Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments

Raphael Y. de Camargo, Fabio Kon, and Alfredo Goldman
Department of Computer Science
Universidade de São Paulo, Brazil
{rcamargo,kon,gold}@ime.usp.br

Abstract

Executing long-running parallel applications in Opportunistic Grid environments composed of heterogeneous, shared user workstations, is a daunting task. Machines may fail, become unaccessible, or may switch from idle to busy unexpectedly, compromising the execution of applications. A mechanism for fault-tolerance that supports these heterogeneous architectures is an important requirement for such a system.

In this paper, we describe the support for fault-tolerant execution of BSP parallel applications on heterogeneous, shared workstations. A precompiler instruments application source code to save state periodically into checkpoint files. In case of failure, it is possible to recover the stored state from these files. Generated checkpoints are portable and can be recovered in a machine of different architecture, with data representation conversions being performed at recovery time. The precompiler also modifies BSP parallel applications to allow execution on a Grid composed of machines with different architectures. We implemented a monitoring and recovering infrastructure in the InteGrade Grid middleware. Experimental results evaluate the overhead incurred and the viability of using this approach in a Grid environment.

1. Introduction

Grid Computing [7] allows leveraging and integrating distributed computers to increase the amount of available computing power, providing ubiquitous access to remote resources. Grids for opportunistic computing [14] are mainly composed of commodity workstations such as household PCs, corporate employee workstations, and PCs in shared university laboratories. The objective is to use the idle computing power of these machines to perform useful computation, allowing organizations to use their existing computing infrastructure for high-performance computing.

Executing scientific applications over shared workstations requires a sophisticated software infrastructure. Users who share the idle portion of their resources with the Grid should have their quality of service preserved. If an application process was executing 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 parallel application consisting of processes that exchange data, stopping a single process usually requires the reinitialization of the entire application. To allow the execution of parallel applications on these environments, we need to address two issues: the unreliability of the execution environment and the heterogeneity of the machines.

The unreliability issue can be solved by mechanisms such as checkpoint-based rollback recovery [6]. Using this mechanism, the application state is periodically saved into checkpoints. In case of failure, the application can be reinitialized from an intermediate execution state contained in a previously saved checkpoint.

To deal with the heterogeneity issue, it is necessary to use mechanisms that consider the differences among machines. The state stored in a checkpoint should be recoverable in a machine with a different architecture. Also, to better utilize the available resources, parallel applications should be able to execute using machines with different architectures. The portability permits the development of more efficient preemptive scheduling strategies and migration mechanisms.

In a previous work, we presented a preliminary version of our checkpoint-based rollback recovery mechanism for sequential, parametric, and *Bulk Synchronous Parallel* (BSP) [22] applications executing over the InteGrade Grid middleware [5]. In this paper, we extend our previous work by modifying the checkpointing mechanism to generate portable checkpoints. We also performed more detailed experiments with the checkpoint-based rollback recovery mechanism to test its suitability in a Grid for opportunis-

tic computing. Finally, we used the precompiler technique to modify the source code of BSP applications to allow its execution and recovery in heterogeneous machines.

The structure of the paper is as follows. Section 2 describes the use of source code transformation of applications to perform portable checkpointing and communication for BSP applications. Section 3 presents a brief description of the InteGrade middleware and its architecture, while Section 4 focuses on the implementation of portable communication and checkpointing. Section 5 shows results from experiments performed using the checkpointing library and InteGrade. Section 6 presents related work on portable checkpointing. We present our conclusions and discuss future work in Section 7.

2. Source code transformation of applications

Source code transformation is a technique where the application source code is instrumented to perform some additional tasks, such as profiling, logging, and state persistence. In this work, we use this technique to modify an application source code to save its execution state in a portable way, allowing the application to later recover its execution from an intermediate state, possibly in a machine of different architecture. We also use this technique to allow data exchange among processes of a parallel application executing on machines with different architectures

In this work we use parallel applications based on the *Bulk Synchronous Parallel* (BSP) model [22]. But the concepts presented in this work can also be used in other types of parallel applications, such as MPI [16] and PVM [20] applications.

