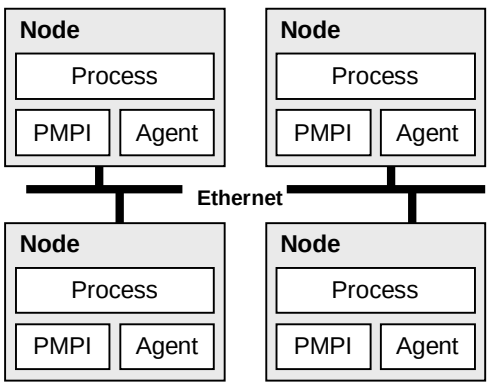

Figure 1: Four nodes using PMPI

Figure 1 shows a basic PMPI infrastructure. The figure shows a structure with four nodes running on one administrative domain connected by local Ethernet network. The processes may be running on different platforms, and each process may be written in a different programming language.

PMPI communication infrastructure is constructed on .NET Remoting, and in turn, is based on TCP/IP. .NET Remoting can be customized to support other protocols [Rammer 2002].

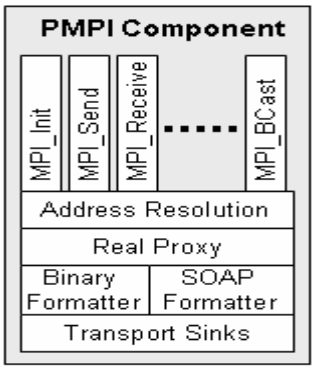

Figure 2: PMPI layered view

Figure 2 shows PMPI component layers. On the top, we have the MPI interface that is available to programmers. When a MPI call is made, it passes through the address resolution module to check which administrative domain the destination node belongs to, and what communication method is to be uses to reach the node that costs less. For example, nodes behind firewalls may be reachable only through port 80 using the SOAP protocol which is firewall friendly in contrast to the binary protocol. On the other hand, SOAP consumes more network bandwidth and is less efficient than binary formatting [McLean 2003].

Figure 3 shows a sketch of a MPI computation spanning two administrative domains where each administrative domain is located behind a firewall. In this figure, MPI calls made from one administrative domain to the other are done through the PIPs of the administrative domain. The PIP will serve as a proxy on behalf of nodes making the call. The scenario in figure 3 assumes that we have barriers in both administrative domains. In other words, nodes in administrative domain 1 cannot reach nodes in administrative domain 2 directly using remote object calls. Instead, they should use the PIP proxy service to exchange messages.

