Adding self-adaptation to scientific applications

Eduardo Brito and João Luís Sobral

{pg12188@alunos.uminho.pt, jls@di.uminho.pt}, Universidade do Minho

Abstract. Existing Grid infrastructures are based on a resource-centered approach where the execution of tasks is delegated to the Grid middleware. This approach prevents applications from using specific features of the environment where they are executed.
Recently there has been interest in developing an approach that is more focused on the tasks themselves and where these tasks are smarter and capable of adapting to the resources. These resources may or may not be available when the application is started but may become available afterwords and they may become unavailable yet again and so on.
In this paper we explore this task-centered approach by analyzing some case studies, we suggest a model for specifying the self-adaptation for this kind of applications and we aim at doing this self-adaptations without stopping the application. We also look into the possibility of using this approach on several languages through the use of virtual machines, bytecode and AOP.

General terms

Scientific computing, Parallel programming, Software engineering, Aspect Oriented Programming

Keywords

Self-adaptation, Self-optimization, Scientific applications, Java, Checkpoint, Restart, Specification, Domain specific languages, Generic aspects, Aspect Oriented Programming, Message Passing, Distributed computing, Parallel computing, Multicore processors, Virtual machine, Bytecode

1 Introduction

Reconfiguration and adaptation has become something very important in today's technological world. New and old terms such as cloud computing, migration, checkpointing, snapshots and virtual machines have gained momentum in these past few years and have become the focus of investigation in some areas.

Even so, most reconfiguration and adaptation that has been done up until now is done to a system as a whole or to applications that are not parallel and/or

distributed. Furthermore, the way that most applications run is completely defined at compile time and they can not restructure themselves as new resources become available or unavailable.

What we aimed to do was to focus on scientific applications and to find ways that would make it possible for these applications to self-adapt to new resources.

This work does not try to explore how to find new resources in a cluster or a grid, nor does it try to estimate when it is the best time to reconfigure the applications. Instead, what we tried to do was to find a generic way for the programmer to specify when and how will the application self-adapt to the new resources, without having to code it himself.

While expanding an application may prove itself to be trivial in some cases, it is not trivial when the application has to distribute itself across a cluster; furthermore, if the application wants do reduce itself (e.g. the grid/cluster infrastructure wishes to give more resources to another application that has more privileges) this can become extremely complex to handle.

To be able to do this, it was necessary to investigate and study large amounts of work that had been done with checkpointing and restarting applications, usually associated with fault-tolerance. As a consequence of that work, we also included a way for the programmer to add fault-tolerance to the application by specifying what it is that he wants to save and when and by adding mechanism for the restart of the application.

2 Related work

2.1 PSL, DDSL and Concurlib

The PSL (Parallelization Specification Language) was a domain specific language in which we worked on in the past. It is a language where we specify clearly in a file how we want to parallelize an application. This PSL simplifies greatly the parallelization of an application and it also locates in a single file how will the parallelization of the application work. Another of its great advantages is that, because the parallelization is not tangled with the base code, we can choose to ignore it and the program will behave as it was originally designed, without the parallelization code.

The DDSL (Data Distribution Specification Language) was the DSL that followed the PSL. While the PSL focused on the methods/functions of the code and how they would parallelize, the DDSL added constructs to ease the parallelization and distribution of the data structures.

Both of these DSLs, although they do not force a concrete implementation, they were implemented through the use of aspects and AOP. Because of this particularity, although AOP prevents tangling, it forces some minor re-factoring of the base code.

The Concurlib is a library of generic aspects that was created to facilitate adding concurrency, parallelization and distribution to applications, using generic aspects that have to be instantiated with the pointcuts that we want to use. It was a predecessor of the PSL.

2.2 Checkpointing and Restart

There exists several kinds of checkpoint philosophies. We present here a summary of the study that we did on these philosophies and on some of the available tools and systems to support them.

The first of these approaches is called System Level Checkpointing. SLC works by performing a snapshot of the program and all of its memory. This kind of checkpoint has to perform very low level code and it has to store all the information of the program, including stack, pointers and such stuff, so that it can restart the program later.

While some applications that do this are able to checkpoint the program without having to halt it (e.g. Berkeley Lab's Checkpoint/Restart), it has the disadvantage of forcing the program to be linked to a certain library, at compile time, which inhibits the use of certain languages such as Java. Also, because of its nature, the time it takes to take a snapshot of the program is longer than other approaches and the size of the checkpoint is usually very big. Depending on the tool that it is used, it can have support for distributed programs that use MPI (e.g. BLCR).

