Universidade do Minho
Escola de Engenharia

Edgar Manuel Fernandes da Mota Sousa

Incrementally Gridifying Scientific Applications

November 2009

Universidade do Minho
Escola de Engenharia

Edgar Manuel Fernandes da Mota Sousa

Incrementally Gridifying Scientific Applications

Dissertation for MSc degree in Informatics

Supervisor:
João Luis Ferreira Sobral

November 2009

DECLARAÇÃO

Nome Edgar_Manuel Fernandes da Mota Sousa

Endereço electrónico: edgar@di.uminho.pt Telefone: 912338058 /

Número do Bilhete de Identidade: 12995012

Título dissertação /tese Incrementally Gridifying Scientifc Applications

Orientador(es): Dr. João Luís Ferreira Sobral

Ano de conclusão: 2009

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Informática

Universidade do Minho, 05/ 11/ 2009

Assinatura: 

Acknowledgements

First, i must thank to my supervisor, João Sobral by all the time he spent with me discussing innumerable subjects, by the freedom he gave me to pursue the lines of work that I considered more interesting.

My acknowledgement also goes to Rui Gonçalves, Diogo Neves and Jorge Pinho, my fellow project members for their support to my work.

And last but not the least, I want to thank João Barbosa for sharing with me its experience.

This work was supported by the project AspectGrid (GRID/GRI/81880/2006) funded by Portuguese FCT (POSI) and by European funds (FEDER).

Abstract

Nowadays, with the ever increasing computing power, science is often done in front of a computer instead of a laboratory. Scientists from biology, chemistry, physics, math, engineering and social sciences are using computer analysis and simulation as their first working tool.

However, with increasing power comes more difficulties, mainly when dealing with recent multicore systems, computing clusters and grid environments.

This work devises a stepwise evolution of scientific applications using Aspect Oriented Programming techniques that starts with applications scientists already wrote. An annotation-based aspect library for multicore programming and a framework for supercomputer environments were developed to implement this vision.

Results from applying these techniques to benchmarks from Java Grande benchmarks suite show that this approach is feasible. The drawback is that this approach is targeted at Java.

Resumo

Nos tempos actuais, com o cada vez maior poder de computação, a ciência é feita muitas vezes à frente de um computador ao invés de um laboratório. Cientistas da biologia, química, física, matemática, engenharia e ciências sociais usam análise por computador e simulação como as suas principais ferramentas de trabalho.

No entanto, com maior poder de computação vêm mais dificuldades, principalmente quando se trata dos recentes sistemas multicore, clusters computacionais e ambientes grid.

Este trabalho apresenta um processo incremental para a evolução de aplicações científicas usando técnicas de Programação Orientadas ao Aspecto partindo de aplicações que os cientistas já escreveram. Uma biblioteca de aspectos baseada em anotações para programação em sistemas multicore e uma framework para ambientes de supercomputador foram desenvolvidas para implementar esta visão.

Os resultados da aplicação destas técnicas a casos de estudo da suite Java Grande mostram que esta abordagem é possível. A principal limitação desta abordagem é o seu foco em Java.

Contents

Chapter 1

Introduction

With the advent of the ENIAC [37], the first general purpose electronic computer built to compute artillery firing tables for the U.S. Army, a new era in science begun. Since then, scientists from all fields heavily rely on computers on their research.

The first massive utilization was seen with mainframes in the fifties, where programs were executed in batches. The evolution of hardware and its lowering cost, and the massification of the personal computer in the nineties opened the possibility of scientific computing on desktop machines.

The decreasing hardware prices and increasing demand for processing power led to a new class of supercomputers. The most popular class became the computer clusters, named Beowulf, after the cluster built by Thomas Sterling and Donald Becker in 1994, with that name, using off the shelf components [45].

With the rising popularity of computer clusters, computer scientists begun to envision supercomputers using connected by the internet, i.e. the grid. An important landmark was the publication of the book "The Grid: Blueprint for a New Computing Infrastructure" [13].

The grid true growing was met with the development of EGEE by CERN, an European Commission funded project started in 2004 to develop software and prepare infrastructures for handling the massive amount of data to be generated by the Large Hadron Collider [23, 31].

While these developments enable scientists to use more computational power than ever, the complexity of the systems is also greater than ever. An application designed to run on a cluster (and take advantage of it) is more complex than one designed for desktop computers and one designed to run on a grid environment is more complex than the cluster one. One last piece of complexity is added with the rise of new processors architecture, namely the multicore processors and Cell architecture, making the effort needed to leverage all the processing power available unbearable for a non-computer scientist. For example, the Roadrunner supercomputer, the fastest as of July 2009[1], uses in each TriBlade two dual core Opteron and four PowerXCell 8i (multicore) processors. [28].

This work has two main goals: to devise a process of incrementally adapt existing scientific applications designed for single core desktop machines into applications suitable to run multicore machines, on clusters and then on grid environments being able to explore the computational power available; and study the possibility of Aspect Oriented Programming to develop production libraries for this purpose.

The rest of this document is as follows: chapter 2 presents the key concepts and related work. Chapters 3 and 4 presents the developed work: a OpenMP-like annotation language, a framework for cluster adaptation of applications and its grid version. Chapter 5 concludes, summing the results of applying the proposed methodology to several example codes.

[1] `http://top500.org/list/2009/06/100`

Chapter 2

Background Concepts

This chapter presents the key concepts that led the development and presents a survey of related, previous work.

2.1 Aspect Oriented Programming

The concept of Aspect Oriented Programming (AOP) was first introduced by Kiczales and his team at Xerox Palo Alto Research Center. In their paper presented on ECOOP'97 [26], they argue that many design decisions that the program must implement are not properly modularized on object-oriented designs. The result of these *crosscutting concerns* is *scattered* code on implementation, most of the times *tangled* with other functionality code. Their work led to the development of AspectJ [25], the most well-known AOP language. AspectJ implements the quantification and obliviousness traits that identify an AOP language [11].

2.1.1 Key Concepts

For this work purpose, here is presented the main AOP concepts as implemented by AspectJ.

Aspects

The modular units composed of pointcuts, advice, inter-type declarations and ordinary auxiliary Java code are called aspects.

Join Points

The join point model provides the frame of reference that enables the dynamic (i.e., runtime/control flow) existence of aspects. It specifies the possible different places where aspects can introduce their behavior. In AspectJ, join points are "well-defined points in the execution of the program" [25, p. 3], including events like calling or executing a method, reading or writing a field and creating objects.

Pointcuts

Pointcuts are the language which the programmer uses to refer to a collection of join points and captures values at those join points.

