Universidade do Minho
Escola de Engenharia

Matheus Barros Almeida

Separation of Concerns in Parallel Programming using Feature-Oriented Programming

Outubro de 2010

Universidade do Minho
Escola de Engenharia

Matheus Barros Almeida

Separation of Concerns in Parallel Programming using Feature-Oriented Programming

Master in Informatics

Supervisor :
Doctor João Luís Ferreira Sobral

Outubro de 2010

Acknowledgements

First of all, I would like to express my deep and sincere gratitude to my supervisor, Professor João Luís Sobral for accepting me in his research group. I want to thank all the hours/days/weeks he spent reviewing my work (including this one). The past year was a really great experience and, without him, this work would never exist.

I want to give a special thanks to all member of the Professor João Luís Sobral research group. Rui Gonçalves for helping me at the beginning with AspectJ and latter with his expertise in model driven development; Edgar Sousa for being an incredible *all arounder*; Diogo Neves for helping me to grow in presentations and Jorge Pinho with his solid knowledge about the JECoLi. Last but not least, I want to thank my friend Rui Sabino for the tremendous number of hours we spent in the lab and his interest about the work I was doing.

To my friends Pedro Gomes, Emanuel and Rui Sabino (again) for being always there. We created great habits(lunch and coffee break) and I'm sure I'll miss it. It was a great year. To my roommates Nelson and Ricardo for their support and help.

To my parents, Manuel and Isabel for being the best parents in the world.

To Ana, for being the love of my life.

Finally, I want to thank the financial support of this project : Parallel Refinements for Irregular Applications (UTAustin/CA/0056/2008) funded by Portuguese FCT and European funds.

Abstract

With the mainstream commercialization of parallel processors and current clusters having hundreds of thousands of computing units, parallel programming is still seen as a complex and expensive solution used mainly by scientific projects. We address these issues by reinforcing the idea that parallel programs present, in general, lack of modularity. This dissertation presents the study of Feature Oriented Programming (FOP) as an alternative to Aspect Oriented Programming (AOP) in the development of parallel programs to promote modular and incremental development. In our approach, each parallelization feature encompasses a set of modifications to the domain code in the form of class refinements, i.e., new unit of modularity that encapsulates a well defined parallelization concern.

We have successfully applied our approach to several case studies including a medium-sized Object-Oriented framework of the Biological computational field. As a result, we managed to enhance the framework's performance by introducing the parallelization in an external compositional step. This step allows us to bind parallelization features into domain specific code to match different target parallel platforms. Several benefits arise from our approach including no impact in the evolution of the framework to cope with new algorithms as well as the greater modularization of the parallelization concerns when compared with traditional parallel programming techniques. The overhead introduced by this loosely coupled development in terms of performance is minimum as shown by *low-level* benchmarks and several case studies.

Resumo

Com a vasta comercialização de processadores paralelos e clusters actuais com centenas de milhares de unidades computacionais, a programação paralela ainda é vista como uma solução complexa e dispendiosa usada principalmente em projectos científicos. Esta dissertação aborda esses problemas reforçando a ideia que os programas paralelos apresentam, geralmente, falta de modularidade. Apresentamos o estudo da Programação Orientada às Funcionalidades (POF) como uma alternativa à programação Orientada aos Aspectos (POA) no desenvolvimento de programas paralelos de modo a permitir um desenvolvimento modular e incremental. Na abordagem proposta, cada funcionalidade de paralelização engloba um conjunto de modificações ao código do domínio na forma de refinamentos de classes, i.e., uma nova unidade de modularização que encapsula uma funcionalidade de paralelização bem definida.

Aplicamos com sucesso a nossa abordagem a vários casos de estudo incluindo uma *framework* orientada aos objectos de tamanho médio pertencente ao campo da computação biológica. Como resultado, foi possível melhorar a performance da *framework* ao introduzir a paralelização num passo externo de composição. Este passo permite-nos *colar* as funcionalidades de paralelização em código específico do domínio de modo a utilizar diferentes plataformas paralelas de execução. Esta abordagem trouxe vários benefícios entre os quais a melhor compatibilidade com evolução da *framework* de modo a acrescentar novos algoritmos e também na modularização das funcionalidades paralelas. Com este desenvolvimento desacoplado, a perda em termos de desempenho é mínima como comprovado pela implementação de testes de *baixo-nível* e de outros casos de estudo.

List of Figures

List of Tables

List of Programs

Contents

Chapter 1

Introduction

1.1 Context

For many years, one of the main characteristic of a Central Processing Unit (CPU) was its frequency. The major advantage of higher frequencies is to increase the throughput of instructions and, as a general rule, each new generation of processors transparently increased application's performance. Two main problems arose from this approach [Koc05]:

1. others components have not followed this development (eg.: early 1990s, the number of required clock cycles to access the main memory was 6 to 8 and by 2005, that number grew 20x, 224 clocks);

2. the increase of power consumption is proportional to the clock frequency and this leads to heating problems.

The solution adopted to those problems is to integrate into a single CPU a set of independent processing units (*cores*). With this approach, processor's designers no longer need to raise clock frequencies to increase computational power.

The older variant of parallel computing but still very important today is related to distributed computing (e.g.: Cluster) where the computation is spread across a number of nodes connected by a high performance network. The most important benefits of this approach are:

1. large number of nodes that can be interconnected;

2. the nodes can be composed by commodity hardware making it a low cost solution.

Both multi-core and cluster computing require a different programming style, as programmers need to specify parallel activities within applications. Thus, the development of parallel applications requires both knowledge of traditional programming and expertise in parallel execution concerns (e.g.: data partition, thread/process communication and synchronization). Generally, these two concerns are mixed because the code that supports the parallel execution is *injected* into the base functionality, resulting in *tangled code*. Also, the lack of structure of this approach leads to scattered code since the code to enable parallel execution is spread over different places of the base code. Program 1 illustrates the problem of invasive modification and *tangling* by showing the simplified cluster oriented parallelization of a molecular dynamics simulation [SBO01]. In black it can be seen the base code and in red (italic) the parallelization statements.

The problems of *tangled code* manifests in a variety of forms. Programmers may need to adapt or evolve the base code to cope with new requisites or to improve the base code's performance. This evolution can be compromised since many parallelization features requires structural changes to the base code. *Isolate* the base code from the parallelization concern is most of the times not a *trivial* task. The disadvantages of *tangled code* are not only at the base code. The parallelization of an algorithm may use different approaches/models depending on the domain of the problem and the algorithm's codification. Since there is no catalogue with all parallelization approaches, *tangling* prevents the use of different parallelization models(e.g.: *Heartbeat*, *Divide and Conquer*). Consequently, it also prevents the exploitation of different parallel platforms since the algorithm's codification specifies the mapping to the parallel platform.

1.2 Motivation and Objectives

Previous studies [GS09, Sob06] argue that the separation of the base functionality from the parallelization structure allows :

1. better maintenance and reuse of the core functionality, reducing or eliminating the problem of *code tangling* and *scattering*;

2. easier understanding of the parallel structure and better reuse of the parallel code;

3. enhancement of the parallelization structure by promoting incremental development.

```
public class MD {
   Particle [] one; // Vector with all particles
   int mdsize; // Problem size (number of particles)
   int movemx; // Number of interactions

    //Declare auxiliary variables to MPI parallelization
    double [] tmp_xforce;
    double [] tmp_yforce;
    double [] tmp_zforce;
    ...
    public void runiters throws MPIException {

       for (move = 0; move < movemx; move++) { // Main loop
          for (i = 0; i < mdsize; i++) {
             one[i].domove(side); // move the particles and
          }                       // update velocities
      ...
      MPI.COMM_WORLD.Barrier();
      computeForces(MPI.COMM_WORLD.Rank(),MPI.COMM_WORLD.Size());
      MPI.COMM_WORLD.Barrier();
      for (i = 0; i < mdsize; i++) {  //Copy forces to temp vector
        tmp_xforce[i] = one[i].xforce;  // to use in MPI operation
        tmp_yforce[i] = one[i].yforce;
        tmp_zforce[i] = one[i].zforce;
      }
      //Global reduction
      MPI.Allreduce(tmp_xforce,0,tmp_xforce,0,mdsize,MPI.DOUBLE,MPI.SUM);
      MPI.Allreduce(tmp_yforce,0,tmp_yforce,0,mdsize,MPI.DOUBLE,MPI.SUM);
      MPI.Allreduce(tmp_zforce,0,tmp_zforce,0,mdsize,MPI.DOUBLE,MPI.SUM);

      //Update forces based in reducted values
      //Scale forces and calculate velocity
    }
```

Program 1: MD cluster based parallelization.

Aspect Oriented Programming (AOP) is a programming paradigm that aims to modularize cross-cutting concerns [KLM+97]. It introduces a new unit of modularization (aspect) that encompasses code that otherwise would be scattered among the entities of the problem. It was used successfully to separate parallelization concerns from the domain code [HG04, GS09]. In this approach, the parallelization features are coded in aspects following the principles of the *join-point* model. The AOP weaving mechanism is responsible to merge the parallelization features and the base code. The experience gained with the implementation of several case studies [Sob06, GS09] using AOP was the main motivation for this study. We argue that only a small subset of AOP was used and that subset can be replaced by a less complex mechanism based on Object-Orientated programming.

Feature Oriented Programming (FOP) is a programming paradigm that aims to deal with the lack of software reuse and customizability [Pre97]. It promotes incremental design by defining that programs must be built by choosing different features from a *feature repository*, i.e., collection of functionalities that can be applied to any program. Our purpose is to investigate the use of FOP to build parallel programs by composing parallelization features. As an alternative to the AOP approach, programmers introduce parallelization features following the same principles of Object-Oriented inheritance mechanisms with the difference that each feature encompasses a set of program *increments*. We explore an approach where each *increment* is implemented with Class Refinements, i.e., class that introduces new sate and behaviour to another class in a rewriting mechanism [DBS03, SB02, DBR04, BDN05, NC08].

We argue that given the heterogeneity of parallel architectures and the composition step provided by FOP, more complex parallel programs can be synthesized as simply as composing feature's layers. We expect several key benefits of this approach :

- Ease the migration of programmers because the rules and abstractions applied are similar as the ones found in Object-Oriented programming;
- Incremental development of parallelization features. More complex features can be developed by the composition of simple ones (e.g.: hybrid models);
- Independent development both of the base code and the parallelization features.

1.3 Thesis' Outline

This document is structured as follows. Chapter 2 presents the traditional methodologies, main libraries and parallel languages used to develop shared and distributed memory parallel applications. The main focus is to discuss their ability to allow incremental development and separation of concerns. Chapter 3 focuses in published research regarding the use of separation of concerns that can be applied to parallel programming. Aspect Oriented Programming, Feature Oriented Programming and tools implementing those concepts will be discussed. We present a parallelization model and its application to several case studies in chapter 4. Chapter 5 compares our FOP based approach against the introduction of parallelization using AOP. The conclusion of this work and the future lines of research are presented in chapter 6.

Chapter 2

Parallel Computing

Parallel computing is becoming increasingly important for obtaining high performance from a computer system. The number of processing cores is expected to double every eighteen months [AL07] and advances in the semiconductor industry allows the development of smaller *dies*[1] that allows, for instance, the reduction of energy consumption in newer generation of processors. One of the great benefits is the increase number of nodes in computing clusters[2].

Parallelization is the process of decomposing large computations (e.g.: iteration loops) by several *entities* so that each *entity* computes a subset of the problem concurrently with all others. The main purpose is to reduce the computing time. It can be implemented in two distinct programming models. Shared memory parallelization is implemented *via* thread-level parallelism where the communication among threads is done by shared data structures. Distributed memory parallelization is characterized by process-level parallelism where a process is an entity with its own physical memory space and communication is done by message passing. The popularity of computing clusters composed by multi-core processors motivated the use of hybrid models that are a mixture of shared with distributed memory model. This kind of model can better conciliate the advantage of the locality of data and low-latency communication between cores in a processor with the larger number of nodes supported in a distributed memory architecture.

The study of fully automatic parallelization using compilers is far from an efficient technique to introduce parallelization mainly because compilers have a limited scope. Most of the parallelization is done by modifying the algorithm's structure and that kind of parallelism can only be introduced by

[1]Term used in integrated circuits to refer to blocks of semiconducting material.

[2]Modern clusters have a few thousand nodes with hundreds of thousands of processing *cores*.

the programmer. For instance, the solution generated by parallel algorithms can be at most an approximation comparing with the solution generated by the sequential algorithm. This happens because some algorithms have parallelization blockers, i.e., the algorithm's codification does not allow parallel execution. One example is a dependency between iterations in a loop where the computation of the next iteration depends on the previous. A solution for such problem can force the relaxation of the algorithm. This is strictly a programmer's decision.

In the next subsections we introduce mainstream parallel programming models, libraries and languages that represent an efficient solution over the traditional approach by hiding most of the underlying complexity (e.g.: communication issues or thread/process creation) and offering a good set of abstractions for typical problems.

2.1 Thread-based parallelism

A natural way to implement a parallel program is to use the programming language mechanisms to support, for instance, multi-threading. Thus, thread creation and synchronization mechanisms are examples of tasks that have to be explicitly used or implemented. Current specification of the Java language defines two standard ways to specify a new execution flow (thread) :

- **Runnable Interface** : this interface only defines one method with the signature *public void run()*. Implementations of this method correspond to the code that is executed in the new execution flow;
- **Thread extension** : Classes extend the class *Java.lang.Thread* and override the method *public void run()*. This class already implements the interface Runnable. Spawning a thread is made by executing the method *start* in the subclass object.

Program 2 illustrates the thread creation mechanism using a *Runnable* object. The object *t* is created using an anonymous *Runnable* object passed to a Thread constructor. The method *start()* defines the start point where the new thread is launched. There are other more advanced mechanisms to manipulate thread creation. For instance, in a program that heavily relies on thread creation and destruction, the use of *Executors* and *Thread Pools* can be a great benefit. A more detailed view can be found at [GP].

Modern programming languages support multi-thread and offer abstractions to manipulate threads. The traditional approach to parallelization mixes the previous artefacts that manipulate execution flows with the base

```
Thread t = new Thread(new Runnable(){
  @Override
  public void run(){
    //method body
  }

});

t.start();
```

Program 2: Java Thread example.

code. Once the domain code is populated with artefacts regarding the parallelization concerns, modularity is lost. This doesn't allow, for instance, to change the parallelization to match other target platforms (e.g.: Distributed Memory) or to perform incremental development to enhance the parallelization or the domain code. Both codes are glued and dependent on each other.