While there exists certain tools that were made for supporting some kind of SLC for Java (e.g. JavaThread, Wasp, JavaGo, Brakes, Sumatra, Merpati, ITS, CIA), they were not usable at this time because they were unavailable or outdated or did not support things that were needed.

Another form of checkpointing is Application Level Checkpointing. This form of checkpointing was used mostly by Keshav Pingali et al in his works with the C3 system. With C3 and ALC, there is need to add new code to the base code that limits the areas that are going to be checkpointed. This approach is smarter than the SLC because it knows better what needs to be checkpointed and the system also provides ways to work with MPI and OpenMP and the several problems that rise from checkpointing programs that use these models of parallel computing.

Even so, having to add more code to the original code is one of its greatest disadvantages. In this case, this forces one to create parsers for all the languages that we may want to support and, if we want to support virtual machines such as the Java VM, the backend needs to be completely rewritten to support the memory model and stack model of Java (the same would happen with almost any virtual machine).

Another form of checkpointing that is used is called Algorithm Based Checkpointing. This is a form of checkpointing that depends on the algorithm that is used and saves only the information that is strictly needed. In this case, the programmer adds its own code to save the state of the application in a manner that he desires and needs to also add code to load and restart the application.

The last form of checkpointing is a kind of mixture between SLC and ABC; it is called Language Level Checkpointing. In this checkpointing style, there is constructs in the language that make it easier to save its state and the programmer must write code, just like in the ABC approach, to save and restore the application. In this case, most of the program state is saved and the program-

ming effort that has to be done is usually less than in the ABC approach because the language makes it easier in some way.

3 Multiple languages and virtual machines

The work presented in this paper was done focusing on the Java Virtual Machine and on the Java language because of the work that had already been done in Java and AspectJ (PSL, DDSL, Concurlib).

With this in mind, we tried to find ways for code that had been written in C, C++ and Fortran 77, 90, to be compiled to Java bytecode, because they are the languages that are most widely used by the scientific community. If this could be done easily and automatically, we could have instantly supported those languages, besides Java, using the same approach and tools.

We found a program called NestedVM that did support this. This program works by having GCC compile a program to MIPS R2000 assembly and by converting the generated assembly code to Java bytecode. To do this it is needed to use the supplied GCC compiler, that only works with C, or to create a cross compiler for MIPS R2000 that supports all those languages. Also, we encountered problems when some "non-standard" libraries were being used. We did not try to solve this problems because we noticed that the generated code by this tool was too slow.

We present here a table with some benchmarks; the CPU time is presented in seconds. The program source code used for the C and Java versions were downloaded from the "The Computer Language Benchmarks Game" site.

Program	C compiled without optimization flag	C compiled with O3	Java	Converted Java with NestedVM
Fannkuch	16,87	6,28	10,51	97,02
Mandelbrot	30,63	17,78	14,12	617,7
Spectral Norm	34,7	28,81	47,47	415,55
N-Bodies	71,3	37,45	52,61	2347,52

We also found another compiler and virtual machine called LLVM that did not generate Java bytecode but was capable of generating bytecode for the Mono/.Net platform and assembly code for the MIPS. Unfortunately, the back-end of the compiler for these targets was still at an experimental stage and was not working properly with our test-bench, so it was not a feasible solution.

We also found two compilers, one for C (LCC) and one for fortran (Lahey Fortran compiler) that could compile to the Mono/.Net platform but the Lahey compiler is not open source and the LCC only supported C.

4 Self-adaptation

Self-adaptation can be used in various scenarios. We can use it to improve the performance of an application by using all the available resources, we can use

it to reduce the application and give more "space" for other applications to run, we can reduce it when our computer/cluster/grid wants to lower the power consumption and we can expand it when the computer/cluster/grid is working at full throttle.

Self-adaptation in this case can also be seen as self-optimization because we try to use in the best possible way the system where the application is going to be working, whether it is a desktop computer at home or a TOP500 cluster/supercomputer. To use the self-adaptation in the best possible way, instead of relying on "smart compilers" or similar kinds of strategy, we have the programmer specify what it is that he wants to do and what parts of the code he wants to benefit with the self-adaptation.

