Parallel Algorithmic Patterns in Java

Jacob Barbour

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

A series of experiments were conducted to compare the efficacy of the Java Fork/Join framework to a manually hand-threaded approach using Java Threads. A number of algorithms were tested with these different approaches, including: MergeSort, QuickSelect, Fast Fourier Transform, and Strassen's Matrix Multiplication. These experiments concluded that the Java Fork/Join framework was an effective, performant abstraction requiring significantly less implementation effort than the hand-threaded approach. Following from this, **Spooky D&C**, a lightweight parallel algorithmic D&C skeleton built atop the Fork/Join framework, was introduced. It aims to abstract away Fork/Join framework constructs from the programmer, without doing so in an overly prescriptive manner. In contrast to existing skeletons, it prioritises the simplicity of: interface, extensibility, and the skeleton implementation itself. This allows the skeleton to grow with the requirements of software it supports, rather than limiting development through overly restrictive generalizations. A further series of experiments were carried out on a subset of the algorithms tested previously in order to compare the efficacy of **Spooky D&C** to the pure Java Fork/Join framework approach. These experiments concluded that **Spooky D&C** was capable of matching, or even exceeding, the performance of the pure Java Fork/Join approach whilst requiring less implementation effort.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jacob Barbour*)

Acknowledgements

Firstly, I'd like to thank my family for their constant support and encouragement. Furthermore, I'd specifically like to thank my parents who, through their love and support, are responsible for the fortunate position I find myself in today.

I would also like to extend my gratitude towards my supervisor, Prof. Murray Cole, who provided invaluable guidance throughout the entire project. Perhaps more importantly, he was always available and a source of encouragement as I took my first steps into the world of technical writing.

I also extend my thanks to my flatmates, Cammy and Matthew, for enduring my endless dialogue about my dissertation and, more importantly, skeletons. They would certainly not be wrong for mistaking every day in the flat over the past six months as the 31st of October!

Finally, I'd like to thank anyone who spent time, no matter how long, proofreading my numerous drafts and keeping my interpretation of grammar in check!

Table of Contents

Chapter 1

Introduction

The advent of the parallel programming paradigm has required programmers to adapt. This shift has brought new potential performance to the table, at the expense of increased hardware and software complexity, and programmers have been forced to turn to one of their most reliable tools: abstraction. As such, abstractions have been introduced in general purpose programming languages such as Java, the focus of this project. This project investigates the applicability of existing abstractions in Java to parallel divide-and-conquer (D&C) algorithms and aims to introduce a new one, in the form of a parallel algorithmic D&C skeleton. Specifically, the Java Fork/Join framework is compared to a manually hand-threaded approach through a series of experiments to determine the benefits, if any, in performance and implementation effort for a sample of four algorithms. The results of these experiments are then used to inform the design and implementation of **Spooky D&C**, a lightweight parallel algorithmic D&C skeleton built atop the Fork/Join framework. Finally, experiments are carried out comparing **Spooky D&C** against the baseline pure Fork/Join framework in terms of performance, overhead, and implementation effort for Java programmers. An overview of the project and contributions are given below:

- Chapter **2** discusses the background associated with the project along with related work in order to contextualize the subsequent work and contributions.
- Chapter **3** details the experiments carried out for the selected algorithms, comparing the Fork/Join framework approach to manually hand-threaded solutions. It contains a description of the implementation and testing work carried out, a discussion of results per-algorithm, and the conclusions reached.
- Chapter **4** introduces **Spooky D&C**, the parallel algorithmic skeleton based on the conclusions reached from the previous chapter and is the primary contribution of the report. This is followed by experiments comparing it to the Fork/Join framework approach, with another discussion of results per-algorithm.
- Chapter **5** serves as an overall conclusion, containing a retrospective on the entire project. It contains a summary of the project focussing on the results and conceptual problems solved, lessons learned throughout, and a discussion of future work to be carried out.

Chapter 2

Background

In this chapter a brief outline of the history, current state, and challenges of general-purpose microprocessors, which gave rise to the 'parallel programming' paradigm, is given (Section **2.1**). This is followed by a gentle introduction to parallelism in the context of a modern computer system's hardware and software configuration (Section **2.2**).

An overview of parallelism in the Java programming language is given, consisting of a more in-depth look at **Java Threads** and the **Fork/Join** framework (Section **2.3**). Finally, the concept of 'algorithmic patterns' is introduced (Section **2.4**), which is followed up by an overview of 'parallel algorithmic skeletons' (Section **2.5**). This overview includes: a description of what they are, challenges faced by them, and an examination of previous research on them.

2.1 Modern Computer Architecture

Over the past few decades, general-purpose microprocessors have improved at a rapid rate due to a combination of architectural and technological advances. One key technological factor in the evolution of modern microprocessors is the increasing of transistor density via improvements in the fabrication process, commonly known as 'die shrinks'. Consequently, an observation known as 'Moore's Law' [Moore (1965)] was posited that predicted a doubling of transistor count in integrated circuits every year (later amended to every two years [Moore et al. (1975)]. This in tandem with Dennard scaling [Dennard et al. (1974)], a scaling law which facilitated constant power density and increased operating frequencies between successive fabrication processes, was the driving force behind the huge generational leaps in microprocessor performance seen over the last few decades. Since the early 2000s, however, the year-on-year exponential improvement of microprocessors has significantly slowed due to the breakdown of Dennard scaling around 2007 [Bohr (2007)] and the subsequent end of Moore's Law around 2015 [Theis and Wong (2017)]. Figures **2.1** & **2.2** illustrate this.

