Strategies for Optimization of Parallel Programs

Alcides Fonseca

Thesis Proposal

for the partial fulfillment of the requirements for the degree of Doctor of Philosophy in Informatics Engineering.

Under supervision of Prof. Bruno Cabral
Department of Informatics Engineering
University of Coimbra

Coimbra, August 30, 2012
Abstract

Multi-core processors are present in most forms of computing, from a pocket-size smartphone to supercomputers. Consequently, parallel and concurrent programming has re-emerged as a pressing concern for everyone interested in exploring all the potential computational power in these machines. Writing parallel, and specially concurrent, programs is not a trivial task as it requires a different reasoning model about the program. Moreover, most of the existing computer is sequential, and does not take advantage of the underlying parallelism of multicore CPUs. The proposed work intends to improve and further advance techniques that automatically parallelize a program. The result of this work should provide better ways of generating parallel programs that are faster and correct.

Keywords

Parallel, Concurrent, Heterogeneous Platforms, Compiler, Runtime, Programming Languages

Estratégias para Optimização de Programas Paralelos

Sob a orientação do Prof. Bruno Cabral,
Professor Auxiliar Convidado,
Departamento de Eng. Informática,
Universidade de Coimbra

Keywords

Programação Paralela, Programação Concurrente, Plataformas heterogéneas, Compilador, Runtime, Linguagens
Contents

List of Acronyms ix

1 Introduction 1
   1.1 The Rise of Parallel Programming 1
   1.2 Advancing Parallel Programming 2
   1.3 The Æminium Project 3
   1.4 Contributions 3
   1.5 Organization 4

2 State of the Art 5
   2.1 Concurrent Models 5
      2.1.1 Declarative Concurrency 5
      2.1.2 Message-Passing Concurrency 6
      2.1.3 Shared-State Concurrency 6
   2.2 Parallelism Shape 6
      2.2.1 Symmetrical Parallelism 6
      2.2.2 Asymmetrical Parallelism 7
   2.3 Automatic Parallelism 8
      2.3.1 Parallel Collections 8
      2.3.2 Composition of Library-based Concurrency 9
      2.3.3 Loop Parallelization 9
      2.3.4 Parallel-by-Default Languages 10
   2.4 Optimization of Concurrent Programs 10
      2.4.1 Static Analysis Optimizations 11
      2.4.2 Runtime Optimizations 11
   2.5 Summary 12

3 Research Objectives 13
   3.1 Potential Parallelism in Sequential Programs 13
   3.2 Balancing the Granularity of Parallel Programs 14
   3.3 Exception Handling in Implicit Concurrent Programs 16
   3.4 Summary 16

4 Current Work 19
   4.1 Potential Parallelism in Sequential Applications 19
      4.1.1 Definition of Potential Degree of Parallelism 19
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Symmetrical Parallelism in Master-Worker models</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Asymmetrical Parallelism</td>
<td>7</td>
</tr>
<tr>
<td>4.1</td>
<td>Architecture of the extraction of parallelism from sequential programs</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Box plot of the distributions of accuracy of several classifiers</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Box plot of the distributions of costs of several classifiers</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Gantt chart for the Thesis plan</td>
<td>37</td>
</tr>
</tbody>
</table>
List of Tables

2.1 List of Automatic Compilers ........................................ 9
4.1 List of features .......................................................... 25
4.2 Features rank using Information gain ............................... 27
5.1 Target Conferences ...................................................... 39
List of Acronyms

**PL**  Programing Languages
**CPU**  Central Processing Unit
**GPU**  Graphics Processing Unit
**GPGPU**  General Purpose GPU programming
**GCD**  Grand Central Dispatch
**STM**  Software Transactional Memory
**JML**  Java Modeling Language
**EH**  Exception Handling
1

Introduction

This chapter introduces the topic of this thesis as well as its goal and main contributions.

1.1 The Rise of Parallel Programming

CPUs have suffered significant changes during the last 15 years. The most significant evolution was the move from single-core to multi-core architectures, which had a profound impact on the way software is designed, implemented and executed. Today, the fundamental programming model is imminently concurrent.

This change was motivated by physical limitations related with chip manufacturing. Heating and high energy consumption made the increase of the frequency of CPUs impossible[45]. In order to keep improving processors, manufacturers have been increasing the number of cores inside each one. In this new architecture, the number of instructions executed per second increases vastly by a factor of up to the number of cores. While consumer-grade computers have 2 and 4 cores, production server machines have up to 24 cores. Such computer has the potential to execute a program in a 1/24 of the time it would take in a single-core processor.

Despite the encouraging speedup factor, the performance of most existing programs is not automatically improved with the new hardware. Most programs are single-threaded, or use multiple threads for IO and event loops, such as controlling Graphical User Interfaces or listening to several sockets at the same time.

In order to take advantage of this new generation of processors, programmers had to write parallel programs, in which several parts could be executed at the same time. Writing parallel and concurrent programs is not a trivial task. Human thinking has a sequential
CHAPTER 1. INTRODUCTION

nature. Programmers read and write code linearly, and instinctively model the sequence of instructions in the code as sequential in execution order. When thinking of parallel programs, a person has to consider all the possible interleaves between instructions. For complex programs this is not feasible as there are too many possible states[41].

Frequently in parallelism, different threads need to access the same resources, or may need to perform work in a certain order. In both cases, there is the need for synchronization mechanisms to guarantee that there are no inconsistencies in the program. Due to the complexity of programs, these synchronization mechanisms, such as locks, are not properly used and programs may suffer from problems such as dead-locks, live-locks and other data races. In these cases, programs may block infinitely, either paused or wasting CPU cycles. It is also possible for programs to reach a inconsistent state, possibly leading to data corruption, operation errors and crashes.

1.2 Advancing Parallel Programming

Writing parallel programs is not easy. With the advent of multi-core processors there has been a renewed interest parallel programming as an optimization method. Thus, new tools have been developed that aim to ease the writing of parallel programs. Some examples include compiling tools, such as OpenMP[19], runtime libraries, of which Grand Central Dispatch[38] is an example, and debuggers, such as KDB[8].

Another approach is to embed parallelism in the programming language itself. Several extensions to existing languages(Cilk[5], DPJ[6]) were proposed adding new parallel constructs, such as parallel regions. Also, new languages have been created from scratch which were designed to extract as much parallelism as possible from the source code. Fortress[42], X10[17] and Chapel[15] were designed to be parallel by default, in which all language constructs behave in a parallel manner as much as possible without the need for programmers to explicitly create parallelism. These three languages all have in common the target niche: High Performance Computing. It is important to notice that these languages have not yet reached the mainstream level and they are mostly used as research tools.

The languages mentioned so far focus more on Embarrassingly Parallelism, in which there are no concurrent accesses to the same memory address by different parallel processing units.

Æmininium[44], on the other hand, is a language that was designed to be concurrent-by-default. In Æmininium, programmers annotate parameters of functions with requirements regarding the uniqueness of alias in the program to the object being passed. The compiler than parallelizes as much of the program as possible, guaranteeing that the annotated
requirements are not violated. The resulting program will have as much parallelism as possible, with the concurrent accesses limited by the annotations. Furthermore, this language has the advantage of generating deadlock-free programs, thus improving the quality of programs.

1.3 The Æminium Project

Æminium is a project to explore the idea of a language concurrent-by-default. The goal of the language is to make it easier to write parallel programs and have them execute faster than they would otherwise, guaranteeing the correctness of the program.

The project is a joint effort of Carnegie Mellon University, University of Coimbra and University of Madeira. Carnegie Mellon is responsible for developing the compiler from the Æminium language to Java source code. University of Coimbra has been working on the Runtime and how to speedup parallel programs. University of Madeira has been working on the verification of the techniques employed in Æminium.

Previously, we, the team in University of Coimbra, have developed the Æminium Runtime, the library that powers the parallelism of Æminium and executes tasks using a work-stealing policy. We have also worked on a GPU extension to the compiler and runtime that executes data-parallel operations on collections on either the GPU or CPU.