4.1 Case studies

The case studies that we used for this work were originally taken from the Java Grande Forum and they are: LUFact, MD and SOR.

LUFact means LU Factorization. It solves an N x N linear system using LU factorization followed by a triangular solve.

SOR means successive over-relaxation. It is a numerical method used to speed up convergence of the Gauss-Seidel method for solving a linear system of equations.

MD means Molecular Dynamic. It is an N-body code, modeling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions.

We used these case studies because we already had sequential, multi-threaded and DDSL versions of these programs that had been created for other projects.

These scientific applications share several common traits. They all have an outer control loop where they perform complex computations, these complex computations act upon bi-dimensional matrices in an iterative fashion and they can be modified to be multi-threaded and distributed without having to change the algorithm. This kind of scientific applications are the ones we want to focus on.

4.2 Checkpointing and Restart

To checkpoint the application we need to define where we want it to be done. This is usually done at the outer control loop of the main algorithm. To restart the program, we let it execute normally until it reaches the same point where it does the checkpoint; when the application reaches that stage, it loads the previous state of the program and continues executing.

We will analyze each study case individually.

The SOR application works in the following way:

```
//Initialization
(...)
for(p=1; p<LIMIT; p++){
//SOR algorithm
(...)
}
```

We re-factored the code block inside of the loop into a method/function called iterate. This was needed to be done because we do not have a way of using a for loop as a pointcut. Also, we have defined the p variable as being a class variable and not a local variable; this is done to facilitate the access to its value. We use its value as a way of defining "when" we want to do checkpoint. This can be expressed in the following way:

```
Checkpoint<SOR.iterate(..), (SOR.p % VALUE == 0), SOR.p, SOR.GG>;
```

This checkpoints the application at each VALUE iteration of the outer control loop. When it checkpoints, it saves the SOR.p variable and the SOR.GG variable.

The restart can be defined in the following way:

```
Restart<SOR.iterate(..), (SOR.p==VALUE), SOR.p, SOR.GG>;
```

If we want to restart the application as soon as we begin to iterate, we can define the VALUE as being 0. By default, an application only restarts once.

In LUFact, most of the time that is spent is on a method/function called dgefa. This method factors a matrix by gaussian elimination. It does the following:

```
//Initialization
(...)
//Check condition
(...)
//for(k=0; k < limit; k++){
//Perform computations over matrix A
(...)
}
```

As we see here, it is very similar to the SOR. We can define the checkpoint in a similar fashion:

```
Checkpoint<Linpack.for_dgefa(double[][] a, int ipvt[]), (Linpack.k % VALUE
== 0), Linpack.k, a>;
```

This will checkpoint the application at every VALUE iteration. Here we are using parameters of the method call as things that are going to be checkpointed.

The restart can be defined as:

```
Restart<Linpack.for_dgefa(double[][] a, int ipvt[]), (Linpack.k==VALUE), Lin-
pack.k, a>;
```

As for MD, it is the same situation as before. We have a method called runiters that does this:

```
for(move=0; move<LIMIT; move++){
//Perform calculations
(...)
}
```

As before, we also re-factored the block of code to a method that we called runitersfor and defined move as being a class variable.

The checkpoint and restart are exactly the same as before but in this case we have to save a lot of class variables, besides the iteration.

We can not compare this approach directly to other approaches such as SLC because of the nonexistence of tools for Java but we can extrapolate from the work that has been done by others that this in fact reduces the save time and the size file of the checkpoint.

Another thing that is worth mentioning is that, while approaches like ALC need ways to do the restart and replay of the stack of the program and they also need to add lots of work to deal with pointers, stack, message passing, openmp and such low level stuff, we can discard all that because, in this approach, we force the application to replay itself to the point where it was saved and then it loads all the necessary things that it needs to continue. We consider this to be simple but very effective.

4.3 Multi-threaded adaptation

As we have seen before, we have this outer control loop where everything that is important happens. The next step, so that we can have a multi-threaded version is to re-factor the code inside of that control loop, if it is not already re-factored, to separate the things that are to be parallelized.

Lets start again with the SOR study case.

We have seen that there was an outer control loop that was re-factored to a method/function that we called iterate. This iterate method does the following:

```
for(...){
//Some initializations
(...)
//Computes a line
(...)
}
```

What we do now, is to move the code that computes the line to another method that we shall call calcLine.

