Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware*

Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon, and Alfredo Goldman
Dept. of Computer Science
University of São Paulo
São Paulo, SP, Brazil
rcamargo,andgold,kon,gold@ime.usp.br

ABSTRACT

InteGrade is a grid middleware infrastructure that enables the use of idle computing power from user workstations. One of its goals is to support the execution of long-running parallel applications that present a considerable amount of communication among application nodes. However, in an environment composed of shared user workstations spread across many different LANs, machines may fail, become unaccessible, or may switch from idle to busy very rapidly, compromising the execution of the parallel application in some of its nodes. Thus, to provide some mechanism for fault-tolerance becomes a major requirement for such a system.

In this paper, we describe the support for checkpoint-based rollback recovery of parallel BSP applications running over the InteGrade middleware. This mechanism consists of periodically saving application state to permit to restart its execution from an intermediate execution point in case of failure. A precompiler automatically instruments the source-code of a C/C++ application, adding code for saving and recovering application state. A failure detector monitors the application execution. In case of failure, the application is restarted from the last saved global checkpoint.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Distributed Systems—*distributed applications*; D.1.3 [**Programming Techniques**]: concurrent programming—*parallel programming*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart, fault-tolerance*

General Terms

Languages, Performance, Reliability

*This work is supported by a grant from CNPq, Brazil, process #55.2028/02-9.

2nd Workshop on Middleware for Grid Computing Toronto, Canada

Keywords

Fault-tolerance, Checkpointing, BSP, Grid Computing

1. INTRODUCTION

Grid Computing [7] represents a new trend in distributed computing. It allows leveraging and integrating computers distributed across LANs and WANs to increase the amount of available computing power, provide ubiquitous access to remote resources, and act as a wide-area, distributed storage. Grid computing technologies are being adopted by research groups on a wide variety of scientific fields, such as biology, physics, astronomy, and economics.

InteGrade [8, 11] is a Grid Computing system aimed at commodity workstations such as household PCs, corporate employee workstations, and PCs in shared university laboratories. InteGrade uses the idle computing power of these machines to perform useful computation. The goal is to allow organizations to use their existing computing infrastructure to perform useful computation, without requiring the purchase of additional hardware.

Running scientific applications over shared workstations requires a sophisticated software infrastructure. Users who share the idle portion of their resources should have their quality of service preserved. If an application process was running on an previously idle machine whose resources are requested back by its owner, the process should stop its execution immediately to preserve the local user's quality of service. In the case of a non-trivial parallel application consisting of many processes, stopping a single process usually requires the reinitialization of the whole application. Mechanisms such as checkpoint-based rollback recovery [5] can be implemented in order to solve this kind of problem.

We implemented a checkpoint-based rollback recovery mechanism for single process and *Bulk Synchronous Parallel* (BSP) [20] applications running over the InteGrade grid middleware. We provide a precompiler that instruments the application source-code to save and recover its state automatically. We also implemented the runtime libraries necessary for the generation of checkpoints, monitoring application execution and node failures, and coordination among processes in BSP applications [1].

The structure of the paper is as follows. Section 2 describes the major concepts behind the BSP model and the

[1]Actually, only the monitoring and reinitialization parts of the code are specific for InteGrade. Consequently, this mechanism can be easily ported to other systems.

checkpointing of BSP applications. Section 3 presents a brief description of the InteGrade middleware and its architecture, while Section 4 focuses on the implementation of the checkpoint-based recovery mechanism. Section 5 shows results from experiments performed with the checkpointing library. Section 6 presents related work on checkpointing of parallel applications. We present our conclusions and discuss future work in Section 7.

2. CHECKPOINTING OF BSP APPLICATIONS

In this section we present a brief introduction to the BSP computing model and our approach for checkpointing BSP applications.

2.1 The BSP Computing Model

The *Bulk Synchronous Parallel* model [20] was introduced by Les Valiant, as a bridging model, linking architecture and software. 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.

An advantage of BSP over other approaches to architecture-independent programming, such as the message passing libraries PVM [19] or MPI, lies in the simplicity of its interface, as there are only 20 basic functions. Another advantage is the predictability of performance. The performance of a BSP computer is analyzed by assuming that in one time unit an operation can be computed by a processor on the data available in local memory, and based on the three parameters: the number of virtual processors (P), the ratio of communication throughput to processor throughput (G) and the time required to barrier synchronize all processors (L).

