An Aspect-Oriented Approach to Fault-Tolerance in Grid Platforms

Bruno Medeiros, João Luís Sobral
Departamento de Informática, Universidade do Minho, Braga, Portugal
jls@di.uminho.pt

Abstract. Migrating traditional scientific applications to computational Grids requires programming tools that can help programmers to update application behaviour to this kind of platforms. Computational Grids are particularly suited for long running scientific applications, but they are also more prone to faults than desktop machines. The AspectGrid framework aims to develop methodologies and tools that can help to Grid-enable scientific applications, particularly focusing on techniques based on aspect-oriented programming. In this paper we present the aspect-oriented approach taken in the AspectGrid framework to address faults in computational Grids. In the proposed approach, scientific applications are enhanced with fault-tolerance capability by plugging additional modules. The proposed technique is portable across operating systems and minimises the changes required to base applications.

1 Introduction

Enabling scientific applications to run on computational Grids requires mechanisms to enable scientific applications to address resource faults. This is especially important for long running applications to avoid losing work when a fault occurs, due to the need to restart the application from the beginning.

One effective technique to tolerate faults is to periodically checkpoint the application to disk, in order to restart the execution from the last checkpoint, when a fault occurs.

System Level Checkpointing (SLC) takes a snapshot of the program and all of its memory. This kind of checkpoint has to store all the information of the program, including stack, pointers, so that it can restart the program later. While some tools that do this are able to checkpoint a program without having to halt it (e.g. Berkeley Lab's Checkpoint/Restart [1]), the program has to be linked to a certain library, at compile time. Because of its nature, the time to take a SLC snapshot of the program is longer than with other approaches and the checkpoint usually is larger. Some tools also support parallel programs built with MPI (e.g. BLCR). SLC approaches require support from the underlying middleware and the checkpoint data is intrinsically non-portable across machines, since it is saved on a machine dependent format.

Application Level Checkpointing (ALC) adds new code to the base application that limits the areas to be checkpointed. This approach is smarter than SLC because it uses the knowledge of what needs to be checkpointed, causing fewer problems when working with MPI and/or OpenMP parallel applications. Having to add code to applications is one of its greatest disadvantages. Application-level checkpointing

mechanisms for MPI were proposed in [2, 3]. Both approaches are based on a compiler that assists the programmer to identify the state and places in the program where checkpoint can be performed. Application-level Checkpointing mechanisms for OpenMP were proposed in [4].

In Grid systems it is important to provide portable checkpoint mechanisms. Portability should be two-fold: 1) by implementing checkpoint without requiring changes to the current Grid middleware and 2) by saving checkpoint data in a portable format. Saving checkpoint data in a portable format brings the additional benefit of making it possible to restart applications on a different set of resources. This is suitable for computational Grids since available resources could change during the application run time.

The approach taken in the AspectGrid framework addresses the previous issues by relying on application level-checkpoint mechanisms. In the proposed approach, described in this paper, scientific applications are enhanced with checkpointing capabilities by plugging additional modules implemented with Aspect Oriented Programing (AOP) techniques [5]. Portability is addressed by being a Java-based approach, where application and data are independent of specific platforms. Moreover, provided application level mechanisms avoid changes to the current Grid middleware and the checkpoint data is also portable, supporting the migration of checkpoint data across platforms.

The AspectGrid approach differs from previous works by providing portability in Grid platforms. The framework is fully based on pluggable AOP modules that allow a uniform approach to checkpoint sequential, thread-based and MPI based applications. Pluggable AOP modules combined with a Java based approach add the possibility to take snapshots and to restart applications in different sets of Grid resources and in any of these execution modes (e.g., sequential, thread-based and MPI based applications).

The remainder of this paper is organised as follows. The next section introduces aspect oriented programming techniques and section 3 introduces the AspectGrid approach to checkpoint. Section 4 provides a performance evaluation and section 5 concludes the paper.

2 Overview of Aspect Oriented Programing

Aspect Oriented Programming was proposed to address the problem of crosscutting concern in software systems. These concerns are normally transversal to the application base functionality and are not effectively managed with traditional modularisation techniques. One typical example is the logging functionality, whose implementation with traditional mechanism entails changing the implementation of each function to log.