The Æminium project plays an important role in this thesis. Æminium is one of the newest languages that shares the parallel-by-default semantics with X10, Chapel or Fortress. While being able to extract the same type of parallelism, it can extract even more concurrency. Thus, it can be used to test techniques that would work on other languages/runtimes and new ones as well.

1.4 Contributions

The proposed thesis is that parallel programming can be done more efficiently. Programs should be easier to express in their parallel form, and the execution should have an high benefit in terms of performance.

The expected contributions of this thesis are:

- An evaluation of the potential parallelism implicit in existent sequential programs
- New methods and techniques for improving the automatic parallelization and the execution of parallel programs.
CHAPTER 1. INTRODUCTION

- The verification that automatic parallelization tools are deterministic and do not change the semantics of the original program.
- A framework that incorporates the techniques previously mentioned and is ready for public use.

The second contribution is very generic and allows for different approaches. One example, detailed later in Chapter 4, is to automatically decide if offloading some parts of a program to the GPU will be beneficial or not. Other intended approaches are to automatically define the best granularity for parallel tasks, or to choose the best scheduling order of tasks.

The subject of parallelization is very broad, specially considering the multitude of parallel hardware available. The scope of this thesis is limited to the parallel processors available in commodity hardware, such as multicore CPUs, and programmable GPUs since they are becoming more popular in both consumer and server-grade systems. The focus is also in parallelism inside a single machine, since considering distributive parallelism is another topic with its own characteristics and challenges.

1.5 Organization

The remaining of this document is organized as follows: Chapter 2 introduces the state of the art on the topic. Chapter 3 introduces the research goals. Chapter 4 explains the approach taken in the current work and introduces some preliminary results. Chapter 5 presents the work-plan for the remaining of the thesis and the dissemination strategy. Finally, Chapter 6 concludes the document.
In this chapter, we will start by studying the basic concepts of parallelism which will be later related with the techniques that explore the properties of those concepts. Then we will look at state-of-the-art methods for parallelization of code. Finally, we will address the most used and more efficient techniques for optimization of parallel programs. This chapter will provide the groundwork for later improving existing techniques.

2.1 Concurrent Models

Van Roy[47] presents three main approaches for expressing concurrency in computer programs: Declarative, Shared-State and Message-Passing Concurrency. These models will be studied as their characteristics are important to understand what can be optimized.

2.1.1 Declarative Concurrency

In Declarative Programming, the control flow of the program is not described directly, it is the consequence of the computational logic of the program. Declarative programs have different possible executions as long as they have the same logical outcome. This outcome can be that all possible executions do not terminate, or they terminate with equivalent results. These conditions are the same as saying that the program is deterministic.

The approach of Declarative Concurrency is a special case of Declarative Programming. The program is still expressed in the same way, without explicitly expressing the order of executions. Each part of the program can be executed in its own thread as long as the result of the program is not changed, i.e., the deterministic behavior is guaranteed. This model can also be considered as DataFlow programming, since what defines the
program is the logical connection between variables, and not the lexicographical order in which operations are written in the source code. CC++[16] is an example of such language.

### 2.1.2 Message-Passing Concurrency

In Message-Passing Concurrency, threads are isolated and independent of each other. Each thread has its own memory space and, according to the model, does not access any other memory address. In order to communicate and share data, threads may only communicate using messages. The message handling can be either synchronous or asynchronous.

The Actor Model follows this concurrent model, and each actor (in its own thread of execution) has an inbox of asynchronous messages and processes them during its lifetime. This model is non-deterministic but can be used in a deterministic way[22].

### 2.1.3 Shared-State Concurrency

In Shared-State Concurrency, different threads of execution share the same resources, mainly memory. Since multiple threads may change the same resource at the same time, this model frequently requires synchronization of data accesses and for coordination of work. Performance-wise, this is a overhead of this model, but allows for more complex forms of concurrency and frequently faster.

Furthermore, the general nondeterminism of this model can cause problems with the consistency and validation of the program. Verification of concurrent programs is important to reduce the uncertainty in programs following this concurrent model.

### 2.2 Parallelism Shape

Another important characteristic of a parallel program relates to the shape of the parallelism, which also influences the kind of optimizations that can be done.

#### 2.2.1 Symmetrical Parallelism

Programs that are “embarrassingly parallel” are easily divided in subtasks that execute the same code, but in different memory spaces. The implementation usually follows a Master-Worker model (Figure 2.1) in which the master initiates the workers, manages
their work, receives the output and terminates all the workers. The workers all execute
the same work, hence the symmetry, most of the times without knowledge of the other
workers.

![Symmetrical Parallelism in Master-Worker models.](image1)

**2.2.2 Asymmetrical Parallelism**

On the other hand, there is asymmetrical parallelism (Figure 2.2) in which the com-
putation cannot be equally divided, and different concurrent tasks execute different
instructions. This kind of parallelism is harder to optimize as tasks take more or less
time to execute and often result in unbalanced distribution of tasks to processors.

![Asymmetrical Parallelism.](image2)

There is a more specific asymmetrical model that has a particular recursive structure.
In the Fork-Join model, tasks can be created for recursive calls of methods, very useful
in the exploration of tree-like structures. This model has been popularized by Cilk[5]
and the Java ForkJoin framework[33].
2.3 Automatic Parallelism

Writing sequential programs is easier than writing parallel ones. Knowing what to parallelize, how to manage the control flow and how to protected shared data are extra concerns for programmers that distracts them from the conceptual idea they are expressing. Automatic Parallelism is important to lift this burden from the programmer and still get the performance benefits of having parallel code.

In this section, we will present the most relevant techniques for automatically generating parallelism will be presented. These are not the only ones, but they are the most important considering that the scope of this thesis is focused on multicore CPUs and not custom parallel hardware nor distributed systems.

2.3.1 Parallel Collections

Several languages and libraries explore the data-level parallelism by including operations on certain data structures (arrays, matrices, trees, etc). These collections can be found in functional languages (Haskell[14]) and object-oriented (Java and Scala[39]).

These approaches take advantage of operations over arrays that can treat subsets independently and in parallel. The only decision made by the programmer is to use the parallel version of data structures. The parallelism is automatically generated, based on the rules defined by the library or language authors. When there is the need for custom operations on these structures that have a more complex synchronization scheme, Parallel Collections may not be suitable. An example of Scala Parallel Collections is shown in Listing 2.1 calculating PI through a Monte Carlo Simulation. Notice the .par method call on the Range, converting a regular list into a parallel version of a list, in which methods such as map, foreach and so on are implemented in parallel.

```
import scala.util.Random
val RESOLUTION = 100000000
val r = new Random
val c:Double = Range(0,RESOLUTION).par
.map(_ => (r.nextDouble, r.nextDouble))
.filter( p => p._1 * p._1 + p._2 * p._2 < 1 ).length
println(c * 4/RESOLUTION)
```

8
2.3. Composition of Library-based Concurrency

Several parallel and concurrent patterns can be extracted into libraries, as it is the example of the aforementioned Parallel Collections. However, then a certain program requires interaction between two or more concurrent data structures, combining off-the-shelf solutions may not be the most performant solution.

Reagents[46] solve this problem by defining some properties that concurrent data structures must support, in order to interoperate with each other. The idea is to expose fine-grained concurrency operations that allow for more custom connections between concurrent structures.

Reagents must define 3 operations: choice, which will receive the first response from two reagents; sequencing, in which two atomic operations can be merged into one single atomic operation maintaining the order; and pairing, in which two operations are also merged atomically, but with no guarantee of order. These 3 operations allow for library authors to support composability between their library and third party ones.