Advice

Advice are constructs similar to methods used to define the additional behavior to compose at the join points referred by the pointcut.

Inter-type declarations

Not all *crosscutting concerns* are dynamic in its nature. Sometimes, *static crosscutting* is present. Inter-type declarations enable the definition of new operations on existing types, changing the static type signature of the program.

Weaving

Weaving is the compilation process of an aspect. At this stage, the compiled classes and the compiled aspects are woven into a new class where the advice code and inter-type declarations are merged with the base class. This can occur when compiling a project, or alternatively, using an AspectJ provided

class loader, doing only when the class is being loaded by the Java Virtual Machine (JVM). This latter form is known as Load Time Weaving [1]

The next subsection shows how the AspectJ language implements these concepts using a simple example.

2.1.2 Example: Observer Design Pattern

The software design patterns are one of the most used examples to evaluate new AOP languages and techniques [18, 34, 44] and also to show its benefits.

Consider the traditional definition of a two-dimensional *point* with x and y fields and its correspondent getters and setters. If there is a *screen* which should have its state updated when the point changes, the observer pattern offers an alternative to the *screen* object using a timer or a loop to check the *point* state. The *point* object will have a reference to the *screen* and notify it on each modification of its state. Figure 2.1 shows the class definition for this approach.

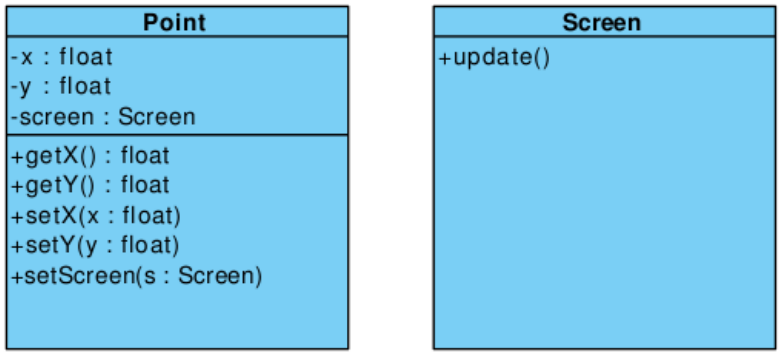

Figure 2.1: Applied observer pattern.

The field *screen* and the method *setScreen* are not part of the *Point* specification. More, both *setX* and *setY* methods would have the line *this.screen.update()* on its body to inform the *screen* of the possible need

[1] It is enabled by passing the option `-javaagent:aspectjweaver.jar` to the JVM.

to update itself. This illustrates *scattering*: the functionality is scattered among different places; and *tangling*: the code is mixed with non-related functionality code.

An aspect-oriented design would leave in its own module the *concern* of updating the screen state. Algorithm 2.1 shows a possible AspectJ implementation of the observer pattern in this case.

On lines 2 to 5 is shown how the static part of the pattern, new fields and methods (inter-type declarations), could be imposed on the base class without modifying its source code.

Lines 6-8 implement the behavioral part of the pattern. The pointcut in line 6 specifies "every call to void methods whose name starts with *set* belonging to the class *Point*, with any number and types of parameters, and bind the variable *p* to object targeted by the call" (*quantification*). The advice body says that after each event specified by the mentioned pointcut, the method *update()* of the related *screen* object should be called.

The *Point* implementation is not conscious of these extra fields, methods and behavior specifically related to the pattern implementation (*obliviousness*).

Algorithm 2.1 Observer pattern implementation.

```
1 public aspect ObservedPoint {
2     private Screen Point.screen;
3     public void Point.setScreen(Screen s){
4         this.screen = s;
5     }
6     after(Point p):call(void Point.set*(..))
          && target(p) {
7         p.screen.update();
8     }
9 }
```

2.1.3 AOP on Parallel Computing

Since its inception, AOP has spawned across different areas: software security [4, 20], logging and security for critical systems [12], debugging [22], wireless sensor networks [30], real time operating systems [36], among many others.

The recent rise of multicore systems is demanding a shift on application development. While popular languages like Java have support for multi-threading programming, doing so by hand is error-prone and causes tangling between threading and application code, making parallel programming a good candidate for aspect-oriented programming.

The first article on using AOP for parallel programming was presented on AOSD'04[19]. Their premise was that it should be possible to use aspect-oriented design to separate mathematical model code from performance code in scientific applications. This line of work influenced several researchers, on work like aspects-enabled Java virtual machines[24], step-wise development of parallel applications[39], mixing component and AOP approaches [3] and (un)pluggable aspects for parallelization [15, 41, 40, 42].

2.2 Grid Programming

Gridification is defined as "the process of writing/modifying an application to utilize the various services provided by a specific Grid middleware" [33, p. 528]. The process includes parallelizing the application code, resource discovery and scheduling applications on available resources.

How simple the process may seem, grid environments impose several issues that the gridification process must address:

- Resource heterogeneity;
- Dynamic changing resources or applications;
- Unknown behavior of the platform deriving from the partial knowledge of the system;
- Security: resources are shared on different administrative domains

2.2.1 Frameworks for Gridifying Legacy Applications

Several frameworks have been proposed to ease the gridification of existing applications. Mateos et al propose a taxonomy for classifying such frameworks [33, pp. 542-550], being invasiveness and granularity the most noteworthy because of its impact on program development. The following survey is made accordingly to this classification.

The Java GAT [1] is a grid API that aims to provide a simple interface to multiple grid middleware. Ibis [35], ProActive [2] and HOCs [16] provide middleware to develop parallel applications that can take advantage of grid systems. Gridgain [17] is an commercially supported open source framework designed specifically to support the development of grid applications. Grid-enabling applications in these approaches require invasive and non-reversible source code changes. In these approaches grid-enabled scientific applications become dependent of the Grid middleware.

GEMLCA [9] and GRASG [21] are two frameworks supporting non-invasive gridification of scientific codes. These approaches perform a coarse grain gridification, by deploying scientific codes as grid services. These approaches lack support for fine-grained decomposition of the application functionality to take advantage of the power of computational grids.

Non-invasive fine-grained gridification has been previously applied to applications that adhere to specific coding conventions. The Pagis system [46] explores the use reflection techniques to gridify applications structured according the paradigm of process networks. AOP techniques have been previously applied to abstract the process of remote execution of Java Thread-based applications [32] and to implement the adaptation of a skeleton framework [10] to cluster and grid environments [38]. These approaches still require coding conventions, but their use of AOP avoids using reflection techniques or preprocessing tools.