2.2 OpenMP

Open Multi-Processing (OpenMP)[CDK$^+$01, CJP07] is a standard multi-platform model for programming multi-threaded applications in a shared memory architecture in C, C++ and Fortran. It offers an Application Programming Interface (API) that consists in a set of compiler directives (annotations) and a supporting library of subroutines. Sections of the source code are annotated using OpenMP compiler directives.

The abstraction provided by OpenMP hides different levels of complexity ranging from thread creation, communication and synchronization to the use of work-sharing constructs. In a more detailed view, program execution starts with a single thread (master thread) and this behaviour only changes when the execution flow finds a parallel region. A parallel region is a delimited block of code that will be executed by a team of threads where each thread has an unique identification (master-thread has *id* 0). To divide the work among threads, OpenMP provides the *for* work-sharing construct that splits the computation in a *for-loop* depending on the schedule policy [CDK$^+$01]. For instance, iterations can be divided equally by all threads (static schedule) or dynamically where iterations are assigned to threads as they request them [3]. Synchronization mechanism are also provided inside parallel regions

[3]A *chunk_size* parameter can be used to define the minimum number of iterations each thread computes.

to control the parallel execution. Definition of an ordered execution (*omp ordered*) restricts an arbitrary block of code inside a *for* work-sharing construct to be executed sequentially and ordered based on the id of each thread; *omp single* restricts the execution of a block of code to only one thread (any thread) or only the master thread (*omp master*). Mutual exclusion and barrier synchronization is achieved by *omp critical* and *omp barrier*, respectively.

In Program 3 is shown an OpenMP example for a C program. The *omp parallel* creates a parallel region with an arbitrary number of *workers* (by default it is the number of cores in a CPU) and the *omp for* splits the computation dynamically among all workers. The second parameter is related to the *for* scheduling. In this case, the value 2 means that each thread computes two iterations at a time before requesting more computation (dynamic schedule).

```
#pragma omp parallel
{
  #pragma omp for schedule(dynamic,2)
  {
  for(int i = 0; i < N; i++)
  //Some computation ...

  }
}
```

Program 3: OpenMP example.

The generic solution provided by the *for construct* is insufficient to solve all scheduling policies. Sometimes it is necessary to use each thread identification to access data based in pre-computed indices to specify how loop iterations are assigned to workers. Program 4 illustrates this problem by showing an example of the implementation of a sparse matrix-matrix multiply. The arrays *base* and *limit* contain, for each index, a start and a finish position, respectively, that corresponds to indexes in two arrays: *row* and *col*. Each thread executes a subset of the iterations of the *for loop* based in the pre-computed indexes that are defined in the arrays *base* and limit, i.e., *base[i]* and *limit[i]* contain the start and finish positions for the thread with id *i*.

There are attempts to recreate an OpenMP interface for Java. JOMP [BK00] was the first attempt and it was seen as a source code transformation tool because in order to use it, standard Java comments are used. Other approach [KVBP08] extended the previous work and introduced new mechanisms and thus took advantages of Object-Orientation and Java concurrency

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  for(int i = base[id]; i < limit[id]; i++)
    y[ row[i] ] += x[ col[i] ] * val[i];

}
```

Program 4: OpenMP example with manual loop partition.

libraries. JPPAL [Sou09] is a library that implements the most common OpenMP mechanisms (e.g.: parallel section and parallel for). It takes advantage of Java annotations to indicate methods that can be potentially used to introduce parallelization.

Although OpenMP has been gaining popularization and it is the *de-facto standard* [CDK+01] for multi-threaded programming, it does not solve the problem of separation of concerns in parallel applications [HG04]. Since OpenMP directives are mixed with the base code, the parallelization feature created has to be spread over and repeated for different entities of the problem. An example is the problem presented in program 4 where, instead of refining the default scheduling policy, OpenMP forces the programmer to use a different construct. This results in invasive modification of the base code. This increases the problem of *code scattering* because changes in data partition or loop scheduling, for instance, have to be made in all affected entities of the problem that uses OpenMP directives. For larger problems, with the need to use class inheritance, for instance, solutions like the JPPAL library that relies in Java annotations suffer from the disadvantage of derived class do not inherit the annotations used in classes higher in the class hierarchy.

2.3 MPI

Message Passing Interface (MPI[4]) is an API for high performance communication. It is used to express parallelization concerns in distributed memory environments (e.g.: Cluster) using the Single Process Multiple Data (SPMD) technique. This programming model is based in the concept of multiple instances of the same program executing in parallel, where each instance is a process with its own address space and a unique identification. Hence, on an MPI program, each process manipulates its own data and communication

[4]See http://www.mpi-forum.org

with other processes (eg.: data exchange or synchronization) is done by interchanging messages (also called two-side communication) that circulates in high performance networks (e.g.: *Myrinet* or *Infiniband*).

Implementations of MPI are available for a different number of computer languages (e.g.: C, C++, Fortran and Java) despite variations of performance [JCSG99]. Program 5 shows an example (C language) of the skeleton of an MPI program. Lines 1 and 8 define a section that will be executed by all processes. Each process can obtain its own unique identification and the computation can be split based on the identification number. This is an example that illustrates the use of the SPMD technique.

MPI processes communicate by exchanging messages (eg.: MPI_Send and MPI_Receive). Messages can be sent directly to other processes based on the identification number of the destination process or they can be sent to all processes in an efficient way (e.g.: MPI_Broadcast). Processes can be organized into communication groups (ex.: MPI_COMM_WORLD) for reducing the traffic in the network and provide an efficient and clean abstraction.

To help the communication protocol, each message contains a *tag* and, for instance, processes can block waiting for a message with a specific tag. In program 5, line 1 marks the beginning of the MPI program that is executed by all instances. The process with $Id = 0$ waits until it receives one message from every other process (lines 2 to 4; for simplicity, it is assumed any message). The others processes do some work and when they finish, each one sends a message to the process with $Id = 0$ (lines 5 to 7).

```
1 MPI_Init(&argc, &argv);
2 if(myId == 0) {
3    while(cont < numberOfProcesses -1)
4     cont++; MPI_Receive(...); }
5 else{
6    dosomework();
7    MPI_Send(...) };
8 MPI_Finalize();
```

Program 5: MPI example.

MPI programs are used to solve large scale problems and to take advantage of the great number nodes in computing clusters. Recent MPI implementation can even use low-latency communication via Operating Systems (OS) when processes are running in different cores of the same processor.

In terms of separation of concerns, once the code is populated by MPI instructions, it is hard to reuse or even understand the sequential code because of the changes that are necessary to cope with the distributed memory paradigm. For instance, code to express data partition, synchronization mechanisms and data exchange is mixed with the base code. Moreover, reusing the parallel feature is also hard because the code is written *all-at-once.*

2.4 Parallel Languages with PGAS

Several parallel languages have been developed over the years with different purposes. A great number of those languages is used to program in the SPMD model and offer abstractions over creation of processes and communication concerns. From these languages, a subset of them is gaining popularity because they rely on the Partitioned Global Address Space (PGAS) model. This model, unlike the traditional distributed memory model presented in the previous subsection, offer a global view of memory to the programmer. This global address space abstracts the programmer from thinking on distributed processes computing on local data and communicating with message passing. The set of languages designated by PGAS languages simplifies the programming task by offering a *shared memory* abstraction to distributed memory programming. Access to non-local memory references is dealt transparently by the compiler by generating all communication code to data exchange between distributed processes. Some relevant languages implementing this model are :

Co-Array Fortran (CAF) [NR98]. It was the first PGAS language and it is current part of the Fortran specification. The solution adopted by the CAF developers is to give array data structures a new *dimension* (i.e.: co-array). This new dimension can be seen as an array where programmers have the ability to access different copies of the same data structure (i.e.: different position in the co-array dimension) but in other distributed processes. Communication code is automatically generated by the compiler. This is a simple and elegant solution to abstract over processes' communication.

Unified Parallel C (UPC) [CDC+99]. It is an explicit parallel extension to ISO C developed by a consortium of vendors and universities. It defines two types of memory accesses. Shared memory refers to the global address space (i.e.: distributed data) and private memory is local to each thread [5]. UPC creates a new semantic to C *pointers.*

[5]Note that each thread has a segment of shared memory that is *private* because it was allocated by the thread process

Pointers to data structures can be defined shared or private. A shared *pointer* can reference memory only in the shared space by a given data-thread affinity. A private pointer can reference data in the private memory or in the bounds of shared memory in each thread. UPC also provides global operations like the *upc_forall* where the computation of a loop is divided by all the distributed processes. UPC abstracts the programmer from the communication task (generated by the compiler) between program instances relying in a *disguised* but simple shared memory paradigm.

Titanium [YSP+98]. It is an extension to Java that is translated to C so there are some Java mechanisms not available because of there is no Java Virtual Machine (JVM). It has a memory model similar to UPC and offers a large set of global synchronization and computation operations over partitioned data. One of those mechanisms is the *foreach* loop that splits the computation among all processing entities (i.e.: distributed processes) and all the communication code is dealt by the compiler.

PGAS languages offer an abstraction over communication and data distribution in a higher level compared with traditional languages. They offer a *shared memory environment* to develop large scalable applications in distributed memory environments. Communication is handled by compilers and these languages use a model called one-side communication. In simple terms, implementations like GASnet [Bon02] offer an API to handle communication where just one entity makes part in the communication operation (i.e.: one-side). The other entity is passive and memory access in transparently held by the communication layer.

Although parallel languages are evolving and including new abstractions, the abstractions they provide do not include separation from the base code and the parallelization itself. More particularly, parallel languages have an intrinsic disadvantage. They tend to inhibit programmers that are used to mainstream languages because the need to learn a new memory model, new abstractions and generally a new syntax. Moreover, parallel programs are all about performance and most of the times, *tuning* a program is a difficult task without a great knowledge of the language, i.e., how the code is generated and what are the most efficient constructs. Thus, high level languages suffer from tangling when the programmer wants to fine-tune performance for his specific problem. This, when possible, usually implies resorting to low level mechanisms, introducing tangling as it was shown in the OpenMP example in program 4. One of the aims of this study is to improve modularity in the development of parallel programs in a mainstream Object-Oriented language by moving all performance related concerns into new units of modularity.

Chapter 3

Separation of Concerns

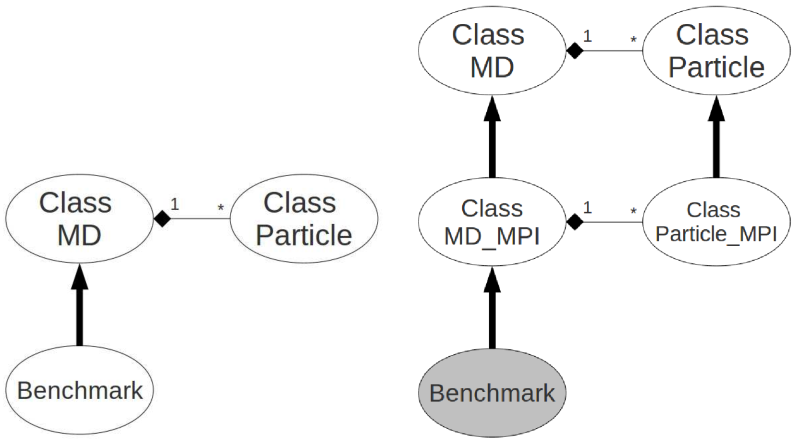
Early works studied the benefits of modularization in software development [Par72, Dij78, Par79]. Properties like maintainability, reusability and extension are very important and not always achieved even in modern programming paradigms (e.g.: Object-Oriented Programming (OOP)).

Inheritance in Object-Oriented Programming is a fundamental relationship mechanism to promote reuse of basic entities while encapsulating the implementation of common functionalities. Classes inherit functionality and state from a single parent class (Single Inheritance) or multiple parents (Multiple Inheritance). In the remainder of this dissertation, we refer to inheritance in OOP as the implementation of single inheritance where subclasses specialize the behavior of the parent class, i.e., subclasses can add or modify inherited behaviour from parent classes[1]. This specialization relationship is static because the relation between parent classes and subclasses is *glued* in the subclass definition. This creates a chain or class hierarchy where subclasses can refer to their parents by using *super*. The class hierarchy does not work in the other way, i.e., parent classes have no information about their respective subclasses. This mechanism is relatively simple compared with multiple-inheritance in C++ and other mechanisms like CLOS [BC90] where classes need to invoke a special function name *call-next-method* to visit other classes in the inheritance chain.

Object-Oriented inheritance has been extensively used in OO applications but it fails to cope with variability of program increments. Each increment extends a previous entity (parent class) and can define new state and behaviour. Thus, program increments are statically defined in a class hierarchy, i.e., the information about the parent class is *hard coded* in the subclass definition. Later, the access of the new behaviour or state defined in the subclass

[1]This inheritance mechanism is the one implemented in modern Object-Oriented languages like Java and C#.

has to be done explicitly by using the name of the subclass for instantiation. As an example of this problem, figure 3.1 shows the use of OO inheritance to separate the parallelization concern shown in program 1. Figure 3.1(a) represents the base class hierarchy where *Class MD* contains a set of references to *Particles*. The class *Benchmark* is responsible to instantiate the *Class MD* to configure the molecular dynamics simulation and to execute the method that starts the simulation. The use of traditional OO inheritance to encompass parallel execution in distributed memory (MPI) is shown in figure 3.1(b). The creation of two subclasses (*Class MD_MPI* and *Class Particle_MPI*) is needed to introduce the parallelization feature reusing the base functionality [AS10]. In order to use the MPI feature, the *class Benchmark* needs to be changed to instantiate the class *MD_MPI*. Clearly, this solution does not scale if we want to encapsulate several parallelization mechanisms to match different target platforms (e.g.: Cluster or Multi-core), each in its own module (e.g.: subclass) because it is required to make changes to client modules to compose them.



(a) Base class representation. (b) Extended class representation.

Figure 3.1: Incremental software design with Object-Oriented Inheritance.

This dissertation covers exclusively the study of methodologies and technologies that introduce modularization improvements in Object-Oriented programs. More specifically, we focus on the separation of the parallelization concern from already implemented algorithms. We argue that the code to enable parallel execution of an algorithm should be encapsulated in its own module. Ideally, modules encompasses platform specific mappings that can be composed to create more sophisticated parallelization models that can take advantages of the heterogeneity of some parallel architectures.

In the remainder of this chapter, we present different approaches that allow modularization improvements in Object-Oriented programs.

3.1 Aspect Oriented Programming

Object-oriented programming is a successful paradigm supported by most of today's modern languages. That success came mainly from the fact that the object model better suits real domain problems[KLM+97].

Aspect Oriented Programming[KLM+97] aims to separate and modularize cross-cutting concerns that are not well captured by the Object-Oriented (OO) approach. Aspects are units of modularization that encapsulate code that otherwise would be scattered and tangled with the base code.

Cross-cutting concerns can be either static or dynamic. Static cross-cutting allows the redefinition of the static structure of a type hierarchy. For instance, fields of a class can be added or an arbitrary interface can be implemented. For dynamic cross-cutting, AOP introduces the concepts of *join point* and *advice*. Join point is a well defined place in the base code where arbitrary behaviour can be attached and advice is the piece of code to execute when a join point is reached. A set of join points can be defined by the use of the *pointcut* designator that allow the use of logical operators to define set of points in the program flow.

Weaving is the mechanism responsible to compile the aspects. It will merge both the aspects and the base classes. This process can be done either in the compilation phase or during class loading.

In program 6 is shown an example of AOP with AspectJ. This aspect shows the use of dynamic cross-cutting by tracing calls to the method *Deposit* defined in *Bank* class that have one argument of type *int* (line 3). This line corresponds to the definition of a *poincut.* Before method calls to *Bank.Deposit* (line 5), a piece of *advice* is executed. Lines 6 and 7 correspond to the advice.

3.1.1 AOP implementations

There are many AOP implementations for a large number of programming languages. We will list only the main AOP implementations[2] for the most common Object-Oriented languages.

[2]JBoss AOP and Spring AOP are important implementations but its use goes beyond the scope of this study.
A comparison of AOP implementations can be found at http://www.ibm.com/developerworks/java/library/j-aopwork1/

