The Implementation of the BSP Parallel Computing Model
on the InteGrade Grid Middleware

Andrei Goldchleger, Alfredo Goldman, Ulisses Hayashida, Fabio Kon
Department of Computer Science
University of São Paulo, Brazil
{andgold,gold,ulisses,kon}@ime.usp.br
http://gsd.ime.usp.br/integrade

ABSTRACT
InteGrade is an object-oriented grid middleware infras-
tructure whose goal is to leverage existing computational re-
sources in organizations. Rather than relying on dedicated
hardware such as reserved clusters, InteGrade focuses on
using desktops in users’ offices, machines in computer labo-
raries, shared workstations, as well as dedicated clusters.
In this paper, we describe the support for the execution
of highly coupled parallel applications on top of InteGrade.
The paper describes the implementation of the middleware
to support BSP parallel applications (with global synchro-
nization points), and presents experimental results.

1. INTRODUCTION
InteGrade [6] is a Grid Computing system aimed at com-
modity workstations such as household PCs, corporate em-
ployee workstations, and PCs in shared laboratories. It uses
the idle computing power of these machines to perform use-
ful computation. Our goal is to allow organizations to use
their existing computing infrastructure to perform useful
computation, without requiring the purchase of additional
hardware. Moreover, users who share the idle portion of
their resources should have their quality of service preserved
by the InteGrade middleware.

In spite of the great computing power available today in
most organizations in the form of desktop PCs, there are still
difficulties in using the idle cycles of these machines for use-
ful computation. To solve this, we implemented support for
distributing and executing two different kinds of parallel ap-
lications. First, we extended the interface of InteGrade to
support parametric applications in which there is no commu-
nication among application nodes. This kind of application,

*This work is supported by a grant from CNPq, Brazil, pro-

included in the bag-of-tasks class, is currently supported by
other grid middleware such as OurGrid (www.ourgrid.org)
and BOINC [1], on non-dedicated machines. Second, we im-
plemented a modern parallel computing model (Bulk Syn-
chronous Parallel (BSP) [21, 16]) to support applications
whose nodes do communicate with each other, i.e., highly-
coupled parallel applications. The BSP reference implemen-
tation is University of Oxford’s BSPlib [19]. The BSPlib
core library is simple and is composed of only 20 functions.
When compared to PVM [18] and MPI [5], two popular par-
allel computing libraries, BSP offers a much more elegant
computing model and simpler programming library.

Within BSP, we have global synchronization points among
the processes of a parallel application. Using this synchro-
nization points the BSP applications can be better adapted
to an environment subject to frequent changes such as the
Grid. The BSP synchronization points greatly facilitates the
implementation of checkpointing to permit recovery in the
presence of failures, which are very common in Opportunis-
tic Grid Computing. Also, using checkpointing, the BSP
parallel applications can use a larger, or smaller number of
processors, expanding or shrinking dynamically, adapting to
the Grid resource availability.

In this paper, we discuss the implementation of the BSP
model on top of the InteGrade grid middleware, using its
distributed scheduling and allocation services. The struc-
ture of the paper is as follows. Section 2 discusses support
for parallel applications in other grid platforms and Section 3
describes the major concepts behind BSP and BSPlib. Sec-
tion 4 presents a brief description of the InteGrade system
and architecture. Section 5 focuses on our implementation
of the BSP model. We present our conclusions in Section 6.

2. RELATED WORK
Supporting parallel applications on heterogeneous envi-
nments, such as grid systems, is not trivial. Many issues
have to be addressed, such as communication overhead, fault
tolerance, parallel computing support, legacy compatibility,
checkpointing, job migration and synchronization, and so
forth.

Some grid systems already provide support for parallel ap-
lications. Grid systems such as Legion (www.cs.virginia.
eduard/legion} and Condor (www.cs.wisc.edu/condor) support the MPI and PVM parallel programming models.