2.3.3 Loop Parallelization

Loops, specially the ones iterating over data, are typically good candidates for being parallelized. Table 2.1 shows a list of the most relevant tools that perform automatic loop parallelism with their input and target languages as well as the technique they use.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Source Code</th>
<th>Target</th>
<th>Technique for Loop Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Par4All[2]</td>
<td>C, Fortran, Java and Python</td>
<td>OpenMP, Cuda, OpenCL or MPI</td>
<td>Symbolic Analysis</td>
</tr>
<tr>
<td>Cetus[20]</td>
<td>C</td>
<td>OpenMP, Cuda, MPI</td>
<td>Symbolic Analysis</td>
</tr>
<tr>
<td>Pluto[7]</td>
<td>C</td>
<td>OpenMP</td>
<td>Polyhedral Model</td>
</tr>
<tr>
<td>Intel C++ Compiler[31]</td>
<td>C++</td>
<td>OpenMP</td>
<td>Symbolic Analysis</td>
</tr>
<tr>
<td>SUIF[49]</td>
<td>Fortran and C</td>
<td>Parallel C</td>
<td>Symbolic Analysis</td>
</tr>
</tbody>
</table>

Table 2.1: List of Automatic Compilers

The typical loop parallelization process includes a first pass to expose parts of the code that can be parallelized by applying methods such as constant and forward folding. A second pass identifies all the possible areas where the program can be parallelized. Then, algorithms find the best regions to be executed in parallel. The Polyhedral Model[10] is very used in this step. After the split into different parallel threads has been made, the
result is validated using static verification tools to ensure that the output of the parallel version is the same. Finally, the parallel code is compiled to a parallel architecture like Pthreads, OpenMP, CUDA or OpenCL.

### 2.3.4 Parallel-by-Default Languages

The previous methods for automatic parallelization consider all as input source code written in sequential programming languages. More recently, new languages have been created that are parallel by default. These languages have been designed with parallelism in mind, so most language constructs such as for-cycles, tuples and function calls have parallel semantics.

X10\[17\], Chapel\[15\] and Fortress\[43\] are examples of such languages, targeting High Performance Computing. The latter has a syntax based on mathematical notation and takes advantage of the independence in mathematical operations to automatically parallelize parts of the program.

Recent studies have been focusing on having Concurrency-by-Default in Programming Languages. The Par Monad in Haskell\[36\] has been used to automatically generate concurrent programs by scheduling the program in a way that protects concurrent memory accesses guaranteeing determinism. Æminium\[44\] is a language designed from scratch to support concurrency by means of data permissions. Æminium uses a Declarative Concurrency approach in which the programmer writes what the code is intended to do, without caring for the order of execution. By annotating variables and method signatures with access permissions, the compiler is aware of what variables can be accessed with or without synchronization.

These approaches perform static analysis on the source code and generate tasks with dependencies between them. These dependencies are generated from the data-flow in the program, making the approach deterministic. During execution, there is a runtime system that schedules these tasks in the available hardware. Current runtimes use a mix of threadpools and work-stealing techniques.

### 2.4 Optimization of Concurrent Programs

There are already several works on optimizing concurrent programs in order to take the most advantage of the parallel hardware. This section identifies relevant studies that will be used as starting points to further improve the performance of concurrent programs. We will first address optimizations at compile-time, using static analysis, and then optimizations done at runtime.
2.4. OPTIMIZATION OF CONCURRENT PROGRAMS

2.4.1 Static Analysis Optimizations

As mentioned in the last section, the Polyhedral Model\cite{10} can be used to automatically infer parallelism from loops, and works on nested loops. One of the trivial optimizations it does is to try and parallelize the outer level loops first. By having the largest parallel tasks as possible, the compiler is reducing the overhead in task switching and will try to adjust the number of threads to the number of processors on the machine.

Another optimization done by these automatic parallelization tools is to use as much local memory inside each task and only update the global state at the end, thus reducing the memory access collisions during the main execution of each task. This type of transformation is typically called a reduction, with special OpenMP syntax, and is only done if semantically equivalent of the original code.

Symbolic Analysis\cite{28}\cite{4} is an alternative to the Polyhedral Model. The general idea is model the data and flow dependencies in the program to understand what can be parallelized. This approach can understand function side-effects such as global IO that can limit the parallelism in the program. While parallelizing loops, symbolic analysis allows to avoid the computation of counter variables and instead introduce all the values it would take statically into the program. Another advantage of this technique is that allows to estimate the computation effort of each task, which allows to decide between several equivalent code alternatives. This estimation is also helpful for deciding the number of tasks in which to partition a given loop.

2.4.2 Runtime Optimizations

While in some trivial cases it might be possible to adjust the number of tasks to the number of processing threads, in most parallel programs the number of tasks varies and can be lower or higher than the number of threads. The naïve approach is to create one thread per task, but the thread scheduler will occur in a overhead to context-switch between all running threads. If the application is very memory-intensive, it might not be feasible to have in memory everything required for all threads. The most popular solution is the usage of a threadpool that limits the number of running threads, and queues the tasks in excess.

Classical threadpools have a bottleneck because of the synchronization required when taking tasks from the queue, in order to avoid executing the same task twice. The most popular and successful solution is to use work-stealing threadpools, which powers the schedulers of Cilk\cite{5}, Java ForkJoin\cite{33}, Microsoft Task Parallel Library\cite{35}, Grand Central Dispatch\cite{3} and many more. In this approach there is one queue per thread, and
when a thread empties its own, it steals its next task from the other queue of another. This approach minimizes the conflicts when accessing the same task in a queue.

While the overall technique for Work-Stealing is widely accepted, there is still ongoing research on the best policy on stealing work\cite{27}\cite{21}. The task to steal may be chosen according to its position on the queue, the influence of the task on the continuation of the program, on the parent-child relation of tasks, or even at random. The best policy is also dependent of the nature and structure of the programs.

Another approach to optimize parallel programs is Lazy Task Creation\cite{37}. By using this approach, the decision to parallelize something, in contrast to execute it inline, is done at runtime. This decision is based on the CPU load at the moment.

In some cases, the duplication of tasks\cite{1} can be useful to speedup the execution of tasks that depend on a previous light one. By inlining the latter in all dependent tasks, the program execution time may be reduced.

A different approach is to decide the scheduling statically using genetic programming techniques. While it can generate optimal solutions, it requires a good heuristic for the execution time of a program, and it takes time to reach the optimal solution.

2.5 Summary

In this chapter we have seen different parallelism models, of which the Shared Memory is the most popular for performant applications. We also have seen techniques for automatically parallelizing code. These techniques can be library-based or applied during compilation. Of the latter, the Polyhedral Model and Symbolic Analysis are the most relevant models for automatically paralleling loops. More recently, there have been studies on Parallel and Concurrent by Default languages that have parallel semantics in the language constructs.

Finally, for managing the execution of tasks threadpools are still the most widely used method, although there are several techniques for choosing the best scheduling of tasks. Choosing the right granularity for tasks is the other main issue in optimizing the execution of concurrent programs.
Writing parallel and concurrent programs is not an easy task, requiring more effort than to write sequential code. Programmers traditionally have to specify which parts of the program execute in parallel, how they synchronize and communicate among them, as well as protect the program from data races such as deadlocks.

Our main goal is to reduce the difficulty in doing parallel and concurrent programming. Programmers should be able to write and debug parallel programs in a much clear way. Those programs should also execute faster than their sequential counterpart and the process should be as automatic as possible.

Given the broad scope of this goal, we feel the necessity to narrow it down to more concrete questions. The order of these questions is important as the outcomes from early ones may influence the approach in later ones. It is also important to remember that the focus is on single-machine multicore CPUs parallelism and programmable GPUs available in commodity hardware.

### 3.1 Potential Parallelism in Sequential Programs

There are two main approaches to integrate parallelism in PLs. The first and more traditional approach is to explicitly create parallel execution threads using language constructs. A more recent and subtle approach is to parallelize implicitly, in which the compiler decides automatically what should execute in parallel.