```
public aspect Logging {
    int count = 0;
    pointcut deposit() : call (void Bank.Deposit(int));

    before() : deposit() {
        Logger.log(...);
        count++;
    }
}
```

Program 6: AOP logging example with AspectJ syntax.

- AspectJ[KHH+01] is the most mature and the most accepted AOP implementation.
- CaesarJ[AGMO06] tries to address some of the problems mentioned earlier that Object-oriented approaches do not suit. It is strongly focused on reusability but one of the main drawbacks[SM08] is the fact that CaesarJ does not support Java features introduced after the second main revision (Java 2);
- AspectC++ was developed with the aim of create the first aspect oriented extension for languages like C and C++. It was argued[SGSP02] that most embedded systems were being developed using languages like C++ because the lack of resources to deal with the overhead created by the Java runtime system. AspectC++ was based on AspectJ and shares the most part of the concepts introduced by the last one.

3.1.2 Parallelization using AOP

The main idea behind of the first work published about parallelization using AOP[HG04] was to use the potential provided by AOP to separate the mathematical model from scientific applications and the parallelism itself. That approach was applied to programs presented in the Java Grande Forum (JGF[3]) benchmark.

All algorithms requires different amounts of refactoring in the base code. That need is justified[HG04] from the facts:

1. The design of the application is a fundamental issue for the use of AOP. For instance, the use of an Object-oriented language does not mean that the program has a well defined Object-oriented structure;

[3]See http://www.epcc.ed.ac.uk/javagrande

2. The join point model offered by AspectJ does not support fine grain (e.g.: *instruction level*) join points. For instance, it is not possible to *intercept* iterations in a loop structure.

The work is also important because it proved that it is possible to separate the parallelization feature from the base code in scientific applications using AOP.

Another approach that used AOP with parallel applications defined a methodology that allow a better modularization of parallel programs and enables a more incremental application development[Sob06]. It was postulated that parallelization concerns can be divided into four categories : function or/and data partition, concurrency, distribution and optimization. This division allows (un)pluggability of concerns. Given that each concern is implemented in a separated module, any combination of modules can be plugged or unplugged or switched by another one.

A library[CSM06] of reusable aspects was also developed to implement a collection of common concurrency patterns. Examples of those concurrency patterns are : one-way, futures, barriers, *etc.*.. Higher modularity, reusability, understandability and (un)pluggability are the key benefits claimed. This library is an alternative to the development of concurrent applications without using Java concurrency constructs directly. Adding concurrency as an incremental feature makes the application more modular and easier to debug.

Purushotham V. Bangalore presented a work[Ban07] that uses AOP to parallelize a distributed matrix-matrix multiply algorithm. Unlike the two previous approaches, the language chosen was C++ and AspectC++ the AOP language. Communication and synchronization concerns were cross-cut allowing the algorithm's core to be isolated from the explicit use of the communication library (MPI). The results showed that there was no significant loss of performance from the use of AspectC++ compared to a hand-written solution.

3.2 FOP and Class Refinements

Feature Oriented Programming (FOP) is a programming technique that relies in program synthesis by composition of features. Conceptually, a feature is an increment in program functionality, i.e., a new service. Features can be implemented using several alternatives [Pre97, SB02, DBR04, ALS06, AKL09].

The concept of refinement[4] has a broader scope and can be used not only to refer to the addition of code, *i.e.*, code refinement, but it can also encompass [DBR04] the addition of documentation, *makefiles* or any other entity in a software project. Entities other than refinement of code are out of the cope of this dissertation.

We refer to **Class Refinements** as an incremental change to a base class where new specialized behaviour is introduced without creating a new entity. It can be seen as a function that map classes-to-classes where new behaviour is introduced in a rewriting mechanism. In the next subsections we present and discuss tools that implement class refinements in Object-Oriented environments.

3.2.1 AHEAD

Algebraic Hierarchical Equations for Application Design (AHEAD) [DBR04] is an architectural model that allow refinements of both code and non code artefacts. Refinements are feature additions using a set of equations based in the GenVoca [BO92] design methodology. Programs are represented by constants and refinements are functions applied to those programs. Program 7 represents this strategy.

```
b // Program with feature b

f(b) // addition of feature f to program b
app = f(b) // Program with feature f and b
```

Program 7: GenVoca expressions that can express mathematically composition of features.

Refinements in AHEAD can be feature additions in the form of source code or other representations like makefiles or project documentation (e.g.: XML or HTML data). The feature composition operator is polymorphic in the sense that refinements are implemented depending on the source of the entity, i.e., each refinement type (e.g.: source code) will have an unique implementation. Thus, it is possible to have an application composed by source code, makefiles and documentation and be described in the same GenVoca expression.

AHEAD implements code refinement in the Java language by source code transformation. It extends the Java language to cope with refinements and

[4]Refinements can be used in different applications or contexts (e.g.: Refinement Calculus).

other mechanisms that are out of the scope of this study. Jak [BLS98] is the result of that extension and it is a *superset* of Java. The Jak language introduces some keywords to allow the refinement mechanism. To the comprehension of the refinement mechanism, we discuss only two :

- **refines**. This new keyword is used to identify a new feature or refinement. Program 8 shows an example of a refinement where the *method1(int)* in the base class A is extended;
- **super()**. The super keyword has the same meaning as in Java or C#. It is a mechanism to reuse the code defined in the parent class.

```
refines class A {
  int b;

  public void method1(int a){
    super.method1(b);
    ...
  }
}
```

Program 8: Refinement in AHEAD. Introduction of state (field *b* and extension of method *method1*.

The process to compile an AHEAD program is done in three steps:

1. Merge all refinements being applied to each class into new classes still in the Jak format. This process can be done using one of two available tools : **mixin** or **jampack**;
2. Compile the generated Jak classes into Java classes;
3. Use a Java compiler to generete the corresponding Java bytecode.

This process is presented graphically in figure 3.2.

3.2.2 Classboxes

Classboxes [BDNW05] were introduced to solve problems in Object-Oriented solutions caused by unanticipated changes to the class hierarchy. For instance, duplicated code in suboptimal class hierarchies.

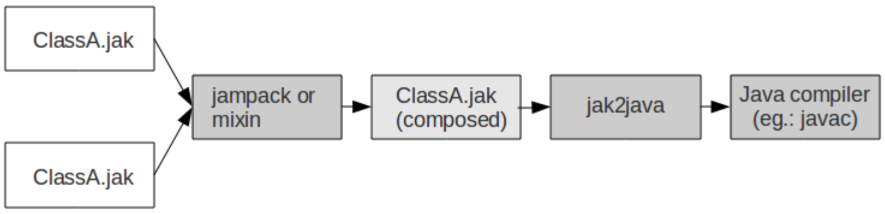


Figure 3.2: AHEAD source code compilation steps.

The solution proposed is the creation of a unit of modularity (classbox) with some special properties. In this section, classboxes will be explained by its implementation, Classbox/J [BDN05] that implements the classbox mechanism for Java :

1. Inside a classbox, classes can be defined, imported and refined. Classes can only be defined in a unique classbox but can be refined and imported by others classboxes. Refinement is the property that allows the addition and modification of classes features;

2. A refinement inside a classbox is only visible to that classbox and others classboxes that directly or indirectly import the refined class. This mechanism is possible because the import clauses are transitive in Classbox/J;

3. Local refinements have precedence over imported refinements.

```
package RefinementExample;
import src.Example; //Class that will be refined

refine Example{
   private String name; //Field Addition
   @Override
   public void example(){ //new code}
}
```

Program 9: Example of refinement in Classbox/J.

Program 9 shows an example of refinement. The field *name* was added and the method *example()* overrides the original behaviour. It is possible, however, to use the behaviour of the method in the base class by using the reserved keyword *original()*.

Applying the refinements directly to the original class having control of the scope of change is a powerful mechanism. It is a mechanism that allows the manipulation of cross-cutting concerns.

Classbox/J is a prototype and thus its implementation shows an increment in the execution cost (22 times slower) [BDN05]. The problem is related to exposing the context of a method call in the Java bytecode. Another implementation [BDW03] showed that the use of classboxes brought little overhead (about 1.2 times slower) when implemented in the Smalltalk virtual machine.

3.2.3 GluonJ

To address the rapid evolution of software and to minimize the efforts in software extension, Shigeru Chuba [NC08] developed a tool/mechanism that extends the Java language allowing a static and dynamic redefinition of an existing class. These extensions are made in a new unit of modularity, the class refinement. A class refinement can be seen as a standard Java subclass where new fields, methods and interfaces can be applied to a parent class. Methods of the parent class can be also extended or overridden using the same semantics as Java inheritance except for class methods (static methods) where, instead of method *hiding*[5], GluonJ implements method overriding. This is a comprehensible decision because GluonJ introduces modifications directly to the base class using Javassist [CN03]. Javassist is a tool that provides high level abstraction to deal with Java *bytecode*. The Java *bytecode* of classes in a base hierarchy are manipulated in load-time by the GluonJ runtime system depending on the definition of the class refinements.

Class refinements are defined using Java annotations, thus, they can be compiled like the base classes with a standard Java compiler (e.g.: *javac*).

Program 10 shows an example of how to create a class refinement and how to override and append a new method in an existing class. Line 1 corresponds to the creation of a *@Glue* class. This class can encompass several class refinements that can be applied to different classes or the same class. In the latter case, the order in which the class refinements are written is the order used by the GluonJ runtime system. Line 2 shows how to define a class refinement. In this example, *Point2* is a class refinement of class *PointImp*, i.e., state and behaviour defined in *Point2* will be introduced by the GluonJ runtime system to the class *PointImp*. Lines 3 to 8 shows the definition of

[5]Method hiding is a mechanism implemented in Java that allows subclasses to have class methods with the same signature as methods defined in the parent class. In these circumstances, *casting* a subclass instance to the type of the parent class makes that invocations to these static methods are made in the parent class while overridden methods are called in the subclass.

a method *movePoint()* that extends the implementation from the *PointImp* class by introducing the *println* instruction before calling the original method. Another refinement is the introduction of a new method called *resetPoint()* (line 9 to 11). To perform calls to new methods introduced by a refinement, it is required that the target object be *casted* to the refinement class type since it is not possible to directly instantiate a class refinement.

```
1  @Glue class GluonJTest{
2   @Refine public static class Point2 extends PointImp {
3     @Override public void movePoint(int x,int y){
4       System.out.println(''Point will move '' + x
5           + '' in xcoordinate'' + y
6           + '' in ycoordinate'');
7       super.movePoint(int x,int y);
8     }
9      public void resetPoint(){
10       xCoordinate = 0; yCoordinate = 0;
11     }
12  }
13 }
```

Program 10: GluonJ example of a static refinement. Demonstration of the override and append mechanisms.

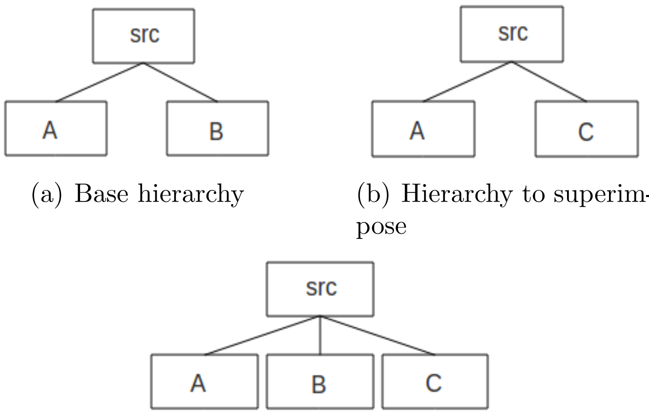
Another mechanism implemented is called dynamic refinement. With dynamic refinements, the behaviour defined in class refinements is only applied to a base class during specific contexts. For instance, as shown in program 11, the method *resetPoint()* is only *available* during the execution of the control flow of method *clean()*, i.e., on calls from the method *Point.clean()*.

