# Parallelisation of desktop environments

by Nasser Giacaman

Supervised by Dr Oliver Sinnen

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering, The University of Auckland, 2010.

## Abstract

The objective of this research was to investigate the parallelisation of desktop environments in light of multi-core processors becoming mainstream. In contrast to the parallelisation of typical high performance computer applications, parallel computing for desktop applications involves further challenges. Although improving performance is the primary aspect of parallel computing, a vital focus of this research was on the software engineering approach. In particular, this included developing an object-oriented solution to desktop parallelisation.

To address these issues, two concepts were developed in this research. The first concept, the Parallel Iterator, is an object-oriented solution for data parallelism. The Parallel Iterator is the parallel extension to the standard sequential iterator, to support parallel traversal of elements in an arbitrary collection. The second concept, Parallel Task, is an object-oriented solution for task parallelism. An important design aspect of Parallel Task was its semantic integration with the structure of typical desktop applications.

Both of these concepts have been successfully implemented. The solutions provide ease of use by following familiar programming approaches and encapsulating parallelisation details away from programmers. The results for both the Parallel Iterator and Parallel Task concepts show superior performance compared to standard parallelisation approaches.

This research has made a vital contribution to parallel computing on mainstream desktop systems. By identifying challenges specific to the parallelisation of desktop applications and their current structure, the concepts developed are in line with object-oriented programming and the software engineering approach. The concepts and tools developed not only ease the programming effort, but also enhance the user's desktop experience by promoting responsive and interactive applications.

# Dedication

**To mama and baba,**

**So many things would not be possible without the sacrifices you have made and the love and support you have continuously given me all my life, this being one of those things. With my deepest love, this thesis is for you.**

# Acknowledgments

As this section shows, I am indebted to many people... and so many more. But it is impossible to thank everyone, so I must apologise in cutting it short.

First, I would like to show my gratitude to those closest to me: mama, baba, and my dearest brothers and sisters. It is your continuous support and encouragement in so many aspects of my life that has made this possible. Especially in the last four years, thank you for putting up with all the times my stress levels peaked and excusing me from many chores. My brothers in particular, you have always taught me since school to be strong and pursue anything my heart feels right. To my nieces and nephews, thank you for your patience. You make me proud to be your uncle. As promised, the laptop is all yours now... and yes, I will reinstall the car game.

To my supervisor, Dr Oliver Sinnen, it is an honor being your first PhD student! I have so many reasons to thank you, that if you got to proof-read this section then you would end up suggesting a separate subsection! ;-) In times when doubt lingered in my head, a few words of encouragement were enough to get me rolling again. You always inspired me to pursue so many opportunities: from taking a few extra weeks to sightsee while attending conferences, to experiencing an overseas study exchange, these experiences have been part of a dream come true for me. Thank you for your guidance, my confidence in presentations and especially in report writing have magnified. Coming into the research, I dreaded any writing. But halfway through, I found myself documenting things on my own because I actually *enjoyed* (!!!) it. These skills will forever help me.

I have met so many fellow postgrads that became great friends. Since day 1, the kindness of Ravikesh Chandra and Claudio Camasca Ramirez impacted me. Ravikesh, I will never forget my first week when you pulled me a chair beside your desk and walked me through example database searches for my research topic. Claudio, you have always been a great friend, the ice cream breaks and "walk and talks" around campus were awesome. The best example of generosity is Wei-Tsun Sun. In addition to your friendship, I am so grateful for your numerous Linux lessons and continuous hacks getting my system to work. To the other guys in room 235, it was fun - but we still need a new poster!

To my dearest friends from the Auckland SK clubs, you have made the last four years so much full of life and I look forward to many more to come! Marek Koliandr, you've been a great friend since day 1 of high school. It's amazing how fast 13 years will pass by! Michael Kerr, thanks buddy... you have been there for me and it has been fun. Elias Billeh, in my undergrad years I learnt a lot from you. You continued supporting me, and I enjoy our catch-up lunches. Ionuţ Petrache, you made my study exchange experience very enjoyable.

A special thank you to the department office ladies Aruna Shandil, Priyanka Shandil, Daisy Mitchell and Kim Francis for putting up with my petty annoyances. Similarly, the IT department, in particular Jamie Walker thank you for numerous help with Linux issues. Thank you also to Associate Professor Ewan Tempero, always being so welcoming and refereeing for various funding applications. I owe deep gratitude to Associate Professor Low Yoke Hean Malcolm for hosting me at NTU for 3 months. Thank you for your kindness, this experience allowed me to progress both at the research and personal level at a critical time. To Associate Professor Bruce MacDonald and Dr Morteza Biglari-Abhari, thank you both for your interest in my research, especially in the initial phase and providing feedback for my first conference paper.

I would also like to acknowledge the support of the various awards that supported me during my PhD study. To the University of Auckland, thank you for the Doctoral Scholarship, the Graduate Research Fund and also the Faculty of Engineering PhD Award. Along the way, I was extremely fortunate to receive support from the BuildIT Travel Fund and also the New Zealand Postgraduate Study Abroad Award, each of which funded two overseas experiences. I am also most grateful for the financial support during my undergraduate years from the New Zealand Computer Society (Tertiary Scholarship Award) and the Catholic Diocese of Auckland (Robert Ned Covich Scholarship), without which I would have struggled reaching postgraduate level.

# Contents

**1 Introduction** **1**

**2 Background** **6**

2.1 Inhibitors for desktop parallelisation . . . 6

2.1.1 General parallel computing challenges . . . 6

2.1.2 Additional challenges for desktop parallelisation . . . 10

2.2 Data parallelism . . . 15

2.2.1 The Java-style sequential iterator . . . 15

2.2.2 Traditional parallelism approaches . . . 16

2.3 Task parallelism . . . 19

2.3.1 Graphical user interfaces . . . 20

2.3.2 Where are the multi-threaded toolkits? . . . 21

2.3.3 Multi-threaded GUI applications . . . 22

2.3.4 The threading model versus the tasking model . . . 23

2.3.5 Typical GUI parallelisation approaches . . . 25

**3 Parallel Iterator concept and related work** **29**

3.1 Parallel Iterator concept . . . 29

3.1.1 Interface and usage . . . 29

3.1.2 Construction . . . 32

3.1.3 Semantics . . . 34

3.1.3.1 Scheduling policies and chunk size . . . 35

3.1.3.2 Supported collections . . . 37

3.1.3.3 Modifying elements versus modifying the collection . . . 37

3.1.3.4 Parallel `remove()` semantics . . . 38

3.1.4 Reductions . . . 41

3.1.4.1 Reductions with the `Reducible` object . . . . . 43
3.1.4.2 User-defined reductions . . . . . . . . . . . . . . 44
3.1.4.3 Alternative interface . . . . . . . . . . . . . . . . 45
3.1.5 Break semantics . . . . . . . . . . . . . . . . . . . . 48
3.1.5.1 Global `break` . . . . . . . . . . . . . . . . . . . 48
3.1.5.2 Local `break` . . . . . . . . . . . . . . . . . . . 49
3.1.6 Exception handling . . . . . . . . . . . . . . . . . . . 50
3.1.7 Iterating over multiple collections . . . . . . . . . . . . . 54
3.1.8 Tree Parallel Iterator . . . . . . . . . . . . . . . . . . 56
3.2 Related work . . . . . . . . . . . . . . . . . . . . . . . . . 64

**4 Parallel Iterator implementation and performance 68**
4.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . 68
4.1.1 Scheduling policies . . . . . . . . . . . . . . . . . . . 69
4.1.1.1 Random access collections . . . . . . . . . . . . 69
4.1.1.2 Inherently sequential collections: on-demand copying . . . . . . . . . . . . . . . . . . . . . . . . 70
4.1.2 Alternative interface implementation . . . . . . . . . . . 74
4.1.3 Reductions . . . . . . . . . . . . . . . . . . . . . . . 75
4.1.4 Parallel remove . . . . . . . . . . . . . . . . . . . . . 76
4.1.5 Parallel break . . . . . . . . . . . . . . . . . . . . . . 77
4.1.6 Exceptions . . . . . . . . . . . . . . . . . . . . . . . 77
4.1.7 Tree Parallel Iterator . . . . . . . . . . . . . . . . . . . 78
4.2 Performance . . . . . . . . . . . . . . . . . . . . . . . . . 81
4.2.1 Comparing to traditional Java parallelism approaches . . 82
4.2.2 Scheduling policies . . . . . . . . . . . . . . . . . . . 84
4.2.3 Disk-intensive applications . . . . . . . . . . . . . . . . 86
4.2.3.1 Image resizing . . . . . . . . . . . . . . . . . . . 86
4.2.3.2 Word count and permutation (reductions) . . . . 88
4.2.4 Tree Parallel Iterator . . . . . . . . . . . . . . . . . . . 91

**5 Parallel Task concept and related work 94**
5.1 Parallel Task overview . . . . . . . . . . . . . . . . . . . . 94
5.1.1 Model overview . . . . . . . . . . . . . . . . . . . . . 94
5.1.2 Syntax and semantics . . . . . . . . . . . . . . . . . . 96
5.1.2.1 Semantics of a task (as opposed to a method) . 97
5.1.2.2 Private and shared variables . . . . . . . . . . . 97

5.1.2.3 Different task types . . . . . . . . . . . . . . . . 98
5.1.2.4 Nested parallelism . . . . . . . . . . . . . . . . . 102
5.1.2.5 Canceling a task . . . . . . . . . . . . . . . . . . 103
5.1.2.6 ParaTask clauses . . . . . . . . . . . . . . . . . . 104
5.2 Dependences . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
5.3 Task completion . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.3.1 Blocking on a TaskID . . . . . . . . . . . . . . . . . . . . 106
5.3.2 Blocking on non-ParaTask data structures . . . . . . . . . 109
5.3.3 Synchronisation inside multi-tasks . . . . . . . . . . . . . 111
5.3.4 Non-blocking: The `notify` clause . . . . . . . . . . . . . . 113
5.3.5 Interim progress and notifications . . . . . . . . . . . . . 119
5.4 Exception handling . . . . . . . . . . . . . . . . . . . . . . . . 121
5.4.1 Asynchronous exception handling . . . . . . . . . . . . . . 122
5.4.2 Exception handling while blocked on `TaskID` . . . . . . . 129
5.4.3 Dependences and notifications . . . . . . . . . . . . . . . 129
5.4.4 Multi-task exceptions . . . . . . . . . . . . . . . . . . . . 131
5.5 Grouping tasks . . . . . . . . . . . . . . . . . . . . . . . . . . 132
5.6 Scheduling schemes . . . . . . . . . . . . . . . . . . . . . . . . 133
5.6.1 Work-sharing schedule . . . . . . . . . . . . . . . . . . . . 133
5.6.2 Work-stealing schedule . . . . . . . . . . . . . . . . . . . . 135
5.6.3 Mixed schedule . . . . . . . . . . . . . . . . . . . . . . . . 136
5.7 Adherence to object-oriented programming . . . . . . . . . . . 137
5.7.1 Encapsulation . . . . . . . . . . . . . . . . . . . . . . . . 137
5.7.2 Inheritance . . . . . . . . . . . . . . . . . . . . . . . . . . 138
5.7.3 Polymorphism . . . . . . . . . . . . . . . . . . . . . . . . 138
5.7.4 Inheritance anomaly . . . . . . . . . . . . . . . . . . . . . 138
5.8 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

**6 Parallel Task implementation and performance 148**
6.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 148
6.1.1 ParaTask source-to-source compiler . . . . . . . . . . . . . 149
6.1.1.1 Parsing task declarations . . . . . . . . . . . . . 152
6.1.1.2 Parsing task invocations . . . . . . . . . . . . . . 154
6.1.2 ParaTask runtime system . . . . . . . . . . . . . . . . . . 158
6.1.2.1 Work-sharing schedule . . . . . . . . . . . . . . . 160
6.1.2.2 Work-stealing schedule . . . . . . . . . . . . . . 161
6.1.2.3 Mixed schedule . . . . . . . . . . . . . . . . . . . 163

6.2 Performance . . . 163
6.2.1 Comparing to traditional Java parallelism approaches . . 164
6.2.1.1 Compute-intensive applications . . . 165
6.2.1.2 Disk-intensive applications . . . 167
6.2.1.3 Recursive applications . . . 167
6.2.2 User-perceived performance . . . 170
6.2.3 Overhead . . . 171
6.3 Example application: ParaImage . . . 172
6.3.1 ParaImage features . . . 174
6.3.2 ParaImage implementation . . . 180
6.4 Combining ParaTask & the Parallel Iterator . . . 184

**7 Conclusions** **186**

**A ParaTask grammar** **190**
A.1 Tokens . . . 190
A.2 Task declarations . . . 191
A.3 Task invocations . . . 192

**Bibliography** **195**

# List of Figures

1.1 Example of four tasks with dependences . . . . . . . . . . . . . . 3

2.1 Structure of a single-threaded GUI application . . . . . . . . . . 20
2.2 Structure of a multi-threaded GUI application . . . . . . . . . . . 22

3.1 Visualisation of the Parallel Iterator usage . . . . . . . . . . . . . 33
3.2 The major scheduling policies supported by the Parallel Iterator 36
3.3 Example of static cyclic scheduling . . . . . . . . . . . . . . . . . 38
3.4 Example of a simple reduction . . . . . . . . . . . . . . . . . . . . 42
3.5 Visualisation of the `Reducible` object . . . . . . . . . . . . . . . 47
3.6 Tree Parallel Iterator class diagram . . . . . . . . . . . . . . . . . 58
3.7 Example of an SVG file . . . . . . . . . . . . . . . . . . . . . . . . 58
3.8 Partial ordering with the Tree Parallel Iterator . . . . . . . . . . 59

4.1 Parallel Iterator implementation: random access collections . . . 71
4.2 Parallel Iterator implementation: inherently sequential collections 73
4.3 Parallel Iterator implementation: on-demand copy . . . . . . . . 74
4.4 Example run of work-stealing with the Tree Parallel Iterator . . . 79
4.5 Parallel Iterator performance: traditional Java approaches . . . . 83
4.6 Parallel Iterator performance: Mandelbrot . . . . . . . . . . . . . 85
4.7 Parallel Iterator performance: image resizing . . . . . . . . . . . 89
4.8 Parallel Iterator performance: word count . . . . . . . . . . . . . 90
4.9 Parallel Iterator performance: word permutation . . . . . . . . . 90
4.10 Tree Parallel Iterator performance: SVG application . . . . . . . 91
4.11 Tree Parallel Iterator performance: Newton-Raphson . . . . . . . 92

5.1 Example of a deadlocked recursive task . . . . . . . . . . . . . . 107
5.2 Example of a recursive task with avoided deadlock . . . . . . . . 109

5.3 Thread movement with ParaTask's `notify` clause . . . . . . . . . 114
5.4 Semantic Semantic differences of `dependsOn` versus `notify` . . . 117
5.5 Class diagram and categories of Java exceptions . . . . . . . . . . 122
5.6 Call stacks of enqueuing threads versus worker threads . . . . . . 125
5.7 Example of nested task parallelism . . . . . . . . . . . . . . . . . 135

6.1 Steps for a task, from creation to execution . . . . . . . . . . . . 149
6.2 ParaTask development process . . . . . . . . . . . . . . . . . . . . 150
6.3 ParaTask runtime implementation . . . . . . . . . . . . . . . . . . 159
6.4 ParaTask implementation: work-stealing scheduling . . . . . . . . 162
6.5 ParaTask implementation: mixed scheduling . . . . . . . . . . . . 163
6.6 ParaTask performance: traditional Java approaches . . . . . . . . 166
6.7 ParaTask performance: applications with high disk access . . . . 168
6.8 ParaTask performance: compared to JCilk . . . . . . . . . . . . . 169
6.9 ParaTask performance: expected versus actual finishing order . . 172
6.10 ParaTask performance: compared to sequential methods & threads173
6.11 ParaImage Flickr search project . . . . . . . . . . . . . . . . . . . 175
6.12 ParaImage Flickr search project: parallel mode . . . . . . . . . . 176
6.13 ParaImage image editing project . . . . . . . . . . . . . . . . . . 177
6.14 ParaImage image editing project: parallel mode . . . . . . . . . . 179
6.15 ParaImage image editing project: example filters . . . . . . . . . 180

# Chapter 1

# Introduction

Despite the performance benefits, parallel computing has traditionally been, and still remains, notoriously difficult for software developers [38, 84]. In addition to the usual challenges of developing a sequential program, parallel computing presents a new range of complications. First come the theoretical challenges of task decomposition, dependence analysis and task scheduling. Then there are the practical challenges such as synchronisation, debugging and portability. The situation is even worse for parallel desktop applications since they are generally irregular with short run-times. They also run on non-dedicated systems, let alone knowing what the system specifications are in the first place (e.g. number of processors, amount of memory and so on).

Since the objective behind parallel computing is to reduce the execution time of a program, parallelising code has traditionally been paired with general code optimisations for performance (especially in the scientific and engineering area). It is therefore no surprise that applications for these domains have been (and are still) written in low-level, but speed-efficient, languages like Fortran or C [25]. When it comes to desktop applications and object-oriented languages, however, one wants to improve the performance through parallelisation, but without sacrificing the benefits of high-level languages and the software engineering approach to programming. Hence the challenge is to parallelise object-oriented code without resorting to low-level approaches, defying the purpose of code abstraction, encapsulation and so on. This thesis addresses this challenge. The primary motivation is to relieve programmers from many of the effortful and tedious aspects demanded from parallel computing (section 2.1), but of course without neglecting the end performance.

In proposing new tools to parallelise desktop applications, it is vital to understand their structure. Consequently, this thesis focuses on object-oriented languages due to their popularity [102], especially for desktop application development. For example, common languages for Windows programming include C++, C# and Java. The K Desktop Environment (KDE) for Linux is developed in C++ using Trolltech's Qt toolkit [103] and Mac OS X is developed using Objective-C. However, focusing on the parallelisation of *object-oriented applications* is typically not enough: one must also look deeper at the *structure of desktop applications* (section 2.3.3).

Parallel computing has arrived at mainstream desktop systems in the form of multi-core processors because of the difficulties maintaining improvements in uni-processor clock-speed. Even though parallel computing is decades old, desktop parallelisation is fairly new. Users will not witness any performance improvements if desktop applications are not parallelised with performance in mind [11, 99].

This thesis addresses two major programming idioms: iterative computations and dataflow-style computations. Iterative computations usually carry the lion's share of computational load, and in object-oriented languages this is often implemented with *iterators*:

```
Collection items = ...
Iterator<Object> it = items.iterator();
while (it.hasNext()) {
  process(it.next());
}
```

For the second class of problems, consider the example application of figure 1.1. The programmer has identified 4 independent computations to be executed in response to the user pressing a button. This idiom of responding to user actions through a visual interface is typical of desktop applications. Such a visual interface is called a graphical user interface (GUI) and will be further discussed in section 2.3. In a sequential program, the following code is written:

```
public void actionPerformed(ActionEvent e) {
  File file1 = compute1(‘‘myimage.jpg’’);
  File file2 = compute2(‘‘myimage1.jpg’’);
  File file3 = compute3(‘‘myimage1.jpg’’);
```

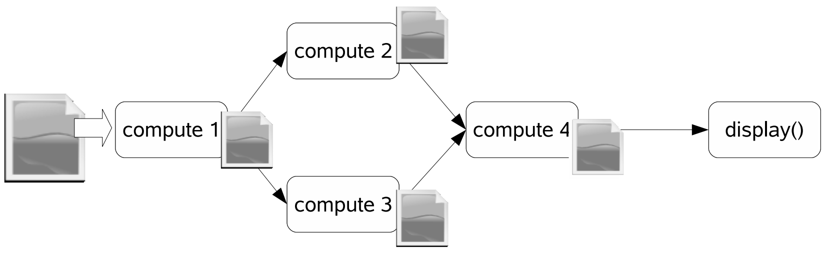


Figure 1.1: Four tasks with dependences amongst them. Only the second and third computations may execute in parallel, since they must wait for the first computation to complete. The `display()` method performs GUI-related computations.

```
        File file4 = compute4(‘‘myimage2.jpg’’, ‘‘myimage3.jpg’’);
        display(‘‘myimage4.jpg’’);
    }
```

As will be shown in section 2.3, the `actionPerformed()` is an event handler whose computation should complete with minimal time. Even though parallelisation may help speed the 4 computations, this still might be insufficient for a responsive event handler. In fact, the `actionPerformed()` (and any other event handler) should appear instantaneous because it is executed by an event handling thread that must be free to respond to other events. Yet only this very same thread may call `display()` when the 4 computations are completed: this is required by most graphical user interface (GUI) toolkits, as will be discussed in section 2.3.3. In event-based applications, different threads have different roles.

### Graphics Processing Units (GPUs)

Parallelism in a desktop system may be realised by the multi-core processor or a graphics processing unit (GPU). The GPU typically has the specialised role of accelerating graphics rendering by offloading the computation away from the CPU. In fact, the computational power of today’s GPUs easily outperforms the most powerful CPU [74, 101]. In order to execute on the GPU, programmers are required to write low-level code using libraries that are different from the standard object-oriented languages and libraries that are used for the CPU; for example, CUDA (Compute Unified Device Architecture) is a C-based library for NVIDIA GPUs. General purpose computing on graphics processing units

(GPGPU) extends the applicability of GPUs and is the process of using the GPU to execute code typically executed by the CPU.

Even with frameworks such as CUDA, the programmability of GPUs for general purpose computing is still difficult when attempting to achieve good performance [60]. For this reason, this thesis does not take GPUs into consideration for general purpose computing. As discussed earlier, the focus here is on object-oriented languages and the software engineering process. Achieving good performance on GPUs for general purpose computing requires programmers have a good understanding of the underlying hardware (unlike simple GPU programming, where only some of the graphics pipeline was accessible). This in particular violates important software engineering concepts such as encapsulation and high-level abstraction.

Although this thesis addresses the problem of data parallelism (namely the Parallel Iterator of sections 3 and 4), GPUs cannot be used for many of the collections that are supported by the Parallel Iterator (for example graphs and trees, including XML documents). The Parallel Iterator's primary purpose is to iterate a collection in an object-oriented approach with minimal change to the sequential approach and abstracting away details of the underlying collection; such an approach is not possible using GPUs for complex data collections such as trees.

## Contributions

This thesis presents two parallelisation concepts targeting object-oriented languages: the *Parallel Iterator* and *Parallel Task* (short ParaTask). The Parallel Iterator targets the parallelisation of iterative computations, using an approach familiar to programmers who have used standard sequential iterators. For applications that require the parallelisation of computation (applications with a dataflow-style structure), ParaTask is presented; it has the unique feature of including support for the parallelisation of GUI applications.

**Parallel Iterator** The Parallel Iterator (chapter 3) is a new powerful concept for object-oriented programming that serves to iterate any collection type in a thread-safe manner, even those inherently sequential (e.g. linked-lists and trees). The typical iteration code of the program remains the same, meaning that the structure of the program is not changed. The Parallel Iterator has also been implemented with the exact interface of the familiar Java-style sequential iterator. When the Parallel Iterator is used in combination with OpenMP [82] (which is supported by many compilers, including Visual C++, Intel and GCC), the loop code does not require restructuring even when reductions are necessary. The concepts proposed are applicable to most object-oriented languages, this thesis presents the C++(Qt) and Java implementations (chapter 4).

Several scheduling schemes are supported: static, dynamic and guided (covering all major loop-scheduling schemes). In addition, the scheduling policy and chunk size may be decided dynamically during run-time for each loop. The concept of *reductions* is integrated with the Parallel Iterator concept, providing a true object-oriented solution for reductions generalised to allow user-defined reductions on any data type. Global semantics for the `break` statement are also presented to achieve early loop termination, as well as providing helpful means to object-oriented exception handling in a parallel traversal. Performance of the Parallel Iterator (chapter 4) reveals that the overhead is small and justifiable for object-oriented code using iterators in the first place. Finally the Parallel Iterator may be used with any threading environment and very elegantly with OpenMP.

**Parallel Task** In light of multi-cores becoming mainstream on typical desktop systems, the vision is not only to introduce task parallelism but also to create a simple and intuitive approach to event-based applications in a parallel environment. Parallel Task (chapter 5) allows a wide range of computation problems to be catered for by integrating different task types into the same model (one-off tasks, interactive tasks and multi-tasks) and respecting object-oriented concepts. Further features are presented, including support for intuitive dependence handling, non-blocking notification, exception handling and reduction for multi-tasks. Parallel Task also supports flexible scheduling options: work-stealing (for nested parallelism), work-sharing and a mixture of both. The implementation and performance compared to a range of typical parallelisation approaches are also presented (chapter 6).

# Chapter 2

# Background

Since this thesis presents parallelisation concepts targeting object-oriented languages, different forms of parallelism in the context of desktop applications are explored. But firstly the challenges of parallel computing are discussed, focussing on the additional challenges particular for desktop applications.

## 2.1 Inhibitors for desktop parallelisation

What are the challenges that programmers face when parallelising desktop applications? The programmer is first faced with the general challenges of parallel computing (section 2.1.1). Unfortunately, parallel computing is further complicated for desktop applications (section 2.1.2).

### 2.1.1 General parallel computing challenges

The prominent challenge in parallel computing is the theoretical and practical difficulty of producing an effective program to exploit the available hardware. The developer will need to deal with many more issues that do not exist in sequential programming.

#### Theoretical challenges

Some of the challenges surrounding parallel computing are inherent in the problem itself, regardless of any practical solutions that may be implemented.

***Missing central parallel model*** Much of the success of the uniprocessor computer industry is to be credited to the von Neumann model that connects the hardware and software of sequential programming. This allows software to be developed independent of hardware, and hardware to be developed independent of software. Even with this separation, the software and hardware complement each other and the result consistently executes with satisfactory performance. Parallel computing is missing such a central unifying model that the sequential computing has been so successful on [67, 100, 106].

Numerous models for parallel computing have been developed and proposed, but parallel computing is still lacking a single dominant model. The characteristics of a successful model include simplicity to understand and use, architecture independence and accurate performance prediction [91]. These characteristics are contradictory and as a result have resulted in a large number of parallel computing models being developed. Highly abstract models make it easy to develop portable parallel programs, but such programs will result in highly uncertain performance. On the contrary, highly detailed models allow optimised programs but are considerably more difficult and less portable due to the explicit details.

The PRAM model for example is very simple to use, but at the expense of hiding vital performance influencing factors such interprocessor communication and synchronisation [53]. Other models such as LogP and BSP have gained some attention [12], but parallel computing is still waiting for the one dominant model to rise above all and standardise parallel computing.

***Task decomposition*** In order to increase the throughput of an application, the developer needs to examine and decompose the original sequential problem into sub-tasks. Identification of such sub-tasks will determine which areas of the problem may be executed simultaneously. Whether the developer is modifying an already existing sequential program, or coding parallel code from scratch, task decomposition is the first thing to perform.

A developer may also decide to restructure the program in a way to increase the inherent parallelism of the problem, such as seeking to use alternative algorithms. This is based on the idea that some conceptual models may be parallelised with greater ease than other models. Therefore, seeing the problem in a different light may ease the difficulty and consequently make the inherent parallelism more noticeable.

***Dependence analysis*** Even when the problem has been decomposed into independent tasks, this does not mean the tasks are free to execute in any random order. There are ordering constraints that govern the way tasks ought to proceed. Such constraints prohibit tasks from simply operating independently of each other, as they may frequently need to halt due to the state of other tasks [10]. Dependences must be respected in order to ensure that the meaning of the program remains correct.

Dependences take on many forms and primarily fall into the categories of either data or control dependences. A data dependency is when two or more active tasks both require access to the same data, and the order in which the data is modified influences the correctness of the program. A control dependence is when the execution of some part of the program is dependent on the outcome of another part of the program.

Flow dependency is a type of data dependency where the result of a task is required as an input to a future task. The consequence is that a lot of code is forced to execute sequentially in the program, consequently reducing the inherent parallelism of the problem. The consequence of control dependences means that code cannot be explicitly parallelised since the execution flow is uncertain until runtime.

After attempting task decomposition and dependency analysis, the developer may conclude that the problem is not a suitable candidate for parallelisation. The classification of a problem's inherent parallelism is rarely black or white. A single problem may have a wide range of inherent parallelism, simply depending on the specific goal or instance of the problem class [96]. For example, the problem of installing light bulbs is not inherently parallel if the goal is to install a single bulb. But the problem is inherently parallel if the goal is to install ten bulbs.

Some problems simply cannot be parallelised due to the inherent requirement of consecutive instructions that need to be executed one after the other [27]. Even if the developer attempted to force some level of parallelism into the implementation of such a problem, the performance improvement is likely to be quite minimal. It may actually be faster to execute the problem sequentially due to the fact that there exists a lot of dependences between the tasks.

***Scheduling*** It is well known that determining the optimal schedule for a set of tasks is an NP-hard problem [90]. Scheduling becomes more difficult when the tasks to schedule have varying priorities, or when the main objective of the scheduling changes. The user may have important tasks, and the response of those tasks is of more interest rather than the overall execution time of all the tasks in the system. In some cases, static scheduling is impossible since the tasks are unknown up front and as a result dynamic scheduling needs to be used.

It generally makes sense that load balancing is important in order to create tasks that take roughly the same amount of time to execute, otherwise some processors will simply be idle while others have a large amount of computation to complete [46]. However, it is interesting to note that achieving a well balanced load may not necessarily always result in the best performance. If a large amount of dependences exist between a set of tasks, the high communication costs between processors could mean that having select processors do more work will result in a lower execution time.

### Practical challenges

In addition to the inherent theoretical challenges of parallelisation, other issues commonly tend to arise due to the way that things have traditionally been done. Much of the practical challenges are a result of the underlying theoretical challenges.

***Portability*** When parallel programs are written, developers will always want to ensure that they effectively exploit the available hardware to achieve maximum performance. In order to optimise the parallel program, it is necessary to understand the specific hardware details for the target system. Many parallel programming solutions are therefore architecture and machine dependent [100]. Simply porting the program to another machine would not provide the same performance and the program would require modification, or in the worst case be rewritten from scratch.

Parallel programming using the message passing model usually results in higher performance in comparison to the shared memory model [55]. The downside of such an approach is that it involves explicit parallelism where the developer needs to carefully examine the problem and implement the parallelism. Due to the weak support available for parallel programming, developers have largely produced specific and custom tools for their particular purpose [67]. Along with the issue of portability, this creates a wide mass of non standardised tools and programs.

***Synchronisation*** The need for synchronisation arises when data locations are reused in the program, and so this issue is of particular interest to the shared memory paradigm. Synchronisation further complicates the challenge of scheduling, since now there is timing constraints between some tasks that must be adhered to throughout the entire system execution. It is important to respect the synchronisation requirements of a parallel program if it is to make satisfactory progress [46]. If the multiple threads (of a parallel program) were scheduled independently, they may end up running sequentially (on the same processor): such a scheduling would prove counterproductive for the parallel application due to the overhead introduced in the parallelisation. A task that is falling behind could also impede the progress of other dependent tasks if undesirable scheduling takes place [65].

***Debugging*** A major problem in having tasks execute concurrently is that they may need to access the same data, and the order in which this is done may change the correctness of the program. This is known as a race condition, and this non-deterministic behaviour makes it difficult to reproduce when attempting to understand the error [34]. The debugging process itself may actually interfere with the monitored program, therefore altering its behaviour and not necessarily displaying the same behaviour without the debugging mode. Debugging has become increasingly complicated and tools to assist developers are limited [45]. Debugging of a parallel program may generally mean that a large amount of data needs to be gathered which may be difficult to manage.

### 2.1.2 Additional challenges for desktop parallelisation

The challenges discussed in section 2.1.1 apply to parallel programs in general. The development of parallel desktop applications involves additional challenges that generally did not apply to traditional parallel programs. Such challenges need to be discussed since parallelisation is needed on the desktop. So far, users have been interested in faster machines; hence, it may be assumed that they are still interested. Parallelisation is therefore needed otherwise there will be no improvement given the current trend in processor technology, as well as increasing user demands.

***Different application types*** Most of the effort for parallel computing has focused on theoretically trivial applications that possess a high degree of inherent parallelism. The decomposition of such problems is often trivial and the resulting tasks have minimal, if any, communication between each other. Such problems are commonly termed *embarrassingly parallel* and usually fall into the area of scientific, engineering and database problems that are usually highly regular in structure [55]. These ideal candidates for parallelisation also tend to be computationally intensive and require little, if any, user intervention during their execution.

However, desktop users have different needs and are generally not interested in running such scientific and engineering applications like weather forecasting programs on their desktop. This contributes to why desktops were not traditionally equipped with numerous processors. However, parallelisation on the desktop is of interest where day to day users may benefit. So the question is, what are those applications that desktop users run that may benefit from parallelisation? The first concern is to address the issue why parallelism has not been exploited for daily desktop users.

The first clear observation is that most applications running on the desktop cannot be classified as embarrassingly parallel problems. The types of problems that are likely to run on a desktop are irregular in nature, therefore making the issues of task decomposition and dependency analysis more troublesome for developers. As well as being irregular, desktop applications are rather interactive, constantly accepting input from the user and are not as computationally demanding. Irregular problems tend to be dealt with using threads where sections of the program are identified as having the potential to be executed simultaneously alongside other sections of the program.

***Graphical user interfaces*** As will be discussed in section 2.3, GUI applications have an additional threading demand compared to their console-equivalent versions. Since most desktop applications involve a GUI, this further complicates the parallelisation of desktop applications as programmers must serialise all GUI-related aspects [21, 62, 73, 79].

***Actual performance versus perceived performance*** The traditional motivation behind parallelisation has been to reduce wall-clock time, i.e. the actual time to execute the program. Although reducing this physical time will in reality improve the *actual performance*, desktop users might not necessarily concur with this improved performance. Desktop users typically rely on mental estimations rather than objective measurements to determine duration (and consequently performance) [89]. It is therefore the user's *perceived performance* that ultimately dictates performance for interactive desktop applications [44]. To truly benefit from multi-core desktop systems, programmers must now strive to improve both actual time *and* perceived time:

- Actual time is important to compare the application against benchmarks and verify performance objectives.
- Perceived time is important for a successfully responsive application and positive user experience.

So if reducing the execution time (for example by parallelisation) does not necessarily contribute to perceived performance, what else can the programmer do? Possible examples include ensuring that the user interface does not "freeze", progress indication and completing tasks in an expected order. Parallel programmers must now be concerned with such human-computer interaction (HCI) issues that were traditionally unnecessary for parallel computing, especially in the engineering and scientific fields.

***Short runtime*** Desktop applications tend to be interactive in nature, where the user initiates tasks with a series of input events. During the execution of these tasks, the user usually remains seated in front of the desktop waiting for the tasks to complete. Most of these tasks tend to have a relatively small runtime in comparison to the computationally intensive programs that would normally run on high performance computers. The user generally expects most tasks to be accomplished "soon" while they wait.

Since these tasks tend to have short runtimes anyway, chances are that improving their response time through parallelisation would simply go unnoticed by the user. One of the leading contributors as to why such an attempt would be futile is the fact that sequential overhead will be introduced in the parallelisation process. The overhead will be introduced when decomposing the problem into tasks, the communication between the tasks, and interpreting the tasks at the end. If all this overhead is significant in comparison to the parallelised computation, then this will actually have an adverse effect on the overall execution time of the task.

In the past, the benefits of parallelising desktop applications were usually shadowed by the cost of this extra effort meaning that applications were simply not parallelised. This is due to the non-trivial nature of parallelising most problems that are likely to run on desktops. It has been easier for desktop users to "just wait" until a faster processor became available (or more affordable) in order to run their day to day applications (also, multi-cores were not the norm). This contributes as to why there was little pressure in the past to use parallelism in order to improve desktop performance. The aim now is to change this, making parallelisation more common on desktop environments.

Introduced overhead is only acceptable when the simultaneous execution of multiple parts of the program will eventually compensate for the overhead. In these cases, even though some regions experienced a delay, the overall execution time is reduced. However, desktop applications are overhead sensitive as such delays will impact the user's experience. The consequence of having such overhead sensitivity means that a lot of desktop tasks may never be parallelised, and the focus will become on large time demanding tasks.

***Non-dedicated*** Developers have mostly produced parallel programs under the assumption that it will be the sole program executing on the system [57, 80]. Such an assumption is especially necessary when using parallel models such as message passing that involve explicit parallelism. There would be minimal amounts of system processes, if any, to interrupt the execution of the parallel program, simply because issues such as scheduling are already taken care of.

Unfortunately, developers do not have the luxury of making this assumption when developing parallel desktop applications. A number of varying programs will be running alongside the parallel application. The set of applications on the user's desktop cannot possibly be determined by the developer. Even if the user decides to "dedicate" the system for a parallel application, there will still be system processes that will periodically interrupt. Linux, for example, would typically have as much as 100 processes once booted when the user is doing "nothing".

Due to the high rate of context switching that occurs in operating systems, this increases the chance of poor performance due to the fact that when a thread is rescheduled it may have some of the cache contents it requires overwritten [57]. Even small system noise, including the periodic clock tick, has been found to interfere with the performance of fine-grained parallel applications [105]. Consequently, this all means that the program will not display the intended performance as it competes for resources with other applications.

***Non-deterministic*** Traditional computationally intensive scientific applications running on high performance computers will generally follow a predetermined execution flow. Once the program has been set up, it will largely run with minimal user intervention until it finishes. Consequently, the developer knows what the computation to be performed is and may therefore optimise.

However, desktop applications possess a high degree of non-determinism due to their interactive nature. There is non-determinism in what parts of the application will be executed, and when they will execute. There is also non-determinism in the workload that the application will be exposed to, not to mention that the workload would vary from time to time.

Interactive applications differ from batch-like programs in the sense that the developer does not know what will be the exact execution flow of the program. Such non-deterministic programs means it is unknown until runtime what the data is, therefore making it difficult, if not impossible, for the developer to determine how to decompose the problem [15] since the program flow is dependent on input from the user.

Execution of certain applications cannot begin until the user performs the necessary activating action. This sporadically timed action is accompanied with input parameters that are possibly from an infinite data range, which will determine the specific instance of the problem to be executed. This non-determinism means that strategies to improve performance cannot be exercised during development. The ability to decompose and perform dependence analysis is extremely impaired, and the developer cannot perform static scheduling since the tasks are unknown.

***Unknown target system*** Section 2.1.1 discussed how parallel programs are typically optimised to maximise their performance. However, it is difficult for the developer of a desktop application to know the target system, especially when there is such a diverse range of desktop systems available. Consequently, the developer needs to make assumptions and simplifications about the target system's architecture and components. The effect of not knowing the details of the target platform means that the parallel program cannot be optimised.

How many processors will there be available for the application to use? Factors such as the architecture and cache level may also impact the design of the parallel application. Cache thrashing has a worse impact on parallel machines since a miss could also involve communication and cache coherency between the processors [55].

Desktop applications need to be able to run on a large variety of hardware. Some users might have a desktop with just a uni-processor, while others may have a dual-core, a quad-core processor, or other multi-core processor. Yet users would certainly expect the application to be capable of performing well on all hardware. Developers will not reach a large market of users if the software they produce is not suitable for certain systems. The consequence of this uncertainty means that developers are limited when it comes to developing optimised applications for desktops. Optimisation must be dynamic at runtime which is different to conventional parallel systems.

***External influences*** Some candidate desktop applications could theoretically benefit from parallelism, and might even be categorised as embarrassingly parallel. Unfortunately, achieving the theoretical parallelism is currently impractical due to external practical factors. These factors limit the rate of information that is available for keeping the processor busy. Highly parallel file searching may be limited by the disk bandwidth. Internet downloading is limited by network speed. Consequently, some high latency tasks disturbing the desktop user cannot be remedied with parallel computing due to the slow I/O speed of networks and disks.

## 2.2 Data parallelism

In object-oriented programming, *collections* (or containers) are used to store objects (i.e. *elements*). They come in many forms, including vectors, linked-lists, trees, sets, maps, stacks and so on. Some of these collections are random-access (amortised constant-time access to random elements), e.g. array-lists, while others are inherently sequential, e.g. linked-lists. Regardless of the collection type, *iterators* provide a consistent means to access the elements. Iterators are not only used for basic collections, but also for complex data types, for example traversing an XML document.

### 2.2.1 The Java-style sequential iterator

The Java-style sequential iterator provides two primary methods. The first method, `hasNext()`, inquires to see if at least one element remains to be traversed. If this returns `true`, then `next()` can be invoked to retrieve that element. To illustrate the concept of the Parallel Iterator, consider the following very simple, but typical, sequential code segment making use of a sequential iterator to resize each image in a list of images:

```
List list = getImages();
Iterator<Image> it = list.iterator();
while (it.hasNext()) {
  Image image = it.next();
  resize(image);
}
```

The example above traverses an iterator using a `while` loop. Alternatively, it is just as common to achieve the same using a `for` loop:

```
List list = getImages();
for (Iterator<Image> it = list.iterator() ; it.hasNext() ; ) {
  Image image = it.next();
  resize(image);
}
```

The `for` loop has the advantage of limiting the iterator's scope to the loop [14], but the `while` loop has the advantage of increased legibility. Although the `while` loop has been favoured for the examples throughout this thesis, programmers may use either style they prefer with the Parallel Iterator.

### 2.2.2 Traditional parallelism approaches

The inherent parallelism of the above example is to resize the images in parallel. Unfortunately, it is not thread-safe in a parallel environment to simply share the sequential iterator since it leads to a classical race condition. The sequential Java-style iterator requires two separate method calls in order to retrieve an element: the first (`hasNext()`) to check if any element exists, while the second (`next()`) actually retrieves it. Multiple calls to `hasNext()` may return `true` to more than one thread, when in fact only one element remains in the iterator. This causes problems when the multiple threads invoke `next()`, as only one thread will be successful.

Consequently, what are the possible tools to be used by a desktop application developer? Thread libraries are available for most object-oriented languages, the easiest solutions a developer could implement are discussed below (the performance of these approaches compared to the Parallel Iterator are presented in section 4.2.1):

- *Locking on each iteration:*
  In this approach, a sequential iterator is created and shared amongst all the threads:

  ```
  Iterator<Image> it = list.iterator();
  Lock lock = new ReentrantLock();
  CountDownLatch barrier = ...
  ```

  In order to be thread-safe, any thread that attempts to get an element must gain exclusive access to the iterator using a lock: this allows the thread to atomically call `hasNext()` and `next()`. Notice that the programmer is responsible to ensure thread-safe access to the iterator, as well as ensuring all threads finish the loop at the same time:

  ```
  // each thread does this
  while (true) {
      lock.lock();
          if (it.hasNext()) {
              Image image = it.next();
              lock.unlock();
              resize(image);
          } else {
              lock.unlock();
              barrier.countDown();
              barrier.await();  // wait for other threads
              break;
          }
      }
  ```

- *Concurrent collection:*
  This approach involves using a thread-safe *queue* (as in Java's `java.util.concurrent` package) which results in a small-grain dynamic load distribution:

  ```
  ConcurrentLinkedQueue<Image> queue =
            new ConcurrentLinkedQueue<Image>(list);
  ```

  By sharing this collection with all the threads, iterations are distributed one at a time as a thread requests work (note that this queue does not permit `null` elements):

```
// each thread does this
Image image = null;
while ((image = queue.poll()) != null) {
    resize(image);
}
barrier.countDown();
barrier.await();  // wait for other threads
```

- *Synchronized method:*
  Another common approach is to create a monitor by using `synchronized` methods (or `synchronized` statements) as supported by Java. This causes threads to acquire the object's *intrinsic lock* [62] before processing the `synchronized` code:

  ```
  public synchronized Image getNext() {
      if (it.hasNext())
          return it.next();
          else return null;
  }
  ```

  The user-code then looks like the following, where `myObj` denotes the object for which the intrinsic lock is acquired:

  ```
  // each thread does this
  Image image = null;
  while ((image = myObj.getNext()) != null) {
      resize(image);
  }
  barrier.countDown();
  barrier.await();  // wait for other threads
  ```

- *Static decomposition of the collection:*
  This involves manually creating a *static distribution* of the images. The collection is distributed into multiple sub-collections where each sub-collection is assigned to a thread. This corresponds to a static scheduling policy where iterations are assigned to threads before any work begins:

```
for each thread {
    List<Image> privateList = list.subList(...);
    Iterator<Image> it = privateList.iterator();
    ...
}
```

Unfortunately, the above solutions are unfavourable for several reasons:

- There is a considerable amount of manual programming effort required by the programmer. This includes the responsibility to ensure thread-safety depending on the respective approach taken (for example, a barrier synchronisation at the end of each loop to ensure no thread progresses past the loop while other threads are still traversing their share),
- Each of the above approaches resembles a single scheduling policy. This reduces flexibility since:
  - The various load distribution (due to the scheduling policy) might not be adequate to achieve good performance (as will be discussed in section 4.2.1), and
  - If the programmer desires a different scheduling policy, not only will this involve considerably more work on the implementation side, but the user's iteration code might need to be modified also (notice how the iteration code is different for each of the approaches presented above).
- Code concerning the original program model (the business logic, here image resizing) becomes tangled with code dealing with parallelism [58] as it requires restructuring the original code. In fact, the amount of restructuring in this particular example was quite minimal since the business logic was already in its own method.

## 2.3 Task parallelism

In parallelising desktop applications, one must first understand the structure of desktop applications and the threading model that programmers must adhere to.