2.1. Executing BSP applications on heterogeneous environments

The BSP model is a bridging model for parallel computing, linking architecture and software [22]. A BSP abstract computer consists of a collection of virtual processors, each with local memory, connected by an interconnection network. 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.

Several implementations of the BSP model have been developed, including Oxford's BSPlib [9], PUB [2], and the Globus implementation, BSP-G [21]. Although these implementations can execute on different architectures, all the processes from a single BSP application must execute on computers with the same architecture. The problem is that the API from the BSPlib does not provide a way to specify the type of data being exchanged between processes. Data is transfered as a stream of bytes.

We extended the BSPlib API, adding a parameter to some methods, which describes the type of data being transmitted. To allow programs written for the conventional BSPlib API to execute without extra modifications, our precompiler modifies the BSP application source code to use this extended API transparently. The type information is now used to convert the data between different architectures in the communication among application processes.

A limitation imposed by this approach is that the programmer is not allowed to perform arbitrary casts to data registered with the BSPlib as shared memory. This is a reasonable requirement to provide portable data exchange among processes executing on different architectures.

Thus, using source code instrumentation, we allow existing BSP applications to execute on heterogeneous architectures in a transparent way.

2.2. Portable checkpointing of applications

Application-level checkpointing consists in instrumenting an application source code to save its state periodically, thus allowing recovery after a fail-stop failure [3, 12, 19]. 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 [15, 18].

Since in application-level checkpointing we manipulate application source code, semantic information regarding the type of data being saved is available both when saving and recovering application data. This semantic information allows the data saved by a process on an architecture to be recovered by a process executing on another architecture [12, 19]. This is an important advantage for applications running on a Grid composed of heterogeneous machines since it allows better resource utilization.

The main drawback of application-level checkpointing is that manually inserting code to save and recover application state is a tedious and error prone process. But this problem is solved by providing a precompiler that 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[1].

In the case of parallel applications, we have also to consider the dependencies among application processes. A global checkpoint is a set consisting of one checkpoint from each application process. The global state formed by the states contained in the checkpoints of the individual processes is not necessarily consistent. There are several parallel checkpointing protocols that deal with this problem. The protocols are classified as coordinated, uncoordinated, and communication-induced [6].

[1] In application-level checkpointing, the process state can only be saved in predefined execution points.

Coordinated protocols have several advantages. The protocol always generates consistent global checkpoints, and there is no need to implement a separate algorithm to find consistent global checkpoints. Also, garbage collection of obsolete checkpoints is trivial, since all checkpoints except for the last can be considered obsolete. The disadvantage is the necessity of a global coordination before generating new checkpoints.

In this work, we use a coordinated checkpointing protocol. It is the natural choice for BSP applications since this model already requires a synchronization phase after each superstep. Coordinated checkpointing protocols can also be used for MPI and PVM applications [3].

3. The InteGrade Grid Middleware

The InteGrade project [8, 10] is a multi-university effort to build a novel Grid Computing middleware infrastructure to leverage the idle computing power of personal workstations for the execution of computationally-intensive parallel applications.

The basic architectural unit of an InteGrade Grid is the cluster, a collection of machines usually connected by a local network. Clusters can be organized in a peer-to-peer intercluster federation, allowing it to encompass a large number of machines. The current InteGrade version supports only a single cluster. An intercluster protocol for InteGrade is under development.

Figure 1 depicts the most important 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. *Workstations* export part of its resources to Grid users. They can be shared workstations or dedicated machines.

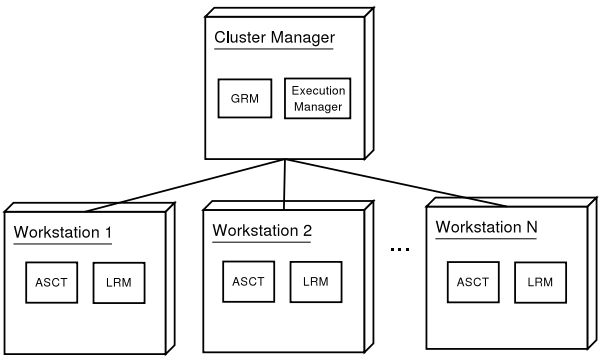

Figure 1. Intra-Cluster Architecture

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

The *Application Submission and Control Tool (ASCT)* allows InteGrade users to submit Grid applications for execution, monitoring the execution, and collecting the results.

