Running Highly-Coupled Parallel Applications in a Computational Grid

Andrei Goldchleger*, Carlos Alexandre Queiroz , Fabio Kon , Alfredo Goldman

Department of Computer Science – University of São Paulo
{andgold,carlosq,kon,gold}@ime.usp.br
http://gsd.ime.usp.br/integrade

Abstract. *InteGrade is an object-oriented grid middleware infrastructure whose goal is to leverage existing computational resources in organizations. Rather than relying on dedicated hardware such as reserved clusters, InteGrade focuses on using user desktops, machines in instructional laboratories, shared workstations, as well as dedicated clusters.*
In this paper, we describe the recently added support for the execution of parallel applications on top of InteGrade. Using these new capabilities, it is possible to distribute a single application over a collection of InteGrade nodes distributed across local- and wide-area networks. The paper describes the implementation of the middleware supporting BSP parallel applications.

1. Introduction

InteGrade[1] [Goldchleger et al., 2004] is a Grid Computing system aimed at commodity workstations such as household PCs, corporate employee workstations, and PCs in shared laboratories. InteGrade uses the idle computing power of these machines to perform useful 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.

Before the work presented in this paper, InteGrade allowed for the distribution and execution of applications in Grid machines but it lacked explicit support for parallel applications. If a user wanted to execute a parallel application in which each part of the application ran on a different Grid machine, the only possibility was to program it manually, since no middleware support was available for distributing a single application across many nodes.

To solve this limitation, we implemented support for distributing and executing two different kinds of parallel applications. First, we extended InteGrade's interface to support parametric applications in which there is no communication among application nodes. Second, we implemented a modern parallel computing model (*Bulk Synchronous Parallel* (BSP) [Valiant, 1990]) to support applications whose nodes do communicate with each other. The BSP reference implementation is University of Oxford's BSPlib

*Andrei Goldchleger is partially supported by a graduate fellowship from CAPES, Brazil
[1]This work is supported by a grant from CNPq, Brazil, process #55.2028/02-9.

[Hill et al., 1998]. The BSPlib core library is simple and composed of only 20 functions. When compared to PVM [Sunderam, 1990] and MPI [MPI Forum, 1993], two popular parallel computing libraries, BSP offers a much more elegant computing model and simpler programming library.

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.

2. BSP over InteGrade

In order to ease application migration to the grid environment, 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 Oxford's implementation targeted for the C language[2]. The task of converting an existing BSPlib application to execute over InteGrade consists only in including a different header file, recompiling and re-linking the application with the appropriate InteGrade libraries.

Another important design decision was not to overload the core InteGrade interfaces with methods related to BSP. For example, the scheduling system remains unchanged even with the addition of parallel applications. It is the responsibility of the BSP library to arrange for application startup, and it does so by building over the existing scheduling system for regular applications. Although we consider this independency important, we recognize that this approach has its drawbacks. The impossibility of the scheduler to make gang scheduling decisions may lead to sub-optimal scheduling performance. However, we are trying to find a compromise solution between the two extremes.

Finally, our BSP implementation uses CORBA for inter-task communication. CORBA gives us the advantages of an easier and cleaner communication environment, shortening development and maintenance time. One could argue that CORBA's IIOP is far from being the ideal communication protocol for a parallel programming library. However, we remind that InteGrade benefits from otherwise wasted computing resources, and applications are executed on a highly dynamic environment, so raw performance is not one of our major objectives at the moment. Additionally, some experiments [Román et al., 2001] with compact ORBs show a slowdown in communications of only 15% when comparing CORBA to raw sockets. This means that using CORBA does not necessarily imply in poor communication performance.

2.1. The Implementation

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 all of DRMA, which already allows simple BSPlib applications to be executed. We have also implemented other library methods such as the initialization routine, which is mandatory

[2]There is also a Fortran implementation of the BSPlib. However, InteGrade currently does not support Fortran.

for all BSP programs, the barrier synchronization, and some simple enquiry methods. The complete list of implemented methods follows.

- `bsp_begin`: initializes a BSP application.
- `bsp_pushregister`: declares that a given memory address can be accessed by other tasks.
- `bsp_popregister`: makes a given memory area unavailable for remote access.
- `bsp_put`: writes on the memory of another task.
- `bsp_get`: fetches data from the memory of another task.
- `bsp_sync`: the synchronization barrier.
- `bsp_pid`: returns the BSP process ID of the calling task (local method).
- `bsp_nprocs`: returns the number of tasks of the parallel application.

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 methods 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 potentially communicate with all the other tasks, the pool organization of these stubs allows us to save memory by sharing only one copy of the O^2 library[3].

BSP parallel applications need a means to initialize the execution, spawn additional tasks, and manage synchronization barriers. In our implementation, these functionalities are built directly in the library, without requiring any additional services dedicated to parallel applications. The first process that compose a BSP application, from now on called *Process Zero*, is responsible for spawning the remaining tasks by engaging in negotiation with the Global Resource Manager (GRM) [4]. It is also responsible for allocating a PID to each of the remaining BSP tasks, and coordinate synchronization barriers.

Parallel applications are executed in the following way: users register applications in an application repository using the Application Submittion and Control Tool (ASCT). Parallel applications are registered in the same way as sequential ones. When a user wants to execute a registered parallel application, he uses the ASCT to make a request to the GRM. This request is identical to what is done when requesting the execution of a sequential application. He also specifies a configuration filename specific for parallel application execution. It is important to note that this filename is not used by the GRM in any way, it is simply forwarded to the Local Resource Managers (LRMs) which will host each of the tasks that compose the parallel application. When a request reaches a LRM, it downloads the configuration file from the ASCT.

The `bsp_begin` method determines the beginning of the parallel section of a BSP application. As we stated before, Process Zero is responsible for launching all remaining tasks, so it is essential that the library knows wether a given task is Process Zero or not.

[3] O^2 (`http://www.tecgraf.puc-rio.br/luaorb/o2`), our CORBA ORB, is written in Lua and it is loaded by the Lua runtime in the beginning of the application.

[4] The InteGrade architecture and modules are described in detail in [Goldchleger et al., 2004]

This information is obtained from the configuration file, which also holds the number of tasks that must be spawned. During `bsp_begin`, the configuration file is read. If the process is the Zero, it is responsible for spawning the remaining application tasks. Otherwise the process is just a plain task: it instantiates its servant, sends a registration message to Process Zero, which is reachable by the IOR contained in the configuration file, and waits until it receives a PID and, subsequently, the IOR of all other processes.

At the end of `bsp_begin`, 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 between tasks are made through BspProxies and StubPools, as CORBA remote method invocations.

Computation in the BSP model is composed of supersteps, where each superstep is composed of computation and communication, followed by synchronization barrier. Operations such as `bsp_put` and `bsp_pushregister` only become effective at the end of the superstep. `bsp_synch` is the method responsible for establishing synchronization. In our implementation, it works as follows: when each task calls `bsp_synch` (including Process Zero), it sends a synch message to Process Zero and then stops executing. When Process Zero receives synch messages from all other processes, it broadcasts a `synch_done` message to the other processes, which then can process all pending operations, such as `bsp_put`, `bsp_pushregister`, etc...

3. Conclusions

In this paper, we described the implementation of the support for parallel 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 to add.

References

Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C. (2004). InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. Accepted for publication in Concurrency and Computation: Practice & Experience, Volume 16.

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

MPI Forum (1993). MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*.

Román, M., Kon, F., and Campbell, R. (2001). Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5).

Sunderam, V. S. (1990). PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340.

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