Message Passing over .NET-based Desktop Grids

Carlos Queiroz, Marco A. S. Netto, Rajkumar Buyya

Grid Computing and Distributed Systems Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
ICT Building, 111 Barry Street, Carlton, VIC 3053
{carlosq, netto, raj}@csse.unimelb.edu.au

Abstract. The use of virtual machines and the object-oriented paradigm has became popular in both industry and academy. The .NET platform is similar to Java but with advantages such as supporting several programming languages from script-based to strong typed ones. At the same time, research on harnessing the computing power of idle desktop machines has been carried out. In this context, Alchemi, a desktop grid, has been developed to provide services such as user management, data file transfer, and application scheduling. The Message Passing model, responsible for enabling inter-process communication, is broadly used for developing complex parallel scientific and engineering applications and is neither available on Alchemi or any other .NET based environment. In this work we present the design, implementation and evaluation of a novel .NET-based implementation of two message passing models, Message Passing Interface (MPI) and Bulk Synchronous Parallel (BSP), over the Alchemi's Desktop Grid.

1 Introduction

Desktop computers located in universities, organizations and home environments are underutilized most of the time. These machines offer a considerable computing power that can be harnessed to execute complex parallel and distributed applications. Naturally, executing such applications on these machines imposes some challenges since they are not designed for this purpose. Therefore, in order to leverage the networked computing power of desktop computers, several projects have developed software infrastructure to execute applications modelled as parameter sweep applications. One of the most successful projects that use such a model is SETI@home (Search for Extraterrestrial Intelligence) [1]. One of the reasons for this success is its simplicity in enabling contributors to donate computational resources—when the computer screensaver is activated the application starts by making a request to a remote server to download tasks to be processed. Another reason is its support for Windows operating system, since the majority of the desktop machines around the world run Windows. Based on the same concept of SETI@home there are BOINC@home [2], FightAIDS@home [3], and Folding@home [4]. All of these projects are primarily targeted for applications that can be expressed as parameter-sweep applications. They have

no or lack of support for creating applications consisting of tasks that need to communicate and coordinate their activities by exchanging messages among themselves.

Nevertheless, there are important applications that require inter-process communication. For example, in a forecast weather application, where each application process is responsible for evaluating a part of a map, the common map borders must be synchronized between two or more processes. This kind of application is challenging compared to embarrassingly parallel applications (e.g. parameter sweep or Bag-of-Tasks). That is because a machine failure does not compromise the execution of the entire application since the other processes are independent. Also, there is no problem if one process executes a task faster than the others. On the other hand, these issues are a problem for message passing applications; a machine failure has to be handled to stop the whole application and the differences on machine performance have to be carefully handled to minimize the influence in the final result. Issues such as scalability, security, and scheduling are even more critical for message passing applications due to the inter-process communication in dynamic and open network environment.

In order to tackle with the challenges of creating and using Desktop Grids, middleware systems must provide simple and efficient mechanisms for both Grid application developers and computing resource donators. In addition, they also need to embrace modern and standard based network application development and execution environments such as .NET. Alchemi [5] is one of the desktop Grid middleware technologies that meet these requirements. However, like many other related Desktop Grid environments under Windows and .NET environments, it was primarily targeted for task-farming applications and had no support for message passing applications. To over this limitation, we have extended Alchemi environment by implementing two message-passing models: Message Passing Interface (MPI) and Bulk Synchronous Parallel (BSP) model.

The main contributions of our work are: (i) devising message passing model for parallel programming on .NET-based Desktop Grids; (ii) design, implementation and the integration of two message-passing libraries into Alchemi Grid environment; and (iii) performance evaluation of the implemented libraries.

The rest of the paper is organized as follows. Section 2 describes the two message-passing models we ported; Section 3 presents the related work and discusses the limitations of existing solutions; Section 4 describes the design, implementation details and integration of the libraries into Alchemi; Section 5 provides performance evaluation results; and Section 6 concludes the paper and discusses the further work.