Chapter 3

JPPAL - Java Parallel Programming Annotation Library

3.1 Introduction

The very first step in our gridification process is to explore the parallelism possibly available on the application. Despite that in the last twenty years parallel programming was the target of a large number of software engineering works with the goal to simplify development process, this process is still perceived as complex and issues as under utilization of resources and race conditions are still problematic even for skilled programmers.

This complexity is the reason why there is a lot of effort to develop parallel programming aids for languages that already support natively parallel programming, namely Java. However, the most well know approach for shared memory parallel programming, OpenMP, has not made its way on the Java language.

This chapter describes the created OpenMP-like programming interface for Java, in library form made of reusable aspects. Similarly to OpenMP, this library, implemented using AspectJ, provides annotations for use in an object-oriented way in Java programs.

The following sections describe the library rationale (3.2), its implementation (3.3.4) and findings (3.5).

3.2 Library Overview

The main previous work motivating the work presented on this chapter was done by Carlos Cunha and presented on AOSD'06 [7]. Cunha implemented common concurrency patterns like one-way, futures, active objects, barriers and guards. The novelty in his work was that all patterns were implemented as a library of reusable aspects. The only extra code needed was the specific pointcut or a Java annotation on the base code to apply the intended pattern.

However, the goal of making OpenMP available to Java programmers is not new. Bull et al. [6] presented an implementation of the OpenMP specification as close as possible using an extension to JavaCC to insert new behavior based on comments, which could be seen as a form of source code transformation, and Klemm et al. [27] extended their approach with new mechanisms to cope with object-orientation and Java specifics. Both works try to mimic the OpenMP behavior as seen on popular C/Fortran compilers, even the memory model.

Different from these works, this library follows the same approach as the concurrency patterns library aforementioned, using annotations instead of comments to apply the parallelization mechanisms and being a library instead of a source code transformation tool. The use of annotations has three main reasons: programming with annotations is natural to Java programmers, removes the need to be knowledgeable about aspect-oriented programming, AspectJ and the internal working of the library, and avoids pointcut fragility [29].

The library implements some of the most used mechanisms found in OpenMP: *parallel*, *single*/*master*, *critical*, *barrier* and *for*. Each of those mechanisms gave origin to one annotation (except two for barrier). For each of those mechanisms, aspects were created that look for annotation presence to apply the advice.

3.2.1 Mechanism use

Each construct is applied to code by tagging a method with the intended annotation. The simplest example, to execute a method by several threads,

is shown on algorithm 3.1. The **@Parallel** annotation specifies that the method will be executed in this case by 4 threads.

Algorithm 3.1 Example of annotation use.

```
public class Foo {

   @Parallel(n=4)
   public void someMethod(){
     ...
   }
   ...
}
```

3.2.2 Design decisions

OpenMP is a specification targeted only at C/C++ and Fortran. So, we decided not to mimic the programming constructs as defined by OpenMP because the differences between those languages and Java.

First, mechanisms apply only to methods, as they are the fundamental part of the application interface. Doing so, one can write a subclass and use the same annotations on the overriding methods to keep the parallelization.

Second, we follow the language spirit when dealing with thread access to variables. Class members are shared and not guaranteed to be up to date unless declared *volatile*; method arguments and local variables are private.

Third, we impose refactoring to expose the right context. This results in more explicit API's as opportunities of parallelism became part of it.

3.2.2.1 Refactoring

The library was tested against the Java Grande benchmark suite [5]. Similarly to Harbulot [19, section 4] we found that refactoring was needed in order to expose the needed pointcuts for parallelization. Since methods are the smaller unit that annotations can apply to, the main code transformation was to replace a block of code with a method, or as we call it, to give it a name. The programmer can do this easily with a tool like Netbeans[1]

[1]See `http://wiki.netbeans.org/Refactoring`.

Algorithms 3.2 and 3.3 show the code before and after applying this refactoring technique to *Series* benchmark of Java Grande.

By moving the computation loop to a method of its own that exposes the variables controlling the loop (*start*, *end*, *step*), an annotation can be applied to it and the advice can get the proper context information.

Algorithm 3.2 Main Loop of *Series* before refactoring.

```
1  void Do()
2  {
3      double omega;       // Fundamental frequency.
4
5      // Calculate the fourier series.
6      // Begin by calculating A[0].
7      TestArray[0][0]=TrapezoidIntegrate(
8                              (double)0.0,
9                              (double)2.0,
10                             1000,
11                             (double)0.0,
12                             0) / (double)2.0;
13
14     omega = (double) 3.1415926535897932;
15
16     for (int i = 1; i < array_rows; i++)
17     {
18         TestArray[0][i] = TrapezoidIntegrate(
19                             (double)0.0,
20                             (double)2.0,
21                             1000,
22                             omega * (double)i,
23                             1);
24
25         // Calculate the B[i] terms.
26         TestArray[1][i] = TrapezoidIntegrate(
27                             (double)0.0,
28                             (double)2.0,
29                             1000,
30                             omega * (double)i,
31                             2);
32     }
33 }
```

Algorithm 3.3 *Series* **after** refactoring.

```
1  void Do()
2  {
3      double omega;        // Fundamental frequency.
4
5      // Calculate the fourier series.
6      // Begin by calculating A[0].
7      TestArray[0][0]=TrapezoidIntegrate(
8                                 (double)0.0,
9                                 (double)2.0,
10                                1000,
11                                (double)0.0,
12                                0) / (double)2.0;
13
14     omega = (double) 3.1415926535897932;
15     this.execFor(1,array_rows,1,omega);
16 }
17
18 private void execFor(int start, int end,
19                           int step, double omega) {
20
21     for (int i = start; i < end; i+=step) {
22         TestArray[0][i] = TrapezoidIntegrate(
23                             (double)0.0,
24                             (double)2.0,
25                             1000,
26                             omega * (double)i,
27                             1);
28
29         // Calculate the B[i] terms.
30         TestArray[1][i] = TrapezoidIntegrate(
31                             (double)0.0,
32                             (double)2.0,
33                             1000,
34                             omega * (double)i,
35                             2);
36     }
37 }
```

3.2.2.2 Technology

Being implemented using AspectJ brings, in this particular case, three immediate advantages. First, the library itself could be distributed in binary form, and using Load Time Weaving, the same binary package can be used on all the applications without recompiling it (akin to shared libraries); second, one can write new aspects for new concerns applied to the same pointcuts specified by the annotations to improve certain functionality; and third, if needed, anyone can change the library and specify pointcuts instead of annotations in case where the access to the application source code is not possible[2].

Other important fact is that by using AspectJ, we don't need to use other more complex tools to introduce the extra behavior. The alternative to AspectJ would be parsing and transforming the source code.