AOP address this kind of functionality by introducing a new unit of modularity: the aspect. An aspect can intercept a well-defined set of events in the base program (a.k.a., join points) and attaches aspect specific behavior to intercepted events. Additional behavior can be, for instance, to print the name of the intercepted method call. A point-cut specifies a set of events to intercept and point-cut designators can be used to gather information specific to each intercepted event.

AspectJ is an [6] extension to Java that includes mechanisms for AOP. In AspectJ it is possible to capture various kinds of events, including object creation, method calls or accesses to instance fields. Objects and primitive values specific to the context of the captured event are obtained through point-cut designators *this*, *target* and *args*. Fig. 1 shows the example of a logging aspect, applied to a class Point. In this example, a message is printed on the screen on every call to methods moveX or moveY. The wildcard in the pointcut expression is used to specify a pattern for the call's signature to intercept.

```
public aspect Logging {

    void around(Point obj, int disp) : call(void Point.move*(int)) && target(obj) && args(disp) {

        System.out.println("Move called: target object = " + obj + " Displacement " + disp);

        proceed(obj,disp);                              // proceed the original call

    }

}
```

Fig. 1. Example of an aspect for logging

The important AspectJ characteristic is that it allows plugging additional functionality into base applications in a non-invasive manner. In the previous example the program base does not need to be changed to include the logging functionality. Moreover, the logging aspect is "pluggable" in the sense that it can be included in the program when logging functionality is required.

3 Aspect Oriented Checkpointing in the AspectGrid Framework

This section describes extensions made to the AspectGrid framework [7], by providing AspectJ modules that help to include checkpointing capabilities into scientific applications, minimising the amount of changes required to base programs. The provided approach is completely implemented at application level, avoiding the need to change the current Grid middleware. Moreover, it also saves checkpointing data in a portable manner allowing the application to restart on a different set of resources. Portability is also extended to parallel applications developed with AspectGrid tools [8], which include applications that provide Java thread-based parallelism and MPI-based parallelism.

Application-level checkpoint requires saving of application data into a permanent storage. Application data includes the data structures used by the application as well as the call stack, which specifies the particular point in execution where the checkpoint was taken. Application level mechanisms also rely a set of pre-defined points in execution where checkpoint can be taken. This set is required since application-level techniques require cooperation from the programmer/compiler to define the checkpoint frequency and the corresponding places in execution flow.

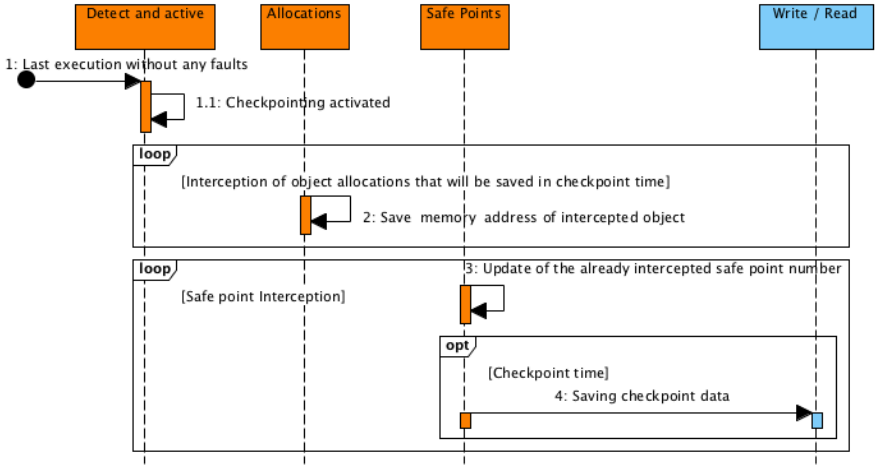

Fig. 2. AspectGrid checkpointing phases

The AspectGrid approach to checkpoint is based on the indication of a set of application data fields (object allocations) to be saved into the checkpoint and a set of safe points that provide points in execution where checkpoint can be taken. Both are specified through AspectJ pointcuts. Checkpointed applications execute as follows (Fig. 2): 1) at application start-up, the *DetectActive* aspect verifies if the last execution was concluded without failures; by intercepting the execution of the "main" method and checking the existence of checkpoint data; 2) if no failure occurred in the last execution the application runs normally and the *Allocations* aspect keeps track of the address of data that must be saved; 3) when a safe point in execution arises the *SafePoints* aspect increments the number of executed safe points and 4) when a predefined number of safe points is executed the data in addresses gathered by the *Allocations* aspect is saved into a file, along with the number of executed safe points.

