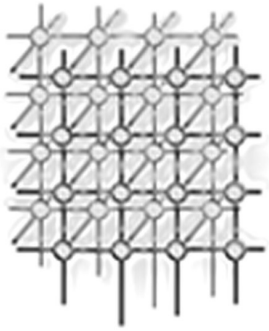
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–10 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Checkpointing-based rollback recovery for parallel applications on the InteGrade Grid middleware

Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon and Alfredo Goldman
{rcamargo,andgold,kon,gold}@ime.usp.br

Dept. of Computer Science - University of São Paulo
Rua do Matão, 1010. 05508-090, São Paulo - SP, Brazil

SUMMARY

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.

KEY WORDS: 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 both sequential applications and parallel applications written in *Bulk Synchronous Parallel* (BSP) [21] 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 [*].

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

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

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. The BSP Computing Model

The *Bulk Synchronous Parallel* model [21] was introduced by Leslie 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 performance predictability. 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 three parameters: the number of virtual processors (P), the ratio of communication throughput to processor throughput (G), 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], PUB [2], and the Globus implementation, BSP-G [20].

2.3. Protocols for Checkpointing of Parallel Applications

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.

[†]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

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 forms a consistent state, requiring the application to restart its execution from its initial state. This problem is usually referred to 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, possibly resulting in a large number of extra checkpoints. 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 leverage 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

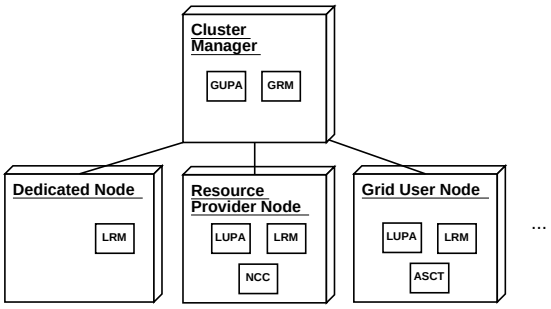


Figure 1. InteGrade's Intra-Cluster Architecture

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, the checkpointing libraries and the infrastructure for monitoring and recovering application execution.

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) and *Bulk Synchronous Message Passing*, 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 2.2, 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_sync` is the method responsible for establishing synchronization in the end of each superstep.

BSP parallel applications needs coordination in order 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 in order to allow them to communicate directly. 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.

4.2. Saving and Recovering the Application State

In order to generate a checkpoint from the application state, it is necessary to save the execution stack, the heap area and the global variables. The precompiler modifies the application code so that it interacts with a checkpointing runtime library in order to save and recover the application state.

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++.

The current implementation of the precompiler has limited C++ support. Features such as inheritance, templates, STL containers and C++ references will be implemented in future versions. Also, the precompiler does not support the C constructs longjmp and setjmp.

4.2.1. Saving the Execution Stack

The execution stack contains runtime data from the active functions during program execution, including local variables, function parameters, return address, and some extra control information. This execution stack is not directly accessible from application code. Consequently, the stack state must be saved and reconstructed indirectly.

The concept for doing this is simple. During checkpoint generation we save the list of active function and the values of their parameters and local variables. When recovering, we call the functions in this list, declare the local variables and skip the remaining code. The values of the local variables and function parameters are initialized from the values in the checkpoint. If we define that checkpoints will be generated only in certain points during execution, for example when calling a function Θ, we know exactly the location of the program counter when generating the checkpoint. The execution after recovery can then continue from that point.

To accomplish this reconstruction, the precompiler modifies the functions from the source program, instrumenting them with checkpointing code. 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[‡] 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.

To keep track of the function activation hierarchy, an auxiliary local variable `lastFunctionCalled` is added to each function f in ϕ. This variable is updated before calling functions from ϕ. Using this value, it is possible to determine which function f was calling during checkpoint generation. This permits that we determine all the function in the execution stack during checkpoint generation.

In order to save the local variables, we use an auxiliary stack which keeps 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.

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 alive before each call to functions of ϕ. The remaining code is skipped, otherwise the application would be executing unnecessary code[§]. After reaching the checkpoint generation point, the execution continues normally.

In Figure 2 we present a C function instrumented by our precompiler, where the added code is represented by bold text. The local variable `lastFunctionCalled` is added by the precompiler to record the currently active functions, while `localVar` represents a local variable from the unmodified function. Global variable `ckpRecovering` 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.