The *Execution Manager* performs monitoring and recovering of failed applications in InteGrade. It maintains a list of active processes executing on each node and a list of the processes from a given parallel application. When the GRM detects that an LRM is unreachable, it sends a message to the Execution Manager, and reschedules the processes that were executing on that node for execution on another node. If the unreachable node contained processes from a BSP parallel application, the Execution Manager coordinates the reinitialization of the entire application from the last consistent global checkpoint. We decided to use a centralized manager because each InteGrade cluster should not have more than a few dozen machines. Besides, fault-tolerance for this component can be achieved using replication strategies.

4. Implementation

The precompiler implementation is based on OpenC++ [4], an open source tool for compile time reflective computing. It also works as a C/C++ source-to-source compiler, generating an abstract syntax tree (AST) that can be analyzed and modified before generating C/C++ code again. By using this tool, we did not need to implement the lexer and parser for C/C++.

The precompiler is used for inserting the checkpointing code and to modify the BSP calls to deal with architecture portability. The current implementation of the precompiler covers the C language and has limited C++ support. Features such as inheritance, templates, STL containers, and C++ references will be implemented in future versions.

4.1. Data transformation

Different architectures have different memory representations for data. To allow applications to execute on heterogeneous architectures, it is necessary to provide data converters for each supported architecture. For n architectures, we can implement converters for each pair of architecture, which results in $n^2 - n$ converters, or use an intermediate representation, where each architecture has a conveyor for this intermediate representation, resulting in $2n$ converters.

The first approach produces a faster conversion, since there is no conversion to an intermediate representation. Also, using an intermediate representation requires a tradeoff between memory usage and data precision. The disadvantage of the direct conversion is that the number of con-

verters increases quickly with the number of architectures supported.

We decided to use the direct conversion. We do not expect to support more than about four architectures, which keeps the number of converters relatively low[2]. Also, if the number of available architectures increases, we can use a mixed approach, using one of the supported architectures as the intermediate representation.

We defined one class for each conversion, containing a method for each type of primitive data. We implemented conversions between three architectures: x86 and x86_64 running Linux and PowerPC G4 running MacOS X. The x86 and x86_64 use little endian data representations, differing only on the size of some data types. The PowerPC G4 uses big endian data representations and differs from the other two architectures in all data types. These three architectures also differ regarding data alignment and memory address representations, but in our approach these differences are resolved easily.

4.2. Portable BSPlib

The InteGrade BSP implementation [17] allows C/C++ applications written for the Oxford BSPlib to be executed on an InteGrade Grid, requiring only recompilation and relinking with the InteGrade BSP library.

In the BSPlib API, the `bsp_sync` method is responsible for the synchronization phase of a superstep. Our *BSP coordination library* provides an extended version of `bsp_sync`. This method is responsible for the coordination of the parallel checkpointing protocol, guaranteeing that global consistent checkpoints are generated. It is also responsible for managing obsolete checkpoints, removing checkpoints that will no longer be used.

The BSP model provides interprocess communication based on *Direct Remote Memory Access* (DRMA) and *Bulk Synchronous Message Passing* (BSMP). In the DRMA case, application processes can register local memory addresses as virtual shared addresses using the `bsp_pushregister` method. Once registered, processes are allowed to write and read from these virtual memory locations through the `bsp_get` and `bsp_put` methods. We modified the call to `bsp_pushregister` to include the data type of the registered address. When a remote process writes to a registered memory location, the receiving process checks the architecture of the other process and, if necessary, performs the format conversion. In the case of BSMP, the communication is performed using the `bsp_send` and `bsp_move` methods. Our extended BSPlib API contains a version of `bsp_move` with an additional parameter containing the data type to receive. Again, a conversion is performed only if the machine architecture of the sending process is different.