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. The mapping of virtual BSP processors to physical processors is hidden from the user, no matter what the real machine architecture is. These implementations include the Oxford's BSPlib [10], and PUB [2].

2.2 Application-Level Checkpointing

Application-level checkpointing consists in instrumenting an application source-code in order to save its state periodically, thus allowing recovering after a fail-stop failure [3, 18, 12]. It contrasts with traditional system-level checkpointing where the data is saved directly from the process virtual memory space by a separate process or thread [17, 15].

The main advantage of the application-level approach is that semantic information about memory contents is available when saving and recovering checkpoint data. Using this approach, only the data necessary to recover the application state needs to be saved. Also, the semantic information permits the generation of portable checkpoints [18, 12], which is an important advantage for applications running in a grid composed of heterogeneous machines. The main drawback is that manually inserting code to save and recover an application state is a very error prone process. This problem can be solved by providing a precompiler which automatically inserts the required code. Other drawbacks of this approach are the need to have access to the application source-code and the impossibility of generating forced checkpoints[2].

2.3 Checkpoint Coordination

When checkpointing parallel and distributed applications, we have an additional problem regarding the dependencies between the application processes. This dependency is generated by the temporal ordering of events during process execution. For example, process A generates a new checkpoint c_1 and then sends a message m_1 to process B. After receiving the message, process B generates checkpoint c_2. Lets denote this message sending event as $send(m_1)$ and the receiving of the message as $receive(m_1)$. Here there is a relation of causal precedence between the $send(m_1)$ and $receive(m_1)$ events, meaning that the $receive(m_1)$ event must necessarily happen after the $send(m_1)$. The state formed by the set of checkpoints $\{c_1, c_2\}$ is inconsistent, since it violates this causal precedence relation.

A global checkpoint is a set containing one checkpoint from each of the application processes and it is consistent if the global state formed by these checkpoints does not violate any causal precedence relation. If processes generate checkpoints independently, the set containing the last generated checkpoint from each process may not constitute a consistent global checkpoint. In the worst case scenario, it can happen that, after a failure, no set of checkpoints form a consistent state, requiring the application to restart its execution from its initial state. This problem is usually referred as domino effect.

There are different approaches to prevent the domino effect [5]. The first one, called communication-induced checkpointing, forces the processes to generate extra checkpoints in order to prevent some types of dependencies among processes. The main problem with this approach is that the number of forced checkpoints is dependent on the number of messages exchanged, what can result in a large number of extra checkpoints in some cases. Also, it requires sophisticated algorithms for global checkpointing construction and collection of obsolete checkpoints. Another possibility is to use non-coordinated checkpointing with message logging [5].

Coordinated checkpointing protocols guarantee the consistency of global checkpoints by synchronizing the processes before generating a new checkpoint. Since the newly generated global checkpoint is always consistent, there is no need to implement a separate algorithm for finding this global checkpoint. Also, garbage collection is trivial, since all checkpoints except the last one are obsolete. This is the natural choice for BSP applications since BSP already requires a synchronization phase after each superstep.

3. INTEGRADE ARCHITECTURE

The InteGrade project is a multi-university effort to build a novel Grid Computing middleware infrastructure to lever-

[2]In application-level checkpointing, the process state can only be saved when checkpoint generation code is reached during execution. In system-level checkpointing, since the state is obtained directly from the main memory by a separate thread or process, it can be saved at any moment

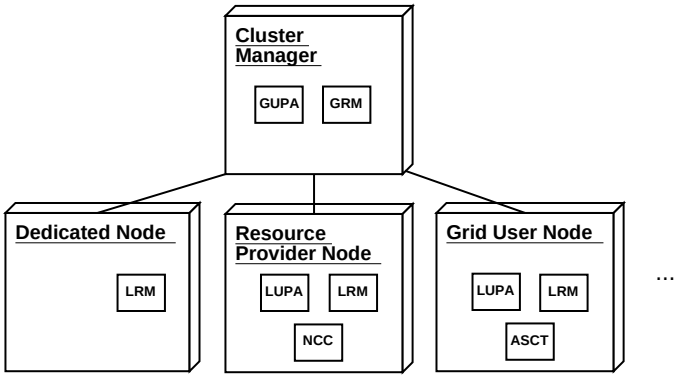

Figure 1: InteGrade's Intra-Cluster Architecture

age the idle computing power of personal workstations. InteGrade features an object-oriented architecture and is built using the CORBA [16] 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.