Application restart in the case of a failure relies on a set of *ignorable methods* that can be skipped during restart (also specified by means of a pointcut). Application restart proceeds as follows (Fig. 3): 1) at application start-up, the *DetectActivate* aspect identifies a failure in the last execution activating the replay mode; 2) the *IgnorableMethods* aspect skips the execution of methods that can be safely ignored. 3) the *SafePoints* aspect increments the number of executed safe points and 4) when the number of safe points saved in the checkpoint file is accomplished the checkpoint data is loaded and execution proceeds normally from that point. Notice that this process rebuilds the calling stack by replaying the original application, ignoring a set of method calls specified by the programmer. Thus, a highly portable solution is attained, since all mechanisms are implemented at application level.

To summarise, in the AspectGrid framework, the programmer has to write three pointcuts: 1) data allocations; 2) safe points and 3) ignorable methods. The AspectGrid framework provides the required additional code to take application snapshots and to restart the application. Moreover, the framework provides a profiling tool that helps the programmer to find and write those pointcuts.

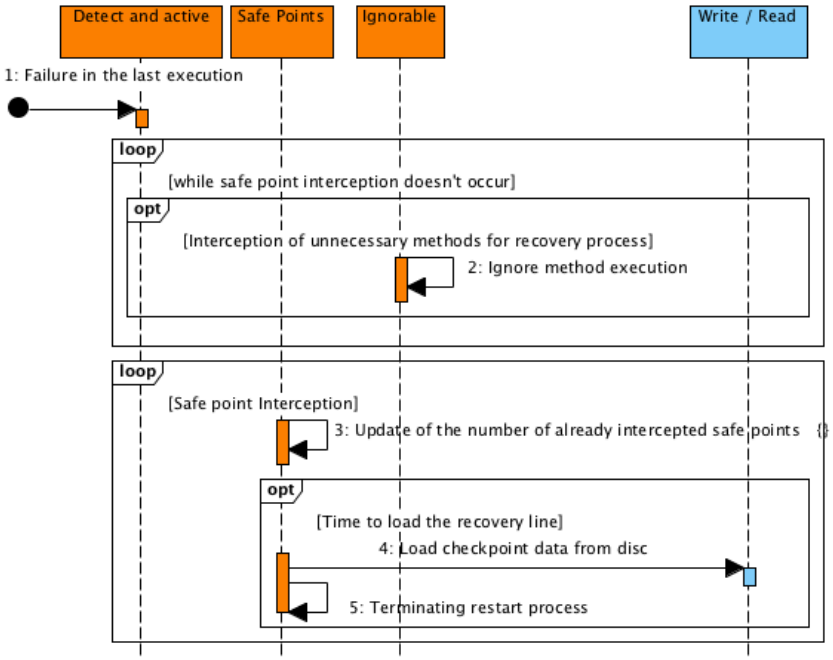

Fig. 3. AspectGrid restart phases

Safe points and *ignorable methods* allow an effective checkpointing strategy. During normal execution, the aspect counts the number of safe points executed. During restart, the application is replayed, ignoring the specified methods, until the same safe point is reached. The selection of the set of safe points is a trade-off between checkpointing overhead and computation lost when a failure occurs. Note that a checkpoint might be taken only after a set of safe points.

AspectGrid approach provides two important benefits: 1) the base code (domain-specific code) remains unchanged following the philosophy of the framework, by providing an additional set of aspects that localise fault-tolerance related issues and 2) the framework automatically provide mechanisms to perform checkpointing in shared and distributed memory systems.