3.3 Mechanisms

3.3.1 Parallel

The *Parallel* annotation declares the start of a region to be executed by several threads simultaneously. It has one mandatory argument, n, the number of threads (in case that $n < 1$, it will use the number of available processors).

Method parameters are private to each thread, including object typed ones. Java semantics apply in this case: the variable is a private *reference* for a object that is shared by all threads. On the other side, object fields are shared among threads.

3.3.1.1 Aspect Implementation

Algorithm 3.4 shows the base implementation of this aspect that is active when a method tagged `@Parallel` is called (pointcut *parallelMethod*), creating n threads (lines 16-18).

[2]This approach relies on the assumption that code is already exposing the needed joinpoints for parallelization.

For method to run in parallel, $(n-1)$ *Runnable* objects are created[3]. Each one will execute the special *proceed* AspectJ statement when the ThreadPool Executor runs it (lines 20-28). After proceeding with the method call (line 29), the main thread will join with the other threads (lines 30-38).

The aspect also provides for the threads' use, a shared storage area, various information about the parallel zone (master id, group size, etc.), a barrier and a lock.

[3]There is no missing thread. The working (main) thread will execute the same code as the others.

Algorithm 3.4 *ParallelMethod* aspect implementation.

```
1  import java.util.*;
2  import java.util.concurrent.*;
3
4  public aspect ParallelMethod {
5      public ExecutorService exec=null;
6      public ReentrantLock groupLock;
7      public TournamentBarrier groupBarrier;
8      /* Other parallel region state variables */
9      pointcut parallelMethod() :
10                     call(@Parallel * *.*(..)) ;
11
12     Object around(final Parallel pm) :
13         parallelMethod() &&  @annotation(pm) {
14         groupSize = pm.n()>0 ? pm.n():
15             Runtime.getRuntime().availableProcessors();
16         if(exec == null && groupSize>1)
17             exec = Executors
18                     .newFixedThreadPool(groupSize-1);
19         /* Other initializations */
20         for(int i=1;i<pm.n();i++)
21             Runnable r = new Runnable() {
22                 public void run() {
23                     myid.get();
24                     proceed(pm);
25                 }
26             };
27             futures.add(exec.submit(r));
28         }
29         proceed(pm);
30         if(groupSize>1) {
31             try {
32                 for(Future<?> f : futs)
33                     futures.get();
34             } catch (Exception e) {
35                 e.printStackTrace();
36             }
37             futs.clear();
38         }
39         return null;
40     }
41 }
```

3.3.2 For

This is the only supported work sharing construct. At this time is mandatory that the method it applies has a subclass of *Number* as return type.

To be able to use the `@For` annotation, the method must have as its first three parameters three *int*s, meaning (in order): the loop iterator initial value, end value and step; also the method muss be called from inside a parallel region.

When a thread executing a parallel region finds a call to an annotated method with `@For`, it will modify the aforementioned parameters based on their values and its own id in order to share the work, proceeding with n calls to the method.

The annotation itself has one argument, *reduction*, that can be `NONE`, `AVG`, `MAX`, `MIN`, `SUM` or `PROD`, specifying how to combine the individual results from each method call into one.

Algorithm 3.5 shows this annotation applied to *Crypt* benchmark.

Algorithm 3.5 Using the `@For` annotation.

```
1  @Parallel(n=4)
2  private void cipher_idea(byte[] text1, byte[] text2,
3                           int[] key){
4    //...
5    this.execFor(0,text1.length,8,text1, text2, key);
6    //...
7  }
8
9  @For(reduction=Reduction.NONE)
10 private Number execFor(int st, int en, int step,
11       byte[] text1,byte[] text2, int[] key) {
12
13   for (int i = st; i < en; i += step) {
14       //...
15   }
16   return null;
17 }
```

3.3.3 Other Mechanisms

BeforeBarrier & AfterBarrier

These annotations have no arguments and causes the threads to synchronize before (after) entering (leaving) the annotated method.

Master & Single

Causes a method to be executed only by one thread. With *Single*, the first thread entering the method executes it. All threads synchronize at the end of the method.

Critical

This annotations marks a section of the code that can be executed only by one thread at a time. No order between threads is guaranteed.

3.3.4 Implementation Limitations

3.3.4.1 ParallelMethod

AspectJ supports the multiple instances of an aspect. To achieve it one can write `public aspect X percflow(somepointcut()) {` and have a different instance in each join point matched by the pointcut. That instance is accessed using X.*aspectOf()* method. However, if one thread is created a that point, a call to X.*aspectOf()* will return a null object. For this reason, it is not possible to have nested parallel zones, because only a per VM aspect can exist.

3.3.4.2 ParallelFor

The method signature is imposed by the mechanism and does not allow for other return types than *Number*. Also, there is no load balancing at all (a workaround is to specify a number higher than the number of processors on `@Parallel` annotation.

3.3.4.3 Single

Its impossible with AspectJ based tools to implement a non-blocking version of *Single*. Algorithm 3.6 shows an example why it is impossible. When a thread encounters the call to *doIO()*, its impossible without maintaining a complex flow analysis if it should proceed to the call or skip it.

Algorithm 3.6 *Single* example.

```
1  @Single
2  public void doIO(){
3      /*...*/
4  }
5
6  @Parallel(0)
7  public void method(){
8      for(i=0;i<n;i++){
9          /*...*/
10         doIO();
11     }
12 }
```

3.4 Performance Results

When we wrote this library, someone raised the question that such a library would impose significant overhead.

The results obtained with the some of the benchmarks from Java Grande suite show the library does not cause significant performance loss.

Values were gathered on a 2 x Intel Xeon E5420 @ 2.50GHz (QuadCore) machine, with 7GB RAM, running CentOS 3.4 kernel 2.6.9-42.0.2.ELsmp x86_64, Sun Java SE 1.6.0_u11 and AspectJ 1.6.3. Time is in seconds. The first tables show the values obtained with the original multithreaded versions and the second ones the values obtained with the library.

Threads	1	2	4	8
SizeA	0,658	0,308	0,184	0,112
SizeB	3,652	2,151	1,028	0,478
SizeC	8,929	4,389	2,691	1,135

Threads	1	2	4	8
SizeA	0,694	0,381	0,221	0,159
SizeB	3,990	2,192	1,133	0,610
SizeC	10,332	5,385	2,728	1,409

Table 3.1: Crypt execution times.

Threads	2	4	8
SizeA	2,14	3,58	5,88
SizeB	1,7	3,55	7,64
SizeC	2,03	3,32	7,87

Threads	2	4	8
SizeA	1,82	3,14	4,63
SizeB	1,82	3,52	6,54
SizeC	1,92	3,79	7,33

Table 3.2: Crypt speedups.