The basic architectural unit of an InteGrade grid is the cluster, a collection of 1 to 100 machines connected by a local network. Clusters are then organized in a hierarchical intercluster architecture, which can potentially encompass millions of machines.

Figure 1 depicts the most important kinds of components in an InteGrade cluster. The *Cluster Manager* node represents 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. This kind of node is shown to stress that, if desired, InteGrade can also encompass dedicated resources. Note that these categories may overlap: for example, a node can be both a *Grid User Node* and a *Resource Provider Node*.

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*.

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 LUPA executes in each node that is a user workstation and collects data about its usage patterns. Based on long series of data, it derives usage patterns for that node throughout the week. This information is made available to the GRM through the GUPA, and allows better scheduling decisions due to the possibility of predicting a node's idle periods based on its usage patterns.

The *Node Control Center* **(NCC)**, allows the owners of resource providing machines to set the conditions for resource sharing. The *Application Submission and Control Tool* **(ASCT)** allows InteGrade users to submit grid applications for execution.

4. IMPLEMENTATION

We have implemented a checkpoint-based rollback recovery system for BSP applications running over the InteGrade middleware. In this section we present our BSP implementation, our precompiler for inserting checkpointing code into an application source-code, and the runtime libraries.

4.1 The BSP Implementation

The InteGrade BSP implementation [9] allows C/C++ applications written for the Oxford BSPlib to be executed on the InteGrade grid, requiring only recompilation and relinking with the InteGrade BSP library. Our implementation currently supports interprocess communication based on *Direct Remote Memory Access* (DRMA), which allows a task to read from and write to the remote address space of another task. Message passing support is currently being implemented.

The `bsp_begin` method determines the beginning of the parallel section of a BSP application. As previously described in section 2.1, computation in the BSP model is composed of supersteps, and each of them is finished with a synchronization barrier. Operations such as `bsp_put` (a remote write on another process' memory) and `bsp_pushregister` (registration of a memory address so that it can be remotely read/written) only become effective at the end of the superstep. `bsp_synch` is the method responsible for establishing synchronization in the end of each superstep.

4.2 The Checkpoint Precompiler

The precompiler implementation uses OpenC++ [4], an open source tool for metacomputing which also works as a C/C++ source-to-source compiler. It automatically generates an abstract syntactic tree (AST) that can be analyzed and modified before generating C/C++ code again. Using this tool saved us from implementing the lexer and parser for C/C++.

4.2.1 Saving the Execution Stack

The execution stack contains runtime data from the active functions in a particular moment during program execution. It includes the local variables and parameters values, the return address, and some extra control information for each active function. This execution stack is not directly accessible from application code. Consequently, the stack state must be saved and reconstructed indirectly.

A solution for reconstructing the stack state is to call the functions that were active during the last checkpoint in the same order as before, declaring the local variables for each of these functions. The values of these variables are then recovered from the last generated checkpoint.

To accomplish this reconstruction, the precompiler modifies the functions from the source program so that the current active functions and the values from the local variables are saved during normal execution. This data is then used during recovery. Only a subset of the functions need to be modified. This subset includes the functions that can possibly be in the execution stack during checkpoint generation. Let us denote by ϕ the set of functions that needs to be modified. A function $f \in \phi$ if, and only if, f calls a checkpointing

function[3] or, f calls a function $g \in \phi$. To determine which functions need to be modified, the precompiler initially adds functions that call a checkpointing function. Then it recursively inserts into ϕ all functions that call functions in ϕ, until no more functions are added.

In order to save the local variables, we use an auxiliary stack where we keep the addresses of all local variables currently in scope. A variable enters scope when it is declared and leaves scope when execution exits from the block where the variable was declared. Execution can exit a block by reaching its end or by executing `return`, `break`, `continue` or `goto` statements. When a checkpoint is generated, the values contained at the addresses from this auxiliary stack are saved to the checkpoint. To keep track of the function activation hierarchy, the precompiler adds a new local variable `currentGotoLabel` into all functions in ϕ. The value of this variable is then modified before calling any function from ϕ. Global variables are added to the stack by the `main` function before any other local variable.

Saving structures is similar to local variables. It is only necessary to stack the structure address and size. When the structure contains pointers, these must be stacked separately. In the case of classes, the precompiler adds methods for saving and restoring the class fields.