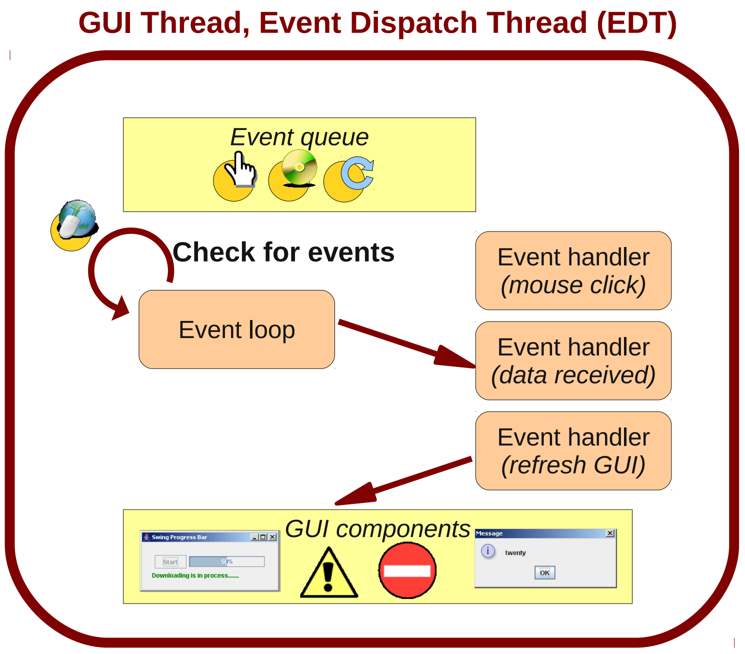

GUI Thread, Event Dispatch Thread (EDT)
Event queue
Check for events
Event loop
Event handler (mouse click)
Event handler (data received)
Event handler (refresh GUI)
GUI components

Figure 2.1: A vital component of interactive graphical applications is the event loop. Here, the GUI responds to events by executing the respective event handler. The application appears unresponsive if events backlog in the event queue; it is therefore important that control of the GUI thread remains in the event loop.

### 2.3.1 Graphical user interfaces

Desktop applications allow users to interact through a graphical user interface (GUI). The application displays a range of visual components, some acting as a form of input (e.g. buttons, text fields) while others display application status (e.g. labels, progress bars). Such an application would be based on the event-driven paradigm, where the program's execution flow is determined by *events* (e.g. mouse clicks, messages from other threads). As figure 2.1 shows, the application has an *event loop* waiting for events to arrive. The events are then dispatched to the appropriate *event handler* to take the appropriate action.

Many toolkits are available for programmers to ease the development of event-based applications. These toolkits provide many graphical components as well as the event loop and event handling. Generally, programmers only need to specify the logic of event handlers (e.g. the response to a certain button being clicked). Programmers must ensure that these event handlers are short so that control returns to the event loop: otherwise, events will backlog and the application appears unresponsive (for example, GUI applications appear to "freeze" when the GUI does not refresh). Figure 2.1 also shows how GUI components are typically manipulated by a single thread, hence the name *GUI thread*.

### 2.3.2 Where are the multi-threaded toolkits?

Most GUI toolkits available are single-threaded. Only one dedicated thread is allowed to access the GUI components: the *GUI thread*. In Qt, the *main thread* [103] (the initial thread that starts the program) is also the GUI thread. Java is similar in that it allows only one thread to access GUI components, however this thread is not the main thread: it is a special thread created by Java, known as the *event dispatch thread* (EDT) [68]. In most Java GUI applications, the main thread will exit once the GUI and EDT are started. Why is it that such GUI toolkits are single-threaded? Even more interestingly, why are they single-threaded even though they are part of a library that contains threading support?

In attempting to develop a thread-safe GUI toolkit, one faces the standard parallelisation challenges. To avoid non-deterministic behaviour, mutual exclusion (usually implemented with locks) is required when multiple threads access shared data (and at least one thread is modifying the data). Even though such synchronisation protects the data by serialising the access, it unfortunately introduces other problems. The programmer must now decide the granularity of mutual exclusion: coarse-grained locking reduces the concurrency while fine-grained locking complicates the programming and increases the possibility of deadlock.

Ultimately, a decision has to be made whether multi-threading will be supported in the toolkit. If the answer is yes, then all aspects of the toolkit need to be thread-safe. This incurs a performance penalty for applications not multi-threaded. Combined with the performance versus programmability trade-off toolkit implementers face, GUI toolkits are single-threaded [21, 62, 73, 79] (this decision was taken before the arrival of mainstream multi-cores). In fact, the single-threaded model discussed above is not limited only to GUI toolkits. As an example, some native libraries require that all access be made from the same thread [21].

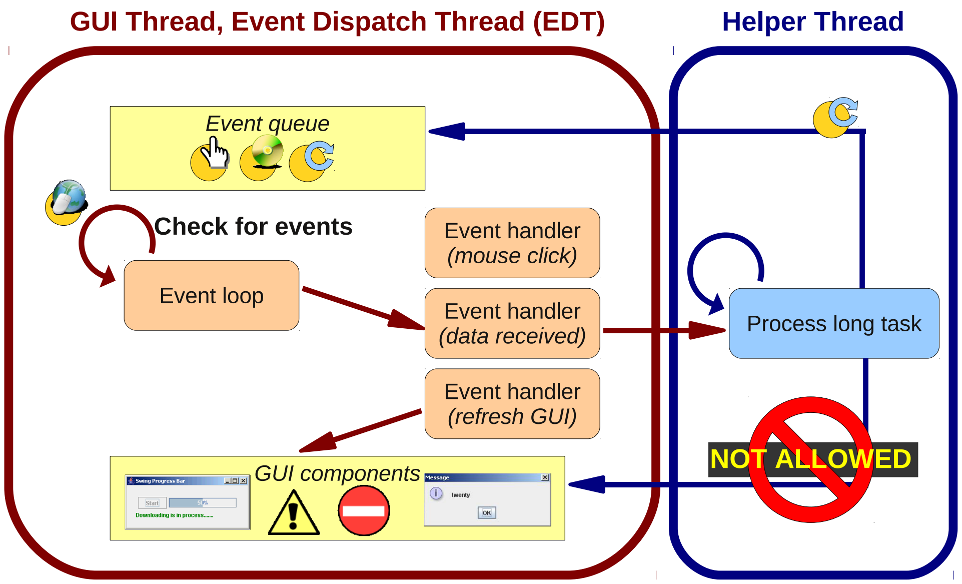

GUI Thread, Event Dispatch Thread (EDT)
Helper Thread
Event queue
Check for events
Event loop
Event handler (mouse click)
Event handler (data received)
Event handler (refresh GUI)
Process long task
NOT ALLOWED
GUI components

Figure 2.2: Structure of a multi-threaded Java GUI application. Notice that the purpose of multi-threading so far is primarily to improve the application's *responsiveness* (rather than improve *performance* through parallelism). This is achieved by dispatching all long executing tasks to a helper thread, allowing the EDT to return to the event loop.

### 2.3.3 Multi-threaded GUI applications

Although ParaTask has been implemented for Java, it is also applicable to other object-oriented languages. For example, many of the concepts presented here were initially prototyped using C++(Qt) [50]. ParaTask has matured considerably since then, the improved semantics are still applicable to other object-oriented languages. In order to simplify discussions, the examples from herein shall only focus on Java.

As previously mentioned, any Java GUI application (AWT or Swing) involves at least 2 threads. The *main thread* is the initial thread created by the Java Virtual Machine (JVM) to start executing the application's main method. An *event dispatch thread (EDT)* is also created by the JVM and is used to execute the event handler code listening to the respective events. Since Java's main thread usually serves no purpose as soon as the EDT is initialised, it subsequently terminates [95].

The programmer's understanding of a multi-threaded Java GUI application is depicted in figure 2.2. Since the EDT is responsible for responding to events, the event handlers must complete quickly to maintain a responsive user interface. In situations where event handlers require more time to complete, such code must be executed on another *helper thread* to allow the EDT control of the event loop. Programmers are responsible in creating these additional helper threads and ensuring they only perform background work. Only the EDT is responsible for GUI manipulations, such as painting components on the screen since the GUI toolkit is not coded thread-safe (section 2.3.2). Using such a model, it is not recommended for the main method to even initialise the GUI [95] (programmers may schedule such code to execute on the EDT using `invokeLater()` or `invokeAndWait()`).

In fact, even the "multi-threaded" application of figure 2.2 might be considered inadequate to benefit from multi-cores; the helper thread is overwhelmed with all the computational workload, while the EDT essentially remains idle waiting for more events. This is because the purpose of multi-threading here was to achieve responsiveness by freeing the EDT. Even if the application will knowingly only execute on a uni-processor, multi-threading would still be implemented. However, desktop applications must now be multi-threaded with a different goal in mind: exploiting the inherent parallelism of these multi-core processors.

### 2.3.4 The threading model versus the tasking model

#### The threading model

In order to improve the parallelism, the programmer may decide to create a new thread for each computation. Although this has the potential to exploit the parallelism, many difficulties are encountered. From a performance point of view, this incurs excess overhead that causes the application response time to suffer:

- Creating new threads solely for the purpose of executing short-lived computations presents extra work for the JVM. This includes creating the thread, starting it and then cleaning up after it terminates [62].

- *Oversubscription* occurs, where the number of active threads exceeds the available number of cores. The performance could easily degrade due to resource contention, scheduling overheads and memory bandwidth limits [23]. The high number of threads for the application could in turn reduce fairness for other applications to execute on the available cores [81].

As well as high overheads, this threading model also presents difficulty for the programmer:

- The code of the independent computations must be migrated into the `run()` method of a `Thread` class. This is performed for each of the independent computations. This is further complicated if the original code made reference to any shared variables.

- The model might introduce coupling that was not present in the original sequential program. For example, notice the dependences between the 4 computations (the example in section 1): `compute2` and `compute3` must wait for `compute1` to complete, and `compute4` must wait for both `compute2` and `compute3` to complete. The programmer must now enforce these dependences using condition variables amongst the different computations. Not only does this introduce coupling amongst the computations, but it also reduces the re-use of these computations for other applications that do not have these dependences.

**The tasking model**

In overcoming the performance issues of the threading model, a tasking model is usually implemented. This involves having a fixed pool of threads that are ready to execute tasks [62, 81]. The thread pool technique serves well for applications with many short-lived independent computations. Rather than creating threads, programmers encapsulate independent code within some form of a lightweight task object. These task objects are then scheduled to be executed by one of the threads in the pool.

Although the tasking model addresses the performance issues of the threading model, it generally does not address the programming difficulty. Programmers are typically still required to restructure the code into task objects (such as `Runnable`) and handle any dependences amongst these tasks. In particular, this still presents the programmer with the problem of newly introduced coupling amongst the tasks.

### 2.3.5 Typical GUI parallelisation approaches

Since this research focuses on the parallelisation of GUI applications, let us have a look at the current approaches a programmer may take to parallelise a GUI application (in Java as a case study). Unfortunately, programmers are limited to only two options. Section 6.2.1 discusses the performance of these approaches in comparison to ParaTask. For illustration, the same image application example of figure 1.1 is parallelised.

#### Java Threads

Threads have been an integral part of Java since its initial release. Consequently, parallelising a GUI application manually using Java Threads has traditionally been the norm. The first step in this approach is to offload all the computation away from the EDT and into helper thread(s). Since the programmer is interested in improving performance due to parallelism (and not only just to improve application responsiveness), the computation is offloaded into multiple helper threads. A possible solution may resemble that below:

```
public void actionPerformed(ActionEvent e) {
  final Thread t1 = new Thread() {
    public void run() {
      File f1 = compute1(‘‘myimage.jpg’’);
    }
  };
  final Thread t2 = new Thread() {
    public void run() {
      t1.join();
      File f2 = compute2(‘‘myimage1.jpg’’);
    }
  };
  final Thread t3 = new Thread() {
    public void run() {
      t1.join();
      File f3 = compute3(‘‘myimage1.jpg’’);
    }
  };
  Thread t4 = new Thread() {
    public void run() {
      t2.join();
      t3.join();
      final File f4 = compute4(‘‘myimage2.jpg’’, ‘‘myimage3.jpg’’);
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          display(‘‘myimage4.jpg’’);
        }
      });
    }
  };
  t1.start();
  t2.start();
  t3.start();
  t4.start();
}
```

The above code has many undesirable aspects:

- In terms of improving the program *responsiveness*, at least one helper thread must be created (to free the EDT by allowing `actionPerformed()` to complete immediately). In terms of improving the program *performance*, at least 2 helper threads must be created (since in this example the inherent parallelism only allows for tasks 2 and 3 to execute concurrently). However, 4 helper threads have been used here to simplify the example; developing an optimised implementation will require even more effort on the programmer's behalf (for example sharing condition variables amongst the threads, creating and managing a thread pool, and so on).

- Notice the explicit use of `invokeLater()`. This is necessary since the `display()` method may only be computed on the EDT (as discussed in section 2.3.3).

- The original program code has become largely tangled with parallelisation code. This tangling of concerns significantly reduces the code legibility [58].

- The concurrency is coordinated by the caller, rather than the callee, therefore breaking encapsulation [85] (discussed further in section 5.7).

- In addition to reduced legibility, the tangling of concerns reduces code reuse since the different tasks are now coupled with each other (for example, the second thread waiting for the first thread).

There are variations as to how Java threads might be used. In the code snippet above, a `Thread` is created for *every* task. For a large number of fine-grained tasks, the excessive number of threads will decrease performance (as was discussed in section 2.3.4). Therefore, the programmer may decide to create a fixed number of threads and manually assign the tasks to the different threads. Although such a static decomposition reduces the runtime overhead, this unfortunately reduces the load-balancing [3]; it is additional effort on the programmer's behalf to find the appropriate balance. Both of these approaches are compared in section 6.2.1.

### SwingWorker

SwingWorker is a significant improvement compared to manually using Java Threads. Rather than placing code inside a `Thread`, the code is placed within a lightweight `SwingWorker` instance. There are 2 methods to override: time consuming computations are placed within `doInBackground()`, while the optional `done()` is used for GUI-related computations.

```
public void actionPerformed(ActionEvent e) {
  final SwingWorker<File, Void> sw1 = new SwingWorker<File, Void>() {
    protected File doInBackground() throws Exception {
      return compute1("myimage.jpg");
    }
  };
  final SwingWorker<File, Void> sw2 = new SwingWorker<File, Void>() {
    protected File doInBackground() throws Exception {
      sw1.get();
      return compute2("myimage1.jpg");
    }
  };
  final SwingWorker<File, Void> sw3 = new SwingWorker<File, Void>()   {
    protected File doInBackground() throws Exception {
      sw1.get();
      return compute3("myimage1.jpg");
    }
  };
  SwingWorker<File, Void> sw4 = new SwingWorker<File, Void>() {
    protected File doInBackground() throws Exception {
      sw2.get();
      sw3.get();
      return compute4(‘‘myimage2.jpg’’, ‘‘myimage3.jpg’’);
    }
    protected void done() {
      display(‘‘myimage4.jpg’’);
    }
  };
  sw1.execute();
  sw2.execute();
  sw3.execute();
  sw4.execute();
}
```

Compared to the approach of manually using Java Threads, the advantage here is that the programmer no longer needs to be concerned with manually creating threads and managing a thread pool. However, many of the same disadvantages still exist (for example: tangling parallelisation code, breaking of encapsulation, coupling amongst tasks and so on). This thesis addresses these problems in the following chapters.

# Chapter 3

# Parallel Iterator concept and related work

This chapter introduces the *Parallel Iterator* concept [48, 49, 51] as a data parallelism solution for object-oriented computations. The concept is then compared to related work that target object-oriented data parallelism. The next chapter discusses the implementation and experimental results.

## 3.1 Parallel Iterator concept

The solution proposed here is the concept of a *Parallel Iterator* (targeted for data-parallelism, as discussed in section 2.2). It may conceptually be considered a thread-safe wrapper around a sequential iterator. Although the Parallel Iterator does not manage thread *creation*, it does possess awareness of the threads accessing it. Threads may be created using threading libraries or OpenMP for C++ (i.e. `#pragma parallel` as used in the last example of section 3.1.1). For illustration, simple examples will first be presented throughout this section; however, the strength of the Parallel Iterator is demonstrated with more complex examples (for example iterating multiple collections in section 3.1.7 and traversing complex data collections in section 3.1.8).

### 3.1.1 Interface and usage

The Parallel Iterator concept has been implemented for two object-oriented languages: Java and C++(Qt). Regardless of the language, the underlying concepts remain the same. The Parallel Iterator uses the same standard interface of the sequential iterator: `hasNext()` returns a boolean denoting whether there are any elements remaining while `next()` returns the next element.

However, there is a slight modification to the usage contract of these methods. If thread $A$ invokes `hasNext()` and `true` is returned, the Parallel Iterator reserves at least one element for thread $A$. Consequently, only thread $A$ can access that element by invoking `next()`. Even if there was only one element remaining in the iterator, all other threads will receive `false` since that element has been allocated to thread $A$. The implication of this approach is that any thread that receives a `true` from `hasNext()` must eventually follow up with a `next()` invocation, otherwise that element will not be traversed by any other thread. If `hasNext()` returns `true`, any subsequent call to `hasNext()` has no effect and will continue to return `true` until `next()` is eventually called (i.e. multiple calls to `hasNext()` will not accumulate more elements).

Each thread must not only have pairing invocations of `hasNext()` and `next()`, but every thread is required to continue invoking the two until `hasNext()` returns `false` . The first reason is because `hasNext()` serves as a synchronisation barrier (explained below). The second reason is because `hasNext()` in general reserves more than one iteration. Similarly, a thread must not invoke `next()` without first receiving `true` from `hasNext()` since an element needs to be allocated first (a run-time exception is thrown in such a case).

Below is the parallel version of the image resizing application using the Java implementation of the Parallel Iterator:

```
List list = getFiles();
ParIterator<Image> it = ParIterator.createParIterator(list);

// each thread does this (using OpenMP or threads)
while (it.hasNext()) {
  Image image = it.next();
  resize(image);
} // Parallel Iterator implicit barrier
```

All logic in regards to scheduling policy and synchronisation are contained within the Parallel Iterator. From the user's point of view, the Parallel Iterator is an ordinary iterator providing a uniform and thread-safe means to traverse elements in a collection. As shown above, the parallel iteration code remains the same as the sequential version; in fact, the interface of the Java Parallel Iterator extends the standard Java `Iterator` interface. Note that although the Java `Iterator` interface specifies a `remove()` method, not all implementations support this (such implementations throw an `UnsupportedOperationException`). Some of the current Parallel Iterator implementations take this approach, since removing elements from a collection in a thread-safe manner depends on the collection type; letting alone the difficulty of defining parallel semantics for `remove()` in an iterator. However, in section 3.1.3.4, these difficulties are discussed in detail and a compromising solution is suggested.

### Implicit barrier synchronisation

In order to preserve semantics of the sequential iterator, the last call to `hasNext()` contains an implicit barrier synchronisation; all threads that have completed their iterations will block at the loop boundary waiting for all other threads to complete. When all the iterations have been completed and the last thread calls `hasNext()`, the barrier synchronisation is released and all threads receive `false`. Therefore, a thread will only receive a `false` when every other thread completes. Consequently, all iterations have been completed before any thread proceeds past the loop (therefore preserving sequential semantics).

If the programmer insists on not having the default barrier, they may explicitly turn it off when the Parallel Iterator is created (section 3.1.2). This might be useful where the threading environment already has a barrier enforced (for example, OpenMP work-sharing constructs or ParaTask's optional multi-task barrier in section 5.3.3).

### Alternative interface

In the interface discussed above, the `hasNext()` and `next()` methods are somewhat connected: an invocation of `next()` must be preceded by a `hasNext()` invocation (that returned `true`), and a `hasNext()` invocation that returns `true` must be followed by a `next()` invocation. Consequently, an alternative interface is available for languages that support *reference variables* (such as C++). This interface combines the two methods into a single *atomic* `boolean next(E& e)` method, where `E&` denotes a *pass by reference* parameter (as opposed to *pass by value*) to an element of type `E`.

It is atomic in the sense that it combines the sequential iterator's `next()` and `hasNext()` methods into a single method. In this way, `next()` is used in an atomic *test and set* fashion. The Parallel Iterator first *tests* to see if an element remains. If there is an element remaining for the *requesting thread* (the thread calling `next()`), then the reference variable (`e`) is *set* to that element and `true` is returned in an indivisible operation. Otherwise, `false` is returned when no more elements exist for the requesting thread.

To use the Parallel Iterator's alternative interface for the image resizing example, the required code is presented below (using C++ syntax). OpenMP's `parallel` construct is used, causing multiple threads to execute the `while` loop. The only change compared to the sequential version is the use of the Parallel Iterator and condensing the `hasNext()` and `next()` methods into one atomic `next()` method. The approach and structure of the program remain the same.

```
List list = getFiles();
ParIterator<Image> *pi =
        ParIterator::createParIterator<Image>(list);
#pragma omp parallel {
  Image image;
  while (pi->next(image)) {
    resize(image);
  } // Parallel Iterator implicit barrier
}
```

Figure 3.1 provides a visualisation how the above code will work. It shows two threads, $A$ and $B$, that have shared access to the Parallel Iterator `pi`. Each thread has a private variable `image`, which is passed to the Parallel Iterator to be set. In the situation `image` is set to an element, `true` is returned (e.g. thread $A$). Otherwise, `false` is returned (e.g. thread $B$) meaning no more iterations exist for that thread.

This example shows the Parallel Iterator's flexibility to be used with either OpenMP [82] or a threading library to manage thread creation. The advantage of using OpenMP in combination with the Parallel Iterator is that the structure and context of the program remain unchanged (note that an OpenMP-like interface is also available for Java [24]).

### 3.1.2 Construction

A factory class is provided to simplify the creation of a Parallel Iterator. The examples in section 3.1.1 illustrate the default creation of the Parallel Iterator. The user is required to at least supply the collection containing the elements to

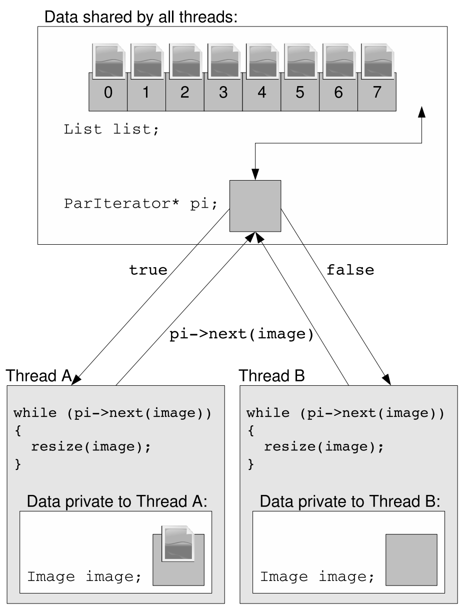


Figure 3.1: Visualisation of the Parallel Iterator usage (alternative interface), situation shown after the last iteration has been allocated to thread $A$.

traverse. For added flexibility, the user may also specify a scheduling policy and chunk size in order to override the default policies (discussed in section 3.1.3.1) as well as the number of threads accessing the Parallel Iterator (defaults to the number of processors). The programmer also has the option to turn off the synchronisation barrier:

```
public static ParIterator createParIterator (
              Collection collection,
              PI.Schedule schedulePol /* optional */ ,
              int chunksize           /* optional */ ,
              int threadCount         /* optional */ ),
              boolean barrierOff      /* optional */ );
```

In order to have a single programming paradigm, the Parallel Iterator may be used to also traverse arrays and integer ranges. Traversing an array is made possible by specifying the array to the Parallel Iterator:

```
public static ParIterator createParIterator (
              Object[] array,
              PI.Schedule schedulePol /* optional */ ,
              int chunksize           /* optional */ ,
              int threadCount         /* optional */ ),
              boolean barrierOff      /* optional */ );
```

Consider a developer simply wishing to traverse over an integer range. It would be redundant if a collection had to be populated just to specify the range. In such a case, the developer may specify the starting value, size of the range, and optionally the increment (also known as stride) between values (which defaults to `1`) as shown below:

```
public static ParIterator createParIterator (
                 int start,
                 int size,
                 int increment           /* optional */ ,
                 PI.Schedule schedulePol /* optional */ ,
                 int chunksize           /* optional */ ,
                 int threadCount         /* optional */ ),
                 boolean barrierOff      /* optional */ );
```

The above code is shown for the Java implementation, but the C++(Qt) Parallel Iterator also has the equivalent implementation.

### 3.1.3 Semantics

The Parallel Iterator may conceptually be considered a thread-safe wrapper around a sequential iterator. As a consequence, it can virtually support all collections. How this is possible for inherently sequential collections is explained in the implementation section (section 4.1).

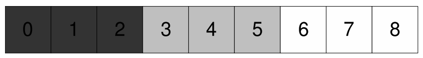
One important assumption is that iterations can be processed in any order, because the Parallel Iterator will not process them in sequential order. If a certain order is necessary for *all* the loop iterations, then such a loop cannot be parallelised. However, if only a *partial* order is necessary, a Parallel Iterator may be created that enforces the (partial) order (demonstrated in section 3.1.3.1 and section 3.1.8). Hence, again the implementation details are encapsulated in the Parallel Iterator.

Note that access to shared variables and synchronisation in the loop body are not addressed by the Parallel Iterator. These can be handled in the usual way (such as locks from thread libraries or the corresponding primitives and clauses in OpenMP). From the Parallel Iterator perspective, any thread may safely modify elements it receives from the Parallel Iterator, but elements should not be inserted into or deleted from the collection being traversed (just like Java's *fail-fast iterators*). In section 3.1.3.4, semantics for supporting element removal is discussed.

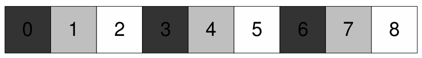
#### 3.1.3.1 Scheduling policies and chunk size

A scheduling policy determines how the iteration space is divided amongst threads into smaller *chunks*. The Parallel Iterator supports *static*, *dynamic* and *guided* scheduling policies [82]. A *chunk size* may also be specified, where the next `chunksize` iterations are reserved for the same thread. The purpose of the chunk size is to find the best trade-off between good load balancing and low overhead [3]. Figure 3.2 shows examples of the major scheduling policies when 3 threads iterate over a collection. In the following the available policies are detailed [3, 61]; the necessary parameters for the Parallel Iterator to achieve the respective scheduling policy are also shown. Let $n$ be the number of iterations to be distributed amongst $p$ threads.

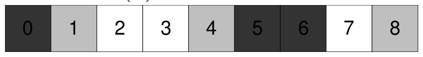
- *Static*: all iterations are assigned to threads before the execution of the loop. This may either be *block* or *cyclic*:
  - *Block*: each thread is assigned one large chunk. If $\lceil n/p \rceil = \lfloor n/p \rfloor$, i.e. the number of iterations is divisible by the number of processors, each thread gets $n/p$ iterations. Otherwise, i.e. $\lceil n/p \rceil \neq \lfloor n/p \rfloor$, the first $p-q$ threads will get a chunk of $\lceil n/p \rceil$ iterations, while the other $q$ threads get a chunk of $\lfloor n/p \rfloor$ iterations, where $q = p \times \lceil n/p \rceil - n$. Figure 3.2(a) shows static block scheduling, for which the Parallel Iterator parameters are:
    `createParIterator(mycollection, PI.Schedule.STATIC);`
  - *Cyclic*: the iterations are grouped into smaller chunks of `chunksize` iterations and threads are assigned chunks in a round-robin fashion. Figure 3.2(b) shows cyclic scheduling for the example chunk size of 1, for which the necessary Parallel Iterator parameters are:
    `createParIterator(mycollection, PI.Schedule.STATIC, 1);`
- *Dynamic*: each thread requests a chunk of iterations to process. When all iterations of a chunk have completed, another chunk is requested until all



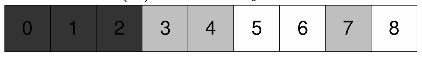
(a) Static block



(b) Static cyclic



(c) Dynamic



(d) Guided

Figure 3.2: The major scheduling policies supported by the Parallel Iterator. These examples show some of the possible ways a collection of nine elements might be allocated amongst three threads (each colour represents a particular thread).

chunks have been assigned. Figure 3.2(c) shows an example of dynamic scheduling with the default chunk size of 1, for which the necessary Parallel Iterator parameters are:

```
createParIterator(mycollection, PI.Schedule.DYNAMIC);
```

- *Guided*: similar to dynamic, except the size of each chunk decreases as iterations are distributed. A thread requesting a new chunk is assigned $q = \lceil r/p \rceil$ iterations, where $r$ is the remaining number of unassigned iterations and $p$ is the number of threads. If $q <$ `chunksize`, then `chunksize` iterations are assigned (except on the last chunk where there may be fewer than `chunksize` iterations remaining). Figure 3.2(d) shows an example of guided scheduling with the default chunk size of 1, for which the necessary Parallel Iterator parameters are:

  ```
  createParIterator(mycollection, PI.Schedule.GUIDED);
  ```

If no policy is specified, then the Parallel Iterator selects dynamic scheduling. The default chunk size for static scheduling is *block* chunk, while the default for dynamic or guided scheduling is 1 (all defaults and chunk size calculations comply with the OpenMP standard [82]). As the experimental results will show in section 4.2, these scheduling policies are very important.

In some cases, an additional purpose that chunk sizes serve is to enforce partial ordering for iterations that have simple ordering constraints. As well as enforcing this partial ordering, the Parallel Iterator hides the details from the programmer. For example, assume a partial ordering on a collection such that every $i^{th}$ element is associated with the $(i+1)^{th}$ element, where $1 \leq i < collectionSize$ and $i$ is an odd integer. In such a situation, a Parallel Iterator may be produced (specifying either dynamic or static scheduling) with a chunk size of 2 (or any positive even integer).

#### 3.1.3.2 Supported collections

The major advantage of the Parallel Iterator is that its usage will be familiar to object-oriented programmers who have used iterators in general and the Java-style sequential iterator in particular. Just as the sequential iterator supports traversal of virtually any collection type, the Parallel Iterator supports this too. Currently supported collections are those available in Qt, namely `QList`, `QVector`, `QSet`, `QLinkedList`, `QMap`, `QHash` and any of their derived subclasses. Similarly, all implementations of Java's `Collection` interface are supported.

The user has a lot of flexibility. For example they may traverse a `Set` in parallel using a dynamic scheduling policy, where each thread is assigned 10 iterations at a time:

```
ParIterator<MyObj> pi = ParIterator.createParIterator(myset,
                                        PI.Schedule.DYNAMIC, 10);
```

The user may just as easily traverse a `LinkedList` in parallel using a static scheduling policy with chunk size of 5:

```
ParIterator<MyObj> pi = ParIterator.createParIterator(mylist,
                                          PI.Schedule.STATIC, 5);
```

In these examples, the actual iteration code does not change even though the scheduling policies and collection types have changed. This provides a real object-oriented approach to parallel traversal of any collection.

#### 3.1.3.3 Modifying elements versus modifying the collection

As a thread accesses elements assigned to it through the Parallel Iterator, it may safely modify those elements; this follows the same policy of the sequential iterator. What this means, is that elements of the existing collection being traversed may be modified in-place (this is not the same as *modifying the collection* by adding or removing elements). Consider the example of figure 3.3 where a linked-list collection, consisting of 9 elements, is shared between two threads $A$ and $B$. Assume that a Parallel Iterator has been created with a static scheduling policy of chunk size 2. In this case, thread $A$ may modify any of the shaded elements without synchronisation since these iterations have been reserved for

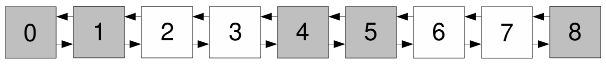


Figure 3.3: Example of static scheduling with chunk size of 2. The elements of this linked-list will be shared amongst two threads; the first thread is assigned the gray elements, while the second thread is assigned the white elements.

it according to the scheduling policy. Similarly, thread $B$ may modify any of the white elements without synchronisation.

When it comes to modifying a collection while it is being traversed by an iterator, different libraries might have different policies. For example, Java's *fail-fast iterators* do not allow direct modification of collections while an iterator traverses the collection (a `ConcurrentModificationException` is thrown if this occurs). Qt, however, allows collections to be directly modified but the non-mutable iterator ignores any modifications and continues to traverse the original collection. Consequently, the Parallel Iterator naturally follows the policy of the respective library used since a sequential iterator is created at the time the Parallel Iterator is created. Namely, this means programmers cannot modify the collection if the Java implementation is used, but may modify the collection if the C++(Qt) implementation is used (but the Parallel Iterator continues to traverse the original collection). This demonstrates the flexible semantics of the Parallel Iterator concept, as it adopts the policies of the sequential iterator for its respective library.

#### 3.1.3.4 Parallel `remove()` semantics

In addition to the `hasNext()` and `next()` methods, some sequential iterators define a `remove()` method. Following on from section 3.1.3.3, this is the only way a collection may be modified while traversing it for a Java-style fail-fast iterator. The `remove()` method in Java's *sequential* `Iterator` interface is defined as follows [94]:

> Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behaviour of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

This is the only way programmers may remove elements from a collection being traversed (the change is applied through the iterator rather than directly to the collection). Extending this concept to the Parallel Iterator is possible, although this will require some compromise between the expected semantics and performance. From the user's point of view, there is no semantic difficulties of a parallel `remove()` since:

- The whole idea behind iterators is that the the underlying collection's implementation is hidden from the user (therefore no elements should appear to "shift" within the original collection), and
- The number of iterations remain the same even if `remove()` is used.

Therefore, the ideal semantic for supporting `remove()` with the Parallel Iterator would be: although elements are removed from the underlying collection, the original scheduling policy stays in place. Namely, the Parallel Iterator continues to traverse the *original* underlying collection, and elements removed by arbitrary threads do not affect the original scheduling policy. Not only might this definition ease implementation, but it is essential for the semantics of traversing a collection in parallel.

### Implementation (and performance) challenges for a parallel `remove()`

Although supporting the above ideal parallel remove semantic is possible, it will unfortunately affect performance. A "true" remove (where the elements are really removed from the underlying collection *during* traversal) would require a clone of the original collection being traversed. This allows arbitrary elements from the underlying collection to be safely removed without affecting the original scheduling policy (i.e. changes in the underlying collection are not reflected in the Parallel Iterator).

Naturally, this cloning process is essential for collections such as arrays (in order to avoid the shifting down of elements). The efficiency may of course be improved, for example using a copy-on-write approach. A viable optimisation would be that the clone is only performed when the first `remove()` is called - therefore avoiding the cloning overhead for loops where no `remove()` is used. In addition, sometimes a partial clone might suffice even when `remove()` is used (e.g. if half the collection has already been traversed, then no need to clone the completed half).

### Proposed semantics for a performant parallel `remove()`

Assume the above semantics of a parallel `remove()` is supported. As is expected when parallelising a loop, it is the case that the iterations are independent of each other. Therefore, the timing of an element's removal from the underlying collection (through the Parallel Iterator) should not affect the correctness of other iterations (since they should be independent). For this reason, the following definition of a parallel `remove()` integrates nicely with the Parallel Iterator:

> Marks for removal from the underlying collection the last element returned by this Parallel Iterator to the current thread. Although the actual removal *might* take place during traversal, the element is only guaranteed to be removed from the underlying collection only when all threads have finished traversal. This method can be called only once per call to `next()`. The behaviour of a Parallel Iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

By only marking the element for removal, this solves implementation difficulties discussed above (while still maintaining acceptable semantics for parallel `remove()`). With this definition, an implementation may decide to delay all deletions until the end of traversal when all threads complete. This means that the underlying collection appears unchanged during traversal (even if elements are removed via the Parallel Iterator); from a semantics point of view, this is acceptable since iterations are supposed to be independent of each other anyway:

```
List<String> lines = getLinesInFile();
ParIterator<String> pi = ParIterator.createParIterator(lines,
                                       PI.Schedule.STATIC);
// each thread does this
while (pi.hasNext()) {
   String s = pi.next();
   if (s.startsWith(“//”)) {
      pi.remove();        // marks the last element for removal
      int size = lines.size();    // changes not reflected yet
   }
} // all removes have been committed at this point
int finalSize = lines.size();
```

In this example, multiple threads traverse a collection where each element represents a line of code in a file. If the line is a comment (line 6), then it is marked for removal from the collection (line 7). Since the parallel semantics for `remove()` only marks the element for removal, the underlying collection might remain unchanged. Consequently, calling `size()` on the underlying collection (line 8) might not reflect the updated collection size (in a parallel environment this is not a problem since concurrent removes can occur anyway). The underlying collection might only be updated to reflect the removals when all threads have exited the loop (line 11).

Although it may be undesirable in some applications to serialise these removals in the underlying collection, this compromise provides a balance between the expected semantic and implementation challenge discussed above. This definition allows for a clean and consistent behaviour since the timing of removals are guaranteed to be performed before the loop completes.

### 3.1.4 Reductions

First, a short background is presented on the underlying concepts that the proposed object-oriented reduction concept has been developed upon: *reductions* and *thread-local storage*. First, the interface for the Java implementation is presented (which is applicable to most object-oriented languages) and finally an alternative interface for other languages, like C++, is presented.

#### Reductions

The Parallel Iterator described so far is sufficient for many, but not all, iterative computations. For programs that share variables, programmers must provide mutual exclusion to ensure correct results. In a threading library, this is typically solved using a mutex. Unfortunately, programs with fine-grained parallelism would suffer heavily in performance [6]. A reduction is a standard parallelisation problem [43] and is explained by the simple example in figure 3.4. The 3 threads calculate the sum of a list containing 9 elements, but each thread calculates only a partial sum for 3 elements. Rather than sharing a variable (the total sum) between all threads, each thread maintains its own copy of the variable to avoid excessive locking. However, this means each thread will have a partial result (`sumA`, `sumB` and `sumC` respectively) that need to be *reduced* (i.e. added) into a final result.

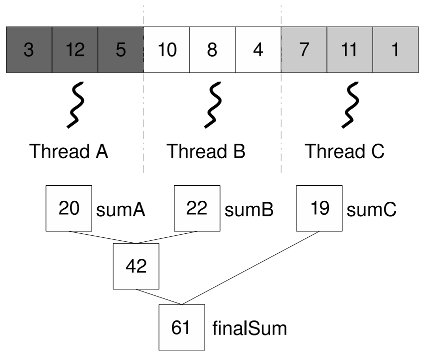


Figure 3.4: An example of a simple reduction. Three threads (each thread is a different shade) work in parallel to calculate a sum for their allocated elements. When all threads complete, the sub-results from each thread (`sumA`, `sumB` and `sumC` respectively) are reduced into a single result.

OpenMP provides a restricted solution with the `reduction` clause where only reductions such as `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`, *minimum* and *maximum* are supported. As well as these common reductions, the primary advantage of the solution proposed here is that it is object-oriented allowing any kind of reduction and using any data type. By integrating reductions with the Parallel Iterator, the scope of the Parallel Iterator is extended to handle even more situations, without programmers having to manually implement them. Below is the reduction solution that is applicable to most object-oriented languages.

### Thread-local storage

Thread-local storage is a common method used in parallel computing: although threads refer to the same global variable, they actually refer to different memory locations (hence the variable is local to each thread) [87]. Several languages have been extended to support thread-local storage, below is an example using Java's `ThreadLocal` class [94]:

```
ThreadLocal<Integer> localCount =
                   new ThreadLocal<Integer>(); // global variable
...
// increment localCount for the current thread
localCount.set( localCount.get() + 1 );
```

By creating `localCount` as a thread-local variable, each thread may read and write to `localCount` (using `get()` and `set()` respectively) with no need for locking. However, there is no way a thread may access all the thread-local values within the thread-local variable. This means that if a programmer wishes to reduce all the local values within the thread-local variable, then each thread must copy its local value to another (non thread-local) global variable (since only that thread is able to read the local value).

#### 3.1.4.1 Reductions with the `Reducible` object

By combining the two concepts of reductions and thread-local storage, an object-oriented solution to reductions is proposed. This is presented in the form of the `Reducible` object: it is especially attractive when used in conjunction with the Parallel Iterator. The `Reducible` object is used just like a typical thread-local variable, where threads access the local storage using `get()` and `set()`. In addition to these thread-local methods, there is a `reduce()` method that will perform a specified reduction across the thread-local values when threads have finished their work and return the final result.

The example below illustrates these components when trying to find the maximum of a list of unsigned integers. A Parallel Iterator is created in order to manage the parallel traversal of the list. A `Reducible` object of type `Integer` is created to act as the thread-local maximum variable. In this example, each thread updates its thread-local maximum on every iteration. After all threads have finished, the `reduce()` operation is invoked on the `Reducible` object specifying the maximum reduction. To implement this example in OpenMP would require the list to be copied to an array first. Furthermore, this simple example is only used to convey the concept (which may be applied to more complex situations not handled by OpenMP).

```
// initialise Parallel Iterator and Reducible
List list = ...;  // get list of numbers
ParIterator<Integer> pi = ParIterator.createParIterator(list);
Reducible<Integer> localMax = new Reducible<Integer>(0);

// each thread does this
while (pi.hasNext()) {
  int v = pi.next();
  if (v > localMax.get())
```

```
        localMax.set(v);
    }

    // final code, executed by any thread
    int finalMax = localMax.reduce(Reduction.IntegerMAX);
```

Using the `Reducible` object is just like a `ThreadLocal` object, with the addition that the local values may be reduced by any thread. The current implementation provides a number of type-specific reductions, the example above using `Reduction.IntegerMAX` shows the `MAX` reduction on the `Integer` class. This is necessary since Java does not support operator overloading (section 3.1.4.3 discusses an alternative for languages supporting operator overloading).

Reductions can generally be performed sequentially or in parallel (using a tree-network [43]). From the user's point of view, invoking a reduction in parallel is straightforward: the `reduce()` method has an optional boolean parameter to denote if the reduction is to be performed in parallel (the default is `false`):

```
    // reduction executed in parallel
    int finalMax = localMax.reduce(Reduction.IntegerMAX, true);
```

A reduction is only calculated once, therefore subsequent calls to `reduce()` result in the pre-calculated value being returned. Finally, when the user instantiates a `Reducible` object, they have the option of specifying a default value (which is assigned to each of the thread's private copy, just like OpenMP's `firstprivate`). If no default value has been specified, then a runtime exception is thrown when the user attempts to:

- `get()` the local value for a thread that has not `set()` a value, or
- Call `reduce()` when no threads have `set()` any value.

#### 3.1.4.2 User-defined reductions

Providing only a few common reductions may be insufficient since there may be specific reductions that a programmer requires. The user is therefore allowed to define any custom reduction, which is simply used in place of the supplied reductions. This is achieved by the user providing an object that implements the `Reduction` interface. Only one method needs to be implemented, defining the reduction of two elements into one. For example, the following defines a possible reduction on `Color` objects (section 3.1.4.3 presents an easier alternative to create user-defined reductions for languages that support function pointers):

```
Reduction<Color> colorReduction = new Reduction<Color>() {
  public Color reduction(Color first, Color second) {
    // User code defining reduction logic
  }
};
```

A reduction may then be performed as usual, only this time specifying the user-defined reduction:

```
Color finalColor = localColor.reduce(colorReduction);
```

This approach allows the programmer to easily define complex reductions that could even involve entire data structures, for example concatenating lists or maps. The reduction must be *associative* (the order of performing the reduction makes no difference) and *commutative* (the order of the thread-local values makes no difference) since the interface does not specify order. In the case the reduction is executed sequentially (for example, as in figure 3.4), the user-defined reduction does not need to be coded thread-safe.

#### 3.1.4.3 Alternative interface

Languages such as C++ allow a more flexible interface for the proposed object-oriented reduction concept due to the supported language features. Of particular interest are features such as pointers, operator overloading and function pointers. Again, just as before, the `Reducible` object is used essentially like a thread-local. However, rather than a thread modifying its thread-local value through the `set()` method, the thread-local value is directly modified through a pointer (which is retrieved using the `getMyCopy()` method). Consequently, the parallel code looks even closer to the sequential equivalent since thread-local values are not accessed through methods. The same example is shown below, this time using the alternative interface in C++ syntax:

```
List list = ...;  // get list of numbers
ParIterator<int> *pi = Factory::createParIterator<int>(list);
Reducible<int> localMax(0);
```

```
#pragma omp parallel
{
  int v;
  int* myMax = localMax.getMyCopy();
  while(pi->next(v)) {
    if (v > *myMax)
      *myMax = v;
  }
}
max = localMax.reduce(Reduction::MAX);
```

Each thread gets a pointer to its thread-local value through the `getMyCopy()` method. Finally, when all threads have finished their work, the `reduce()` method is invoked to perform the specified reduction on all thread-local values and return the final result. Note that the example `Reduction::MAX` used with the `reduce()` method is not type-specific (unlike the Java implementation above) since C++ supports operator overloading. Therefore, the supplied reductions work for any data type, as long as the data type supports the respective operation. As an example, if `MAX` is to be used to reduce a list of `MyClass` objects, then the `MyClass` class needs to have the $>$ operator defined. Also, user-defined types need to have a copy constructor defined so that the values inside `Reducible` may be initialised correctly for each thread.

The advantage of this approach compared to related work (discussed later in section 3.2) is that only minor modifications are required for the sequential code. In fact, the structure of the code remains unchanged when used in combination with OpenMP. In the case that OpenMP is not used, this concept becomes especially useful since the programmer does not have to manually implement the reduction.

Figure 3.5 shows a visualisation of the above code example. The Parallel Iterator (`ParIterator *pi`) is used on a list (`List list`) of 8 numbers and is traversed by two threads ($A$ and $B$) using a static scheduling policy with chunk size of 2. The shaded numbers in `list` correspond to iterations for thread $A$, while the white numbers correspond to iterations for thread $B$. The `Reducible` object stores private copies of the maximum for each thread and is updated as each thread progresses.

Figure 3.5 shows the state of the reducible object after 2 elements have been processed by each thread (so far thread $A$ got 25 and 9, while thread $B$ got 19

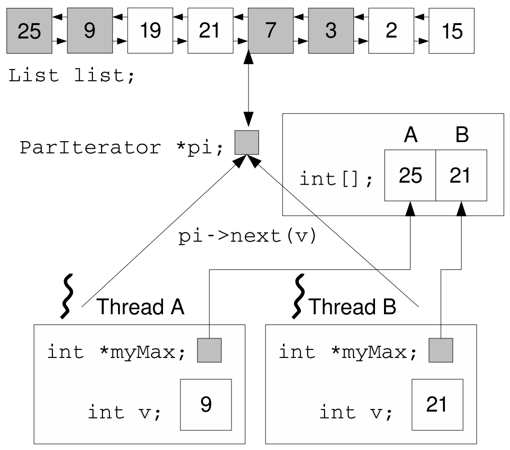


Figure 3.5: Visualisation of the `Reducible` object (using the alternative interface), situation shown after 2 elements have been processed by each thread. Thread *A* has been allocated the gray elements, while thread *B* has been allocated the white elements from list.

and 21). The most recent element assigned to each thread is stored in variable `v` (9 and 21 respectively for threads *A* and *B*). The maximum element traversed by each thread is stored in `myMax` (25 and 21 respectively). When all iterations have been completed, the values stored in the reducible object will be reduced into one final value (achieved through the `reduce()` method).