Threads	1	2	4	8
SizeA	9,061	4,811	2,599	1,195
SizeB	85,836	43,408	21,541	10,825
SizeC	1307,229	808,388	420,502	220,324

Threads	1	2	4	8
SizeA	9,112	5,104	2,605	1,359
SizeB	83,487	48,471	23,583	11,095
SizeC	1307,005	809,516	419,377	226,306

Table 3.3: Series execution times.

Threads	2	4	8
SizeA	1,88	3,49	7,58
SizeB	1,98	3,98	7,93
SizeC	1,62	3,11	5,93

Threads	2	4	8
SizeA	1,79	3,5	6,7
SizeB	1,72	3,54	7,52
SizeC	1,61	3,12	5,78

Table 3.4: Series speedups.

3.5 Conclusion

This work presents a simple way to express parallelization, by using Java annotations to express the parallelization points. The library does not hinder program gains from parallelization.

On the negative side, we have the lack of AspectJ's ability to target loops. Thus, the base code needs to expose in its interface the suitable points of parallelization. At this time, the only way to accomplish that is to move the loop to a method of is own where this parameters are arguments to the method enabling AspectJ advice to capture and change them (applying the refactoring approach shown on subsection 3.2.2.1).

To overcome this limitation when using either pointcut or annotation approach, Java could be extended with true named blocks[4] and then AspectJ could target them. Algorithm 3.7 proposes a syntax for this that is able to expose local context.

[4]Java supports named blocks at source level. However, those names are not available in the generated bytecodes.

Algorithm 3.7 Named blocks example.

```
int foo(){

    //statements
    fooLoopName(int start, int max):{
        for(int i=start;i<max;i++){
            //loop body
        }
        //after loop statements
    }
    //statements
    return value;

}
```

With those facilities, it would be possible to apply annotations directly without refactorings. The code expressed on algorithm. 3.8 could be a reality.

Algorithm 3.8 Inline annotations on named blocks example.

```
int foo(){

    //some statements
    //more statements
    @Parallel(n=DEFAULT)
    @For(reduction=Reduction.NONE)
    fooLoopName(int start, int max):{
        for(int i=start;i<max;i++){
            //loop body
        }
        //after loop statements
    }
    //statements
    return value;

}
```

Chapter 4

The *AspectGrid* Framework

4.1 Introduction

One of the mandatory steps in gridifying an application is transforming it to use a supercomputer (cluster) environment. The traditional approach is to rewrite from scratch all the structure, keeping only the functionality that is modularized. Most of the times this conversion results in parameter sweep applications, where some "kernel" code is executed many times with a different parameter on each execution.

Applications that do not rely on parameter sweep require an additional burden to be decomposed into a set of independent tasks that can be executed on multiple computing nodes. Moreover, parameter sweep may not be affordable for some kind of applications as some run(s) may be dependent on another(s).

To help this gridification step, it was developed an aspect based framework [1] to declare, in a non-invasive manner, the different tasks composing the application. The framework provides an easy way to execute the tasks on a supercomputer without manually introducing code for distributed memory. To achieve this we use the concept of *pluggable grid services*[42] that can be composed to meet the requirements of the application and/or platform. This is similar to the concept of component binding described by Dangelmayr [8].

[1] This work is an evolution of the embryonic work presented at Ibergrid'08 [43]

4.2 The framework

The foremost requirements for the framework were that the gridification would be non-invasive, the framework itself would be lightweight and the application programming interface would be minimalist. This goal is accomplished by relying on a well defined workflow that provides the adequate join points to plug additional services. Figure 4.1 shows this workflow. The framework adapter serves as an entry point to the framework domain that non-invasively converts domain computations into framework tasks. Each of the eight steps are subject to be intercepted by any new module. The core service, aided by execution services and a "Task API" compose the framework. The non-invasiveness is achieved by creating adapters to the user code, using AspectJ.

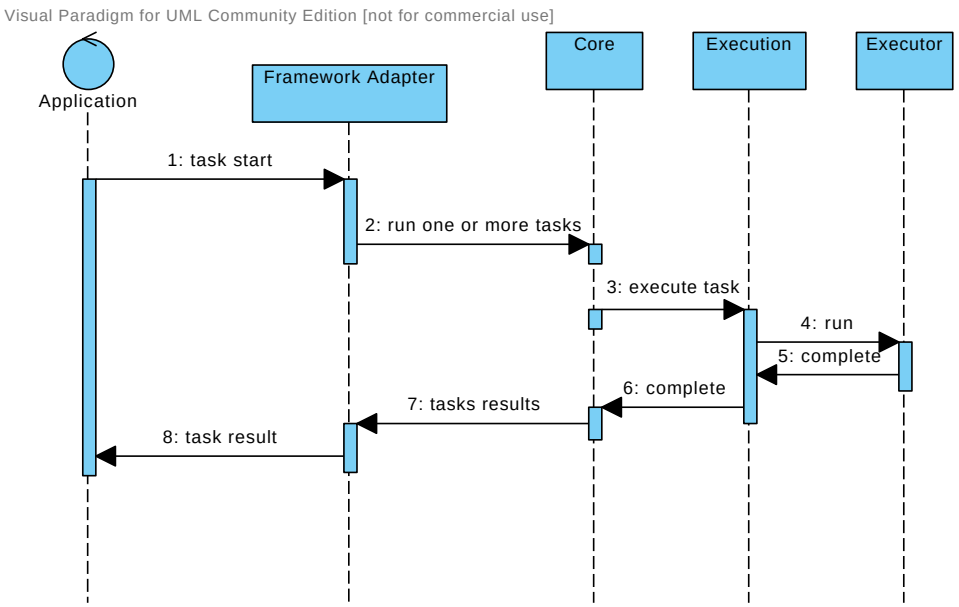

Figure 4.1: Framework conceptual architecture.

4.3 Global Architecture

The class diagram in figure 4.2 shows the main and essential parts of the framework that materialize the generic concepts from the previous subsection. Using a program for computing the Mandelbrot set, the subsections present in detail the role of each one.