Legion supports MPI and PVM parallel applications via emulation libraries that use Legion's run-time library. Existing applications only need to be recompiled and re-linked to run on Legion. Therefore, issues such as checkpointing and job migration are treated by emulation libraries.

Condor provides a framework for running PVM applications in its environment, the Condor-PVM. It does not define a new API, instead programs use the existing resource management PVM calls. Regular PVM and Condor-PVM are binary compatible. The same binary, which runs under regular PVM, also runs under Condor, and vice-versa. There is no need for re-linking for Condor-PVM, thus, application development is easier.

Condor supports MPI through MPICH. A problem is that machines running MPI jobs must be dedicated [22], which means that once they begin the execution of a program, they will continue executing the program until the program ends, which is a problem for environments where dedicated resources are not available.

Globus (http://www.globus.org), a toolkit that provides services for grid applications, supports MPI through MPICH-G2, a customized MPI implementation for grid applications. MPI applications can run under MPICH-G2 without changes. MPICH-G2 uses services provided by the Globus Toolkit to coordinate and manage work on multiple computer systems, automatically convert data in messages sent between machines of different architectures, and support multi-protocol communication. Recently, Globus also provided a BSP implementation, BSP-G [20]. Although the BSP model has the cleanest and simplest programming model, among the systems above, only for Globus there is an implementation.

To the best of our knowledge, the work described in this paper is the first implementation of BSP to run on an opportunistic grid middleware. Our BSP implementation is open-source and it benefits from support for checkpointing and security available in our middleware.

3. THE BSP COMPUTING MODEL

The Bulk Synchronous Parallel model (BSP) [21] was introduced by Leslie Valiant, as a bridging model, linking architecture and software. BSP offers both a powerful abstraction for computer architects and compiler writers, and a concise model of parallel program execution, enabling accurate performance prediction for proactive application design.

A BSP abstract computer consists of a collection of virtual processors, each with local memory, connected by an interconnection network whose only properties of interest are the time to do a barrier synchronization and the rate at which continuous randomly addressed data can be delivered. A BSP computation consists of a sequence of parallel supersteps, where each superstep is composed of computation and communication, followed by a barrier of synchronization.

The BSP model is compatible with the conventional SPMD / MPMD (single/multiple program, multiple data) model, and is at least as flexible as MPI, having both remote memory (DRMA) and message-passing (BSMP) capabilities. The timing of communication operations, however, is different since the effects of BSP communication operations do not become effective until the next superstep.

The postponing of communications to the end of a superstep is the key idea for implementations of the BSP model. It removes the need to support non-barrier synchronizations between processes and guarantees that processes within a superstep are mutually independent. This makes BSP easier to implement on different architectures and makes BSP programs easier to write and to analyze mathematically. For example, since the timing of BSP communications makes circular data dependencies between BSP processes impossible, there is no risk of deadlocks or livelocks in a BSP program. Also, the separation of the computation, communication, and synchronization phases allows one to compute time bounds and predict performance using relatively simple mathematical equations [16].

Moreover, there are plenty of algorithms developed for CGM (Coarse Grained Multicomputer Model) [4], which has the same principles of BSP and can be easily ported to BSP.

Several implementations of the BSP model have been developed since the initial proposal by Valiant. They provide to the users full control over communication and synchronization in their applications. Existing BSP implementations for local area networks include: Oxford's BSPlib [10] (1993), JBSP [9] (1999): a Java version, and PUB [2] (1999).

3.1 BSP and Grid Computing

Although not yet common, the use of the BSP model for Grid Computing on non dedicated resources fits very well with two fundamental characteristics of such environments: dynamism and heterogeneity. In both cases, the BSP model brings optimization opportunities, which are not straightforward in other models such as MPI.