Despite the interest and effort put in automatic parallelization in PLs, there have not been any studies showing how much latent parallelism can be explored in sequential programs. We have a huge legacy of source code in several programming languages from
CHAPTER 3. RESEARCH OBJECTIVES

which we could study how much parallelism is hidden in those programs. This is important for three reasons: firstly, to evaluate if these techniques can generate parallelism from general purpose programs; secondly, to generate parallel programs that may have a better performance; thirdly, to gather information about the parallelism in existing irregular programs to try and improve their performance.

To answer this question, it is first necessary to define a metric that describes how much parallelism a program can have. The shape of parallelism extracted by automatic techniques is an irregular one. In some simple cases, it is trivial to measure it, such as master-worker and one-thread-per-socket. However when a program is fully parallelized several of this simple patterns may arise, as well as more complex ones.

As we have seen in Section 2.2, task and dependencies between them can form an asymmetrical and unbalanced graph. A starting point to measure parallelism may be to access some properties of this graph, such as the average width of the graph.

Several techniques generate different granularity of tasks. Most of the techniques try to create large tasks to avoid context-switching overheads. For our purpose, the goal is to have a fine grain granularity with parallelism at the instruction level. Having small tasks allows to evaluate more possibilities by merging them together and achieving, if necessary, larger sizes. But to be able to do that, the parallelization technique should find as much parallelism as possible.

3.2 Balancing the Granularity of Parallel Programs

Parallel programming is important today because it allows speeding up a program by using multiple processing units at the same time. However the relation between the parallel task and the hardware threads do not always match directly.

Parallel and Concurrent programs may have an irregular shape that changes during the execution of the program. In there can be mismatches between the computer architecture and the program tasks. Current approaches provide a runtime that can scale to any number of tasks at the cost of spending CPU time and memory in context-switching operations.

When generating automatic symmetrical parallelism, the number of tasks running at a given time should fit the number of hardware threads for better performance. If the number of tasks is lower, there will be unused hardware resources that could be used to further improve the performance of the program. If the number of tasks is too high, then there will be some overhead in context-switching. Thus, achieving the right granularity is very important for a better performance.
3.2. BALANCING THE GRANULARITY OF PARALLEL PROGRAMS

If there is asymmetrical or heterogeneous parallelism, each task is different from another. Tasks may contain more instructions, use more memory, have more branching conditions and include IO operations. Tasks can have a certain size, which can be the number of instructions or an estimated execution time. For automatic parallelization techniques to produce performant programs, the tasks should be organized in a way that maximizes the usage of the hardware resources and minimizes the context switching. Given the high number of combinations possible, easily achieving the better arrangement of tasks for execution is also important for producing performant programs.

Furthermore, in heterogeneous computing, there is also irregularity on the hardware side, having processing units with different architectures, memory and parallel execution models.

The different architecture and hardware specifics have impact on the type and details of the parallelization being made. In order to optimize the parallelization, these aspects should be taken into account. The optimization can be done in several ways, including:

- **Reordering of scheduling**: When there are more tasks than hardware processing units, the order in which they are executed does not matter, assuming they are deterministic parallel with the right dependencies. Thus, the execution of tasks can be reordered to optimize the execution, leading to a decreased memory usage and less synchronization conflicts, both of which will improve the performance of the program.

- **Merging of parallel units**: When the number of tasks largely exceeds the number of available CPU resources, the context-switching between tasks is prejudicial to a short execution time. It is possible to consider two parallel tasks as only one, executing them sequentially. By avoiding parallelism in those cases, we are optimizing the program to a given hardware.

- **Decide how to divide the work in heterogeneous hardware**: Different tasks may execute in parallel in different physical processing units. For instance, GPUs have an higher throughput but a higher latency as well and it brings only a speed-up in certain conditions. By looking at the program by instructions, memory use and CPU load, it should be possible to take the best decision on which platform should that unit be scheduled to.

The goal of this work, and the most important part of this thesis, is to develop a method to automatically decide the better granularity, either during compilation or execution, that achieves a good, if not the best, performance. This is the major expected contribution to answer the second Research Question.
CHAPTER 3. RESEARCH OBJECTIVES

3.3 Exception Handling in Implicit Concurrent Programs

Exception Handling (EH) is an aspect so important that most general-purpose PLs must support it. Concurrent EH is even more challenging as the sequential nature of the code cannot directly represent all possible concurrent exceptional executions and corresponding repercussions of each one in other tasks.

Existing solutions for Concurrent EH require programmers to explicitly describe the concurrency structure of the program. When writing programs for automatic parallelization, the structure of the program is not clear. In fact, the same source code can generate several programs, each with its own concurrent structure.

In these cases, the traditional EH mechanisms cannot be applied because of side-effects in exceptions after parallelization, and existing Concurrent EH require the structure to be described in the source.

When programmers write code to handle exceptions, they assume that the following instructions inside the protected block will not execute until the exception is properly handled. When programs write explicit parallel programs, they work under different assumptions and have to write extra code with possible extra synchronization to guarantee the semantics of the operations and a consistent state.

With implicit parallel languages, programmers do not know how the parallelism will be done, and therefore cannot protect the execution explicitly. Programmers also cannot assume that the behavior of sequential programs because while one thread might throw an error, another might already be executing the following instruction.

Existing solutions are therefore not applicable for this kind of languages and a new approach is in need. This is also another goal of this work that aims to improve how programmers write parallel and concurrent programs.

3.4 Summary

In this chapter we have introduced three goals for this thesis. Firstly, to measure the implicit parallelism in existing sequential applications by applying state-of-the-art techniques that aim for fine grained granularity of tasks.

Secondly, to optimize the execution of a parallel program, such as the result from the previous goal, by locally reducing the granularity of tasks, merging two or more together by executing them sequentially.
Thirdly, to address the issue of the lack of a solid Exception Handling Model that works in Implicit Parallel Languages.
While the previous chapter has introduced the main goals for this thesis, this chapter will introduce the work that has begun on reaching those goals. They will be addressed by the same order as they were presented.

### 4.1 Potential Parallelism in Sequential Applications

The first goal of the proposed work is to evaluate how much parallelism is there in sequential programs. This step is important to justify the effort spent on optimizing automatic concurrency techniques.

#### 4.1.1 Definition of Potential Degree of Parallelism

It is easy to measure parallelism in embarrassingly parallel programs. The number of independent parts in which the problem can be divided is usually considered as the degree of parallelism of that problem. Being independent means that they have no side-effects on each other and can be executed in parallel. For instance, in the trapezoidal integration, the maximum parallelism is equal to the resolution pretended. For \( N \) trapezoids of resolution, one can divide the work in \( N \) parts (or \( 2N \) if parallelizing the point function). For other divide-and-conquer problems, the maximum parallelism is equal to the number of parts possible to divide.

In more complex concurrent problems, specially those generated by automatic parallelization techniques, the parallelism is not symmetrical. Therefore there is no easy rule to measure the degree of parallelism as the maximum number of parallel units running at the same time changes during the execution of problem. Moreover, that number may
depend on the scheduling order of those units, making it hard to define the maximum
degree of parallelism statically.

Notice that this value is the theoretical maximum degree of parallelism and it is not
the most optimized value. This notion of degree of parallelism should be hardware
independent.

Previous work[32] has used three metrics to measure parallelism. These metrics can be
collected either at runtime, or during compilation if the scheduling is decided then. The
metrics are the following:

- **Parallelism Profile** - How many elements were executed in each round (iteration
  of the scheduler) of the program.

- **Parallelism Intensity** - The ratio between the amount of work in each round
  and the overall work available to be executed at that point.

- **Constrained parallelism** - An estimation of the critical path in the Task Graph
  if it would be executed on a limited number of processors. The critical path is
  the sequence of tasks that would limit the most the execution of other tasks, and
  therefore the subset of tasks that will limit the parallelism in a program.

Moreover, there is information from the execution of a program that is typically used to
evaluate parallelism:

- **Speedup of Parallel vs Sequential**. The speedup is a performance metric that
can be used with any two versions of a program. For this purpose, it is frequently
used the speedup comparing the parallel with a sequential version. The speedup is
the ratio between the execution times of the sequential and the parallel versions.
In these cases, the speedup represents how many instructions can be executed in
parallel on average for that program.