For languages such as C++ that support function pointers, an even simpler solution is provided to create user-defined reductions. Rather than creating an instance of an interface, the programmer only needs to define a method reducing two elements into one:

```
Color combineRGB( Color a, Color b ) {
  // User code defining reduction logic
}
```

A reduction may then be performed as usual, only this time specifying the user-defined method:

```
Color finalColor = globalColor.reduce(combineRGB);
```

### 3.1.5 Break semantics

An important concept in iterative computation is the `break` statement. In a sequential loop, this statement stops the execution of any more iterations in the loop. But in a loop being executed by multiple threads, does the `break` statement mean to cancel only in the local thread or across all threads? The decision should be left to the developer, since there are legitimate cases for both approaches as discussed below. The simple, but powerful, concept below has been implemented for both the C++ and Java versions.

The programmer needs to be aware of an important aspect before breaking from a parallel environment [86]. In the sequential version of a loop, the `break` occurs at a particular iteration $x$ after completing a subset S of the entire iteration space. In the parallel traversal of the loop, iterations are partitioned between the different processors according to a particular scheduling policy. If the parallel loop terminates also at iteration $x$, the completed iterations are not necessarily the same as those completed during the sequential version (subset S). However, for a large and common class of problems this does not matter (this condition is in most cases the same as the condition that the iterations are independent). A prominent example is searching in a data structure until a certain object has been found.

#### 3.1.5.1 Global `break`

The most likely type of break to be performed in a parallel loop is that of a global break: this is when the programmer wishes to cancel loop iteration across all threads. For example, an item has been found in a parallel search or the user pressed the cancel button. In such situations, all threads should stop their iterations. This is achieved by one thread invoking `globalBreak()` on the Parallel Iterator. All threads will then receive a `false` the next time they call `hasNext()` and they all return synchronised from this last `hasNext()` call. The advantage of this approach is that each thread breaks out of its loop in a controlled manner at an iteration boundary.

```
boolean itemFound = false;
while ( pi.hasNext() ) {
  itemFound = searchDocument( pi.next(), searchQuery );
  if ( itemFound )
    pi.globalBreak();
  }
```

In this example, multiple threads use a Parallel Iterator to traverse a collection of documents. If one of the threads finds the search query in one of the documents, then all threads are requested to stop using the `globalBreak()`.

Without the break semantics the iterator could not be used for search, or only in a naive implementation by always covering all iterations even after the target has been found. The benefit can be analysed statistically: if the 'solution' is at a random position in the iteration space, a single processor on average needs to do $N/2$ iterations. For $P$ processors this becomes $N/2P$ iterations, as each processor gets $N/P$ iterations (balanced load). Without the global break, the entire iteration space always needs to be explored, hence resulting in $N/P$ iterations per processor. This difference is a factor of 2 in average speedup.

#### 3.1.5.2 Local `break`

An alternative type of parallel loop breaking is that of a local break: only the current thread wishes to cancel, while the other iterations should still be processed. An example includes a program checking whether too many threads are being used and decides the current thread should stop iterating (to reduce disk contention, for instance). In this case the thread should execute a `break` with local semantics so that iterations for the other threads are not affected. This is achieved by invoking a `localBreak()` to allow the Parallel Iterator to clean up, such as releasing any elements previously allocated to the breaking thread. Even though a thread successfully calls `localBreak()`, the `hasNext()` remains a synchronisation barrier (section 3.1.1) to ensure that no thread (including the breaking thread) prematurely progresses past the loop:

```
  while ( pi.hasNext() ) {
    resize( pi.next() );
    if ( tooManyThreads ) {
      boolean broke = pi.localBreak();
      ...
    }
  }
```

An important aspect of the local break is that it guarantees unprocessed elements (originally assigned to the breaking thread) to eventually be processed; this is what distinguishes a global break from a local break. Even if all threads called `localBreak()`, the Parallel Iterator will guarantee that at least one thread remains to traverse any unprocessed elements. Therefore, a boolean is returned from the `localBreak()` to denote whether the current thread was successfully excused from traversing the loop.

The next question therefore is, how are the unprocessed elements distributed amongst the remaining threads? If a thread calls `localBreak()`, then all it's previously allocated elements are released. If there are still other threads traversing the Parallel Iterator, then they share these elements (after those threads complete their normal iterations) using a dynamic schedule with chunk size 1 (regardless of the original schedule). In such a successful call to `localBreak()`, `true` is returned since that thread has successfully broken locally.

If the thread calls `localBreak()`, but all the other threads have completed (and are waiting at the barrier), then that thread will end up executing the elements itself (therefore the `localBreak()` has no effect and returns `false`). If multiple threads attempt to call `localBreak()`, then all of them will succeed except the last thread. Therefore, the last thread does all the remaining elements, ensuring that all elements are processed; a `localBreak()` implies that the user wants the elements to still be processed, otherwise a `globalBreak()` should be used.

### 3.1.6 Exception handling

An exception is an event that diverts a program from its normal execution flow [54]. Many programming languages support exceptions to separate error-code from the actual user code and generalise error handling. This potentially produces more readable and efficient code since error handling is not integrated within the normal execution flow. Exceptions are important in object-oriented languages, hence this section will discuss the relation of the Parallel Iterator and exceptions.

The duties of the Parallel Iterator are solely to dispense elements amongst threads. As such, the Parallel Iterator must be aware of what the threads are doing. Consequently, the Parallel Iterator itself cannot "catch" exceptions. However, when used in combination with standard exception handling mechanisms, the Parallel Iterator provides a convenient interface to manage exceptions in a meaningful way in a parallel environment. Consider the following loop being executed by multiple threads using a Parallel Iterator:

```
while ( pi.hasNext() ) {
        ...
        // Exception thrown
        ...
    }
```

In the Java threading model, for example, an exception encountered within this loop will be propagated up the call stack. If the exception is unhandled, the thread will terminate: all the other threads continue executing and are completely oblivious to the thread's termination. Even if the exception is eventually handled (but not handled within scope of the Parallel Iterator), then that thread has still abruptly exited traversal of the parallel loop without the Parallel Iterator's awareness. All the other threads will forever wait at the barrier since the (potentially terminated) thread still has not come back to call `hasNext`.

To overcome this, exceptions must be caught (either using a try/catch or try/finally) within the Parallel Iterator's scope so that the Parallel Iterator may be acted upon; otherwise, the Parallel Iterator remains unaware of the thread's state and continues forcing other threads to wait (this idiom of catching potential exceptions is equivalent to that of Java `Lock`s [94], where any potential exceptions must be caught in order to release the `Lock`). When a thread encounters an exception, the following behaviours are possible:

- *Do nothing:*
  The thread catches an exception but decides to ignore it, so it continues to call `hasNext()` and `next()` as usual

- *Stop locally:*
  The thread catches an exception and decides only it should stop iterating, so it calls `localBreak()`

- *Stop globally:*
  The thread catches an exception and decides all iterations should stop, so it calls `globalBreak()`

This shows that the `localBreak()` and `globalBreak()` of section 3.1.5 are crucial in meaningful exception handling.

### The `register` helper method

It is difficult to define what should happen when an exception is encountered in a parallel loop [109]. In a sequential loop, an exception may be encountered by any iteration: the exception is propagated up to the nearest handler and it is assumed this was the only exception to have occurred. However, in a parallel loop it is possible that multiple concurrent iterations encounter an exception (for example, `FileNotFoundException`). In such a case, the programmer may want to know about all the exceptions that occurred. Due to the crucial role of exceptions in Java [70], the following has been implemented for the Java version of the Parallel Iterator.

The `hasNext()` method acts as a boundary between the iterations: a new iteration $i$ is said to begin when `hasNext()` is invoked, regardless of whether the element has been accessed yet or not. If the element has not been accessed yet (using `next()`), then we are in the pre-`next()` region (this implies that the thread is intending to invoke `next()` to access the element). If the element has been accessed, but still not finished the iteration, then we are in the post-`next()` region. The iteration of $i$ is considered complete only when the thread invokes `hasNext()` again, where the next element becomes the current element being traversed by this thread.

In order to handle exceptions in parallel, the programmer would need to manually implement logic to record the exception, the iteration in which it occurred, the thread that encountered it and then notify the other threads. Managing all this is difficult for the programmer, especially in determining the current iteration if the exception occurs in the pre-`next()` region. The Parallel Iterator provides a helper method to conveniently record this information. The programmer is only required to catch exceptions (standard procedure as if using a sequential iterator). When an exception is caught, `register(Exception)` is invoked which will record the following information (storing them within a `ParIteratorException`):

- The `Exception` encountered
- The `Thread` that encountered the exception (for potential debugging purposes)
- The iteration in which the exception occurred (determined using `hasNext()` as iteration boundary)

What is the expected behaviour when a thread encounters an exception? Is this exception bad enough that all iterations should stop, or should it only be noted down while processing of the other iterations continue? The Parallel Iterator allows the programmer freedom in deciding how to treat exceptions. Consider the following example of processing a collection of files, where a number of exceptions could arise. For example, some files cannot be processed since they are not found (consequently causing a `FileNotFoundException`), whereas other exceptions require that the entire iteration space be canceled:

```
while ( pi.hasNext() ) {
  try {
    ...
    // Exception thrown
    ...
  } catch(FileNotFoundException fe) {
    // no thread will stop, just record exception
    pi.register(fe);
  } catch (TooManyThreadsException te) {
    pi.register(te);   // disk contention, reduce thread count
    pi.localBreak();   // only the current thread will stop
  } catch (DiskFullException de) {
    pi.register(de);   // full disk, might need to clean up
    pi.globalBreak();  // all other iterations should stop
  }
}
```

This example shows, depending on the exception encountered, the different possible behaviours. In some cases, the programmer may only wish to record the exception and then continue processing the other elements; in other cases, the programmer may wish to either cancel the current thread or all threads. Even if the programmer does not wish to use the `register()` method provided, they must at least catch the exceptions so that the thread does not terminate before it has a chance to inform the Parallel Iterator (otherwise the other threads will continue waiting for it to complete). This requirement is equivalent to that demanded by Java locks, where any potential exceptions must be caught in order to release a locked `Lock` [94].

Finally, when the Parallel Iterator has completed traversing the collection, all exceptions that occurred during the parallel traversal may be accessed using `getExceptions()` (the programmer may then use Java standard methods such as `printStackTrace()` on the exception object to determine the exact location at which the exception occurred):

```
ParIteratorException[] piExceptions = pi.getExceptions();
for (int i = 0; i < piExceptions.length; i++) {
  ParIteratorException pie = piExceptions[i];

  // object for iteration in which exception was encountered
  Object iteration = pie.getIteration();

  // thread executing that iteration
  Thread thread = pie.getThread();

  // actual exception thrown
  Exception e = pie.getException();
  ...
  // print exact location of exception
  e.printStackTrace();
  ...
}
```

### 3.1.7 Iterating over multiple collections

So far, the various features of the Parallel Iterator have been explored in the form of simple examples. Now, an example that aims to help envision how the Parallel Iterator is used in more complex applications is presented. Consider an application where the user wishes to ensure that the contents of two directories are identical. If the filenames in both directories do not match, then the comparison is immediately discontinued. Otherwise, the contents of each file are compared and any differences stored to a map.

In this implementation, the files for a particular directory are contained within a sorted set. The elements (i.e. the files) within the sorted sets have a one-to-one mapping (the $i^{th}$ element in the first collection relates to the $i^{th}$ element of the second collection):

```
SortedSet<File> dirOne = ...
SortedSet<File> dirTwo = ...
```

When these collections are processed in parallel, the correctly paired elements from each collection must be used. For example, when thread $A$ processes the

10th element from the first directory, then it should also be assigned the 10th element from the second directory (i.e. comparing the same file from both directories). Naturally, the Parallel Iterator encapsulates this co-ordination of elements between the different collections to ensure a correct distribution:

```
 1:  ParIterator<File> pi_one =
             ParIterator.createParIterator(dirOne, PI.Schedule.STATIC);
 2:  ParIterator<File> pi_two =
             ParIterator.createParIterator(dirTwo, PI.Schedule.STATIC);
 3:  Reducible<Map<String,List>> diffs = new Reducible();

 4:  // each thread does this
 5:  Map<String,List> mydiffs = new HashMap<String,List>();
 6:  diffs.set(mydiffs);
 7:  while (pi_one.hasNext() && pi_two.hasNext()) {
 8:    File f1 = pi_one.next();
 9:    File f2 = pi_two.next();
10:    if (!f1.getName().equals(f2.getName())) {
11:      // files in directory do not match
12:      pi_one.globalBreak();
13:      pi_two.globalBreak();
14:      directoriesMatch.set(false);
15:    } else {
16:      List contentDiff = getDifferences(f1,f2);
17:      if (contentDiff != null) {
18:        // contents in files do not match
19:        mydiffs.put(f1.getName(), contentDiff);
20:      }
21:    }
22:  }
     ...
23:  Map<String,List> allDiffs = diffs.reduce(myMapReduction);
```

Lines 1 and 2 create a Parallel Iterator for each respective directory, while line 3 creates the `Reducible` to store the differences between file contents. Beginning on line 5, each thread creates a private map to store the differences, and this reference is stored to the `Reducible` (line 6). Because of this `Reducible` object, the programmer does not need to make use of synchronised collections in this example.

Each thread keeps traversing both Parallel Iterators (line 7) and accesses the files from each respective directory (lines 8 and 9). If the names of the two files do not match, then the directories do not contain identical files (the collections are sorted). In such a case, a `globalBreak()` is called (lines 12 and 13) to stop the other threads iterating both collections. Otherwise, if the files have the same name, then their contents are compared. A helper method, `getDifferences()`, returns any differences between the files (line 16). These differences, if any, are stored in the thread's private map for the respective file (line 19). Finally, a user-defined reduction is performed on line 23 to collate all the differences.

Clearly, such an application is only semantically correct if the programmer can determine the underlying schedule that is used. With the Parallel Iterator's current implementation, a static schedule immediately allows for such an application: in a dynamic or guided schedule, the chunks might not line up to the same thread if the timing is off. Future versions of the Parallel Iterator will investigate allowing multiple collections to be traversed using dynamic and guided scheduling; this requires that the Parallel Iterators have knowledge of each other (in order to maintain the one-to-one mapping and avoid mis-aligned chunks of the different collections).

Without the Parallel Iterator, programmers might feel discouraged in attempting a parallel solution and may therefore resort to a sequential solution. It would be considerably more work to manually implement this application using, for example, concurrent collections or locking primitives. The Parallel Iterator, however, elegantly maintains the mapped distribution of elements.

### 3.1.8 Tree Parallel Iterator

The Parallel Iterator is a powerful way to parallelise code. One of the conditions of using it efficiently and elegantly is that there are no dependences between the iterations (otherwise manual dependence control must be implemented by the user). There are some collections, however, whose processing requires a certain partial order. A very important and widely used class of such collections are tree structures. When traversed, it might for example be important that an element (node) is processed before its child node (i.e. top-down traversal), or the other way round (i.e. bottom-up). Even with such a restriction, a lot of iterations can be processed in parallel (for example all nodes on the same level are independent of each other, and in particular all leafs).

When processing such a collection in parallel, one would like to separate the business logic from the code responsible for the correct parallel execution, including scheduling and precedence order enforcement. This thesis proposes a generic and flexible Tree Parallel Iterator that does exactly that. It implements the same simple interface as the regular Parallel Iterator. As such, the Tree Parallel Iterator stands in the tradition of patterns or skeletons for parallel processing [36]. It provides an encapsulated, tested and optimised form of parallel processing. Apart from its direct contribution, it illustrates the powerfulness of the Parallel Iterator concept in conjunction with sophisticated forms of parallel processing. Many other parallel processing patterns can be elegantly implemented using this concept.

In order to traverse a tree, one needs a tree collection. Since Java does not have a tree collection as part of the Collections framework, a `Node` interface is defined to represent the *nodes* in a tree. All nodes in a tree, except for the *root* node, have a *parent* node. A node may have *children* nodes, otherwise the node is a *leaf* node. Finally, a node has a *name* and stores a *value*. Below is the definition of a simple `Node` interface:

```
public interface Node<E> {

    public void setValue(E value);
    public E getValue();
    public Node<E> getParent();
    public List<Node<E>> getChildren();
    public boolean addChild(Node<E> child);

}
```

By storing data within this interface, users may take advantage of the Tree Parallel Iterator. In addition to this generic Tree Parallel Iterator, a Document Object Model (DOM) [94] Tree Parallel Iterator is defined to traverse a DOM `Document`. Such a `Document` represents an entire HTML or XML document, and will be used in the following example. Figure 3.6 shows the relation of these classes from the user's point of view: the Tree Parallel Iterator is a generic extension of the Parallel Iterator.

The discussion throughout this section is generic for the Tree Parallel Iterator (i.e. it applies to both `NodeParIterator` and `DOMParIterator`, and any potential sub-classes of `TreeParIterator`). The concrete example application involves traversing a Scalable Vector Graphics (SVG) file [37]. The SVG file

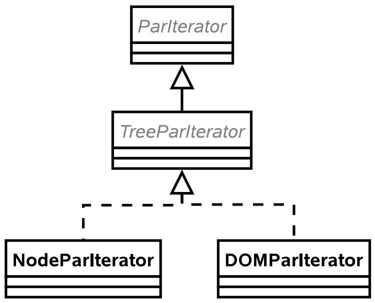
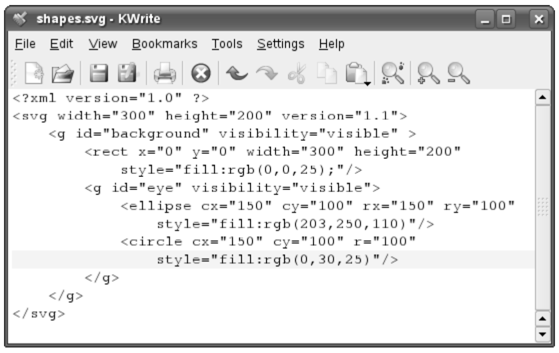
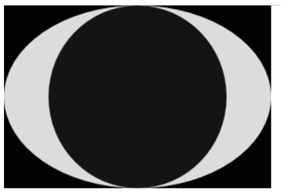


Figure 3.6: The Tree Parallel Iterator (`TreeParIterator`) is an interface extension of the Parallel Iterator (`ParIterator`). Currently two implementations of the Tree Parallel Iterator have been developed: one to allow traversal of DOM Documents (`DomParIterator`), and one to allow traversal of an arbitrary collection implementing the `Node` interface (`NodeParIterator`).



```
<?xml version="1.0" ?>
<svg width="300" height="200" version="1.1">
    <g id="background" visibility="visible" >
        <rect x="0" y="0" width="300" height="200"
            style="fill:rgb(0,0,25);"/>
        <g id="eye" visibility="visible">
            <ellipse cx="150" cy="100" rx="150" ry="100"
                style="fill:rgb(203,250,110)"/>
            <circle cx="150" cy="100" r="100"
                style="fill:rgb(0,30,25)"/>
        </g>
    </g>
</svg>
```

(a) Contents of an SVG file, in XML format



(b) Graphical outcome

Figure 3.7: Example of an SVG file, where vector-based graphics are defined in XML format. This example defines 3 shapes.

is essentially an XML document that defines vector-based graphics. First, the general scheduling of the Tree Parallel Iterator is explained. Second, the SVG example is further explored to illustrate the Parallel Iterator's ability to be used elegantly for more complex collections. Figure 3.7(a) shows an example SVG file that defines 3 shapes and the respective output in figure 3.7(b).

**Partial ordering in the Tree Parallel Iterator**

The Tree Parallel Iterator implements a well-established work-stealing schedule [17]. Here, the schedule in terms of the partial order semantics from the user's point of view is discussed; the implementation is discussed in more depth

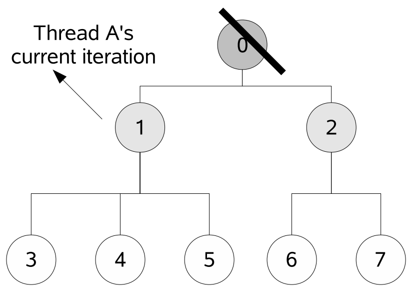


Figure 3.8: Partial ordering with the Tree Parallel Iterator: when a thread is assigned a node (e.g. node 1), this guarantees the parent (node 0) has already completed and the children (nodes 3, 4, 5) will not be scheduled until it completes.

in section 4.1.7. As will be shown in this section, the user code to traverse a more complex collection (in this case a tree) is no more complicated than the code to traverse a simple collection (such as a list); this is essentially the objective behind iterators! This shows how the Parallel Iterator concept may be extended to incorporate other schedules to traverse other collection types.

Figure 3.8 shows a simple example of a tree, consisting of 8 nodes. The partial ordering requires that a node (i.e. an iteration in the Parallel Iterator) only be executed when its parent has completed. Therefore, initially only the root node may be processed. When the root (node 0) has completed, then nodes 1 and 2 may be scheduled; nodes 3 to 7 remain unscheduled until their respective parent node has completed. This partial ordering retains the structure of the tree; without such a policy, then the nodes in the tree are no different than elements in a list (in which case the standard Parallel Iterator may be used). In addition to maintaining the tree structure, this policy allows us to achieve the following:

- A node may safely inherit computed values from its parent node (since the parent has completed),
- A node $n$ (and subsequently the sub-tree rooted at $n$) may safely be removed during parallel traversal (i.e. extending the `remove()` method from the iterator interface),
- Similarly to `remove()`, a `replace()` method may also be defined in parallel semantics (illustrated below).

Note that other scheduling policies are possible for trees (for example performing a bottom-up traversal rather than top-down), all without changing the user-code (only the logic inside the Parallel Iterator needs to be implemented). Since the objective is to illustrate the elegance of the Parallel Iterator concept from the user's point of view, other scheduling schemes are not elaborated.

### SVG shape recognition example

The example application is a parser that reads an input SVG file containing various shapes defined as generic *path* elements[1]. A path is essentially a finite series of *x,y points*, meaning that a circle might be defined by multiple points around its circumference (rather than defining it using the center point and radius). Not only does this reduce the graphics resolution (i.e. it is no longer truly scalable as SVG should be), but it also increases the SVG file size; for example, a circle defined as a *path* element consumes around 750 bytes, while the circle defined using the *circle* element only consumes around 70 bytes. Therefore, the program transforms the input SVG file into a true SVG file, by replacing paths with the correct shapes and their parameters.

In this example, DOM `Document` represents the tree being traversed:

```
DOMParser parser = new DOMParser();
parser.parse(new File(“shapes.svg”));
Document doc = parser.getDocument();
```

Once the `Document` has been constructed, it is passed to the Tree Parallel Iterator as follows:

```
DomParIterator pi = TreeParIterator.createParIterator(doc, numThreads);
```

and a number of `Reducible` objects (section 3.1.4.1) are created to calculate the total number of each shape and the biggest shape:

```
Reducible<Integer> numCircles = new Reducible<Integer>(0);
Reducible<Integer> numRectangles = new Reducible<Integer>(0);
Reducible<Shape> biggestShape = new Reducible<Shape>(null);
```

[1]This application is motivated by the SVG export feature of the OpenOffice.org application suite, which defines all shapes as generic *path* elements.

By specifying the `Document` (i.e. the tree) and the number of threads, a Parallel Iterator has just been constructed: it may be used as seen throughout this chapter. The application is presented in full below, and then the various parts are explained:

```
// each thread does this
     ...
     // private variables for each thread
 1:  int myCircles = 0;
 2:  int myRectangles = 0;
 3:  Shape myBiggestShape = null;
 5:  ...
     // start loop
 6:  while (pi.hasNext()) {
 7:    Node n = pi.next();    // DOM Node
 8:    ...
 9:    Element e = (Element) n;
10:    if (e.getNodeName().equals(‘‘g’’)) {
         // element is a group, delete if attribute set to invisible
11:      String visibility = e.getAttribute(‘‘visibility’’);
12:      if (visibility.equals(‘‘invisible’’))
13:        pi.remove();
14:    } else if  (e.getNodeName.equals(‘‘path’’)) {
         // element is a path of points, parse into a shape
15:      Shape shape = ShapeGuessor.guess(e);
         // update myBiggestShape
16:      if (myBiggestShape == null) {
           // first shape for this thread
17:        myBiggestShape = shape;
18:      } else if (shape.getArea() > myBiggestShape.getArea()) {
           // new shape is bigger than the current biggest shape
19:        myBiggestShape = shape;
20:      }
21:      switch (shape.getType()) {
         // switch on type of shape
22:      case CIRCLE:
           // shape is a circle, record the circle's attributes
23:        numCircles++;
24:        Circle circle = (Circle) shape;
25:        int radius = circle.getRadius();
26:        int centerX = circle.getCenterX();
27:        int centerY = circle.getCenterY();
           // create a new element to represent the circle
28:        Element newElement = createElement(“circle”);
           // set the attributes to the new circle element
29:        newElement.setAttribute(r, String.valueOf(radius));
30:        ...
           // replace the old path element with the new circle element
31:        pi.replace(newElement);
32:      case RECTANGLE:
33:        ...
34:      default:
35:        // unknown shape, keep as path
36:      }
38:    }
39:  } // end loop
     // update Reducible values with the private values
40:  numCircles.set(myCircles);
41:  numRectangles.set(myRectangles);
42:  biggestShape.set(myBiggestShape);
```

The highlight of the Tree Parallel Iterator is that the user code to traverse the nodes essentially looks identical to the standard code to traverse any of the other collections. This is the whole idea behind iterators: to hide the collection implementation. The Tree Parallel Iterator, just like the Parallel Iterator, remains faithful to this idea.

The first point of interest in the above example is that of the `remove()` method on line 13. This behaves consistently with the idea discussed in section 3.1.3.4: the last node returned by the Parallel Iterator is to be removed from the underlying collection. However, the semantics of a `remove()` on a tree varies slightly compared to a flat collection (such as a list). In a tree, removing a node implies that the children nodes are also to be removed. Therefore, calling `remove()` ensures that the children nodes will not be scheduled for traversal (since they were removed when their parent node was removed). Achieving this is possible, even in a parallel traversal, due to the semantics discussed in figure 3.8 since the children nodes have not been scheduled.

The other point of interest is that of line 31. Similar to the semantics of `remove()`, the Tree Parallel Iterator also supports a `replace()` method that allows a new element to be put in place of the last element returned by the Tree Parallel Iterator (this functionality is essentially a thread-safe wrapper around the `replaceChild()` method provided by the DOM `Node` interface). In this example, path elements are replaced with the respective shape element. Again, this is fine in a parallel traversal since the parent has already completed and the children not started yet.

When each thread completes the loop, it updates the `Reducible` object to denote the respective sub-results it has computed (lines 40-42). Finally, the main thread will execute the reduction across the respective `Reducible` values (notice the use of a customised reduction for the `Shape` object):

```
int totalNumCircles = numCircles.reduce(Reduction.IntegerSUM);
int totalNumRectangles = numRectangles.reduce(Reduction.IntegerSUM);
Shape finalBiggestShape = biggestShape.reduce(new Reduction<Shape> () {
    public Shape reduce(Shape first, Shape second) {
      if (first.getArea() > second.getArea())
        return first;
      else
        return second;
    }
});
```

This example did not make use of features such as parallel break semantics (section 3.1.5) or exception handling (section 3.1.6) in order to maintain the example size; however, these features may still be used with the Tree Parallel Iterator. In section 4.2.4, performance benchmarks of the Tree Parallel Iterator are presented (both for this particular SVG example, as well as another computationally intensive application).

## 3.2 Related work

The importance of loop parallelisation and loop scheduling have been extensively studied before [3, 26, 61]. The work presented here is distinct since it promotes preserving the qualities of object-oriented sequential code while still providing flexibility. It applies standard parallel concepts (such as scheduling policies and reductions) in a way object-oriented programmers are familiar with, namely using iterators and without code restructuring. The semantics for local and global breaks have also been integrated with the Parallel Iterator. In other tools, if the programmer wants the behaviour of the global break then they must manually implement this.

Over 100 proposals for concurrency in object-oriented languages were surveyed in [85]; the most influential and relevant ones are discussed below in addition to others. At first glance, it may seem that many previous attempts have proposed a solution to the problem of parallel iteration in object-oriented languages. However, the Parallel Iterator presented here differentiates in important aspects.

Some approaches such as DatTel [13], Parallel Standard Template Library (PSTL) [64] and Standard Template Adaptive Parallel Library (STAPL) [5] aim to provide parallel extensions to the Standard Template Library (STL) [92] in C++. DatTel is a data-parallel template library that overloads certain STL functions. Although the parallelism is hidden from the user, DatTel is not suited for mainstream object-oriented parallel computing. It is restricted to containers composed of simple data types and therefore does not support collections with user-defined types which is an essential aspect in object-oriented programming.

PSTL provides a `par_apply` algorithm; STAPL provides a `pforall` *parallel region manager*. Both of these are used similarly to the `parallel_for` in Threading Building Blocks (TBB) [63] which was recently released by Intel for parallel programs in C++. Unfortunately, STAPL, PSTL and TBB do not allow the user to directly traverse collections (examples of valid ranges to traverse include integers, STL random iterators and pointers). The main problem is the restructuring of the code and the creation of a new object. First, the programmer must create a function object to specify work to be done for a range of elements. Secondly, the context of the code changes since now the logic is in another class (the function object).

CC++ [30] provides a `parfor` statement (parallel semantics of the sequential `for` statement) to iterate over a collection where each iteration is executed in parallel. However, efficiency is lost when iterations contain low computation since the initialisation, update and test parts of the `parfor` loop remain sequential. In [8], a `for_each_par` template method was proposed based on the idea of *range partition adaptors* which converts a range into a collection of sub-ranges. However, only collections with random access iterators are supported and the user must create objects to partition the collection. Other attempts exist but have been superseded by OpenMP's `parallel for` [82]. Despite OpenMP's success, it is insufficient for many object-oriented collections as it can only be applied to an integer range and the collection needs to be accessible using the loop index. It therefore cannot be used directly to traverse many collections, such as linked-lists or sets.

The style and purpose of iterators in scientific-targeted languages such as Chapel [66] is substantially different from that of Java or C++. The purpose of iterators here is to traverse a collection; Chapel iterators on the other hand are essentially functions that return a sequence of values (as opposed to a function that only returns one value). Therefore, the iterator concept in Chapel is tightly coupled with the loop; however, in Java and C++, iterators are associated with a collection of elements [66]. The semantics and syntax of the Chapel iterator is very similar to that of the CLU iterator [71].

Modern parallel languages (such as Chapel, Fortress [4] and X10 [33]) tend to target large-scale scientific applications; therefore they focus on a distributed memory model. Since this thesis addresses desktop applications in light of mainstream multi-core processors, the focus is on a shared memory programming model. The different target applications is further highlighted by the code syntax of Fortress: it mimics that of mathematical notation as it is aimed for scientists.

The Microsoft Parallel Extensions [76] to the .NET Framework support parallel semantics of the foreach statement with the `Parallel.ForEach` static method. Since the parallelism is controlled by the Parallel Extensions, the programmer does not have any control over threads, scheduling policies and parallel break semantics. By catching an `AggregateException`, programmers may handle exceptions thrown from within the body of the `Parallel.ForEach` method. Although programmers can analyse the exceptions thrown, it is unknown in which iteration or thread the exceptions occurred.

Some approaches support aggregate operations, such as PLINQ [77] and ParallelArray [69], thereby removing the loop altogether. Consider for example ParallelArray. The collection to traverse in parallel is stored in a ParallelArray class:

```
ParallelArray<Image> images = ...
```

This allows various operations to be applied:

```
images.apply(resizeProcedure);
```

By removing the loop, this is essentially a change in the programming style; it provides a higher level and black-box style of programming. Whether this is an advantage or disadvantage is a matter of preference: those with database programming or functional programming experience may prefer this style [69]. The Parallel Iterator is not a new programming style: it is an extension to the sequential iterator using the same object-oriented principles. For example, sometimes the programmer may want more explicit control: the Parallel Iterator allows this with minimal difficulty, such as fine-tuning the number of threads, scheduling policy, or early termination (with local/global semantics).

**Reductions**

OpenMP provides a `reduction` clause but is limited to only a few predefined reductions; furthermore, aggregate types (such as arrays), pointer types and reference types may not appear in the reduction clause. TBB and QtConcurrent [104] support user-defined reductions. But since TBB requires reductions to be defined within a function object, this causes code restructuring and context change. Furthermore, multiple reductions cannot be defined within the same class definition.

More recently, QtConcurrent provides a `mappedReduced` method that is based on MapReduce [40], where the programmer specifies two methods. The first method (executing in parallel) defines the computation to be performed on individual elements from the collection (the *map*). The second method (executing sequentially) combines the intermediate results into one final result (the *reduce*). Just like TBB's reduction, this requires restructuring the loop code into a new method defining work for one element; every loop iteration now results in a separate method call. QtConcurrent's reduction method is executed as many times as there are elements. With the Parallel Iterator, reductions are only executed as many times as there are threads. This potentially improves performance since less time is spent in the reduction stage, as observed in the experimental results of section 4.2.3.2.

### Tree Parallel Iterator

The implementation of the work-stealing [28] schedule is based on the randomised work-stealing variant [17]. Processing an XML document in parallel has been explored using both static [83] and dynamic [72] partitioning schemes. However, the implementation of the Tree Parallel Iterator was merely to demonstrate how the simplicity of the Parallel Iterator concept is applied to hide the details of traversing complex data collections in parallel.

Chapter 4

# Parallel Iterator implementation and performance

This chapter discusses the Parallel Iterator's implementation and performance. Although the Java version is mainly focused upon, the C++(Qt) version is also addressed.

## 4.1 Implementation

The Parallel Iterator uses the sequential iterator's standard interface: `hasNext()` only returns a boolean while the `next()` method returns a reserved element, if any, for the calling thread. Although the calls of `hasNext()` and `next()` are interleaved, they need to appear atomic to the threads accessing the Parallel Iterator. The primary purpose of the Parallel Iterator's `hasNext()` method is to reserve elements. If `hasNext()` has reserved at least one element for the thread, then `true` is returned. Otherwise the thread waits at a barrier until all threads complete, i.e. all iterations have been executed, and then returns `false` to denote the traversal has completed.

### 4.1.1 Scheduling policies

The Parallel Iterator has different implementations depending on factors such as the collection type, scheduling policy and chunk size. Before discussing schedule-specific implementation details, an overview of the main concepts is presented. This is largely based around two categories of collections:

- *Random access collections* (section 4.1.1.1) provide index-based constant-time access to elements in the collection, such as array-lists or vectors.
- *Inherently sequential collections* (section 4.1.1.2) do not provide index-based constant-time access to the collection, such as linked-lists or sets. The Parallel Iterator therefore uses a sequential iterator to access elements.

Note that the copying, locking and other management discussed below are all implemented within the Parallel Iterator and hidden from the user.

#### 4.1.1.1 Random access collections

If the collection to traverse supports index-based access in (amortised) constant-time, then index computations are performed within the Parallel Iterator to keep track of the next iteration for each thread. Consequently, elements are accessed directly using the integer index. Each thread stores the following integers:

- `nextIndex` specifies the next index to iterate
- `chunkStop` specifies the index just outside the current chunk

Using these two indices, and $n$ to denote the collection size, the *index boundary test* (used below) is defined as follows:

```
if (nextIndex > n)
    // no more elements for this thread
else
    chunkStop = min(chunkStop,n) // “left-overs” for last chunk
```

Whenever `nextIndex` is less than `chunkStop`, this means at least one element (i.e. the element at `nextIndex`) remains for the thread. If so, this element is returned. Otherwise, depending on the scheduling policy, one of the following is performed to determine the next chunk of elements:

- *Static scheduling*:

1. Increment `nextIndex` by
   `chunksize`$\times$(`numThreads` $-1$)
   This moves `nextIndex` to the start of the next chunk
2. Increment `chunkStop` by
   `chunksize`$\times$`numThreads`
   This moves `chunkStop` to the end of the next chunk
3. Perform *index boundary test*

- *Dynamic or guided scheduling*:
  For dynamic and guided scheduling, `nextFree` is a shared integer index that represents the start of the next chunk to be allocated.

  1. Obtain exclusive access to `nextFree` by acquiring a lock
  2. Update `nextIndex` to `nextFree`
  3. (For *guided* scheduling) calculate new `chunksize`
  4. Increment `nextFree` by `chunksize`
  5. Perform *index boundary test*
  6. Update `chunkStop` to `nextFree`
  7. Release lock

Figure 4.1 shows an example where a list of 9 elements are being traversed using a chunk size of 4. Thread $A$ has iterations 0, 1, 2, and 3 while thread $B$ has iterations 4, 5, 6 and 7. If this is static scheduling, then thread $A$ also gets iteration 8. If this is dynamic scheduling, then the first thread to finish their chunk will get iteration 8. The elements shaded in the collection are those that have been accessed by their respective thread. In the next iteration, thread $A$ will execute 3 while thread $B$ executes 5.

#### 4.1.1.2 Inherently sequential collections: on-demand copying

In section 4.1.1.1, elements from the underlying collection were accessed using integer indexing. Unfortunately, this is inefficient for collections that do not support constant-time access to random elements. The working principle now is that the Parallel Iterator essentially acts as a thread-safe wrapper to the sequential iterator. Since the sequential iterator is traversed only once, this means the sequential iterator will not usually be pointing to the next correct element

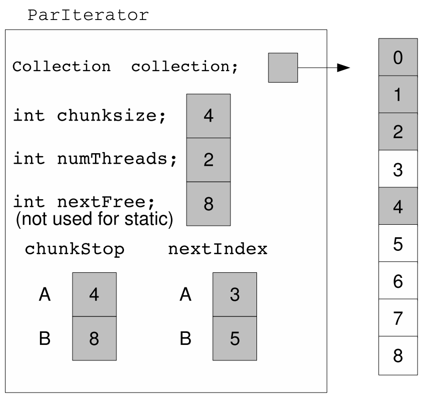


Figure 4.1: Implementation for random access collections. The Parallel Iterator maintains indexes representing the next element for each respective thread. Situation shown after the first 4 elements have been allocated to thread $A$ (3 of which have completed) while the next 4 elements have been allocated to thread $B$ (1 of which has completed).

for the current thread. In order to handle this, elements need to be copied (accessed using the sequential iterator) and reserved for the correct thread to access them. During this process, the sequential iterator is locked to avoid other threads interrupting.

The semantics of **copying** as used throughout this section must be clarified. Copying actually refers to *retaining a pointer* to the original object and storing this pointer in a buffer (essentially an array). It does not refer to making a deep clone or duplicate of the original object. In this way, minimal amounts of memory and time are used. The second concept used by the Parallel Iterator that extends copying is **on-demand copying**. Rather than copying *all* the elements of a collection, elements are copied into the buffer only as required. The advantages of on-demand copying is that copying is performed in parallel as other threads are executing iterations. Also, in the case the Parallel Iterator breaks (as in section 3.1.5), less time would have been wasted on unnecessary copying.

The approach differs depending on the scheduling policy of the Parallel Iterator (the implementation for guided scheduling is essentially identical to dynamic scheduling):

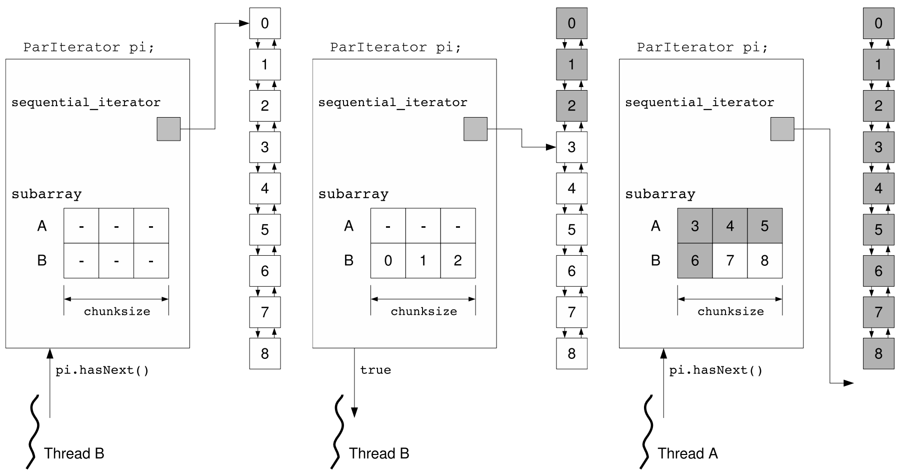
**Dynamic scheduling** When a thread first calls `hasNext`, it copies the next `chunksize` iterations: this requires exclusive access (by acquiring a lock) to the sequential iterator as the next `chunksize` elements are copied to the thread's private subarray. Locking is not needed for every call to `hasNext`, but is performed by each thread only once every `chunksize` calls to `hasNext`. Each thread keeps up to `chunksize` unprocessed iterations in its private subarray at any one time. The first thread to complete its private subarray will copy the next `chunksize` iterations. Subsequent `hasNext` calls will not require locking if there are unprocessed elements remaining in the thread's private subarray.

Figure 4.2 shows the internals of a Parallel Iterator used to traverse a linked-list (an inherently sequential collection) using a dynamic scheduling policy with chunk size of 3. In figure 4.2(a), thread *B* calls `hasNext()` for the first time. Since none of the iterations have been requested yet, 3 elements are copied over from the sequential iterator to the private buffer for thread *B*, and `true` is returned to thread *B* as shown in figure 4.2(b). Thread *B* now has 3 iterations reserved for it. Further calls to `hasNext()` by thread *B* do not affect the state of the iterator since there are still 3 elements to be processed, and these are only accessible when thread *B* calls `next()`.

Copying is necessary when accessing inherently sequential collections such as a linked-list since accessing elements from the collection takes time proportional to the collection size. Only during such copying (inside a `hasNext()` invocation) will locking of the sequential iterator occur. No locking occurs when a thread calling `hasNext()` already has unprocessed elements in the private buffer. No locking occurs inside `next()` also, since this method only accesses reserved elements from the private buffer.

Figure 4.2(c) shows the state of the Parallel Iterator when all elements have been assigned to threads (due to `hasNext()`), but not necessarily processed yet (since some are yet to be accessed using `next()`). Thread *A* has completed iterations 3 to 5, while thread *B* is working on iteration 6 but has already reserved iterations 7 and 8 as well (due to the scheduling chunk size). Therefore thread *A* invokes `hasNext()` to see if any more elements remain, but `false` will be returned.

**Static scheduling** Implementation of static scheduling is slightly more involved compared to the other scheduling policies. Just as copying was performed for dynamic scheduling above, it is necessary to copy elements here also (since elements are most efficiently accessed using a sequential iterator since



(a) Thread *B* is first to call `hasNext()`.

(b) Thread *B* returns from `hasNext()` call.

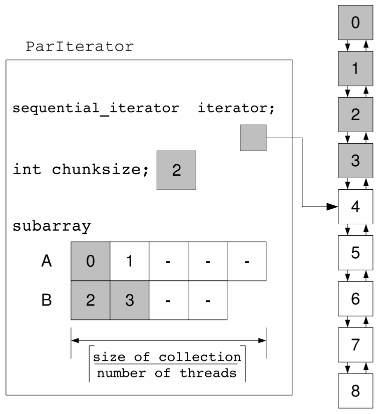
(c) Thread *A* calls `hasNext()` but no more elements remain.

Figure 4.2: Implementation of dynamic scheduling for inherently sequential collections. When a thread's private buffer is empty, elements are reserved from the collection.

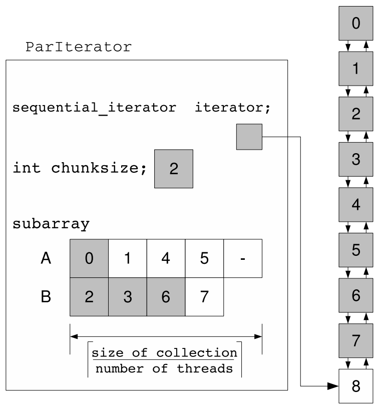
no constant-time access is supported to random elements). The difference now is that the buffer size is collection size/number of threads. Consequently, the static scheduling might need more memory compared to dynamic scheduling (this could be significant for very large collections).

By definition, static scheduling requires that the iterations are allocated first before any iteration is executed. But rather than naively copying all the elements at the beginning, the elements are copied *on-demand*. This is illustrated using the same collection of figure 3.3 where thread *A* is to be allocated iterations 0, 1, 4, 5 and 8 while thread *B* gets 2, 3, 6 and 7 (i.e. static cyclic scheduling with chunk size 2). The Parallel Iterator for this implementation may be visualised using figure 4.3. In figure 4.3(a), it is seen that only iterations 0 to 3 have been copied over into the `subarray` section; thread *A* has also completed iteration 0 while thread *B* has completed iterations 2 and 3. The sequential iterator is currently pointing at iteration 4 as the next element to copy across.

Assume now that thread *B* is ready for its next iteration, which should be iteration 6. Since the sequential iterator is still pointing at 4, thread *B* copies across all iterations up to iteration 7 as in figure 4.3(b); it then processes iteration 6 and leaves iterations 4 and 5 for thread *A*. This is *on-demand* copying,



(a) Before copy



(b) After copy

Figure 4.3: Static scheduling with *on-demand* copy. Situation shown when thread $B$ needs its next iteration (element 6), but the sequential iterator still points at element 4 (due for thread $A$) (a). Therefore, thread $B$ also copies elements on behalf of thread $A$ (b).

where iterations are only copied when required. When thread $A$ finishes iteration 1, it will find that its next iteration chunk (iterations 4 and 5) has already been copied over. The advantage of on-demand copying is that the copying is not a startup cost, since the copying and processing (by other threads) occurs at the same time and overlaps with computation.

### 4.1.2 Alternative interface implementation

As explained in section 3.1.1, the Parallel Iterator may take the form of an alternative interface: the `hasNext()` and `next()` methods are combined into one atomic `next()` method. The implementation however largely remains the same, except that the elements are reserved in the `next()` method (as there is no `hasNext()` method for this alternative Parallel Iterator interface).

With that said, there is an optimised implementation for this Parallel Iterator interface when using a dynamic scheduling policy and a chunk size of 1, regardless of the collection type. Here, the Parallel Iterator is implemented simply as a wrapper to a sequential iterator. Each call to `next()` on the Parallel Iterator involves acquiring a lock to the sequential iterator and calling `hasNext()` on the sequential iterator. If `hasNext()` on the sequential iterator returns `true`, then `next()` is invoked on the sequential iterator to retrieve the actual element before finally releasing the lock and returning the element.