The available resources in a Grid change frequently. Using the BSP model, it is possible to deal with this dynamism by using checkpointing in the synchronization points, avoiding the loss of computation when one or more machines being used by a BSP parallel application becomes unavailable. It is also possible to deal with resource availability fluctuations by shrinking or expanding the BSP parallel application, in the synchronization points [8]. This can be done, transparently to the application, by placing more than one of the BSP processes of an application in the same machine. That is, a BSP application with $n$ processes can be executed on $n$ to $n$ machines, where the maximum value for $k$ is determined considering primarily memory limitations.

The BSP model also helps with regard to the heterogeneity of processing speeds among Grid nodes. In a heterogeneous environment, the time of a superstep is determined by the slowest processor; thus, a processor allocation scheme where the processes with larger computing times go to the faster machines can be used. Finally, as the communications are done at the end of the supersteps, it is easier to find communication patterns and exploit this information to implement optimized Grid-aware scheduling in wide-area networks [7].
4. INTEGRADE ARCHITECTURE

InteGrade features an Object-Oriented architecture and is built using the CORBA [15] industry standard for distributed objects. InteGrade also strives to ensure that users who share the idle portions of their resources in the Grid shall not perceive any loss in the quality of service provided by their applications. To achieve this goal, the software that runs on resource providing workstations use OIL [3], a lightweight CORBA implementation. We are also working towards using a user level scheduler (DSRT) [14] to provide QoS guarantees for users of resource provider nodes.

The basic architectural unit of an InteGrade grid is the cluster. A cluster contains a number of machines, which typically varies from 1 to about 100. Clusters are naturally mapped to LANs, although this is not required. Clusters are then organized into a hierarchy which can potentially encompass millions of machines. This hierarchy can be organized in any convenient fashion, as there is no predetermined model. This overall architecture was proposed in the 2K Operating System [12] and was slightly modified to suit InteGrade’s needs.

Figure 1 depicts the major components in an InteGrade cluster. The Cluster Manager is composed of one or more nodes that are responsible for managing that cluster and communicating with managers in other clusters. A Grid User Node is one belonging to a user who submits applications to the Grid. A Resource Provider Node, typically a PC or a workstation in a shared laboratory, is one that exports part of its resources, making them available to grid users. A Dedicated Node is one reserved for grid computation. Note that these categories may overlap: for example, a node can be a Grid User Node and a Resource Provider Node at the same time.

The Local Resource Manager (LRM) and the Global Resource Manager (GRM) cooperatively handle intra-cluster resource management. The LRM is executed in each cluster node, collecting information about the node status, such as memory, CPU, disk, and network utilization. LRMs send this information periodically to the GRM, which uses it for scheduling within the cluster. This process is called the Information Update Protocol.

The GRM and LRMs also collaborate in the Resource Reservation and Execution Protocol, which works as follows. When a grid user submits an application for execution, the GRM selects candidate nodes for execution, based on resource availability and application requirements. For that end the GRM uses its local information about the cluster state as a hint for locating the best nodes to execute an application. After that, the GRM engages in a direct negotiation with the selected nodes to ensure that they actually have the sufficient resources to execute the application at that moment and, if possible, reserves the resources in the target nodes. In case the resources are not available in a certain node, the GRM selects another candidate node and repeats the process. The GRM is also responsible for communication with other clusters.

Similarly to the LRM/GRM cooperation, the Local Usage Pattern Analyzer (LUPA) and the Global Usage Pattern Analyzer (GUPA) handle intra-cluster usage pattern collection and analysis. The Node Control Center (NCC), which is still under construction, will allow the owners of resource providing machines to set the conditions for resource sharing, if they so wish. The Application Submission and Control Tool (ASCT) allows InteGrade users to submit grid applications for execution by using a graphical interface.

5. BSP OVER INTEGRADE

One of the objectives of the InteGrade BSP implementation is to allow existing applications written for the Oxford BSPLib to be executed over InteGrade with little or even no modifications. Thus, we strictly adhere to the API defined by the Oxford implementation targeted for the C language. The task of converting an existing BSPLib application to execute over InteGrade consists only of recompiling and relinking the application with the appropriate InteGrade libraries. This is a considerable advantage for programmers, since they will be able to execute existing applications over resources controlled by InteGrade without the cost of porting the applications.