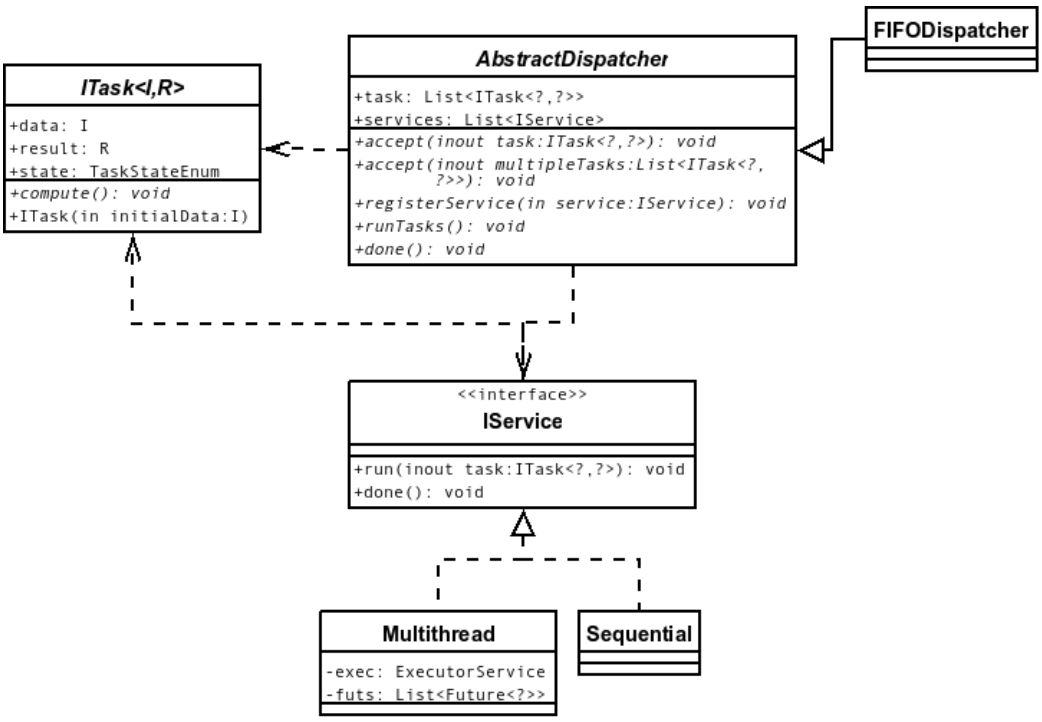

Figure 4.2: *AspectGrid* class diagram.

4.3.1 *ITask*

This datatype serves to encapsulate every task submitted to the framework. The abstract method *compute* should comprise the needed steps to perform the computation having the object *data* of parametric type I as input and storing the result of type R in *result*.

Applications using the framework (see section 4.4) should define the required subclass and instantiate it as needed. Algorithms 4.1 and 4.2 shows how this can be accomplished on the Mandelbrot set example from Java

Grande benchmark suite [5].

The class *MandServer* is where the computation of the Mandelbrot set is implemented (method *mandelBrot*). Algorithm 4.1 shows the implementation of *ITask* interface where the *compute* method just redirects the computation to a MandServer object. The referred class *MandData* encapsulates computation parameters.

Algorithm 4.2 is a snippet from a *Framework Adapter*.

Algorithm 4.1 *ITask* use.

```
1  public class MandelbrotTask extends
2          ITask<MandData, Short[][]> {
3
4      public MandelbrotTask(MandData initialData) {
5          super(initialData);
6      }
7
8      @Override
9      public void compute() {
10         MandServer m = new MandServer();
11         this.result=m.mandelBrot(this.data.c0,
12                                  /*...*/);
13     }
14 }
```

Algorithm 4.2 Creating *ITask* objects.

```
1  Short[][] around(Complex c0, double gap, /*...*/):
2      call(* mandelbrot.MandServer.mandelBrot(..) &&
          /*...*/ {
3
4      /* other statements */
5
6      MandData md = new MandData(c0, gap, /*...*/);
7      MandTask mt = new MandTask(md);
8
9      /* submitting tasks and returning values */
10 }
```

4.3.2 *IService*

Each different execution service for the framework must implement the interface *IService*. This interface exposes the method *run*, to accept one task for execution and the method *done* with which the dispatcher signals the end of the submissions for this set of tasks. The core implementation of the framework provides sequential, Java ThreadPool and Remote Method Invocation (RMI) based implementations. Algorithms 4.3 and 4.4 shows how the framework components interact with *IService* interface. Algorithm 4.5 shows an actual implementation of a such service.

Algorithm 4.3 Creating and registering *IService* instances.

```
1 before() : execution(public static void *.main(..)){
2     IService s = new MultiThread();
3     dispatcher.registerService(s);
4 }
```

Algorithm 4.4 Using *IService* to run tasks.

```
1 public class FIFODispatcher extends
     AbstractDispatcher {
2     /* ... */
3     public void runTasks(){
4         for(ITask<?,?> t : this.tasks){
5             /* housekeeping */
6             this.getAvailableService().run(t);
7         }
8     }
9 }
```

Algorithm 4.5 Multithread *IService* running a task.

```
1  public class MultiThread implements IService {
2
3    public void run(ITask<?,?> task) {
4      Runnable r = new Runnable() {
5        public void run() {
6          task.state = TaskStateEnum.Running;
7          task.compute();
8          if(task.state ==TaskStateEnum.Running)
9            task.state = TaskStateEnum.Complete;
10       }
11     };
12     futures.add(executor.submit(r));
13   }
14 }
```

4.3.3 *AbstractDispatcher*

The Dispatcher is the core service responsible for accepting new tasks, schedule them for execution and gather the results. The abstract class does not impose any complexity. Management of available resources, like seen on algorithm 4.4 is optional (a naive implementation can send all the tasks to the first registered service).

The application entry point (or an aspect managing it) needs to create the appropriate dispatcher, perform the initial discovery of services and register them.

4.4 Creating Tasks

As stated on introduction of this chapter, the framework was intended to declare, in a non-invasive manner, the different tasks composing the application.

The basic steps are:

- Create a class to encapsulate all the input data

- Create a class extending *ITask* where the *compute()* method implements the computation code
- Instantiate data and task objects to submit to the dispatcher.

The following subsections describe alternatives to create such tasks.

4.4.1 Manual Creation

The manual approach is invasive but offers complete control on the creation of tasks, specially useful when dependencies between tasks are complex or the parallelization model does not follow a usual pattern.

On the Mandelbrot set example, the method *mandelBrot* belonging to the class *MandServer* would need to encapsulate its arguments on several objects (according to the data partitioning strategy) and create objects, initialized with those data objects, from a class extending *ITask* that implemented the old method body. The execution of the tasks should be accomplished by normal framework services.

4.4.2 *FrameworkAdapter*

The *FrameworkAdapter* follows the same concept as the adapter software design pattern [14, p. 139]. The requirement for the wanted non-invasiveness of the framework is only achieved if all the changes needed for the framework be able to compose with the application are done on its separate module. The *FrameworkAdapter*

This aspect can be divided in three parts:

- Framework initialization, by creating the dispatcher and registering services (as seen on algorithm 4.3);
- Definition of *ITask* subclass and data encapsulation class;
- Intercepting the normal control flow and binding the framework to it. In the Mandelbrot set example, it would be a pointcut capturing method *MandServer.mandelBrot* calls and its arguments, create the

required objects and submitting them to dispatcher service and waiting for results (see algorithm 4.2).