### 4.1.3 Reductions

The `Reducible` class may essentially be viewed as an extended thread-local variable with thread-local `set()` and `get()` methods (just like Java's `ThreadLocal` class [94]). In addition to behaving like a thread-local, the class defines a `reduce()` method that accepts a `Reduction` object:

```
public class Reducible<E> {
  private Map<Thread,E> locals = ...;
  public E get() { ... }
  public void set(E e) { ... }
  public E reduce(Reduction<E> red) {
    if (has not been reduced yet)
      reduce local values and store result
    else
      return result
  }
```

Unlike a typical thread-local method which behaves locally to the calling thread, the `reduce()` method may be invoked by any thread. All the internal thread-local values within the `Reducible` object are accessed and reduced to a single result using the `Reduction` object (which defines an arbitrary reduction); the result is then stored in the case `reduce()` is called again (i.e. the reduction is performed only once). Section 3.1.4.1 showed an example using `Reduction.IntegerMIN`, which is one of many pre-defined reductions:

```
public interface Reduction<E> {
  public E reduction (E a, E b);
  public static Reduction< Integer > IntegerMIN =
                             new Reduction<Integer>() {
    @Override
    public Integer reduction(Integer a, Integer b) {
      return a < b ? a : b;
    }
  };
  public static Reduction< Integer > IntegerMAX = ...
  public static Reduction< Double >  DoubleMIN  = ...
  ...
}
```

The alternative interface for reductions (section 3.1.4.3) is implemented similarly. The only differences include using function pointers as arguments to the `reduce()` method (as opposed to a `Reduction` object) and thread-local values of the `Reducible` object are accessed through pointers. Since C++ also supports operator overloading, this interface allows the definition of type-independent reductions:

```
class Reduction {
public:
  template<typename T>
  static T min(T a, T b) {
    return a < b ? a : b;
  }
  ...
};
```

To make use of type-independent reductions for arbitrary types, programmers overload the respective operator. For example, to make use of the above *min* reduction to determine the smallest `Shape`, the $<$ operator is overloaded:

```
class Shape {
public:
  bool operator<(const Shape& other) const {
    return area() < other.area();
  }
};
```

### 4.1.4 Parallel remove

Supporting `remove()` is fairly straightforward. The `remove()` acts on the last element given to the respective thread through `next()`. Consequently, the Parallel Iterator stores this element until the thread calls `next()` again (in which case the new element is stored). In the case of most Parallel Iterator implementations, this element is stored into a collection containing elements to be removed; at the end when all threads reach the synchronisation barrier, one thread commits all the deletions from the underlying collection. In the case of Java, the `removeAll()` defined for the `Collection` interface is used to ensure efficiency.

### 4.1.5 Parallel break

The local and global breaks are implemented as conditions inside the Parallel Iterator. An invocation of `localBreak()` will set the condition for that thread, as well as release any allocated elements (to allow another thread to traverse them). The `globalBreak()` sets the condition for all threads. This way, all subsequent calls to `hasNext()` made by any thread return `false`, regardless of the number of elements yet to be processed.

If a thread successfully calls `localBreak()` and breaks from the Parallel Iterator, all reserved elements are released for the other threads to traverse. The breaking thread then waits at the barrier in the `hasNext()` method (therefore the `localBreak()` is a way to safely reduce contention). When the `globalBreak()` is called, the current iterations being executed are allowed to complete in order to safely terminate. For example, assume thread $A$ is working on an iteration and then thread $B$ calls `globalBreak()`. If thread $C$ calls `hasNext()`, it will be blocked (to synchronise loop termination between all threads) and `false` will be returned as soon as currently executing iterations are completed. Note that in comparison OpenMP does not allow a `break` statement within a work sharing construct such as `parallel for`.

### 4.1.6 Exceptions

Registering exceptions as presented in section 3.1.6 has been implemented for the Java Parallel Iterator only. When a thread invokes `register()` on the Parallel Iterator, it stores the exception to a `java.util.concurrent.ConcurrentHashMap`. The Parallel Iterator creates a wrapper exception (`ParIteratorException`) to associate other information with the exception:

```
public class ParIteratorException<E> {
  public Exception getException();
  public E getIteration();
  public Thread getRegisteringThread();
}
```

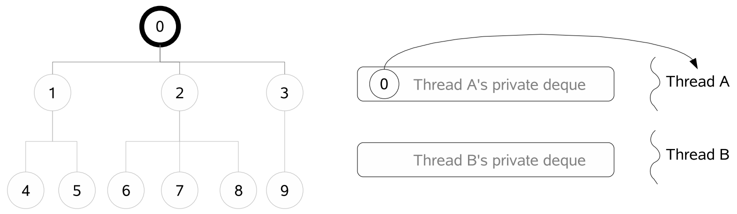
Along with the exception that has occurred, this also includes the current thread registering the exception (determined using `Thread.currentThread()`) and the element the thread is currently working on (according to the `hasNext` boundary as explained in section 3.1.6).

### 4.1.7 Tree Parallel Iterator

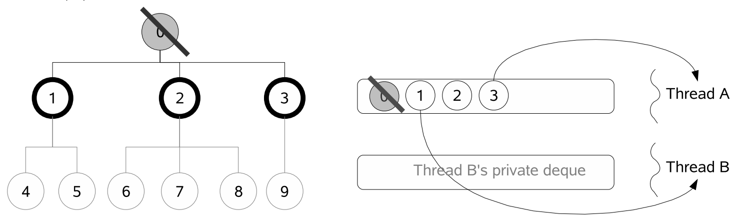
To make use of the Tree Parallel Iterator, any tree collection can be handled as long as it implements the `Node` interface mentioned in section 3.1.8 (since no generic tree collection exists in Java). Alternatively, the tree collection may be represented by a DOM `Document`. The scheduling implementation discussed in this section applies to both cases. The Tree Parallel Iterator must enforce a partial order: a node may only be executed when its parent node has completed. To enforce this partial order, the Tree Parallel Iterator only enqueues the children of a node at the time it completes.

A potential scheduling scheme to achieve this partial ordering is work-stealing [28]. This concept is not new and others have enhanced it with numerous variations. The variant that the Tree Parallel Iterator implements is based on the randomised work-stealing [17], which will be summarised below in the context of the Tree Parallel Iterator. Each thread has a private *deque* (a double ended queue) to store nodes that are ready to execute. When a thread operates on nodes on its own deque, a last in first out (LIFO) policy is used (therefore operating on the *latest* local node). When a thread's private deque is empty, it becomes a *thief* and selects a *victim* thread at random. The thief attempts to steal the *oldest* node on the victim's deque, therefore using a first in first out (FIFO) policy when stealing.

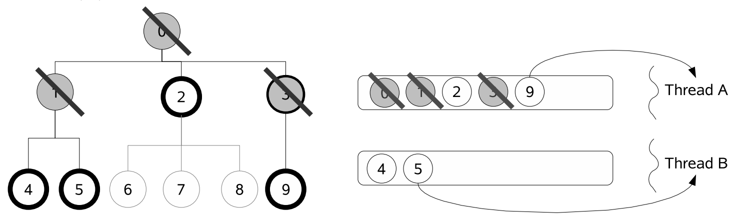
The benefit of work-stealing has been extensively developed and evaluated [18]. The reason that nodes are executed using a LIFO policy on the local deque is to take the initiative to execute a process towards the *depth* of the tree [28]; such behaviour models that of a sequential depth-first traversal (therefore reducing parallelism overhead when sufficient work exists). Threads encourage parallelism when they steal with a FIFO policy, as this expands the *breadth* of the tree (the thief takes ownership of a new sub-branch). This naturally encourages good data locality and cache reuse [72] since thieves favour the victim's oldest node (i.e. the *coldest* node in the victim's cache), while the *hottest* nodes are left for the local thread. In fact, some work-stealing variants further enhance data locality by performing a *locality-guided* steal (rather than a *random* steal)



(a) Initially, only the root node is ready for execution.



(b) Thread B steals the oldest node from thread A.



(c) Both threads take from their local deque.

Figure 4.4: Example run of the work-stealing implementation using 2 threads. Each thread maintains a local deque to store nodes that are ready to execute. Threads favour nodes on their local deque using a LIFO policy, while threads steal from other deques using a FIFO policy. Nodes are only added to the deque when the respective parent node has completed.

[1].

Figure 4.4 shows an example of 3 stages of the Tree Parallel Iterator, where 2 threads are traversing a tree consisting of 10 nodes. As discussed in section 3.1.8, a node is only scheduled to execute when its parent node has been completed. Therefore, initially only the root node is ready to execute, and this is placed on one of the thread's deque (figure 4.4(a)). In the context of the Parallel Iterator, the thread to process this node is the one that calls `hasNext()` first: in this example, this happens to be thread $A$. In the meantime, thread $B$ is trying to steal from another random thread (in this case it only has one other thread to steal from).

Since thread $A$ has been assigned node 0 (when it called `hasNext()`), it follows up with a call to `next()` to retrieve this node. The node is considered complete when thread $A$ calls `hasNext()` again, implying that it has completed node 0 and wishes to be assigned another node. In this situation, the Tree Parallel Iterator will enqueue the children of node 0 to thread $A$'s private deque (figure 4.4(b)). Consequently, nodes 1, 2 and 3 are ready to be executed.

Now that 3 nodes have been enqueued to thread $A$'s deque, thread $B$ has found its victim: it steals the oldest node from thread $A$, which happens to be node 1. In the meantime, thread $A$ takes it's latest node (node 3) and executes it. This example illustrates the discussion above, where thread $B$ has encouraged parallelism by stealing work in a breadth manner. On the other hand, thread $A$ has sufficient amount of work and continues to execute in a sequential depth-first manner (therefore minimising unnecessary parallelism overhead).

When thread $B$ completes it's computation (figure 4.4(c)), it enqueues the children (nodes 4 and 5) of the last node it completed. Consequently, thread $A$ does not need to perform another steal since it has unprocessed nodes on it's deque; thread $A$ now executes its most recent local node. Similarly, node 9 becomes thread $A$'s most recent local node when node 3 is completed. The parallel traversal of the tree is considered complete when all threads are attempting to steal. The `remove()` for the Tree Parallel Iterator is implemented similarly as discussed for the other collections in section 4.1.4, with the addition that children nodes are not enqueued if a `remove()` was called.

The major advantage of the Parallel Iterator is that implementation details are hidden from the user. First, by encapsulating the parallelisation logic within the Parallel Iterator, this separates it from the business logic of the loop body (this is especially valuable when traversing non-trivial collections in parallel, such as trees). As a result, the business logic is very similar to the sequential version. Second, the scheduling scheme details are hidden: this allows for other further scheduling implementations of the Tree Parallel Iterator. For example, another potential scheduling scheme might involve distributing nodes only when the children nodes are complete (i.e. processing leaf nodes first, therefore processing the tree in a bottom-up manner). Since the implementation details would be hidden from the user, the user code (to iterate nodes) does not require modification when a different scheduling scheme is used.

## 4.2 Performance

In this section, the Parallel Iterator is extensively tested using a number of benchmark applications for both the C++(Qt) and Java implementation. The objective is to evaluate the overhead it introduces and its ability to exploit the inherent parallelism of an iterative computation. Hence, the scalability is compared across a number of processors and the overhead in comparison to the sequential execution. The benchmarks were executed 4 times (each a separate JVM), taking the median of each. The times were measured using Java's `System.currentTimeMillis()` (or `System.nanoTime()` for fine-grained iterations) or Qt's `QTime::elapsed()`. In order to reduce effects of caching, benchmarks were repeated at least 3 times (5 times in most cases).

The performance of the Parallel Iterator is compared to the traditional Java parallelism approaches in section 4.2.1, while section 4.2.2 presents performance of the C++(Qt) version executing unbalanced and computationally intensive workloads. Section 4.2.3 focuses on more disk-intensive and potential desktop applications, and compares the performance of reductions to a commercial implementation of Qt. Finally, the Tree Parallel Iterator is evaluated in section 4.2.4. Note that these results were obtained during the course of the thesis research, and are therefore for different setups and workloads.

All the C++(Qt) benchmarks use OpenMP to create threads, while the Java benchmarks use Java threads. In every benchmark below, the baseline in determining the speedup is the sequential code that uses the standard sequential iterator. The benchmarks ran on shared memory systems which may be considered typical future desktop platforms running Linux. The first has two Quad-Core Intel Xeon processors (total of 8 cores) running at 1.86GHz with 8GB of RAM. The second system has four Quad-Core Intel Xeon processors (total of 16 cores) running at 2.4GHz with 64GB of RAM. Where noted, synthetic workloads (e.g. the Mandelbrot and Newton fractals) were selected because of high computation and low I/O, allowing for a better understanding of the Parallel Iterator's overhead.

#### Supporting sequential collections

The results discussed in this section refer to iterating over objects stored in random access collections (`QList` for C++(Qt), `ArrayList` for Java). The code was then modified to store the objects in inherently sequential collections that do not support random access to elements in constant time (`QLinkedList` for

C++(Qt), `LinkedList` for Java). The difference in time was imperceptible in the measurement noise, showing that the implemented parallel iterator is also suitable for these kind of inherently sequential collections. Namely, the buffering is not relevant in comparison to overhead.

### 4.2.1 Comparing to traditional Java parallelism approaches

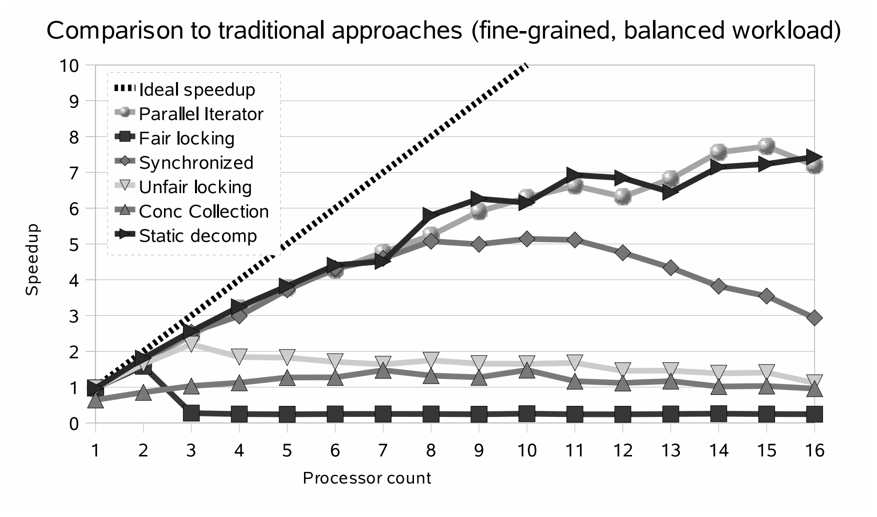
The first set of results presented are to emphasize the competitiveness of the Parallel Iterator, even in terms of speed. In this section, performance of the Parallel Iterator is compared to some of the traditional approaches (as discussed in section 2.2.2) that programmers would typically take to parallelise traversal of a collection of elements (namely in Java). The locking approach is further broken down into two categories, depending on the underlying implementation of the Lock:

- *Fair locking:* When the lock is competed for, access to the lock is favoured for the thread that has been waiting the longest [94].
- *Unfair locking:* When competed for, no order is guaranteed for access to the lock [94].

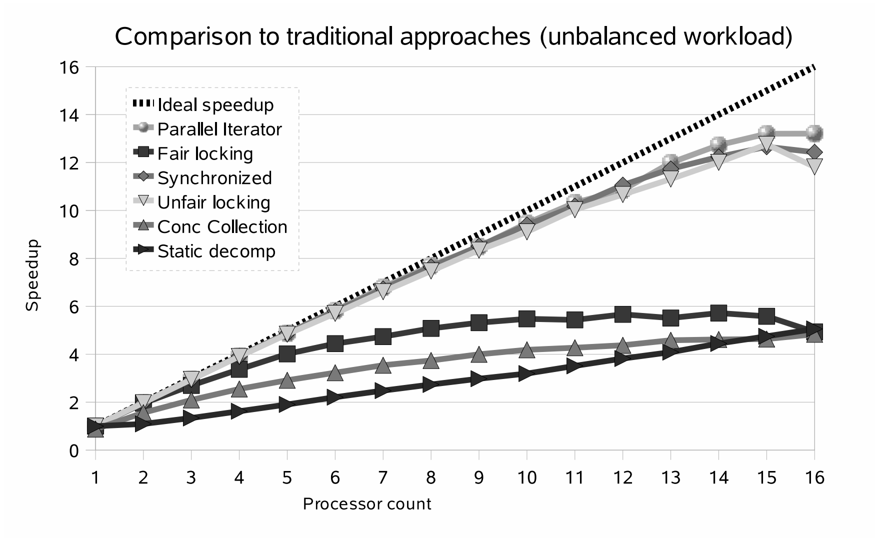
Each of the graphs in figure 4.5 represents a different workload (each workload contains 1 million iterations). Figure 4.5(a) shows the speedup for a balanced, but fine-grained, workload (each iteration takes on average $3.5\mu s$). Out of the traditional approaches, the best was static decomposition since this approach minimises runtime overhead and eliminates lock contention. The Parallel Iterator (here using guided scheduling with chunk size of 5) executes with similar performance. As expected, the locking approaches (especially fair locking) perform very poorly. Rather surprising is the concurrent collection's poor performance (using a concurrent queue from `java.util.concurrent`).

Figure 4.5(b) shows the speedup for an unbalanced workload. The Parallel Iterator (in this case using a dynamic scheduling policy with a chunk size of 10,000) is again the leading solution. Notice the inconsistency of the other traditional approaches: the static decomposition performed best for the first workload (figure 4.5(a)), while it performed worst in the second workload (figure 4.5(b)). The synchronised code and unfair locking were the only other approaches with respectable speedup.

These benchmarks show that the Parallel Iterator (with policy and chunk size tuning) is the only consistent solution across the different workloads. Most



(a) Balanced workload with fine-grained iterations.



(b) Unbalanced workload.

Figure 4.5: The Parallel Iterator is compared to some of the traditional Java parallelism approaches using different workloads.

importantly is that the user's iteration code remained unchanged across all workloads, even when a different scheduling scheme was used for the Parallel Iterator. This re-usability, combined with the performance across a range of workloads, is a very valuable contribution to object-oriented parallel programming. Without this easy fine tuning of policy and chunk size, programmers might struggle with the tuning especially for workloads with unknown characteristics.

### 4.2.2 Scheduling policies

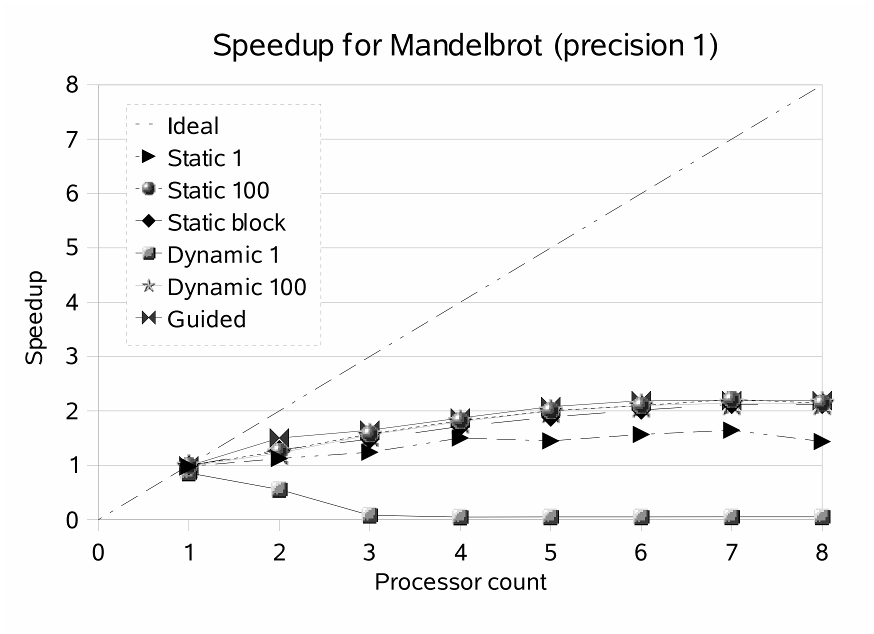
The Mandelbrot set [108] is a popular fractal consisting of a set of points in the complex plane. Because there is no dependence between the points, it is well suited to measure the overhead and the load balancing abilities of the Parallel Iterator. Although this is not a typical object-oriented application, it is interesting because of the unbalanced load (the amount of computation varies from point to point). Six levels of precision were tested, and their respective timings for the sequential code are shown in table 4.1. Level 1 produces a set with low precision, while level 6 produces a high precision set.

| Level | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Time(s) | 0.2 | 0.4 | 1.3 | 5 | 19 | 75 |

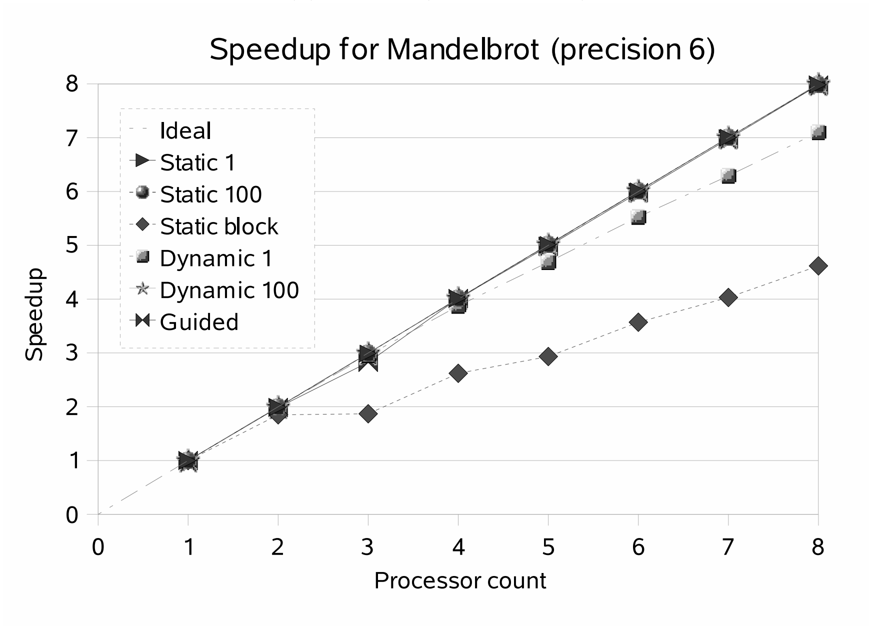
Table 4.1: Levels of precision and their sequential computation times

Figures 4.6(a) and (b) show the speedup over the number of employed processors for the different scheduling policies and chunk sizes. Figure 4.6(a) is for level 1 precision, while figure 4.6(b) is for level 6 precision. Both figures show that the overhead introduced is low when running the parallel code on a single processor. For level 1 precision, the application ran within 2% of the time for sequential code across all scheduling policies, except for dynamic scheduling with a chunk size of 1 in which case the program was 14% slower. For level 2 precision and above, the overhead quickly became imperceptible for most scheduling policies when running the parallel code on a single processor.

All scheduling policies demonstrated speedup greater than 1, showing it was worthwhile parallelising any levels of precision; figure 4.6(a) shows the only exception was for level 1 when used in conjunction with dynamic scheduling of chunk size 1. Even for the 3rd level of precision, speedups as high as 80-95% of the linear speedup were achieved when used with the guided scheduling policy.



(a) Level 1 (low precision)



(b) Level 6 (high precision)

Figure 4.6: Speedup for Mandelbrot benchmark. One benchmark involved low amounts of computation per iteration, while the second benchmark involved a higher degree of computation. In either case, Mandelbrot involves unbalanced workloads.

This was respectable considering the low computation in each iteration, as the sequential version itself was already only 1.3 seconds for all iterations. For the level 6 precision as in figure 4.6(b), all scheduling policies performed well except for static scheduling with block chunks. This was due to unbalanced regions of computational intensity in the Mandelbrot set, therefore threads are assigned with unbalanced workloads due to the scheduling policy. Most of the other scheduling schemes demonstrated almost linear speedup when used on level 6 precision. The unbalance in the Mandelbrot set is not completely random, and therefore even the cyclic static approaches (i.e. static 1 and static 100) balanced the load well.

The Mandelbrot benchmark illustrates the importance of having the correct scheduling scheme [90]. For example, the speedup for level 1 (lowest level) precision using 8 processors was 2.19 when using a guided scheduling policy, whereas a dynamic scheduling policy with a chunk size of 1 resulted in a speedup of 0.05 (20 times slower than the sequential version). Such a scheduling scheme would have been used in the case of a work queue, locking or `synchronized` method as discussed in section 2.2.2.

### 4.2.3 Disk-intensive applications

The scalability of the Parallel Iterator using more realistic desktop applications is now investigated, in particular those requiring high amounts of disk access. These results use the C++(Qt) implementation of the Parallel Iterator.

#### 4.2.3.1 Image resizing

In the next experiment, the application is image resizing of a folder of images. This is an example of a typical desktop application with inherent parallelism. Three different image sizes were used as shown in table 4.2, where resizing higher-resolution images is computationally more intensive. Testing was performed on three sets of images, each containing a total of 96 images as shown in table 4.3.

| Size | Resolution |
|---|---|
| small | 400x300 |
| medium | 800x600 |
| large | 1600x1200 |

Table 4.2: Image sizes and their resolutions

| Set | #Small | #Medium | #Large | Image Ordering |
|---|---|---|---|---|
| A | 0 | 0 | 96 | uniform |
| B | 32 | 32 | 32 | by size (L,M,S) |
| C | 32 | 32 | 32 | random |

Table 4.3: Sets of images used

Figures 4.7(a), (b) and (c) show the speedup over the number of employed processors for the different scheduling policies and chunk sizes. Figure 4.7(a) is for set A, figure 4.7(b) is for set B, and figure 4.7(c) is for set C. Regardless of the scheduling policy or set of images, all three figures show the parallel code executed with imperceptible overhead when running on a single processor.

For speedup, dynamic scheduling with low chunk sizes performed most consistently across all sets of images, achieving speedups between 90-99% of the linear speedup. Figure 4.7(b) shows static block and guided scheduling performed poorest when using set B, producing speedups of roughly 47% of the linear speedup. This is expected as the first thread gets the biggest portion of the large images. However, guided scheduling performed well on sets A and C, commonly producing speedups between 90-99% of the linear speedup. Static scheduling really only performed well for set A, with speedups of over 95% of the linear speedup when the iteration space is divided evenly amongst the threads.

These results once again confirm the importance of having a flexible way to modify the scheduling scheme of an iteration space. Static scheduling with a block chunk size produced poor results for computations with varying loads, yet this is possibly one of the easiest policies for programmers to manually achieve (section 2.2.2). Dynamic scheduling with a low chunk size would be most effective for image resizing, yet it was inappropriate for the Mandelbrot application. Consequently, the Parallel Iterator provides this flexibility since the chunk size and scheduling policy are parameters that may be chosen dynamically during run-time.

#### 4.2.3.2 Word count and permutation (reductions)

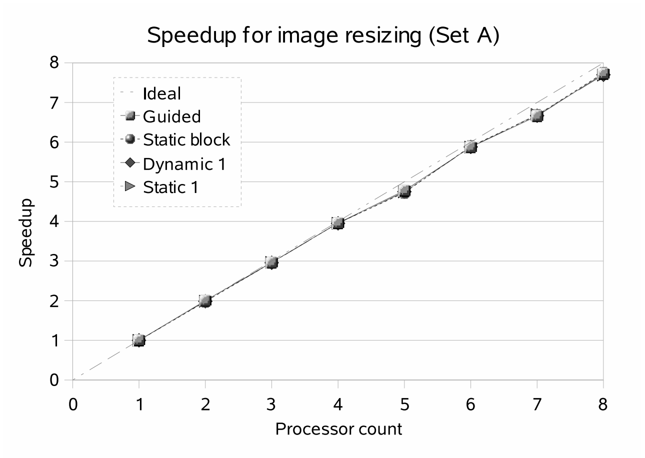
The first benchmark, word count, reads text files in a folder (and subfolders recursively) and counts the occurrence of each word. Such an application is typical of a search functionality on a desktop environment. The second benchmark, word permutation, also recursively reads text files in a folder. However, rather than just counting occurrences of each word, more complex permutations are performed on each word. Such behaviour would be typical of a spell checker functionality on a desktop environment. In both benchmarks, since files are searched in parallel, a reduction is required to collate individual file results into one final result set.

Figure 4.8 shows the speedup of the word count benchmark, comparing the `mappedReduced` method (section 3.2) of QtConcurrent (QTC) with the Parallel Iterator (PI). Two folders with C++ source files were used, one contained 650 files while the other contained 1755 files; the average sequential times were 0.5 and 2.1 seconds respectively. The first observation is the performance degradation due to the disk access bottleneck, but remember this is a realistic desktop application. The Parallel Iterator scaled better than QtConcurrent reaching a speedup of 2.9 with 6 processors for 1755 files. QtConcurrent peaked at 4 processors with a speedup of 2.2, but this required disabling processors since there was no support to limit the thread count[1]. The Parallel Iterator is more flexible since it allows the thread count to be adjusted as required per loop.

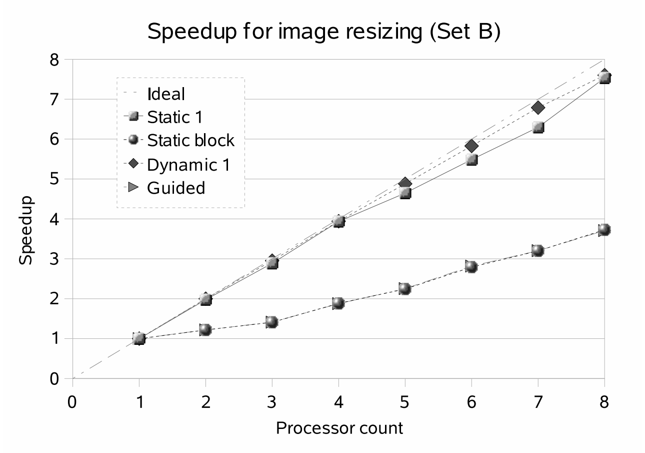
The speedup for the computationally intense word permutation program is shown in figure 4.9. Two folders containing C++ source files were used; sequential times took 18 seconds for 85 files and 186 seconds for 650 files. Disk access became less of an issue because more computation occurs for each file. On average, the Parallel Iterator performed slightly better than QtConcurrent on both folders. However, with only 85 files used, the efficiency slightly degrades as more threads came into contention.

In figure 4.8, the Parallel Iterator's performance is slightly better than QtConcurrent. This may be attributed to the high number of reductions that QtConcurrent makes. Assume a parallel program where N elements and P threads are involved. For QtConcurrent, N-1 reductions (therefore 1754 reductions in

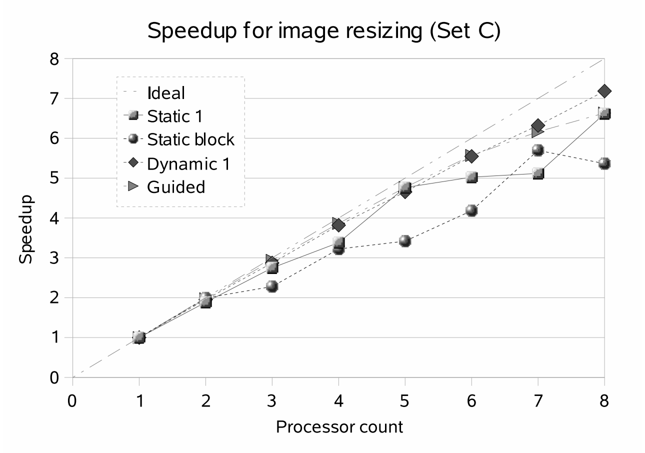
[1]In the meantime Qt 4.4 introduced the `QThreadPool` class to allow control over thread count. However, this thread count will consequently be applied to the entire `ThreadPool`, not only to one loop.



(a) Set A: The images in this dataset all had the same resolution.



(b) Set B: The images in this dataset had different resolutions. The images were ordered within the collection from largest to lowest resolution.



(c) Set C: The images in this dataset had different resolutions. The images were ordered randomly within the collection.

Figure 4.7: Speedup for image resizing benchmark. The scheduling policies where evaluated on varying datasets.

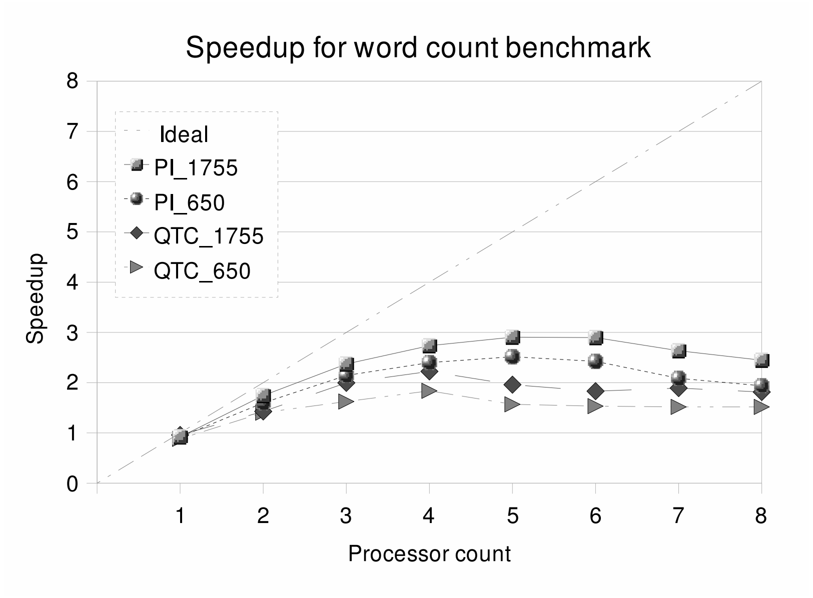


Figure 4.8: Speedup for the word count application. Performance in such an application is sensitive to the resource contention (disk access in this case).

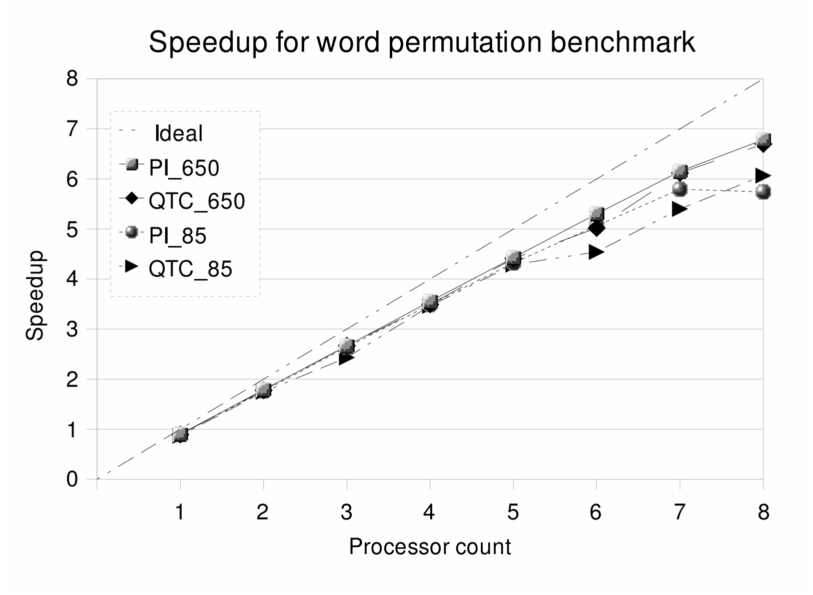


Figure 4.9: Speedup for the word permutation application. Although there is contention for disk access in this benchmark, this is alleviated by the high workload per iteration.

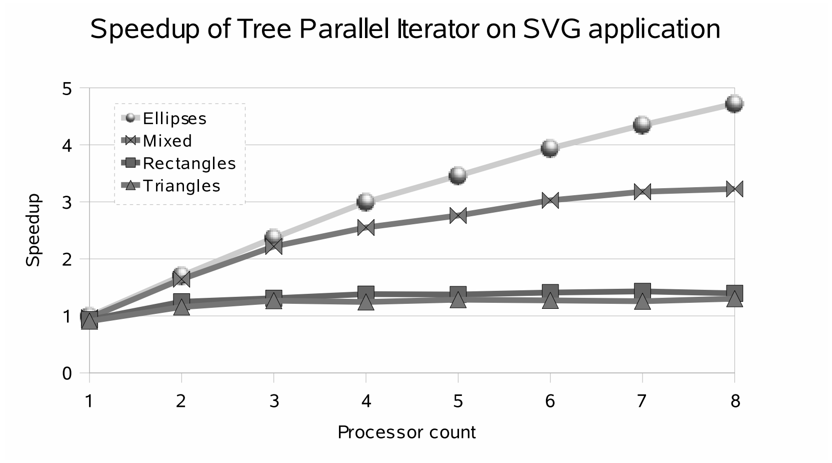


Figure 4.10: Speedup of the Tree Parallel Iterator for the SVG application. Four different workloads (each a different SVG file) were used: ellipses only, rectangles only, triangles only, or a mixture of all the shapes. Each SVG file contained 50,000 shapes.

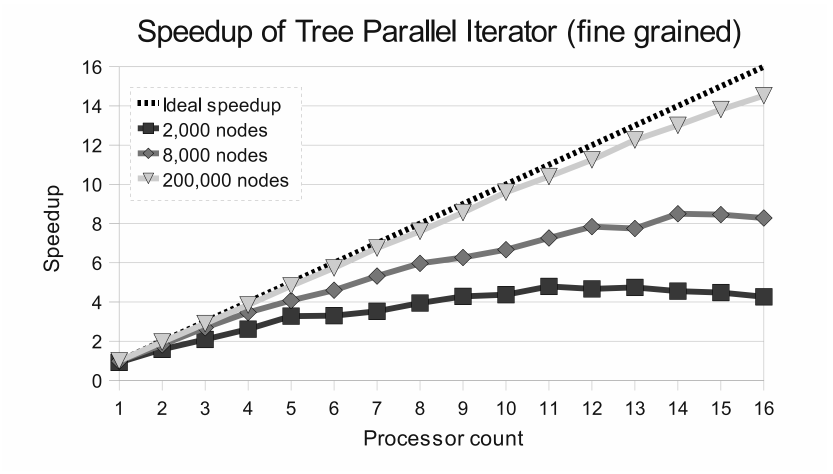
the case of the 1755-file word count) are required regardless of the thread count. The Parallel Iterator however only requires P-1 sequential reductions (therefore 7 when 8 threads are used). In addition to this, QtConcurrent requires that the code for each iteration be restructured into a separate method.

QtConcurrent uses all the available processors, but in some cases the programmer might want to reduce the thread count (this would have been very beneficial in the example of figure 4.8). This is especially important in a desktop environment when other applications are executing, especially with disk-intensive applications such as searching. By default the Parallel Iterator uses all available processors, but this may easily be modified for each loop.

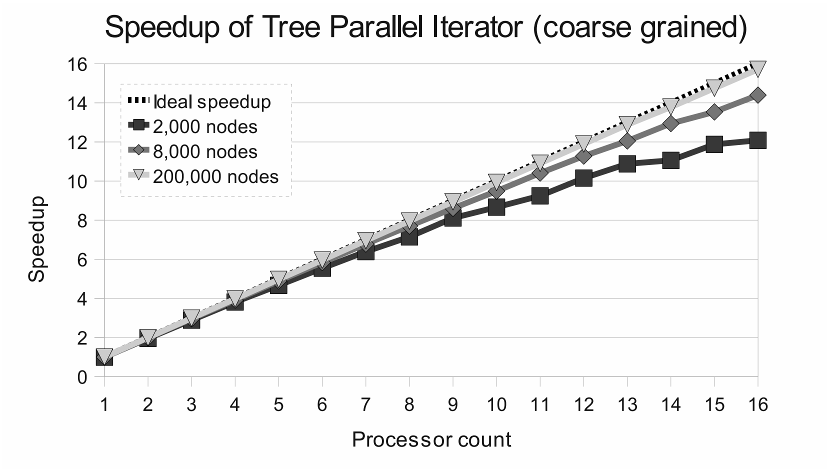
### 4.2.4 Tree Parallel Iterator

This section evaluates the Java implementation of the Tree Parallel Iterator as discussed in section 3.1.8. Two benchmarks are used: the first is the SVG shape application presented in section 3.1.8, while the second one is more computationally intensive. The important observation here is that good speedups can be achieved with the Parallel Iterator for very high level programs, employing object-oriented code, XML and SVG.

Figure 4.10 shows the speedup for the processing of an SVG file where the



(a) Low computation (0.4ms per node)



(b) Higher computation (6ms per node)

Figure 4.11: Speedup for Tree Parallel Iterator where each node performs a Newton-Raphson method. Notice how the speedup scales better for larger trees.

shapes are recognised. The 4 workloads shown denote the shape types in the SVG file. Notice that triangles and rectangles are a lot easier to recognise than ellipses. The fourth workload is a mixture of triangles, rectangles and ellipses. Just as was explored in the previous benchmarks, speedup improves with more computationally intensive applications.

The next set of benchmarks involve computing a synthetic load (here the Newton-Raphson method to produce a Newton fractal) at each node of the tree. The first benchmark of figure 4.11(a) contains fine grained computations, where each node involves approximately 0.4ms of computation. This shows that the benefit of parallelisation is greater for larger trees. For example, a tree with only 2000 nodes scales up to 11 threads to a speedup of almost 5. However, a tree with 200,000 nodes scaled close to linear speedup. This is quite encouraging considering the low amounts of computation at each node and the complex structure of the collection.

Figure 4.11(b) repeats the same experiment, only this time performing more computation at each node: in this case, each execution of the Newton-Raphson method takes 6ms. In such a workload, the speedup for each tree size is significantly better. The largest tree scaled to over 98% the linear speedup, while the smallest tree produced speedups of over 75% the linear speedup. This is encouraging, since for example, the runtime of the 2000 node tree in figure 4.11(b) is reduced from 12 seconds to 1 second; in terms of traditional parallel computing this is a very small workload.

In conclusion, the Tree Parallel Iterator is not only easy to use as discussed in section 3.1.8, but it also yields good speedup. By traversing the nodes of a tree, threads are assigned nodes that are ready to execute. The user code is simple and resembles standard iterator logic: users need not concern with children nodes and so on. The Tree Parallel Iterator hides all implementation details from the user, in particular the scheduling and synchronisation of nodes amongst multiple threads.

# Chapter 5

# Parallel Task concept and related work

This chapter introduces *Parallel Task* (short ParaTask) [50, 52], a task parallel solution for object-oriented applications (in particular for graphical user interface applications). This section discusses the features of ParaTask, while the full grammar is presented in appendix A. Related work that also targets object-oriented task parallelism is also discussed.

## 5.1 Parallel Task overview

### 5.1.1 Model overview

To investigate and implement a concept of task parallelism for object-oriented programs, the overall issue of tasks and threads was thoroughly analysed at the beginning of the research. The objective was to develop one general and unifying concept that would be a true task parallel concept. Threads in contrast are rather virtual processors, hence the thinking process of a programmer and the implementation is quite different to the general concepts of tasks. With that said, programmers are still left with the possibility to use threads alongside Parallel Task if they wish.

**Different types of tasks**

In analysing threading and concurrent programs, various task concepts can be identified. Some tasks have short runtimes and are computationally intensive; other tasks have long runtimes and are interactive (i.e. react to input/output). Some tasks execute once, while others need to be executed multiple times. Different task types are therefore supported by ParaTask, all unified in a single model:

- *One-off tasks*
  These tasks are CPU-bound computations. When invoked, a single instance of the task is enqueued to be executed from start to finish by any of the processors.

- *Multi-tasks*
  These are multiple tasks for data parallelism, hence they map to different processors.

- *Interactive tasks*
  These tasks are I/O-bound computations, for example background tasks waiting for events (e.g. mouse or key press). They should not be defined as one-off tasks since they would cause a backlog of ready-to-execute tasks [62]. Many tasks are perfect candidates for interactive tasks, for example web-based tasks. These tasks correspond to classical threads.

**Different types of threads**

Since ParaTask is a task-model, rather than a threading model, programmers think in terms of tasks rather than threads. This means that programmers do not require an understanding of ParaTask's underlying threading implementation. However, since ParaTask focuses on parallelising object-oriented GUI applications, programmers should be mindful of the existence of the following threads in order not to violate the structure of GUI applications:

- *ParaTask worker threads*
  A ParaTask application consists of a fixed number of worker threads (the default thread pool size is equal to the number of processors, but programmers may initialise the thread pool to a different size)[1]. Worker threads

[1]ParaTask may create more worker threads for special tasks, see section 5.1.2.3.

have the sole duty of executing tasks. Therefore, worker threads must not access any of the Swing components (as discussed in section 2.3.3).

- *The Event Dispatch Thread (EDT)*
  All GUI applications employ an EDT; its purpose is to handle events, and all access to GUI components must be restricted to the EDT. The EDT should not execute time-consuming computations, since an application with a graphical user interface must remain responsive.

- *User-defined threads*
  From the point of view of ParaTask, any other additional thread that is neither a ParaTask worker thread nor the EDT is a user-defined thread (therefore the main thread also falls under this category). Most applications have no need for such threads; however, ParaTask is aware of them just in case the user produces them (for example, ParaTask will not execute any tasks on user-defined threads). Finally, just like worker threads, these threads should not access any of the GUI components.

### 5.1.2 Syntax and semantics

The 3 task types presented in section 5.1.1 are implemented and supported by ParaTask. They are implemented with a single keyword, only using small modifiers to make them multi or interactive. For illustration, the same image application example from figure 1.1 is parallelised. To define a one-off task, the programmer annotates the method declaration with the `TASK` keyword:

```
public class ImageApplication {
  TASK public File compute1(String filename) { ... }
  ...
}
```

The `TASK` keyword acts as a modifier to a method declaration. This one-off task may now be invoked like a typical method:

```
TaskID<File> id = compute1(“image.jpg”);
```

The only difference to the standard method invocation is that a `TaskID` object is always returned, even if the original method signature has a `void` return type. All tasks have a unique global ID, accessed using `CurrentTask.globalID()`. Tasks with a return value are supported by ParaTask and will be discussed in section 5.3.

