Message Passing over Windows-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
{carlosq, netto, raj}@csse.unimelb.edu.au

August 23, 2006

Abstract

Message Passing is a mechanism to enable inter-process communication in parallel and distributed computing. Complex scientific and engineering applications have been developed upon such a communication model to be executed on cluster machines. Due to the huge amount of computer power being wasted in desktop machines, there is an increasing interest in using idle machines to execute these applications. However, most of the current middleware systems are aimed at executing only embarrassingly parallel applications, i.e. with no inter-process communication. Moreover, incredibly, these existing systems are based on UNIX-like when we know that the very most of the desktop machines around the world are based on Windows. In this work we present the design, implementation and evaluation performance of a Windows-based implementation of two message passing models, Message Passing Interface (MPI) and Bulk Synchronous Parallel (BSP), over the Alchemi's Grid middleware.

1 INTRODUCTION

Desktop machines located in universities, organisations and home environments are underutilised most of the time. These machines offer a considerable processing power that can be explored 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 computing power of desktop machines, several projects have been developed software infrastructure to execute parameter sweep applications. One of the most successful projects is SETI@home (Search for Extraterrestrial Intelligence) [2]. One of the reasons of its success is the simplicity to donate computational resources – when the computer screensaver is activated the application starts running then it makes a request to a remote server to download tasks to be processed. Another reason is its support for Windows machines, since the majority of the desktop machines around the world run Windows.

Based on the same concept of SETI@home there are BOINC@home [1], FightAIDS@home [7], Distributed.net [6] and Folding@home [13]. These projects do not provide support for inter-process communication. Therefore, they only support embarrassingly parallel applications.

Nevertheless, there are important applications that require inter-process communication. For example, in a forecast weather application, each application process is responsible for evaluating a part of a map. Some borders are common for more than one process – where the area of a process finishes and the area of another process begins. The processes that have the same border must exchange data to synchronize their computation.

The execution of this kind of application is challenging compared to the execution of 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 to 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.

Problems such as scalability, security, scheduling, are also peculiar for message passing applications due to the inter-process communication.

In order to tackle with the challenges of using Desktop machines, middleware systems must provide simple and efficient mechanisms for development of applications, and also have to simplify the donations of the computational resources. Currently, the Desktop Grid technologies either do not support Windows environment or do not provide message passing programming libraries.

In this paper we present the current status of our effort to provide an easy to use environment for executing message passing applications on Windows-based Desktop machines as well as an API to develop such applications. *The main contributions of our work are: (i) devising message passing model for parallel programming on Windows-based Desktop Grids; (ii) design, implementation and the integration of two message-passing libraries over a Grid middleware; and (iii) performance evaluation of the implemented libraries.*

A Grid middleware is essential in this context since it provides functionalities that can simplify the usage of this environment. In our case we utilized Alchemi, which is based on .NET and is a Windows-based middleware. We ported two message-passing libraries to Alchemi: Message Passing Interface (MPI) and Bulk Synchronous Parallel (BSP) model. In this paper we present the design, current status and some experimental results of these libraries and we also discuss the next steps of this project.

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 the contribution of our work; Section 4 describes the design, implementation details and integration of the libraries into Alchemi; Section 5 provides some 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 implementations are LAM/MPI [1]and MPICH[2]. On the BSP model, BSPLib [10] is the most famous one. Below there is a brief description of each one of these models.

2.1 Message Passing Interface (MPI)

MPI [9] 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 (two-party) and collective communication (multi-party) 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 successfully been used by most of researchers in high-performance computing (HPC) field. Gropp [10] highlights some requirements that must be satisfied for a parallel programming model to succeed and argues that MPI addresses each one of them. Some of them are portability, performance and completeness.

2.2 Bulk Synchronous Parallel Model

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 [16]. The BSP model is compatible with the conventional SPMD/MPMD (single/multiple program, multiple data) model and has both remote memory (DRMA - Direct Remote Memory Access) and message-passing (BSMP - Bulk Synchronous 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 (desktop grids), differently from MPI, these barriers assist checkpointing making it easy to be implemented.