As a consequence of Dennard scaling and, to a lesser extent, Moore's Law ending

modern microprocessor designs have tended towards multicore configurations[1] [Blake et al. (2009)] in order to breach through the physical limitations constraining the performance of an individual core[2]. The one caveat, however, is that this change in design philosophy has shifted a large amount of responsibility onto the individual programmer to deliver software that efficiently orchestrates computations between these cores. This added layer of complexity has significantly increased the implementation effort faced by programmers. As such, there has been a drive to introduce new abstractions to reduce the implementation effort associated with this form of 'parallel programming'. [Legaux et al. (2014)].

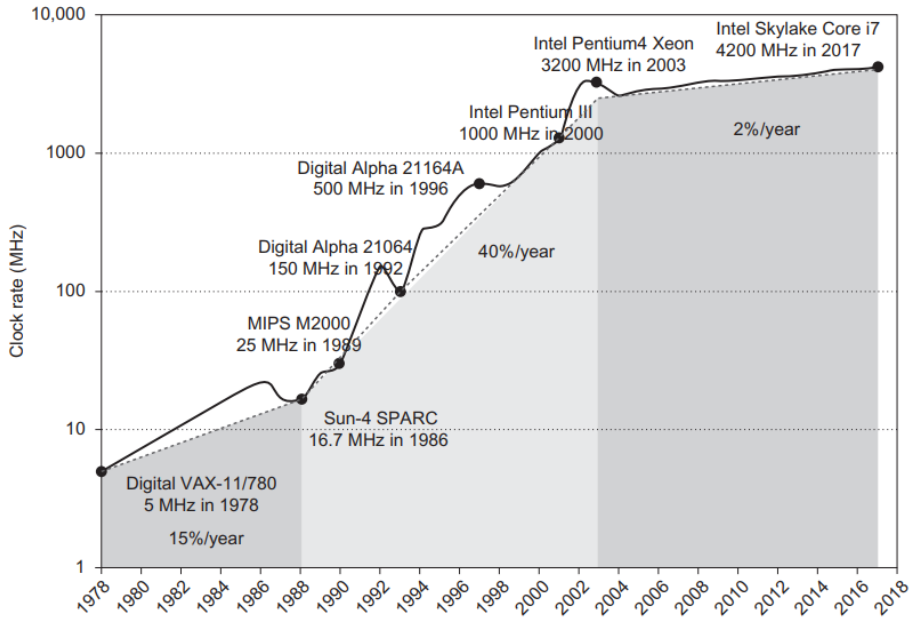

Figure 2.1: Processor clock rates over time. A plateau can be seen starting around 2004 as the limits of Dennard scaling were being reached. **Source: Hennessy and Patterson (2011)**

[1] refers to a microprocessor with multiple execution units

[2] single execution unit

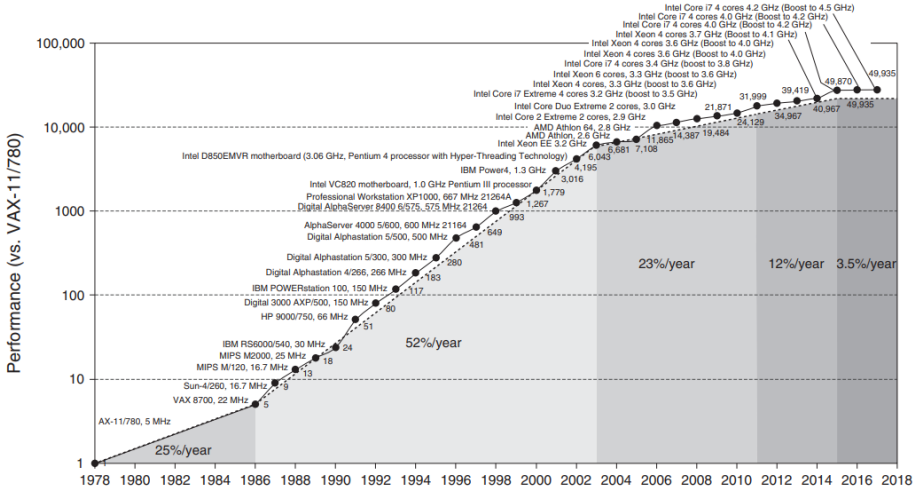

Figure 2.2: Processor benchmarks over time. Performance gains begin to plateau around 2004, likely as a combinational effect of Dennard scaling ending and Moore's Law slowing down. **Source: Hennessy and Patterson (2011)**

2.2 Parallelism

2.2.1 Parallel Computation

At a high level, parallel computation can be defined as the carrying out of multiple computational tasks simultaneously. When considered in more detail, it can be observed that there are multiple distinct forms of parallel computation, all nested at varying levels of abstraction away from the individual programmer.

Instruction-level parallelism exists at the lowest level within the microarchitecture of microprocessors, leveraging many clever techniques to execute instructions in parallel and effectively boost the number of instructions completed per clock cycle.

Data-level parallelism also exists at a similarly low level; however, it is focussed around maximizing the amount of data operated on by a single instruction/action. For example: designing the computer architecture around vector operations rather than exclusively scalar ones.