#### 5.1.2.1 Semantics of a task (as opposed to a method)

Throughout this thesis, any method annotated with the `TASK` keyword is referred to as a *task* (to distinguish it from a standard *method*). By making use of the `TASK` keyword, what is its significance?

- *Asynchronous execution*
  A task is essentially separated into two components: task *invocation* and task *execution.* After a task is invoked, its execution is always asynchronous with the caller (i.e. the task is executed by one of the ParaTask worker threads). Consequently, parallelism has been introduced by a single keyword. Programmers do not need to restructure the code, wrap sections of code in `Runnable` objects, or create and manage threads.

- *TASK keyword as a form of documentation*
  Since the `TASK` keyword is associated with the method declaration, it denotes that the method code is task-safe[2]. This places responsibility upon the task *implementer* to ensure safe asynchronous execution. As a result, *user*s of tasks are relieved from this burden of determining whether a task is parallel-safe.

By combining these two points, the usefulness of tasks is especially evident for event handlers: invoking a task from an event handling thread (such as the EDT) will enqueue it for execution on another thread (i.e. a ParaTask worker thread). This frees the EDT to respond to other events, supporting a responsive GUI. This also means that tasks must not contain GUI code since GUI components may only be accessed from the EDT (section 5.3.4 presents ParaTask features that support GUI-related work).

Since there are a fixed number of worker threads, blocking is discouraged within one-off tasks (unless blocking on ParaTask data structures, discussed in section 5.3.1). Similarly, lengthy background tasks should not be defined as one-off tasks as this causes a backlog of tasks that are ready to execute.

#### 5.1.2.2 Private and shared variables

When a programmer implements a task, variables will be used to define the task. Since a task is executed concurrently with its caller (and also concurrently with

[2]The term *task-safe* (as opposed to *thread-safe*) is used to denote parallel-safe code since ParaTask is a tasking-model (not a threading-model). This means that programmers think in terms of tasks rather than threads.

other tasks), it is important that the variables are used is a task-safe manner. ParaTask makes it natural for programmers to reason about which variables are task-private, and which variables are shared amongst other tasks and methods. Consider the following example code:

```
public class ImageApplication {
  private int numberOfErrors;
  ...
  TASK public String concatenate(List argsList) {
    String tempStr = ‘“’;
    ...
  }
}
```

Programmers should already be familiar with the following concepts since they are standard programming scope policies:

- *Private variables*
  Local variables defined within a task are immediately task-safe since their scope is limited to the task (for example `tempStr`). These task-private variables make it easy to reason about the task-safety of the task.

- *Shared variables*
  Variables defined outside a task (for example `numberOfErrors`) have a larger scope that allows them to be accessed by many tasks and methods (therefore no need to restructure code into separate classes); but, as usual in parallel programming, developers must regulate the access to such variables to ensure their task-safety. A subtle point that programmers should be aware of is that task arguments passed as references (for example `argsList`) are also shared since other code segments have reference to the object instance.

#### 5.1.2.3 Different task types

**Multi-tasks**
Multi-tasks support the concept of data parallelism or SPMD (Single Program Multiple Data), where the same task is executed multiple times. There are differences between invoking a multi-task once versus invoking a one-off task multiple times:

- A multi-task provides better documentation since the programmer is aware of the intention to execute it multiple times.
- The subtasks of a multi-task map to different worker threads in a round robin fashion (static scheduling) as opposed to all going to the same queue (dynamic scheduling).
- A multi-task has group awareness; this can be used for efficient partitioning or scheduling of the workload and allow operations such as reductions to be performed.

Whereas a one-off task is annotated with `TASK`, a multi-task is annotated with `TASK(*)` meaning it is created for each ParaTask worker thread. Alternatively, annotating the multi-task with any integer $n$ (instead of using *) will create $n$ tasks ($n$ may be an integer constant or integer variable name). In addition to global IDs, the following concepts are useful for multi-tasks:

- *multi-task size* represents the number of subtasks within a multi-task, and
- *relative ID* is the ID of the subtask with respect to the other subtasks in the same multi-task (starting at 0 and ending at `multiTaskSize-1`).

The following is an example of a multi-task:

```
TASK(*) public String multiTask() {
  int myPos = CurrentTask.relativeID();
  int num = CurrentTask.multiTaskSize();
  if ( myPos == 0 )
    print(‘‘Multi-task has ’’+num+’’ subtasks.’’);
  String name = ‘‘subtask’’+myPos;
  print(‘‘Hello from ’’+name);
  return name;
}
```

A multi-task is enqueued the same way as a one-off task, except that a `TaskIDGroup` is returned:

```
TaskIDGroup<String> multiID = multiTask();
```

A `TaskIDGroup` is used to store the `TaskID`s of the subtasks. A particular subtask may be accessed using `getSubtask(int relativeID)` by specifying the subtask's relative ID, or by iterating through all the `TaskID`s:

```
Iterator<TaskID<String>> it = multiID.groupMembers();
while (it.hasNext()) {
  TaskID<String> task = it.next();
  String result = task.getResult();
  ...
}
```

**Reductions** Section 3.1.4 discussed reductions, particularly in context with the Parallel Iterator concept. In respect with ParaTask, below is the code to implement the same example of figure 3.4 with ParaTask (in combination with the Parallel Iterator from chapter 3):

```
TASK(*) public int sum(ParIterator<Integer> pi) {
  int s = 0;
  while (it.hasNext()) {
    s += it.next();
  }
  return s;
}
```

By combining the Parallel Iterator with ParaTask multi-tasks, the programmer easily enqueues multiple tasks as follows:

```
1:   List<Integer> list = ...;        // get the list of numbers
2:   ParIterator<Integer> pi = ParIterator.create(list);
3:   TaskIDGroup<Integer> sumTask = sum(pi);
4:   int finalSum = sumTask.reduce(Reduction.IntegerSUM);
```

Line 2 creates a `ParIterator` instance for the list of numbers, which is then passed as an argument to the multi-task (line 3). Consequently, the multi-task has been enqueued and each of the subtasks will share the `ParIterator` instance. Line 4 performs a reduction: return values of the multi-task is reduced to one[3]. This is very intuitive, as a multi-task is also handled as one task.

Although this was a simple and trivial example, it is used for illustrative purposes. The real power of ParaTask comes out as more complex reductions

[3]Although a Parallel Iterator is used in this example, this ParaTask reduction is not to be confused with the `Reducible` object of section 3.1.4.1. The difference with the ParaTask reduction here is that it reduces return values from multi-tasks.

are handled elegantly. This example shows the power and ease of the ParaTask multi-task approach; the programmer only needs to focus on the business logic of their application since many parallelisation concerns are hidden:

- With just a single keyword, concurrency is introduced; the multi-task code (the `sum` task) is essentially the same as the original sequential method.
- By specifying * as the multi-task size, the optimal number of tasks is created (this equals to the size of the worker thread pool). Programmers may easily customise this with any integer value.
- Details and implementation of the reduction is hidden, this includes the necessary synchronisation of the sub-results. The sub-results of the multiple tasks are elegantly reduced into a final result using a single statement.

ParaTask provides a range of common reductions (e.g. sum, minimum, maximum). ParaTask also allows the programmer to define custom reductions; this object-oriented solution allows any kind of reduction while using any data type. The reduction must be *associative* (the order of evaluating the reduction makes no difference) and *commutative* (the order of the thread-local values makes no difference) since the interface does not specify order. Customised reductions are easily composed by providing an object that implements the `Reduction` interface. Only one function needs to be implemented, defining the reduction of two elements into one. Below is an example of a custom reduction:

```
public Reduction<Shape> biggestShape = new Reduction<Shape>() {
  public Shape reduce(Shape a, Shape b) {
    if (a.getArea() > b.getArea())
      return a;
    else
      return b;
  }
};
```

The user-defined reduction is then used as an argument to the `reduce` method on the `TaskIDGroup`. Reductions can generally be performed sequentially or in parallel (using a tree-network [43]). From the user's point of view, invoking a reduction in parallel is straightforward: the `reduce()` method has an optional boolean parameter to denote if the reduction is to be performed in parallel (the default is `false`):

```
int finalSum = sumTask.reduce(Reduction.IntegerSUM, true);
```

**Interactive tasks**
I/O-bound computations that block waiting for external events should not be enqueued on the global task pool. Tasks with such characteristics should be identified by the task's developer in order to preserve encapsulation (the user of the task should not need to know any implementation details). ParaTask enforces this responsibility by requiring such tasks be annotated using the `INTERACTIVE_TASK` keyword rather than the standard `TASK` keyword. Consider the following task that retrieves information from the web:

```
INTERACTIVE_TASK public File webSearch(String query) {
  // involves blocking and long waiting times
}
```

This interactive task is used in the same manner as an ordinary task:

```
TaskID<File> id = webSearch(‘‘foo’’);
```

Interactive tasks execute on a separate interactive thread (not enqueued on a worker thread) as soon as all `dependsOn` dependences (section 5.2), if any, have been satisfied. Interactive tasks execute immediately without waiting for a free worker thread, therefore improving the responsiveness of important interactive activities. Many tasks are perfect candidates for interactive tasks, for example web-based tasks: these tasks should start as soon as possible and should not preoccupy a worker thread when it starts.

#### 5.1.2.4 Nested parallelism

ParaTask allows nested parallelism, defined as a task enqueuing other tasks; this is especially important for recursive divide and conquer applications. Consider the following nested parallelism example:

```
TASK public void taskA() {
  TaskID id1 = taskB();
  TaskID id2 = taskC();
}
```

In this example, the parent task (`taskA`) enqueues the children tasks (`taskB` and `taskC`). Since ParaTask is an asynchronous model, the parent task is considered "complete" possibly before the inner tasks even start. Some threading models, on the other hand, are *fully strict* [16] and would include an implicit barrier at the end of `taskA`. Although this simplifies reasoning about the behaviour of the program (since the current method/task blocks until the children tasks complete), it unfortunately blocks the enqueuing thread. For example, the `actionPerformed` method of any interactive Java application must never block waiting for tasks it enqueued.

Tasks should be viewed as a substitute for threads. In a threading library, the programmer is allowed to create threads from within other threads, and the parent thread is allowed to end before the children thread ends (a common example is Java's main thread ending as soon as the EDT is up and running). Since threading libraries do not impose such an implicit barrier, then ParaTask also does not impose this restriction. If programmers want the current method to block until all inner tasks complete, then they may explicitly code this (just like they would have to do with a threading library). Section 5.3 discusses various solutions to easily achieve this.

#### 5.1.2.5 Canceling a task

ParaTask allows tasks to be safely canceled:

```
TaskID id = myTask();        // task is now enqueued to execute
...
boolean canceled = id.requestCancel();  // attempt to cancel task
```

The `requestCancel()` method may be called on a `TaskID` in an attempt to cancel it. If `true` is returned, ParaTask guarantees the task has not started executing yet and will therefore not be scheduled to execute in the future. If `false` is returned, this means the task has already started executing, or the task has already completed. If the task is already executing, then `requestCancel()` has no effect unless the task periodically queries its own status to check if a cancel has been requested:

```
TASK public void myTask() {
  ...
  if (CurrentTask.cancelRequested())
```

```
      return;
    ...
    if (CurrentTask.cancelRequested())
      return;
    ...
  }
```

#### 5.1.2.6 ParaTask clauses

The rest of the chapter will present various ParaTask clauses that may be used in conjunction with a task invocation. A task invocation may contain zero or more ParaTask clauses, in the form of the following example:

`TaskID myID = myTask(‘‘Hello’’)` *(ParaTask clause)*`*;`

Where *ParaTask clause* may be one of `dependsOn` (section 5.2), `notify` (section 5.3.4), `notifyGUI` (section 5.3.4), `notifyInterim` (section 5.3.5) or `asyncCatch` (section 5.4).

## 5.2 Dependences

In sequential programs, ordering constraints are naturally obeyed since code segments are executed one at a time in the specified order. For example, recall the dependence graph of figure 1.1. The following sequential program naturally observes the ordering constraints:

```
File f1 = compute1(‘‘myimage.jpg’’);
File f2 = compute2(‘‘myimage1.jpg’’);
File f3 = compute3(‘‘myimage1.jpg’’);
File f4 = compute4(‘‘myimage2.jpg’’, ‘‘myimage3.jpg’’);
```

Figure 1.1 requires that `compute2` and `compute3` do not commence until `compute1` is complete, and similarly `compute4` must wait for both `compute2` and `compute3`. In a sequential program as above, these constraints are naturally observed since invocation of these methods is synchronous with the caller (since the calling thread executes them).

However, if these methods were asynchronous (i.e. annotated with the `TASK` keyword), then their execution would overlap and possibly violate the ordering constraints. Specifying such ordering constraints in ParaTask is made simple by stating the dependences at the time the task is invoked:

```
public void actionPerformed(ActionEvent e) {
   TaskID<File> id1 = compute1(‘‘myimage.jpg’’);
   TaskID<File> id2 = compute2(‘‘myimage1.jpg’’) dependsOn(id1);
   TaskID<File> id3 = compute3(‘‘myimage1.jpg’’) dependsOn(id1);
   TaskID<File> id4 = compute4(‘‘myimage2.jpg’’, ‘‘myimage3.jpg’’)
                                dependsOn(id2,id3);
   ...
}
```

Tasks 2 to 4 cannot proceed until the tasks specified in the `dependsOn` clause have completed. The programmer does not need to manually code synchronisation mechanisms to manage such dependences; in this example, no synchronisation mechanisms such as barriers or wait conditions are required at all since the synchronisation will be handled by runtime system. By using the `dependsOn` clause, blocking is not needed in dependent tasks. Not only does this ease the programming effort, it also improves performance since only ready tasks are scheduled.

**No coupling between tasks**

The `dependsOn` keyword has an important benefit: it promotes the design of maintainable code [107] since there is no coupling introduced between tasks. In the particular example above, even though the 4 tasks have ordering constraints, the tasks remain decoupled; the tasks have no knowledge of each other. The programmer does not need to code any synchronisation mechanisms within the tasks. This allows the tasks to be reused for other applications that do not have the same dependence structure of figure 1.1. In threading libraries, the programmer must explicitly create wait conditions between the tasks, and this would be specific for the particular application.

**Deadlock**

Using the `dependsOn` keyword, programmers declare dependences at the time of a task's invocation. Therefore, a circular dependence cannot be created since a task may not depend on tasks yet to be enqueued (this makes sense because the same principle applies in sequential programs). Therefore, deadlock is not possible using the `dependsOn` clause. However, programmers may find themselves in a deadlock situation if they do not use other synchronisation mechanisms carefully (such as locks and wait conditions from threading libraries).

## 5.3 Task completion

A task is essentially a method that is executed asynchronously with the caller. Accessing the return value of a task (or simply synchronising with the task's completion) may be achieved using a blocking (section 5.3.1) or non-blocking approach (section 5.3.4). Since various threads have different roles, ParaTask distinguishes at runtime between the 3 different thread types (as discussed in section 5.1.1).

The definition of a *blocking* function refers to the situation where the computation following the function will not be executed until the function has completed. In other words, the calling thread is unable to progress until the function returns. Such a function is blocking, regardless of whether the calling thread is doing something useful in the meantime or not. In some cases, the calling thread will sleep or poll (i.e. not process anything useful). But in other cases, as will be discussed in this section, the calling thread may actually process other work while it remains blocked from the caller's point of view. ParaTask supports such "blocking", where in some cases control is given to the ParaTask runtime.

### 5.3.1 Blocking on a TaskID

Continuing on the image application example, the programmer may wish to access the result of the task invocation:

```
File finalImage = id4.getResult();
```

If the task has not yet completed, then the current thread will block until the result is ready. This is the common approach taken by implementations of the *future* concept (as discussed in section 5.8). Similarly, a programmer may block on the `TaskID` to wait for it to complete:

```
id4.waitTillFinished();
```

This is useful when waiting on a task that does not have a return value, or when the return value is not needed (therefore used purely as a synchronisation mechanism). In fact, `waitTillFinished()` is used implicitly when the programmer calls `getResult()`.

Both these blocking methods provide a convenient synchronisation mechanism for the programmer, and the blocking is generally acceptable so long as the waiting is for a short length of time. In using these methods, the programmer should be aware which thread is blocking. This is discussed below.

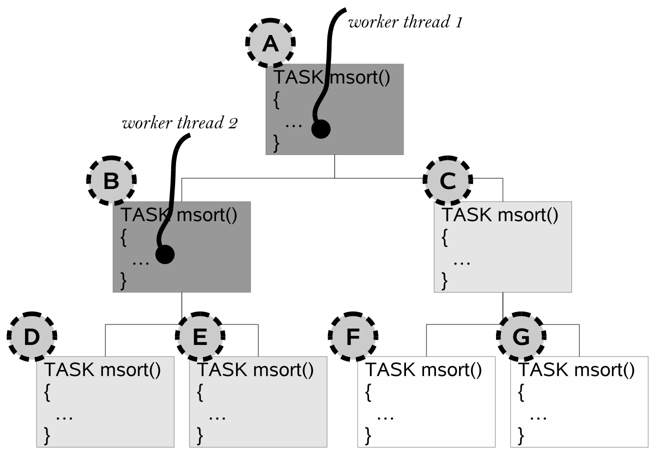


Figure 5.1: Visualisation of a recursive task with only two worker threads. Tasks A and B start executing, tasks C, D and E are created (but do not start executing), while tasks F and G are not even created. ParaTask avoids such deadlock situations.

### Blocking from within an event handling thread

Blocking has the disadvantage that the thread cannot progress until the task has completed. This impact is especially harmful when it is an event handling thread blocking (such as the EDT). Since this could cause a backlog of waiting events, programmers are encouraged never to block from an event handling thread. This rule of thumb not only applies to blocking on a `TaskID`, but any blocking in general made on an event handling thread [68]. Section 5.3.4 presents an alternative mechanism to support non-blocking notification of task completion.

### Blocking from within a ParaTask worker thread (nested parallelism)

ParaTask supports nested parallelism: namely, tasks may invoke other tasks. The reason this is a special scenario is because tasks might block on the `TaskID` of another task. As discussed in section 2.3.4, there is a fixed number of worker threads. When all worker threads are busy executing tasks, any other tasks that are ready to execute are enqueued until one of the worker thread frees up.

This poses a potential problem (namely deadlock) if all worker threads block indefinitely. For example, consider the following recursive divide and conquer task. It executes a sequential algorithm for small inputs up to a certain cutoff, while larger inputs are divided and the task is called recursively (effectively creating more tasks). The current task blocks until results from the subtasks are complete, at which point the final answer is returned:

```
TASK List mergeSort(List nums) {
  if (nums.size() < cutoff)
    return sequentialSort(nums);
  int middle = nums.size() / 2;
  TaskID left = mergeSort(nums.subList(0,middle));
  TaskID right = mergeSort(nums.subList(middle,nums.size());
  return merge(left.getResult(), right.getResult()); // blocks till both complete
}
```

Using such a model, the programmer assumes that there are sufficient worker threads to execute the other subtasks while the current worker thread blocks waiting for the sub-results. Figure 5.1 shows a visualisation of this task executing on a task pool of two worker threads. This example clearly shows how deadlock is possible if worker threads end up blocking. The first worker thread executes task A, creating two more tasks (tasks B and C) and waits for those tasks to complete. In the meantime, the second worker thread executes task B, creating two more tasks (tasks D and E) and waits for those tasks to complete. Both worker threads are blocked waiting for tasks to complete: deadlock has occurred. Tasks F and G have not even had a chance to be created.

One possible solution to avoid such deadlock would be to temporarily create additional worker threads to offload some of the tasks, therefore eventually allowing the original worker threads to continue. Unfortunately, such an approach is easily susceptible to large amounts of overhead due to the creation and management of the extra threads [23]. This is especially undesirable in a recursive situation that could potentially create lots of small tasks, as in this example.

Therefore, ParaTask takes the following approach to avoid deadlock: whenever a worker thread blocks on a `TaskID` (e.g. worker thread 1 blocking on task B and C), it executes another task from the ready queue. The next task it receives is dependent on the scheduling scheme (section 5.6) and may not necessarily be the particular task it is waiting on. In this example, worker thread 1 created tasks B and C. When worker thread 1 blocks on task B, it finds that task B has not completed yet (but it has started on worker thread 2). Therefore, rather than waiting for it to complete, worker thread 1 attempts to get another

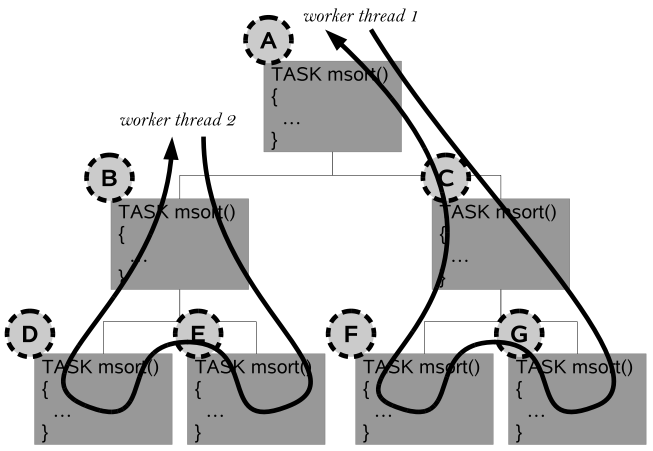


Figure 5.2: Visualisation of how ParaTask avoids deadlock, by executing other ready tasks when a worker thread blocks on a `TaskID` for a task that has not yet completed.

ready task: in this case, it finds task C and executes it. This behaviour repeats recursively, as worker thread 2 also blocks on unfinished tasks.

If tasks B and C have still not completed, then the worker thread attempts to execute yet another task. Figure 5.2 shows a possible execution of the same example using this approach. When the first worker thread creates tasks B and C, the second worker thread starts executing task B and creates tasks D and E. When the first worker thread blocks waiting for tasks B and C to complete, it gets a new task to execute: it finds task C is the next ready task. Tasks F and G are now created and enqueued. In the meantime, when the second worker thread blocks waiting for tasks D and E, it starts executing the next ready task: this happens to be task E, and similarly task D. When the leaf tasks are complete, the worker threads return to their original task: deadlock has just been avoided.

### 5.3.2 Blocking on non-ParaTask data structures

In the case that a worker thread blocks on a `TaskID`, then ParaTask determines this and automatically substitutes another task to execute as discussed above. As with any parallel programming environment, programmers should be careful with blocking on data structures that ParaTask has no knowledge about (e.g. concurrent data structures and locks from other libraries). Any form of blocking on a non-ParaTask object should be avoided while inside a one-off task or multi-task. At best, a blocked task will prohibit other ready tasks from executing (reducing performance). At worst, multiple tasks blocked will result in deadlock. To avoid these problems, programmers should use interactive tasks instead.

**Thread-safe is not necessarily task-safe**

The example of figure 5.2 illustrated an important point: a worker thread could potentially be switching between multiple tasks (when it blocks on an incompleted `TaskID`). For this reason, programmers should be careful when using certain mechanisms:

- *Thread-local storage*
  Using thread-local variables are discouraged since tasks might not execute atomically on a worker thread (in the case they are substituted when blocking on a `TaskID`). In fact, thread-local storage generally does not make sense in a tasking model such as ParaTask. This is because the programmer is not guaranteed which worker thread will execute a particular task (unlike a threading model where the programmer explicitly controls the threads executing methods).

- *Locks and condition variables from threading libraries*
  Since semi-executed tasks might be substituted with another task, locking inside of tasks could lead to nested locking [81]. This poses a potential problem as shown in the following simple example:

```
TASK public void taskA() {
   mylock.lock();     // start critical region
   ...
   TaskID id = taskB();
   id.waitTillFinished();
   ...
   mylock.unlock();   // end critical region
}

TASK public void taskB() {
   mylock.lock();    // start critical region
   ...
   mylock.unlock();  // end critical region
```

```
    }
```

> Consider the worker thread executing `taskA` that has been granted exclusive access to `mylock`. Consequently, this worker thread has just entered the critical region and assumes it has exclusive access until it unlocks at the end of `taskA`. Within this critical region, it blocks on another task. Since `taskB` is still waiting to be executed, the current worker might end up executing it (as discussed earlier). Consequently, the same worker thread has entered the critical region of `taskB` (since the worker thread already holds the lock to `mylock` [81]). In some situations, this might not be the desired behaviour since the critical region of `taskA` has not been exited yet.

The OpenMP tasking feature [82] also requires programmers to take the same precautions as above, since tasks might be substituted at various *task scheduling points* [9]. The interactive tasks in ParaTask allow the use of thread-local storage and blocking on non-ParaTask objects since these tasks are in fact threads.

## 5.3.3 Synchronisation inside multi-tasks

Synchronisation inside SPMD is a common and useful idiom in parallel computing and cannot be neglected. In the context of ParaTask, this particularly applies to multi-tasks when synchronisation is required between the subtasks. For example, consider the following situation:

```
// multi-task
TASK(*) public void A() {

    // perform computation
    ...
    barrier.wait();     // synchronise with sibling subtasks
    ...
    // continue

}
```

Consider if the `barrier.wait()` statement refers to an external library data structure that ParaTask has no knowledge about. As an extension of the discussion in section 5.3.2, this could lead to deadlock if other (multi-)tasks contained such blocking. Before presenting a solution to this problem, another situation is presented that is even more error prone: nested multi-tasks. Assume the following application has 4 worker threads and all multi-tasks are also of size 4, where $X_y$ refers to the $y^{th}$ subtask of multi-task X (e.g. $A_2$ is the second subtask of multi-task A):

```
// A(), B(), C(), D() and E() are all multi-tasks
TASK(4) public void A() {
    ...
    TaskIDGroup<Integer> id;
    int result;
    if (inside A1) {
        id = B();
        result = id.getResult();
        barrier.wait();  // deadlock if this blocks
    } else if (inside A2) {
        id = C();
        result = id.getResult();
        barrier.wait();  // deadlock if this blocks
    } else if (inside A3) {
        barrier.wait();  // deadlock if this blocks
        id = D();
        result = id.getResult();
    } else {
        barrier.wait();  // deadlock if this blocks
        id = E();
        result = id.getResult();
    }
    ...
    // do something with result
}
```

In this example, subtasks $A_1$ and $A_2$ spawn a multi-task each and reach the barrier *after* getting the result from their multi-task (`B()` and `C()` respectively). But subtasks $A_3$ and $A_4$ wait at the barrier *before* creating their multi-task (`D()` and `E()` respectively). If `barrier.wait()` is a blocking function external to ParaTask (i.e. control is not given back to the ParaTask runtime), this means deadlock has been reached: threads 3 and 4 will not execute their share of multi-tasks `B()` and `C()` (i.e. $B_3$, $B_4$, $C_3$, $C_4$) until the barrier is released, and threads 1 and 2 will not reach the barrier until all of `B()` and `C()` are completed.

Extending from the idea of the non-blocking `getResult()` in section 5.3.1, ParaTask supports a synchronisation barrier for multi-tasks. Instead of using an external library to synchronise the subtasks, programmers use `CurrentTask.barrier()`. This function is a barrier in the sense that the subtask does not progress past it until the other subtasks also reach it. Every ParaTask multi-task has its own barrier, initialised to the size of the multi-task. When the barrier has been reached, it is automatically reset to allow for further synchronisations within the same multi-task.

In the above example, here is how this avoids deadlock. When threads 3 and 4 reach the barrier, they check the queues to find another waiting (sub)task. They happen to find subtasks $B_3$, $B_4$, $C_3$ and $C_4$, which are eventually all executed since the threads cannot progress past the barrier. In a similar fashion, threads 1 and 2 will complete subtasks $B_1$, $B_2$, $C_1$ and $C_2$ while they are in `getResult()`. At this point all subtasks of `B()` and `C()` are completed and the barrier is released.

### 5.3.4 Non-blocking: The `notify` clause

As discussed in section 5.3.1, event handling threads (such as the EDT) should never block waiting for a task's completion. It is important for event handling threads to maintain control in the event loop so that events are handled promptly [31]. While blocked, the thread cannot process any pending events (for example, stalled repaint events will produce a frozen GUI).

Consequently, ParaTask provides non-blocking task synchronisation: when the programmer invokes a task, a comma-separated list of slots may be specified using a `notify` clause. These slots are called automatically when the task has completed, meaning that no thread or task is blocked while waiting for the task to complete. If the task returns a value, the user may `notify` a slot accepting a `TaskID` parameter (allows the result to be accessed from within that method). Adding to the code of section 5.2, the example initially introduced in figure 1.1 is completed:

```
public void actionPerformed() {
  TaskID<File> id1 = compute1(“myimage.jpg”);
```

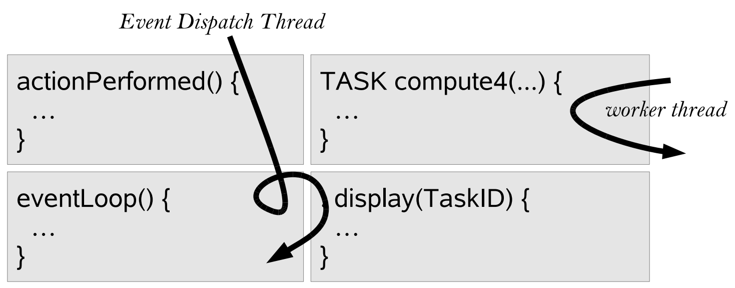


Figure 5.3: Using the ParaTask's `notify` clause allows the enqueuing thread to later synchronise with completion of the task without blocking. This is especially important for event handling threads.

```
    TaskID<File> id2 = compute2(‘‘myimage1.jpg’’) dependsOn(id1);
    TaskID<File> id3 = compute3(‘‘myimage1.jpg’’) dependsOn(id1);
    TaskID<File> id4 = compute4(‘‘myimage2.jpg’’,‘‘myimage3.jpg’’)
               dependsOn(id2,id3)
               notify(display(TaskID)[4]);
    // event-handling thread does not block
  }
```

The `display()` slot is defined as an ordinary method:

```
  public void display(TaskID<File> id) {
    File result = id.getResult();
    ...
  }
```

This example is illustrated in figure 5.3. Since `actionPerformed()` is an event handler, it is executed by the EDT. Consequently, the EDT is the *enqueuing thread* of the `compute4()` task: a worker thread executes this task while the EDT is immediately allowed to return to the `eventLoop()` to process other events. The EDT is informed (via the event loop) when the task completes, therefore allowing `display()` to be executed.

### Sequential semantics

Since ParaTask allows for multiple methods to appear in the `notify` clause, this supports good coding practices since the programmer may create sepa-

[4]The TaskID is needed if the method being notified accepts TaskID as parameter, otherwise it is omitted. This ensures at compile time that the correct method signature is used.

rate methods for different components and not need to put all code into one method. The `notify` clause has more semantic contributions than just being a non-blocking synchronisation mechanism: it essentially introduces *"sequential" semantics* that make it easier to reason about the program behaviour:

- *The original enqueuing thread executes the `notify` clause (not the worker thread that executes the task)*: In most cases, this means that slots inside the `notify` clause need not be coded task-safe since they will only be executed by the original enqueuing thread[5]. This is especially useful for the EDT:
  - When an event occurs, the EDT is easily and immediately freed since time-consuming tasks are dispatched to worker threads just by using the `TASK` keyword.
  - Any necessary updates are easily scheduled on the enqueuing thread using the `notify` clause.
- *Slots in the `notify` clause are executed in the same order they are declared:* For example:

  ```
  TaskID id = computeTask()
              notify( saveFile(), updateGUI(TaskID) );
  ```

  When `computeTask` completes, `saveFile` and `updateGUI` are executed in that same order. If the slots in the `notify` clause refer to tasks (rather than ordinary methods), then the *execution* order is not guaranteed (but the *enqueuing* order is guaranteed); such task "slots" will in fact be executed on worker threads rather than the enqueuing thread, since they are actually defined as parallelisable tasks[6].
- *Slots in the `notify` clause are executed before the `dependsOn` clause:* A task is not considered "complete" until the slots in the `notify` clause (and also slots in the `asyncCatch` clause as will be discussed in section 5.4.1) are executed. This distinction is important when used in combination with the `dependsOn` clause, as shown below:

[5]Slots might be executed concurrently with other threads (and would therefore need to be task-safe) if the same slot is used from multiple threads; there is only an implicit ordering with the enqueuing thread. In other words, they are synchronous with the enqueuing thread.

[6]If t1() and t2() are both TASKs, then *t1 notify(t2)* is equivalent to *t2 dependsOn(t1)*.

```
    TaskID id1 = task1() notify(slot1(), slot2());
    TaskID id2 = task2() dependsOn(id1);
```

In this example, when a worker thread completes `task1`, `slot1()` is executed by the original enqueuing thread (followed by `slot2()`). When `slot1()` and `slot2()` have both completed, only then will `task2` be ready to execute.

### The `notify` clause used from within a worker thread

Since ParaTask allows nested parallelism, this means that tasks may invoke other tasks. This means that the `notify` clause might also be used from within a worker thread. The following question now arises: if multiple worker threads use the `notify` clause, then how is the "sequential" semantics (see above) of the `notify` clause preserved? This is achieved by having a special worker slot thread (SWST) that is only responsible for executing *all* the `notify` clauses from *all* the worker threads. This way, the programmer need not make the slots in the `notify` clause task-safe, etc.

### Forcing slots to always execute on the EDT

In some cases, the programmer may want some slots to *always* execute on the EDT, regardless of which was the enqueuing thread. For example, consider the following:

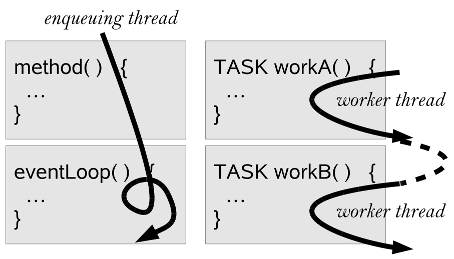
```
TASK public taskA() {
  TaskID id = taskB() notify(slot1(), updateGUI());
}
```

In this example, `taskB` is invoked from within another task (`taskA`). The problem with this code is that the `updateGUI()` is now due to execute on the SWST (since the original enqueuing thread is one of the worker threads). This is not the desired behaviour because any GUI-related work must only execute on the EDT. For this reason, ParaTask provides a slight extension to the `notify` clause: the `notifyGUI` clause; all methods inside a `notifyGUI` clause are executed on the EDT, regardless of who the enqueuing thread was. The above example is now correctly re-written (assuming that `slot1()` does not involve any GUI computations):

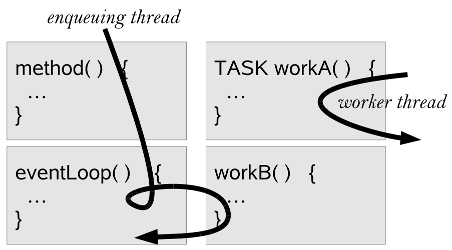
```
TASK public void workA() {...}
TASK public void workB() {...}
...
public void method() {
  ...
  TaskID id1 = workA();
  TaskID id2 = workB()
               dependsOn(id1);
  // no blocking
}
```

```
TASK public void workA() {...}
public void workB() {...}
...
public void method() {
  ...
  TaskID id = workA()
              notify( workB() );

  // no blocking
}
```



(a) taskB() as a TASK



(b) taskB() as a method

Figure 5.4: Semantic differences of task dependences (`dependsOn` clause) versus non-blocking notification (`notify` clause).

```
TASK public taskA() {
  TaskID id = taskB() notify(slot1()) notifyGUI(updateGUI());
}
```

Remaining consistent with the sequential semantics discussed above, the order of the `notify` and `notifyGUI` clauses will determine their execution order. Furthermore, programmers may interleave multiple `notify` and `notifyGUI` clauses to achieve a desired execution order for multiple slots.

### Semantic difference between `notify` and `dependsOn`

The methods in the `notify` clause will be executed when the task completes. A semantic question arises now: what is the difference between the two scenarios of figure 5.4? In both cases no blocking is involved, and `workA` must complete before `workB` begins. In scenario (a), `workA` and `workB` are both `TASK`s: they are both task-safe. Scenario (b) however treats `workB` as a normal method: it is not task-safe to be executed by any other thread.

Even though `taskB` in figure 5.4(b) is not task-safe, the programmer does not wish to block inside `method()` until `workA` completes (since the enqueuing thread should return to the event loop). For this reason, ParaTask will record the enqueuing thread of `workA` so that the methods in the `notify` clause will be executed on the enqueuing thread. This shows that methods in a `notify` clause have sequential semantics, as opposed to tasks, since they are executed by the same thread that enqueued the task.

### Using the `notify` clause on other instances

Consider the following use of the `notify` clause:

```
TaskID id = taskA() notify( slot1() );
```

From a semantic point of view, it is clear that `slot1()` must be a method defined within the current class. Consequently, slots are by default executed on `this` object instance (unless the slot is static, in which case no instance is needed). If the programmer wishes to invoke a slot on another object instance, they may do so:

```
TaskID id = taskA() notify( slot1(), myObj::slot2() );
```

In this case, there are two slots to notify. The first one is executed using `this` instance, while the second one is executed on the `myObj` instance. If the modifier of `slot2()` is not public, then the program will not compile. This preserves the standard Java access rules.

### Incorrectly using slots with a `notify` clause

When the programmer specifies slots in a `notify` clause, ParaTask enforces that the slots are valid at compile time (rather than delaying this to runtime as in Qt [103]). Some of the common errors that ParaTask avoids include: incorrect spelling of a method name, incorrect parameter type and invalid access (e.g. trying to notify a private method of another instance, or accessing an instance method from a static method). If such errors exist, then the ParaTask program will not compile and the programmer must fix these errors. Essentially, ParaTask ensures that methods are accessed correctly as the programming language requires, even if the methods are accessed through the `notify` clause.

**Threads without an event loop**

Since `notify` clauses are handled by posting events, each thread wishing to use the `notify` clause must have its own event loop. Where does this event loop come from? In the case of the EDT, it already has an event loop and ParaTask will enqueue `notify` clauses to be executed on the EDT. When programmers wish to use their own threads, those threads must register into a ParaTask event loop. This has been simplified for the programmer and involves two method calls to be executed by the registering thread:

```
public void run() {
  EventLoop.register();
  ...
  // create tasks, etc
  ...
  EventLoop.exec();
}
```

The `EventLoop.register()` method will initialise an event loop for the current thread. After calling this method, the thread is free to invoke tasks and use the `notify` clause. However, the notify methods will only be queued and will not be executed until the thread enters the event loop by calling `EventLoop.exec()`. The thread remains in this event loop until it calls `EventLoop.exit()` (this may, for example, be called from within a slot). One of the design features of ParaTask is that to be a generalised tasking model that may also be used for non-GUI applications. By providing this event loop, this allows `notify` to be used in a generalised fashion like the rest of the ParaTask features.

### 5.3.5 Interim progress and notifications

To improve the interactivity of an application, it is desirable to display a task's progress as it proceeds. For example, consider a task that retrieves photos from the web: it is undesirable to update the GUI only when all photos have been retrieved. Ideally, the task should update its progress (e.g. as a percentage) and display any retrieved photos (i.e. interim results) to the user. Recall from section 2.3.3 that only the EDT is allowed to access GUI components (e.g. the progress bar and other visual results). Achieving this is easy with ParaTask. First, methods interested in the task's progress (and interim results) are registered using the `notifyInterim` clause:

```
TaskID<List<Photo>> id = getPhotos(list)
           notifyInterim(updateSearchDisplay(TaskID,Photo));
```

The `notifyInterim` clause behaves much like the `notify` clause: the methods are processed in a non-blocking fashion by the enqueuing thread. The only difference is that methods inside the `notifyInterim` clause are invoked whenever the task publishes interim results:

```
 1:  INTERACTIVE_TASK public List<Photo> getPhotos(List<String> names) {
 2:    List<Photo> results = new ArrayList<Photo>();
 3:    for (int i = 0; i < names.size(); i++) {
 4:      Photo p = Flickr.getPhoto(names.get(i));  // web I/O
 5:      CurrentTask.setProgress((i+1)/names.size()*100);
 6:      CurrentTask.publishInterim(p);  // results published
 7:      results.add(p);
 8:    }
 9:    return results;
10:  }
```

In this example, `getPhotos()` is defined as an interactive task (line 1) since it retrieves results from the web. Each time a photo is retrieved (line 4), the task updates it's current progress (line 5). This allows other interested components (e.g. `updateSearchDisplay()`) to query the progress of the task. Line 6 shows an interim result (the newly retrieved photo) being published. Since the publication of interim results is non-blocking, this allows the task to progress with the computation without waiting for the interim result being delivered. All methods registered with the `notifyInterim` clause are executed by the enqueuing thread. In this example, the `updateSearchDisplay()` is executed by the EDT (the enqueuing thread) to update the GUI every time a new photo is retrieved. For example, the progress bar is updated and a thumbnail of the new photo is displayed:

```
public void updateSearchDisplay(TaskID id, Photo p) {
  progressBar.setProgress(id.getProgress());
  thumbnailsPanel.add(p);
}
```

Using the `notifyInterim` clause allows the tasks to remain decoupled from the methods interested in interim results. For example, the `getPhotos()` task above has no knowledge about `updateSearchDisplay()` or any other method interested in the interim results. This allows other methods to be connected/disconnected without affecting the task. Much like the `notifyGUI` version of the `notify` clause, there is also a `notifyInterimGUI` version of the `notifyInterim` clause. The `notifyInterimGUI` clause is necessary when the enqueuing thread is not the EDT, yet the user wishes to ensure the intermediate results are delivered to the EDT.

## 5.4 Exception handling

An exception is an event that diverts a program from its normal execution flow [54]. Many programming languages support exceptions to separate error-code from the user-code. This potentially produces more readable and efficient code since error handling is not integrated within the normal execution flow. Exceptions are especially important in object-oriented languages, hence exception handling with ParaTask is discussed.

When a thread runs, it invokes some methods that may in turn invoke other methods. This ordered list of method invocations is known as the *call stack*. When an exception is thrown, the call stack is analysed in the reverse order to find an exception handler capable of catching the exception. If such a handler is found, then the program continues executing from the handler. If no capable handler is found in the call stack, then the thread terminates. In the case of a single-threaded program, the program terminates.

Figure 5.5 shows the relationship between the different kinds of exceptions in Java, categorised into two main groups [110]:

- **Checked exceptions**
  These are exceptions that an application is expected to anticipate and recover from.
- **Unchecked exceptions**
  These are exceptions that an application is not expected to anticipate and recover from. Unchecked exceptions may further be broken down into:
  - *Errors:* These exceptions are external to the program, such as a system or hardware failure.
  - *Runtime exceptions:* These exceptions are internal to the program, such as a typical programming bug or logic mistake.

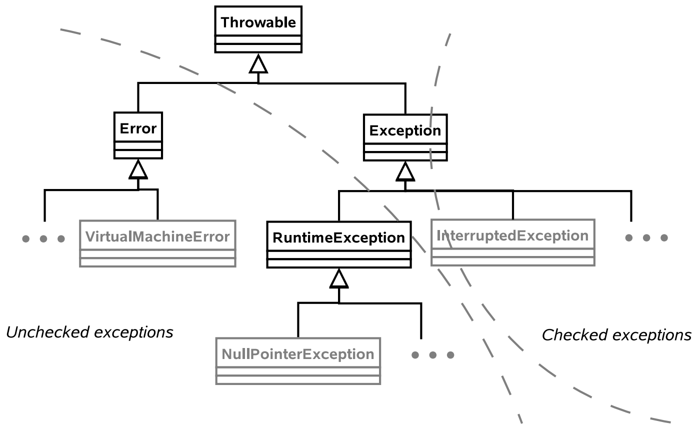


Figure 5.5: In Java, all exceptions and errors extend the `Throwable` class. As shown, these fall into one of two main categories: *checked* exceptions and *unchecked* exceptions. Applications are expected to anticipate and recover from all checked exceptions by conforming to the Catch or Specify Requirement.

**The Catch or Specify Requirement**

Since an application is expected to anticipate (and hence recover from) checked exceptions, how does Java enforce this? Code that might throw such an exception must conform to the Catch or Specify Requirement [110], otherwise the program will not compile. This requires the programmer to take one of two options:

- Surround the code with a try/catch block, or
- Use a `throws` clause for the current method to specify it throws such an exception.

Some programmers circumvent the Catch or Specify Requirement by developing programs that only throw unchecked exceptions (e.g. making all exceptions subclass `RuntimeException`). The temptation is convenience from omitting the extra code while avoiding complaints from the compiler. This defeats the intention of the Catch or Specify Requirement, possibly causing problems for the users since checked exceptions are a part of the method's signature [110].

### 5.4.1 Asynchronous exception handling

Consider the following task that throws two checked exceptions:

```
TASK public int myTask() throws MyExceptionA, MyExceptionB { ... }
```

Since this task throws checked exceptions, Java requires the programmer to follow the Catch or Specify Requirement. Unfortunately, the standard approaches to either "specify" or "catch" do not automatically extend to an asynchronous model as shown in the following cases:

### Incorrectly using Specify

In attempting to honor the Catch or Specify Requirement, a programmer may *specify* the checked exceptions in the signature of the enclosing method (e.g. `compute()` in the following example):

```
        // incorrect exception handling for tasks
1:      public void compute() throws MyExceptionA, MyExceptionB {
2:         TaskID id = myTask();
3:         // asynchronous model: enqueuing thread continues...
4:      }
```

Unfortunately, this is incorrect in an asynchronous model since `compute()` might return without throwing either of `MyExceptionA` or `MyExceptionB` since `myTask` (line 2) has not even started yet. Consequently, callers of `compute()` could face confusing results since `compute()` does not throw the checked exceptions synchronously (even though `compute()` is defined as a synchronous method). Of course, one solution would be to block the current thread (until `myTask()` completes) before returning from `compute()`. Although such blocking would honor the Catch or Specify Requirement, this is undesirable for event handling threads.

### Incorrectly using Catch

Alternatively, a programmer may attempt to *catch* the checked exceptions in order to honor the Catch or Specify Requirement:

```
         // incorrect exception handling for tasks
1:       try {
2:         TaskID id = myTask();
3:         ...
4:       } catch (MyExceptionA e) {
5:         myHandlerA();
6:      } catch (MyExceptionB e) {
7:        myHandlerB();
8:      }
9:      // asynchronous model: enqueuing thread continues...
```

Such a try/catch block around the method invocation only works in a synchronous model. But in an asynchronous model, such as ParaTask, the try/catch block is futile: the caller (enqueuing thread) continues to progress past line 9, possibly before a worker thread even starts executing `myTask()` on line 2. Again, a possible solution is to block the current thread at line 3 until `myTask` completes but this is often undesirable as discussed earlier.