The multi-threaded version can now be described has:

```
Thread<SOR.calcLine(..), SOR.iterate(..)>;
```

With this we define that it will launch new threads with the SOR.calcLine method and that it will join the results at SOR.iterate. We can also add a condition to it that states that it will only start to use threads at the X-th iteration and that it will stop at Y-th iteration.

We use Concurlib to implement this multi-threaded version.

We are using the value of the iteration variable because we do not have a system that says which resources are available so, we use the iterations to simulate that when it reaches the X-th iteration, some resources have become available and when it reaches the Y-th iteration they become unavailable.

Now lets move on to the LUFact study case.

As we had seen, we had a for_dgefa where all the intense computation was being done. The most important part and the one that takes the longest is the row elimination. We choose to make this part multi-threaded so we re-factor and move it to a method that we shall call execFor.

```
for_dgefa(...){
//Initialization
(...)
//Computes several things
(...)
//Row elimination
execFor(...)
}
```

```
execFor(...){
for(...){
execute(...);
}
}
```

Now that we re-factored for_dgefa and execFor we can now turn this into a multi-threaded application.

```
Thread<Linpack.execute(..), Linpack.execFor(..)>;
```

Once again, we do the multi-threading version using the Concurlib.

For the MD study case we also have the same pattern. We already had re-factored to runitersfor. This method/function works in the following way:

```
{
//Initialization
(...)
//Compute lots of things
(...)
//Compute forces
(...)
//Do some more calculations
(...)
}
```

The most important part in this algorithm is where we compute the forces so we re-factor this code into two methods.

```
computeForcesFor(){
for(...){
computeForces(...);
}
}
```

And to add multi-threading, we just do the same thing again.

```
Thread<MD.computeForces(...), MD.computeForcesFor(...)>;
```

With this approach, it becomes easier to self-adapt into a multi-threaded application and we do not even need to use checkpointing to adapt the application. Also, checkpointing is not affected by this modification because all of the parallelization happens inside the control loop, beginning and ending there, so we are safe to checkpoint the application without having to worry that some thread might still be computing and that some result has not yet become available.

4.4 DDSL, MPI and putting it all together

While adapting an application to become multi-threaded had proven itself to be simpler than it had been estimated, making the application become self-adaptive to clustering through the use of the DDSL and MPI proved itself to be much more difficult to implement.

One of the main problems of adding clustering and using MPI is that there will be several instances of the same application. The DDSL handles this but all the implementations that were done before in this paper had to take this into account as well.

Once again, because we do not have a system that gives us information on the resources that become available, we will have to simulate this.

We start the application with prunjava and several MPI processes. When it starts, the DDSL handles the initialization of MPI and of all things that it needs. Since we are using MPI 2.0, we will be able to start and terminate MPI processes as we see fit but, for now, we did not use this feature.

After everything is started, the application runs as if there was only one instantiation of the application. Since we do not have a method that adds new MPI processes, we start by allocating the data structures that are needed on all of the MPI processes as soon as the processes start. This should be done when we received a "request" to create new processes but since we are simulating this, we allocate all the data structures at the beginning.

This handled the initialization of MPI and the data allocation. We can now use what we have done in the previous sections and run the multi-threaded and checkpoint versions normally. The difference is that, when we decide to start the MPI distribution, the checkpointing will not work properly.

This happens because our checkpoint mechanism is only working for shared memory. To make it work with the DDSL and MPI, it needs to be improved. Also, because we do not have a way to checkpoint all the previous states of the application (we just save the last state), we can not return from the distributed version to a sequential or multi-threaded version.

When the application is distributed, several processes can be at different stages at the same time; to have a coherent state between all of them, we need to have a coherent checkpoint mechanism that saves several states of the application.

While in the previous section, we could start and stop the multi-threaded version at will, in the MPI version we can not revert to previous versions or adapt to add more MPI processes because of our incoherent checkpoint methodology. So, for now, we can only expand the application only once to work with MPI but we can not adapt it again and expand it or reduce it because of this limitation.