```
@Cflow(”void␣Point.clean()”)
@Glue class GluonJTest{
  @Refine public static class Point2 extends PointImp {
    public void resetPoint(){
      xCoordinate = 0; yCoordinate = 0;
    }
  }
}
```

Program 11: GluonJ dynamic refinement. Example of the *cflow* mechanism.

3.2.4 FeatureHouse

FeatureHouse [AKL09] is a general framework and architecture model that relies in the concept of superimposition to compose software artifacts. The superimposition mechanism implemented in FeatureHouse corresponds to the process of decomposing software artifacts into feature trees and software composition is made by *merging* their corresponding substructures. Figure 3.3 illustrates the superimposition mechanism by showing how two source code packages are merged. Figure 3.3(b) is superimposed to the base hierarchy (Figure 3.3(a)). The junction of the two hierarchies is shown in figure 3.3(c).



(a) Base hierarchy (b) Hierarchy to superimpose

(c) Superimposed hierarchy

Figure 3.3: Superimposition example.

However, this example also shows that both initial hierarchies have the same *node*, i.e., the same class (A). The superimposition mechanism deals with this particularity at source code level. This can be seen as a rewriting mechanism where two or more hierarchies can be merged to generate a final hierarchy that encompasses the modifications introduced by each composed hierarchy. Hence, this superimposition mechanism can be used to implemented refinements where each new hierarchy introduces an incremental change. Programs 12 and 13 illustrates superimposition at source code level with a simple example of parallelization. The superimposition of the hierarchies creates a parallel version of the *method2*.

FeatureHouse does not introduce explicitly the notion of base hierarchy because superimposition generically does not distinguish each feature tree. However, sometimes it is necessary to stablish order in the superimposition. The latter example is an example of this case where one of the classes calls the *original* method (Program 13). Original is not a language extension but a mechanism to implement order in superimposition by allowing method extension and refinements.

```
package src;
public class A{
  private void method1(){...}
  public void method2(){
    for(int i = 0; i < N; i++){
      //Computation
    }
  }
}
```

Program 12: Class A - Base.

```
package src;
public class A{
  public void method2(){
    Thread[] list = new Thread[NT];
    for(int i = 0; i < NT; i++){
      list[i] = new Thread(new Runnable(){
        @Override
        public void run(){
          original();
        }
      }).start();
    }
  }
}
```

Program 13: Cass A - New hierarchy to be superimposed.

FeatureHouse is open to the integration of new languages based in their grammar and new software artefacts to implement superimposition operators. This mechanism makes FeatureHouse an interesting implementation to superimposition and more detail can be found at [AKL09].

3.2.5 Comparison

All refinement implementations presented earlier try to introduce modularization improvements over Object-Oriented applications. We present a comparison using properties that we believe are fundamental to achieve an improvement on separation of concerns in Object-Oriented programs. More particularly, properties that can leverage on the construction of parallel programs by incrementally adding parallelization features to a sequential base.

Source code transformation. We believe that a tool that relies in external *scripts* have an intrinsic disadvantage. They do not take advantages of Integrated Development Environment (IDE) that are popular nowadays. For instance, IDEs can check compile-time errors while the programmer is coding. GluonJ is the only tool that complies with the host language (Java) rules (e.g.: inheritance rules or type *casts*). The other tools relies on externals program to generate code to be compiled. This process can be tedious while debugging an application since changes cannot be made in the generated code to avoid different versions of the same code (i.e.: code generated being different of the original code).

Original access : fundamental to build software incrementally. All tools allow access of the original behaviour. AHEAD and GluonJ use *super()* to conform with Java or C# nomenclature and the other use *original*. There is virtually no difference in terms of use.

Unit of modularity : all tools create a new unit of modularity. Thus, changes or increments are well encapsulated. We argue that the nature of FeatureHouse can be a disadvantage regarding a new unit of modularity because to support incremental changes, the programmer has to clone the package hierarchy every time he wants to add a new feature in a cumbersome process.

Composition : extremely important to scale the development of large programs. It is the base of Feature-Oriented Programming because FOP relies in the development of programs by feature composition. Thus, it is specially important in our study because refinements are used to codify parallel concerns separated from the base code. For instance, those concerns can be used to match different target platforms (e.g.: Shared memory or Distributed memory). All tools claim to have been successfully used in large scale projects. AHEAD is probably the most known and it is an interesting example because AHEAD was used to build the tools that implements the AHEAD model [DBR04]. Composition in AHEAD is achieved using *jampack* or *mixin* tools applied to a set of refinements (section 3.2.1). These tools create a single interface or class containing all the features applied(*jampack*) or in the case of *mixin*, an inheritance/refinements hierarchy. *FeatureHouse* decomposes software artifacts into *feature structure trees* and apply a superimposition mechanism to merge their corresponding nodes. The level of granularity can go from packages to classes up to code. In the latter case, *FeatureHouse* implements order in composition by the introduction of a *keyword original().* Composition in GluonJ is achieved by the use of a special *@Glue* class containing only the inclusion of references to previous defined class refinements preceded by the annotation *@Include.* This class is then passed to the GluonJ runtime mechanism.

Performance : critical issue in parallel programming. The separation of concerns overhead has to be minimal. AHEAD and FeatureHouse claim little overhead since the source code transformation tool tries to *inline* method calls in method extensions. The overhead in Classbox/J is problematic [BDN05] to be used in real-life applications. GluonJ's overhead is studied in detail in section 5.5.

Readability : one of the benefits of separation of concerns. This property comes along with the benefit of locality of changes. It is important to distinguish where and how refinements affects the base program. All tools offer good readability but we think that FeatureHouse is worse than the others because there is no special treatment for the software increment with the exception of the use of the *original keyword* in methods bodies. AHEAD and ClassBox/J introduce the *keyword* **refines** to define an increment/evolution while GluonJ uses Java annotations (e.g.: @Refines). Methods can also be annotated with *@Override* and validated by the IDE or compiler.

From the analysis of the tools presented, we opted to use GluonJ to implement class refinements. The decision was made in the ability of GluonJ to implement class refinements by the extension of a popular Object-Oriented language, Java. Moreover, class refinements are implemented like subclasses, thus having valid types that can be used to check compile-time errors. This can be extremely helpful in the development phase because GluonJ can be integrated on an IDE. This is a great advantage in comparison with the other approaches that rely in source-code transformation tools. Another advantage of GluonJ is the little overhead introduced. A detailed study is presented in section 5.5.

Chapter 4

Parallelization with Class Refinements

In this chapter we study the use of class refinements and FOP in the development of parallel programs in Object-Oriented Programming. More particularly, and by the lack of previous studies on separation of concerns in parallel programs using class refinements and FOP, we introduce a Parallelization Model validated by the implementation of several case studies resulting in enhancements in modularization with little or none base code *refactoring.* A Feature Model, typical in FOP, is also introduced to help to build families of parallel programs that share the same base algorithm but differ, for instance, in the target platform.

4.1 Parallelization Model

In order to achieve the best performance of a system, parallel programs need to take full advantage of the underlying execution platform. This implies, in many cases, that the parallelization feature must be coded and tuned according to each target parallel platform (distributed *versus* shared memory). We advocate that parallelism related statements must be coded separately from the application code to better cope with this variability. We also believe that the separation of the parallelization feature from the domain code is the best decision regarding modularity[1] and how a parallel program should be developed.

Our parallelization model addresses incremental development by relying on class refinements. A class refinement is a class extension similar to stan-

[1]The benefits that comes from good modularity decisions are well understood and accepted by the community.

dard subclass mechanisms [Bat06] by adding new fields and methods to a target class as well as extending existing methods (*a la* method overriding in class inheritance). Class refinements differs from standard inheritance in OOP in the sense that a refinement augments the definition of a given class by adding a new feature and keeping the name of this class. This transformations are the core of this work. In our approach, parallelization is the process of mapping application's functionality into a target platform by using class refinements to progressively insert code that copes with more platform-specific details.

Figure 4.1 illustrates a simple example of class refinement. Figure 4.1(a) defines a base class A. Its refinement (Figure 4.1(b)) encapsulates new state and behavior that can be added to A. Figure 4.1(c) is the result of composing this refinement to A. Refinements are transformations. Transformations avoid scalability problems that arise when using traditional inheritance mechanisms of object-oriented (OO) programming in layered designs [SB02]. This topic was presented with an example in chapter 3 and figure 3.1.

```
class A {
    int a;
    void method1(){...};
}
```

(a) Base class

```
class Aref refines A {
    int b;
    void method2(){...};
}
```

(b) Refinement

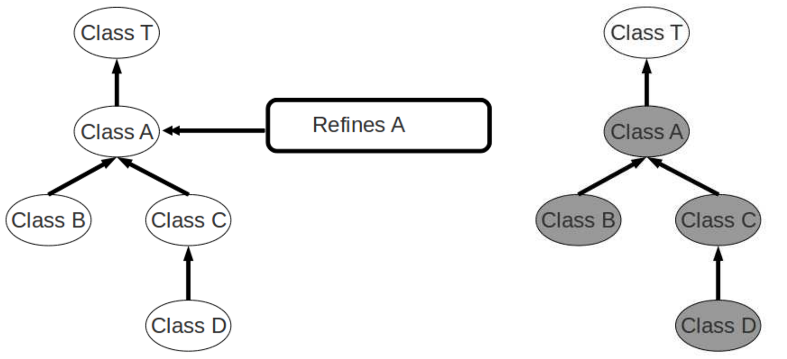
```
class A {
    int a;
    Int b;
    void method1(){...};
    void method2(){...};
}
```

(c) Final Class

Figure 4.1: Class Refinement example.

4.1.1 Compatibility with Inheritance

Software enhancement/evolution (e.g.: parallelization) and dealing with unanticipated changes are two important factors in software engineering. Extending a class by means of traditional Object-Oriented inheritance implies alterations in the class hierarchy to take advantage of the new behavior defined in the subclass. For instance, the subclass has to be instantiated, changing the original class hierarchy. This problem is noticeable in parallel programs where parallelization statements may be inserted anywhere in the class hierarchy. Using class refinements, methods extensions are defined in a new unit of modularity and propagated in the class hierarchy by applying arbitrary refinements at any point of the class hierarchy. Method extension is the ability



(a) Original Class Hierarchy (b) Class affected by refinement

Figure 4.2: Inheritance compatibility.

to redefine a method defined higher in the class hierarchy but using the *super()* keyword in the method body. This is fundamental to achieve modular parallel programs since these redefinitions correspond to entry-points where parallelization can be injected. Making a comparison/connection to AOP and, in particular, AspectJ, method extension corresponds to the use of **before** and **after** *advices*. In Figure 4.2(a), it is shown a class hierarchy and a refinement being applied to the A class. The classes affected by the changes (shaded) are illustrated in Figure 4.2(b). As an example, method execution in instances of D keeps the semantics of method-lookup of traditional Object-Oriented languages. The difference happens when refinements are applied to classes. In that case, method-lookup will first execute the method extensions defined in the refinements. Two or more refinements can be applied to the same class or to different classes of a class hierarchy. Composition of refinements is extremely important because it allows to develop more complex abstraction by the composition of simple ones. Moreover, these abstractions can be used to provide a base to future refinements.

Keeping intact the class hierarchy and applying refinements as transformations is important to avoid code scattering. Otherwise, there will be multiple class hierarchies to cope with multiple parallelization models and their platform mappings.

4.1.2 Parallelization Layers/Features

Incremental development is the ability to build programs by successively adding more functionalities/features. In our parallelization model, features are not restricted to be a single class refinement but they are formed by a set of class refinements as well as regular classes. As an abstraction, features can be seen as layers where a new abstraction or implementation is introduced. Layers provide an *atomic* view of a feature in the sense that members of a layer cannot exist by themselves.

Figure 4.3 illustrates this layer mechanism. Feature/Layer 1 is an example of what can be a parallelization in shared memory (SM). It consists of a refinement of class 3 and an introduction of a new class R1. Feature/Layer 2 is an example of a parallelization in distributed memory by refining class 2 and 3. In this example, there is no dependency between Layer 1 and Layer 2 but that possibility is valid. In such case, Layer 2, could, for instance, refine class R1.

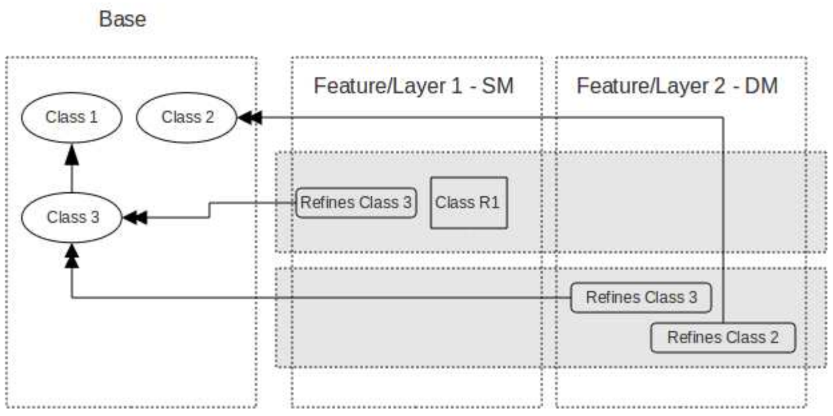
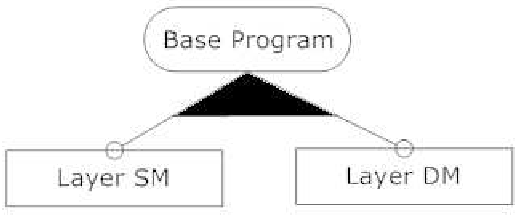


Figure 4.3: Example of a Class Refinement Model.

4.1.3 Composition

Since our model supports multiple parallelization layers, it incorporates an explicit composition step. This step is fundamental to specify which layers must be applied in which order to the base program. Designing a program is composing these layers. This step is particularly important in parallel programs since different parallelization features can be combined to take the best advantages of each environment. For instance, parallel algorithms are called hybrid when there is more than one paradigm being applied. This hybrid solution can be obtained by the composition of both the shared and distributed memory parallelization layers.

In order to specify valid combination of features, we borrowed the notion of feature model [CE98, Bat05] that is used to describe restrictions in feature compositions. With a feature model, programmers define the valid



(a) Feature Diagram