2 Message Passing

Message Passing is a communication paradigm to develop parallel and distributed applications that require inter-process communication. The Message Passing Interface (MPI) and the Bulk Synchronous Parallel (BSP) computing models are specifications for parallel processing that have numerous different

implementations. For MPI, the most well-known and current available implementations are LAM/MPI* and MPICH**. On the BSP model, BSPLib [6] is the most famous one.

MPI [7] is a *de facto* standard for developing high-performance applications for parallel computers. It provides an expressive number of functions, 128 in MPI-1 and 194 in MPI-2, to perform both point-to-point and collective communication procedures. It also supports several types of communication, for instance, synchronous and asynchronous communication through blocking and non-blocking send and receive functions. Although there is a considerable complexity on developing applications using this model it has been successfully used by most of researchers in high-performance computing field.

Inspired in the von Neumann model, Leslie Valiant, in 1990, introduced the Bulk Synchronous Parallel (BSP) computing model as a bridging model to link hardware and software for parallel computation [8]. The BSP model is compatible with the conventional SPMD/MPMD (single/multiple program, multiple data) model and has both remote memory access and message-passing support. In BSP, a parallel program executes as a sequence of parallel *supersteps*, where each superstep is composed of computation and communication followed by synchronization barriers. The barriers act as a process synchronizer; the processes must achieve the same point of execution for then continue their computation. In the context of non-dedicated machines, differently from MPI, these barriers assist checkpointing making it easy to be implemented. Even though MPI provides support for barriers they are not mandatory.

3 Related Work

There are Desktop Grid systems being developed to support parallel applications in Windows/.NET environments. They include application-specific Desktop Grid systems, such as SETI@home [1], FightAIDS@home [3], Folding@home [4]; and general purpose Desktop Grid systems, such as BOINC@home [2], United Devices Grid MP, and Alchemi. All these systems are primarily targeting for parameter sweep type applications. Recently, a number of works have investigated possibility of supporting message-passing applications within Desktop Grid systems.

In Bayanihan system [9] users donate their resources by pointing their browsers to a particular website to be part of the parallel network. The system is based on Java and consequently can be used in Windows environments. It supports BSP applications and checkpointing facility. One limitation of this solution is that donators have to login to their machines and access a specific website. As the user application is a Java applet the web browser has to be on that specific website as long as the user wants to donate computational resource. If we consider organizations, or laboratories in universities, where people are encouraged

* LAM/MPI project site: http://www.lam-mpi.org/

** MPICH project site: http://www.mcs.anl.gov/mpi/mpich/

to logout of their computers stations as soon as they leave their desks, this approach is not feasible. Another problem is that all messages are sent through the server machine which becomes an enormous bottleneck.

InteGrade [10] supports execution of parallel applications based on the BSP model [11]. The BSP support is based on Oxford BSPlib with all inter-node communication in CORBA. The BSP API is not part of InteGrade core. However, it has access to all InteGrade services, such as checkpointing and authentication. InteGrade does not provide support for Windows and user applications must be written in C to support checkpointing. Currently, they support UNIX based systems only.

BSP-G [12] is a project that aims at fostering the Globus Toolkit (GT) services to create an API for developing and executing BSP applications over GT. BSP-G is not aimed at executing applications on desktop machines and also relies on Globus, which has a poor support for Windows environments. PUBWCL [13] is a Java-based system aimed at executing BSP applications on non-dedicated machines. It provides support for load balancing, fault tolerance and process migration at run-time through the use of JavaGo. It is a standalone library that does not take benefit from middleware functionalities, such as user management, resource discovery, security, and so forth. MPICH-V [14] is the representative project in the context of executing MPI applications over Desktop Grids. It is an MPI implementation based on MPICH and its main focus is on fault tolerance. Similar to PUBWLC, MPICH-V does not benefit from Grid middleware, which means, it is one more library for executing MPI applications over volatile resources. MPI-C#-CLI [15], although implemented on .NET, it is a binding port that still relies on native MPI implementations by calling external libraries into C# applications.