Task-level parallelism exists at a slightly higher level and is primarily based around splitting a task into various sub-tasks that can be coordinated and run sequentially at the same time, i.e. run 'in parallel'.

Thread-level parallelism is a subtype of task-level parallelism where each individual 'thread' contains a sequence of instructions relating to a subtask and these 'threads' are executed concurrently by the microprocessor.

It is also worth noting that these various forms of parallelism are not mutually exclusive and, in most practical applications, actually complement each other.

Most practical implementations of thread-level parallelism allow each core on the microprocessor to concurrently execute more than one thread, a technique known as **multithreading** [Tullsen et al. (1995)].

2.2.2 Parallel Programming

The concept of a thread exists at both a hardware level and an operating system (OS) level. For example: say you have a microprocessor possessing 4 cores, with hardware multithreading enabled, such that it supports the execution of up to 8 threads at a time. From this, the naive intuition may be that a program should not utilize more than 8 threads. However, as the OS handles the execution scheduling of threads [Arora et al. (1998)] the number of threads within a program is primarily constrained by available memory/limitations imposed by the OS.

The vast majority of modern general-purpose programming languages (GPLs) support access to these logical threads by the programmer, usually in an abstract way implemented in a library/package or otherwise. This allows the programmer to fully leverage thread-level parallelism by not only partitioning the computational work of their program into threads, where appropriate, but also orchestrating the threads by defining how they communicate/interact. Whilst this abstract concept of threads eases the implementation effort that would otherwise be faced by the programmer, there are still some challenges that would not arise in a fully sequential implementation [Lee (2006)]. Namely the issue of non-determinism (from the programmer's perspective) as, more often than not, the threads rely on some shared state and may attempt to modify this state at the same time[3] or a thread may make a false assumption about the state whilst executing an operation as it was silently modified by another thread. There are also important performance considerations that the programmer must take into account, for example: it is vital to consider the granularity[4] and number of threads employed as each thread comes with both computational and memory overhead.

It is also worth noting the limitations of parallelism in general and, by extension, parallel programming. Amdahl's Law [Amdahl (1967)] places a limit on the potential speed-up that can be achieved when parallelizing a fraction of a computation.

$$S_{overall} = \frac{1}{(1-f) + \frac{f}{S}}$$
$$\implies \lim_{S \to \infty} \frac{1}{(1-f) + \frac{f}{S}} = \frac{1}{(1-f)}$$

where f = parallelized fraction, S = speed-up of fraction

This can be generalized to account for additional fractions with varying speed-ups:

$$S_{overall} = \frac{1}{(1 - (\sum_{i=1}^{n} f_i)) + (\sum_{i=1}^{n} \frac{f_i}{S_i})}$$

where f_i = parallelized fraction i, S_i = speed-up of fraction i

[3] commonly known as a 'race condition'

[4] amount of work done by each thread

2.3 Parallelism in Java

2.3.1 Threads

Support for threads has been included in Java's standard library since JDK1[5] and, as of writing, the latest release is JDK21[6] — so they are very much a longstanding and foundational construct in the language. The following classes are of particular interest:

- **Thread** - Defines a simple thread object that performs some task that can be started/stopped. They can be assigned priorities, so the most critical threads get the execution time they need when they need it.
- **ThreadGroup** - Allows for logical grouping and control of threads. Furthermore, ThreadGroup is a composite object as it allows for the inclusion of nested ThreadGroups.
- **ExecutorService** - Provides a useful abstraction for running tasks on a collection of threads. It allows you to define a pool of thread resources and simply pass tasks to the service, rather than having to manually create/start/stop threads for each task. In addition to the convenience provided, there are also performance benefits as the overheads associated with thread creation/management are reduced.
- **Future<T>** - A parameterized wrapper around some object, T, that **may or may not** contain a value at any time. In essence, it is a promise that at some unknown point in the future it will contain the definitive result of a task, provided the task completes.

2.3.2 Fork/Join

The Java Fork/Join framework [Lea (2000)] provides a further abstraction on top of threads and executors. It allows the programmer to define tasks that can be recursively split into smaller tasks. These recursive tasks are then split[7], processed, and recombined[8] into a result using a pool of threads. The following classes are of particular interest:

- **ForkJoinPool** - This defines a pool of both threads and tasks. The threads actively execute tasks submitted to the pool, with these tasks potentially spawning more tasks as a result of their execution.
- **ForkJoinTask** - This is the abstract base class defining a task to be executed by a ForkJoinPool. Its subclasses (listed below) may be extended by the programmer to define the computations they wish to perform.
  - **RecursiveAction** - This abstract class defines a ForkJoinTask that has no return value.
  - **RecursiveTask<V>** - This abstract class defines a ForkJoinTask that has a return value.

[5] JDK - Java Development Kit
[6] https://www.oracle.com/uk/java/technologies/downloads/#java21
[7] this is what 'fork' refers to
[8] this is what 'join' refers to

2.4 Algorithmic Patterns

2.4.1 Overview

Algorithmic patterns are essentially archetypal solutions to computational problems that can be used to categorise algorithms. One clear benefit of using algorithmic patterns is that the programmer does not need to work from first principles every time they solve a problem, rather they can pick the most appropriate pattern and adapt it to their needs. There are a myriad of algorithmic patterns including, but not limited to:

- **Recursion**
- **Dynamic Programming**
- **Divide and conquer**
- **Branch and bound**
- **Wavefront**

2.4.2 Divide and Conquer Pattern

Divide-and-conquer (D&C) is an incredibly versatile and well-known pattern that appears in many popular algorithms. At a high level, the pattern can be defined as the following steps: take a task, recursively split it into sub-tasks, repeat until the sub-tasks are directly solvable, compute the results of the sub-tasks, and then combine these results to obtain the result of the original task. A classic example of this is the MergeSort algorithm, Figure **2.3** visualizes the execution using a binary tree.