- **Occupation of processors** is another metric, more hardware independent than
  the speedup. The same program may have different speeds in machines with
  a different number of processing cores. However the average occupation can be
  calculated by dividing the speedup by the number of cores (which is the maximum
  theoretical speedup possible) and can be used to compare in machines with different
  number of cores. For instance, in a quad-core machine, the theoretical maximum
  speedup is 4 times. If a program has an actual speedup of 3 times, then the
  occupation is 75%. This occupation can then be compared with that of a dual-
  core machine, for instance.

Although less interesting for algorithmic studies, these values are very used in practice.
The speedup is used to measure the benefit of a parallel implementation and also to
4.1. POTENTIAL PARALLELISM IN SEQUENTIAL APPLICATIONS

compare several parallel implementations among themselves. The occupation of processors is used to evaluate whether the program is taking advantage of the available resources and whether that is the bottleneck (in occupations above 90%) or is it limited by synchronization. Additionally, the instant occupation is very useful for debugging as it allows the programmer to see where the program might be over-synchronizing.

4.1.2 Measuring the Potential Parallelism

Having defined metrics that can describe the potential parallelism, it is important to look at existing code to measure it. Despite having a large code-base of sequential programs available, the measurement itself is not easy.

In order to evaluate those programs, it is necessary to parallelize as much as possible each program. To achieve this objective, we will conduct the experiment depicted in Figure 4.1, starting with the original Java programs and ending with a log file containing statistics of the performance and parallelism of the program.

Figure 4.1: Architecture of the extraction of parallelism from sequential programs.

The first step is to automatically parallelize Java programs. For this goal, we will use both traditional benchmarks such as SPECjvm98[18] and Java Grande[9], as well as a selection of open-source applications and libraries popular in the industry.

The automatic parallelization is being done in the J2JPar compiler that is being completed at this stage. The compiler first analyses the data dependencies in the program by looking at variable initializations and usages. After having this information, the compiler follows a set of rules for each kind of Abstract Syntax Tree nodes.

In these rules, any operation can be extracted in its own task that has a dependency on the tasks that will have last modified the variables used in the current task. The
CHAPTER 4. CURRENT WORK

difference between this approach and the ones presented in the state of the art is that the goal is to have fine-grained parallelism, while other tools already optimize and group tasks together.

For instance, in the code `UserDB.register(getUser(), getPassword())`, the methods `getUser` and `getPassword` will be extracted into their own task and executed in parallel. The same applies for two unrelated statements, loops, etc...

The resulting Java code is then compiled and executed on the JVM. Because of familiarity and availability of a profiler, the resulting code will make use of the Æminium Runtime, although any scheduler that executes a Task Directly Acyclic Graph (DAG) would work. But to collect the kind of metrics related to parallelism, we need to introspect the runtime and periodically gather information about Tasks, Queues, GPUs and Memory. It is this information that will allow for a measurement of parallelism, either by looking at the statical properties of the graph, as well as metrics related to its performance.

The result will be a measurement of the parallelism in existing sequential programs. This information will be used to select programs to parallelize, as well as to have an understanding if we can optimize programs based on their parallelism.

4.1.3 Future Work

Having the J2JPar compiler finished, the next step is to run the benchmark suite through the compiler, and then measure the parallelism and take conclusions about it.

While this will answer the first Research Question, it is a very good starting point for the second one, in which the goal is optimize parallel programs, which we will have after using the compiler.

Based on the work currently being done, there are a few ideas on how to improve the performance of parallel programs. The granularity of tasks is one of such examples and one of the most important issues to make the resulting parallel programs from the compiler faster than their original versions. By analyzing the statistics gathered from the execution of parallel programs, other scheduling policies related to the task queue distribution can be inferred.
4.2 Balancing Software between Heterogeneous Platforms

4.2.1 Problem Definition

In the last years as GPUs became programmable, they have been used to complement the CPU in parallel intensive operations. Due to the very specific limitations and capabilities of GPUs, a new research field has emerged, focused on General Purpose Programing on GPUs (GPGPU).

Since then, GPUs have been an interesting target for parallelizing compilers. Recently, there have been efforts in trying to make use of the GPUs in high-level languages[13][11][26]. In order to do that, compilers must identify the parts that would be improved by the GPU, typically data-parallel cost-intensive operations such as Linear Algebra, for instance.

The main problem is that it is not possible to know before hand if a certain part of the program will execute faster on the GPU. GPUs have a different architecture and require that the data is copied from the main memory to the GPU special memory and the back. The organization of threads in groups that share the same program counter can also make programs with branch instructions much slower. As such not all programs are faster on the GPU and those should execute on the GPU.

Taking this decision is important because the wrong choice may add undesired overheads, which can last for several seconds. Furthermore, the programmer has to take into account several aspects of the program such as the memory accesses, branching instructions and program parallelism. The decision becomes harder because the actual performance also depends on the size of data used as input and on the available GPU’s specifications (number of cores, available memory, etc...). In most GPGPU libraries, this information is only available at runtime, which makes it nearly impossible for programmers to predict which platform will be always faster.

The goal of this work is to present a new framework which simplifies the task of writing data-parallel programs for transparently executing in GPUs, with improved performance. This work will be published in the coming month under the title “ÆminiumGPU: An Intelligent Framework for GPU Programming”[25].

In previous work[23], as well as in the state of the art, we have developed a parallel programming framework that was able to execute the same program on either GPUs or CPUs. Despite reducing the complexity of having to write GPU-specific code, programmers still have to decide in which platform they want to execute it.
4.2.2 Approach

The taken approach was to use Machine-Learning techniques to learn from existing programs what features could indicate the best platform, building a classifier. Each operation can be classified as either Best-on-GPU or Best-on-CPU. Since we are using a ÆminiumGPU [23], a framework that generates hybrid binaries that can execute on both platforms, the classification can be performed at runtime, immediately before choosing the platform.

Since decisions are hardware dependent (CPU and GPU combination), we considered two ways for tackling the problem: training the classifier in each machine; or considering CPU and GPU specifications as features in a general classifier. The former was selected for this work, although it can be extended to the later in the future. Using a large number of features would increase classification time and it would be a very hard to train a general classifier with a large set of CPU and GPUs. This means that when installing ÆminiumGPU, it is necessary to execute a standard benchmark for collecting training data.

The critical aspect for having a good classification is choosing the right features to represent programs. For instance, it is not feasible to consider the full program in ascii, since the length would be variable and the abstraction level ill-suited for classification techniques. Table 4.1 lists all the features used in the classification process.

Features can be extracted either during compilation or during runtime. This means that a given program will always hold the same values for the first features, while the last three features may be different, depending on the conditions of execution. Features marked with a size of 3 have three values, one for each depth of loop scopes. Listing 4.1 shows an example in which three functions are considered in 3 different loop levels. This distinction is important since operations in inner levels are executed more times than ones in the outer levels.