Checkpoint in shared memory systems is performed as follows. When a checkpoint is to be taken (i.e., on a safe point) we introduce a barrier before and another after the safe point. When all threads have reached the first barrier the master thread saves the data specified and the number of safe points executed. Restart is preformed by replaying the application as on a sequential execution, but thread-creation constructs are still executed to rebuild the number of threads and their corresponding call stack. A barrier is introduced after the safe point where the checkpoint was taken. The master thread reads the saved data when reaching that safe point and then releases the other threads waiting at the barrier.

Checkpoint in distributed memory systems is performed as follows. We perform checkpoint on each process as in the sequential case, only special care must be taken to ensure that every process takes the snapshot on the same safe point. We provide two implementation alternatives to save data fields. In the first case, each process

takes a local snapshot. In that case we need to introduce two global barriers, as in the case of the shared memory. In the second alternative we collect the partitioned data on the master node, which avoids the need for barriers (this is possible in our programming model, since we know how the data is partitioned among processes).

Collecting the data and taking the snapshot at the master process has the advantage of making it possible to restart the application on any of the execution modes supported: 1) sequential execution; 2) parallel execution in shared memory systems and 3) parallel execution in distributed memory systems. This is possible since the checkpointed data is the same in all environments. Thus, adaptation can be performed by saving the checkpointing data and restarting with a different configuration. An additional benefit of this approach is that the framework can also checkpoint a hybrid shared/distributed memory parallelisation.

3.1 Illustrative Example

This subsection illustrates the proposed approach by showing how to introduce checkpointing capabilities into a typical scientific application: a Successive Over Relaxation (SOR) that computes the solution to a set of a linear system of equations. This version uses the red-black variation of the algorithm to enable parallelism. This benchmark is a typical scientific application, where a five-point stencil is successively applied to a matrix.

Fig. 4 presents a code snippet of the benchmark (this code is based on the version provided by the Java Grande Forum [9]). The *doIteration* method iteratively calls method *iteration* on red and black matrix elements, alternatively. The *iteration* method calls the *updateRow* on each row, which applies the stencil to all elements in the row.

```
public class Sor {

   static double[][] G;
   static int Mm1, Nm1;
   static double of, omf;

   static final void doIterations(int num_iterations) {
      Mm1 = ...
      for(int p=0; p<num_iterations; p++) {
         iteration(0);     // iteration on "red" elements
         iteration(1);     // iteration on "black" elements
      }
   }

   static final void iteration(int is_red) {
      for(int row=1; row<Mm1; row++)
         updateRow(row, (row+is_red)%2+1);
   }
```

```
   static final void updateRow(int row, int start_elem) {
     double[] Gi=G[row];
     double[] Gim1=G[row-1];
     double[] Gip1=G[row+1];

     for(int j=start_elem; j<Nm1;j+=2){
       Gi[j]=of*(Gim1[j]+Gip1[j]+Gi[j-1]+Gi[j+1])+omf*Gi[j];
     }
   }
}
```

Fig. 4. Base code for the SOR benchmark

The first step to introduce checkpoint capabilities is to identify potential safe points. This can be done using the AspectGrid provided profiling tool. In this case there are three potential points in execution to introduce a safe point: 1) *doIterations*; 2) *iteration* and 3) *updateRow*. Selecting the best place for safe points involves a trade-off between checkpoint frequency and overhead. In this case, the *doIterations* is called only once during program execution. The *iteration* method is called 200 times, with an interval of approximately 2 seconds and *updateRow* is called 20 000 000 with an execution time of a few miliseconds. Thus, in this case, the AspectGrid profiling tool suggests placing safe points on calls to the *iteration* method.

After selection of the safe points, the programmer needs to define the application data structures that must be saved on those safe points. Those correspond to data that is changed between two consecutive executions of safe points. In this case the AspectGrid tool indicates the matrix G.

The last step is the identification of ignorable methods. In this case, the tool suggests that the execution of the code inside safe points can be ignored. The programmer can also indicate other methods that can be ignored.