In practice, however, it is worth noting that the recursive calls required to implement the D&C pattern are not without cost. Each call will consume some amount of memory and compute, caused by the allocation of a related stack frame and subsequent return operation. As such, the majority of practical implementations opt to use a 'recursion cut-off' where the D&C algorithm switches to a non-recursive algorithm to solve sub-tasks that fall below the threshold. Consider MergeSort, instead of recursively dividing the input array all the way down to singleton arrays, an InsertionSort could be used to sort arrays whose length falls below the recursion cut-off.

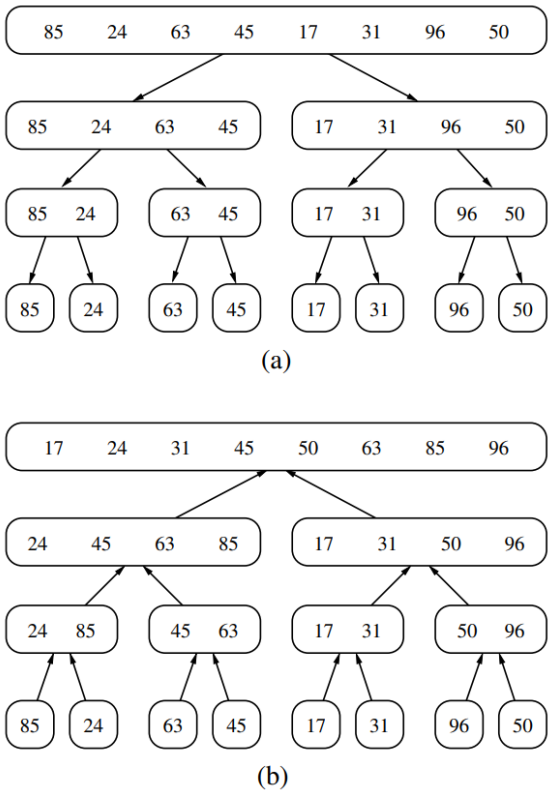

Figure 2.3: **(a)** illustrates the 'divide' phase as the input array is recursively split. **(b)** illustrates the 'conquer' phase as the subtask results are merged to produce the sorted input array. **Source: Goodrich et al. (2013)**

It is also worth noting that algorithmic patterns do not explicitly prescribe a sequential or parallel implementation. As seen in Figure **2.3**, the structure of the subtasks appears compatible with a parallel implementation and maps quite directly to the Fork/Join framework.

2.5 Parallel Algorithmic Skeletons

2.5.1 Overview

There has been extensive research into the development of parallel algorithmic skeletons [Cole (1989), Cole (2004), González and Fraguela (2010), Philippe and Loulergue (2019)]. These skeletons provide an abstraction on top of various algorithmic patterns and allow the programmer to define, at a high level, the computation they wish to perform. Then the underlying implementation of the skeleton, hidden from the programmer, should orchestrate the parallel execution of the computation, applying various optimizations based on its analysis of the computational structure.

There are multiple challenges faced in the skeleton approach [González-Vélez and Leyton (2010), Danelutto et al. (2021)], the most prominent of which is balancing the level of abstraction with flexibility for the programmer to define computations that

do not strictly adhere to the formal definition of a pattern. Another problem faced is the lack of a generalized specification, which reduces the ability of programmers to collaborate and reason about skeletal implementations across different programming environments. However, that is not to say the concept of skeletons has not seen success in industrial applications, with **Google's MapReduce** [Dean and Ghemawat (2008)] framework and **Apache's Spark** [Zaharia et al. (2016)] framework seeing widespread use in the 'big data'[9] community.

2.5.2 Related Work

Skandium [Leyton and Piquer (2010)] and **Calcium** [Caromel and Leyton (2007)] are two closely related parallel algorithmic skeleton libraries written in Java. The key distinction between them is that the former is designed for multicore parallel processing on a single machine, whereas the latter is built on top of the **ProActive** [Caromel et al. (2006)] framework designed for parallel processing across multiple networked machines. The express goal of these libraries is to shield the programmer from all parallel programming constructs, to the furthest extent possible, so that they only need to concern themselves with writing simple sequential code (referred to as *muscles*). Furthermore, they make use of powerful composition to allow the programmer to not only compose a *skeleton* with *muscles* but also with other *skeletons* to enhance performance and remove sequential bottlenecks[10]. Additionally, Calcium features a *blame system*, which discovers performance issues and highlights the potential cause to the programmer so they can rectify them by modifying code or tuning parameters.