[1]LAM/MPI project site: http://www.lam-mpi.org/
[2]MPICH project site: http://www.mcs.anl.gov/mpi/mpich/

Even though MPI provides support for barriers they are not mandatory. As a consequence this brings more flexibility for developers, as well as a way to achieve better performance.

3 RELATED WORK

As mentioned previously there are several projects that provide support for execution of parallel applications. SETI@home [2], FightAIDS@home [7], BOINC@home [1] and Folding@home [13] are some examples. Users donate the computing power to be used by these systems. However, SETI@home, FightAIDS@home and Folding@home are not middleware systems, but applications developed for a single purpose. On the other hand, BOINC@home is more flexible. It provides a platform in which developers can build their own applications. However, in BOINC@home users cannot develop general parallel applications only embarrassingly are supported due to lack of supporting message-passing applications.

In **Bayanihan** system [14] 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 is already Windows supported. Bayanihan users may execute BSP applications and make use of checkpointing. One problem of this solution is that donators have to login to their machines and access a specific website. As the application is a Java applet the web browser has to be on that specific website as long as the user wants to donate his/her computational resource. If we consider organizations, or laboratories in universities, where people are encouraged to logout 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 what becomes a bottleneck.

InteGrade [8] supports execution of parallel applications based on the BSP model [5]. 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. One limitation presented on the InteGrade is that it does not provide support for Windows that is *de facto* operating system on organizations and universities around the world. Currently, they support UNIX based systems only.

BSP-G [15] 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 that also relies on Globus, which has a poor support for Windows environments.

PUBWCL [3] 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 stand-alone library that does not take benefit from middleware functionalities, such as user management, resource discovery, security, and so forth.

MPICH-V [4] 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.

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, Alchemi.

4 MESSAGE PASSING OVER ALCHEMI

In this section we briefly introduce the Alchemi's Grid middleware and discuss some benefits of relying on a Grid system 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 [11], 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 (VB), Python, and COBOL.

4.2 Message Passing Implementation

In this initial implementation we provide the basic functions for MPI library and all functions for 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* – it is the first function to be called in an MPI/BSP program.
2. *MPI_Finalize/bsp_end* – it is the last function to be called in a MPI/BSP program.
3. *MPI_Comm_size/bsp_nprocs* – it returns the total number of MPI/BSP processes.
4. *MPI_Comm_rank/bsp_pid* – it returns the process identifier of a MPI/BSP process.
5. *MPI_Send/bsp_send* – it sends data to a remote node.
6. *MPI_Recv/bsp_*move – it reads the local receiving buffer.
7. *MPI_Barrier/bsp_sync* – it synchronizes the processes.
8. *MPI_Bcast/bsp_bcast* – it 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 into account the similarity of several functions 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 into the core structure and a wrapper to keep up to the functions signature for each library was created. For instance:

Initialization and Finalization of processes. When a process starts the initialization method (Init and Begin – MPI and BSP respectively) 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 so as to allow the processes to find each other during the execution. Once a process executes the finalization procedure (*MPI_Finalize/bsp_end*) 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 called *Message* that encapsulates all parameters defined on both send methods. An object of this class is passed as a parameter to our send method

Send(Message m) that is responsible for sending the message to the remote process. The broadcast methods are a variant of the send method.

When the finalization method (Finalize and End – MPI and BSP, respectively) is called the process pid is removed from the manager's executor table and its listener of 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 *MPI_Barrier/bsp_sync* procedure, it checks whether all other executors signalling message have been received. If not, it locks itself, waiting for the signal of the other processes. When a process (a slave process) calls *MPI_Barrier/bsp_sync* the 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 super step). 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 [12]. 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. This runner is responsible for calling the GApplication that then creates input and output files and submit our DLL and the parallel application for execution. When the manager receives these data it defines on which machines the application will run (based on a schedule algorithm) and then sends 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. Figure 1 summarizes the process to execute the parallel applications into Alchemi Grid middleware. There are, basically, four steps:

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.