**Parallel semantics for exception handling: the `asyncCatch` clause**

As shown above, using a standard try/catch block (or specifying exceptions in method signatures) will only produce the desired result in a *synchronous* model; it is therefore required to have parallel semantics for exception handling. In particular, this includes developing semantics for the Catch or Specify Requirement in an *asynchronous* model: ParaTask achieves this by using the non-blocking `asyncCatch` clause:

```
TaskID id = myTask() asyncCatch( MyExceptionA myHandlerA(TaskID),
                                 MyExceptionB myHandlerB(TaskID));
```

Using this `asyncCatch` clause is the asynchronous equivalent to the sequential try/catch block. A ParaTask exception handler is a standard method. Like the `notify` clause, methods in the `asyncCatch` clause are executed on the enqueuing thread (as shown in figure 5.4). The programmer may access the exception through the `TaskID`:

```
public void myHandlerA(TaskID id) {
  print("Task " + id.getID() + " threw an exception:");
  id.getException().printStackTrace();
}
```

The `asyncCatch` clause must be used when invoking tasks with checked exceptions, otherwise the ParaTask program will not compile. This ensures that the Catch or Specify Requirement is honored in the asynchronous model. The

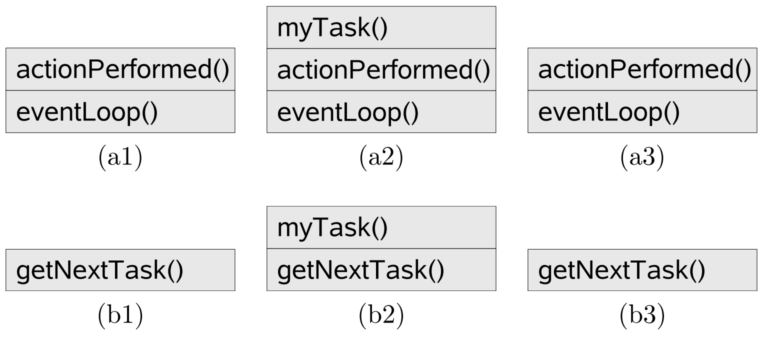


Figure 5.6: Visualising the respective call stacks for two threads. The progression of the enqueuing thread's call stack is shown in (a1) to (a3) as the task is *enqueued*. Figures (b1) to (b3) show the call stack of the worker thread as it *executes* the task.

`asyncCatch` clause may also be used on any task invocation (to also catch unchecked exceptions, not just for tasks with checked exceptions). The majority of the semantics of the `asyncCatch` clause are equivalent to the semantics of the `notify` clause as discussed in section 5.3.4.

### Propagating up the "task call" stack

In understanding the semantics of exceptions in a parallel environment, it is important to distinguish between the synchronous and asynchronous models:

*Synchronous exception handling:* In the synchronous model, a single thread calls methods. As a new method is called, the method is added to the top of the thread's *call stack*. Since the methods are synchronous, the top of stack refers to the currently executing method. When a method is completed, it is removed off the call stack.

If the thread encounters an exception, it starts looking for an appropriate handler. If one is not found in the current method, then the previous method on the call stack is analysed and so on until an appropriate handler is found. If no handler was found after analysing the entire call stack, then the thread's default handler is executed. In most cases, this means the exception's stack trace is printed and the thread terminates.

*Asynchronous exception handling:* In a model with asynchronous method invocations, such as ParaTask, things are slightly different than the synchronous model. Although threads still have their call stack, there is one particularly important difference: when an (enqueuing) thread removes an asynchronous method (i.e. a task) off it's call stack, this does not necessarily mean that the task has completed. This is illustrated in figure 5.6: figures (a1) to (a3) show the call stack of the enqueuing thread as a task is *enqueued*, while figures (b1) to (b3) show the call stack of a worker thread as the same task is later *executed*.

In (a1), the enqueuing thread is inside the `actionPerformed()` method and *calls* the task `myTask()`. At (a2), the thread is now inside `myTask()` in the process of *enqueuing* the task to be executed by another thread. In (a3), the task has been *enqueued* and the thread returns to the `actionPerformed()`. Therefore, `myTask()` has been removed off the call stack of the enqueuing thread (but the task has not started executing yet). Later on in (b1), a worker thread comes along looking for a task to execute. The worker thread starts *executing* the task in (b2), and finally the task is *completed* in (b3).

So, what is the significance of this asynchronous model when it comes to exception handling? First, unhandled exceptions that escape a task are termed as *asynchronous exceptions* (to differentiate them from standard exceptions that escape a sequential method). Therefore, if a worker thread encounters such an exception while executing a task (i.e. the position denoted by figure 5.6(b2)), then which call stack should it analyse? It cannot analyse it's own call stack since the previous method (i.e. `getNextTask()`) did not enqueue the task. It also cannot analyse the call stack of the original enqueuing thread since this could have largely changed (e.g. `actionPerformed()` removed from figure 5.6(a3), other methods added, and so on).

Consequently, the notion of a *task-call stack* (as opposed to a *call stack*) is introduced: this task-call stack is essentially the stack that is created due to nested parallelism (when a task enqueues another task). When a worker thread encounters an *asynchronous exception*, it traverses up the task-call stack to find an appropriate *asynchronous exception handler* (i.e. a `asyncCatch` clause). As discussed earlier in this section, the standard ways to catch or specify synchronous exceptions do not transfer to an asynchronous model. For this reason, the only valid exception handlers of exceptions resulting from tasks are those specified using a `asyncCatch` clause. The following example illustrates this:

```
public void method() {
  try {
    TaskID id = taskA()
```

```
                      asyncCatch(RuntimeException myHandler(TaskID));
 4:     } catch (Exception e) { ... } // never used
 5:     ...
 6:   }
 7:   TASK public void taskA() {
 8:     TaskID id = taskB() asyncCatch(IOException fileHandler(TaskID));
 9:     ...
10:   }
11:   TASK public void taskB() throws IOException {
12:     ...
13:     // exception thrown
14:     ...
15:   }
```

If an `IOException` occurs at line 13, then `fileHandler(TaskID)` is called since an asynchronous exception handler was registered on line 8. However, if a `NullPointerException` occurs at line 13, then the asynchronous exception handler of line 8 cannot support this exception (since `NullPointerException` is not a subclass of `IOException`). Therefore, ParaTask determines that `taskB` was invoked from within another task (`taskA`), so ParaTask propagates the asynchronous exception up the task-call stack. It determines that when `taskA` was enqueued (line 3), an asynchronous exception handler was registered: this handler is therefore called since a `NullPointerException` is a subclass of `RuntimeException`.

If ParaTask does not find an appropriate asynchronous exception handler (for example, assume line 13 threw a `ClassNotFoundException`), then the stack trace is printed and the worker thread continues to execute another task. This behaviour is equivalent to the EDT when it encounters an unhandled exception: the idea is that the entire application should remain responsive and not be terminated due to a single exception. Notice how the try/catch block of lines 2 and 4 are ignored by ParaTask, because such *synchronous exception handlers* are insufficient to handle *asynchronous exceptions* (even though `ClassNotFoundException` is a subclass of `Exception`).

As a note, there is one point that programmers should be aware of: an asynchronous exception handler could potentially be called multiple times due to nested parallelism (since multiple tasks may have been forked within the region guarded by an `asyncCatch` clause). For example, assume that `taskA` throws a NullPointerException at line 9: this exception is thrown after taskB has been enqueued. Assume that taskB also throws a NullPointerException at line 13 (unrelated reasons to the exception from line 9). In both these cases, the asynchronous exception handler on line 3 will handle these exceptions (once for taskA and once for taskB). Inside the myHandler method, the programmer may easily inquire on the TaskID to distinguish between them.

The power of ParaTask's asyncCatch clause is that it essentially follows the same expected behaviour as the standard rules of the synchronous try/catch block, but modified to conform in a parallel environment. In particular, this means that the programmer may specify multiple exception handlers on the same task invocation. Consider the following sequential program, using synchronous exception handling:

```
try {
  work();
} catch (IOException e) {
  fileHandler();
} catch (RuntimeException e) {
  myHandler();
} catch (Exception e) {
  defaultHandler();
}
```

The equivalent code for the a ParaTask program making use of asynchronous exception handling would consist of the following:

```
TaskID id = workTask() asyncCatch(
                       IOException      fileHandler(TaskID),
                       RuntimeException myHandler(TaskID),
                       Exception        defaultHandler());
```

The ParaTask syntax could even arguably be considered more legible than the original sequential code! In order to further illustrate the point, the asynchronous exception handlers (essentially just ordinary methods) may also accept a TaskID argument. This allows the programmer to determine which task encountered the exception for debugging purposes.

### 5.4.2 Exception handling while blocked on `TaskID`

The `asyncCatch` clause is intended for, but not limited to, tasks that specify checked exceptions. Therefore, the `asyncCatch` clause is optional for tasks that do not specify throwing any checked exceptions. However, just because a task does not throw checked exceptions, it does not mean that the task will not throw any other unchecked exception (e.g. runtime exceptions such as a `NullPointerException`).

For this reason, a thread that is blocked waiting for a task to complete must take this into consideration. The standard approach used by Java Futures [94] is to re-throw the exception when a thread attempts to access the result. ParaTask also supports this for the blocking methods `waitTillComplete()` and `getResult()` (i.e. the caller must be made aware that something might have gone wrong):

```
TaskID<Integer> id = myTask();
...
try {
  int ans = id.getResult();  // blocking method
}  catch (ExecutionException e) {
  e.getCause().printStackTrace();
} catch (InterruptedException e) {
  ...
}
```

The `ExecutionException` (part of the `java.util.concurrent` package) is thrown when attempting to get the result of a task that terminated by an exception. The `InterruptedException` (part of the `java.lang` package) is required since `getResult()` is a blocking method. Although most examples throughout this thesis have omitted this detail for readability, programmers are required to follow the Catch or Specify Requirement for these checked exceptions.

### 5.4.3 Dependences and notifications

The next question that arises is what happens in the case of dependences and notifications? Consider the following example:

```
TaskID id1 = taskA() notify(update())
                     asyncCatch(Exception myHandler(TaskID));
```

```
TaskID id2 = taskB() dependsOn(id1);
```

If a task (for example `id1`) exits with an exception, it is considered "complete" after the following takes place:

1. If an appropriate asynchronous exception handler is specified using the `asyncCatch` clause (e.g. `myHandler(TaskID)`), it is executed by the enqueuing thread (before continuing to points 2 and 3 below). If no asynchronous exception handler is found, the stack trace is printed (in this case points 2 and 3 below are not performed, essentially canceling any dependent tasks).

2. Next, if any method is specified in the `notify` clause (e.g. `update()`), they are executed by the enqueuing thread.

3. Only at this stage (after all the `asyncCatch` and `notify` clauses are executed) is the task considered "complete". Therefore, other tasks waiting (e.g. `id2`) are ready to be executed by a worker thread (assuming no more dependences on other tasks).

Therefore, the ordering of the clauses is not important (for example, if the `asyncCatch` clause in the above example was specified before the `notify` clause). The reason for this is that the `asyncCatch` clause allows a task to clean up after an exception occurs; only after cleaning up is it safe to continue with the sequential methods of the `notify` clause. Similarly, dependent tasks may then proceed since it is assumed that exceptions were cleaned up.

If any uncaught exceptions are encountered from methods inside a `notify` clause (or even a `asyncCatch` clause!), then the stack trace of those exceptions are printed. For simplicity, ParaTask does not allow exception handlers to be specified around clauses, since those exception handlers might in turn throw more exceptions. Furthermore, unhandled exceptions should not terminate the entire application (this follows the same policy as Java's EDT); but these unhandled exceptions will terminate dependent tasks as discussed in point 1.

### Canceling dependent tasks

As discussed above, dependent tasks will automatically be cancelled if the task they depend on terminates with an uncaught exception. If programmers do not want dependent tasks terminating, then an explicit `asyncCatch(Exception)` clause would be sufficient to catch any potential exceptions. In the case where exceptions are caught, the programmer may cancel dependent tasks in the exception handler. For example, the exception handler of the above example might look like this:

```
public void myHandler(TaskID taskWithExc) {
  TaskIDGroup dependentTasks = taskWithExc.getDependentTasks();
  dependentTasks.requestCancel();
  ...
}
```

The advantage of this solution is that `taskA()` and `taskB()` remain decoupled. Furthermore, the `requestCancel()` (section 5.1.2.5) will always succeed since none of the dependent tasks would have started yet. As an example, `taskA()` might refer to an image being loaded, while `taskB()` refers to it being processed, and the exception in `taskA()` would be `FileNotFoundException`.

Another solution would be to check for exceptions in the dependent task(s). This requires adding a `TaskID` to the parameter list of the dependent task(s):

```
TaskID id2 = taskB(id1) dependsOn(id1);
```

Here, the `TaskID` of the first task is passed to the second task, this allows checking to see if the first task completed without error using `getException()` on the `TaskID`. Alternatively, the exception will be re-thrown when attempting to get the result (section 5.4.2). This approach could also be applied to methods in the `notify` clause. Therefore, the programmer may customise whether the dependent tasks (such as `taskB()`) or notify methods (such as `update()`) should be executed if `taskA` encountered an exception. Although this solution is less elegant because it couples tasks together and requires passing `TaskID`s, it may be useful if dependent tasks need to know of the reason another task failed.

### 5.4.4 Multi-task exceptions

Unlike one-off tasks or interactive tasks, a multi-task might result in more than one exception. This is because a multi-task consists of multiple tasks. Consider the following multi-task definition, with a fixed size of 4:

```
TASK(4) public String multiTask() throws IOException {
  ...
}
```

By following the policies of section 5.4.1, below is a valid invocation of this multi-task:

```
TaskIDGroup<String> group = multiTask() asyncCatch(
                              IOException  handlerA(TaskID),
                              RuntimeException  handlerB(TaskID));
```

Notice that `IOException` is a checked exception, while `RuntimeException` is an unchecked exception: the discussion here applies equally to both cases. Even though this is a multi-task, the semantics is the same as has been discussed throughout this section. The only thing to note is that the handlers are invoked once for each exception instance. For example, assume that two of the subtasks encountered an `IOException`, while one subtask encountered a `RuntimeException`. This means that `handlerA` is called twice, while `handlerB` is called once; in each case, the `TaskID` passed is that of the respective subtask that encountered the exception (from the subtask's `TaskID`, the programmer may gain access to the `TaskIDGroup` that represents the entire multi-task).

## 5.5 Grouping tasks

A programmer may wish to invoke multiple independent tasks, yet treat them collectively as a group to simplify their management. Consider the following example:

```
TaskID id1 = task1();
TaskID id2 = task2();
...
TaskID id9 = task9();
```

Here, nine independent tasks are created. These tasks may be added to a task group as follows:

```
TaskIDGroup group = new TaskIDGroup();
group.add(id1, ..., id9);
```

Tasks may also be added to the group one at a time. However, at some stage, ParaTask needs to know how many tasks are inside the group (for example, in order to correctly determine when the group is complete). Therefore, the group is "sealed" automatically before it is used:

- *In a* `dependsOn` *clause:*

  `TaskID finalID = finishTask() dependsOn(group);`

  Here, the necessity of sealing is obvious since ParaTask needs to know when `finishTask()` may safely start.

- *When blocking:*

  `group.waitTillComplete();`

  Again, the necessity of sealing is clear since ParaTask needs to know when the blocking thread may safely resume. Not only is this simpler than waiting on multiple `TaskID`s, but it is also more efficient since the blocking thread will only wake up when all tasks have completed. Otherwise, the thread will be woken up, only to block on the next `TaskID`.

If the programmer attempts to add another `TaskID` to a group after it has been sealed, then a runtime exception is thrown. The programmer may access the individual `TaskID`s of the tasks using `getMembers()`, therefore allowing access to the individual results of each task. This is especially useful for multi-tasks.

## 5.6 Scheduling schemes

Throughout this thesis, ParaTask has been shown to achieve different examples of parallel patterns [36]. For example, not only is the *embarrassingly parallel* pattern covered, but also the use of the *divide and conquer* pattern (the merge sort example of section 5.3.1). The underlying scheduling scheme is crucial for the performance of a parallel application. Three scheduling schemes are currently supported, but more may be added as plugins. Users may select the most suitable one for their application (one schedule is allowed per application). ParaTask's default schedule is *mixed* (section 5.6.3) as it combines the benefits of *work-sharing* (best for fairness) and *work-stealing* (best for nested parallelism).

### 5.6.1 Work-sharing schedule

The first scheduling scheme is a simple work-sharing policy: tasks are executed using a fair policy (i.e. in the order they were originally enqueued). This is beneficial for *pipeline* pattern applications where the user expects older tasks to complete before newer tasks. Such a scheduling policy is important for perceived performance from the user's point of view (as demonstrated in section 6.2.2) since the user expects tasks to complete in the rough order they were launched (such as thumbnail previews). For example, consider an application that is expected to behave like the *pipeline* pattern:

```
for (int i = 0; i < data.size; i++) {
  TaskID id1 = stageA(data[i]);
  TaskID id2 = stageB(id1) dependsOn(id1);
  TaskID id3 = stageC(id2) dependsOn(id2);
}
```

The purpose of showing this pipeline pattern is to illustrate the scheduling of *dependent tasks* in ParaTask. This pipeline has three stages, and the output of each stage is used as input to the next stage (in this example, assume that `stageA` is much longer than the other stages). Notice that `stageA` is always ready to execute (it has no dependences), therefore a possible output of the execution trace might result as follows:

```
stageA: iteration 0
stageA: iteration 1
stageA: iteration 2
stageB: iteration 0
stageB: iteration 1
stageB: iteration 2
stageC: iteration 0
stageC: iteration 1
stageC: iteration 2
```

The output above results if the implementation executes dependent tasks in the order they were *ready* (not the order they were originally enqueued). For example, `stageA` of all iterations were ready before any iteration of `stageB` (and were therefore put on the ready queue earlier). To avoid this behaviour, tasks are prioritised according to their *original* enqueuing timestamp (rather than the time they were ready to execute). This allows the trace to be as follows:

```
stageA: iteration 0
stageB: iteration 0
stageC: iteration 0
stageA: iteration 1
```

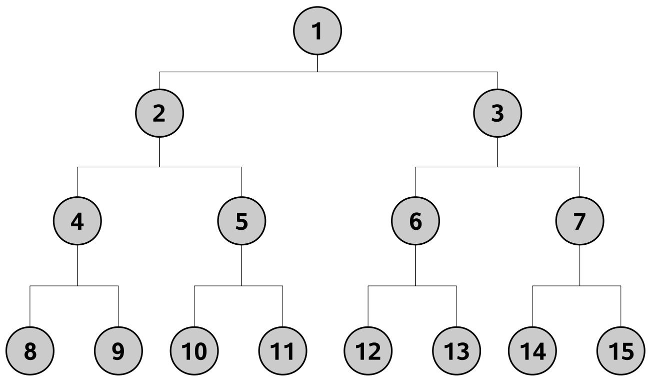


Figure 5.7: This recursive application is an example of nested task parallelism. In this case, a parent task is only considered complete when its children are completed. The numbers denote the time stamp of when the task was originally enqueued. To increase efficiency, tasks should not be executed in this same order (and even becomes impractical in larger applications).

```
stageB: iteration 1
stageC: iteration 1
stageA: iteration 2
stageB: iteration 2
stageC: iteration 2
```

Of course, the actual order is not guaranteed (except when the `dependsOn` clause is used between tasks); however, this example illustrates that the next favoured task to execute is the one with the earliest original enqueue. Such a scheduling is achieved using a work-sharing policy.

### 5.6.2 Work-stealing schedule

The performance of some applications might suffer significantly with a work-sharing schedule (as will be shown in section 6.2). This is especially the case for recursive applications, for example those based on the *divide and conquer* pattern. Consider the example recursive application of figure 5.7: the numbers denote the order the tasks were originally enqueued. If the tasks were executed in this same order (i.e. using the fair work-sharing policy of section 5.6.1), then this will result in extremely poor performance:

- Since each task is executed in the original order it was enqueued, this implies that the whole task tree needs to be in memory since a task is not considered complete until its children are complete. In other words, a thread's call stack keeps increasing until the last enqueued tasks are reached. Therefore the memory footprint is extremely high. Not only will this reduce performance due to increasing garbage collection, but it also becomes impractical to execute larger applications due to limited amounts of memory.

- Executing such an application in a breadth manner results in poor cache reuse [72, 1], since a thread executes "colder" tasks in preference to the latest tasks spawned.

Clearly, an intuitive schedule would be to execute the children tasks first (depth-first manner in order to complete the current task sooner); however, the work-sharing schedule of section 5.6.1 will instead jump branches and execute the tree in a breadth-first manner. For this reason, ParaTask also supports a work-stealing schedule [28]. This ensures a minimal call stack for the threads, since it only consists of the depth of the task tree. The work-stealing variant used by ParaTask is based on the well-established randomised variant [17], which will be discussed in section 6.1.2.2.

### 5.6.3 Mixed schedule

What happens in an application containing a component that requires fairness, while another component involves highly nested parallelism? Which of the scheduling policies should be chosen? A work-sharing policy has the benefit of fairness (but fails for the nested parallelism component), while work-stealing has the benefit of aptitude for nested parallelism (but fails in fairness). ParaTask supports a combined scheduling scheme that defaults to work-sharing (i.e. is fair), but whenever nested parallelism is detected, these tasks are handled in a work-stealing manner. This overall policy ensures that ParaTask maintains fairness whenever possible, but temporarily (yet necessarily) resorts to a work-stealing schedule for nested parallelism components. As will be shown in section 6.2, this scheduling policy allows us to take the best of work-stealing and work-sharing.

## 5.7 Adherence to object-oriented programming

By introducing concurrency to an object-oriented language, one must discuss the impact this has on important object-oriented concepts; the three most important ones are encapsulation, inheritance and polymorphism [107]. A discussion on the inheritance anomaly is also presented.

### 5.7.1 Encapsulation

Encapsulation, an important aspect of object-oriented programming, protects the attributes and methods of an object from improper use [107]. A major aspect of encapsulation is data hiding, since any irrelevant internal details are hidden. ParaTask promotes encapsulation in the following ways:

- *Concurrency is coordinated by the callee:*
  In ParaTask, concurrency is coordinated by the *callee* rather than the *caller* (the `TASK` keyword is a modifier associated with the declaration rather than invocation). Whether a task is safe to execute concurrently is the responsibility of the implementation. This preserves encapsulation for two reasons [85]. First, the implementation details of a task remain hidden from the caller. Second, modification of a task's implementation will not require modification of any caller code since the task signature remains the same.

- *ParaTask enforces policies of access specifiers:*
  The access specifiers (for example, public, protected and private) of the programming language are enforced by ParaTask. Not only does this include access to tasks, but also access to methods specified inside `notify` and `asyncCatch` clauses (presented later in this chapter). For example:

  ```
  TaskID id = myTask() notify(myObj::target());
  ```

  Assume that `target` refers to a protected method defined inside the class of instance `myObj`. Since the statement `myObj.target()` is not allowed inside the current class that `myTask` is called from, the above code will not compile. Therefore, ParaTask ensures that programmers do not circumvent the policies of access specifiers.

### 5.7.2 Inheritance

Reusing code is very important in object-oriented programming. One of the ways this is achieved is by having classes inherit code from other classes, known as inheritance. From Java's point of view, a ParaTask task is essentially an ordinary method; consequently, tasks are seen as standard class members and are therefore inherited by subclasses. Tasks may even be overridden by subclasses, just as methods are overridden; the only requirement is that the `TASK` keyword is also carried down, otherwise the program will not compile. Therefore, the `TASK` keyword is viewed as a modifier and is very much a part of the method's signature.

### 5.7.3 Polymorphism

Following on closely to inheritance, polymorphism is also a very powerful object-oriented concept. Multiple subclasses inherit the interface of a common superclass, allowing for each subclass to respond differently to the same method. As discussed in section 5.7.2, the `TASK` keyword is a modifier and subclasses must therefore be consistent. Even though the logic of tasks may be overridden, their synchronicity may not: methods will always execute synchronously and tasks will always execute asynchronously. Therefore, there is no confusion when programmers invoke tasks.

### 5.7.4 Inheritance anomaly

The integration of concurrency with object-oriented languages is said to introduce a new set of problems termed *the inheritance anomaly* [75, 78]. Although this is an interesting and important issue, it is not confined specifically to concurrent object-oriented programming. In fact, this section shows that the inheritance anomaly still affects sequential object-oriented programs: it is merely more visible and apparent for concurrent object-oriented programs. Therefore, a counter-example against the original definition of inheritance anomaly is presented, and then an improvement of the definition is proposed.

The argument made here is that the inheritance anomaly occurs when the (subclassed) *object* is responsible for maintaining correct usage of the object (rather than relying on the *user*). A similar example from [78] is re-used to explain this. Consider the following sequential code defining a buffer:

```
    public class Buffer {
```

```
  ...
  public Object get() {
    if (empty)
      throw new NoSuchElementException();
    ...
  }
  public void put(Object v) {
    if (full)
      throw new IllegalStateException();
    ...
  }
  public int size() { ... }
}
```

From the code above, it is evident that the *user* of the buffer object is responsible for its correct usage. For example, the user should not insert elements into a full buffer or attempt to withdraw an element from an empty buffer. The programmer now wishes to extend this buffer by introducing a new method `gget()` that works like `get()`, except that it may not be immediately executed after a `get()` (this is the same example of [78]):

```
public class HistoryBuffer extends Buffer {
  ...
  public Object gget() {
    return super.get();
  }
  // put(Object), get() and size() all inherited
}
```

In this case, the inheritance anomaly does not arise because the `put(Object)`, `get()` and `size()` methods are inherited. However, it is the responsibility of the *user* to ensure that `gget()` is used correctly (and not the responsibility of the *object* `HistoryBuffer`). The programmer now redefines the `HistoryBuffer` in such a way that it contains some error handling to ensure it is used correctly (rather than naively relying on the user):

```
public class HistoryBuffer extends Buffer {
  boolean afterGet = false;
  ...
  public Object gget() {
    if (afterGet)
      throw new IllegalStateException(``Cannot call after get()!'');
    afterGet = false;
    return super.get();
  }
  public Object get() {
    Object o = super.get();
    afterGet = true;
    return o;
  }
  public void put(Object v) {
    super.put(v);
    afterGet = false;
  }
  public int size() {
    int s = super.size();
    afterGet = false;
    return s;
  }
}
```

Note that the programmer has not introduced concurrency, yet the inheritance anomaly exists: `get()`, `put(Object)` and `size()` must be redefined! This counter-example illustrates that the inheritance anomaly is not specific to the introduction of *concurrency* within the object, but rather to the introduction of *responsibility of correct usage* within the object. Since synchronisation is a subset of this responsibility, this explains why the inheritance anomaly exists in concurrent objects. Therefore, this thesis proposes that the following definition is better suited for inheritance anomaly:

> The hindrance of inheritance when the *responsibility of an object's correct usage* is incorporated within the object rather than solely relying on the object's user.

For this reason, the inheritance anomaly is something that also affects sequential object-oriented programs: it is only *more likely* to occur for concurrent programs since thread-safety in these programs will demand responsibility.

## 5.8 Related work

An important aspect of developing new parallelisation tools is to focus on the user requirements [84], in this case the desktop application developer. Most related work have the objective of *combining parallelism with object-oriented applications.* Even though most desktop applications are in fact object-oriented, this objective is insufficient for successfully parallelising desktop applications. Rather, this thesis proposes that the focus should be on *combining parallelism with (graphical) object-oriented desktop applications.* The keyword here, *desktop*, is vital: one must understand and respect the structure of graphical desktop applications (as discussed in section 2.3.3) before attempting to introduce parallelism [95]. This is one of the first aspects that distinguish ParaTask from previous work. ParaTask's objective is not only to use concurrency to improve performance, but also to create responsive GUI applications that do not freeze. Moreover, ParaTask remains generalised enough to be applicable to console applications without a GUI.

There has been numerous research on combining concurrency with object-oriented programming: Briot et al [22] has classified some previous work into various categories, while Philippsen [85] surveyed over 100 concurrent object-oriented languages. In both surveys, the most relevant work includes Concurrent Object Oriented Language (COOL) [32], Compositional C++ (CC++) or some form of the *active object* pattern [59, 2]. More recent work includes Cilk++ [35], ThreadWeaver [19], QtConcurrent [104], Intel Threading Building Blocks (TBB) [63], the new OpenMP tasking feature [82, 9], Apple's Grand Central Dispatch (GCD) [7], Java 1.6's SwingWorker [94] and Foxtrot [20]. Other languages currently under development include X10 [33] and the Visual Studio 2010 Task Parallel Library (TPL) [76].

The first primary difference is that ParaTask, as presented here, uniquely integrates different task types into one concept. Second, none of the related work provide support for automatically handling task dependences (except for ThreadWeaver). Third, ParaTask has the primary focus of providing parallelism to desktop applications without code restructuring: this implies adherence to the threading model of graphical applications (section 2.3.3) and conforming to object-oriented concepts. In particular, ParaTask guarantees that tasks are always executed asynchronously with the enqueuing thread (unlike, for example, OpenMP, Cilk++, TBB and TPL). This is a fundamental requirement for responsive concurrent applications [97, 98]. Further specific differences are discussed below.

An active object is essentially a proxy that separates method invocation from method execution. Although this pattern works well for many distributed computing applications, it is not well suited for shared memory models such as programming desktop applications. Furthermore, most active object implementations do not support intra-object concurrency: only one method executes at a time while others are delayed. Even though this makes the program correctness easier to justify, it greatly reduces parallel performance [85].

ThreadWeaver and Intel TBB have a task concept where programmers enclose independent code snippets within a separate object. But just like active objects and thread libraries, these tools require considerable code restructuring. For example, ThreadWeaver requires the task's user code to be defined within the `run()` method of a `Job` instance: very similar to the requirements of threading libraries. `Job`s can only be used once, therefore a new instance needs to be created for each invocation of the task. Intel TBB is similar, except that the term `task` is used instead of `Job`, and `execute()` instead of `run()`. For example, below is the definition of a task:

```
class MyTask: public task {
public:
  ...
  task* execute() {
    // user code
  }
}
```

To make use of this task, programmers must allocate space for the task and explicitly spawn the task to make it start:

```
MyTask& t = *new (task::allocateSpace()) MyTask(); // create task
task::spawn(t);  // start task
```

Some languages, such as X10 (which is designed for non-uniform cluster computing rather than the shared memory systems of current desktops) and CC++, create concurrency using a special keyword in front of the method invocation. This brings convenience to the programmer since code does not need to be restructured into a separate object. However, here lies a fundamental disadvantage typical of concurrent languages that coordinate concurrency by the *caller* (as opposed to the *callee*): *encapsulation*, an essential object-oriented programming concept, is immediately broken [85]. First, implementation details of an invoked method must be understood by the caller. Second, all caller code must be carefully analysed every time the callee code is modified.

OpenMP's `task` construct also falls under this category, where programmers wrap code with compiler directives:

```
#pragma omp parallel
{
  ...
  #pragma omp single
  {
    ...
    #pragma omp task
    myTask(‘‘Hello, World’’);
    ...
  }
} // implicit barrier
```

Note the number of additional pragmas required: the `parallel` pragma creates a team of threads, while the `single` pragma ensures the `task` is enqueued only once (since the `task` pragma must be used within a `parallel` pragma environment). The `parallel` pragma itself includes an implicit barrier at the end, therefore all tasks must complete before exiting this construct; it is therefore unacceptable to place the `parallel` pragma within an event-handler. Synchronisation is achieved by using an optional and explicit `taskwait` pragma (for nested tasks), causing the *current task* to block until *all* its *children tasks* finish. In order to achieve finer-grained synchronisation (i.e. subset of the tasks), programmers must wrap the task subset within a *dummy* task and wait on that subset [47]. All of these show that the underlying threading model of OpenMP is not well suited for developing interactive GUI applications.

Rather than using a keyword, other concurrent languages coordinate concurrency by the use of a special enqueuing method. The advantage here is that concurrency is introduced within libraries, therefore eliminating the need for an additional intermediate compiler. Examples of these languages include QtConcurrent and TPL. Below is a QtConcurrent example:

```
QFuture<String> future =
      QtConcurrent::run(QtConcurrent::bind(myMethod, "Hello, World"));
```

The *future* object [88] allows return values to be accessed once the asynchronous task has completed. Unfortunately, these languages still possess the disadvantage of breaking encapsulation since concurrency is coordinated by the caller. Furthermore the resulting code is less legible, especially when the programmer must bind the arguments to the method.

COOL takes a different approach: concurrency is coordinated at the callee side by prefixing the `parallel` keyword to method declarations (just like for ParaTask). Although COOL addresses the broken encapsulation issue discussed above, it still possesses some setbacks that are also common to CC++. First, return results are discarded. Second, these languages spawn user-level threads for each task, resulting in poor performance when a large number of smaller tasks are created (especially in recursive divide and conquer applications) [62].

Cilk++ allows programmers to annotate method declarations with a special keyword to denote parallelisable methods. The advantage is that the serial version of the program is produced when the keywords are removed from the parallel code. ParaTask's work-stealing implementation is motivated by Cilk++'s well-proven work-stealing scheduling policy [17]. Unfortunately, Cilk++ does not focus on event-based desktop applications; one cannot parallelise a GUI application using this model (section 6.2.1.3). The same semantics have also been applied to JCilk [39]: a Java implementation that also takes into consideration exception handling. The difference is Cilk++ and JCilk implement a *work-first* policy, while ParaTask implements a *help-first* policy [56].

In the work-first policy, threads leave their current task to sequentially execute a newly created task. Other threads may then steal from where the thread left. This performs well for extremely fine-grained nested parallelism since the thread completes the task before another thread has a chance to steal (where stealing is expensive). In the help-first policy, threads enqueue the newly created task (in the hope that another thread will execute it) while the thread continues with its current task. In terms of desktop application semantics, a work-first policy would violate the structure of multi-threaded GUI applications: this would mean the enqueuing thread (i.e. the GUI thread) executes the tasks (which will reduce the application's responsiveness), and other worker threads would continue where the GUI thread left off (but only the EDT is allowed to access GUI components). In fact, it is not possible to develop a parallel GUI application using this model. Assume the method `myTask()` is annotated with the `cilk` keyword; below is the "required" code, for example using JCilk:

```
// code will not compile
cilk public void actionPerformed() {
  // EDT thread starts to execute event handler
  ...
  result = spawn myTask();
  ...
  // implicit cilk barrier waits for spawned tasks
}
```

The above code fails on many aspects in an event-handling environment:

- Since `actionPerformed()` is calling a `cilk` method (i.e. `myTask()`), then it must be annotated with the `cilk` keyword itself (otherwise the Cilk compiler will complain). But this is not possible, since the Java compiler will not allow the signature of `actionPerformed()` to be modified. Even if it were possible, it would provide poor documentation since `actionPerformed()` is now a cilk method too!

- When `myTask()` is spawned, the EDT begins to execute it (work-first policy). If another thread were to help, it would steal where the EDT left off (i.e. continue the statement after the spawn) - but this violates the GUI threading model (section 2.3.3). Consequently, only a help-first policy is viable for GUI applications.

- The implicit barrier at the end of the method means that the EDT cannot progress until all child tasks are completed. This means no other events can be handled, therefore reducing interactivity.

Most importantly, the `TASK` keyword in ParaTask serves as a form of documentation to ensure encapsulation is maintained for tasks. This is not so for Cilk++, since methods that call other tasks must themselves be annotated with the same keyword. As will be presented in section 6.2.1, the work-stealing of JCilk does not produce good performance for embarrassingly parallel code.

The languages mentioned so far do not make any special consideration for parallel GUI applications, which is vital for desktop applications. Java 1.6 introduced the SwingWorker class to assist programmers in developing responsive GUI applications. Every invocation of a task is enclosed in a SwingWorker object where programmers implement a `doInBackground()` method to be executed on a worker thread. When the worker thread completes, the EDT will execute the `done()` method. SwingWorker is only designed to be used once, therefore the programmer must create a new instance for each task invocation. This model is only applicable to the EDT (hence the name), whereas ParaTask is also generalised to be useful for non-GUI parallel applications.

Although SwingWorker is a simpler alternative to manually using threads, programmers unfortunately end up breaking encapsulation since they must know whether the code inside `doInBackground()` is thread-safe. ParaTask, on the other side, encourages encapsulation (and therefore code reuse) since the tasks are naturally documented as thread-safe. SwingWorker requires explicit calls (e.g. to start the worker) and provides no support for specifying dependences amongst tasks.

Foxtrot provides two solutions to avoid freezing the GUI. The first approach is the asynchronous solution, which is essentially identical to the SwingWorker discussed. In the second approach, the synchronous solution, code executes "sequentially" as it appears. In order not to block the EDT when it invokes a task, the EDT is re-routed to continue processing events. When the task completes, the EDT is again re-routed to resume executing where it left off. The benefit of such a model is that callback methods are not needed. Unfortunately this model is less intuitive to use, especially when multiple tasks are invoked. Foxtrot, like SwingWorker, only applies to the EDT.

In 2009, Apple released GCD which supports task parallelism. The major difference, compared to ParaTask and all the above related work, is that GCD is integrated into the operating system rather than at the application-level. This has the advantage that the operating system manages the thread pool and task execution across a number of applications. Programmers enqueue tasks as functions or blocks (a non-standard extension developed by Apple for C, C++ and Objective-C to create closures). This is similar to some of the above approaches, where tasks are queued to various `dispatch` queues. Unfortunately such a syntax breaks encapsulation, requires developers specify which queue to send tasks to and involves tangling parallelisation concern with business logic:

```
dispatch_queue_t queue;
queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

int taskInput = ...;  // input to the task
```

```
// option 1: submit task as block
dispatch_async(queue,^{
  myTask(taskInput);
});

// option 2: submit task as function
//   requires void* type as function input
dispatch_async_f(queue, (void *) taskInput, myTask);
```

Chapter 6

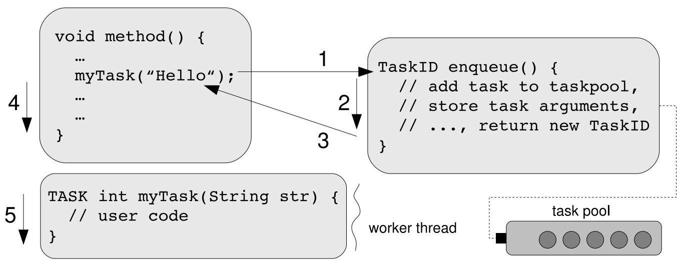
# Parallel Task implementation and performance

This chapter discusses ParaTask's implementation and performance. The full ParaTask grammar may be found in appendix A. This chapter also presents an example application developed using ParaTask, as well as a discussion on the combination of ParaTask with the Parallel Iterator.

## 6.1 Implementation

Figure 6.1 presents an overview of a task invocation. When a task is created, this is placed on a queue that is being monitored by a pool of worker threads (section 5.1.1). When the enqueuing thread calls the task, this is essentially translated into an enqueue of the task (step 1). The enqueuing of a task to the taskpool (step 2) involves the following:

- creating a new `TaskID` object
- storing any task arguments
- recording the enqueuing thread (necessary for `notify` and `asyncCatch` clauses)
- recording the enclosing task (necessary for propagating unhandled asynchronous exceptions up the task-call stack)
- registering any methods to be called in `notify` and/or `asyncCatch` clauses



```
void method() {
  …
  myTask("Hello");
  …
  …
}
```

```
TaskID enqueue() {
  // add task to taskpool,
  // store task arguments,
  // ..., return new TaskID
}
```

```
TASK int myTask(String str) {
  // user code
}
```

Figure 6.1: When a task is called, this results in an asynchronous execution since the enqueuing thread only *enqueues* the task to the taskpool. This involves recording all the necessary information that is later required for a worker thread to then *execute* the task.

When these steps have been completed, the new `TaskID` is returned (step 3) and the caller continues execution (step 4). Meanwhile, a worker thread will eventually execute the newly created task (step 5).

### 6.1.1 ParaTask source-to-source compiler

ParaTask applications are essentially standard Java applications with the additional use of a few keywords (e.g. `TASK`, `dependsOn`, `asyncCatch`, `notify`). Figure 6.2 shows how programmers develop ParaTask source files (`.ptjava` extension), which are then parsed by the ParaTask compiler into standard Java source files (`.java` extension). The ParaTask compiler performs one-to-one preprocessing, namely an equivalent `x.java` Java source file is produced for every `x.ptjava` ParaTask source file. Finally, all Java source files are compiled using a standard Java compiler.

#### The token manager and parser

The parser is generated using JavaCC (Java Compiler Compiler) [93], a popular parser generator for use with Java applications, originally developed by Sun Microsystems. To support ParaTask keywords, the stable and official Java 1.5 grammar released with JavaCC is extended; appendix A presents the modified sections of this grammar to support ParaTask. Consequently, this grammar is stable to parse all Java code up to Java 1.5.

Taking the grammar in appendix A as input, JavaCC produces some Java source files. Of particular interest are the token manager and parser:

- `ParaTaskTokenManager.java` (the token manager)

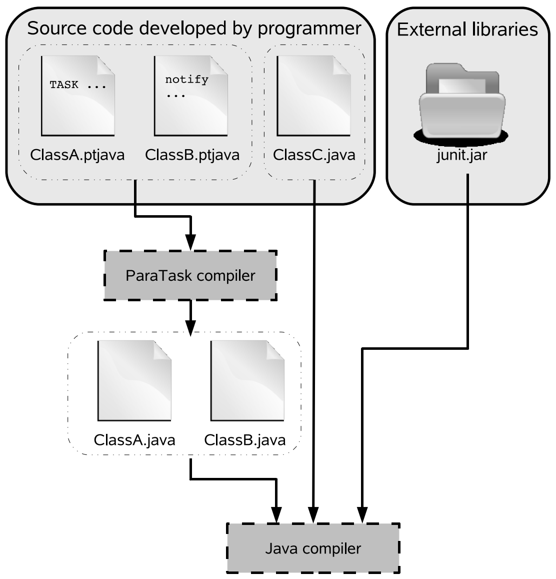


Figure 6.2: Developing ParaTask source code is just like Java source code, except the source files are preprocessed by the ParaTask compiler before the Java compiler.

A lexical analyser that analyses the input stream into meaningful tokens (e.g. “int”, “(”, “main”, “dependsOn” ).

- `ParaTaskParser.java` (the parser)
  The structure of the tokens produced by the token manager are analysed by the parser, producing an abstract syntax tree (AST).

Once these files (the parser, etc.) have been produced, input streams may now be tested to ensure they conform to the ParaTask grammar of appendix A:

```
CompilationUnit ast =
  ParaTaskParser.parse(new File(‘‘/home/user/ImageApplication.ptjava’’));
```

If the input stream does not conform, the parser reports an exception and ends. For example, assume the programmer accidentally types `dependOn` instead of `dependsOn`:

```
********* Failed to parse ImageApplication.ptjava
japa.parser.ParseException: Encountered "dependOn" at line 7, column 36.
Was expecting one of:
```

```
"dependsOn" ...
";" ...
...
```

### The tree visitor

If the input stream is correct, then an AST is produced (i.e. `CompilationUnit`) by the parser. Since this AST contains ParaTask-specific nodes, it is visited to replace them with standard Java code. This is accomplished with the `TaskVisitor`, which is based on JavaCC's default `DumpVisitor` (it essentially just reproduces the input stream from the AST).

```
TaskVisitor visitor = new TaskVisitor();
ast.accept(visitor);  // TaskVisitor traverses AST
String fileContent = visitor.getSource(); // get output produced by visitor
printContentsToFile(fileContent); // create a corresponding ".java" file
```

Below is a small extract from the `TaskVisitor` showing, as an example, how a task declaration node (see grammar in appendix A) is visited. An example of the output produced by the visitor follows in section 6.1.1.1.

```
public final class TaskVisitor {
  // the output is added to this printer
  private SourcePrinter printer = new SourcePrinter();
  ...
  public void visit(TaskDeclaration td) {
    // get the method
    MethodDeclaration method = td.getMethodDeclaration();
    int modifiers = method.getModifiers(); // e.g. public static
    printer.print(Modifier.toString(modifiers));
    printer.print(" ");
    if (td.isMultiTask())  // if multi-task, return TaskIDGroup
      printer.print("TaskIDGroup<");
    else
      printer.print("TaskID<");
    Type retType = method.getReturnType();  // original return type
    printer.print(retType);
    printer.print("> ");
    printer.print(method.getName()); // print method name
    printer.print(“(”);  // start parameter list
    ...
    // enqueue to the task pool
    printer.print("return TaskpoolFactory.getTaskpool().enqueue");
    ...
  }
  public void visit(LineComment n) {
    printer.print("//");
    printer.printLn(n.getContent());
  }
  ...
}
```

#### 6.1.1.1 Parsing task declarations

Consider a ParaTask source file with the following declaration for a one-off task:

```
TASK public int myTask(String str) {
  /* user-code */
}
```

The `TaskVisitor` translates the above into standard Java code:

```
private Method _pt_myTask_String = null;

public TaskID<Integer> myTask(String str) {
  return myTask(str, null);
}

public TaskID<Integer> myTask(String str, TaskInfo taskinfo) {
  if (_pt_myTask_String not initialised) {
    ...
    // parameter types for the task
    Class[] params = new Class[] { String.class };
    // the class containing this task
    Class currentClass = getClass();
    // get Method using Java reflection
    _pt_myTask_String = currentClass.getDeclaredMethod(‘‘_pt_myTask’’, params);
  }
12:     Object[] args = new Object[] {str};
13:     if (taskinfo == null)
14:       taskinfo = new TaskInfo();
15:     taskinfo.setMethod(_pt_myTask_String);
16:     taskinfo.setTaskArgs(args);
17:     taskinfo.setEnqueuingThread(Thread.currentThread());
18:     return Taskpool.enqueueOneOff(taskinfo);
19:   }

20:   public int _pt_myTask(String str) {
21:     /* user-code */
22:   }
```

The first thing to notice is that the original user-code has been moved into another method whose name has been changed (lines 20 to 22); the reflected `Method` variable of line 1 refers to this method that contains the user-code. Therefore, the name of the original task now refers to new methods that perform the enqueuing of the task. The first method, declared on line 2, is used in the case that no ParaTask clauses, such as `dependsOn` etc., are used when the task is called.