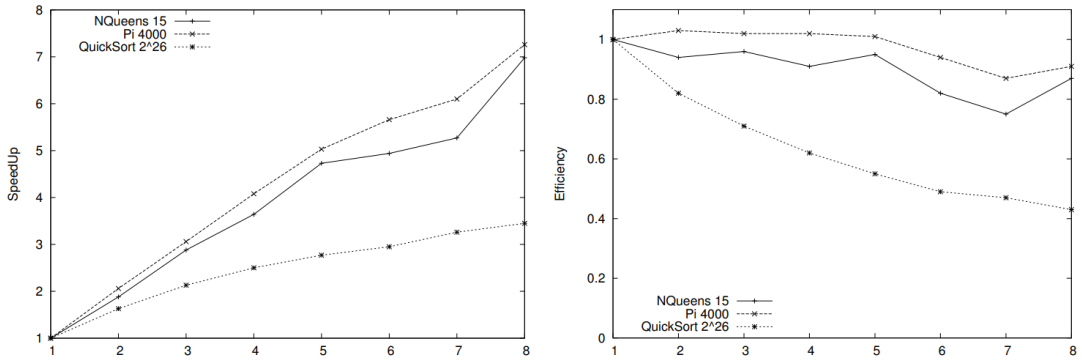

Figure 2.4: Results comparing the speed-up and efficiency of three algorithms (NQueens, Pi, QuickSort) utilizing Skandium's skeleton implementation. **Source: Leyton and Piquer (2010)**

Both of the aforementioned skeletons have seen good performance results in regard to both speed-up and efficiency, with Skandium's results shown in Figure **2.4**. Of particular relevance to this project is the inclusion of a divide-and-conquer skeleton in both Skandium and Calcium. In fact, both the QuickSort and NQueens implementation utilize the divide-and-conquer skeleton provided in Skandium, with the former based

[9] related to the processing of large volumes of data

[10] a situation that may limit throughput and reduce performance

solely upon it. This suggests, and indeed evidences, the viability of applying the skeleton approach to parallel divide-and-conquer problems.

Threaded Building Blocks (TBB) [Reinders (2007), Kukanov and Voss (2007)] is a parallel algorithmic skeleton library created by Intel for C++. As with other parallel algorithmic skeletons, the primary goal of TBB is to guide programmers towards writing efficient parallel programs without needing to concern themselves with the minutiae of thread management and work scheduling. In the words of the lead developer of TBB at its inception, Arch D. Robison: "The key notion is to separate logical task patterns from physical threads, and to delegate task scheduling to the system.". The library provides many skeletons for the programmer to leverage, such as: **parallel_for**, **parallel_reduce**, **parallel_scan**, and **parallel_pipeline**. Notable skeletons closely related to this project are **parallel_for** and **parallel_reduce** which can both be used to implement parallel divide-and-conquer algorithms.

In practice, TBB has proven popular and offers a good speed-up in many applications, even potentially outperforming manually hand-threaded implementations, shown in Figure **2.5**. Furthermore, owing to academic interest and the long period of active development, it offers many additional features aimed to alleviate the implementation effort associated with its use. Examples of which are the **auto_partitioner** and **affinity_partitioner** which provide automatic, heuristically-driven management of task granularity to optimize performance by reducing scheduling overhead and, for the latter, optimizing cache behaviour [**Robison et al. (2008)**].

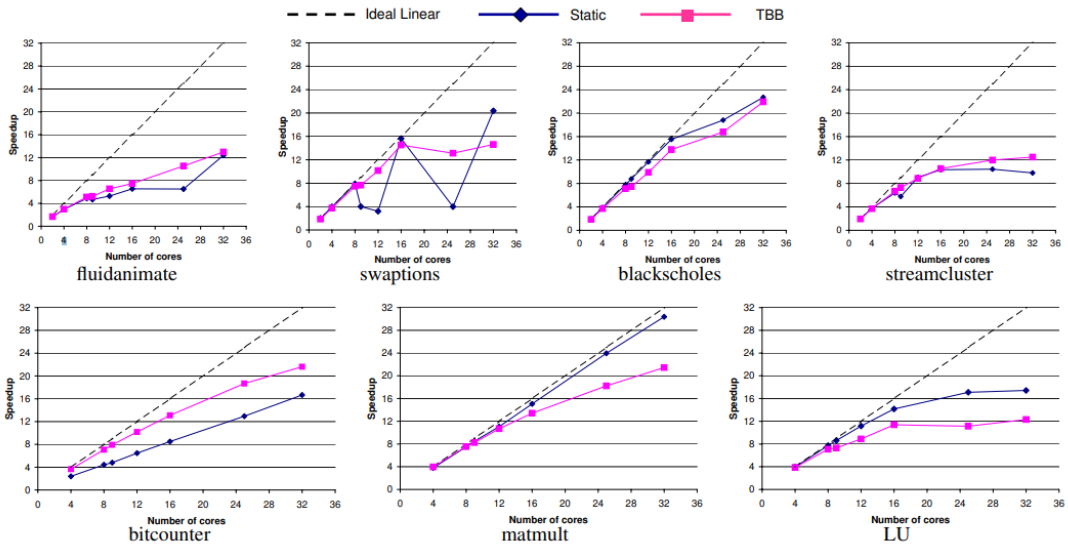

Figure 2.5: Results comparing multiple TBB-based algorithm implementations versus manually hand-threaded versions referred to as *Static*. **Source: Contreras and Martonosi (2008)**

Chapter 3

Experiments with D&C Algorithms

In this chapter, a series of experiments performed on four distinct D&C algorithms are presented. The purpose of these experiments was to examine the performance of the Fork/Join framework when compared to a manually hand-threaded solution. The overall goal was to determine whether the Fork/Join framework would be a suitable foundation on which to build further abstractions in the form of a parallel algorithmic skeleton, presented in Chapter **4**. Additionally, throughout this chapter references are made to Appendix **A** which contains raw, tabulated results data and an additional runtime chart for each of the tests conducted.