```
Program : [Layer SM]  [Layer DM] Base
```

(b) Feature Grammar

Figure 4.4: Definition of valid combination of features.

combinations of layers that can be applied to a base program. There are different ways of expressing these combinations and for the remainder of this document, we use feature diagrams and grammars [Bat05]. Figure 4.4 shows two possible representations of a valid composition of layers presented in figure 4.3. The feature diagram is a graphical representation of the feature model (Figure 4.4(a)). A grammar, shown in figure 4.4(b) is another representation.

4.2 Case Studies

Several case studies from different domains were implemented to validate our parallelization model. We did not focus only on academic problems but we tackled the parallelization of a medium-sized framework that has been developed for the past few years in University of Minho and has been used in many fields but with particular emphasis in *Bioinformatics*. Overall the framework has currently 125 classes and a total of 5600 lines of Java code. These numbers exclude code for the case studies and libraries that the framework depends on. The other case studies belong to the Java Grande Forum (JGF) [SBO01] benchmark suite.

In the next subsections we detail and highlight the implementation of these case studies.

4.2.1 JECoLi

The Java Evolutionary Computation Library (JECoLi) is a framework created to cope with recent advances in the Genetic and Evolutionary Computation (GEC) domain. These domains are being used to solve complex optimization problems in a wide range of scientific and technological fields. Optimization problems are the computation of an optimal solution using numerical methods given a set of constraints.

JECoLi is designed as a framework to provide *out of the box* meta-heuristic algorithms like Evolutionary Algorithms (EAs), Genetic Programming (GP) or Differential Evaluation (DE). Evolutionary Algorithms are inspired in biological evolution in the sense that it defines abstractions like population and individuals of populations that interact with each other. That interaction creates new individuals with characteristics *inherited* from their parents (e.g.: reproduction). Figure 4.5 represents the typical main steps of an Evolutionary Algorithm. The first step is the creation of a new population or solution set[2]. The next step is to evaluate each individual of a population by calculating a fitness value. A fitness value represents the *quality* of each individual. A termination criteria is a boolean expression that can be based, for instance, in the number of iterations or the quality of each solution. Selection is a step where the best individuals are chosen (based in the fitness value) to participate on the *reproduction*. Reproduction is represented by the operators step since it consists in the application of genetic operators, i.e., mutation or recombination to produce a new population.

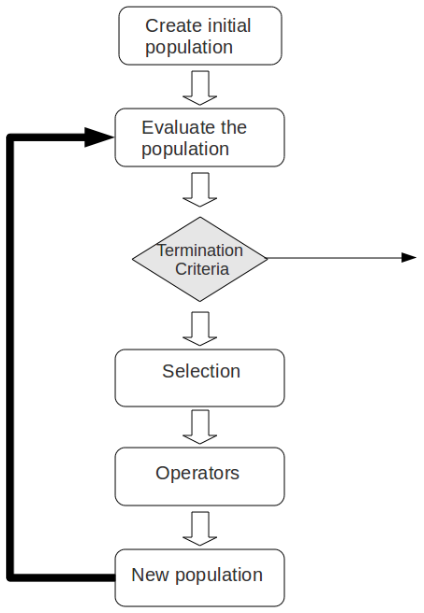


Figure 4.5: Main steps of an Evolutionary Algorithm.

Generically, JECoLi users are interested in the computation of optimization problems. Using an analogy to biology evolution, this can be seen as

[2]Population or solution set are terms that can be used interchangeably. The same happens to individual or solution.

the creation of the strongest population by selecting the best individuals in each generation to be used in the reproduction process to form stronger next generations.

Figure 4.6 illustrates a simplified class diagram. We omit the representation of Interfaces for simplification purposes but each of the classes represented implements an Interface with the corresponding *service* it provides. The *AbstractAlgorithm* represents the *skeleton* of an optimization method. It must be extended by each optimization method implementation to provide its specific behaviour. Each algorithm encompasses an *AlgorithmState* that collects information about the execution of an optimization method like current solution and previous calculated solutions for statistical purposes.

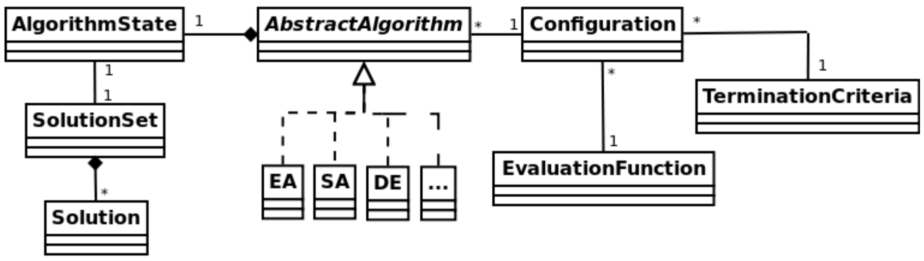


Figure 4.6: JECoLi's simplified class diagram.

Figure 4.7 shows how to instantiate the framework to use an Evolutionary Algorithm[3]. First, users must instantiate a *Configuration* that has all necessary information to the optimization algorithm :

- **Evaluation Function** : used to compute the fitness of a solution, i.e., value to optimize given an objective;
- **Termination Criteria** : a criteria to *stop* the algorithm. For instance, number of iterations;
- **Selection Operator** : operator to select how individuals evolve over iterations.

The created configuration is passed to an instance of an *AbstractAlgorithm*. In this example we show the use of an *EvolutionaryAlgorithm*. The method *run()* is responsible to *prepare* the simulation and calling the method

[3]Note : In this example, we use the *Evolutionary Algorithm* but the framework has implemented others (e.g.: Simulated Annealing or Differential Evaluation).

iteration(). This method is constituted by a loop that iterates until the termination criteria is met calling the method *evaluate(SolutionSet)* at each iteration to evaluate the population.

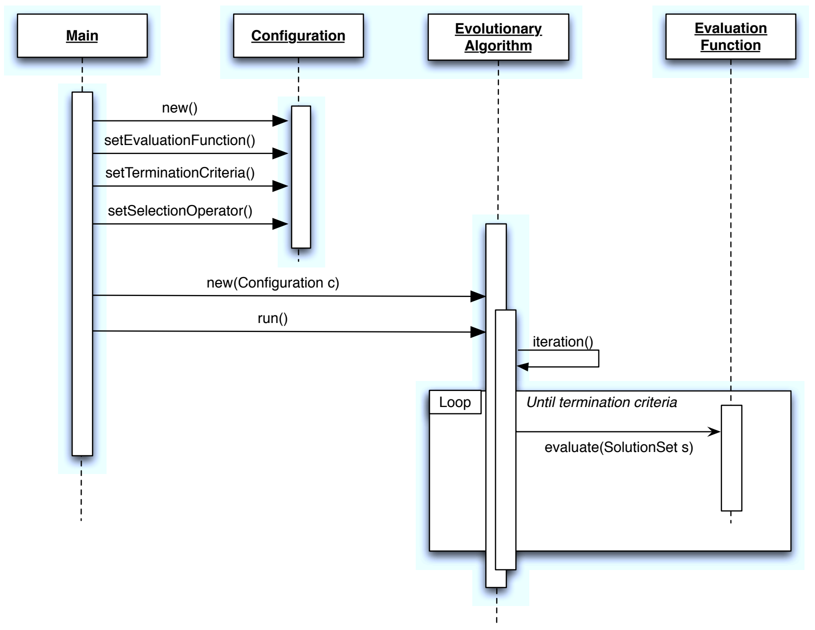


Figure 4.7: Typical workflow to use the JECoLi framework.

Developing a parallel version of the framework requires rewriting classes **AbstractAlgorithm** and **EvaluationFunction**. Using traditional parallel programming techniques, direct modifications to the base framework would introduce tangling, since parallelism related code would be mixed with the code of the base framework. Further, introducing code into the **AbstractAlgorithm** class may enforce a particular parallelization model for all optimization methods as they usually extend this class. This could limit algorithm's scalability since it could force a non optimal parallelization for a given algorithm. Additionally, derived classes could accidentally override the parallelization code (or could even completely override the default behavior).

A more fundamental problem is the long term maintenance of the framework. Ideally, parallelism-related code should be localized in specific modules and not scattered across multiple classes, tangled with basic framework code. Moreover, each optimization method may support a subset of available parallelization models/target platforms, so these constraints should be made explicit. This is particularly important for long term evolution, as new parallelization models, target platforms and optimization methods may be supported.

JECoLi - Parallelization

There are two well known techniques (that can also be combined to achieve better performance) to execute evolutionary algorithms in parallel. In this dissertation we present two :

1. **Parallel Evaluation** : the evaluation process is done in parallel. Solutions are computed in parallel in each iteration. This parallelization does not scale for distributed memory systems due to the excessive communication overhead to send and receive solution sets;

2. **Island Model** : creation of a new abstraction where the solution set is divided into multiple sets (islands) that evolve in a loosely coupled way. Island models usually require periodic migration of solutions among islands.

JECoLi's parallelization is challenging because there are properties that must be addressed carefully :

Evolution : The semantics of the framework should not change. Parallelization should not compromise the introduction of new algorithms as well as the maintenance and evolution of the existing ones;

Adaptability : Parallelization should be adaptable to match different target platforms. There is no one-parallelization-fits-all as it depends on the target platform (e.g. cluster vs. multi-core);

Performance : The execution time must decrease in comparison with the sequential algorithm and it must generate solutions as good as the sequential implementation. Parallelization should perform better than a sequential algorithm; otherwise the sequential implementation should be used instead;

Solutions : This is particularly important for island models as they might produce a different solution (usually better as they avoid local minima in the search process).

Figure 4.8 shows a layered view of the parallelization features. At the bottom of the diagram, we represent the JECoLi framework (base framework with no parallelization). Features are applied on top of the base framework or on top of other features.

An Island Model and Parallel Evaluation are the first features that can be applied to the base framework. These are two different but complementary functionalities that can be used to enhance the framework. The former

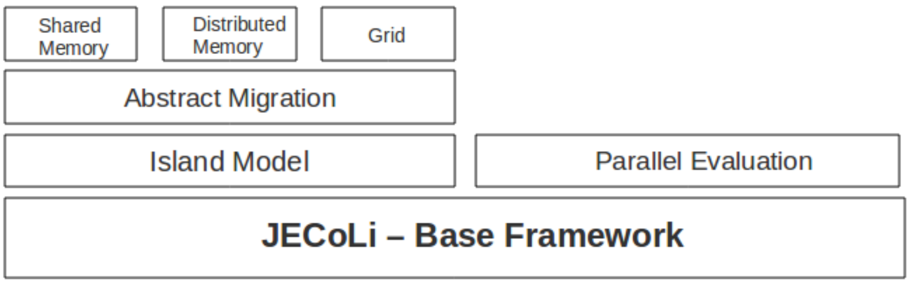


Figure 4.8: Layer architecture

introduces the island model abstraction where solutions are split by several islands. The latter introduces the parallel behaviour for the solution's set parallel evaluation. The order in which these features are applied to the base is unimportant as they are mutually exclusive, i.e., they affect different modules/sections of the base code. They can be both applied to the base to attain an hybrid parallelization model, i.e., each island evaluates the solution set in parallel.

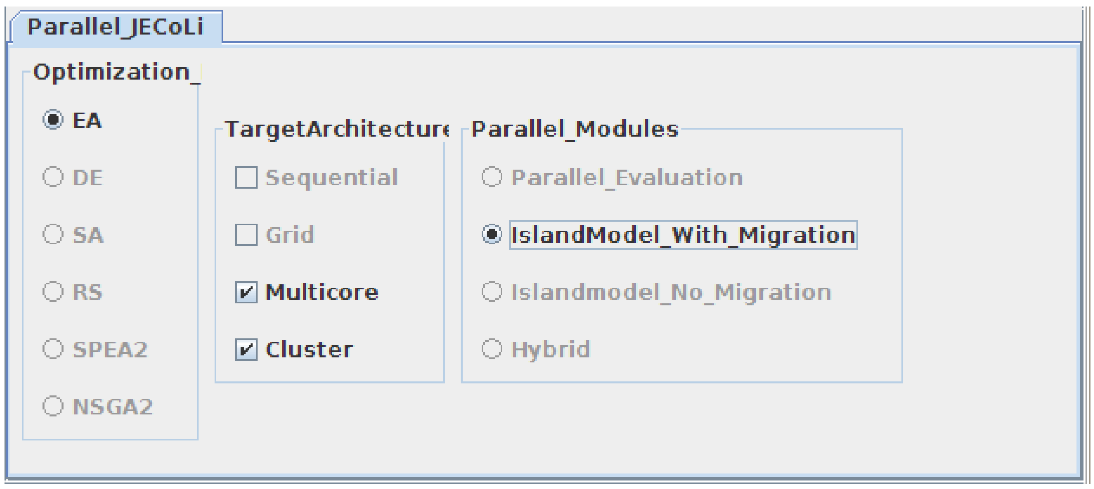
The Abstract Migration feature corresponds to the introduction of common functionalities about migration of solutions among islands that are need for the Island parallelization model. This represents a dependence since it requires the Island Model Feature. Alone, this layer does not affect the frameworks instance execution or behavior. This layer is responsible to provide rules (an interface) and basic functionalities so that other layers that can be applied on top of that.

The Island Model and Abstract Migration are parallelization features that implement the supported parallelization models. They encapsulate common platform independent behavior. The last (upper) layer encapsulates the mapping of these parallelization layers into specific target platforms (multi-core, cluster and grid). Currently, the Parallel Evaluation is only supported in multi-core systems, as our case studies do not justify the support for other target platforms but the flexibility of this model allows us to implement more platform mappings in the future.

During the parallelization of the JECoLi framework we incrementally developed the above mentioned features. As the number of features grew, as well as the identification of constraints among them, it became evident the need for a tool to explicitly manage composition issues. For this purpose we use GUIDSL[4] to create a graphic interface based on a grammar that specifies valid combination of features.

[4]http://userweb.cs.utexas.edu/users/schwartz/ATS.html

Figure 4.9 shows a simple screen of GUIDSL to configure all possible parallel versions. The leftmost rectangle corresponds to the optimization method used in the framework instance. In the middle rectangle, the target architecture is selected. For instance, selecting multi-core and cluster means an hybrid approach. In the last rectangle (rightmost), the parallelization model is selected. Note that not all parallelization models can be selected for every combination of optimization methods and target architectures. For instance, parallel evaluation is not possible in a sequential architecture nor it can be used with Simulated Annealing optimization method.


Parallel_JECoLi

Optimization_
[✓ EA]
DE
SA
RS
SPEA2
NSGA2

TargetArchitecture
Sequential
Grid
[✓ Multicore]
[✓ Cluster]

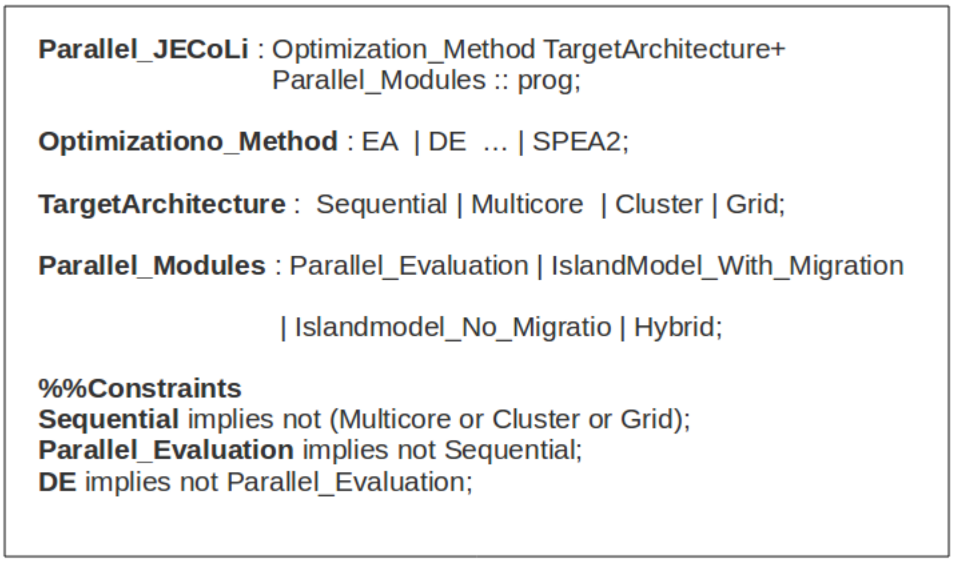
Parallel_Modules
Parallel_Evaluation
[✓ IslandModel_With_Migration]
Islandmodel_No_Migration
Hybrid

Figure 4.9: GUIDSL feature composition.

GUIDSL uses a grammar to define valid combination of features. Figure 4.10 shows a simplification of that grammar.

Implementation overview

Parallel evaluation is a technique that performs the evaluation of the set of individuals or solutions in parallel. Each optimization algorithm in a framework instance has to provide a method to evaluate the solution at each iteration step. Program 14 shows the abstract class that belongs to the framework with the evaluation method. The concrete method *evaluate* will iterate over the solution set and call, for each solution, the evaluate method provided by framework instances. The purpose of this separation is to introduce flexibility while dealing with different solution representations. In other words, framework instances have to extend this class and provide a valid implementation to the abstract method listed in program 14.