4.5 Pluggable Services

The framework services are loosely coupled. This enables that new aspects can be used to bind new services. This section discusses available and proposed plugins.

4.5.1 Monitor

The monitoring service allows to see the progress of the application. It counts how many tasks were created, how many are actively running as well how many faulty tasks were.

For this purpose, the monitor intercepts calls to task object constructors (`call(ITask+.new(..))`, task execution (`call(* IService.run(..))`) to build its statistics.

4.5.2 Parallellization

Parallelization is a family of services that can apply more complex parallelization patterns using an approach similar to the skeletons paradigm [10].

A parallelization service intercepts task creation events and splits the task into smaller ones according to the pattern it implements. For example, *ScatterGather* intercepts task creation and creates several tasks using a user provided *scatter* function to split data accordingly. When all tasks return their results, another user provided function, *gather*, combines the results into a final one that is returned transparently as the result of the first task. Other skeletons, like *Pipeline* or *Farm* are also available. To manage dependencies, this skeletons can be extended to implement the specific rules.

4.5.3 Load Distribution

On previous versions of the framework, the Load Distribution service changed the default round-robin scheduling strategy to a demand-driven one, sending new tasks to resources on demand.

On the last version, *AbstractDispatcher* was introduced, giving more freedom to concrete dispatchers to manage resources and implement any desired scheduling strategies. For example, *FIFODispatcher* chooses one of the resources that has it marked as free.

4.5.4 Fault Recovery

Sometimes computing elements fail. To minimize the impact, the fault recovery service implements a timeout mechanism (based on the average time needed per task so far) and removes services from the services list and submit the task it was computing again for execution.

It monitors the same join points as the monitor service (4.5.1). In fact, both services are implemented jointly (they can still be used separately).

4.5.5 Remote Executor

To explore multiple computers, essential if the application is to leverage the supercomputer environment, exists a small agent that executes on each machine and using RMI receives *ITask* objects, executes their *compute()* method locally and returns their result.

4.6 Grid

It is possible to execute a framework-enabled application on EGEE without further modifications. The traditional job submission methods can be used to start the application.

However, using the *AspectGrid* framework brings some immediate advantages:

- The plugins mentioned on section 4.5 are available;

- Having the code parallelized with JPPAL enables taking immediate advantage of possible multicore systems available on the grid-assigned cluster;

- Existing services can be used and custom ones can be easily written and added to the framework. Example: saving files to a storage element while the computation is running can be achieved by an aspect instead of modifying the application code.

Chapter 5

Conclusions

Because of its nature, Aspect Oriented Programming is a good paradigm to encapsulate some kinds of functionality. The methodology presented on this thesis exploits that for progressive evolution of scientific applications, namely by developing new ways to express parallelization concerns when adapting existing scientific applications to multicore and supercomputer systems.

The *JPPAL* library does not require previous knowledge of AOP or AspectJ to be able to use it, making a better choice than Java Threads for parallelization.

The *AspectGrid*framework is able to non-invasively, i.e., without changing the program flow, adapt a program to supercomputer environments. It is a lightweight design with minimalist base components and small interfaces, and its implementation is open making possible to create, enable and remove services as needed.

5.1 Limitations

While AspectJ is the tool that made this work possible, it imposed some limitations itself. As mentioned on sections 3.2.2.1 and 3.5, refactoring is needed in order to expose loops and its control variables to AspectJ pointcuts. While this in our opinion is a good thing (forces the parallelization to be present in the API), the scientist will perceive the refactoring as a burden.

Other limitation lies with the *AspectGrid* framework where the programmer must know how to interact with the cluster/grid job submission systems, namely, the startup script must request the needed resources and to ensure that the remote execution agents are started prior to the application.

The biggest limitation of this work is the Java-only approach. Applications written in C or Fortran do not benefit from this approach mainly because there is no AOP tool for Fortran, and the ones for C/C++ are not supported and mature as AspectJ is.

One mechanism that from our experience is useful to help parallelize legacy code is *ThreadLocal*. However, such mechanism is very difficult to implement due to several reasons. The first one is that we could not agree on what must be its behavior: assuming that a object field was annotated as being *ThreadLocal*, when should the thread local copies be created? Should the mechanism force the field type implement *Cloneable*? After leaving the parallel zone, how to reduce the value?

The second reason was AspectJ related. The only way to make the copies at the start of a Paralell zone is to use reflection and iterate over all fields and check if the annotation was applied. That is not acceptable if performance is a goal.

5.2 Future Work

The *JPPAL* library can be expanded to implement more mechanisms, including new ones introduced by OpenMP 3.0. The *AspectGrid* framework will benefit from more services in its portfolio, mainly grid-related ones and job submission interfaces.

Other line of research would be do fill the gap between Aspect Oriented Programming and other languages than Java, should it be by new AOP tools (language independent AOP would be a possibility), tools for automatic conversion from that languages to Java or doing a major refactoring to the C/Fortran code, creating a shared library implementing the needed algorithms while rewriting the application control flow in Java and calling the library functions using Java Native Interface (JNI).

Bibliography

[1] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer, "The grid application toolkit: Toward generic and easy application programming interfaces for the grid," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 534–550, March 2005.

[2] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, *Grid Computing: Software Environments and Tools*. Springer-Verlag, January 2006, ch. Programming, Deploying, Composing, for the Grid.

[3] P. V. Bangalore, "Generating parallel applications for distributed memory systems using aspects, components, and patterns," in *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. New York, NY, USA: ACM, 2007, p. 3.

[4] N. Belblidia, M. Debbabi, A. Hanna, and Z. Yang, "AOP extension for security testing of programs," in *Proc. Canadian Conference on Electrical and Computer Engineering CCECE '06*, 2006, pp. 647–650.

[5] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. Java grande. [Online]. Available: http://www.epcc.ed.ac.uk/projects/archive/java-grande

[6] J. Bull, M. Westhead, M. Kambites, and J. Obdrvzalek, "Towards OpenMP for Java," in *European Workshop on OpenMP (EWOMP 2000)*, 2000.

[7] C. Cunha, J. Sobral, and M. Monteiro, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms," in *Proceedings of the 5th international conference on Aspect-oriented software development*. ACM New York, NY, USA, 2006, pp. 134–145.