For one of the algorithms presented, Strassen's Matrix Multiplication, an effective abstraction for matrices is introduced in Section **3.5.1**. The purpose of the abstraction was to allow the programmer to easily perform operations on submatrices backed by existing matrices in memory. This was invaluable for the implementation of the algorithm: providing greater control over memory usage whilst maintaining good performance and keeping the client code clean. The benefits provided by the abstraction may extend further to other algorithms that rely heavily on the recursive slicing of matrices.

3.1 Overview

3.1.1 Motivation

The primary motivation of the following experiments was to gain an understanding of, and evaluate, two of the mainstream standard library approaches to implementing parallel divide-and-conquer algorithms in Java. A summary of these approaches, **Threads** and **Fork/Join**, can be found in Section **2.3**.

For this purpose, four distinct divide-and-conquer algorithms were selected for implementation. Each algorithm was implemented in 3 distinct ways: purely sequential, parallel via **Threads**, and parallel via the **Fork/Join** framework. Each algorithm/method combination was evaluated both qualitatively based on the implementation effort and quantitatively via performance testing methods. The algorithms selected were as follows:

- **QuickSelect**
- **MergeSort**
- **Fast Fourier Transform**
- **Strassen's Matrix Multiplication**

These algorithms were selected as they represent a robust sample of the different kinds of divide-and-conquer algorithms that a programmer may commonly encounter. Specifically, aside from varying in obvious ways such as input/output type, they also differ subtly in the degree[1] to which each task is divided into subtasks. This can have a direct, and noticeable, impact on how much each algorithm benefits from parallelism. For example: **QuickSelect** was deliberately chosen as each task can only be divided into, at most, one subtask meaning there should be no tangible benefit from parallelism as there is only one subtask to execute at a time.

The forward-looking goal of this experimentation was to determine the viability of the Fork/Join framework for implementing high performance, parallel divide-and-conquer algorithms. Then, if found to be viable, to explore the building of further abstractions on top of it such that it can be utilized by programmers in a more prescriptive, easy-to-use algorithmic skeleton-like fashion.

3.1.2 Implementation Methodology

All relevant implementation code can be found in the following GitHub[2] repository: **https://github.com/barbourja/ug4-project**.

Each selected algorithm was implemented in a standardized way. The general steps were:

1. Create specialized <**Algorithm Name**>**Strategy** interface extending **GenericStrategy**
2. Implement purely sequential algorithm in **Sequential** class
    - Include classes required to properly model the problem domain
    - Additionally implement useful helper methods for re-use later
3. Implement parallel algorithm using Fork/Join framework in **ForkJoin** class
4. Implement parallel algorithm using Java Threads in **Threaded** class

A more detailed explanation of the constituent classes of each algorithm can be found in the sections below.

[1] the number of subtasks that can be created from a single task
[2] https://github.com/

3.1.2.1 GenericStrategy

A specific implementation of an algorithm is referred to as a 'strategy'. This interface contains the methods that all valid strategies must implement. The primary purpose of this was to allow an easy way to interact with each strategy in an abstract way for testing purposes to reduce code duplication and, in turn, hopefully reduce errors in testing.

All the selected algorithms individually extend this interface to produce a more specialized interface tailored to the specific input/output types of the algorithm. This specialized interface is then implemented by all of the algorithm's concrete[3] strategy class implementations. For example: MergeSort has its own **MergeSortStrategy** interface that is implemented by the concrete **Sequential** strategy, amongst others.

3.1.2.2 Sequential

This is a purely sequential implementation of an algorithm following the divide-and-conquer pattern. This is used to establish a performance baseline in order to determine the speed-up afforded by a parallel implementation. It is worth noting, however, that the majority of sequential implementations stray slightly from the exact definition of the pattern, where they will switch to a non-recursive algorithm for small enough sub-tasks. This is done to reduce the overhead incurred by recursive calls, and the point at which this happens is determined by a user provided 'recursion cut-off' value.

3.1.2.3 ForkJoin

This is a parallel implementation of an algorithm based upon the Fork/Join framework. The related **Sequential** implementation is used to execute the base case computation once the size of a subtask falls below the user-specified parallelism cut-off size[4]. The majority of logic specific to the algorithm can be found within a private nested class that provides a concrete implementation of either a **RecursiveTask** or **RecursiveAction**.

The level of parallelism is controlled by the user when instantiating an object of the class. The parallelism value provided is then used to instantiate a ForkJoinPool with that number of threads available for executing tasks.

3.1.2.4 Threaded

This is a parallel implementation of an algorithm based upon Java Threads. It constitutes the most complex implementation type and, similar to the **ForkJoin** implementation, uses the related **Sequential** implementation to execute the base case computation for sufficiently small tasks. In this implementation strategy, the majority of logic specific to the algorithm can be found within a private nested class that provides a concrete implementation of the **Runnable** interface.