```
Parallel_JECoLi : Optimization_Method TargetArchitecture+
                  Parallel_Modules :: prog;

Optimizationo_Method : EA  | DE  … | SPEA2;

TargetArchitecture :  Sequential | Multicore  | Cluster | Grid;

Parallel_Modules : Parallel_Evaluation | IslandModel_With_Migration

                   | Islandmodel_No_Migratio | Hybrid;

%%Constraints
Sequential implies not (Multicore or Cluster or Grid);
Parallel_Evaluation implies not Sequential;
DE implies not Parallel_Evaluation;
```

Figure 4.10: GUIDSL grammar specification.

Shared memory parallelization of this algorithm can be decomposed in the following steps :

1. Extend the method that performs the evaluation in the abstract class to :
   (a) Divide the domain of the problem (solution set). Method *divideSet(ISolutionSet)*;
   (b) Spawn threads to compute on solutions' subsets in parallel;
   (c) Wait for threads to resume execution with updated values.

To illustration and comparison purposes, we describe the implementation details of the **Parallel Evaluation**. It is important to remember that JECoLi is a framework and the *ParallelEvaluation* is an abstract class of the framework's core. Thus, several optimization algorithms use the functionalities of this class. Given this, the parallelization feature must be inserted in a way that does not change the semantics of the framework, i.e., it does not enforce changes in the optimization algorithms. We present the Parallel Evaluation in four different implementations :

The traditional (tangled) approach in program 15. The class *EvaluationFunction* is populated with variables to enable the introduction of the parallelization feature (array to reference threads and data structure to support the division of the solution set). The body of the method *evaluate(ISolutionSet)* is copied into a new method to support calls from a Thread object keeping the semantics of the original method, i.e., receiving a *ISolutionSet* and performing the evaluation presented in program 14. The method *evaluate(ISolution)* is re-written to support the division of the solution set and threads' creation.

Object-Oriented Inheritance in program 16. Using Java inheritance, the parallel feature is coded in a subclass and each thread calls the original method evaluate of the base class to keep the semantics of the application. This mechanism was implemented with an auxiliary class presented in program 17 because it is not possible to use *super()* inside a Runnable object or a new Thread. This solution is the only one that we present that forces framework's instances to be changed in order to take advantages of the parallel feature. They must extend the *ParallelEval* instead of the *EvaluationFunction* class. Note that the abstract method *evaluate* has to remain abstract because this class does not represent a framework instance.

AOP (AspectJ) (Program 18). The aspect *ParallelEvalAspect* introduces the parallel feature by intercepting calls to the method *evaluate* in the class *EvaluationFunction*. The *advice* introduces the parallel behavior where is created new instances of the *ThreadEval* class that are responsible to call the original method evaluate in a new thread.

Class Refinement. Program 19 shows the implementation using Class Refinements (GluonJ's notation). The class *EvaluationFunction* is refined to introduce the new parallel behavior by extending the method *evaluate*. The refined method introduces the parallel behavior where each thread calls the original method *evaluate*. This mechanism was implemented exactly like the Java inheritance and the AspectJ examples[5]. The great advantage of this approach, unlike the Java inheritance example, is that framework instances do not need to change their class hierarchy to use the parallel evaluation feature.

[5]We did not used *super()* for the same reason as Java inheritance, i.e., it is not possible to call *super()* inside a Runnable object

```
public abstract class<T extends IRepresentation>
   EvaluationFunction implements IEvaluationFunction<T>
{
  public void evaluate(ISolutionSet set){
    for(i = 0; i < set.getNrSolutions(); i++){
      ISolution sol = set.getSolution(i);
      List<Double> fitness = evaluate((T) solution.
         getRepresentation());
      sol.setFitnessValue(fitness);
    }
  }

  public abstract List<Double> evaluate(T
     solutionRepresentation);

}
```

Program 14: Base code.

Benchmarks

In this section we measure the performance of the parallelization of two case studies that belong to the biological computational field. The former represents an optimization task in fed-batch fermentations [MRFR06] and the latter the knock-out optimization [RMM+08] problem. The understanding of the nature of these problems is not fundamental for the remainder of this section.

For each case study we compare two implementations of the same parallelization feature. One based in AspectJ [PRS10] and the other using an FOP approach using class refinements implemented with GluonJ.

The instrumentation code used to measure the execution time of each implementation is based on the system call *System.currentTimeMillis()* that returns the difference, measured in milliseconds, between the current time and January 1, 1970. The speed-up is calculated based on the median of execution time of each parallelization implementation divided by the same execution time, i.e., the median of the execution time of the base algorithm with no parallelization. Using this approach, we guarantee the same speed-up calculation for all implementations. The execution time was calculated by the median of 5 executions.

```
public abstract class<T extends IRepresentation>
   EvaluationFunction implements IEvaluationFunction<T>
{
  ThreadEvalAux[] workers = new ThreadEvalAux[NRthreads];
  ISolutionSet[] subList;

  public void evaluate(ISolutionSet set){
    subList = divideSet(set);
    for(int i=0; i < NRthreads; i++){
      workers[i] = new ThreadEvalAux(subList[i]);
      workers[i].start();
    }

    for(int i=0; i < NRthreads; i++){
      try{
        workers[i].join();
      }catch(Exception e){...}
    }
  }

  public void evaluateCore(ISolutionSet set){
    for(int i=0; i < set.getNrSolutions(); i++){
      ISolution sol = set.getSolution(i);
      List<Double> fitness = evaluate((T) solution.
          getRepresentation());
      sol.setFitnessValue(fitness);
    }
  }

  public abstract List<Double> evaluate(T
     solutionRepresentation);

  class ThreadEvalAux{
    ISolutionSet set;
    public ThreadEvalAux(ISolutionSet set){this.set = set;}
    public void run(){ evaluateCore(set); }
  }

}
```

Program 15: Traditional (tangled) approach. The class *ThreadEvalAux* is an inner class used to create a thread object to compute on a subset of the initial solution set. The method *run()* only calls the original *evaluate(ISolution)* method that is now called *evaluateCore(ISolution)*.

```
public abstract class<T extends IRepresentation>
    EvaluationFunctionParEval extends
    EvaluationFunction<T extends IRepresentation>
{

  ThreadEval[] workers = new ThreadEval[NRthreads];
  ISolutionSet[] subList;

  @Override
  public void evaluate(ISolutionSet set){
    subList = divideSet(set);
    for(int i=0; i < NRthreads; i++){
      workers[i] = new ThreadEval(subList[i],this);
      workers[i].start();
    }

    for(int i=0; i < NRthreads; i++){
      try{
        workers[i].join();
      }catch(Exception e){...}
    }

  }

  public abstract List<Double> evaluate(T
     solutionRepresentation);

}
```

Program 16: Object-Oriented Inheritance approach.

```
public class ThreadEval extends Thread{

  private ParallelEvaluation pe;

  public ThreadEval(ParallelEvaluation pe){
    this.pe = pe;
  }

  @Override
  public void run(){
    pe.evaluate();
  }
}
```

Program 17: *ThreadEval* auxiliary class.

```
aspect ParallelEvalAspect{
  ThreadEval[] workers = new ThreadEval[NRthreads];
  ISolutionSet[] subList;

  void around(ParallelEval eval, ISolutionSet set) :
    call(void EvaluationFunction.evaluate(ISolutionSet)) &&
    target(eval) && args(set) &&
    !within(ParallelEvalAspect){
      subsets = divideSet(set);
      for(int i=0; i < NRthreads; i++){
        workers[i] = new ThreadEval(subList[i],eval);
        workers[i].start();
      }

      for(int i=0; i < NRthreads; i++){
        try{
          workers[i].join();
        }catch(Exception e){...}
      }
    }
}
```

Program 18: AOP (AspectJ) approach.

```
@Refine public abstract class EvaluationFunctionParEval
    extends EvaluationFunction
{

  ThreadEval[] workers = new ThreadEval[NRthreads];
  ISolutionSet[] subList;

  public void evaluate(ISolutionSet set){
    subList = divideSet(set);
    for(int i=0; i < NRthreads; i++){
      workers[i] = new ThreadEval(subList[i],this);
      workers[i].start();
    }

    for(int i=0; i < NRthreads; i++){
      try{
        workers[i].join();
      }catch(Exception e){...}
    }