In order to delay execution of the task (since it is being enqueued), reflection is used to retrieve the `Method` representing the user-code (lines 7 to 10). This requires the `Class` that the task is contained within (line 9), as well as the signature of the method, including the name and parameter types (line 8). With this information, the method is accessed (line 10). To reduce runtime overhead, ParaTask ensures this step is performed at most once for every task (line 6).

The `TaskInfo` object is used to save details necessary to invoking the task. This includes the task arguments (line 12), the actual user-code to execute, as well as recording the enqueuing thread. This is finally sent to the taskpool (line 18), which returns a `TaskID` after enqueuing the task. Parsing multi-tasks and interactive tasks is similar. There is only a slight difference: rather than calling `enqueueOneOff` on line 18, `enqueueMulti` or `enqueueInteractive` is called respectively. Therefore, the ParaTask runtime system enqueues the task appropriately.

#### 6.1.1.2 Parsing task invocations

If a task is invoked without specifying any ParaTask clauses, then the ParaTask compiler does not modify the invocation. This is because a task invocation is a valid method invocation from the Java compiler's point of view; the invocation of the task refers to the enqueuing of the task rather than the original user-code (line 2 of section 6.1.1.1). Consider now a task invocation making use of at least one ParaTask clause:

`TaskID myID = myTask(‘‘Hello’’)` *(ParaTask clause)+*`;`

Since at least one ParaTask clause is specified at the task invocation, the ParaTask compiler will create a new `TaskInfo` object:

```
TaskInfo _pt_myID = new TaskInfo();
```

The `TaskInfo` is then populated with information regarding the respective ParaTask clause. Once all this information is stored, the final task invocation becomes:

```
TaskID myID = myTask(‘‘Hello’’, _pt_myID);
```

Since the `TaskInfo` now contains all the necessary information required to invoke the task, the original ParaTask clauses are removed. The above call refers to line 5 of section 6.1.1.1. How the `TaskInfo` is populated for the respective clauses is now discussed.

**Parsing a `dependsOn` clause**

Consider the following use of the `dependsOn` clause:

```
TaskID myID = myTask(‘‘Hello’’) dependsOn(id1);
```

Parsing a `dependsOn` clause is very simple. The ParaTask compiler only needs to produce the following:

```
_pt_myID.addDependency(id1);
```

If multiple dependences are specified within the same `dependsOn` clause, the above is repeated for each dependency. If the programmer misspelled the variable name of a dependency (e.g. wrote `it1` instead of `id1`), then the resulting Java code will not compile (assuming `it1` does not refer to another `TaskID` instance within scope).

**Parsing a `notify` or `notifyGUI` clause**

The `notify` and `notifyGUI` clauses have a few components that need careful consideration:

```
TaskID myID = myTask(‘‘Hello’’)
          notifyGUI(update(TaskID)) notify(myObj::complete());
```

Note that multiple methods may be specified within each of the `notify` or `notifyGUI` clauses. The first method in the `notifyGUI` clause above, `update`, has a `TaskID` parameter and must be invoked by the EDT (since it is `notifyGUI`) on `this` object instance (the default if no instance is specified). The second method in the `notify` clause, `complete`, has no parameters but will be invoked on the instance `myObj` by the enqueuing thread (determined in section 6.1.1.1). In either case, ParaTask must enforce that both these methods are accessible from the current scope. Below is the resulting code:

```
     // notifyGUI(update(TaskID))
1:   Method _pt_myID_notify1 = ParaTaskInternal.getDeclaredMethod(getClass(),
                               ‘‘update’’, new Class[] {TaskID.class});
2:   if (false) // never executed - compiler to check syntax
3:     update(ParaTaskInternal.dummyTaskID);
4:   _pt_myID.addNotify(_pt_myID_notify1, this, true);

     // notify(myObj::complete())
5:   Method _pt_myID_notify2 = ParaTaskInternal.getDeclaredMethod(myObj.getClass(),
                               ‘‘complete’’, new Class[] {});
6:   if (false) // never executed - compiler to check syntax
7:     myObj.complete();
8:   _pt_myID.addNotify(_pt_myID_notify2, myObj, false);
```

The `ParaTaskInternal.getDeclaredMethod` on line 1 is a helper function that internally uses Java reflection to retrieve the method. Note that in some cases, the method might actually be inherited from a super-class. The above example assumes that the enclosing method is an instance method (since it is making use of `getClass()`). If the ParaTask compiler determines the enclosing method is static, then the class is attained from the class execution stack (from a Java `SecurityManager`).

Unfortunately, line 1 can only determine at runtime if the programmer correctly used the `notify` clause (e.g. "update" is spelt correctly and the correct parameter types are specified). Therefore, the purpose of lines 2 and 3 is solely to ensure correct usage of the `notify` clause. Line 3 is never executed, but is used for a compile-time check. If the programmer misspelled the method name, attempted to invoke a method with incorrect parameter type, or if the method is out of scope (e.g. declared private in a super-class), then the Java compiler will complain. The ParaTask compiler simplifies its duties by using the Java compiler to determine such problems.

Finally, the method to `notify` is registered inside the `TaskInfo` instance (line 4). If the programmer does not specify an instance to invoke the method on, the default is `this` instance (or `null` if the enclosing method is a static method). The `true` refers to whether the method should be *forced* to execute on the EDT: this is `true` for `notifyGUI` clauses and `false` for `notify` clauses. The second method is determined in a similar way (lines 5 to 8). The differences include finding the method within the class of `myObj` (rather than using the current class). The `notifyInterim` and `notifyInterimGUI` clauses are parsed similarly to the `notify` and `notifyGUI` clauses.

### Parsing a `asyncCatch` clause

The `asyncCatch` also has a few noteworthy aspects. In particular, ParaTask ensures that the programmer adheres to the Catch or Specify Requirement. Assume that the signature of a task declaration includes a `throws` clause, for example:

```
TASK public int myTask(String str) throws IOException {
  /* user-code */
}
```

This task declaration is parsed exactly as discussed in section 6.1.1.1: the `throws` clause in the method signature is also included in the output source code. Since `IOException` is a checked exception, then the programmer must now use the `asyncCatch` clause (the `asyncCatch` clause may also be used for general exception handling when invoking any task). Below is a corresponding task invocation written by the user:

```
TaskID myID = myTask(‘‘Hello’’) asyncCatch(IOException handler(TaskID));
```

By specifying an exception handler to catch the `IOException`, the following code is produced by the ParaTask compiler:

```
     // asyncCatch(IOException handler(TaskID))
1:   Method _pt_myID_exc1 = ParaTaskInternal.getDeclaredMethod(getClass(),
                                 ‘‘handler’’, new Class[] {TaskID.class});
2:   if (false)
3:     handler(ParaTaskInternal.dummyTaskID);
4:   _pt_myID.addExcHandler(IOException.class, _pt_myID_exc1, this, false);
5:   TaskID id = null;
6:   try {
7:     id = myTask(‘‘Hello’’, _pt_myID);
8:   } catch (IOException _pt_e) { /* dummy try/catch block */ }
```

Notice how lines 1 to 3 are essentially identical to that produced when parsing a `notify` clause: these lines are used to ensure that the method specified in the exception handler exist (as discussed above). Line 4 registers the exception type, `IOException`, with the exception handler so that the ParaTask runtime knows which handler to invoke when an `IOException` occurs. The actual task invocation is now reconstructed as in lines 5 to 8. First, the `TaskID` declaration (line 5) is separated from the assignment (line 7) so that the `TaskID` instance stays in the same scope the programmer expects. The assignment is finally surrounded with a try/catch block, catching the exceptions specified in the `asyncCatch` clause.

The try/catch block of lines 6 and 8 is actually a *dummy* try/catch: its purpose is solely to quiet the Java compiler (since a checked exception is thrown by `myTask`) to ensure that the programmer follows the Catch and Specify Requirement. Now assume that an `IOException` was in fact thrown by `myTask` at runtime. Regardless of where the enqueuing thread is when this exception is thrown (even if it passes line 8, after the try/catch block), the ParaTask runtime will catch the exception and invoke the asynchronous exception handler specified in the asyncCatch clause. If the programmer does not correctly use the `asyncCatch` clause to handle an `IOException`, then the resulting Java code will not compile because there will be no surrounding try/catch (lines 6 & 7).

### 6.1.2 ParaTask runtime system

The Java implementation currently consists of three possible scheduling policies for the runtime. The implementation of these policies are discussed below, and the programmer may decide which is more suitable for their particular application. The motivation behind allowing this choice is to show the flexibility of how the underlying runtime system is independent of the ParaTask language syntax. In fact, the current implementation allows for more scheduling policies [90] to be added as plugins. Figure 6.3 overviews the ParaTask runtime system, regardless of the particular scheduling scheme to be used.

An interesting question that arises is whether there should be a limit on the number of active interactive tasks? The motivation behind this is that an interactive task is essentially a thread, therefore creating a vast number of interactive tasks could potentially result in reduced performance [23, 81]. However, if ParaTask were to limit the number of active interactive tasks, then this would be equivalent to a threading library limiting the number of threads an application could make. Therefore, ParaTask simply provides a helper function, `ParaTask.activeInteractiveTaskCount()`, that returns the number of interactive tasks that are currently active. After querying this, the programmer decides on the necessary course of action.

**Task enqueuing** The first phase, naturally, starts when a task is enqueued by the enqueuing thread. During this phase, a `TaskID` is created in order to record the details of the task invocation. If the task `dependsOn` other tasks, it is stored with the **waiting tasks**. Otherwise, the task is ready to execute: one-off tasks and multi-tasks are enqueued according to the scheduling scheme plugin (and any sleeping **worker threads** are woken up), while interactive tasks execute on a new **interactive thread**. As soon as the task is enqueued, the enqueuing thread continues to process other work: in this example, the enqueuing thread returns to the event loop to process other events.

**Worker thread is ready** All tasks, except interactive tasks, are executed by worker threads. The worker thread continues to execute all tasks in its **private ready queue**. Once the private ready queue is empty, a task is taken from the scheduling scheme plugin. If the task is reserved for another worker thread (i.e. in the case of multi-tasks), it is queued to that worker thread's private ready queue. Otherwise the task is executed.

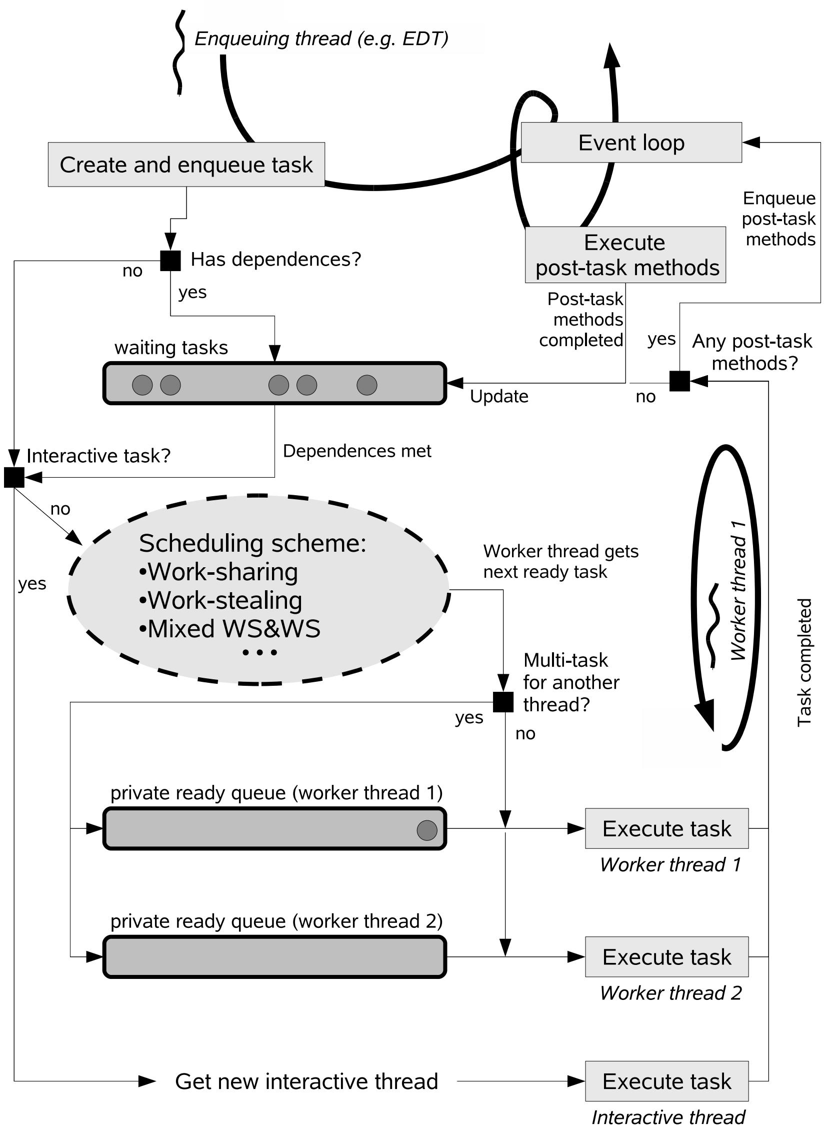


Figure 6.3: ParaTask's runtime system allows for different scheduling schemes to be plugged in. The life of a task starts when the enqueuing thread creates and deposits it for another thread to execute. The task is finally considered complete after any post-task methods (e.g. methods in a `notify` or `asyncCatch` clause) are executed.

**Task execution** Java reflection is used to execute the user-code of the tasks. In most cases, the worker thread will execute the task in its entirety before executing another task. The exception to this is if a worker thread blocks on the `TaskID` of another task that has not yet completed (e.g. task B) while it is currently executing task A. In this case, the worker thread retrieves another task (e.g. task C) from the scheduling scheme plugin and executes it. When task C is completed, the worker thread checks the status of task B. If it has completed, then the worker thread continues where it left off with task A; otherwise, another task (selected according to the scheduling policy) is executed again until task B is completed. This behaviour is also repeated recursively if necessary (e.g. if task C in turn blocks on another unfinished task).

**Task completion** When a task is finally executed by the worker thread (or interactive thread), it is not necessarily considered complete just yet. First, the worker thread checks to see if the task has any post-task methods that need to be executed (e.g. methods in a `notify` or `asyncCatch` clause). If no such methods exist, then the task is considered complete: the worker thread signals this by updating the task dependences.

If a task has post-task methods, then these need to be executed by the enqueuing thread (not the worker thread). Therefore, the worker thread signals the enqueuing thread (by emitting an event) that it should execute the respective post-task methods. When the enqueuing thread executes the post-task methods, a signal is sent back to update the task dependences. If an exception occurs, these are treated in the same manner as post-task methods since they are executing by the enqueuing thread.

#### 6.1.2.1 Work-sharing schedule

Implementing a work-sharing policy is straight forward. When tasks are enqueued to the scheduling scheme plugin, they are placed onto a **shared ready queue** using a first in, first (FIFO) out policy. As discussed in section 5.6.1, the tasks in this shared ready queue are ordered according to the original order they were enqueued (which is not necessarily the same as the order they became ready to execute).

#### 6.1.2.2 Work-stealing schedule

Rather than having a single shared ready queue, the work-stealing runtime consists of a **local deque** (a double ended queue) for each worker thread. Note that this local deque is distinct from the **private ready queue**: the private queue is used to store only multi-tasks reserved for the owner thread, while the local deque stores all the task types and these may be stolen by other threads.

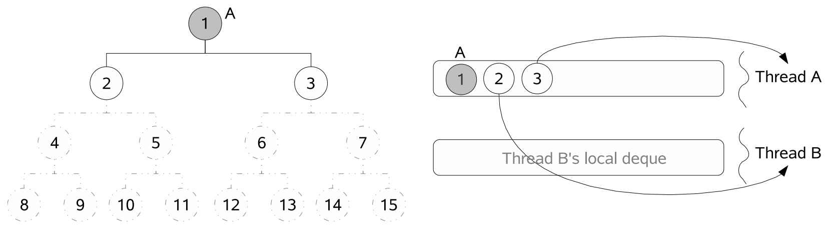
Although the work-stealing implemented for ParaTask is the same as that of the Tree Parallel Iterator (discussed in section 4.1.7), some aspects are repeated here in context of ParaTask. Figure 6.4 shows 3 stages of an application run, where 2 threads are executing the task graph of figure 5.7.

Each thread has a private deque to store tasks that are ready to execute. When a thread operates on tasks on its own deque, a LIFO policy is used (therefore operating on the *latest* local node). When a thread's private deque is empty, it becomes a *thief* and selects a *victim* thread at random. The thief attempts to steal the *oldest* task on the victim's deque, therefore using a FIFO policy when stealing.

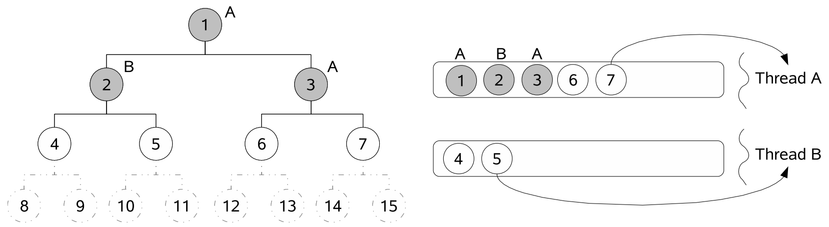
In the example application of figure 5.7, the first (root) task is enqueued. This task is placed on one of the thread's deque. In the example, thread $A$ happens to be the first thread to grab the task. In the meantime, thread $B$ is trying to steal from another random thread (in this case it only has one other thread to steal from). While thread $A$ is processing task 1, it enqueues 2 more tasks (i.e. 2 task invocations are made). These tasks are enqueued to thread $A$'s private deque (figure 6.4(a)). Consequently, tasks 2 and 3 are now ready to be executed.

Now that 2 new tasks have been enqueued to thread $A$'s deque, thread $B$ has found its victim: it steals the oldest task from thread $A$, which happens to be task 2. In the meantime, when thread $A$ blocks on the TaskID on an incomplete task, it takes the latest task from its local deque (node 3) and executes it. As thread $B$ executes task 2 (figure 6.4(b)), it enqueues 2 more tasks (tasks 4 and 5). It soon blocks on the TaskID of these new and incompleted tasks (while processing task 2), and therefore starts a new task. This time, thread $B$ does not need to perform another steal since it has unprocessed tasks on its local deque: it executes the most recent local task (task 5). Similarly, task 7 becomes thread $A$'s most recent local task when task 3 is completed.

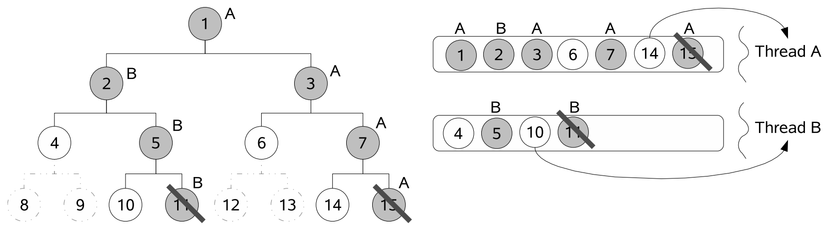
Finally, each thread completes a task that does not spawn another task (i.e. the leaf tasks in the task tree). For example, figure 6.4(c) shows thread $A$ having



(a) When task 1 executes, it creates tasks 2 & 3. Task 2 is stolen by thread B (FIFO), while thread A operates on task 3 (LIFO).



(b) With sufficient tasks on their own deque, each thread uses a LIFO policy. Note that tasks 1-3 still have not completed (recursive parallelism).



(c) Each thread has completed a leaf task in the recursive parallelism, and continue using a LIFO policy while their respective deque is not empty.

Figure 6.4: Example of 2 threads executing a recursive task-parallelism application using the work-stealing schedule. When a ParaTask worker thread blocks on the TaskID of an incomplete task, it executes another ready task from its own local deque (if not empty) using a LIFO policy. Otherwise, a FIFO policy is used when stealing tasks from another worker thread.

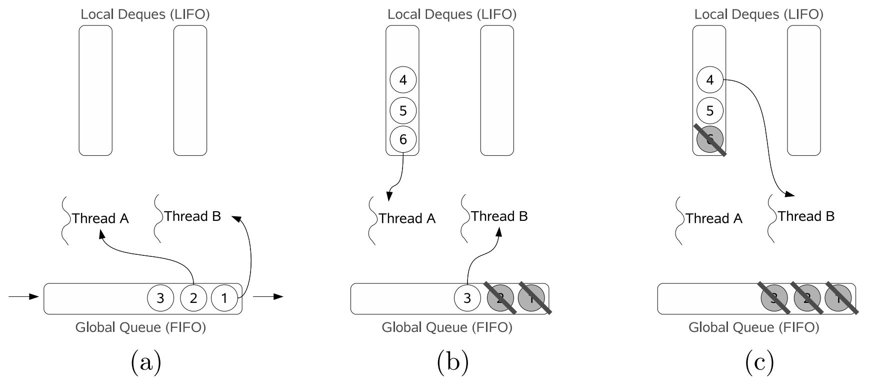


Figure 6.5: Mixed schedule implementation. In (a), both threads are work-sharing. In (b), thread A starts work-stealing due to nested parallelism, while thread B continues work-sharing. In (c), thread B starts work-stealing as no more tasks remain on the global work-sharing queue.

completed task 15. In this case, thread *A* returns to task 7, only to find it is still blocked (since task 14 has not be completed by another thread). In this case, it picks another task from its local deque. This happens to be task 14, which is then executed.

#### 6.1.2.3 Mixed schedule

Figure 6.5 illustrates the mixed work-sharing and work-stealing schedule. Figure 6.5(a) shows 3 tasks enqueued using a FIFO policy behaving much like the work-sharing policy of section 6.1.2.1. In this example, task 2 happens to create 3 more tasks (i.e. nested parallelism). Since work-sharing is unsuitable for nested parallelism, these tasks are processed using the LIFO work-stealing policy of section 6.1.2.2 (figure 6.5(b)). Consequently, thread *A* must temporarily compromise fairness in favour of executing the recursive parallelism. In the meantime, thread *B* strives for fairness by processing tasks from the global queue. When a thread finds no tasks in its local deque or the global queue, it steals a task from another thread (as discussed in section 6.1.2.2). For example, figure 6.5(c) shows thread *B* stealing the oldest task from thread *A*.

## 6.2 Performance

The performance of ParaTask compared to traditional parallelism approaches is evaluated, as well as the overhead relative to sequential code and Java threads.

In light of mainstream multi-cores, benchmarks typical of desktop applications are included as well as considering user-perceived performance. The benchmarks ran on a shared memory system which may be considered a typical future desktop platform running Linux. It has four Quad-Core Intel Xeon processors (total of 16 cores) running at 2.4GHz with 64GB of RAM. All benchmarks were coded in Java, and the sequential code of each benchmark forms as the baseline for all speedup calculations. The default JVM memory allocation was sufficient for most benchmarks, except for those of section 6.2.1.2 where 16GB was allocated. Throughout all the benchmarks, Java's default garbage collector was used.

### 6.2.1 Comparing to traditional Java parallelism approaches

This section compares the performance of a number of typical parallelisation approaches a programmer may take, including:

- ***JT-max***: as presented in sections 2.3.4 and 2.3.5, a new Java thread is created for *every* task. This is a typical approach programmers would take to manually parallelise an application.

- ***JT-min***: as a variation to JT-max, this involves creating the minimum number of threads in order to match the processor count. A static distribution of the tasks is created, where each thread is assigned roughly an equal number of tasks (but this might not necessarily equate to an equal workload at runtime if the tasks are unbalanced). Although this approach requires more work than JT-max, programmers may opt for this in hope to reduce runtime overhead.

- ***SwingWorker***: as presented in section 2.3.5, this involves wrapping each task in a `SwingWorker` object. Being a tasking model, SwingWorker is an improvement to the runtime overhead of the JT-max threading model.

- ***JCilk***: as presented in section 5.8, tasks are defined as methods and annotated with the `cilk` keyword. Although this model cannot be used for GUI applications, it is used in the following benchmarks to compare the work-first policy to ParaTask's help-first policy

- ***PT-sharing***: as presented in section 5, tasks are defined as methods and annotated with the `TASK` keyword. The runtime used is the work-sharing schedule discussed in section 5.6.1.

- ***PT-stealing***: as presented in section 5, tasks are defined as methods and annotated with the `TASK` keyword. The runtime used is the work-stealing schedule discussed in section 5.6.2.
- ***PT-mixed***: as presented in section 5, tasks are defined as methods and annotated with the `TASK` keyword. The runtime used is the mixed schedule discussed in section 5.6.3.

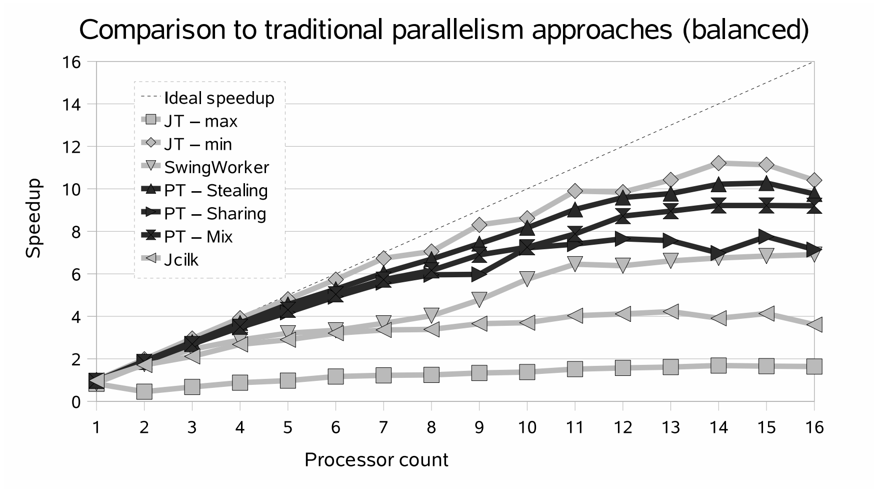
Note that the 3 ParaTask approaches above all used exactly the same coding approach. The only difference is the scheduling selected at runtime (the default is PT-mixed, therefore its performance is obtained without tweaking).

#### 6.2.1.1 Compute-intensive applications

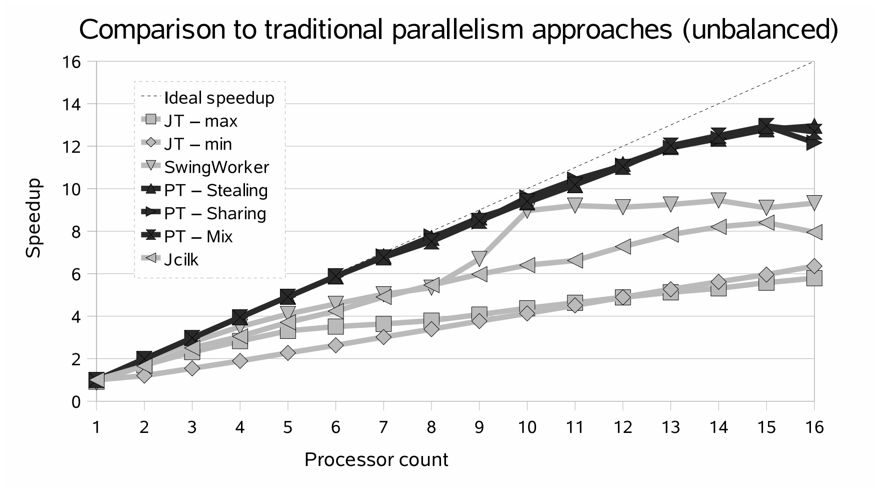
This section aims to understand the overhead and load balancing of the various approaches. The first benchmark computes a synthetic load (here the Newton-Raphson method) for each task. Figure 6.6(a) shows the speedup using a medium-grained and balanced workload (each task takes an average 0.3ms). Using Java threads achieves the best performance when statically allocating computations to the minimum number of threads (JT-min). If a thread is assigned for each computation (JT-max), this achieves the worst performance.

ParaTask achieves better performance using work-stealing over work-sharing as the processor count increases. Even though there is no nested parallelism in this benchmark, the reason that PT-stealing performs better than PT-sharing is because of the high contention. PT-sharing consists of a single queue for the tasks, therefore all threads contend on the same queue (they even contend at the same end of the queue). In PT-stealing however, tasks are randomly distributed to multiple queues (in the case that tasks were enqueue by a non-worker thread). This not only means less contention because of more queues, but also because stealing threads take from the opposite end that the victim thread operates on. ParaTask mixed scheduling incurs slightly higher overhead compared to ParaTask work-stealing because it checks for nested parallelism.

Figure 6.6(b) shows the speedup for an unbalanced workload. Due to this imbalance, JT-min is now one of the worst performers. ParaTask now performs best amongst the other approaches, and the difference between the work-stealing, mixed and work-sharing is less evident since task granularity has increased. These results show that the overhead of ParaTask is very small, while the workload is balanced well to produce consistent speedup between the different workloads. In both benchmarks, JCilk's reduced performance is attributed



(a) Fine-grained and balanced workload



(b) Unbalanced workload

Figure 6.6: Comparing to typical Java parallelisation techniques.

to the high cost of stealing in the work-first policy: a steal involves a new thread taking over the context (e.g. variables and their values) of the victim thread.

#### 6.2.1.2 Disk-intensive applications

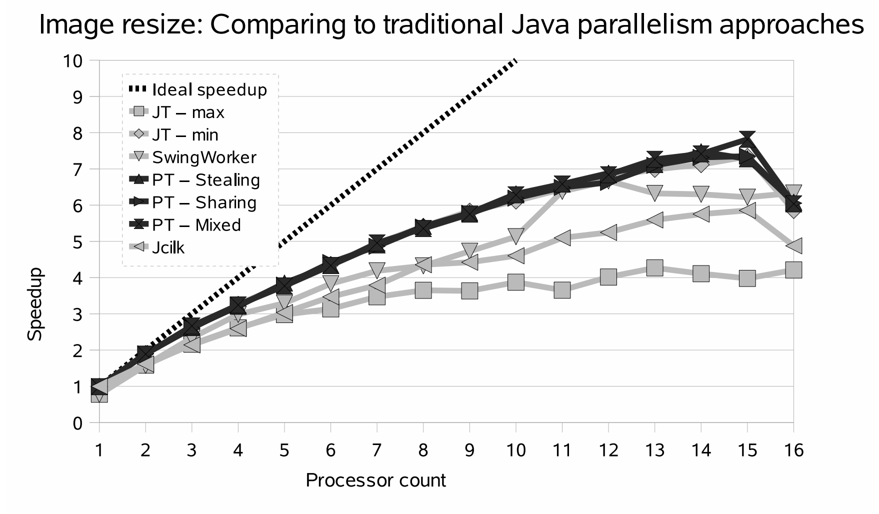
The scalability of the different approaches using more realistic desktop applications is now investigated, in particular those requiring high amounts of disk access. Figure 6.7(a) shows the speedup for an image-resizing application. This involved a collection of 256 identical images, each 1MB (1600×1200 pixels), all stored on the same disk. The resizing of each image was designated as a single task, and no image was reused for another task in order to reduce cache effects. The sequential time to resize all 256 images took an average of 585ms for each image.

Figure 6.7(b) shows the speedup for a word permutation application (for example, typical of a spell checker). This benchmark consisted of 3505 different text files, stored on the same disk within a total of 172 sub-folders. The size of the files ranged from 22 bytes to over 100 KB, each file was designated as a task (unbalanced workload) without reusing any file for another task. These files are in fact the Linux system's C language general-use include files stored under /usr/include. The sequential time to perform the permutation on all these files totaled 26 seconds. Each task consists of reading the words inside a file and performing various string comparisons with the other words within the same file.

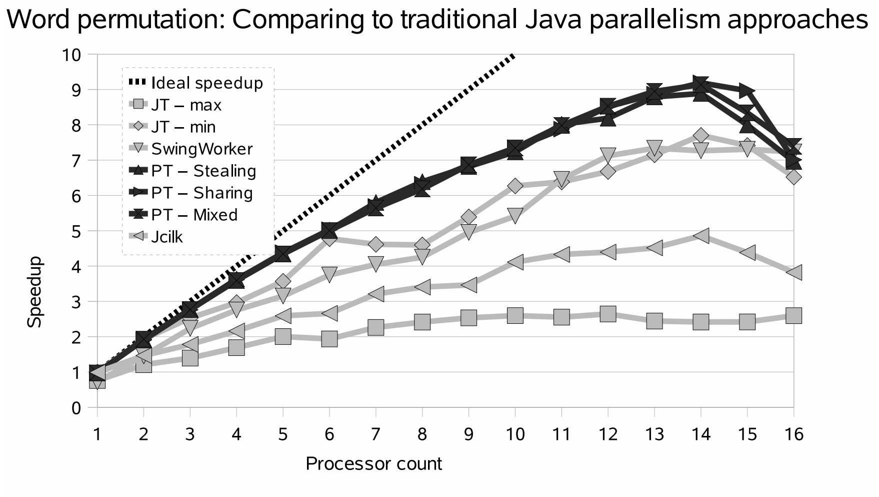
For these benchmarks, most approaches only scale up to around 14 processors due to the high disk contention. ParaTask allows the programmer to easily fine-tune the thread count for such a reason. In both benchmarks, the 3 ParaTask runtimes performed best. In the case of the image resizing benchmark, JT-min also performed well since the workload was balanced. However, this was not the case with the word permutation due to the unbalanced workload. JT-max continues to produce poor results, this time attributed to multiple disk-intensive tasks being active concurrently.

#### 6.2.1.3 Recursive applications

While ParaTask was dominating all previous benchmarks, a critical analysis must also reveal its weaker points. Fine-grained recursive benchmarks are where ParaTask is weaker. Interestingly, JCilk, which did not perform too well before, can shine here. The next benchmark, CountQueens, illustrates this. Given an



(a) Image resizing benchmark



(b) Word processing benchmark

Figure 6.7: Typical desktop applications making large amounts of disk access.

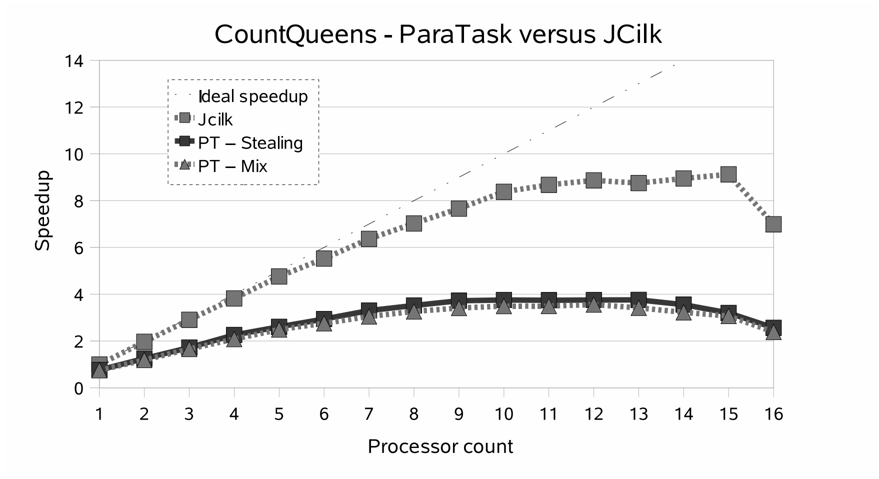


Figure 6.8: CountQueens benchmark, comparing JCilk to ParaTask.

$n \times n$ board, CountQueens finds all the possible solutions to the Queens problem [29]. Not all the approaches could handle this application: Java Threads quickly ran out of memory, SwingWorker resulted in deadlock (since all tasks blocked waiting for sub-tasks) and work-sharing resulted in stack overflow (recursion too deep). The only viable approaches were PT-stealing, PT-mixed and JCilk (all based on the same work-stealing).

Figure 6.8 shows the speedup for both JCilk and ParaTask, using a CountQueens board size of 15. JCilk outperforms ParaTask for this benchmark, even though both implement the same work-stealing [17] schedule. The difference is JCilk implements a *work-first* policy, while ParaTask implements a *help-first* policy [56]. In the work-first policy, threads leave their current task to sequentially execute a newly created task. Other threads may then steal from where the thread left. This performs well for extremely fine-grained nested parallelism (such as CountQueens) since the thread completes the task before another thread has a chance to steal (where stealing is expensive).

The reason that a work-first policy is not implemented for ParaTask is twofold. First, as shown in all the benchmarks above (except for fine-grained nested parallelism such as CountQueens), a work-first steal is extremely expensive as it requires transferring the context (state of variables) from the enqueuing thread to the thief thread. This expense greatly affects performance when many steals occur (i.e. tasks are not fine-grained enough, therefore other threads have the chance to steal).

Second, a responsive concurrent application requires that tasks are guaranteed to always execute asynchronously [97]. A work-first policy will produce synchronous tasks. In terms of desktop application semantics, a work-first policy violates the structure of multi-threaded GUI applications: this means the enqueuing thread (i.e. the GUI thread) executes the tasks and other worker threads would continue where the GUI thread left off (but only the EDT is allowed to access GUI components). Consequently, only a help-first policy is viable for GUI applications.

These results show that ParaTask executes recursive applications well, but only for higher granularity. The CountQueens benchmark may be considered a worst-case for help-first scheduling policies, but a best-case for work-first policies. Consequently, ParaTask would perform better for higher grained applications.

### 6.2.2 User-perceived performance

In section 6.2.1, it was shown that the work-stealing tends to outperform work-sharing as the thread count increases. Therefore, for batch-type applications, the work-stealing would be most useful in reducing wall-clock time. But what about an interactive environment? Due to the interactive nature of desktops, it is generally agreed that the users perception of performance becomes a vital metric in measuring performance [89, 44, 42]. For example, a web-based application addressing multiple users or a desktop application where a single user launches multiple tasks. In such a situation, users expect tasks to be treated fairly: in particular, using a FIFO policy.

The aim of the following benchmark was to quantify how much the actual order deviates from the expected order. In particular, how the deviation is effected as the number of tasks increases. Therefore, the benchmark is interested in the *order* that the tasks are completed in, and not the actual computational time. The tasks used had identical runtimes and involved computing a synthetic load (the Newton-Raphson method). The following table is used as an example to explain the process in calculating the deviation:

| Task number (i.e. enqueue order) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Expected finishing order: $e$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 16 |
| Actual finishing order: $a$ | 3 | 8 | 2 | 1 | 7 | 4 | 9 | ... | 11 |
| Difference: $e - a$ | -2 | -6 | 1 | 3 | -2 | 2 | -1 | ... | 5 |
| Difference$^2$: $(e-a)^2$ | 4 | 36 | 1 | 9 | 4 | 4 | 1 | ... | 25 |
| Sum: $\sum(e-a)^2$ | 218 | | | | | | | | |
| Deviation: $\sqrt{\frac{1}{N}\sum(e-a)^2}$ | 3.69 | | | | | | | | |

This example dataset shows how the deviation would be calculated for 16 tasks in a single run of the benchmark (number of tasks, N = 16). The 1st row lists the tasks and the order they were queued in. Naturally, a user would expect these tasks to finish in this very same order (2nd row). Unfortunately the tasks are likely to finish in a different order at runtime, for example the 3rd row. The deviation of the actual finishing order from the expected finishing order is calculated similarly to the standard deviation. The differences (4th row) are squared (5th row) and summed (6th row), and finally the average is computed (last row). This whole process is repeated for a total of 5 times, and the average is plotted as the deviation of finishing order for 16 tasks. This is then repeated for 32, 64, ... and 65536 tasks for each of the scheduling schemes.

The result is shown in figure 6.9. It shows how the *actual* finishing order deviates from this expected FIFO order as more tasks are introduced (the thread count is fixed at 16). This shows that work-sharing best models the user's expectation of the ordering, consequently resulting in better perceived performance. Of particular interest is the mixed scheduling policy, which achieves similar user-perceived performance of the work-sharing. This result, along with those of section 6.2.1, shows that the ParaTask mixed scheduling allows us to achieve the advantages of work-sharing and work-stealing into the same policy.

### 6.2.3 Overhead

In section 6.2.1, the performance of ParaTask was discussed in relation to other parallelism approaches. ParaTask's performance is now discussed compared to sequential code and Java threads. In particular, one wants to know how much slower are ParaTask tasks compared to sequential methods and how much faster compared to Java threads. The graphs in figure 6.10 show that the speedup factor depends on the number of tasks created and their granularity. All the tasks again involved a balanced synthetic load. One would expect a task be

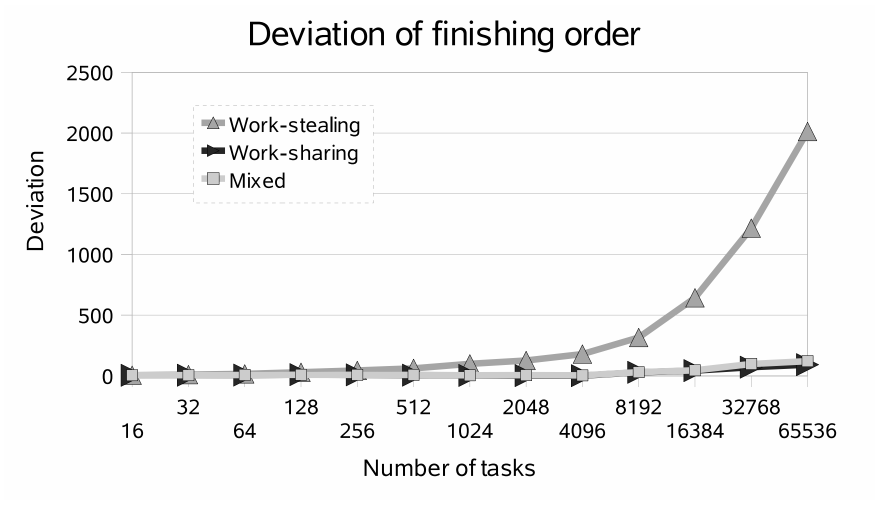


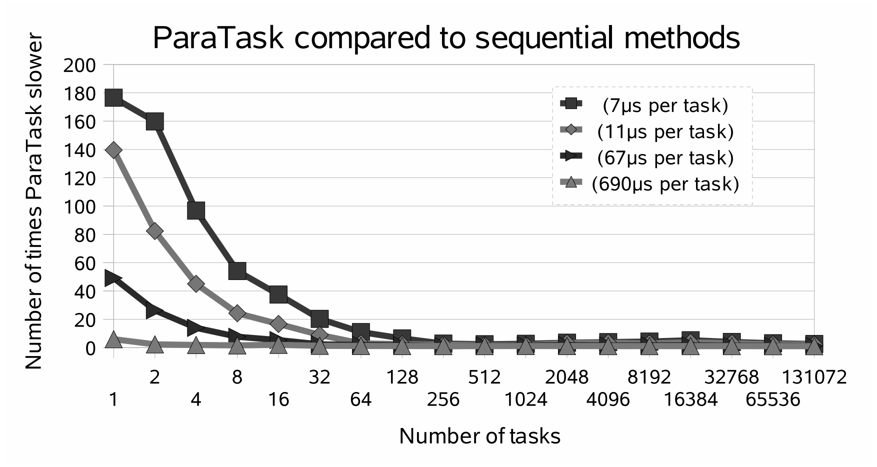
Figure 6.9: Deviation of actual finishing order compared to the expected finishing order. The higher the deviation, the lower the perceived performance from the user's point of view.

several times slower than a method invocation because of the task overhead, especially for fine-grained computations.

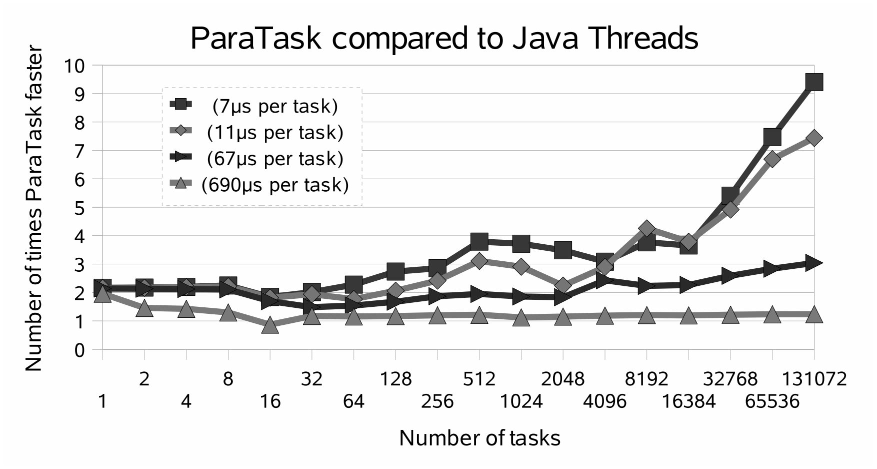
Figure 6.10(a) shows that ParaTask averages out to be around 4 times slower than sequential methods for extremely fine-grained tasks (each task runtime averages to $7\mu s$), 1.5 times slower for medium-grained tasks (task average of $67\mu s$), and 1.1 times slower for coarse-grained tasks (task average of $690\mu s$). This result is encouraging since these tasks are all considered extremely fine-grained for an interactive environment (at least 100ms of visual experience is needed before humans perceive duration [41]). Figure 6.10(b) confirms that the tasking model better scales than the threading model as the number of computations increase. This is especially evident for fine-grained tasks.

## 6.3 Example application: ParaImage

In addition to the benchmarks discussed in section 6.2, ParaTask has been applied to other applications. This section presents ParaImage, an image manipulation application co-developed by the author and Peter Nicolau (a 2nd year Software Engineering undergraduate student). What is most encouraging about this experience is that Peter was previously never exposed to parallel



(a) Comparing to sequential methods



(b) Comparing to Java threads

Figure 6.10: Comparing ParaTask to sequential methods and Java threads.

computing, let alone ParaTask. In particular, he quickly grasped ParaTask just by referring to previous ParaTask publications [50, 52] and following the code of the example application. This section first presents the application features before discussing the underlying implementation.