[3] provides a full implementation for all methods prescribed by its interface and/or superclass i.e. ready for instantiation as an object

[4] often referred to as the 'granularity'

Similar to **ForkJoin**, the level of parallelism is controlled by the user when instantiating an object of the class. However, the method employed to control the parallelism is slightly more complex, using the model of an **m-ary** tree[5] to guide the logic that governs the creation of threads (illustrated in Figure **3.1**). Specifically, the user-provided parallelism value translates to the maximum number of worker nodes that can exist in the tree at any one time. This decision was made as the worker nodes (usually leaves) represent threads performing heavy computational work, whereas the majority[6] of parent nodes are simply waiting for their children to complete their work. The aforementioned logic is implemented in a centralized entity, from which individual threads can request the ability to create more threads.

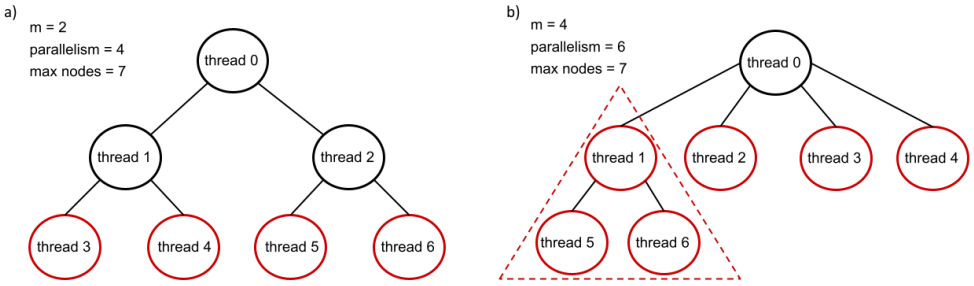

Figure 3.1: Examples of how thread creation is handled in Threaded implementations, using an m-ary tree. **a)** a simple example where the leaves perform all the computational work. **b)** a more complicated example where a parent thread shares some computational work with its children, still resulting in the requested number of parallel worker threads

3.1.3 Testing Methodology

In order to test the performance of each algorithm's implementation strategies, test scaffolding was devised and implemented. This included an abstract class **GenericTestSuite** to work in tandem with the **GenericStrategy** interface introduced in Section **3.1.2.1**. This provided a generic template for testing which was then specialized by each algorithm, allowing for the generation of algorithm specific test data, for example: based on the input type of the algorithm.

Two primary testing methods are defined within the **GenericTestSuite** class: **testVaryingParallelism** and **testVaryingMinSize**. These aim to determine, respectively: the speed-up (relative to **Sequential**) afforded by both parallel strategies (**ForkJoin** and **Threaded**) and the overhead associated with each parallel strategy's work scheduling method. More information on these testing methods can be found in the subsections below.

[5] https://en.wikipedia.org/wiki/M-ary_tree

[6] some parent threads share the work with their children if they are unable to obtain their full requested thread allocation

3.1.3.1 Parallelism Test

The goal of this test was to determine the speed-up and efficacy afforded by each parallel implementation strategy of an algorithm. The following procedure was carried out for each strategy under test:

1. Fix an appropriate recursion cut-off for the **Sequential** strategy, striking a balance between performance and recursive call reduction
2. Fix a sufficiently large test input size
3. For each value of parallelism under test:
   (a) Set an appropriate parallelism cut-off, such that the additional parallel resources available are utilized
   (b) Initialize a pseudo-random number generator with a predetermined fixed seed
   (c) Repeat 10 times:
      i. Generate a valid randomized input of the preset test input size using the PRNG[7]
      ii. Execute the strategy on the generated input and time the execution (in milliseconds)
   (d) Average the runtimes generated across the previous 10 runs, to account for run-to-run variations
4. Calculate and plot useful statistics, such as speed-up, using the average runtimes produced from each parallelism value tested

3.1.3.2 Parallelism Cut-Off (Granularity) Test

The goal of this test was to determine the work scheduling overhead associated with each parallel implementation strategy of an algorithm. This was achieved by creating an artificially difficult operating environment, through the setting of a high parallelism value in combination with a low initial parallelism cut-off value to be gradually increased between tests. The following procedure was carried out for each strategy under test:

1. Fix an appropriate recursion cut-off for the **Sequential** strategy, striking a balance between performance and recursive call reduction
2. Fix a sufficiently large input size
3. Fix a large parallelism value
4. For each value of parallelism cut-off under test:
   (a) Initialize a pseudo-random number generator with a predetermined fixed seed

[7] pseudo-random number generator

(b) Repeat 10 times:

i. Generate a valid randomized input of the preset test input size using the PRNG

ii. Execute the strategy on the generated input and time the execution (in milliseconds)

(c) Average the runtimes generated across the previous 10 runs, to account for run-to-run variations

5. Calculate and plot useful statistics, such as speed-up, using the average runtimes produced from each parallelism cut-off value tested

3.1.4 Testing Configuration

All testing was carried out under the same hardware/software configuration, specified below.

Hardware Configuration:

- **Motherboard:** MSI
  - **Socket:** AM4
  - **Chipset:** X470
- **CPU:** AMD Ryzen 7 5800X3D
  - **Clock Frequency (Base/Boost):** 3.4 / 4.5 GHz
  - **Cores/Threads:** 8 / 16
- **RAM:** 16GB
  - **Generation:** DDR4
  - **Operating Frequency:** 3200MHz

Software Configuration:

- **Operating System:** Microsoft Windows 10
  - **OS Version:** 22H2
- **Java Version:** 18
  - **JDK Vendor:** Oracle

For this testing, the hardware configuration is especially important to note, as core/thread count of the CPU and the system memory can directly affect the speed-up achieved by parallel implementations. For example, with this hardware configuration: the performance of a program that offers an ideal efficiency would be expected to diminish past a parallelism value of 8 (number of cores) and subsequently plateau past a value of 16 (number of hardware threads).