The three pointcuts generated for this case study are provided in figure 5.

```
pointcut safepoints() : call(void iteration(..) );
pointcut allocations() : call (double[][] new(..));
pointcut ignorablemethods() : call(void iteration(..);
```

Fig. 5. Pointcut definitions to introduce checkpoint in the SOR benchmark

3.2 Implementation Overview

The checkpointing mechanism is based on a set of safe points, ignorable methods and safe data fields. The implemented behaviour is different when the application is running normally and when the application is restarting after a failure. Fig. 6 presents a sketch of the implementation. In normal operation the implementation counts the number of safe points and takes the snapshot when requested (lines 07-12). In replay mode the implementation ignores the specified method calls (lines 22-26) while replaying the application and reload the data when the number of safe points defined in the checkpoint is attained (lines 13-17).

```
01 aspect checkpointing {
02   ...
03   pointcut safepoints();
04   pointcut ignorablemethods();
05   Boolean replay;
06
07   void around(): safepoints(...) {
08     numberOfSafePoints++;
09
10     if (!replay) ...
11       if (takeSnapshot)
12         ... // save data fields
13     else
14       if (numberOfSafePoints==chkSafePoints) {
15         ... // get saved data fields
16         replay = false
17       }
18
19     proceed();  // execute original call
20   }
21
22   void around(): ignorablemethods(...) {
23     if (replay) ; // ignore the method call
24     else proceed();
26   }
27 }
```

Fig. 6. Code for checkpointing

4 Performance Evaluation

This section presents an evaluation of the proposed checkpoint mechanism by measuring the overheads relative to hand written versions. These results were collected on a cluster with two machines, dual Opteron 6174 per node (i.e., 24 cores per machine). Presented results are median of 20 executions. Performance results where obtained on a typical scientific application: the Successive over Relaxation (SOR) presented in previous section.

The first test measures the overhead of introducing code for checkpoint, when 0 or 1 checkpoints are taken. Fig. 7 shows the execution time of: 1) the "original" benchmark; 2) when checkpointing is introducing using classic "invasive" techniques and 3) when checkpointing is introduced through AOP. Presented results include sequential execution (seq); execution with 2 to 16 threads (T) and with 2 to 32 MPI processes (P). These results show that: 1) the overhead of checkpointing is very low, as it would be expected, since the overhead is the time required to count safe points, which is less than 1% in most cases; 2) AOP does not impose any additional overhead when compared to traditional invasive programming techniques; 3) there is a relevant overhead required to save checkpointing data that is directly connected to the amount of saved data.

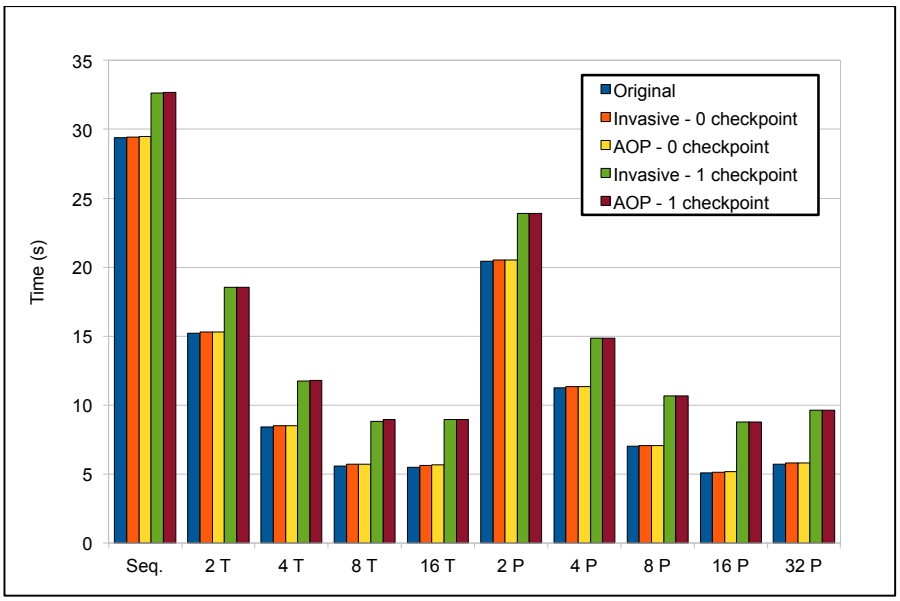

Fig. 7. Overhead of checkpointing

One important point of the proposed approach is the ability to replay the application on a different environment. Figure 8 illustrates such case by showing the time per SOR iteration. In this case the application started with 2 processes and on iteration 26 it restarted on 8 processors, shortening the overall application execution to more than half.