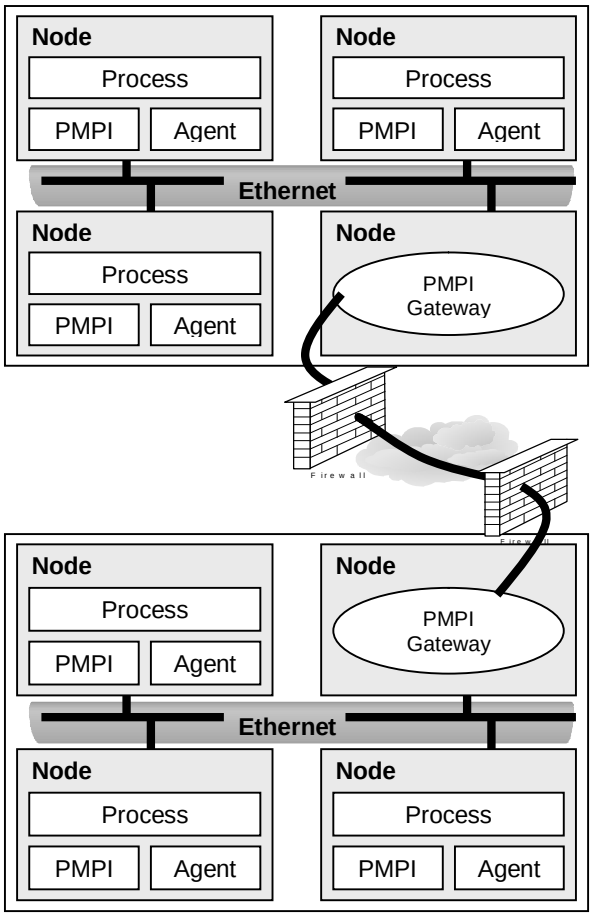

Figure 3: Using PMPI on two administrative domains

To better understand the idea, let's take an example where node A in administrative domain one will make MPI call to node B in administrative domain two. The address resolution layer of PMPI running on node A detects that node B is running on another administrative domain and there is no way to reach node B directly because of a firewall or NAT. The address resolution layer directs the call to the PIP node of administrative domain one. The PIP in turn directs the MPI call to PIP of administrative domain two. The PIP of administrative domain two receives the call and directs it to node B of its domain. If the call is synchronous, then the PIP of administrative domain one block node A until it receives a notification from PIP of the other administrative domain that node B has received the call. The PIP acts as proxy on behalf of the nodes in their corresponding administrative domain.

The rest of this section is divided into two subsections. The first describes MPI standard. The second describes PMPI architecture and constructs.

2.1 MPI: Message Programming Interface

In the message-passing library approach to parallel programming, a collection of processes executes programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages. MPI is a complex system. In its entity, it comprises 129 functions, many of which have numerous parameters of variants [Foster 1995].

In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor. However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as multiple program multiple data (MPMD) to distinguish it from SPMD model in which every processor executes the same program.

Processes can use point-to-point communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications. A group of processes can call collective communication operations to perform commonly used global operations such as summation and broadcast. MPI's ability to probe for messages allows asynchronous communication. Probably MPI's most important feature from a software engineering viewpoint is its support for modular programming. A mechanism called a communicator allows the MPI programmer to define modules that encapsulate internal communication structures [MPI Forum 1994].

2.2 PMPI Basic Architecture

PMPI is built on top of .NET framework. We are using Microsoft .NET framework 1.1 for Microsoft Windows and Mono 1.0.5 for Linux. Although Mono can run on Power PC, BSD and other operating systems and architectures, we based our initial implementation on Windows and Linux operating systems although this can be expanded to other operating systems without any modification in the code.

The initial implementation of PMPI was devoted to implement functionality rather that performance. Because of this, we selected higher level implements of the .NET framework to implement PMPI. For the communication layer, we used .NET Remoting which is based on remote object communication. The classes that make up the .NET framework are layered, meaning that at the base of the framework are simple types, which are built on and reused by more complex types. .NET Remoting is one such complex type which in turn is built as layers where each layer can be customized to programmer needs [Jones et al 2004]. This adds extra overhead compared to using simple raw classes such as socket class [Rammer 2002].

We used C# as the programming language. All .NET programming language compilers targets the CTS (common type system) of the framework. C# compiler helps the programmer adhere to CTS types by setting the "CLSCompliantAttribute" attribute to true [Bock 2003]. In this way, the compiler generates an error whenever you try to use a non CTS type. This guarantees that the generated code is accessed by all .NET programming languages since all .NET programming languages target the CTS [Ritchter and Balena 2002].

Each node participating in the MPI computation should have the .NET framework installed. Nodes running Windows operating systems should install Microsoft Framework 1.1 on their machines. Nodes running Linux should install Mono 1.0.5. Although there are newer versions of the framework for both platforms, PMPI has been tested on earlier frameworks.

In addition to the framework installed on the machines participating in the MPI computation, the nodes should have PMPI installed on each node. The initial implementation of PMPI needs to have bidirectional communication between the nodes. Accordingly, firewalls can cause problems. The implementation of PIP is not yet implemented.

Initially, PMPI implemented 20 MPI functions. Those functions cover basic, asynchronous, collective and modular commands. When MPI computation starts, each node registers PMPI object at a known end point to other nodes using .NET remote object. With .NET remoting, the framework creates a thread pool to receive the calls made against the remote object. When node A sends data to node B within the same administrative domain, node B's PMPI will receive the data and releases the calling object immediately, node A in this. When node B calls MPI_Receive, PMPI will check to see if there is a message with the corresponding tag and source. If it finds a corresponding message, then a pointer to the message is passed to MPI_Receive, and the call returns immediately in node B. If no corresponding message is found with the requested tag-source, the call in node B is blocked until node B receives the requested message. If node A uses synchronous MPI_SSend, then PMPI layer on node A blocks until node B

sends a release signal after the process in node B makes a call to MPI_Receive.