[8] C. Dangelmayr and W. Blochinger, "Aspect-oriented component assembly - a case study in parallel software," *Software: Practice and Experience*, vol. 39, pp. 807–832, 2009.

[9] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk, "Gemlca: Running legacy code applications as grid services," *Journal of Grid Computing*, vol. 3, no. 1, pp. 75–90, 2005.

[10] J. F. Ferreira, J. L. Sobral, and A. J. Proenca, "Jaskel: A java skeleton-based framework for structured cluster and grid computing," in *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 301–304.

[11] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Workshop on Advanced Separation of Concerns, OOPSLA 2000, October, Minneapolis*, 2000.

[12] D. P. Fletcher, F. Akkawi, D. P. Duncavage, and R. Alena, "From research to operations: integrating components with an aspect-oriented framework and ontology," in *Proc. IEEE Aerospace Conference*, vol. 5, 2004, pp. –3078 Vol.5.

[13] I. Foster and C. Kesselman, Eds., *The Grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] R. C. Gonçalves and J. L. Sobral, "Pluggable parallelisation," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing.* New York, NY, USA: ACM, 2009, pp. 11–20.

[16] S. Gorlatch and J. Dünnweber, *Future Generation Grids.* Springer-Verlag, 2006, ch. From Grid Middleware to Grid Applications: Bridging the Gap with Hocs, pp. 241–261.

[17] GridGain Technologies. [Online]. Available: http://www.gridgain.com

[18] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA: ACM, 2002, pp. 161–173.

[19] B. Harbulot and J. R. Gurd, "Using aspectj to separate concerns in parallel scientific java code," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development.* New York, NY, USA: ACM, 2004, pp. 122–131.

[20] G. Hermosillo, R. Gomez, L. Seinturier, and L. Duchien, "Aprosec: an aspect for programming secure web applications," in *Proc. Second International Conference on Availability, Reliability and Security ARES 2007*, 2007, pp. 1026–1033.

[21] Q.-T. Ho, T. Hung, W. Jie, H.-M. Chan, E. Sindhu, G. Subramaniam, T. Zang, and X. Li, "Grasg - a framework for "gridifying" and running applications on service-oriented grids," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, May 2006, pp. 4 pp.–.

[22] T. Ishio, S. Kusumoto, and K. Inoue, "Program slicing tool for effective software evolution using aspect-oriented technique," in *Proc. Sixth International Workshop on Principles of Software Evolution*, 2003, pp. 3–12.

[23] B. Jones, "An overview of the egee project," *Lecture Notes in Computer Science*, vol. 3664, pp. 1–8, 2005.

[24] C. Kaewkasi and J. R. Gurd, "A distributed dynamic aspect machine for scientific software development," in *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*. New York, NY, USA: ACM, 2007, p. 3.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of aspectj," in *Proceedings of ECOOP, Springer Verlag. LNCS*, vol. 2072, 2001.

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97–11th European Conference on Object-Oriented Programming, Finland, June 9-13*, 1997.

[27] M. Klemm, R. Veldema, M. Bezold, and M. Philippsen, "A proposal for openmp for java," *Lecture Notes in Computer Science*, vol. 4315, p. 409, 2008.

[28] K. Koch. Roadrunner platform overview. Los Alamos National Laboratory. Retrieved on July 2, 2009. [Online]. Available: http://www.lanl.gov/orgs/hpc/roadrunner/

[29] C. Koppen and M. Störzer, "Pcdiff: Attacking the fragile pointcut problem," in *First European interactive workshop on aspects in software (EIWAS)*, 2004.

[30] E. Lakshika, C. Keppitiyagama, and D. Wathugala, "AOnesC: An Aspect-Oriented Extension to nesC," in *Proc. NTMS '08. New Technologies, Mobility and Security*, 2008, pp. 1–5.

[31] M. Lamanna, "The lhc computing grid project at cern," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 534, no. 1-2, pp. 1

– 6, 2004, proceedings of the IXth International Workshop on Advanced Computing and Analysis Techniques in Physics Research.

[32] P. H. M. Maia, N. C. Mendonça, V. Furtado, W. Cirne, and K. Saikoski, "A process for separation of crosscutting grid concerns," in *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing.* New York, NY, USA: ACM, 2006, pp. 1569–1574.

[33] C. Mateos, A. Zunino, and M. Campo, "A survey on approaches to gridification," *Software: Practice and Experience*, vol. 38, pp. 523–556, 2008.

[34] M. P. Monteiro and J. M. Fernandes, "Towards a catalog of aspect-oriented refactorings," in *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development.* New York, NY, USA: ACM, 2005, pp. 111–122.

[35] R. V. V. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal, "Ibis: a flexible and efficient java-based grid programming environment. concurrency & computation: Practice & experience," in *Concurrency & Computation: Practice & Experience*, 2005, pp. 7–8.

[36] J. Park, S. Kim, and S. Hong, "Weaving aspects into real-time operating system design using object-oriented model transformation," in *Proc. Ninth IEEE International Workshop on WORDS 2003 Fall Object-Oriented Real-Time Dependable Systems*, 2003, pp. 292–298.

[37] K. Richey. (1997, February) The eniac. Retrieved on June 25, 2009. [Online]. Available: http://ei.cs.vt.edu/~history/ENIAC.Richey.HTML

[38] J. L. Sobral and A. J. Proenca, "Enabling jaskel skeletons for clusters and computational grids," in *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 365–371.

[39] J. Sobral, "Incrementally developing parallel applications with aspectj," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 10.

[40] J. L. Sobral, C. A. Cunha, and M. P. Monteiro, "Aspect oriented pluggable support for parallel computing," in *High Performance Computing for Computational Science - VECPAR 2006*.

[41] J. L. Sobral and M. P. Monteiro, "A domain-specific language for parallel and grid computing," in *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*. New York, NY, USA: ACM, 2008, pp. 1–4.

[42] J. L. Sobral, "Pluggable grid services," in *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 113–120.

[43] E. Sousa, R. Gonçalves, D. Neves, and J. Sobral, "Non-invasive gridification through an aspect-oriented approach," in *2nd Iberian Grid Infrastrucutre Conference (Ibergrid 2008)*, Porto, Portugal, 2008.

[44] E. Sousa and M. P. Monteiro, "Implementing design patterns in caesarj: an exploratory study," in *SPLAT '08: Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*. New York, NY, USA: ACM, 2008, pp. 1–6.

[45] T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake, and C. Packer, "Beowulf: A parallel workstation for scientific computation," in *In Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[46] D. Webb and A. L. Wendelborn, *Computational Science - ICCS 2003*, 2003, ch. The PAGIS Grid Application Environment, pp. 692–693.