Another important design decision was not to overload the core InteGrade interfaces with methods related to BSP. As InteGrade is a system under continuous development, we consider important to keep the core interfaces small, describing only the essential functionality. All BSP related methods, including the internals of our implementation, are kept in a separate module with its own IDL interfaces. For example, the scheduling system remains unchanged even with the addition of parallel applications.

Our BSP implementation uses CORBA internally for inter-task communication. CORBA gives us the advantages of an easier and cleaner communication environment, shortening development and maintenance time and facilitating system evolution. The use of CORBA is transparent to the user who only uses the traditional BSP interface.

Initially, we were worried that the use of CORBA for data exchange could bring a significant performance penalty when compared to an implementation based on raw sockets. But, experimental results demonstrated that the overhead imposed by CORBA was relatively small and the benefits in flexibility and ease of development showed that the choice of CORBA was correct. It is also important to note that CORBA’s IIOP is about 10 times faster than SOAP, the XML-based protocol widely used in Web Services.
Since the InteGrade project’s goal is to benefit from otherwise wasted computing resources, at the moment we are satisfied with the system’s performance. In the future, however, it would be possible to replace the use of CORBA with lower level mechanisms such as raw sockets; in this case, our experiments show that we could expect performance improvements in the order of 15%.

5.1 The Implementation
As mentioned before, the Oxford BSPlib has two means of inter-task communication. Direct Remote Memory Access (DRMA), which allows a task to read from and write to the remote address space of another task, and Bulk Synchronous Message Passing (BSMP), that implements message passing communication between tasks. We have currently implemented the most important functions DRMA and BSMP, the initialization routine (which is mandatory for all BSP programs), the barrier synchronization, and some simple enquiry methods. The following functions were implemented:

- bsp.begin
- bsp.pushregister
- bsp.popregister
- bsp.put
- bsp.get
- bsp.sync
- bsp.pid
- bsp.pprocs
- bsp.send
- bsp.move

In our implementation, each of the component tasks of a parallel application has an associated BspProxy. The BspProxy is a CORBA servant responsible for receiving BSP related communication for a given task. The proxy contains methods corresponding to functions defined in the BSP API, such as bsp.put, and also contains methods that are internal to our implementation. The creation of BspProxies is entirely handled by the library and is totally transparent to library users. The library also creates a StubPool, which is responsible for the instantiation of client stubs to access the proxies of other BSP tasks. As each of the tasks of a given application may communicate with all other tasks, the pool organization of these stubs allows us to save memory by sharing only one copy of OiL ORB.\(^1\)

BSP parallel applications need means to initialize the execution, spawn additional tasks, and manage synchronization barriers. In our implementation, the BSP parallel applications need coordination to perform some initialization tasks, such as attributing unique process identifiers to each of the application tasks, and broadcasting the IORs to each of the tasks to allow them to communicate directly among themselves. The synchronization barriers also requires central coordination. We decided to build those functionalities directly into the library: one of the application tasks, called Processor Zero, is responsible for performing the aforementioned tasks.

Parallel applications are registered in the same way as sequential ones. To execute a registered parallel application on the Grid, the user must use the ASCT graphical interface to send a request to the GRM. This request is identical to the one sent when executing a sequential application. The ASCT silently adds a configuration filename, bspExecution.conf, to the list of the application input files. This filename is not used by the GRM, which simply forwards it to the LRM which will host each of the parallel application processes. bspExecution.conf contains the number of application nodes, the application ID as attributed by the ASCT, and the IOR of the ASCT, which will be used to determine which task will be Processor Zero. When a request reaches the LRM, it downloads the configuration file from the ASCT.