  }

}
```

Program 19: Class Refinement approach.

The specification of the environment (Cluster) is as follows:

- 4 nodes interconnect by *myrinet* 10Gb;
- Each node is composed by 2 Intel Xeon Processor E5420 (total of 8 cores);
- Java(TM) SE Runtime Environment (build 1.6.0_13-b03).
  Java HotSpot(TM) 64-Bit Server VM (build 11.3-b02, mixed mode);
- AspectJ runtime 1.6.9;
- GluonJ 1.5.

The first case study was implemented using the island model parallelization. Figures 4.11(a) and 4.11(b) show the gains compared with the theoretical maximum. We consider the theoretical maximum the number of processing units (threads or MPI processes). For instance, using 4 MPI processes, we expect a gain of 4x or, in other words, the execution time to decrease in a fraction of 4. The same holds for the parallelization in shared memory.

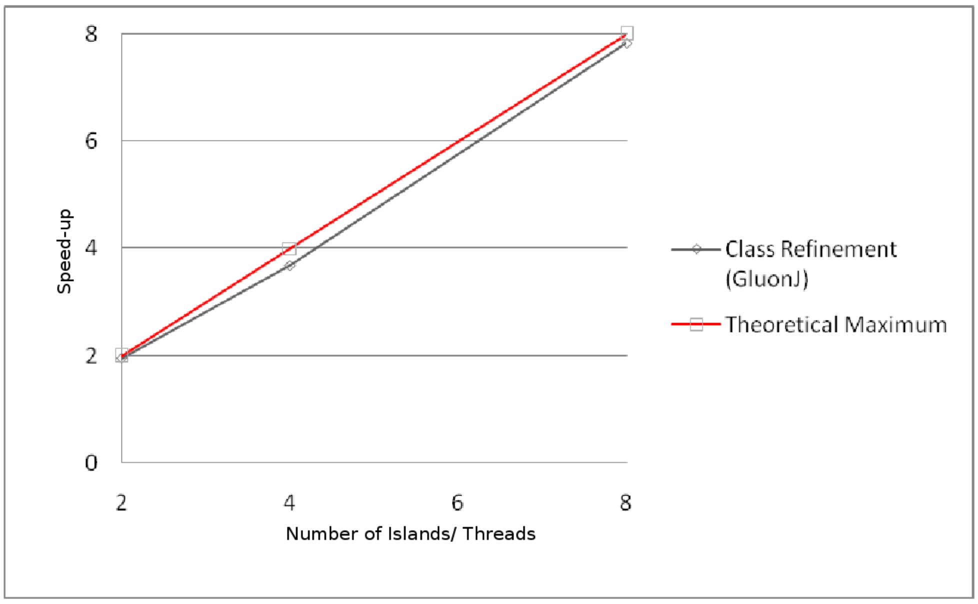
The parallelization gain is almost linear with the number of processing units up to 8 cores and 8 MPI processes. There is a little decrease in larger number of islands because of properties of the parallelization model. With the increase of the number of islands and maintaining the problem's size, fewer individuals populates each island. Thus, the overhead of the communication, i.e., migration of individuals increases.

In a look at the performance of AspectJ and GluonJ implementations, we measure the percentage of the difference in the execution time. The difference was calculated for the GluonJ implementation. Positive differences mean worst execution time. Respectively, negative differences mean better execution time.

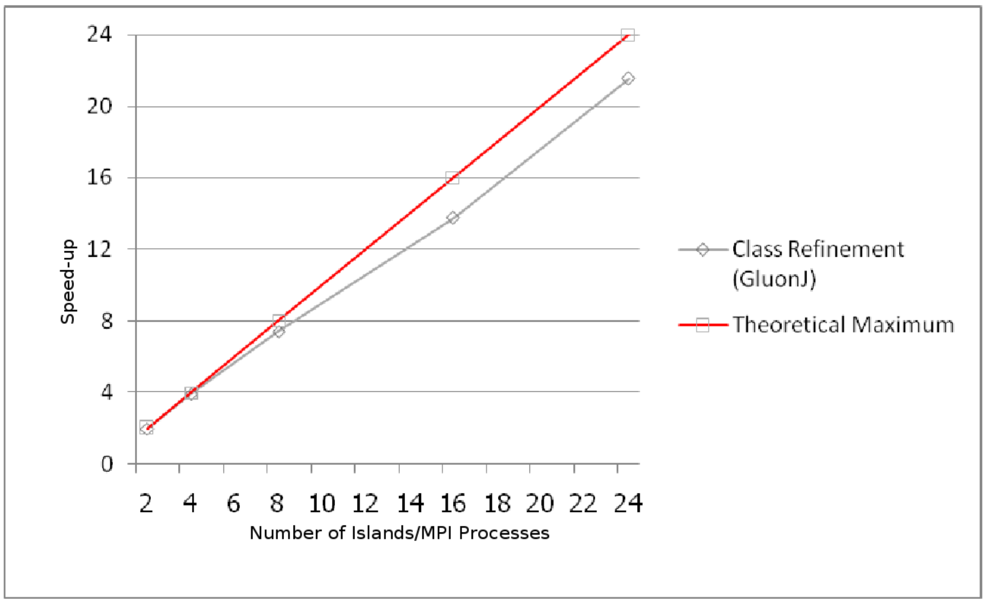
Table 4.1 shows the difference (in percentage) of the execution time for the first case study using the island model mapped to a multi-core (Shared memory) and Cluster (Distributed memory). The results for shared memory migration are not shown for 16 and 24 islands because of the limitation of keeping the ratio of 1 thread per processor core.

The second case study was implemented using the parallel evaluation feature. Table 4.2 shows the difference (in percentage) of the execution time.

The two case studies presented showed similar performance between the two implementations. Given the nature of parallel programs, for instance, creation and destruction of threads, communication between processes and



(a) Shared memory speed-up



(b) Distributed memory speed-up

Figure 4.11: Parallelization speed-up.

Islands	*SM Migration*	*DM Migration*
2	-1.68%	0.68%
4	0.47%	1.24%
8	-0.66%	1.65%
16	*N/A*	-1.95%
24	*N/A*	-2.68%

Table 4.1: Execution time difference in the Island Model.

Threads	*SM*
2	-2.28%
4	-1.86%
8	-0.27%

Table 4.2: Execution time difference in the Parallel Evaluation.

parallel I/O file operations, we advocate that the differences between the two versions are negligible. Both implementations share similar performance and gains.

4.2.2 Java Grande Forum (JGF)

Java Grande Forum (JGF) is a benchmark suite [SBO01] with several algorithms from different domain areas. These algorithms encompass parallelization features in both multi-core processors and MPI (using MPI-Java [BCF+99]). We tested our parallelization model by implementing some case studies and comparing them to an AspectJ approach and the JGF tangled version. A brief description is given for each case study :

Successive Over-Relaxation (SOR) : Variant of the Gauss-Seidel model for solving a linear systems of equations. Sequential version with 2 classes and approximately 170 lines of code (LOC).

Moldyn : Molecular Dynamic (MD) algorithm that simulates the interaction of microscopical particles (e.g.: atoms). Sequential version with 2 classes and approximately 500 LOC;

Ray Tracer : rendering of a scene with 64 spheres in a 3D space. Sequential version with 12 classes and approximately 1000 LOC;

Since our model relies on extending the base code to introduce parallelization features incrementally, the JGF benchmark suite was a good exercise to our parallelization model for its diversity in terms of problems and algorithms' codification. Unlike JECoLi, some algorithms in the JGF are based in older implementations from scientific languages that are not Object-Oriented based (e.g.: Fortran). Since programming on an Object-Oriented language (e.g.: Java) does not mean programming with Object-Orientation, the lack of modularity was sometimes evident. For instance, the SOR algorithm was totally implemented in one method. Without software refactoring, the introduction of parallelization features by incrementally evolving the base code is impossible without using a finer grain decomposition, i.e., refinements on instruction-level. We believe that this is not the solution to this kind of problem and we opted to refactor the base code to enhance modularization. The three kinds of code refactoring used are explained next :

Code movement : The most used refactor was to move a block of code into a named block, i.e., code movement into new methods. This solution was needed to create *plug-points* to allow the introduction of the parallelization features. This situation happens because methods are the smallest named blocks of code in Java that can be *intercepted* by AspectJ and extended in OO inheritance (GluonJ).

Access modifiers : Java default access modifier for methods and variables is *protected.* This access modifier disallows class access or extension outside the package it is defined. Since the base of GluonJ is OO inheritance, we opted to add other permission (i.e. public) to methods and variables to modularize refinements in a different package. This situation could be resolved using two other alternatives :

- The most simple solution was to use the class refinements in the same package as the base code;
- GluonJ offers a mechanism that bypasses access modifiers by using the *@privileged* annotation. This solution is possible because GluonJ works by manipulating the Java *bytecode.*

Data types : The implementation of the *RayTracer* case study forced us to change the data type of a variable in the base code. This situation is due to the current implementation of the *mpiJava* library that does not support the *long* data type. We changed the data type to *int* without loss of precision in the case study.

Benchmarks

We measured the speed-up of the 3 versions, i.e., JGF hand-written (HW), class refinements with GluonJ (CR) and AspectJ (AJ) both with shared and distributed memory parallelization. We used the median of the execution time of 10 executions in the same environment as the one presented for the JECoLi case study. Each case study was executed using the largest input data to minimize the overhead of the instrumentation code to measure the execution time. Shared memory results were executed only in one node.

Table 4.3 shows the results for the shared memory parallelization for each case study. Analysing these results, it is easy to see that the *Sor* and the *RayTracer* case studies show no evidence of overhead introduced by GluonJ and AspectJ. The calculated speed-ups show marginal differences. The *Moldyn* case study shows little differences between all versions until 8 cores. That difference can be explained by the use of a *Thread Pool* in the GluonJ and AspectJ approaches.

In the distributed memory benchmarks, we expect slightly larger differences compared with the shared memory parallelization because of the underlying distributed platform, i.e., we rely in the MPI daemon to start the same program in different nodes of the cluster. Also, messages circulating in the network introduce larger variation in the execution time compared with manipulation of shared data in the same processor. For instance, the load of the switch can introduce variations on the execution time. Adding the fact that even using the larger input data for the JGF benchmarks, the parallelization can diminishes the execution time to less than one second, making the task of analysing the speed-up for 16 and 32 processes harder. Hence, in that order of magnitude, *just-in-time compilation* (JIT)[6] and the state of the network can have a great effect in the benchmarks' results. Given this, we analyse with a greater detail GluonJ and AspectJ with respect to performance in section 5.5.

The results for the distributed memory parallelization are shown in table 4.4. The analysis of these results confirm the larger variation among all the approaches compared with the shared memory parallelization. However, the results are very close among all approaches. In the *RayTracer* case study, it can be noted that the AspectJ has a decrease in performance compared with the other approaches.

[6]Just-in-time compilation is a technique used to improve the performance of interpreted languages like Java. Basically, this technique allows code to be compiled in runtime to reduce the execution time in interpreting a set of instruction. This technique can introduce a great performance boost in some programs but its effect in execution time is noted in smaller applications or applications with little execution time.

	2 cores			4 cores			8 cores		
Application	HW	CR	AJ	HW	CR	AJ	HW	CR	AJ
Sor	1.84	1.87	1.87	3.36	3.37	3.37	3.82	3.83	3.83
Moldyn	1.90	1.87	1.92	3.27	3.20	3.21	5.21	4.1	3.97
RayTracer	1.95	1.96	1.94	3.57	3.57	3.56	6.52	6.52	6.50

Table 4.3: JGF Benchmarks results. Speed-up of the shared memory parallel version.

	P4			P16			P32		
Application	HW	CR	AJ	HW	CR	AJ	HW	CR	AJ
Sor	3.36	3.32	3.24	7.01	6.92	6.97	6.03	6.00	5.83
Moldyn	3.86	3.64	3.67	12.89	12.24	12.26	16.38	15.62	15.37
RayTracer	3.95	3.91	3.94	10.74	11.40	9.35	15.33	16.66	12.01

Table 4.4: JGF Benchmarks results. Speed-up of the distributed memory parallel version.

Chapter 5

Comparison of Approaches

We make a comparison with AspectJ, i.e., the only other approach published in the literature that enables separation on concerns in parallel applications. The main purpose of this comparison is to show that, by the experience gathered in implementing several case studies, the most used AOP mechanisms may be replaced by an approach that is based on Object-Oriented Programming and shares almost the same concepts of OOP inheritance. We also show that FOP with Class Refinements can have solid advantages compared with AspectJ besides its simpler model. Thus, we believe that the FOP approach can leverage the migration of scientist and programmers to a model that introduces little concepts compared to the AOP.

5.1 Static *vs* Dynamic *Cross-cutting*

Aspect Oriented Programming is generally introduced as a programming model that deals with *cross-cutting* concerns that can be either static or dynamic. Static cross-cutting refers to the static structure of a program, i.e., AOP allows the redefinition of the class hierarchy as well as a finer grain redefinition at class level where new sate can be added to classes (e.g.: class' variables). Dynamic cross-cutting is the ability to modify the behaviour of methods following the join-point model explained in section 3.1.

Table 5.1 shows the use of both static and dynamic cross-cutting in all of the case studies implemented in this study. Analysing the results, there is only one case study that takes advantage of static cross-cutting. The JECoLi parallelization takes advantage of this characteristic of the AOP model to introduce state (variables) to another class. Access to these introductions are completely handled by another aspect. The other use was regarding the distributed memory parallelization where a class of the framework was changed to implement the *serializable* Java interface in order to be used as an Object capable of circulating in the network between program's instances. All case studies used dynamic cross-cutting as the main technique to introduce parallelization features.

	Static Cross − cutting	*Dynamic Cross − cutting*
Sor	*No*	*Yes*
Moldyn	*No*	*Yes*
Ray Tracer	*No*	*Yes*
SparseMMM	*No*	*Yes*
JECoLi	*Yes*	*Yes*

Table 5.1: Static vs dynamic cross-cutting. Comparison of Approaches.

Feature Oriented Programming (FOP) also handles static and dynamic cross-cutting. Static cross-cutting corresponds to the definition of state in the class refinement (i.e. class extension). Redefinition of the class hierarchy is not handled by all FOP implementations but, for instance, GluonJ implements that mechanism with the same *expressivity* as inheritance in Java. Dynamic cross-cutting is implemented as method overriding or method extension in OO programs. It follows the same semantics where a feature's implementation (e.g.: class refinement) can modify the behaviour of methods entirely (i.e. overriding) or modifying but using the behaviour defined in the base class (i.e. extension).

5.2 Heterogeneous *vs* Homogeneous *pointcuts*

The definition of homogeneous and heterogeneous *pointcuts* is rather simple. An homogeneous *pointcut* defines the same behaviour, i.e., *advice*, to more than one *join-point*. One example is the use of no unbounded quantification (e.g.: using *wildcards*) regarding the name of a method or the number its arguments. An heterogeneous *pointcut* is the opposite, i.e., it defines exactly one *join-point*.

The use of homogeneous or heterogeneous *pointcuts* depends on the nature of the problem. The most well known problem that takes advantages of the use of homogeneous *pointcuts* is Logging. Some authors argue that the use of homogeneous *pointcuts* can destroy the evolution of application since the addition of new classes or new aspects can be affected by these *pointcuts*.

A conditional *pointcut* is a mid-term case of both homogeneous and heterogeneous *pointcut*. It can be defined with or without (un)bounded quantification and it is evaluated at runtime given a conditional expression. For instance, an arbitrary advice is executed only if a boolean expression is true.

Table 5.2 shows the use of homogeneous, conditional and heterogeneous pointcuts in our case studies. We found no use for homogeneous pointcuts. As a matter of fact, we believe that the use of that kind of pointcuts is limited and there are few applications that can benefit from their use. Conditional pointcuts were used on the distributed memory parallelization to prevent all instances of the program to start the instrumentation code for time measurement, i.e., only allow one instance (e.g.: MPI_ID 0) to start and stop the internal clock. All other related parallelization features were introduced by heterogeneous pointcuts.

The implementation of homogeneous *pointcuts* using FOP tools is possible. The majority of FOP tools are based in code transformation, thus, it is simple to define a *pointcut language* similar to AspectJ, i.e., a syntax to allow unbounded quantification in feature's implementations. For instance, the tool we used to implement class refinements is based in bytecode transformation and allows unbounded quantification. As we found no use for homogeneous pointcuts, we did not explore that facet. Conditional pointcuts can be easily translated to heterogeneous ones. Hence, excluding the case of conditional pointcuts, we could find a relation between one pointcut/advice to one method extension.

The results of this comparison is extremely important since it reflects one of the assumptions of this dissertation, i.e., for the case studies implemented, we could substitute AOP by FOP.

	Homogeneous	*Conditional*	*Heterogeneous*
Sor	*No*	*Yes*	*Yes*
Moldyn	*No*	*Yes*	*Yes*
Ray Tracer	*No*	*Yes*	*Yes*
SparseMMM	*No*	*Yes*	*Yes*
JECoLi	*No*	*No*	*Yes*

Table 5.2: Heterogeneous *vs* homogeneous *pointcuts*. Comparison of Approaches.

5.3 Reusing

Reusing is an important property that we tried to achieve by enhancing modularization of applications. We had successfully achieved reusing of the base code since our parallelization features are well confined on an external module, i.e., class refinement. Our ultimate goal was to achieve reusing of the parallelization features as well, for instance, not to apply them to a single base program but on multiple distinct base programs. Given the different nature of each parallelization algorithm, we found difficult reusing the parallel features using the current parallelization model. More particularly, the tool used (GluonJ) keeps the semantics of Java inheritance, thus the parent class must be known at the time of the definition of the subclass. The other facet is the name of the method to refine, i.e., where to append the parallelization features. Both of these two *problems* could be overcome using a solution based in reflection, i.e., discovering classes at runtime that implement a specific interface, for instance, *OneWayInterface*. With this approach, the name of the classes and the methods could be discovered and the refinements could be applied over that class or classes. A simple case study was developed using this approach but, as expected, the performance gain was heavily penalised by the use of reflection.

AspectJ, on the other hand, allows some forms of reusing :

Abstract aspects and pointcuts : the use of this approach was the core of the work of the first AOP concurrency library, Conclib [CSM06]. Conclib implements common concurrency mechanism like one-way invocations and barriers. The mechanisms are implemented in abstract aspects defining abstract pointcuts. These aspects must be extended by the users of the library to specify the concrete implementation of each abstract pointcut.

***Pointcuts* over Interfaces** : AspectJ allows the introduction of pointcuts over methods defined in Interfaces. Hence, all classes that implements the methods of the Interface are *intercepted* by the pointcut. This approach is being explored in the parallelization of the JECoLi framework to allow reusing of the parallelization feature to cope with the introduction of new algorithms.

In terms of reusing, the AOP approach with AspectJ is currently more mature than our approach with class refinements. However, reusing in AOP and in particular with AspectJ is not a trivial task. Some authors claim that reusing with AspectJ is not always achieved because it is a mechanism that is tightly bounded with the base code because of the *join-point* model [MO04] and software design should be adapted to aspects [DFS04].

5.4 Composition of Aspects and Refinements

Composition is the ability to use multiple aspects or features (in this study implemented with class refinements) to build more complex solutions. AspectJ allows multiple aspects to be applied to an application. Since different aspects can intercept the same *join-point*, it introduces a mechanism named *declare precedence* to define the order of which aspects are applied. Given the nature of the *join-point* mechanism, aspects can intercept events on other aspects. This is called the aspect-interaction problem and it is one one of the main sources of bugs in AspectJ if no special techniques are applied [Tan10].

Composition is the base mechanism in FOP since it promotes incremental development by decomposing a solution of a problem in features. Features are then composed to create different versions of a program to meet different requisites. This idea is the core of product lines research where different combination of features lead to different programs. The order in which features or class refinements are applied dictates the semantics of the application[1]. Feature models are crucial to define the valid members of a product line and their constraints, thus it can be used to define the valid compositions of features in our case studies.

Feature Models represent an important mechanism to cope with the evolution of an application where new features may be developed. The JECoLi case study is the most representative since it supports several optimization methods with two different parallelization models that can be used together with different combinations of target architecture mappings.

5.5 Performance

In this section we present a set of benchmarks developed to compare the overhead of both tools used to implement separation of concerns in our study, i.e., AspectJ and GluonJ. Both allow static and dynamic redefinition of classes but, in these particular benchmarks, we are interested to measure the impact introduced by dynamic redefinition (e.g.: method overriding) since it was largely the main functionality used in our case studies. To situate the overhead of the tools, we also compare to Java Inheritance.

The execution environment and instrumentation code is the same as the one presented in the other benchmarks of this study.

Figure 5.1 shows the two classes created for this benchmark. Note that the classes are very simple because the main purpose is to isolate the over-

[1]Note that the order can dictate also the validity of the solution, i.e., it can lead to infinite recursion.

head of each tool. The class *Count* has only one method that increments a *long* variable. The *Bench* class instantiates a *Count* object and calls the method *incCounter()* in a loop. To minimize the possible overhead of the instrumentation code, the total number of iterations of the loop is large in the order of 10^9.

The aspect and the class refinement created for this benchmark are shown in table 5.3. Both are very simple and do nothing but redefining the method *incCounter()*. A third approach (not shown) is simply a class extension where the method *incCounter()* is overridden by inheritance. The base approach is the one presented in figure 5.1.

Table 5.4 shows the results using inline[2] enabled and disabled. The decision to disable *method-inlining* was to create the *worst* scenario, thus abstracting over the simplicity of this benchmark since the methods could be easily *inlined* by the JVM. With *inline* enabled, there is no overhead introduced by GluonJ and the Inheritance approach. However, it is interesting to point that the version with AspectJ introduced little overhead. Comparing with the time for executing each iteration of the cycle, all the approaches with except of AspectJ, spent an average of 8 *nanoseconds* per iteration[3]. The AspectJ version spent 11 *nanoseconds* per iteration.

Analysing the results for the version with *inline* disabled, it is interesting to see that again the AspectJ approach is the one that introduces more overhead. The result that we were not expecting was that the *Inheritance* approach did not introduce overhead. In this simple example, even with *inline* disabled, the *bytecode* was optimized to reduce the call from a subclass to its parent class. On the other hand, the GluonJ runtime system, as expected, implemented the refined method as a new method created at load time. The *super()* instruction in the method body corresponded to a call to the original method.

These low-level benchmarks were made to confirm our suspects of little overhead introduced by the tools of this study. Using standard optimization techniques, the code introduced by the tools can even be *inlined* to reduce the method-call overhead. However, it is important to assert that we are not saying that all functionalities of AspectJ share the same results. For instance, the use of context information on an aspect using *thisJoinPoint* or similar constructs can introduce great overhead because they rely on reflective mechanisms. Since those kind of constructs were not used in the implementation

[2]Inline is a mechanism that can replace method-calls by the body of the method to avoid the overhead of calling functions or methods.

[3]We simply divided the time spent by each approach by the number of iterations. This measure would be incorrect to calculate the time for each method call since we are not considering the code of the loop structure.

of our case studies, we did not study the actual overhead they introduce.