PMPI uses a hash table data structure to control received message. The key of the hash table is a combination of the source, tag, and communicator ID. The value of the hash table points to a queue whose elements contains a data structure composed of the received message, message size, message type and synchronization objects that the receiving thread will block on. When the node calls MPI_Receive with a particular tag, source and communicator, PMPI checks the hash table for pending messages in the queue. If it finds a message, it pops the message from the queue in a FIFO manner and wakes up the thread using the synchronization objects found in the read queue element. When the waked thread terminates, the message is passed to the MPI_Receive call. Note that if the call is made using MPI_Ssend, which is a synchronous send, the receiving thread will block the sending thread until it is waked up again by MPI_Receive in the manner explained above. If it comes that MPI_Receive is called before a MPI_Send and PMPI finds the queue empty, then it blocks the call on synchrozination objects, enqueue the call with the synchronization objects in the queue whose pointer is stored in a hash table. Later, when PMPI is invoked by MPI_Send, PMPI checks first if a pending MPI_Receive exists. If it find a pending receive, then it pops the queue, wakes the thread using the popped synchronization objects and returns.

When it comes to collective operations, PMPI uses a thread pool to perform the collective task. PMPI uses a simple algorithm for collective tasks. Each communicator has a master node known to all participating nodes. The communicator master node is responsible for coordinating the collective calls. In other worlds, its the master communicator node who decides when the collective call is done. PMPI implements this by using a thread pool in the communicator master node. When the collective call is made, PMPI checks if the node is the master in the target communicator. If it is not, then it uses a methodology similar to Send_Receive explained before. If it finds the node to be the communicator master, then it creates one thread for each node in the communicator, and blocks on the synchronization object. When the thread in the pool terminates, it verifies if other threads in the pool had terminated; if not, then the thread blocks on a synchronization object. If the thread happens to be the last one, then the thread wakes all other threads using the synchronization object. By this means, the communicator master manages the collective operation.

The agents will be a separate component. For MS Windows, the agent is implemented as Windows Service. The agent will be responsible for starting the programs on participating nodes. In addition, the agent will supply managing data about the nodes themselves such as available memory, CPU load, speed, administrative domains and other managing data. Today, most operating systems implement the Web-Based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment. WMI is the Micorsoft implementation of WBEM.

The PIPs are part of PMPI architecture but are not yet implemented. PIPs will be implemented using Web Services. The remote object model explained will be substituted by Web Service model. The PIP will be a gateway on behalf of the calling node. The architecture and implementation of PIP will consider having two communicating PIPs on behalf of the send and receiving nodes.

3. PROGRAMMING MODEL

The programming model is as simple as any existing MPI implementation. The master node initializes the MPI computation using a XML computation file. PMPI is object based. Therefore, the MPI functions should be called as object methods.

When PMPI is initialized, it publishes a remote object at a known end point. Each participating node knows the address and port of all other nodes in the MPI computation. When the program calls a MPI function, PMPI receives the function call and transmits it to the corresponding node after resolving its address internally. Although current implementation did not target nodes running behind NATs and firewall, PMPI layered implementation makes it easy to build semantics to solve the complications raised by firewalls and NATs with out programmer awareness. This helps the programmer to devote his efforts on programming rather than MPI communications. Future works will customize the real proxy of the .NET Remoting object to intercept message calls and select the destination accordingly.