[2]Actually, the data representation may be dependent on the compiler that generated the code. But differences among compilers for a single architecture are usually very small.

4.3. Portable checkpointing of BSP applications

To generate a checkpoint from the application state, it is necessary to save the execution stack, the heap area, and the global variables. Our precompiler instruments the application source code to interact with a *checkpointing library*, which is responsible for saving and restoring the application state. The checkpointing library also provides a timer that allows the specification of a minimum checkpointing interval. The current implementation, allows saving the checkpoint file in the filesystem (either a local or network filesystem) or in a remote checkpointing repository. Saving is performed by a separate thread, allowing the application to continue its execution during the process.

Saving the execution stack state. 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. The execution stack is non-portable, with its structure varying depending on the computer architecture and even on the compiler. But even if it were portable, it is not directly accessible from application code, requiring its state to be saved and reconstructed indirectly.

During application execution, the execution stack state is constructed as functions are called and local variable values are declared and modified. By calling these same functions and recovering the local variable values, it is possible to reconstruct the execution stack.

Our precompiler modifies the functions in the source program, including statements to push into a checkpointing stack the address of declared local variables. The variable address is removed from the stack when the execution leaves the block were the variable was declared. When a checkpoint is generated, the values contained at the addresses from the checkpointing stack are saved into the checkpoint. Since these addresses point to the location of the local variables in the execution stack, they always contain updated values for these variables.

To save the list of active functions, an auxiliary local variable `lastFunctionCalled` is added to each modified function. This variable has its value modified to a different value before each function call. Using the value of this variable for each function in the execution stack, it is possible to know the complete list of active functions at a given time. We use this same technique to save the program counter. We define that checkpoints will be generated only in certain points during execution, for example, when calling a function `checkpoint_candidate`. A call to this

function defines the exact location in the execution where the checkpoint was generated.

During application reinitialization, only function calls and variable declaration code are executed, until reaching the checkpoint generation point. From this point, application execution continues normally. To recover the value of local variables, the address and data type of the variables are passed as parameters to the checkpointing library. The variable type information is used to convert the variable data representation in case the checkpoint was saved on a different architecture. Data is then copied to the local variable addresses.

Since the data conversion is performed only when recovering the application state, there is no overhead during checkpoint generation. This is important because checkpoints are much more likely to be generated than to be used for recovering. Moreover, the overhead only occurs when the machine where the checkpoint will be recovered is of a different architecture.

```
int function () {
  int lastFunctionCalled = -1;
  int localVar = 0;
  ckp_push_data(&lastFunctionCalled,sizeof(int));
  ckp_push_data(&localVar, sizeof(int));
  if ( ckpRecovering == 1 ) {
    ckp_get_data(&lastFunctionCalled, sizeof(int));
    ckp_get_data(&localVar, sizeof(int));
    if( lastFunctionCalled == 0 )
      goto ckp0;
  }
  // Do computations (...)
ckp0:
  lastFunctionCalled = 0;
  functionA ( ) ;
  // Do computations (...)
  ckp_npop_data(2);
  return localVar;
}
```

Original Code
Modified Code

Figure 2. Instrumented code

In Figure 2, we present a C function instrumented by our precompiler, where the added code is shown in bold face. 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, which can be *normal* or *recovering*.

Pointers. Memory addresses referenced by a pointer are specific to a particular execution and cannot be saved directly in the checkpoint file. To achieve portability, our checkpointing library converts memory addresses to offsets in the generated checkpoint file. When recovering, these offsets are then converted to addresses of memory allocated in the target architecture. This strategy is used for both pointers to memory allocated in the heap area and in the execution stack.