4.2.2. Pointers

During checkpoint generation, it is necessary to save the addresses referenced by pointers in a way that allows its recovery. Pointers can point to locations in the execution stack and in the heap area.

In the case of data in the heap area, saving the memory address directly is not an option. When the application is restarted, memory allocated in the heap will probably be in different addresses. A solution to this problem is to save the data from the heap in the checkpoint and insert in the pointer value the position of that data in the checkpoint. During recovery, when memory is allocated for that data, the pointer is then initialized with the address of the memory allocated.

We can use this same strategy for pointers to areas in the execution stack. During checkpoint generation, we insert in the pointer the position of the data that pointer is referencing.

To keep track of the allocated memory, we implemented a heap manager. It keeps information about the memory chunks allocated and respective sizes. It also maintains control information used during checkpoint generation, such as a flag that indicates if that memory chunk has already been saved, and the position in the checkpoint where it was saved.

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

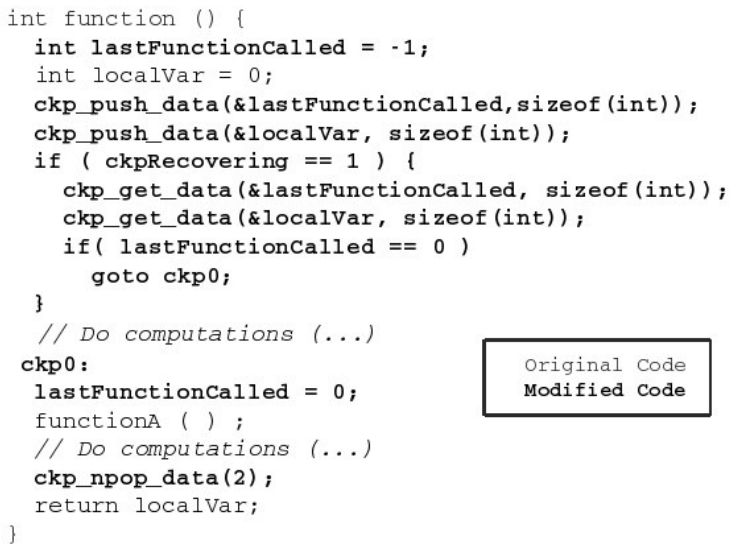


Figure 2. Code instrumented by the precompiler

Replacing the memory addresses by positions in the checkpoint has another advantage. If many pointers reference the same addresses in the heap, they will also contain the same data position in the checkpoint. This guarantees that memory areas from the heap are saved only once and that these pointers that reference the same memory address will continue referencing the same memory area after recovery. This allow pointer graphs structures to be saved and recovered correctly.

To keep the heap manager updated, the precompiler replaces memory allocation system calls – `malloc`, `realloc`, and `free` – in the application source code by equivalent functions in our checkpointing runtime library. These functions update our memory manager before making 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 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 checkpointing coordination. When called by *Processor Zero*, it checks whether a minimum checkpointing interval, which is set by the application, has expired. If positive, it signals all other processes to generate new checkpoints, issues the `bsp_sync` call, and returns true. Otherwise, it issues the `bsp_sync` call and returns false. When `bsp_sync_ckp` is called

by the other processes, it first checks for the signal for generating a new checkpoint. The function then calls `bsp_sync` and returns true if the signal was received, and false otherwise.

Depending on the response from the `bsp_sync_ckp` call, the process generates a new checkpoint. Since the checkpoint is generated immediately after a `bsp_sync` call, addresses registered in the BSP library will contain data from the writes in the previous superstep.

Finally, 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. Another solution would be to save the internal state from the BSP library, but this would save unnecessary information and would require modifying the BSP library.

4.3. Checkpointing Libraries

Two runtime libraries are provided, a checkpointing library and a BSP coordination library.

The *checkpointing library* provides the functions to manipulate the checkpoint stack, save the checkpoint stack data into a checkpoint, and recover checkpointing data, as was described in Section 4.2. It also provide a timer that allows the specification of a minimum checkpointing interval. In the current implementation, the data from the checkpointing stack is first copied to a buffer and then saved to an archive in the filesystem. When the nodes use network filesystem such as NFS, this solution is enough. A robust checkpoint repository is currently being implement.

The *BSP coordination library* provides the functions `bsp_begin_ckp` and `bsp_sync_ckp` used for coordinating the checkpointing process. They also manage obsolete checkpoints, removing checkpoints that are not useful. Since we are using a coordinated protocol that always generates consistent global checkpoints, it is only necessary to keep the last two local checkpoints generated on each process. Two checkpoints are necessary because it is necessary to guarantee that a new global checkpoint was generated before deleting the previous local checkpoint.