Even for sequential applications, the programmer would want to make it responsive. When doing this with ParaTask, the parallelisation is virtually free. If the programmer uses threads to increase the responsiveness of a sequential application (i.e. figure 2.2), not only is this more involved, but it does not help with parallelisation. Yet, the same thought process and restructuring is necessary.

### 6.3.1 ParaImage features

At the bottom of the ParaImage application, there is a panel displaying the current mode of the application. Users of the application may at any time switch between the sequential mode (red bottom panel, for example figure 6.11) or the parallel mode developed purely using ParaTask (green bottom panel, for example figure 6.12). This helps desktop users to encounter first-hand why parallel computing is so important for an interactive desktop experience. Please note that the sequential mode in fact refers to the situation of figure 2.1. The options available to the user are to create a new project from the file menu. Current projects include a Flickr image search or an image editing project, both discussed in detail below. To assist desktop users in experiencing the difference between a sequential and a multi-threaded application (developed using ParaTask), the bottom panel allows them to easily switch between the two modes.

#### Flickr search

The first project provides the user with a search interface to retrieve images from Flickr (a website for sharing images). Figure 6.11 shows a new Flickr search project with the sequential mode selected. On the top panel, users enter their search criteria. This includes the search query (1) and the number of images to retrieve per page (5). Once a search has been submitted (2), the progress (4) is displayed and the cancel button (3) is enabled. Users may also retrieve the previous (6) and next (8) set of results.

When the user submits the search query, thumbnails of the results appear in the centre (9). If the sequential mode is enabled, then users will have an

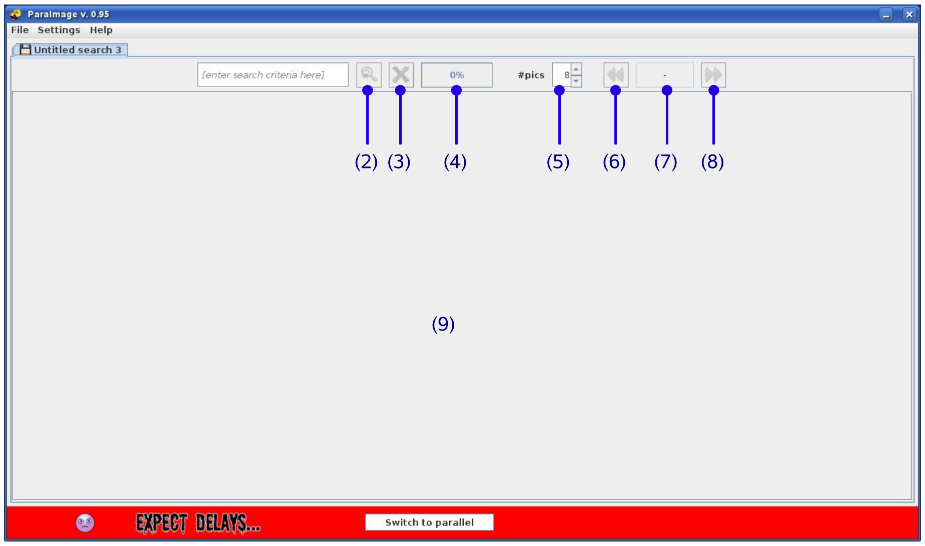
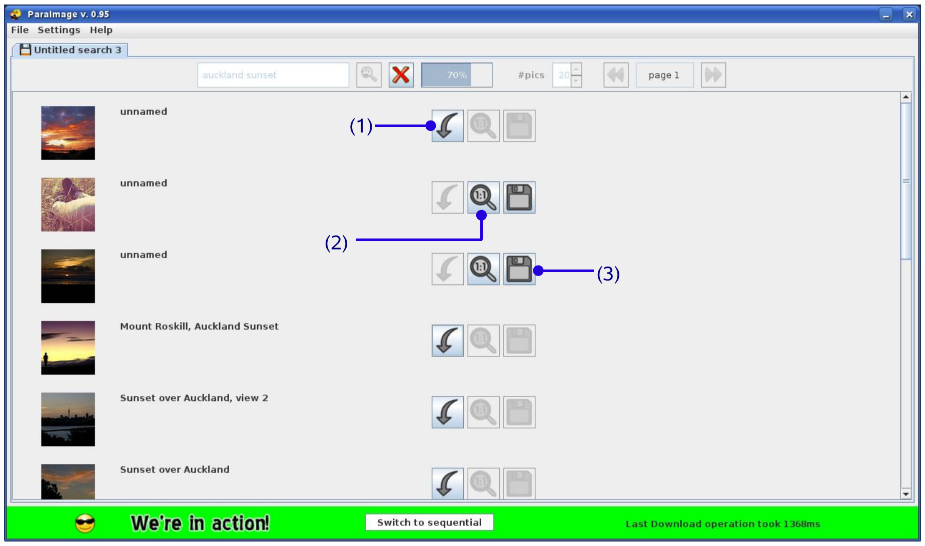


Figure 6.11: ParaImage with an empty Flickr search project. This interface is the same for both the sequential and parallel modes.

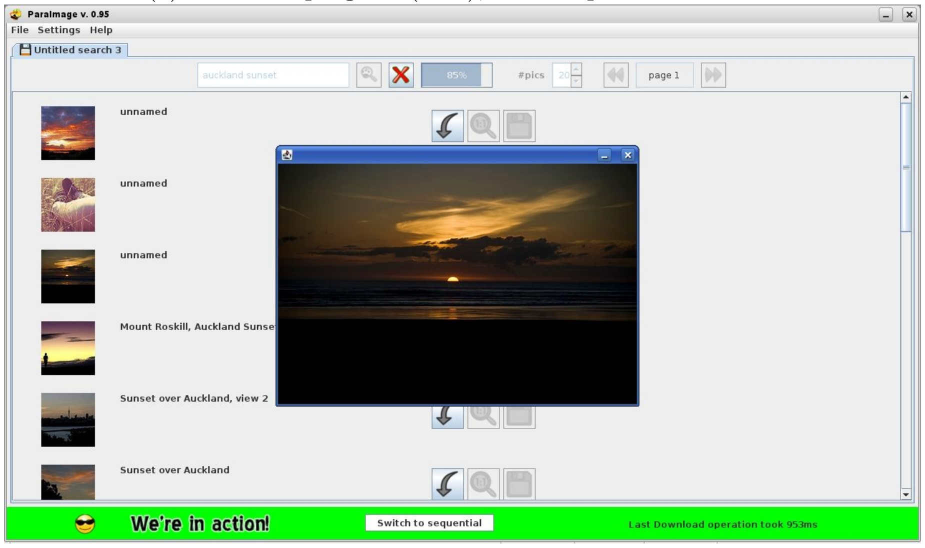
unresponsive application until *all* the results are being retrieved. During this time, none of the buttons or menus will respond. Not even the cancel button will respond in the sequential mode while a search is taking place (since the EDT thread is performing the search). Also negative for the user's experience is that the progress bar does not update until 100% complete.

Figure 6.12 shows an example of the search interface, only this time using the parallel mode. The first difference that users will notice while using the parallel mode is the prompt arrival of intermediate results. When a search is submitted, partial results (in the form of thumbnails) will be displayed as they become available. Figure 6.12(a) shows the stage where 70% of the search has already completed. Notice that the cancel button is enabled: pressing it will cancel the retrieval of any more thumbnails.

The next difference that the user will notice is the high responsiveness of the application. For example, even while the thumbnails are still arriving, the user may download the full size of selected images by pressing (1). After the full size has been retrieved from the server, it may be viewed (2) or saved to disk (3). Figure 6.12(b) shows this responsiveness: the user has entered a search criteria, retrieved some full size images and viewed a full sized image all while only 85% of the search has been completed. In the sequential mode, the user would not



(a) Search in progress (70%), with responsive buttons.



(b) Full size image retrieved and displayed while search still in progress (85%).

Figure 6.12: While in parallel mode, the application remains fully responsive. Not only does this mean an updating progress bar, but thumbnail results are displayed as they come available. Users may also retrieve and save full-size images while the search is still in progress.

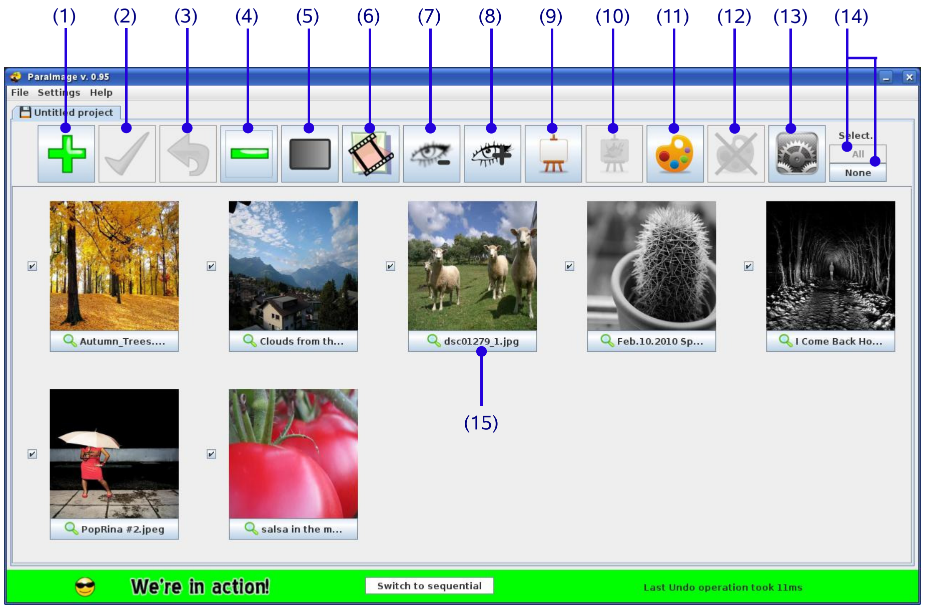


Figure 6.13: ParaImage with an image editing project.

have even seen a single thumbnail during this time.

This responsiveness is not limited to only within a single search project: it is an application-wide responsiveness. For example, the user may decide to create a new search project (which will appear as a new tab in the main screen). Even a new non-search project, such as an image editing project (discussed next), may also be opened. The entire application will remain interactive while the different projects perform their respective operations (and of course, the parallelisation is also improving the wall-clock performance).

### Image editing

The next project that a user may create involves performing various filters on a set of images (figure 6.13). After the user adds images to the project (1), a thumbnail preview appears in the centre, with the option to view the full size (15). The various filters that can be performed include edge detection (5), colour invert (6), blur (7), sharpen (8), mosaic (9) and image mosaic (10). The other options include mosaic settings (13), creating (11) and deleting (12) palettes, saving changes (2), undoing changes (3), removing images from view (4) and shortcuts to selecting/deselecting all images (14).

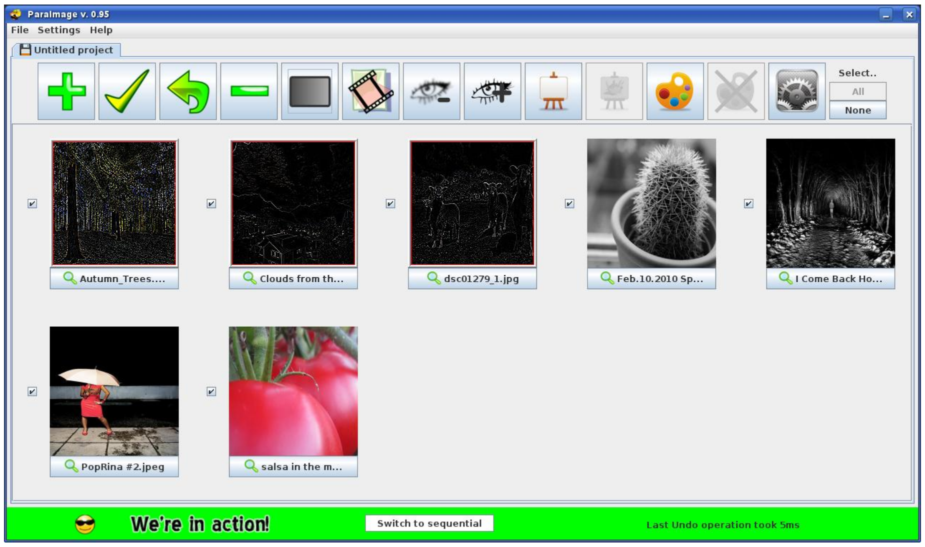
Figure 6.14(a) shows the edge detection filter being applied to the set of images. At this stage, the filter has been applied to the first 3 images. Since the application is running in the responsive parallel mode, the user may enqueue more filters as the current filters are being applied. Figure 6.14(b) shows the colour invert filter being applied. As the user would expect, the filters on each image are applied progressively in the order enqueued. For example, the colour invert is applied to the edge detected image (not to the original image).

In the case that sequential mode was selected, such a project will firstly be more time consuming since only one thread (i.e. the EDT) will be performing all the filters. Secondly, a new filter cannot be enqueued until the current filter is finished. This is because the GUI becomes unresponsive during the time filters are being applied. Finally, as in the case of the Flickr search, intermediate results are not displayed. Instead, the user will be faced with an unresponsive application until the current filter finishes for all the selected images.

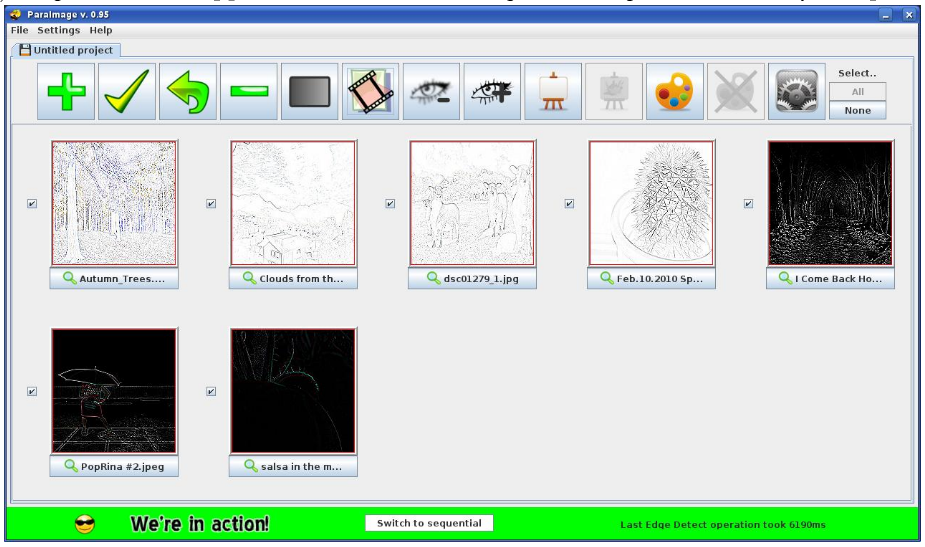
A particularly interesting filter is the image mosaic (filter (10) of figure 6.13). The user first selects a set of images that will be stored in a palette (11). The images in this palette are used as small tiles of the mosaic. Figure 6.15(b) shows an example of such an image mosaic. The tiles of this mosaic consist of the images in the palette of figure 6.13. For example, the red aspects of the lady's dress in figure 6.15(b) would be composed from the tomato image of figure 6.13.

An important aspect for such an application is when multiple filters are applied. For example, consider the blurred image of figure 6.15(c). To achieve this desired level of blurness, the blur filter was repeatedly applied (5 times) since applying it once produces only a slight blur effect. In the sequential mode, the user must wait after each time the filter is applied since the EDT is occupied with the filter.

The interesting part of this scenario is in the parallel mode. In this case, due to the responsiveness, the user manages to press the blur filter 5 times before the very first blur is even completed. The user would expect that each of the 5 blur actions are accumulated (rather than just being applied to the original image). Note that an intermediate blur (necessary for the subsequent blur) would not have been available at the time the blurs were queued. Section 6.3.2 will discuss how this was easily achieved using ParaTask.

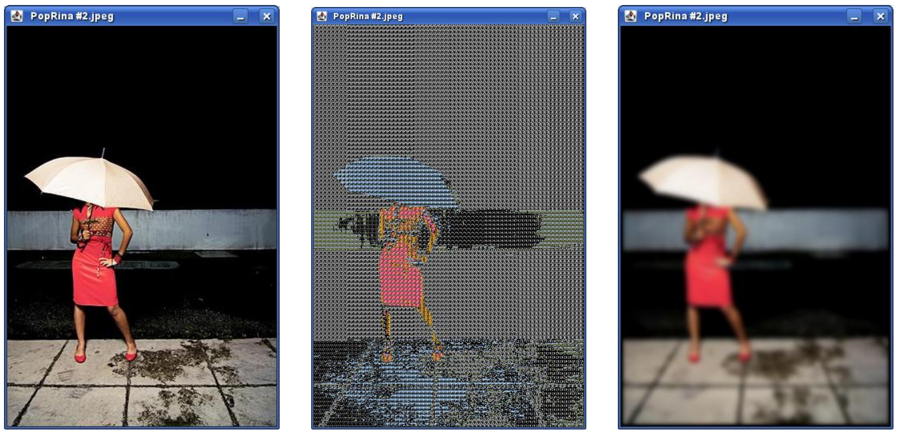


(a) Edge detection applied to all selected images. 3 images have already completed.



(b) Colour invert immediately follows the above edge detection.

Figure 6.14: While in parallel mode, the application the user is able to continue enqueuing multiple filters. The application remains responsive, updating the thumbnails as filters are applied.



(a) Original image (b) Image mosaic filter (c) Blur filter

Figure 6.15: Example of an image mosaic and blur filter. The image mosaic is composed from a set of tiles, each tile an image from the palette in figure 6.13.

### 6.3.2 ParaImage implementation

This section discusses the implementation of the ParaImage features presented in section 6.3.1. The application was first developed as a sequential program by concentrating on the general framework without parallel computing in mind. Once the (sequential) functionality was developed, implementing the parallel mode was simple. Particular implementation aspects of interest are discussed below.

#### Code reuse

The code snippet below shows an example of the sequential code to perform a filter on a photo:

```
public static Image edgeDetect(Image i) {

    ...

}
```

In order to implement the parallel mode for this functionality, another method is created and annotated with the `TASK` keyword. The elegance here is evident as the business logic of the sequential code is reused:

```
TASK public static Image edgeDetectTask(Image i) {
    return edgeDetect(i);

}
```

The event handler uses the above code as follows:

```
if (sequentialModeSelected) {
   Image result = Filters.edgeDetect(image);
   panel.setImage(result);
} else {
   TaskID<Image> result = Filters.edgeDetectTask(image)
        notify(panel::setImageTaskID(TaskID));
}
```

If sequential mode is selected, then the EDT computes the filter and displays the result to the panel. Alternatively, if the parallel mode is selected, then the task is invoked (therefore freeing the EDT) and `notify` clause is used to update the GUI when the result is ready. Since the original `setImage` method requires an `Image` parameter, another method is required. Code reuse is again achieved:

```
public void setImageTaskID(TaskID<Image> id) {

    setImage(id.getResult());

}
```

### Interactive tasks and interim results

Since the search functionality involves external I/O, this functionality is defined as an interactive task. Again, like the examples above, code reuse is achieved:

```
INTERACTIVE_TASK public static List<Image> searchTask
           (String query, int picsPerPage, int pageOffset) {
   return search(query, picsPerPage, pageOffset);
}
```

Recall from section 6.3.1 that the sequential mode of the search did not support canceling or interim updates of partial results (because the EDT is unresponsive during the search). However, implementing these functionalities for the parallel mode is easy with ParaTask. But first, here is the event handler to invoke the search functionalities:

```
if (sequentialModeSelected) {
  List<Image> results = Search.search(query, picsPerPage, pageOffset);
  for (Image image: results) {
    panel.addImage(image);
  }
  progressBar.setValue(100);
  searchCompleted();
} else {
  currentSearchID = Search.searchTask(query, picsPerPage, pageOffset)
    notify(searchCompleted())
    notifyInterim(receiveIntermediate(TaskID,Image));
}
```

In the case of the sequential mode, the thumbnail results are added to the panel only when all images are retrieved. After this, the GUI is updated via `searchCompleted()`. In the case of the parallel mode, the GUI is updated via the `notify` clause. The `currentSearchID` refers to the TaskID of the project's current search. If the user presses the cancel button, then the EDT will respond by sending a cancel request, i.e. `currentSearchID.requestCancel()` (section 5.1.2.5). Via `receiveIntermediate()`, the `notifyInterim` clause (section 5.3.5) is used to populate the panel as each image (intermediate result) arrives:

```
private void receiveIntermediate(TaskID id, Image image) {
  panel.addImage(image);
  progressBar.setValue(id.getProgress());
}
```

So, how are the interim results emitted by the task? And how is the progress of the task updated? A slight modification is added to the *sequential code* of the search functionality. Recall that the sequential code is used in both the sequential and parallel modes (since `searchTask()` calls `search()`):

```
public static List<Image> search(String query, int picsPP, int offset) {
   List<Image> results = new ArrayList<Image>();

   // retrieve IDs of photos that match the search criteria
```

```
        PhotoList pList = Flickr.getPhotoIDs(query, picsPP, offset);

        // retrieve the thumbnails and add to results
        int i = 0;
        for (Photo p: pList) {
            Image thumb = Flickr.getThumbnail(p);
            results.add(thumb);

            // check if inside a task
            if (CurrentTask.insideTask()) {
                // check if task should terminate early
                if (CurrentTask.cancelRequested()) {
                    CurrentTask.setProgress(100);  // update task progress
                    return list;
                } else {
                    // update task progress
                    CurrentTask.setProgress(++i/pList.size()*100);
                    // publish interim result
                    CurrentTask.publishInterim(thumb);
                }
            }
        }
        return results;
    }
```

In order to introduce task-specific code, `CurrentTask.insideTask()` will determine if the current thread is a ParaTask worker thread (i.e the parallel mode). The task will intermittently check whether a cancel request has been submitted. If so, it will cease retrieving the rest of the thumbnails and return immediately. Otherwise, it updates its progress and publishes the newly retrieved thumbnail. This approach demonstrates decoupling since the search functionality does not need knowledge of code that is interested in the progress and intermediate results of the task.

### Accumulating filters

Section 6.3.1 discussed an example where the image in figure 6.15(c) was created by applying the blur filter multiple times. The images in figure 6.14 were created in a similar fashion, where the colour invert filter was applied on top of the edge detection filter. Because the application remains responsive during the parallel mode, it is very likely that the user will be enqueuing filters (i.e. tasks) at a faster rate than they complete. Therefore, in order to achieve an accumulated filter effect in the parallel mode, the TaskIDs of the respective filters are added to the filter history of the respective image (remember that this does not apply to the sequential mode since the interface is unresponsive when a filter is being applied):

```
// each image has a history of filters
TaskIDGroup<Image> history = historyMap.get(image);

// enqueue the new task, but it must wait for previous filters
TaskID<Image> newFilter = Filters.blurTask(image)
        notify(panel::setImageTaskID(TaskID))
        dependsOn(history);

// add the new filter to the history (for future filters)
history.add(newFilter);
```

In this fashion, the `dependsOn` clause ensures that the newly launched filter is not applied until all the previous filters have been applied to the image. Otherwise, the task will be enqueued immediately and the filter applied to the current state of the image.

As shown with the ParaImage application, it is easy to transform a sequential application into a responsive one. As a bonus, the parallelisation is also achieved without any additional effort. If programmers used threading libraries, considerable effort is first necessary to produce a responsive application and then more effort to also introduce parallelisation. With ParaTask, responsiveness and parallelism come together.

## 6.4 Combining ParaTask & the Parallel Iterator

By combining the Parallel Iterator with ParaTask, programmers have an easy way to develop SPMD programs (in particular when combined with multi-tasks):

```
TASK(*) public void processElements(ParIterator pi) {

    while (pi.hasNext()) {
        process(pi.next());
    } // PI implicit barrier synchronisation
    ...
    // all elements processed at this stage
    ...
}
```

As presented in section 3.1.1, the default Parallel Iterator contains an implicit barrier synchronisation. From ParaTask's point of view, the Parallel Iterator's implicit barrier is a form of external blocking. In order to overcome this blocking in the multi-task, the Parallel Iterator should be constructed without the barrier synchronisation (section 3.1.1). If it is necessary to synchronise at the end of a multi-task, there is an implicit synchronisation at the end of multi-tasks that achieves this (i.e. the multi-task's `TaskIDGroup`). If internal synchronisation is necessary within the multi-task, the programmer may use the `CurrentTask.barrier()` presented in section 5.3.3:

```
TASK(*) public void processElements(ParIterator pi) {
    while (pi.hasNext()) {
        process(pi.next());
    } // no blocking in the Parallel Iterator
    CurrentTask.barrier();
    // all elements processed at this stage
    ...
}
```

This allows the multi-task to implement a barrier synchronisation, while ensuring that worker threads do not block. This benefit is twofold. First, it helps to eliminate deadlock since worker threads are always progressing some task (especially important for nested parallelism). Second, it helps to improve performance for other tasks since worker threads are not idle.

# Chapter 7

# Conclusions

The aim of this thesis was to address desktop parallelisation. Traditionally, candidate applications for parallel computing were typically restricted to the scientific, engineering and database fields. With the advent of multi-core processors for mainstream systems, desktop applications must also be parallelised in order to benefit. Unfortunately, parallelisation for the desktop applications is even more complicated than that of general parallel computing problems. One of the biggest reasons for this difficulty is that desktop applications have a different structure: they are event-based and composed of a graphical user interface. This requires a rethinking of parallelism.

Due to the structure of typical desktop applications, this thesis acknowledges that the focus needs to be on object-oriented parallelisation. An important aspect in this approach is that the current software engineering approach must not be broken. As a result, two concepts have been developed: the Parallel Iterator and Parallel Task. Although one is for data parallelism and one for task parallelism, the two concepts complement each other.

The first concept, the Parallel Iterator, was proposed for object-oriented programs on shared memory systems. The Parallel Iterator concept allows parallel traversal of a collection of elements while the structure of the program remains unchanged. This may be performed virtually with any collection, even those inherently sequential, therefore being a faithful extension to the sequential iterator. The interface of the Parallel Iterator even imitates the standard Java-style sequential iterator. The Parallel Iterator promotes encapsulation and separation of concerns by hiding parallelisation and collection details from the programmer. When used in combination with OpenMP, the structure of the sequential program remains virtually unchanged.

The core concept of the Parallel Iterator is sufficient for many, but not all iterative computations. As such, the scope of the Parallel Iterator was expanded to solve more parallel computing situations. The first includes user-defined reductions in an object-oriented approach, essentially allowing programmers to realise arbitrary reductions for any data type. The second included support for exception handling in a parallel loop. This is especially important for object-oriented languages. Another feature included parallel semantics for the loop break, and also parallel semantics for removing elements during traversal. These features complement each other and can even be used elegantly together. The Parallel Iterator concept was further extended to allow parallel traversal of tree structures, such as XML documents, in the form of the Tree Parallel Iterator.

In addition to the ease of use compared to other common approaches, the results show negligible overhead with effective load scheduling which produce the expected inherent speedups. It was also confirmed that flexible scheduling policies are important and this is easy with the Parallel Iterator, even decidable dynamically at run-time. Such fine-tuning available to the programmer includes scheduling scheme, chunk size and the number of threads involved. In particular for interactive desktop applications involving heavy disk usage, controlling the thread count is important. The performance of the Parallel Iterator is in many cases superior to that of traditional parallelism approaches, including QtConcurrent.

The second concept, Parallel Task, is an object-oriented solution for the parallelisation of a wide range of applications. Programmers introduce parallelism with a single keyword: a method modifier that also serves as a form of documentation. The model supports the entire spectrum of common task types. The standard type, one-off tasks, is sufficient for most situations when asynchronous computation is necessary. To support SPMD type computations, multi-tasks are provided to spawn multiple tasks with group awareness. Added functionality for multi-tasks include synchronising with its sibling tasks and performing reductions. Finally, interactive tasks allow the programmer to define asynchronous computations that involve external I/O. All these task types are unified in the same model and may also be used in nested parallelism.

Many parallel applications are non-trivial, where individual tasks will involve dependences between each other. Implementing such synchronisation with ParaTask is trivial with the intuitive `dependsOn` clause. As well as simplifying the programming effort, this clause promotes code reuse because of the decoupling amongst tasks. Also supporting decoupling is the `notify` clause: it allows tasks to inform registered methods of its completion in a non-blocking fashion. This is especially helpful because such communication will mostly occur between different threads.

An important goal in developing ParaTask is to ensure the software engineering approach is maintained by adhering to object-oriented principles. Since the various task modifiers are associated with the method signature, this ensures encapsulation and provides a form of documentation. It also ensures that inheritance and polymorphism are enforced since the modifiers must be consistent in subclasses. Exception handling is also a vital object-oriented concept. In particular, ParaTask ensures that the Catch or Specify requirement is adhered to in an asynchronous model.

An important aspect of this thesis was the parallelisation of desktop applications. As such, interactivity from the user's point of view is vital. ParaTask achieves this interactivity not only with asynchronous tasks, but it also helps the programmer develop programs that publish interim results and the canceling of tasks. Most of the aspects discussed in this thesis were used to implement an example desktop application, ParaImage. In addition to parallelising computations, it allows users to experience first-hand why parallel computing is vital for a successful interactive desktop experience.

ParaTask's help-first work-stealing scheduling policy led speedup performance for a range of applications compared to typical Java parallelisation approaches, while ParaTask's work-sharing scheduling policy demonstrated superior user-perceived performance. The default scheduling policy, using a mixture of work-stealing and work-sharing, allows for the benefit of both policies. As well as traditional parallelism approaches, ParaTask's performance was compared to JCilk, overall demonstrating superior performance. ParaTask has also been successfully applied to various desktop applications. The ParaImage application demonstrated how easily parallelism and responsiveness may be achieved in an application, allowing for high code reuse and decoupling while promoting encapsulation.

## Future work

Overall, the major components of the Parallel Iterator concept have been deeply studied. Consequently, both the semantics and the implementation of the Parallel Iterator are stable. In fact, the Java implementation of the Parallel Iterator and Parallel Task have been released and are available[1] for public use. However, there are some interesting aspects that could be explored further:

- Incorporate a mechanism to allow multiple collections to be traversed such that dynamic and guided scheduling may be used (as mentioned in section 3.1.7).
- Implementing and evaluating further scheduling schemes for the Tree Parallel Iterator (such as bottom-up dependences, or other variants of the work-stealing).
- Study and optimise the Parallel Iterator for data locality. For example, the distribution of elements amongst threads and how performance is affected by cache effects (for example, to avoid false sharing) and NUMA (Non-Uniform Memory Access) systems.

The semantics underlying ParaTask are stable and have integrated well with the structure of typical GUI desktop applications. With that said, there are various optimisations and extensions to consider:

- Integrate the Parallel Iterator and OpenMP into the ParaTask compiler.
- Develop tool support for ParaTask in an integrated development environment, for example: syntax highlighting, code completion, visualisation of `dependsOn` and `notify`.
- Optimise runtime to improve speed.
- Introduce memory awareness for the scheduling, for example to avoid false sharing.

[1]www.parallelit.org

# Appendix A

# ParaTask grammar

ParaTask was essentially implemented as an extension to Java's grammar. Rather than reproducing the entire Java grammar here, only modified sections of the original Java grammar is presented to focus on ParaTask's integration. The ***highlighted sections*** refer to specific additions made to accommodate ParaTask, while the rest refers to aspects of the original Java grammar that remain unchanged. All the following are extracts from the Java 1.5 grammar file that JavaCC uses to parse Java code, now adapted to parse ParaTask code.

## A.1 Tokens

In order to allow the grammar to recognise ParaTask keywords, these have been defined as tokens (added to the end of the standard Java tokens):

TOKEN :
{
  < ABSTRACT : "abstract" >
  < BOOLEAN : "boolean" >
  ...
  ***< TASK : "TASK" >***
  ***< INTERACTIVE_ TASK : "INTERACTIVE_ TASK" >***
  ***< DEPENDS_ ON : "dependsOn" >***
  ***< NOTIFY : "notify" >***
  ***< NOTIFY_ GUI : "notifyGUI" >***
  ***< NOTIFY_ INTERIM : "notifyInterim" >***

```
   < NOTIFY_INTERIM_GUI : "notifyInterimGUI" >
   < ASYNC_CATCH : "asyncCatch" >
}
```

## A.2 Task declarations

A task declaration is essentially a method declaration with a task modifier. This may appear as part of a class or interface declaration.

```
ClassOrInterfaceBody →
        "{" ( ClassOrInterfaceBodyDeclaration )* "}"

ClassOrInterfaceBodyDeclaration →
        Modifiers FieldDeclaration
      |
        Modifiers MethodDeclaration
      |
        TaskDeclaration
      |
        ...

TaskDeclaration →
        TaskModifier [ MultiTaskCount ] Modifiers MethodDeclaration

TaskModifier→
        "TASK"
      |
        "INTERACTIVE_TASK"

MultiTaskCount→
        "("
              "*"
            |
              IntegerLiteral
            |
              Name
        ")"

Name →
        IDENTIFIER ( "." IDENTIFIER )*
```

```
Modifiers →
        (
              “public”
           |
              “static”
           |
              ...
        )*

MethodDeclaration →
        MethodHeader ( Block | “;” )

MethodHeader →
        ResultType IDENTIFIER [ “throws” NameList ] “(” [ ParameterList ] “)”
```

## A.3 Task invocations

A task invocation is essentially a method call expression followed by at least one ParaTask clause. This may be part of a statement expression, or a variable declaration.

```
StatementExpression →
        (
              PreIncrementExpression
           |
              PreDecrementExpression
           |
              PrimaryExpression
              [
                “++”
              |
                “--”
              |
                “=”
                (
                   TaskClauseExpression
                |
                   Expression
                )
            ]
        ) “;”

PrimaryExpression →
        IDENTIFIER
      |
        ...

BlockStatement →
        Statement
      |
        VariableDeclarationExpression “;”
      ...

VariableDeclarationExpression →
        Modifiers Type VariableDeclarator ( “,” VariableDeclarator )*

VariableDeclarator →
        IDENTIFIER [ “=” VariableInitialiser ]

VariableInitialiser→
        ArrayInitialiser
      |
        TaskClauseExpression
      |
        Expression
      |
        ...

TaskClauseExpression→
        MethodCallExpression
        (
          “dependsOn” “(” ArgumentList “)”
        |
          Notify “(” NotifyArgumentList “)”
        |
          “asyncCatch” “(” HandlerArgumentList “)”
        )+
```

***Notify***→

  ***“notify”***

  |

  ***“notifyGUI”***

  |

  ***“notifyInterim”***

  |

  ***“notifyInterimGUI”***

***NotifyArgumentList***→

  ***NotifyArgument*** ( “,” ***NotifyArgument*** )*

***NotifyArgument***→

  [ Expression “::” ] MethodCallExpression

***HandlerArgumentList***→

  ***HandlerArgument*** ( “,” ***HandlerArgument*** )*

***HandlerArgument***→

  Name ***NotifyArgument***

# Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[3] Jose Aguilar and Ernst Leiss. Parallel loop scheduling approaches for distributed and shared memory systems. *Parallel Processing Letters*, 15(1-2):131–152, 2005.

[4] Eric Allen, David Chase, Joe Hallet, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification.* Sun Microsystems, Inc., version 1.0 edition, March 2008.

[5] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *In Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2001.

[6] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Computers, IEEE Transactions on*, 38(12):1631–1644, Dec 1989.

[7] Apple Inc. *Grand Central Dispatch technical brief*, August 2009.

[8] Matthew H. Austern, Ross A. Towle, and Alexander A. Stepanov. Range partition adaptors: a mechanism for parallelizing STL. *SIGAPP Appl. Comput. Rev.*, 4(1):5–6, 1996.

[9] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[10] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization. In *Proceedings of the IEEE*, pages 211–243, 1993.

[11] Luiz Andrè Barroso. The price of performance. *Queue*, 3(7):48–53, 2005.

[12] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. BSP vs LogP. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 25–32, New York, NY, USA, 1996. ACM Press.

[13] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. Generic parallel programming using C++ templates and skeletons. *Lecture notes in computer science*, 3016:107–126, 2004.

[14] Joshua Bloch. *Effective Java.* Addison-Wesley, 2nd edition, 2008.

[15] Wolfgang Blochinger, Michael Kaufmann, and Martin Siebenhaller. Visualizing structural properties of irregular parallel computations. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 125–134, New York, NY, USA, 2005. ACM Press.

[16] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Massachusetts Institute of Technology, 1995.

[17] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[18] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract). *ACM SIGMETRICS Performance Evaluation Review*, 26(1):266–267, 1998.

[19] Mirko Boehm. KDE 4.0 API Reference - Introduction to ThreadWeaver. http://api.kde.org/cvs-api/kdelibs-apidocs/threadweaver/html/index.html, 2008.

[20] Simone Bordet. Foxtrot - easy API for JFC/Swing. http://foxtrot.sourceforge.net/, 2008.

[21] BrianÂ Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Java Concurrency in Practice, 2006.

[22] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[23] Mukund Buddhikot and Sanjay Goil. Throughput computing with chip multithreading and clusters. *Lecture Notes in Computer Science*, 3769:128–136, 2005.

[24] 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.

[25] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, New York, NY, USA, 2001. ACM.

[26] J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *European Conference on Parallel Processing*, pages 377–382, 1998.

[27] W.H. Burkhardt. Limitations to parallel processing. In *Computers and Communications, 1990. Conference Proceedings., Ninth Annual International Phoenix Conference on*, pages 86–93, 1990.

[28] Warren F. Burton and Ronan M. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

[29] Paul J. Campbell. Gauss and the eight queens problem: A study in miniature of the propagation of historical error. *Historia Mathematica*, 4(4):397 – 404, 1977.

[30] Peter Carlin, K. Mani Chandy, and Carl Kesselman. The Compositional C++ language definition. Technical Report 1993.cs-tr-92-02, 12, 1993.

[31] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading*. John Wiley & Sons, 2006.

[32] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *Computer*, 27(8):13–26, 1994.

[33] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[34] Jong-Deok Choi and Sang Lyul Min. Race frontier: reproducing data races in parallel-program debugging. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–154, New York, NY, USA, 1991. ACM Press.

[35] Cilk Arts, Inc. *Cilk++ Programmer's Guide*, 2009.

[36] Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.

[37] World Wide Web Consortium. W3C scalable vector graphics (SVG). http://www.w3.org/Graphics/SVG/, 2009.

[38] Mache Creeger. Multicore CPUs for the masses. *Queue*, 3(7):64–ff, 2005.

[39] John S. Danaher, I.-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, 2006.

[40] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, 2004.

[41] Encyclopædia Britannica, Inc. Time perception. In *Encyclopœdia Britannica*. Chicago, USA, 2009.

[42] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Desktop scheduling: how can we know what the user wants? In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 110–115, New York, NY, USA, 2004. ACM Press.

[43] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing complex scans and reductions. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 135–146, New York, NY, USA, 1994. ACM.

[44] Kristián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism and interactive performance of desktop applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 129–138, New York, NY, USA, 2000. ACM Press.

[45] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis onlarge-scale multiprocessors. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 163–173, New York, NY, USA, 1988. ACM Press.

[46] Eitan Frachtenberg. Process scheduling for the parallel desktop. In *Parallel Architectures,Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, 2005.

[47] Richard Friedman. Using the tasking feature - OpenMP. http://wikis.sun.com/display/openmp/Using+the+Tasking+Feature, 2008.

[48] Nasser Giacaman and Oliver Sinnen. Objected-oriented parallelisation: Improved and extended Parallel Iterator. In *14th IEEE International Conference on Parallel and Distributed Systems (ICPADS '08)*, Melbourne, Australia, 2008.

[49] Nasser Giacaman and Oliver Sinnen. Parallel Iterator for parallelising object oriented applications. In *The 7th WSEAS International Conference*

*on Software Engineering, Parallel and Distributed Systems (SEPADS '08)*, Cambridge, UK, 2008.

[50] Nasser Giacaman and Oliver Sinnen. Task parallelism for object oriented programs. In *9th International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'08)*, Sydney, Australia, 2008.

[51] Nasser Giacaman and Oliver Sinnen. Supporting partial ordering with the Parallel Iterator. In *International Workshop on Parallel and Distributed Algorithms and Applications (PDAA), in conjunction with 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, Hiroshima, Japan, 2009.

[52] Nasser Giacaman and Oliver Sinnen. Parallel Task for parallelizing object-oriented desktop applications. In *IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, USA, 2010.

[53] P. B. Gibbons. A more practical PRAM model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, New York, NY, USA, 1989. ACM Press.

[54] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[55] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *An Introduction to Parallel Computing: Design and Analysis of Algorithms.* Addison Wesley, 2nd edition, 2003.

[56] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009)*, 2009.

[57] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 120–132, New York, NY, USA, 1991. ACM Press.

[58] Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131, New York, NY, USA, 2004. ACM Press.

[59] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[60] Qihang Huang, Zhiyi Huang, Paul Werstein, and Martin Purvis. Gpu as a general purpose computing resource. In *Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 151–158, Washington, DC, USA, 2008. IEEE Computer Society.

[61] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

[62] Paul Hyde. *Java Thread Programming*. Sams, 2001.

[63] Intel Corporation. *Reference for Intel Threading Building Blocks*, 2010.

[64] Elizabeth Johnson and Dennis Gannon. HPC++: experiments with the parallel standard template library. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 124–131, New York, NY, USA, 1997. ACM Press.

[65] Terry Jones, Shawn Dawson, Rob Neely, William Tuel, Larry Brenner, Jeffrey Fier, Robert Blackmore, Patrick Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 10, Washington, DC, USA, 2003. IEEE Computer Society.

[66] Mackale Joyner, Bradford L. Chamberlain, and Steven J. Deitz. Iterators in Chapel. pages 8 pp.–, April 2006.

[67] A. Krikelis. Application-centric parallel multimedia software. *Concurrency, IEEE [see also IEEE Parallel & Distributed Technology]*, 5(4):78–79, 82, 1997.

[68] Doug Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley, 2 edition, 1999.

[69] Doug Lea. JSR 166 overview. http://artisans-serverintellect-com.si-eioswww6.com/default.asp?W9, March 2009.

[70] Seungll Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon, Kemal Ebcioğlu, and Erik Altman. Efficient java exception handling in just-in-time compilation. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 1–8, New York, NY, USA, 2000. ACM.

[71] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[72] Wei Lu and Dennis Gannon. Parallel XML processing by work stealing. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 31–38, New York, NY, USA, 2007. ACM.

[73] Elmar Ludwig. *Multi-threaded User Interfaces in Java*. PhD thesis, University of Osnabrück, Germany, May 2006.

[74] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40:96–100, February 2007.

[75] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. *Research directions in concurrent object-oriented programming*, pages 107–150, 1993.

[76] Microsoft. *Parallel Extensions to the .NET Framework Community Technology Preview (CTP)*, June 2008.

[77] Microsoft Corporation. *MSDN Parallel Computing Developer Center*, 2009.

[78] Giuseppe Milicia and Vladimiro Sassone. The inheritance anomaly: ten years after. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274, New York, NY, USA, 2004. ACM.

[79] Hans Muller and Kathy Walrath. Threads and Swing. The Swing Connection, 2008. http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html.

[80] Wolfgang E. Nagel and Markus A. Linn. Parallel programs and background load: efficiency studies with the PAR-Bench system. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 365–375, New York, NY, USA, 1991. ACM Press.

[81] Scott Oaks and Henry Wong. *Java threads.* O'Reilly Media, Inc, 3 edition, 2004.

[82] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*, 2008.

[83] Yinfei Pan, Wei Lu, Ying Zhang, and Kenneth Chiu. A static load-balancing scheme for parallel XML parsing on multicore CPUs. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[84] C.M. Pancake and C. Cook. What users need in parallel tool support: survey results and analysis. pages 40–47, May 1994.

[85] Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency: practice and experience*, 12:917–980, 2000.

[86] William M. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 188–195, New York, NY, USA, 1998. ACM.

[87] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. Sunos multi-thread architecture. In *In Proceedings of the Winter 1991 USENIX Conference*, pages 65–80, 1991.

[88] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[89] Steven C. Seow. *Designing and Engineering Time: The Psychology of Time Perception in Software.* Addison-Wesley, 2008.

[90] Oliver Sinnen. *Task Scheduling for Parallel Systems.* Wiley, 2007.

[91] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[92] Alexander Stepanov and Meng Lee. *The Standard Template Library.* Hewlett-Packard Laboratories, 1995.

[93] Sun Microsystems. JavaCC Home. https://javacc.dev.java.net/, 2009.

[94] Sun Microsystems Inc. *Java Platform Standard Edition 6 API Specification*, December 2006.

[95] Sun Microsystems, Inc. Concurrency in Swing. The Java Tutorials, 2008. http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html.

[96] Herb Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):–, February 2005.

[97] Herb Sutter. The pillars of concurrency. *Dr. Dobb's Journal*, August 2007.

[98] Herb Sutter. Use threads correctly = isolation + asynchronous messages. *Dr. Dobb's Journal*, March 2009.

[99] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[100] Domenico Talia. Parallel computation still not ready for the mainstream. *Commun. ACM*, 40(7):98–99, 1997.

[101] Julien C. Thibault and Inanc Senocak. CUDA implementation of a navier-stokes solver on multi-GPU desktop platforms for incompressible flows. In *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, 2009.

[102] TIOBE Software BV. TIOBE programming community index. http://www.tiobe.com/tpci.htm, 2008.

[103] Trolltech. *Qt 4.4 Reference Documentation (Open Source Edition)*, 2008.

[104] Trolltech Labs. QtConcurrent. http://labs.trolltech.com/page/Projects/Threads/Qt-Concurrent, 2008.

[105] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM Press.

[106] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[107] Matt Weisfeld. *The Object-Oriented Thought Process.* Addison-Wesley Professional, 3 edition, 2008.

[108] Barry Wilkinson and Michael Allen. *Parallel programming techniques and applications using networked workstations and parallel computers.* Prentice Hall, 2nd edition, 2005.

[109] Joel Winstead. Structured exception semantics for concurrent loops. Master's thesis, School of Engineering and Applied Science, University of Virginia, 2002.

[110] Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Thomas Risser, and Mark Hoeber. *The Java Tutorial: A Short Course on the Basics.* Prentice Hall, 4 edition, 2006.