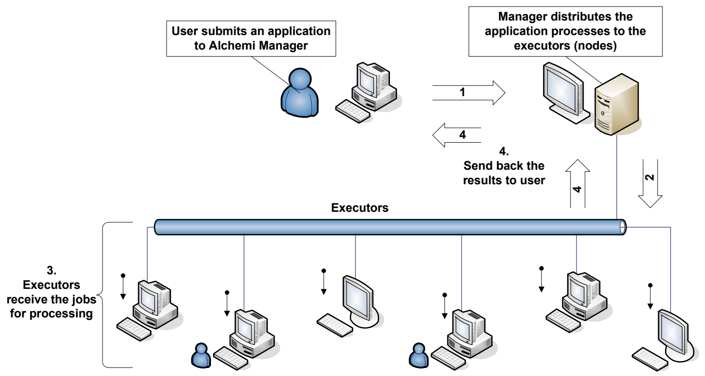
User submits an application to Alchemi Manager
Manager distributes the application processes to the executors (nodes)
1
4
4.
Send back the results to user
4
2
Executors
3.
Executors receive the jobs for processing

Figure 1: Flow diagram for executing message passing applications on Alchemi middleware in a Desktop Grid.

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 270 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 for then 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. The results are showed in Figures 3, 4 and 5 that represent the speedup for the matrices of size 1500x1500, 2000x2000, and 2500x2500 elements, respectively. In Figure 2 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 3 and 4 MPICH overcomes our BSP and MPI libraries. However, we can observe that the results regarding MPICH and MPI over Alchemi are considerably similar.

From the experimental results it is also possible to observe that MPI presented better results than BSP. This is mainly due to the synchronization procedure (synchronization barriers) of the BSP. It increases the idle time of some slave processes. In MPI, as said before, it is possible to send work to a slave process that starts the execution straight away without to wait the other slaves to receive the work. This behavior also repeats when the master process is collecting the results. In BSP, all the slave processes must send the results to the master and hence only after this the master can start storing the results. This has a considerable impact when several machines are used, as can be observed in the final of curves in Figures 2, 3 and 4. Also MPICH, in most cases, provided better results compared to our implementation.

We also measure the latency of sending messages between two processes through the "ping-pong" application. From Figure 5 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 due to the benefits of leveraging functionalities of a Grid middleware.

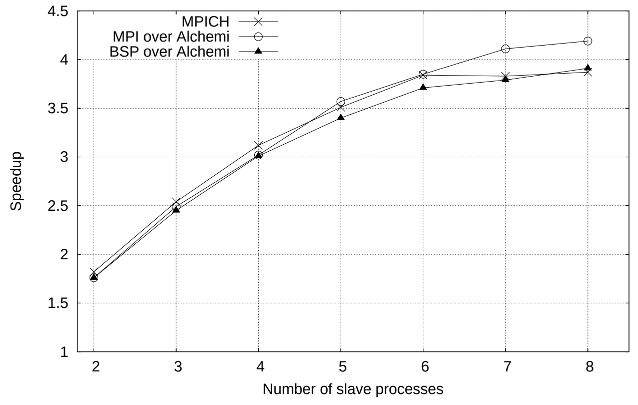

Figure 2: Speedup for Matrix with 1500x1500 elements.

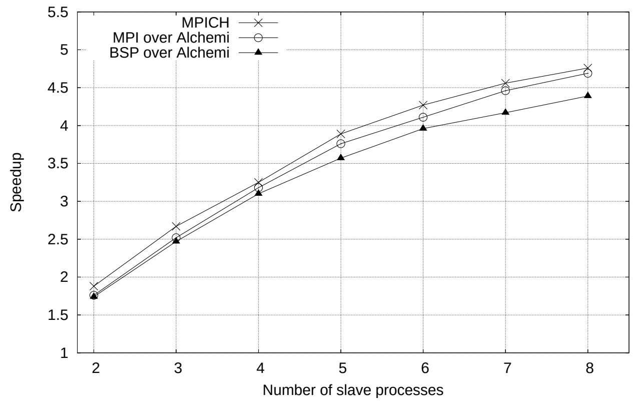

Figure 3: Speedup for Matrix with 2000x2000 elements.