4.4. InteGrade Monitoring and Recovering Infrastructure

Monitoring and recovering of failed applications in InteGrade is performed by the Execution Manager. It maintains a list of active processes executing on each node and a list of the processes from a parallel application. Whenever an execution is scheduled by the GRM, it informs the Execution Manager. The LRMs that receive the application processes also notify the Execution Manager when the execution starts and finishes. This guarantees that the Execution Manager always contains a list of all running processes.

The execution of a process can be finished either normally or abnormally. In the later case, a reinitialization of the application may or may not be desirable. For example, in the case an application process receives a segmentation violation signal or exits due to some other problem in the application itself, it is better not to reinitialize the application and inform the user who submitted the application about the error. But if the process is explicitly killed, for example because the machine owner needs the machine resources, it is necessary to reinitialize the killed process in another node.

In case of regular and bag-of-tasks parallel applications, the reinitialization procedure is straightforward. It is only necessary to reschedule the failed process. In the case of BSP applications, it is necessary to consider the dependencies between the application processes.

When the Execution Manager receives the message that a process from a BSP application failed and needs to be reinitialized, it first asks the LRMs where the BSP processes were executing the number[¶] of the last checkpoint generated. The latest global checkpoint containing a local checkpoint from each BSP application process, all with the same number, is determined. The Execution Manager then reschedules the failed process for execution in another node and sends a message to the LRMs containing the remaining processes of the BSP application in order to restart them from the checkpoint corresponding to the last global checkpoint. Finally, it is necessary to ensure that a checkpoint generated by a process who was assigned a process id pid is used by the process that receives this same process id during reinitialization.

5. Experiments

The experiments were performed using a sequence similarity application [1]. It compares two sequences of characters and finds 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.

We evaluated the overhead caused by checkpoint generation for minimum intervals between checkpoints of 10 and 60 seconds. We also used a case were no checkpoints were generated, used to measure the overhead caused by the additional checkpointing code when no checkpoints are performed. We compared the running time of these 3 cases with the running time of the original code.

The experiments were performed during the night, using 10 1.4GHz machines connected by a 100 Mbps Fast Ethernet network. These machines are used by students during the day and usually remain idle during the night periods. We run the application 8 times for each checkpointing interval, using 16 pairs of sequences of size 100k as input. For the comparison, we used the 5 lowest execution times from each case. The results are presented in Table I.

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 3.1%. 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 a minimum interval of 10 seconds, that is a very small interval, the overhead is 12.8%.

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 application that runs for hours, losing a few minutes of computation is normally not a problem. Also, considering that it would be very difficult to execute a long-running parallel application using shared workstations without some checkpointing mechanism, this overhead is easily justified.

[¶]We consider each checkpoint generated is designated by an increasing number starting from 0

t_{min}	n_{ckp}	t_{total}	t_{orig}	ovh
∞	0	967.1s	967.0s	0.1%
60s	15	997.4s	967.0s	3.1%
10s	70	1091.3s	967.0s	12.8%

Table I. 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 mean execution time of the modified code, t_{orig} is the mean 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 [22] 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 Ongoing 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.

A 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 semantic information regarding the type of data. The precompiler can insert this semantic information during the source code instrumentation. Also, data addresses must be saved in a platform independent way. We are currently working in the support for portable checkpoints. In an heterogeneous environment, such as a Grid, portable checkpoints will allow better resource utilization.

We are also working in the development of a robust storage system for checkpoints and application files. 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[‖]. Current versions of the precompiler and checkpointing runtime libraries are available at the checkpointing subproject page[**].

Acknowledgments

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.

REFERENCES

1. ALVES, C. E. R., CÁCERES, E. N., DEHNE, F., AND SONG, S. W. 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), pp. 99–104.
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* (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.

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

9. GOLDCHLEGER, A., QUEIROZ, C. A., KON, F., AND GOLDMAN, A. Running highly-coupled parallel applications in a computational grid. In *22th Brazilian Symposium on Computer Networks (SBRC'2004)* (May 2004). Short paper.
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* (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 checkpointing 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. TONG, W., DING, J., AND CAI, L. Design and implementation of a grid-enabled BSP. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)* (2003).
21. VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33* (1990), 103–111.
22. 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)* (May 2003). Poster session.