We wrote applications in VB.NET, C#, managed C++, and J#. We ran each application on a different node. All four nodes ran under Microsoft Windows XP operating system. For MONO running on Linux Redhat 9, we were limited to C# since it is the only existing non-beta compiler. For simplicity, we used only the above programming languages, but this can extend to any available .NET programming language. The MPI computation ran as if programs

at all nodes were written in the same programming language.

```
MPI obj = new  MPI();
obj.MPI_Init(args);
id=obj.MPI_Comm_Rank(MPI_Comm_World);
tasks=obj.MPI_Comm_Size(MPI_Comm_World);
obj.MPI_Send(offset, 1,
             MPI_Integer, dest, mtype,
             MPI_Comm_World);
obj.MPI_Send(rows, 1, MPI_Integer, dest,
              mtype, MPI_Comm_World);
```

Figure 4: Part of the sample application

Figure 4 shows part of the sample application written in C# where the code initializes an MPI computation, gets its task Id within COMM_WOLRD, gets COMM_WORLD size, sends data to "dest" node and later receives data from "dest" node. Note that the MPI functions are methods of a PMPI object called "obj". These methods are either static or instance methods. Static methods of PMPI enable us to write multithreaded programs running on a machine where all threads use the same PMPI object. Also, it is possible to start multiple PMPI objects where each object participates in a different MPI computation with out the need to MPI communicators.

4. SAMPLE APPLICATION

We used as a sample application the master-worker model for matrix multiplication (A x B = C). The results of this sample are compared to MPICH2 for Windows in the next section.

The master (task Id 0) sends matrix B to all participating nodes (workers), and distributes the rows of matrix A into worker nodes evenly. Workers perform the multiplication and send back the result to the master node. Master node accumulates the results from all workers into matrix C. The sample application was taken from the examples that install with MPICH2. In this sample application, the master does not participate in the MPI computation. It just sends the data to workers and gets back the results into matrix C.

5. RESULTS

The performance tests are done with the sample application written in C#. We set the number of columns in matrix A to 1200 and the number of columns of matrix B to 500. We varied the number of rows of matrix A to 2400, 4800, 9600 and 19200 respectively. For each problem size, we executed the application on one to all six nodes.

The tests are executed in three sets. The first set of tests is the results obtained executing the sample application on a homogeneous platform corporate network. The second test is done on the same corporate network with both PMPI and MPICH2. The last test is done on a cluster using homogeneous and heterogeneous platforms.

5.1 Results using Corporate Homogenous Platform

We tested the application first on standalone machines with out using parallel MPI computation. We rewrote the application taking out all MPI commands and compiled them using Microsoft Visual C++, Microsoft C# and MONO C# compilers.

The corporate network was composed of AMD 1.5 GHZ, 512 KB cache CPUs with 256 MB RAM and 40 GB HD. The nodes run under Windows XP. One node had dual operating systems: Windows XP and Redhat 9. The obtained results are as follows. C# managed code application executed 27% slower than C++ application on machine running Windows XP or Windows 2003 operating system. On machine running Linux Redhat 9 with mono .NET framework, C++ executed 10 times faster than C#! Comparing .Microsoft NET C# running on Windows XP to MONO 1.05 C# compiler Running under Linux Redhat 9, Microsoft C# executed 5 times faster than MONO C#.

Before going any further, let me clarify some details about array access in managed world and some performance issues. Each time an element of an array is accessed, the CLR ensures that the index is within the array's bound. This prevents you from accessing memory that is outside the array, which would potentially corrupt other objects. If an invalid index is used to access an array element, the CLR throws a System.IndexOutOfRangeException exception.

The index checking comes at a performance cost. If we have confidence in our code, we can access an array without having the CLR perform index checking. This feature is not allowed in all .NET languages and is not CLS complaint. Accordingly, only .NET languages that have this feature will benefit from fast array access such as C#.

To give an idea on how much gain we get using fast array access, we show the following results. C# using managed array access executes 20% slower than C# using fast array access on the machine running Windows XP. On Linux, C# using managed array access, executed 5 times slower than C# using fast array access. As we note, the performance gain in Linux is huge (500%).

The problem with fast array is that not all .NET languages support it since it is not a CLS compliant. In addition, it is harder to code than managed array access since it uses pointers. Accordingly, the benefit of using fast array is limited to only a subset of .NET programming languages.

Later, we executed the application using both MPICH2 and PMPI using managed array access with PMPI. The sample application running on PMPI nodes was written with C#, Java.NET, managed C++ and VB.NET. The compiler choice did not affect the result. We used a various combination of the programming languages and we got the same results. The results are shown only for Windows OS since we used MPICH2 for windows.