As it can be noticed there is no available support for execution of parallel applications (message-passing paradigm) on middleware running over Windows-based Desktops. The goal of our work is to overcome this lack by implementing support for two well-known message passing models, MPI and BSP, over a Grid middleware based on Windows and Alchemi.

4 Message Passing over Alchemi

In this section we briefly introduce the Alchemi's Grid middleware then we discuss some benefits of relying on a Grid middleware for the development of message-passing libraries and the use of these libraries to create parallel applications. Further, we describe the design and current implementation of our libraries.

4.1 Leveraging Grid Middleware Systems

Relying on a Grid Middleware brings several benefits. Some of them are: security, resource discovery, user and data management, and scheduling. In order to leverage such functionalities, we developed our libraries on top of Alchemi [5], a

Windows-based Desktop Grid computing middleware implemented on Microsoft .NET platform. Alchemi is designed to be user friendly without sacrificing power and flexibility. It provides run-time machinery and a programming environment (API), which is required to construct Desktop Grid applications. It also supports execution of cross-platform applications via Web Services submission. The Alchemi middleware relies on the client-server model—a manager is responsible for coordinating the execution of tasks sent by the executors (desktop machines).

The key features supported by Alchemi are Internet-based clustering of desktop computers without a shared file system, federation of clusters to create hierarchical cooperative grids, dedicated or non-dedicated (voluntary) execution by clusters and individual nodes, grid thread programming model (fine-grained abstraction), and a Web Services interface to support a grid job model (coarse-grained abstraction) for cross-platform interoperability (e.g. for creating a global and cross-platform grid environment using a custom resource broker component).

Regarding to the use of APIs, it is important to point out that most of the BSP and MPI libraries are implemented to be used in C, C++ and FORTRAN languages. On these implementations developers must work on low level data types and provide several parameters, such as array of bytes, array size, and array data type for some functions. As we are working top of .NET framework, all these parameters can be reduced to a single object. Another advantage concern to the developers of parallel applications. They can use high level languages supported by the .NET, such as C#, C++, J++, Visual Basic, and Python.

4.2 Message Passing Implementation

In this initial implementation we provide the basic functions for MPI library and the BSP model. Some of these functions are very similar for both MPI and BSP. Below are the main functions we have implemented for MPI and BSP:

1. *MPI_Init/bsp_begin* – message passing initialization method.
2. *MPI_Finalize/bsp_end* – message passing finalization method.
3. *MPI_Comm_size/bsp_nprocs* – returns the total number of processes.
4. *MPI_Comm_rank/bsp_pid* – returns the process identifier.
5. *MPI_Send/bsp_send* – sends data to a remote node.
6. *MPI_Recv/bsp_*move – reads the local receiving buffer.
7. *MPI_Barrier/bsp_sync* – synchronizes the processes.
8. *MPI_Bcast/bsp_bcast* – sends data to multiple processes.

Note that even though these functions are very similar, they have some peculiarities for each model. For instance, the *MPI_Send* function allows defining the type of the message but the *bsp_send* does not. Taking this into account we have designed a single core architecture that is used by both environments. The *send* and *receive* (*move*) functions, as well as, *rank* (*pid*), *init* (*begin*), *finalize* (*end*), and so forth are implemented in a core library and a wrapper is used to keep up with differences in the function signatures for each specification that was implemented.

Initialization and Finalization of processes. When a process starts the message passing initialization method it contacts the Alchemi manager to register it on a table of processes where the rank (process id) is created for each one. This table keeps the identification of each process: the process id, the process IP address and the process port where they are listening to new connections. This identification is required to allow the processes to find each other during the execution. Once a process executes the finalization procedure it contacts the manager to remove itself from the table.