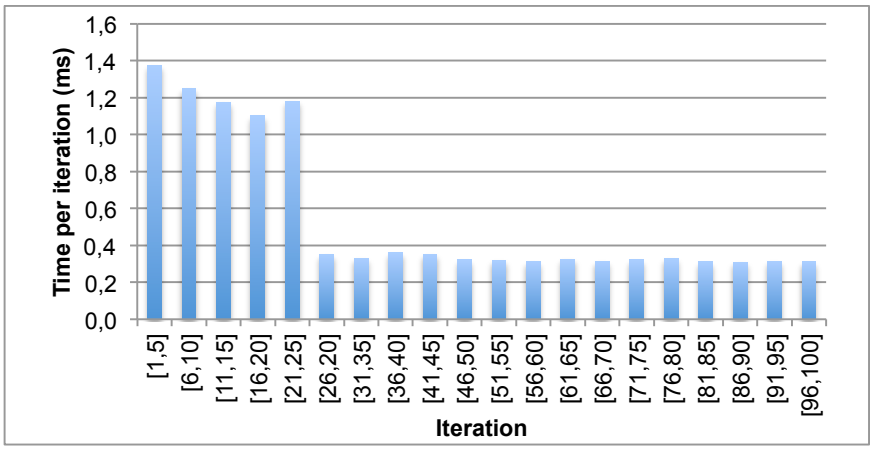

Fig. 8. Application restart increasing assigned resources

5 Conclusion

This paper presented an aspect-oriented approach to checkpointing in computational Grid systems. The approach is based on the ability to plug checkpointing modules in scientific applications. The paper showed the feasibility of the approach and showed that the performance penalty can be very low, when compared with similar hand written versions.

Current implementation of this approach rely on external tools to determinate the optimal set of resources to be used by applications. A natural evolution is to incorporate mechanisms to find opportunities for self-adaptation to improve execution time, by monitoring the application and the system state.

References

1. P. Hargrove, J Duell Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters In Proceedings of SciDAC 2006, June 2006.
2. R. Fernandes, K. Pingali and P. Stodghill, Mobile MPI Programs in Computational Grids, ACM Symposium on Principles and Practices of Parallel Programming (PPoPP), 2006.
3. G. Rodríguez, M. Martín, P. González, J. Touriño, R. Doallo, CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications, Concurrency and Computation: Practice & Experience, Volume 22 Issue 6, April 2010
4. G. Bronevetstsky, K. Pingali, P. Stodghill, Experimental Evaluation of Application-Level Checkpointing for OpenMP Programs, ICS'06, Australia.
5. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An Overview of AspectJ. ECOOP 2001, Budapest, Hungary, June 2001.
6. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting Started with AspectJ. Communications of the ACM, 44(10), October 2001.
7. E. Sousa, R. Gonçalves, D. Neves, J. Sobral. Non-Invasive Gridification through an Aspect-Oriented Approach, 2nd Iberian Grid Infrastructure Conference (Ibergrid 2008), Porto, Portugal, May 2008.
8. J. Pinho, M. Almeida, M. Rocha, J. Sobral, Parallelization Service in the AspectGrid Framework, 4th Iberian Grid Infrastructure Conference, Braga, May 2010.
9. J. Smith, J. Bull, J. Obdrzálek, A Parallel Java Grande Benchmark Suite, Supercomputing Conference (SC 2001), Denver, Nov. 2001.

This work was developed under PRIA (UTAustin/CA/0056/2008) and GAsPar (PTDC/EIA-EIA/108937/2008) projects, supported by Portuguese Fundação de Ciência e Tecnologia.