In figures 5, we show a comparison between PMPI and MPICH2 for different problem sizes executing on 6 nodes. The results demonstrate that PMPI executed slower than MPICH2 between 40% and 70%.

Figure 6 shows the linear relation ship between the number of nodes and the execution time. As we increase the participating nodes, the execution time decreases linearly.

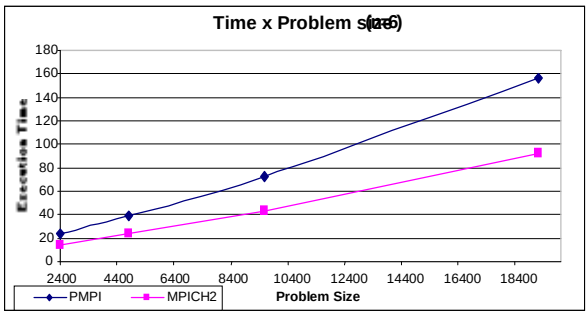

Figure 5: comparison between PMPI and MPICH2

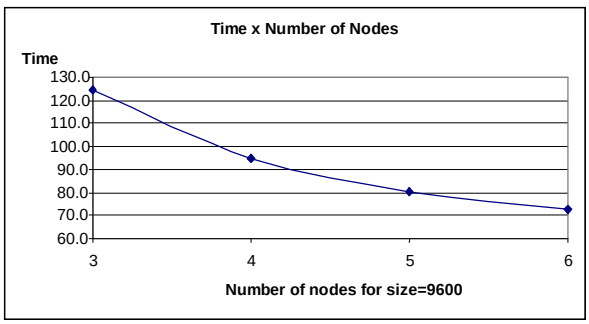

Figure 6: Execution time as a function of participating nodes

5.2 Results using cluster with a Heterogeneous Platform

The cluster, named BIO, is composed of 8 nodes each with dual 2.0 GHZ, 512 KB cache CPUs with 512MB RAM and 40 GB HD.

As before, we tested the application first on a standalone machines with out using parallel MPI computation. We rewrote the application taking out all MPI commands and compiled them using Microsoft C# compiler and mono C#.

Later, we executed the application on the cluster using up to six nodes where nodes varied between nodes running Windows 2003 server and nodes running Linux Redhat 8. The result is shown below in figure 7. As the figure shows, Microsoft .NET platform performed better than MONO .NET framework. When we mixed the nodes between Windows and Linux operating systems, PMPI executed with performance equivalent to the average of executing on each platform independently.

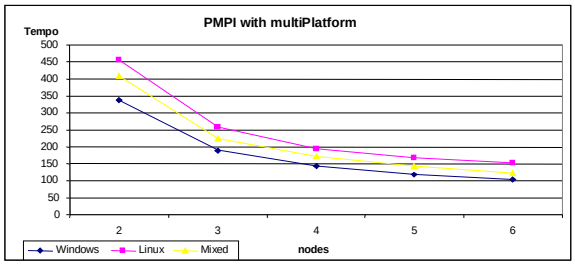

Figure 7: PMPI on a heterogeneous platform

5.3 Result analysis

As shown in section 5, PMPI executes as a linear function of the problem size. The execution time increased linearly as we increased the matrix size. Also, as we increase the number of nodes, the execution time decreased almost linearly.

Although PMPI executes slower than MPICH2, the main overhead is a result of managed array access and the use of high construct communication construct of the .NET framework. This overhead was expected and is subject for future work.

In addition, we detected that the use of thread pool within the program structure, degraded PMPI performance in a master-worker model. This loss of performance resulted from the fact that the operating system has full control of the thread pool which resulted in activating threads to receive the results from nodes while other threads were still sending data to other nodes. With a custom thread pool, PMPI will have full control on the executing thread, and in turn, can block receiving threads while PMPI is sending. This will improve a lot performance especially when we have large number of nodes. This happens because as we increase the number of nodes, we have greater the tendency of nodes completing their jobs before the master.