Replacing memory addresses by offsets requires the checkpoint file to be generated in three phases. In the first phase, all pointers are pushed on a pointer stack that contains the pointer addresses and the memory positions referenced by these pointers. In this phase, pointers with multiple levels of indirection are also resolved, including multi-dimensional matrices allocated dynamically. In this case, every individual pointer of the matrix is pushed on the pointer stack. In the second phase, primitive type data and memory chunks allocated from the heap area are copied into the checkpoint and their locations are inserted into a memory position table. Finally, in the third phase, the memory addresses contained in the pointers are substituted by the offsets contained in the memory position table. These three phases require $O(s + p)$ steps, where s is the number of entries in the checkpointing stack and p is the number of elements in the pointer stack.

During recovery, if the target architecture is different from the original, data from memory chunks are converted to the new architecture. The checkpointing mechanism considers a memory chunk as an array of elements of the dereferenced pointer data type and converts these elements while they are read from the checkpoint file.

To keep track of memory allocated in the heap area, the checkpoint library maintains a table of allocated memory areas, including their sizes and position in the checkpoint. To keep the heap manager up to date, our 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 calls.

Structures. The memory representation of structures varies depending on the architecture and compiler. Data padding and alignment is performed either because of architectural requirements or performance improvements. Our solution to solve this problem is to push the address of the individual structure members into the stack. For the recovery, the value of each structure member is read separately. For each structure in the code, the precompiler generates a function containing code to push all members of the structure into the checkpointing stack and a function to recover the structure members from a checkpoint. Since these functions push and recover only primitive types, structure portability is transparent. The checkpointing library calls these functions during checkpoint generation and recovery.

5. Experiments

We performed experiments using two applications. The first evaluates the similarity between two large sequences of characters [1] and the other is a parallelized matrix multiplication application. Both were written according to the BSP model and use the InteGrade BSPlib implementation.

The sequence similarity application finds the similarity between two character sequences using a given criterion. For a pair of sequences of size m and n, executing on p processors, it requires $O((m+n)/p)$ memory. The matrix multiplication application divides an matrix in p parts, one for each processor, and requires $O(n^2/p)$ memory. It produces larger checkpoints and exchanges more data among processes.

5.1. Checkpointing overhead

We evaluated the overhead caused by checkpoint generation for minimum intervals between checkpoints of 10, 30 and 60 seconds. For each interval, we measured the execution time for several cases: no checkpoints generated (t_{orig}), checkpoints saved to the local filesystem (t_{local}, checkpoints saved to a network filesystem (t_{nfs}), and checkpoints saved in a remote repository (t_{repos}).

We performed experiments with the sequence similarity and matrix multiplication applications. Due to space limitations, we only show the results for the later. We used matrices containing 450x450, 900x900, and 1800x1800 elements of type `long double`. These matrices generates global checkpoints of size 2.3MB, 9.3MB, and 37.1MB respectively. For each matrix size, we performed a series of multiplications: 300, 40, and 6 respectively. For each case, we run the application 5 times. Due to fluctuations present in a network of shared workstations, we considered the mean between the two lowest execution times for each case. We used 10 Athlon XP 1700+ machines with 512Mb of RAM, connected by a 100Mbps Fast Ethernet network.

Table 1 presents the results we obtained for the 450x450 and 1800x1800 matrices. ckp_{int} represents the minimum interval between checkpoints and n_{ckp} represents the number of generated checkpoints.

The results show that saving the checkpoint to the local filesystem or to a remote repository implemented using sockets is faster than using NFS. This probably occurs because of NFS caching policies. When using a remote repository, the checkpointing overhead was consistently below 10%, even for 1800x1800 matrices and a checkpointing interval of $10s$. This indicates that it is feasible to use our checkpointing mechanism for applications that use a moderate amount of memory, especially long-running applications.

matrix size = 450x450

ckp_{int}	n_{ckp}	t_{orig}	t_{nfs}	t_{local}	t_{repos}
10s	12	122.8s	136.0s	134.3s	131.5s
30s	4	122.8s	124.3s	128.5s	126.7s
60s	2	122.8s	125.8s	123.4s	125.5s

matrix size = 1800x1800

ckp_{int}	n_{ckp}	t_{orig}	t_{nfs}	t_{local}	t_{repos}
10s	10	165.2s	188.8s	176.8s	176.3s
30s	4	165.2s	195.1s	172.7s	171.0s
60s	2	165.2s	170.2s	168.1s	169.0s

Table 1. Execution times for the matrix multiplication application.