The choice of some selected features was inspired by other applications of Machine Learning in this area ([40], [12] and [48]). Memory accesses were considered a feature as they are one of the main reasons why GPU programs are not as fast as one would expect. As such, there are features for all three main kinds of memories in GPUs (global

<table>
<thead>
<tr>
<th>Listing 4.1: Examples of Level categorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(); // Level 1</td>
</tr>
<tr>
<td>for (int i=0; i&lt;10; i++) {</td>
</tr>
<tr>
<td>b(); // Level 2</td>
</tr>
<tr>
<td>while (j &lt; 20) {</td>
</tr>
<tr>
<td>c(); // Level 3</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
### 4.2. BALANCING SOFTWARE BETWEEN HETEROGENEOUS PLATFORMS

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Collected during</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OuterAccess</td>
<td>3</td>
<td>Compilation</td>
<td>Global GPU memory read.</td>
</tr>
<tr>
<td>InnerAccess</td>
<td>3</td>
<td>Compilation</td>
<td>Local (thread-group) memory read. This area of the memory is faster than the global one.</td>
</tr>
<tr>
<td>ConstantAccess</td>
<td>3</td>
<td>Compilation</td>
<td>Constant (read-only) memory read. This memory is faster on some GPU models.</td>
</tr>
<tr>
<td>OuterWrite</td>
<td>3</td>
<td>Compilation</td>
<td>Write in global memory.</td>
</tr>
<tr>
<td>InnerWrite</td>
<td>3</td>
<td>Compilation</td>
<td>Write in local memory, which is also faster than in global.</td>
</tr>
<tr>
<td>BasicOps</td>
<td>3</td>
<td>Compilation</td>
<td>Simplest and fastest instructions. Include arithmetic, logical and binary operators.</td>
</tr>
<tr>
<td>TrigFuns</td>
<td>3</td>
<td>Compilation</td>
<td>Trigonometric functions, including sin, cos, tan, asin, acos and atan.</td>
</tr>
<tr>
<td>PowFuns</td>
<td>3</td>
<td>Compilation</td>
<td>pow, log and sqrt functions</td>
</tr>
<tr>
<td>CmpFuns</td>
<td>3</td>
<td>Compilation</td>
<td>max and min functions</td>
</tr>
<tr>
<td>Branches</td>
<td>3</td>
<td>Compilation</td>
<td>Number of possible branching instructions such as for, if and whiles</td>
</tr>
<tr>
<td>DataTo</td>
<td>1</td>
<td>Runtime</td>
<td>Size of input data transferred to the GPU in bytes.</td>
</tr>
<tr>
<td>DataFrom</td>
<td>1</td>
<td>Runtime</td>
<td>Size of output data transferred from the GPU in bytes.</td>
</tr>
<tr>
<td>ProgType</td>
<td>1</td>
<td>Runtime</td>
<td>One of the following values: Map, Reduce, PartialReduce or MapReduce, which are the different types of operations supported by .EminiumGPU.</td>
</tr>
</tbody>
</table>

Table 4.1: List of features
CHAPTER 4. CURRENT WORK

In terms of operations, we performed micro-benchmarks to assess their execution cost. For instance, 4 or 5 plus operator calls execute much faster than one single sin call. As such, OpenCL functions were grouped according to the relative cost they have on execution time.

Besides these features, each benchmark also collected the execution time in both CPU and GPU, and the class to each execution belongs to. This is used for training and also evaluation.

4.2.3 Results

Dataset

Our workload for generating the training and testing dataset is composed by the following 8 programs:

1. A map operation that adds 1 to each element of the input array;
2. A map operation that applies the sin function to each element of the input array;
3. A map operation that applies the sin and cosine functions to each element of the input array and sums the values;
4. A map operation that calculates the factorial for each element of the input array;
5. A map-reduce operation that calculates the integral from 0 to the size of the array for \( f(x) = e^{\sin(x)} \);
6. A map-reduce operation that calculates the minimum value from 0 to the size of the array for \( f(x) = 10x^6 + x^5 + 2x^4 + 3x^3 + \frac{2}{5}x^2 + \pi x \);
7. A map-reduce operation that calculates the sum of all natural numbers up to a given value that are divisible by 7;
8. A map-reduce operation that calculates the sum of all elements of the input array that are divisible by 7.

Each one of these programs was executed several times with varying amounts of input data. The size of input data varies from 10 to \( 10^7 \) elements, executing with 10 values for each power of 10, and in each level multiplied by all natural numbers until 9. Thus, the first sizes would be 10, 20, 30, 40, 50, ..., and the last sizes would be 50^6, 60^6, 70^6, 80^6, 90^6, 10^7. Overall, the dataset has 440 instances of different program executions, from 8 individual programs, each executed with the 55 different data sizes.
4.2. BALANCING SOFTWARE BETWEEN HETEROGENEOUS PLATFORMS

Experimental Setup

These are the specifications of the hardware and software used for the experiments: Intel Core2 Duo E8200 at 2.66GHz; 4GB of RAM memory; NVIDIA GeForce GTX 285, with 240 CUDA cores and 1GB of memory; OS Ubuntu Linux 64bits with the NVIDIA CUDA SDK 5.0 preview 2 with OpenCL 1.1 and OpenJDK 1.7. The results presented here are specific to this particular hardware and software and can not represent all possible combinations.

Feature analysis

To evaluate features we used two feature ranking techniques: Information Gain and Gain Ratio. Both techniques were applied to the whole dataset. The ranking obtained was different for each method, but both returned 3 groups of features: A first group of high-ranked features, a group of low-ranked features and a third group of unused or unrepresentative features. This later group exists because the dataset programs do not cover all possibilities. But, this does not mean that such features should be ignored, on the contrary, they should be studied for particular examples which are out of the scope of this work. Table 4.2 shows the two other groups ranked using the Information Gain method.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Feature</th>
<th>Rank</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2606</td>
<td>DataTo</td>
<td>0.172</td>
<td>OuterAccess1</td>
</tr>
<tr>
<td>0.2517</td>
<td>DataFrom</td>
<td>0.0637</td>
<td>Branches1</td>
</tr>
<tr>
<td>0.1988</td>
<td>BasicOps2</td>
<td>0.0516</td>
<td>InnerAccess1</td>
</tr>
<tr>
<td>0.1978</td>
<td>BasicOps1</td>
<td>0.0425</td>
<td>TrigFuns1</td>
</tr>
<tr>
<td>0.1978</td>
<td>ProgType</td>
<td>0.0397</td>
<td>InnerWrite2</td>
</tr>
<tr>
<td>0.1978</td>
<td>OuterWrite1</td>
<td>0.0397</td>
<td>InnerAccess2</td>
</tr>
</tbody>
</table>

Table 4.2: Features rank using Information gain

Notice that features related with data sizes are high ranked, which is supported by the high penalty caused by memory transfers. Basic Operations are also very representative, since they are very common, specially in loop conditions (BasicOps2). The program type is also important because maps and reduces have a different internal structure. Maps happen in parallel, while parallel reduces are executed with much more synchronization in each reduction level.

Looking at the lower ranked features, it is important to consider that memory accesses also impact the decision. It is also expected that branching conditions would have an
impact on the performance of programs. Finally, trigonometric functions do not have such an high impact as basic operations, but they are still relevant for the decision.

Classifier Comparison

In order to achieve the best accuracy, it is important to choose an adequate classifier. For this task, several off-the-shelf classifiers from Weka[30] were evaluated, and some custom classifiers were also developed. The used classifiers include: a Random classifier that randomly assigns either class to a particular instance; AlwaysCPU and AlwaysGPU that classifies all instances as Best on CPU and Best on GPU; a Naive-Bayes Classifier; a Support Vector Machine (SVM) obtained from a Sequential Minimal Optimization algorithm with $c = 1$, $\epsilon = 10^{-12}$ and a Polynomial Kernel; a Multi-Layer Perceptron (MLP); a DecisionTable classifier; and a Cost-Sensitive version of the DecisionTable(CSDT) that uses 0.4 as the cost for misclassified Best on CPU programs and 0.6 for Best on GPU programs.

Besides these classifiers, we also experimented with a regression-based approach using additional metrics such as: CPUTime and GPUTime. The main idea was to use regression techniques to predict values of CPUTime and GPUTime for each instance and then select the smallest value. However, regressions have shown to have a poor quality with correlation coefficients between 70 and 80%. The final classifier behaved very similarly with the Random classifier. Thus, we decided to not pursue this line of research further.

Classifiers were evaluated using both 7 and 8 fold cross validation. Data was not randomized and was ordered by program. Since the number of folds is lower than or equal to the number of programs, some programs are not present in all the training sets. This simulates the real-world scenario of classifying programs that were not previously seen. The results with 7 and 8 folds were very similar, as well as the results with randomized data. The results presented from here on are with 7 folds and without randomization.