We have only implemented a distributed version of LUFact. The specification would be like this:

```
using DDSL;
//Checkpoint/Restart
Checkpoint<Linpack.for_dgefa(double[][] a, int ipvt[]), (Linpack.k % VALUE
== 0), Linpack.k, a>;
Restart<Linpack.for_dgefa(double[][] a, int ipvt[]), (Linpack.k==VALUE), Lin-
pack.k, a>;
//Multi-thread
Thread<Linpack.execute(..), Linpack.execFor(..)>;
//DDSL Allocation
Allocate<Linpack.a, double[CYCLE-TOTAL][]>;
//DDSL Distribution
ScatterBefore<Linpack.for_dgefa(..), Linpack.a, (Linpack.k==VALUE)> ;
GatherAfter<Linpack.dgefa(..), Linpack.a>;
//DDSL Execution
//New re-factorings
(...)
```

The full version of the DDSL execution can be found at [13].

5 Future work

There are some things that are lacking and that should be improved in the future, if this is to be continued.

- A better system for checkpointing distributed applications with the DDSL, that should allow preserving the previous states.
- With the new checkpointing model, there should be a better strategy for restarting and adapting distributed applications; because every MPI process is running at different "velocities", when reducing the application, we may have to go back to the start of the application to have a coherent state. With a smarter restart and replay algorithm for the DDSL version, this could be avoided at least to some extent.
- Some applications need to change their algorithm when they are parallelized and/or distributed. This is a limitation, not of our approach but of our existing implementation; we can only do this automatically to algorithms

that are not affected by the parallelization and distribution. In the future, there should be a way to specify these modifications to the algorithm.
- The creation of an interface that would connect to an application which would provide the available resources and that would send requests to the application to reduce itself, when needed. This should be generic enough to support several kinds of resources and interactions with the applications.
- Implementing the multi-threaded and distribution phases in more scientific applications and also begin using it in bigger case studies.
- A complete implementation of this DSL would be very useful in a near future.
- It would also be a great addition to the DSL if it could handle lower level pointcuts (e.g. for loops), to avoid doing re-factoring to the code. This would have to be done, probably, by building a new weaver that would support this.

6 Conclusion

As for the problem of having multiple languages working under the Java Virtual Machine, for now, it does not seem feasible because there are not compilers, translators or even converters that are good enough to be a truly useful option.

With the work that has been done, we concluded that it is possible to have a generic way of adding checkpoint/restart and self-adaptation to scientific applications and even though the work was implemented in Java, our approach is generic enough to be adapted to other languages.

We consider our approach to be very flexible and highly customizable because everything that is done is specified by the programmer in a rather simple way, leaving all the heavy work to be done by the system itself. Also, beyond having the ability of adding self-adaptation to the resources, it provides extra functionalities, such as checkpoint and restart, to these applications, something that as also been an issue with applications running in grids and clusters.

References

1. Experimental Evaluation of Application-level Checkpointing for OpenMP Programs, Greg Bronevetsky, Keshav Pingali, Paul Stodghill, 2006
2. Mobile MPI Programs in Computational Grids, Rohit Fernandes, Keshav Pingali, Paul Stodghill, 2006
3. C3: A System for Automating Applicationlevel Checkpointing of MPI Programs, Greg Bronevetsky, Daniel Marques, Keshav Pingali, Paul Stodghill, 2003
4. Zero Overhead Java Thread Migration, Sara Bouchenak, Daniel Hagimont, 2002
5. Making Java Applications mobile or persistent, Sara Bouchenak, 2001
6. Pickling threads state in the Java system, Sara Bouchenak, 1999
7. Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations, James S. Plank, Youngbae Kim, Jack J. Dongarra, 1994
8. Complete Translation of Unsafe Native Code to Safe Bytecode, Brian Alliet, Adam Megacz, 2004
9. Taste of AOP: Blending Concerns in Cluster Computing Software, Hyuck Han, Hyungsoo Jung, Heon Y. Yeom, DongYoung Lee, 2007

10. A Domain-Specific Language for Parallel and Grid Computing, J. Sobral, M. Monteiro, 2008
11. Aspect-Oriented Pluggable Support for Parallel Computing, J. Sobral, C. Cunha, M. Monteiro, 2006
12. Reusable Aspect-Oriented Implementation of Concurrency Patterns and Mechanisms, C. Cunha, J. Sobral, M. Monteiro, 2006
13. Paralelização Modular de Aplicações Científicas, Rui Carlos A. Gonçalves, 2008
14. mpiJava 1.2: API Specification, Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko and Sang Lim, October 1999
15. NestedVM, http://nestedvm.ibex.org/
16. LLVM, http://llvm.org
17. JavaGo, http://homepage.mac.com/t.sekiguchi/javago/
18. The aspectj project, http://www.eclipse.org/aspectj/