For application reinitialization, it is necessary to execute all the function calls and variable declarations until reaching the checkpoint generation point. This requires that for each function of ϕ, the precompiler determines all variables that are in the scope of each call to functions of ϕ. The remaining code is skipped, otherwise the application would be executing unnecessary code[4]. After reaching the checkpoint generation point, the execution continues normally.

Below we present a C function modified by our precompiler. Local variable `currentGotoLabel` is added by the precompiler to record the currently active functions, while `mainInt` represents a local variable from the unmodified function. Global variable `ckp_recovering` indicates the current execution mode, that can be normal or recovering. In this example we see the local variables being pushed and popped from the auxiliary stack, and the data recovering and code skipping that occurs during recovery.

```
int cfunction () {
  int currentGotoLabel = -1;
  int mainInt = 0 ;
  ckp_push_data(&currentGotoLabel, sizeof(int));
  ckp_push_data(&mainInt, sizeof(int));
  if (ckp_recovering==1) {
    ckp_get_data(&currentGotoLabel, sizeof(int));
    ckp_get_data(&mainInt, sizeof(int));
    if(currentGotoLabel == 0)
      goto ckp0;
    if(currentGotoLabel == 1)
      goto ckp1;
  }
  // Do computations
  (...)
 ckp0:
  currentGotoLabel = 0;
  function0 ( ) ;
  // Do computations
  (...)
 ckp1:
  currentGotoLabel = 2;
  function1 ( ) ;
  // Do computations
  (...)
  ckp_npop_data(2);
}
```

[3]Here, we denote checkpointing functions as the functions responsible for saving application state into a checkpoint.

[4]An exception occurs in the case of BSP applications, where some function calls to the library must be re-executed in order to recover the library state.

4.2.2 Saving the Heap State

In addition to saving the value of local variables, it is also necessary to save the contents of memory allocated from the heap. To keep track of the allocated memory, we implemented a heap manager. It keeps information about the allocated memory chunks, their respective sizes, and some control information. During checkpoint generation this information is used to detect cycles in pointer graph structures and to prevent duplication of data in the checkpoints. The precompiler redirects memory allocation system calls – `malloc`, `realloc`, and `free` – in the application source-code to equivalent functions in our checkpointing runtime library. These functions update our memory manager and then make the regular allocation system call.

4.2.3 Calls to the BSP API

In the case of BSP applications, the precompiler has to do some extra tasks. Function calls to `bsp_begin` and `bsp_synch` are substituted by equivalent functions in the our runtime library. Function `bsp_begin_ckp` registers some BSP memory addresses necessary for checkpoint coordination and initializes the BSPLib and checkpointing timer. Function `bsp_sync_ckp` is responsible for checkpoint generation coordination. When called from Process Zero, it checks whether the minimum checkpointing interval has expired. If it did, it signals all other processes to make their checkpoints, issues the `bsp_sync` call and returns true. Otherwise, it just issues the `bsp_sync` call and returns false.

Depending on the response from the `bsp_sync_ckp` call, the process generates a new checkpoint. Values from the addresses registered in the BSP library can be ignored since the checkpoint is generated immediately after a `bsp_sync` step.

During reinitialization, calls to functions that modify the state from the BSPLib, such as `bsp_begin` and `bsp_pushregister`, must be executed again. This is necessary to recover the internal state from the BSPLib. The other solution would be to save the internal state from the BSPLib, but this would save unnecessary information.

4.3 Runtime Libraries

The runtime libraries provide basic functionality for checkpointing of single processes, and specific functionality for checkpointing BSP applications. They provide a C API that allows applications written in both C and C++ to use them.

The basic checkpointing functionality is provided by functions to manipulate the checkpoint stack, to save the stack data to a file, and to recover checkpointing data. They also allow the specification of a minimum checkpointing interval.

Checkpoint data is saved directly from the addresses at the checkpointing stack. It currently saves the data in an

archive in the file system. This can be a problem in the case where the machine where the process was executing becomes unavailable. But when using a network filesystem such as NFS, this solution is enough. We are planning a system for saving checkpoints remotely in a distributed way. Another current restriction is that the saved data is architecture dependent. This dependency arises due to differences in data representation and memory alignment. Making the the checkpoint portable requires saving data in a plataform independent format.