Figure 4.2 shows the accuracy distribution of the evaluated classifiers. AlwaysCPU and AlwaysGPU do not have 0.5 of accuracy because programs that are faster on the GPU are larger number on the dataset. This was not balanced on purpose, to reflect the actual distribution of CPU and GPU execution times for the tested programs. The DecisionTable classifier achieved a very high accuracy, only second to its Cost-Sensitive version which had a slightly higher accuracy with a more condensed distribution.

In this problem, the distinction between False Positives and Negatives is not relevant. This may seem to contradict the usage of a Cost-Sensitive Classifier, but the cost of misclassification does not only depend on the class, but also on the size of the data in that execution. In order to represent the impact of taking the wrong decision, a
measure of cost was introduced to replace the traditional confusion matrix. The cost of a misclassification is the absolute difference between the real GPU and GPU execution times previously measured during the feature extraction.

Figure 4.3 shows the distribution of the total cost of the classification for each cross-validation execution with a logarithmic scale on the Cost (yy axis). The lowest the cost is, the better. A perfect classifier would have a cost of 0. The random classifier has an average cost of $9.8 \times 10^9$, which can be considered as a ceiling for this dataset.

The measure of cost is important because we can see that some classifiers such as Naive-Bayes and SVM have a better accuracy but have an higher penalty on performance than the classifier that executes everything on the GPU. The two versions of the DecisionTable classifier were also the ones with the lowest cost. Another evaluation metric was classification time, since it could not be representative in execution time. Except for the NaiveBayes classifiers, all others classified instances in less than 20 microseconds, which is acceptable for this task. The classifier training time was not considered for this study as it is not relevant since it is only performed once per machine.

Looking at all the metrics, the Cost Sensitive version of the DecisionTable classifier was the best, achieving 92% of average accuracy and the lowest misclassification cost.
CHAPTER 4. CURRENT WORK

4.3 Verification of the Æminium Runtime

One of the crucial aspects of automatically parallelization is maintaining the semantics of the original program. In other words, the parallelization may not change the outcome of the program, it should be deterministic. The topic of verification is here introduced as a way of evaluating the correctness of the task scheduler, which serves as the basis for verifying the correctness of programs, an their parallelization, on top of the scheduler.

4.3.1 Context

Most parallel languages and frameworks rely on a runtime library that manages the distribution of work onto Operative System Threads. All the parallel and concurrent programs that execute on top of these runtimes expect it to be correct. If the runtime has any defect, that defect will influence the execution of all parallel programs. Therefore the correctness of the runtime is of extreme importance.

The Æminium language is one example of a programming language that depends on a runtime library to distribute the parallel tasks across threads. Other examples include Cilk, ForkJoin and Grand Central Dispatch (GCD). Æminium is a more interesting target because of its support for blocking tasks (shared only by GCD) and for concurrent
4.3. VERIFICATION OF THE ÆMINIUM RUNTIME

tasks.

The Æminium Runtime is a work-stealing scheduler that has a queue for each running thread. The number of threads is configurable, but defaults to the number of processors in the machine. Tasks are eagerly scheduled to the runtime, and held before entering the queues only after all the tasks they depend upon are completed. Tasks may be nested and the parent task only completes after all the children have completed.

Although in a stable version, it is not guaranteed that the runtime works according to expectations despite the several tests written. A better approach is to verify the software using Static Verification techniques. Since the Runtime is written in Java, the Java Modeling Language [34] (JML) will be used to describe the Runtime properties.

4.3.2 Properties

In JML properties are usually described in either assertions, contracts or invariants. Assertions were used in the design of the Runtime, but are mostly useful at runtime for debug purposes. Contracts are pre and post-conditions that have to be met, in order for a method to be correct. Invariants are conditions on fields that always hold true. The last two are specially interesting for static verification.

Having this in mind, we conducted a review of the Runtime source code and compiled a list of properties intended for the Runtime and that could be described in the source code. The list of possible properties includes:

- **All tasks should be executable.** This property can be verified using an invariant on \( \#\text{CreatedTasks} = \#\text{ScheduledTasks} \). In the Æminium Runtime tasks are first created and then scheduled. No task should be created and not scheduled. Additionally, tasks cannot be scheduled twice or more.

- **A Waiting-for-Dependencies task must have dependencies.** Tasks can have several states: Unscheduled, Scheduled, Waiting-for-Dependencies, Ready, Running, Waiting-for-Children and Completed. The state is changed in the presented order although some can be skipped. Tasks are scheduled with a set of other tasks that have to complete before the scheduled task executes. These other tasks are called dependencies. If a task has no dependencies, then it skips over the Waiting-for-Dependencies state and changes to Running. Therefore we can state the invariant \( \text{state} = \text{WaitingForDependencies} \Rightarrow \#\text{Dependencies} > 0 \).

- **A Ready, Running or Completed task must not have dependencies.** This property complements the previous one as all dependencies must have finished
before marking the task as ready for execution. The Runtime should follow the invariant \( \text{state} \in \{\text{Ready, Running, Completed}\} \Rightarrow \#\text{Dependencies} = 0 \).

- **A task must not have cyclic dependencies.** This property can already be verified at runtime, at an extraordinary performance penalty. Being able to verify this before executing the program can guarantee a deadlock-free performant program. This property can be verified using as precondition a function that already exists and recursively tries to find the current tasks in the dependencies of each visited task.

- **A Ready task should belong to one and only one queue.** After a task is marked for execution, it is added to one of the available queues. Each thread has its own queue from which it takes tasks to execute. If it finds its queue empty, it will “steal” work from other tasks. Therefore it is important to guarantee that a task is never present in two queues and that it never disappears from the queue if not for execution.

- **A thread can only steal from other queues if its own queue is empty.** Task stealing increases the probability of two threads trying to pop the same object, therefore should be avoided as much as possible.

- **A task must only be executed once.** If, by any chance, a task executes twice (maybe in two different threads) unexpected results might happen in the program. This property can be checked using a ghost variable (variable only used for verification) that is increased every time the `execute` method is called on the task. By the end of the execution, that ghost variable should be equal to 1.

- **Atomic Tasks must not create deadlocks.** Atomic tasks differ from other tasks only in the fact that they are associated with a lock. If two or more atomic tasks share a lock, only one can execute at a time, despite being unordered. Since Atomic tasks can spawn new atomic tasks with other locks, it is important to verify that no deadlocks can arise in any combination of tasks.

- **Tasks do not suffer from starvation.** When a task requires a lock, but it is not available at the time, the task is added to the end of the queue again. It is not easy to see if the complex system with queues, locks and work-stealing is starvation-proof. Using ghost variables to check if the `execute` method is always reached can help to check if the property holds.

- **All acquired locks should eventually be released.** When an atomic task executes it tries to acquire the lock (either successfully or not). If it does, it then executes and releases the lock. We have to guarantee that no thread holds the lock forever and must release it.
4.3. VERIFICATION OF THE ÆMINIUM RUNTIME

- Acquiring a lock should always happen before releasing a lock. This property complements the previous one by defining the order in which acquiring and releasing a lock must happen. For these two properties, a ghost variable can be used to describe them.

- The Runtime must only shutdown once all queues are empty. The shutdown operation happens on only one thread and it is necessary to guarantee that all the threads of the system terminate correctly without leaving any task to execute. This property can be checked with either an invariant or with post-conditions on the shutdown method.

Although some of the properties are trivial to state, they are in fact hard for a human to check just by looking at the code. Since state changes are done concurrently, it is hard to predict all the possible orders of execution. After describing the rules in JML, a static analyzer tool can automatically check if there is any case in which the properties are not verified.

4.3.3 Future Work

The next step is to write the previous properties in JML syntax and run the source code through a static analyzer. It is possible that the verification fails because some properties do not hold. In this case we will have to investigate and decide if it is a bug, or a misspecification. If it is the former, the bug will be addressed.

If all properties hold, than we have a more clear understanding of the correctness of the runtime and can state that the runtime is deadlock and starvation free as well as fair.