Moreover, there are some other code tuning of PMPI that can improve performance such as reducing .NET framework boxing, a mechanism that .NET framework exchange data between the allocated stack and managed heap. Boxing in .NET managed code is known to have performance cost and minimizing it can improve performance a lot.

6. RELATED WORKS

In this section we discuss related work the can be use parallel computing on a multiplatform. In [Fer05], experiment with implementation of parallel programs using C# running on Unix and Windows is done. In [Will01], a binding between an already implemented MPI interfaces and C# is done. In [Car00], a Multiplatform MPI implementation is done for JAVA programming language. However, none of the above works have focused and worked with a Multiprogramming Language MPI.

7. CONCLUSION AND FUTURE WORKS

The first implementation of PMPI was shown to be feasible and it is possible to execute MPI standards on a multi-language and multiplatform systems. Although the first implementation showed that PMPI is slower than MPICH2, the difference is explained by known issues and these issues can be eliminated. Care should be taken when using a heterogeneous system including Linux with managed array access. As shown in the preliminary results, mono performs very poor with managed array access. In such a case, we should consider using fast array access.

The next step in this project is to span PMPI to multiple administrative domains that span geographic area across the internet. In addition, lower communication constructs can improve performance in addition to use a custom thread pool to manage threads instead of the operating system thread pool. This will give us a complete control on the threads. Also, we will do a comparison between JavaMPI to PMPI .

REFERENCES

[And00a] Andrews, G.R. Foundation of Multithreading, Parrallel, and Distributed Programming, pp 115-243, 2000.

[Rit02a] Ritchter,J. and Balena,F. Applied Microsoft Dotnet Framework Programming in Microsoft C# 2002.

[Fos95a] Foster, I.. Designing and Building Parallel Programs, pp 275-310, 1995

[Don03a] Dongarra,J. and Foster,I. and Fox,G. and Gropp, W., Kennedy,K. and Torczon,L. White,A. Sourcebook Of Parallel Computing. 2003.

[Ram02a] Rammer, I. Advanced Dotnet Remoting in C#.2002.

[Boc03a] Bock,J. and Barnaby,T. Applied Dotnet Attributes. 2003

[East04a] Easton, M.J. and King, J. Cross-Platform Dotnet Development. 2004

[Jon04a] Jones, A., Ohlund,J. and Olson, L. Network Programming for the Microsoft Dotnet Framework. 2004.

[Ard02a] Ardestani, K. and Ferracchiati, F. and Gopikrishna,S., Redkar,T., Sivakumar, S., Titus, T. Visual Basic Dotnet Threading. 2002.

[Sha03a] Sharp, J. and Jagger, J. Microsoft Visual C# Dotnet. 2003.

[McL03a] McLean, S. and Naftel,J. and Williams,K. Microsoft Dotnet Remoting. 2003.

[Mar04a] Mariani,R. , Bohling, B., C.Smith, and S.Barber. Improving Dotnet Application Performance and Scalability. 2004.

[MPI94a] MPI FORUM. 1994. The MPI message passing interface standard. University of Tennesse,Knoxville.

[MON05a] The MONO project. http://www.go-MONO.com

[ECMa] ECMA ISO/IEC 23270, ISO/IEC 23271 and ISO/IEC 23272. http://www.ecma.ch and http://msdn.microsoft.com/net/ecma

[Kel02a] Kelly,W., Roe,P. and Sumitomo,J. , *G2: A Grid Middleware for Cycle Donation using Dotnet* , The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, June 2002.

[Kel02b] Kelly,W. and Roe,P., *Donating Cycles over the Internet Using Web Services* , The Eighth Australian World Wide Web Conference, Sunshine Coast, July 2002

[Fer05] Ferreira, F and Sobral, Joao, *ParC#: Parallel Computing with C# in .Net*,* Springer-Verlag Berlin Heidelberg 2005

[Will01] Willcock,J and Lumsdaine,A and Robison,A, Using MPI with C# and the Common Language Infrastructure Indiana University Computer Science Department Technical Report 570

[Car00] Carpenter,B, Getov,V, Judd,G, Skjellum,T and Fox,G MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience* Volume 12, Number 11. September 2000