The BSP specific functionality is provided by the functions `bsp_begin_ckp` and `bsp_sync_ckp`. They do the initialization and coordination of the checkpointing process. They also manage obsolete checkpoints, which, in the case of coordinated checkpointing, is trivial. Since this protocol always generates consistent global checkpoints, it is only necessary to keep a global checkpoint only until a new one is generated.

The library also implements a failure detection system using a heartbeat scheme. Each process monitors the process with the PID immediately below it, except Process Zero, which monitors the process with the highest PID. When a process stops sending updates for a given amount of time, the process monitoring it interprets that as a failure and starts the process reinitialization coordination. Here, precautions must be taken not to restart processes which terminated due to problems such as segmentation fault or that completed the execution normally.

In order to allow the processes to agree about the reinitialization coordinator, we used a two-phase commit protocol. This is necessary in the case where two processes detect failures simultaneously and start the reinitialization process. When all processes reach an agreement about the new coordinator, all processes terminate their execution and restart from the last global checkpoint.

5. EXPERIMENTS

The experiments were performed using a sequence similarity application [1]. It compares two sequences of characters and find the similarity among them using a given criterion. For a pair of sequences of size m and n, the application requires $O((m+n)/p)$ memory, $O(m)$ communication rounds and $O(m*n)$ computational steps.

The experiments were performed on a grid containing 10 1.4GHz machines connected by a 100 Mbps Fast Ethernet network. We used 5 pairs of sequences of size 100k as input. We evaluated the overhead caused by checkpoint generation for minimum intervals between checkpoints of 10, 60, and 600 seconds. The last case generates no checkpoint, so it is used to measure the overhead caused by the additional checkpointing code when no checkpoints are performed. We perform five experiments for each checkpointing interval. The results are presented in Table 1.

The versions with and without checkpointing code runs in roughly the same time, showing that the time consumed by the extra code is very small. When using a minimum checkpoint interval of 1 minute, the overhead is only 2%. This is a reasonable interval to use in the dynamic environment where grids are typically deployed and with parallel applications that may take up to several hours to run. Even in the case of 10-second minimum intervals (which is actually too small, generating checkpoints too frequently) the overhead was bellow 10%.

Checkpointing of applications containing large amounts of data, such as image processing, will cause bigger overheads than the ones measured in our example. In these cases, longer intervals between checkpoints can be used to reduce this overhead. For an applications that runs for hours, losing some minutes of computation is normally not a problem.

t_{min}	n_{ckp}	t_{total}	t_{orig}	ovh
$600s$	0	$339.7s$	$339.9s$	0%
$60s$	5	$347.1s$	$339.9s$	2.1%
$10s$	23	$371.9s$	$339.9s$	9.4%

Table 1: Checkpointing overhead for the sequence similarity application. Generated checkpoints for each process are of size 125k bytes. t_{min} is the minimum interval between checkpoints, n_{ckp} is the number of generated checkpoints, t_{total} is the execution time of the modified code, t_{orig} is the execution time without checkpointing code, and ovh is the relative overhead introduced by checkpointing.

6. RELATED WORK

The Oxford BSPLib provides a transparent checkpointing mechanism for fault-tolerance. It employs system-level checkpointing, so it only works on homogeneous clusters. Application-level checkpointing for MPI applications is presented in [3]. They present a coordinated protocol for application-level checkpointing. They also provide a precompiler that modifies the source-code of C applications.

Recently, some research in the area of fault-tolerance for parallel applications on grids has also been published. The MPICH-GF [21] provides user-transparent checkpointing for MPI applications running over the Globus [6] Grid middleware. The solution employs system-level checkpointing, and a coordinated checkpointing protocol is used to synchronize the application processes.

A checkpointing mechanism for PVM applications running over Condor [14] is presented in [13]. It also uses system-level checkpointing and a coordinated protocol. In this solution, checkpoint data is saved in a separate checkpointing server. There is also a separate module to perform the checkpointing and reinitialization coordination.

An important difference in our approach is the use of application-level checkpointing. It will allow the generation of portable checkpoints, which is an important requirement for heterogeneous grid environments. Also, checkpoints generated are usually smaller than when using a system-level checkpointing approach. Another difference is that our implementation supports the BSP parallel programming model.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we described the implementation of checkpoint-based rollback recovery for BSP parallel applications running over the InteGrade middleware. Application-level checkpointing gives us more flexibility, for example to add support for portability in the near future. A fault-tolerance mechanism is of great importance for the dynamic and heterogeneous environments where the InteGrade middleware operates. It permits execution progression for single process and BSP parallel applications even in the presence of partial or complete execution failures, such as when grid machines

(e.g., user desktops) are reclaimed by their owners. Preliminary experimental results indicates that checkpointing overhead is low enough to be used on applications which needs more then a few minutes to complete its execution.