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

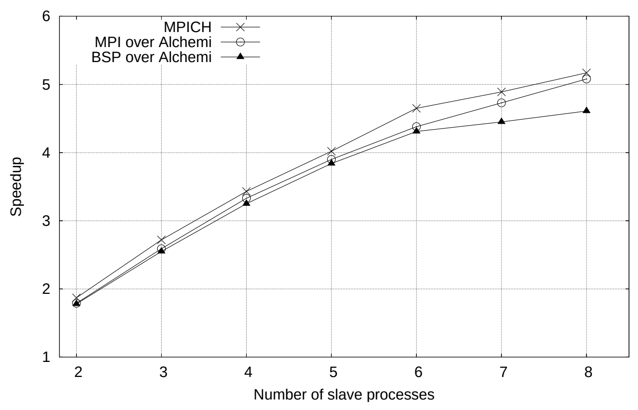

Figure 4: Speedup for Matrix with 2500x2500 elements.

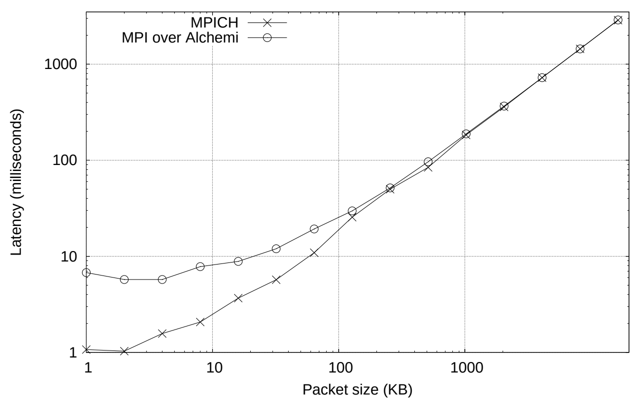

Figure 5: Results of the experiment on latency.

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 Windows-based Desktop Grid middleware. These two libraries, the Bulk Synchronous Parallel (BSP) computing model and the Message Passing Interface (MPI), present two different parallel processing styles. BSP brings the advantage of an easier implementation of checkpointing due to the mandatory synchronization barriers, the "supersteps", that is explored by the related projects, and that will be the focus of our future work. On the other hand, MPI is much more widely used in clusters of dedicated machines and also is more flexible, achieving then a better performance as demonstrated in our experiments.

In this paper 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 high level languages supported by the platform, such as C#, C++, J++, Visual Basic (VB), 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 (e.g. PUBWCL and InteGrade-BSP) 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 Alchemi's website - *http://www.alchemi.net*.

References

[1] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Pittsburgh, 2004.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] O. Bonorden, J. Gehweiler, and F. Meyer auf der Heide. A web computing environment for parallel algorithms in java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006.

[4] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, USA, 2002.

[5] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In *Proceedings of the 2nd workshop on Middleware for grid computing*, pages 35–40, New York, NY, USA, 2004. ACM Press.

[6] Distributed.net. http://www.distributed.net. August, 2006.

[7] FightAIDS@home. http://www.fightaidsathome.org. August, 2006.

[8] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, March 2004.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[10] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[11] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. Alchemi: A .net-based enterprise grid computing system. In *International Conference on Internet Computing*, pages 269–278, 2005.

[12] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. *Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework, High Performance Computing: Paradigm and Infrastructure*. Wiley Press, New York, USA, 2005.

[13] V. S. Pande, I. Baker, J. Chapman, S. Elmer, S. M. Larson, Y. M. Rhee, M. R. Shirts, C. D. Snow, E. J. Sorin, and B. Zagrovic. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Peter Kollman Memorial Issue, Biopolymers*, 68(1):91–109, 2003.

[14] L. F. G. Sarmenta and S. Hirano. Bayanihan: building and studying web-based volunteer computing systems using java. *Future Generation Computer Systems*, 15(5–6):675–686, 1999.

[15] W. Tong, J. Ding, and L. Cai. A parallel programming environment on grid. In *Proceedings of the International Conference on Computational Science*, volume 2657 of *Lecture Notes in Computer Science*, pages 225–234. Springer, 2003.

[16] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.