The bsp.begin method determines the beginning of the parallel section of a BSP application. Applications are executed in the following way: when the method bsp.begin is reached, each launched task contacts the ASCT (with the call registerBspNode); the first one to complete the operation is elected Processor Zero. All other tasks receive Processor Zero reference. After receiving the reference, each task contacts Processor Zero sending its IOR (with the call registerRemoteIor). When Processor Zero receives all IORs, it sends to each task its processor identification (from 1 to the number of tasks minus 1), and broadcasts all the received IORs, to allow direct communication among tasks.

When bsp.begin is completed, each of the processes has a BSP PID and the IORs of all other processes, which are used to instantiate stubs for remote communication. The communication among tasks is performed through BspProxies and StubPools, as CORBA remote method invocations.

When the DRMA methods are used, before reading or writing a remote memory position (with bsp.get or bsp.put), it is necessary to register the position. The registration ensures that the physical memory addresses of a given variable, which are different on each task, are mapped to a logical address, which is the same across all tasks. This is done with the methods bsp.pushregister and bsp.popregister. The correspondence between the logical and physical addresses are stored in a stack in each task.

As previously described in Section 3, computation in the BSP model is composed of supersteps, and each of them is finished with a synchronization barrier. Operations such as bsp.put and bsp.pushregister only become effective at the end of the superstep. bsp.sync is the method responsible for establishing synchronization. In our implementation, it works as follows: when a task calls bsp.sync (including Process Zero), it sends a sync message to Process Zero and then stops executing. When Process Zero receives sync messages from all other processes, it broadcasts a sync done message to the other processes, which then can process all pending operations, in the following order: bsp.get; bsp.put; bsp.pushregister; bsp.popregister; and bsp.move.

5.2 Experiments
To evaluate the performance of our library, we implemented two simple applications. First, Multiple Matrix multiplications, where the algorithm used is based on the systolic approach [17]. Second, DNA sequence alignment, where the amount of communication among tasks is small; for a problem of size n, the computation is $O(n^2)$ and the communication is $O(n)$. We compared the performance of the algorithms on a local network of heterogeneous PCs, running the

\(^1\)OiL, our CORBA ORB, is written in Lua [11] and is loaded by the Lua runtime in the beginning of the application.
same algorithm written in MPI (using a highly-optimized implementation: MPI LAM 7.1.1 [13]) and in BSP over InteGrade. For these experiments we used only dedicated machines.

We carried out experiments for 1, 4, 9, and 16 computers. In the matrix multiplication experiment, the BSP CORBA implementation was surprisingly even faster than the MPI one for 4 and 9 computers (e.g., to multiply matrices of size 1500 by 1500, BSP took 1015.2s while MPI took 1180.5s). However, with 16 processors the MPI implementation was always faster (e.g., it solved the problem in half of the time of BSP for matrices of 600 size by 600). For the sequence alignment program, we obtained similar results, with MPI being a little faster than the BSP version. For this program, however, the difference in performance was at most 11% (the larger difference was for 10 computers with sequences of size 480,000 where BSP took 111.4s and MPI took 100.7s).

MPI performance was better in problems with smaller granularity and presented more stable speed-up. In some cases, the BSP results showed a tendency to loose performance with the increase in the number of machines. This shows that when one programs for this model it is important to pay good attention to the balance between computation and communication.

6. CONCLUSIONS

In this paper, we described the implementation of the support for BSP applications in the InteGrade middleware infrastructure for Grid Computing. Thanks to the object-oriented architecture of InteGrade and its use of an elegant and mature distributed object model (CORBA), the implementation of the extra functionality was relatively easy. We also verified that even if performance was not one of our main objectives it was possible to obtain some performance results close to the MPI implementation. So, the overhead added by the middleware and the CORBA communication were not so relevant.

InteGrade is available for download as open-source software from http://incubadora.fapesp.br/projects/integrade. Documentation and more information is available from the project main site (http://gsd.ime.usp.br/integrade). We would like to encourage researchers and software developers from other institutions both to use InteGrade in new applications and environments and to help extending the middleware, providing new functionalities.

7. REFERENCES