The current implementation of the precompiler has limited C++ support. Features such as inheritance, templates, STL containers and references still need to be implemented. Support for these features will be implemented in a future version.

Our next step is to support portable checkpoints. In an heterogeneous environment, such as a Grid, portable checkpoints will allow better resource utilization. Another necessity is the development of a more robust storage system for checkpoints. Data will be stored in a distributed way, with some degree of replication to provide better fault-tolerance. Once these features are implemented we will then be able to provide an efficient process migration mechanism for both fault-tolerance and dynamic adaptation in the InteGrade grid middleware.

InteGrade is available as free software an can be obtained from the InteGrade project main site[5]. Current versions of the precompiler and checkpointing runtime libraries are available at the checkpointing subproject page[6].

Acknowledgements

Ulisses Hayashida provided the sequence similarity application used in our experiments. José de Ribamar Braga Pinheiro Júnior helped us to solve several network configuration issues in InteGrade.

8. REFERENCES

[1] Alves, C. E. R., Cáceres, E. N., Dehne, F., and W, S. S. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. In *The 2003 International Conference on Computational Science and its Applications* (May 2003), Springer-Verlag, pp. 249–258.

[2] Bonorden, O., Juulink, B., von Otto, I., and Rieping, I. The Paderborn University BSP (PUB) Library—Design, Implementation and Performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing* (1999).

[3] Bronevetsky, G., Marques, D., Pingali, K., and Stodghill, P. Automated application-level checkpointing of mpi programs. In *Proceedings of the 9th ACM SIGPLAN PPoPP* (San Diego, USA, 2003), pp. 84–89.

[4] Chiba, S. Compiler-assisted heterogeneous checkpointing. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (October 1995), pp. 285–299.

[5] Elnozahy, M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (May 2002), 375–408.

[6] Foster, I., and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications 2*, 11 (1997), 115–128.

[7] Foster, I., and Kesselman, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, 1999.

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

[9] Goldchleger, A., Queiroz, C. A., Kon, F., and Goldman, A. Running Highly-Coupled Parallel Applications in a Computational Grid. In *Proceedings of the 22th Brazilian Symposium on Computer Networks (SBRC'2004)* (Gramado-RS, Brazil, May 2004).

[10] 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. BSPlib: The BSP programming library. *Parallel Computing 24*, 14 (1998), 1947–1980.

[11] Integrade. http://gsd.ime.usp.br/integrade, 2004.

[12] Karablieh, F., Bazzi, R. A., and Hicks, M. Compiler-assisted heterogeneous checkpointing. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems* (New Orleans, USA, 2001), pp. 56–65.

[13] Kovács, J., and Kacsuk, P. A Migration Framework for Executing Parallel Programs in the Grid. In *2nd European Accross Grids Conference* (Nicosia, Cyprus, January 2004).

[14] Litzkow, M., Livny, M., and Mutka, M. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems* (June 1988), pp. 104–111.

[15] Litzkow, M., Tannenbaum, T., Basney, J., and Livny, M. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Tech. Rep. UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[16] Object Management Group. *CORBA v3.0 Specification*, July 2002. OMG Document 02-06-33.

[17] Plank, J. S., amd G. Kingsley, M. B., and Li, K. Libckpt: Transparent chieckpointing under unix. In *Proceedings of the USENIX Winter 1995 Technical Conference* (1995), pp. 213–323.

[18] Strumpen, V., and Ramkumar, B. Portable checkpointing and recovery in heterogeneous environments. Tech. Rep. UI-ECE TR-96.6.1, University of Iowa, June 1996.

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

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

[21] Woo, N., Choi, S., Jung, H., Moon, J., Yeom, H. Y., Park, T., and Park, H. MPICH-GF: Providing Fault Tolerance on Grid Environments. In *The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003)* (Tokyo, Japan, May 2003).

[5] `http://gsd.ime.usp.br/integrade`

[6] `http://gsd.ime.usp.br/integrade/checkpointing`.