Sending messages. To keep up to differences between the send messages method on the libraries and to simplify our development we have created a class that encapsulates all parameters defined on both send methods. An object of this class is passed as a parameter to our send method that is responsible for sending the message to the remote process. The broadcast methods are a variant of the send method. When the message passing finalization method is called the process id is removed from the managers executor table and its listener for new connections is closed.

Synchronization barriers. MPI barriers are used only to synchronize the execution of application processes. However, on BSP they are responsible for managing the execution of supersteps. In our implementation the master process, who usually receives the process identification zero, is responsible for controlling the synchronization barriers. When the master process calls the synchronization procedure, it checks whether all executor signaling messages have been received. If not, it locks itself, waiting for the signal of the other processes. When a slave process calls the synchronization procedure, it sends the master process a signal message and it remains locked. When the master process receives the signal messages from all processes, it sends a message back to all others processes in order to release them to continue the execution (in BSP, the next superstep). After this, the master also continues its execution.

Receive operation. The receive operation on MPI and BSP are similar but with a peculiarity. In MPI a process can read a message from any part of the receiving buffer queue. Using the MPI receive function it is possible to specify the source process (who sent that message) and also the type of the message (a tag to identify a message). But in BSP processes can only read from the first position of their receiving buffer.

4.3 Integration with Alchemi environment

Alchemi supports two application models: Thread Model and Job model [16]. The Thread Model is used for applications developed natively for Alchemi (by using Alchemi API). The model defines two main classes: GThread and GApplication. A GApplication is a set of GThreads and a GThread is a unit of execution. The Job Model has been designed to support legacy applications (those not developed by Alchemi SDK). Every executor that executes a legacy application receives a Job object for processing. In our case we make use of the GApplication class. Therefore, the input and output data management and submission of the processes to each executor are responsibility of Alchemi.

In order to execute a parallel application, usually users provide the number of processes and the application name. Other options such as machine files, standard error and output can also be used in other implementations (e.g. MPICH). We support the same features but with different approaches. We developed the BSPRun and MPIRun for submission of BSP and MPI applications, respectively. Because we have the same library supporting both implementations, these runners are only a wrapper for our main runner application that is responsible for calling the GApplication that then creates input and output files and submit our library and the parallel application for execution. When the manager receives these data it defines on which machines the application should run (based on a schedule algorithm) and then send the data to executors.

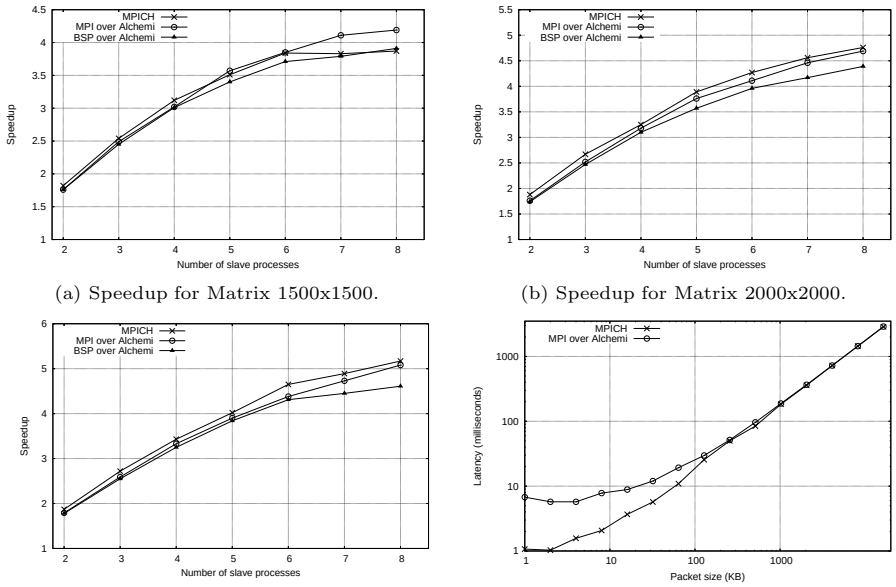
In cluster computing environments, in general, there is a resource manager responsible for creating a list of machines the user can rely to run the application. In our case, Alchemi (Manager) is responsible for finding and selecting the machines. Thus, the runner simply assigns the processes from a list of machines received by the Manager. This process is completely transparent for the users. They only need to specify the number of processes. The steps to execute message-passing applications on Alchemi are basically the same to execute the parameter sweep applications already supported by Alchemi. However, the users now make use of the runners responsible for loading the message passing libraries. There are four main steps to execute the parallel applications into Alchemi Grid middleware:

1. User submits an application to Alchemi Manager (in our case an MPI or a BSP application using MPIRun or BSPRun respectively) by specifying the number of machines and the application.
2. Alchemi Manager selects the nodes to run the application.
3. The executors (Alchemi nodes) start to run the application (in our case the nodes communicate with each other).
4. The results are sent back to the user, through the Manager, that saves them in files, one for each process.

5 Performance Evaluation

In order to evaluate our MPI and BSP implementation on Alchemi, we setup an environment, composed of 9 machines (*Dell OPTIPlex GX 2f0 Pentium IV 3.40 GHz, 1.5 G of RAM and 100 Mbps network device running Windows XP SP2 and .NET 2.0*) connected by a 100 Mbps switch. Moreover, to compare our libraries with a well-known message passing implementation, we also performed our experiments using MPICH2 v1.0.4, over Cygwin 1.5.X and GCC 3.4, on the same Windows machines. We executed an application for multiplication of matrices in order to measure the speed up of both libraries. In this application the master process assigns one matrix and also a portion of the second matrix to each one of the slave processes. In the MPI application, the slave process can compute data as soon as it receives the work from the master. In the case

of the BSP application, all slave processes must receive the work and synchronize for then to be able to start the computation. The implementations are also different in regarding to receiving the results from slave processes. In the MPI application, the master process can receive the results from any slave as soon as a slave finishes the execution. In the BSP application, all slaves must finish the execution, send the results to the master and synchronize before the master has access to the work done. In our experiments we varied the matrix dimension size, as well as the number of processors—the master that distributes the work and collects the results is not taking into account as an executor. We performed the experiments on three different matrix sizes. In Figure 1 (a) we can observe that until 4 slave processes, MPICH provides better results than our implementation, but after that, the MPI over Alchemi provides a better speedup. In Figures 1 (b) and (c) MPICH overcomes our BSP and MPI libraries. However, we can observe that the results regarding MPICH and MPI over Alchemi are considerably similar.



(a) Speedup for Matrix 1500x1500.

(b) Speedup for Matrix 2000x2000.

(c) Speedup for Matrix 2500x2500.

(d) Latency through ping-pong.

Fig. 1. Evaluation performance results for .NET-based Message Passing.

We also measure the latency of sending messages between two processes through the "ping-pong" application. From Figure 1 (d) we can observe that for small packets MPICH provides a better performance compared to our implementation. However, as soon as the packet size is increased, both implementations present the same performance.

From these experiments we can conclude that although MPICH overcomes the performance of our libraries in most cases, the benefits are not expressive, in particular in the matrix multiplication. Therefore we argue that the BSP and MPI implementations over Alchemi are a very attractive alternative to develop message passing applications over desktop machines.

6 Conclusion and Further Work

Many talk about leveraging use of idle desktop machines at University laboratories, organizations and homes to increase the computational power and use it for running all sorts of applications, for instance, parallel systems. However the solutions presented so far are based on operating systems not used by these desktop machines, at least on its majority, it is widely known those machines run over Windows. Therefore, there is a clear need for better supporting of parallel applications on the Windows environment.