5.2. Execution of BSP applications in the presence of failures

We simulated a dynamic environment where resources can become unavailable at any time. For each node, we generated a number of sequences of failure times representing moments on which that node becomes unavailable. When a failure time is reached, the LRM kills all the processes running on that machine.

To generate the failure times, we used an exponential distribution [11], with $1/\lambda$ representing the mean time between failures (MTBF). We simulated two scenarios, using $600s$ as the MTBF in the first case and $1800s$ in the second. For both scenarios, minimum checkpointing intervals of 10, 30 and 60 seconds were used.

We executed the sequence similarity application 5 times for each minimum checkpointing interval and used the mean total execution time t_{total}. Table 2 shows the results.

ckp_{int}	$1/\lambda$	t_{total}	$1/\lambda$	t_{total}
10s	600s	517.4s	1800s	490.3s
30s	600s	571.2s	1800s	519.3s
60s	600s	699.0s	1800s	534.5s

Table 2. Execution of a BSP application in presence of failures.

The results indicate that a smaller checkpointing interval also reduces the total execution time. This occurs because a smaller amount of computation is lost in every reinitialization. However, for larger mean time between failures and for applications that use large amounts of memory, a smaller checkpointing interval will not always be the best choice. Besides, when there are several applications running concurrently in a Grid, it is better to use larger checkpointing intervals to not overload the network.

5.3. Execution of BSP applications on heterogeneous nodes

In this experiment, we executed the matrix multiplication application using nodes with different architectures. The objective is to determine the impact of heterogeneity in application execution time. We used a configuration composed of 3 x86 AMD Athlons running Linux and 1 PowerPC G4 running MacOS X and executed the matrix multiplication application. We selected this application because it transfers large amounts of data among nodes. Moreover, we used matrix elements of type `long double` to represent the most demanding case[3].

We show the results in Table 3. t_{exec} represents the total execution time, t_{x86} the time to transfer data received from the network to the application memory space in the case where the sending and receiving processes share the same architecture, and t_{ppc} the time to transfer the data when the involved processes are executing on different architectures.

Matrix size	t_{exec}	t_{x86}	t_{ppc}
500x500	28.8s	0.042s	0.217s
1000x1000	156.8s	0.078s	0.430s
2000x2000	373.5s	0.066s	0.348s

Table 3. Execution of a BSP application on heterogeneous nodes.

The experiment results show that converting data from a byte stream is more than 5 times slower than a plain data copy. However, the time spent in the conversion is negligible compared to the total execution time, which indicates that it is viable to run BSP application on heterogeneous architectures.

5.4. Restarting an application from a checkpoint

We compared the time necessary to reinitialize an application using a checkpoint generated in machines of different architectures. The architectures selected for the experiments were x86, x86-64 and PowerPC G4. For this experiment, we used an application that generates a graph of structures containing 20k nodes, with each node containing a `long double` number and pointers to other two nodes. We again used a `long double` number because it requires the highest amount of conversions among architectures.

The checkpoints were reinitialized in an Athlon 2.0GHz using checkpoints generated by processes executing on Athlon, Athlon64 and PowerPC G4 machines. When recovering from the x86 architecture, it required 0.179s. To recover from a checkpoint generated in the x86-64 architecture, it needed 0.186s, an overhead of 3.9% compared to the x86 case, and from checkpoint generated in the PowerPC, it needed 0.192s, an overhead of 7.2%. Although the overhead is not negligible, only a very small fraction of the application execution time is spent on the reinitialization. Consequently, this overhead will probably be unnoticeable.

[3]Besides the difference in the endianness, a x86 or x86_64 machine running Linux with GCC3.3 uses 12 bytes of precision for the `long double`, while a MacOS with GCC3.1 uses only 8 bytes.