After having a verified runtime, it is important to verify the original programs. This can be done in two different ways. Firstly, we can produce properties from the data dependency found by the compiler. By verifying those properties, it is possible to guarantee that the optimizations and code generation phases of the compiler do not interfere with the dependencies of tasks.

A second approach, complementary to the first, is to manually state properties on the source programs and check if they are still valid on the final code generated by the compiler. This approach requires more manual effort to find properties in different programs that cannot be broken by the compiler.
4.4 Exception Handling in Implicit Concurrent Languages

During this work, we have published a paper entitled “Handling exceptions in programs with hidden concurrency” [24]. This paper addresses the issue of why existing Exception Handling (EH) mechanisms are not suitable for an Implicit Concurrent Language, such as Æminium.

The analysis is done by evaluating two different families of EH mechanisms. Despite being a solution for sequential systems, the tradition Try-Catch mechanism is the most popular in most languages and the most used by programmers. Even recent languages that lean towards parallelism still use this mechanism.

For Concurrent-by-Default languages like Æminium, they are no longer a good solution. In this kind of languages, the lexicographical order is not followed during execution. The usage of Try-Catch relies on knowing which instructions inside the protected Try block have executed when an error occurs. In implicit concurrent languages, this is not possible to know and will lead to inconsistent program states, which is not acceptable to recover from errors, or even log them.

Listing 4.2: Sequential Solutions for EH in Implicit Concurrent Program

```plaintext
try {
    writeLine(writer, newLine);
    Logger.log("Converted " + newLine);
}
catch (DiskIsFullException e) {
    clearTemporaryFiles();
    retry;
} catch (IOException e) {
    Logger.log("Could not write on the output file.");
    break;
}
```

Listing 4.2 shows an example where something is being written to file and logged at the same time inside a protected block. But since both instructions are independent and share no data, they can be executed in parallel. While there might an error writing to file, the logging may succeed, leaving the program in a inconsistent state, even after the tradition exception handling.

The second family of solutions is the Concurrent Exception Handling Mechanisms. Solutions like Atomic Actions, Conversations, Coordinated Atomic Actions, the Guardian model all share a common requirement: they require concurrency to be explicit in the source code. For instance they required threads to be created in the source code. In Implicit Concurrent Languages this does not happen, and sometimes the decision to create a thread or not is decided only in Runtime, not making it possible to express the Exception Handling in the language itself.
Furthermore, some of the solutions require the usage of Software Transactional Memory (STM) in order to guarantee stable states from which to recover. Currently, STM is very expensive in terms of performance. The evaluation of Atomic Boxes\cite{29} has showed slowdowns between 10 and 100 times. For implicit concurrent languages this is not feasible as the goal is to improve performance by the factor of the available number of cores. Using these systems will not be an improvement over sequential languages.

Concluding, neither existing Sequential nor Concurrent Exception Handling mechanisms are suitable for Implicit Concurrent Languages and for these languages to support Exception Handling — something fundamental in today’s programming environments — a new model must be created.

4.5 Summary

In this section we have presented the developed work on four lines of action. Firstly, we have started working on evaluating the implicit parallelism in existing sequential applications. We have defined some metrics and are finishing an automatic compiler for the Java language that parallelizes as much as possible, as well as a profiler to measure certain properties of the runtime.

Secondly, we are starting to work on finding the best granularity for programs. The developments in the profiler have the goal of helping to decide what kind of tasks will be merged.

Thirdly, we have developed a Machine-Learning System to automatically decide if a given part of a program will be executed on the GPU or CPU automatically.

Finally, we have identified a problem with the new types of Implicit Concurrent Languages: the existing EH mechanisms cannot be used in these languages since there is no known stable state and the concurrent is not explicit to use concurrent EH models.
5

Work Plan and Implications

5.1 Work Plan

![Gantt chart for the Thesis plan.](image)

The plan devised for this thesis extends for 3 years. The gantt diagram describes the coarse-grained tasks and when will they be executed. The major tasks are:

- **Evaluation of Potential Parallelism** — This is the current task and has been described in the first section of the previous Chapter.

- **Optimization of the Execution of Parallel Programs** — This task is the follow up work on the previous. The goal is to modify the program internal representation as well as the runtime to make parallel programs perform better. The first step is to find the best granularity, as it was previously described, for tasks in programs. In the future, this may include finding the best policies for queues and task scheduling during runtime.

- **Verification of the Determinism of Parallel Programs** — This task will guarantee that our techniques do not change the semantics of the original program. This task has already been started and the future work identified (as described in...
CHAPTER 5. WORK PLAN AND IMPLICATIONS

the previous Chapter).

- **Exception Handling in Implicit Concurrent Languages** — This problem has been identified in the current work. The solution has not been identified yet, and it is planned for future work, although with a lower priority than other tasks. The goal of this task is to find a suitable EH mechanism for Implicit Concurrent Languages.

- **Final Work Validation** — While frequent evaluations will be done, there will be a more thorough evaluation at the end. The goal of this task is to ensure that the improvements to the execution of parallel programs are actually valid. This task will be done by evaluating parallel programs in different machines, with different configurations and comparing several configurations with both the state of the art and out developed approach. The benchmark suite will also be larger than the ones used during the recurrent evaluations.

- **Thesis Writing**

5.2 Dissemination Strategy

The target conferences for publishing the work produced are listed in Table 5.1. Although not being restrictive, this list has the preferable targets since they are conferences that have a high reputation in this field.

As for journals, the two most important in the area are being considered: *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems*. 
### 5.2. DISSEMINATION STRATEGY

<table>
<thead>
<tr>
<th>Conference</th>
<th>Deadline</th>
<th>Takes Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPP</td>
<td>January</td>
<td>June</td>
</tr>
<tr>
<td>OOPSLA</td>
<td>March</td>
<td>October</td>
</tr>
<tr>
<td>HotDep</td>
<td>June</td>
<td>October</td>
</tr>
<tr>
<td>LCPC</td>
<td>May</td>
<td>September</td>
</tr>
<tr>
<td>Facing the Multicore Challenge</td>
<td>July</td>
<td>September</td>
</tr>
<tr>
<td>PPoPP</td>
<td>September</td>
<td>February</td>
</tr>
<tr>
<td>IPDPS</td>
<td>October</td>
<td>May</td>
</tr>
<tr>
<td>HotPar</td>
<td>October</td>
<td>May</td>
</tr>
<tr>
<td>PLDI</td>
<td>November</td>
<td>June</td>
</tr>
<tr>
<td>ECOOP</td>
<td>December</td>
<td>June</td>
</tr>
<tr>
<td>InPar</td>
<td>December</td>
<td>May</td>
</tr>
</tbody>
</table>

Table 5.1: Target Conferences
Parallel and Concurrent Programming are hot topics in both industry and research. As we have seen in Chapter 2, there are several different approaches on generating performant parallel programs. The outcome of those techniques is not only measured in performance, but in programming usability and program correctness.

In this thesis, we will first evaluate the potential of further exploring parallelization techniques. As the clock speed of processor stagnates and manufacturers choose to bet on multicores, parallelization techniques are of most importance to improve the performance of programs. The first step of this thesis is to evaluate if existing programs can automatically be adapted to take advantage of this new architectures.

Then the focus will be on improving the performance of parallel programs. This is not an easy task given the complexity in hardware multiprocessors (even more with heterogeneous computing), in programs and in scheduling alternatives. However its gains will help to improve existing and new programs in terms of performance. A first step was already taken in this phase by proposal of a Machine-Learning classification technique to identify is a program should run on GPU or CPUs. Further work will be done on expanding this approach to several parallelization solutions in multicore CPUs. Another goal is to find the best granularity of tasks that will make the execution of programs faster. This is known to be a current issue and an area where improvement can be done.

Concluding, by the end of this PHD it is expected to have a set of new techniques and methods that can generate more performant programs and an evaluation of those techniques.