In this paper we presented our effort to overcome this problem. We described the design, implementation details, and a performance evaluation of two message passing interface libraries over a .NET-based Desktop Grid middleware. We also emphasized the importance of developing the libraries upon a Grid middleware due to the several services that can be leveraged to simplify both the implementation of message passing libraries and their applications. One could argue that the use of MPICH with a cluster resource manager is a better solution in this environment. However, different from Alchemi, existing resource managers are not designed to work with desktop machines, having some limitations such as relying on a shared file system. For instance, in MPICH, the parallel program must be copied manually to each single machine that will be part of execution. In our case, as we leverage the Alchemi infrastructure, this is not required. Alchemi is responsible for sending the user programs to the machines and collecting the results transparently. Also, in our case, as we are working on top of .NET framework, several programming languages supported by the platform, such as C#, C++, J++, Visual Basic, Python, and COBOL can be used.

For future work we intend to perform experiments using more machines and other applications, as well as comparing with other existing implementations in order to have a better evaluation of our implementation. Also we will investigate the checkpointing support through the use of serialisable objects and existing solutions to handle the firewall problems in order to enable multi-site execution. We encourage researchers interested in using Desktop machines for executing MPI or BSP applications to download our libraries and examples at the Alchemis website—http://www.alchemi.net.

Acknowledgement

We would like to thank Krishna Nadiminti, Jia Yu, and anonymous reviewers for their helpful comments.

References

1. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@home: an experiment in public-resource computing. Communications of the ACM **45**(11) (2002) 56–61
2. Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh (2004) 4–10
3. FightAIDS@home: http://www.fightaidsathome.org. August (2006)
4. Pande, V.S., Baker, I., Chapman, J., Elmer, S., Larson, S.M., Rhee, Y.M., Shirts, M.R., Snow, C.D., Sorin, E.J., Zagrovic, B.: Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. Peter Kollman Memorial Issue, Biopolymers **68**(1) (2003) 91–109
5. Luther, A., Buyya, R., Ranjan, R., Venugopal, S.: Alchemi: A .NET-based enterprise grid computing system. In: Proceedings of International Conference on Internet Computing. (2005) 269–278
6. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.H.: BSPlib: The BSP programming library. Parallel Computing **24**(14) (1998) 1947–1980
7. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22**(6) (1996) 789–828
8. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8) (1990) 103–111
9. Sarmenta, L.F.G., Hirano, S.: Bayanihan: building and studying web-based volunteer computing systems using java. Future Generation Computer Systems **15**(5–6) (1999) 675–686
10. Goldchleger, A., Kon, F., Goldman, A., Finger, M., Bezerra, G.C.: InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. Concurrency and Computation: Practice and Experience **16** (2004) 449–459
11. de Camargo, R.Y., Goldchleger, A., Kon, F., Goldman, A.: Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In: Proceedings of the 2nd workshop on Middleware for grid computing, New York, NY, USA, ACM Press (2004) 35–40
12. Tong, W., Ding, J., Cai, L.: A parallel programming environment on grid. In: Proceedings of the ICCS. Volume 2657., Springer (2003) 225–234
13. Bonorden, O., Gehweiler, J., Meyer auf der Heide, F.: A web computing environment for parallel algorithms in java. Scalable Computing: Practice and Experience **7**(2) (2006) 1–14
14. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Hérault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Néri, V., Selikhov, A.: MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In: Proceedings of the ACM/IEEE SC, Baltimore, USA (2002) 1–18
15. Willcock, J., Lumsdaine, A., Robison., A.: Using mpi with c# and the common language infrastructure. Concurrency and Computation: Practice and Experience **17**(7-8) (2005) 895–917
16. Luther, A., Buyya, R., Ranjan, R., Venugopal, S.: Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework, High Performance Computing: Paradigm and Infrastructure. Wiley Press, New York, USA (2005)