```
public class Count {
  long counter;

  public void incCounter(){
    counter++;
  }
}
```

```
public class Bench {

  public void work(){
    Count c = new Count();
    Instrumentor.addTimer("LowLevelBench:Computation");

    for(long i; i < LARGE_N; i++)
      c.incCounter();

    Instrumentor.stopTimer("LowLevelBench:Computation");
  }

}
```

Figure 5.1: Benchmark code.

a) Refinement

```
@Refine
public static class RC
  extends Count{

  @Override
  public void incCounter()
  {
    super.incCounter();
  }
}
```

b) Aspect

```
public aspect Bench {

 pointcut inc() :
  call(void Test.incCounter());

 void around() : inc() {
  proceed();
  }
}
```

Table 5.3: Class Refinement and Aspect. Both redefine the same method invoking the previous defined functionality. Besides simple, both enforces the GluonJ and AspectJ runtime systems to *change* the base functionality.

	Inline		**No Inline**	
Implementation	Avg.	Std. deviation	Avg.	Std. deviation
Base	8.040	0.007	39.093	0.062
GluonJ	8.043	0.012	76.411	0.118
AspectJ	10.916	0.012	112.535	0.123
Inheritance	8.038	0.003	39.028	0.017

Table 5.4: Average results for 10 executions with and without *Inline*. Time in seconds.

Chapter 6

Conclusion

This dissertation addressed the study of separation of concerns in the development of parallel applications. From the experience with Aspect Oriented Programming (AOP) and parallel programming, we developed a parallelization model based in Feature Oriented Programming (FOP) to be similar to AOP, easier to use, maintaining the same degree of separation of concerns. In our model, parallelization concerns are introduced as features. Each feature encapsulates several extensions to the base program in the form of : a) class refinements, i.e., class extensions where new behaviour is attached to the base program in a rewriting mechanism; b) new classes. Features can *refine* other features by *refining* class refinements or new classes. Our model relies in a feature composition step to specify which features must be applied in which order to the base program. To cope with this composition mechanism where features can depend on other features, we explored two alternatives equally valuable : Feature Diagrams and Feature grammars.

Several case studies were implemented to validate our model. We highlight the parallelization of the Java Evolutionary Computation Library(JECoLi) framework because it is a medium-sized framework (approximately 120 classes and 5000 LOC) used in the Biological computation field and is currently used in production. Since features encompass parallelization concerns and are developed apart from the base code, they can be developed independently. Equally important was the layered architecture where some features are only *services providers* (e.g.: Abstract Migration) to other features and some others map the parallelization model to a specific and optimized target execution platform.

The implementation of the case studies leveraged a downside of our current FOP model. Reusing is currently the *Achilles' heel*. Separating the base code from the parallelization feature enables reusing the base code but we found hard to create a parallelization feature that can be reused by other case studies. One of the reasons is that parallelization features are specific to the domain of the problem and the other reason is that our approach refines a class by means of Object-Oriented inheritance extending the behaviour of specific methods. To cope with Object-Oriented rules for inheritance, we need to know the name of the class to extend. We implemented a *proof-of-concept* tool that uses reflective technology but we found that the performance overhead was to high to be discarded.

A comparison of our approach with the implementation of the same case studies using AOP was also presented. We found extremely important to understand what AOP features were being used and if those features could be replaced by the simpler FOP model. We found that, from the implementation of the case studies, with the exception of AOP allowing *pointcuts* over Interfaces, all other AOP mechanisms were based in heterogeneous *pointcuts*, thus easily implemented in FOP. In terms of performance, we showed by the implementation of several case studies and low-level benchmarks, that the overhead is minimum compared with code developed with no extension or modularity in mind.

6.1 Future Work

In terms of future work, we are deeply interested to see how our parallelization model copes with the evolution of the JECoLi framework. More particularly, we are interested to see not only the introduction of the new algorithms but also the the maintenance and evolution of the parallelization features. For instance, we are developing GRID mappings to the island model to use the potential provided by GRIDs.

The development of new case studies is equally important to evolve our parallelization model. We are studying the use of FOP to introduce not only parallelization features but we are also interested in the design of algorithms and data structures based on composition of features. For instance, molecular dynamics (MD) algorithms are a great source of interest by the parallel community because there are several known parallel algorithms depending on how particles are represented, i.e., depending on the data structures used to represent the particles. Each representation can have a performance advantage over the others depending on the parallel architecture (e.g.: GPU or shared memory). Using FOP to cope with this variability is one of the next steps.

As pointed out, reusing parallel features is currently the downside of our approach comparing with AOP. The investigation of a new evolution of our model is one of the next steps to cope with re-usability of parallelization features.

The integration of all the steps in our parallelization model is another future step of this research. We would like to have a tool or environment that encompasses parallelization layers and composition. This mechanism could integrate other tools to check valid composition of features to create the concept of product line.

Bibliography

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.

[AKL09] S. Apel, C. Kastner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[AL07] A. Agarwal and M. Levy. The kill rule for multicore. In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 750–753. IEEE, 2007.

[ALS06] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM Press.

[AS10] M. Almeida and João L. Sobral. Separation of Concerns in Parallel Applications with Class Refinement. In *INFORUM 2010 : Proceedings of the 2th symposium on Informatics*, 2010.

[Ban07] Purushotham 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*, page 3, New York, NY, USA, 2007. ACM.

[Bat05] D. Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.

[Bat06] D. Batory. A tutorial on feature oriented programming and the ahead tool suite. *Generative and Transformational Techniques in Software Engineering*, pages 3–35, 2006.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311. ACM, 1990.

[BCF+99] M. Baker, B. Carpenter, G. Fox, S. Hoon Ko, and S. Lim. mpiJava: An object-oriented Java interface to MPI. *Parallel and Distributed Processing*, pages 748–762, 1999.

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 177–189, New York, NY, USA, 2005. ACM.

[BDNW05] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: controlling visibility of class extension. *Computer Languages, Systems & Structures*, 31(3-4):107–126, October 2005.

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. pages 122–131. 2003.

[BK00] J. M. Bull and M. E. Kambites. Jomp—an openmp-like interface for java. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 44–53, New York, NY, USA, 2000. ACM.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *5th International Conference on Software Reuse*. Citeseer, 1998.

[BO92] D. Batory and S. O'malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[Bon02] D. Bonachea. GASNet Specification, v1. 1. 2002.

[CDC+99] W.W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and language specification.* Citeseer, 1999.

[CDK$^+$01] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[CE98] K. Czarnecki and U. Eisenecker. Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models. *Department of Computer Science and Automation*, 1998.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CN03] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 364–376. Springer-Verlag New York, Inc., 2003.

[CSM06] Carlos A. Cunha, João L. Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 134–145, New York, NY, USA, 2006. ACM.

[DBR04] Jacob Neal Sarvela Don Batory and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[DBS03] Jia Liu Don Batory and Jacob Neal Sarvela. Refinements and multi-dimensional separation of concerns. In *SESSION: Requirements engineering and design*, pages 48 – 57, 2003.

[DFS04] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, page 150. ACM, 2004.

[Dij78] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.

[GP] B. Goetz and T. Peierls. *Java concurrency in practice.* Pearson Education India.

[GS09] Rui C. Gonçalves and João L. Sobral. Pluggable parallelisation. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 11–20, New York, NY, USA, 2009. ACM.

[HG04] Bruno Harbulot and John R. Gurd. Using aspectj to separate concerns in parallel scientific java code. In *AOSD 2004 Conference*, 2004.

[JCSG99] Glenn Judd, Mark Clement, Quinn Snell, and Vladimir Getov. Design issues for efficient implementation of mpi in java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 58–65, New York, NY, USA, 1999. ACM.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming - 11th European Conference*, volume 1241, pages 220–242, June 1997.

[Koc05] Geoff Koch. Discovering multi-core : Extending the benefits of moore's law. *Technology@Intel Magazine*, 2005.

[KVBP08] Michael Klemm, Ronald Veldema, Matthias Bezold, and Michael Philippsen. A proposal for openmp for java. pages 409–421. 2008.

[MO04] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136. ACM, 2004.

[MRFR06] R. Mendes, I. Rocha, E.C. Ferreira, and M. Rocha. A comparison of algorithms for the optimization of fermentation processes. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2018–2025. IEEE, 2006.

[NC08] Muga Nishizawa and Shigeru Chiba. A small extension to java for class refinement. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 160–165, New York, NY, USA, 2008. ACM.

[NR98] R.W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par79] D. L. Parnas. Designing software for ease of extension and contraction. *Software Engineering, IEEE Transactions on*, SE-5(2):128–138, 1979.

[Pre97] C. Prehofer. Feature-oriented programming: A fresh look at objects. *ECOOP'97 - Object-Oriented Programming*, pages 419–443, 1997.

[PRS10] Jorge Pinho, Miguel Rocha, and João L. Sobral. Pluggable Parallelization of Evolutionary Algorithms Applied to the Optimization of Biological Processes. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 395–402. IEEE, 2010.

[RMM+08] M. Rocha, P. Maia, R. Mendes, J.P. Pinto, E.C. Ferreira, J. Nielsen, K.R. Patil, and I. Rocha. Natural computation metaheuristics for the in silico optimization of microbial strains. *BMC bioinformatics*, 9(1):499, 2008.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8, New York, NY, USA, 2001. ACM.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[SM08] Edgar Sousa and Miguel 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*, pages 1–6, New York, NY, USA, 2008. ACM.

[Sob06] J.L. Sobral. Incrementally developing parallel applications with aspectj. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[Sou09] Edgar M. Sousa. Incrementally gridifying scientific applications. Master's thesis, Universidade do Minho, 2009.

[Tan10] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the Eighth International Conference on Aspect-Oriented Software Development*, pages 37–48. ACM, 2010.

[YSP$^+$98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.