6. Related Work

The Oxford BSPlib provides a transparent checkpointing mechanism for fault-tolerance on homogeneous clusters by employing system-level checkpointing [9]. Recently, some research in the area of fault-tolerance for parallel applications on Grids has been performed. MPICH-GF [23] provides user-transparent checkpointing for MPI applications running over the Globus Grid middleware. Kovács et al. [13] presented a checkpointing mechanism for PVM applications running over Condor. All those systems employ system-level checkpointing and a coordinated checkpointing protocol. Consequently, the generated checkpoints are not portable and usually larger compared to application-level checkpointing.

Bronevetsky et al. [3] presented an application-level checkpointing mechanism for MPI applications. They developed a coordinated protocol for application-level checkpointing and a precompiler that modifies the source code of C applications. But differently from our work, the generated checkpoints are not portable.

Porch [19] addresses the generation of portable checkpoints for single process applications. It is a precompiler that instruments a C application source code to generate portable checkpoints. Differently from our approach, the Porch precompiler requires detailed knowledge about data positioning in structures and execution stack for each architecture and compiler it supports.

Karablieh et al. [12] propose using input/output functions from the programming language to save and recover application data in a portable way. The advantage of this approach is that it is not necessary to write conversion routines among the different architectures. The disadvantage is that this process is slower and generates larger checkpoints. Moreover, their approach requires an extra level of indirection for every pointer access during application execution, generating large overheads for applications that use pointers.

7. Conclusions and Ongoing Work

This paper described a mechanism for portable checkpointing and communication for BSP applications on dy-

namic heterogeneous Grid environments. It is supported by a precompiler, a portable checkpointing library, an extended BSPlib API implementation and a monitoring and recovering infrastructure. This mechanism 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.

Our experiments indicate that the overhead of dealing with portability is small and can lead to better resource utilization. For example, it is possible to execute a BSP application in a laboratory containing machines of different architectures. If some of the machines become unavailable, the processes running on these machines can be migrated to another machines, not necessarily sharing the same architecture.

We are currently working on a distributed checkpoint repository, which should improve the scalability and fault-tolerance of the system. We are also working on the support for C++ applications.

References

[1] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. S. W. A parallel wavefront algorithm for efficient biological sequence comparison. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *The 2003 International Conference on Computational Science and its Applications. LNCS, volume 2668*, pages 249–258. Springer-Verlag, May 2003.

[2] O. Bonorden, B. H. H. Juurlink, I. von Otte, and I. Rieping. The paderborn university BSP (PUB) library - design, implementation and performance. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing*, pages 99–104. IEEE Computer Society, 1999.

[3] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In R. Eigenmann and M. Rinard, editors, *Proceedings of the 9th ACM SIGPLAN PPoPP*, pages 84–89. ACM Press, 2003.

[4] S. Chiba. A metaobject protocol for c++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications. SIGPLAN Notices 30(10)*, pages 285–299, October 1995.

[5] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing-based rollback recovery for parallel applications on the InteGrade Grid middleware. In *ACM/IFIP/USENIX 2nd International Workshop on Middleware for Grid Computing*, Toronto, Canada, October 2004.

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

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

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

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

[10] InteGrade Project Home Page, 2004. Available at: `http://gsd.ime.usp.br/integrade`.

[11] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 1991.

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

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

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

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

[16] MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing'93*, Portland, USA, 1993. IEEE Computer Society/ACM.

[17] J. B. Pinheiro Jr., R. Y. de Camargo, A. Goldchleger, and F. Kon. InteGrade: a tool for executing parallel applications on a Grid for opportunistic computing. In *Proceedings of the 23th Brazilian Symposium on Computer Networks (SBRC Tools Track)*, Fortaleza-CE, Brazil, May 2005.

[18] J. S. Plank, M. B. amd G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–323. USENIX Association, 1995.

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

[20] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[21] W. Tong, J. Ding, and L. Cai. 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)*. IEEE Computer Society, 2003.

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

[23] N. Woo, S. Choi, H. Jung, J. Moon, H. Y. Yeom, T. Park, and H. Park. MPICH-GF: Providing fault tolerance on grid environments. In *The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003)*. IEEE Computer Society, May 2003. Poster session.