3.2 QuickSelect

QuickSelect takes an unordered array of values and an integer value, k, as input. It will then recursively partition the input array in order to find the k^{th} smallest value. Partition, in this case, means to select an element, **r**, at random and re-order the array such that all elements smaller than **r** appear before it and all elements larger than **r** appear after it. This is the primary mechanism by which the k^{th} smallest value is found; a high-level overview is given below:

1. Partition the array based on **r**
2. **IF** the index of **r** equals **k THEN r** is the k^{th} smallest value
3. **ELSE IF** the index of **r** is greater than **k THEN** recurse into the subarray with the smaller elements as it contains the k^{th} smallest value
4. **ELSE** recurse into the subarray with the larger elements as it contains the k^{th} smallest value

3.2.1 Implementation

The implementation of this algorithm was straightforward for all strategies. Although the **ForkJoin** and **Threaded** strategies required extra boilerplate code to conform with their respective underlying libraries, the inherently sequential nature of the algorithm meant little extra consideration of synchronization was required.

3.2.2 Results

3.2.2.1 Parallelism

All tests were carried out with an input array size of $67,108,864$ and a k value generated randomly. The recursion cut-off value of **Sequential** was set to 1.

The results of testing matched expectations, with the parallel strategies offering no speed-up over the sequential strategy and slightly diminished performance due to added scheduling overheads (Figure **3.2**, Appendix **A.1**). Furthermore, both parallel strategies appeared to perform similarly with **Threaded** performing slightly better at lower levels of parallelism, whereas **ForkJoin** performed better at higher levels of parallelism.

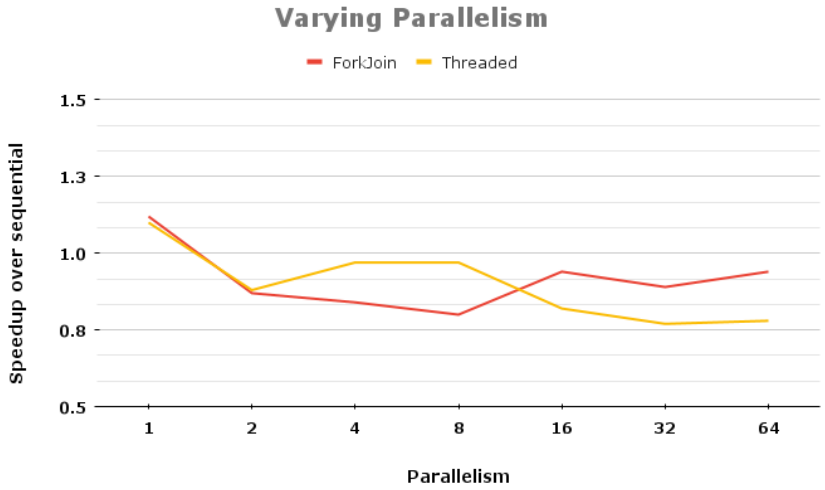

Figure 3.2: Comparing speed-up of parallel QuickSelect strategies when varying parallelism

This is further exemplified when comparing the respective efficiency[8] of both parallel strategies, with consideration[9] to the hardware the testing was run on (Figure **3.3**).

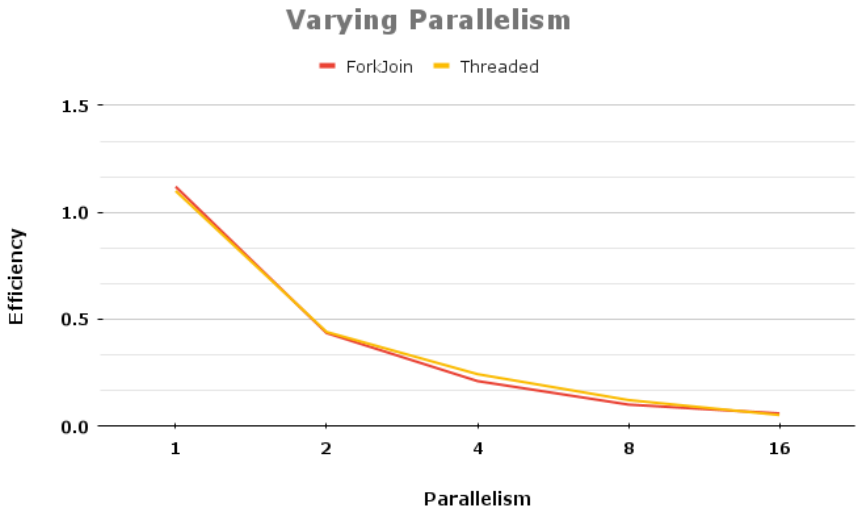

Figure 3.3: Comparing the efficiency of both parallel QuickSelect strategies

3.2.2.2 Parallelism Cut-off

All tests were carried out with an input array size of $67,108,864$ and a k value generated randomly. The parallelism was fixed at 2048 for all parallel strategies. Furthermore, the recursion cut-off value of **Sequential** was set to 1.

[8] speed-up adjusted for the number of parallel processing cores available

[9] only considering parallelism values up to 16, as that is the maximum number of hardware threads offered by the processor

The results of the testing indicated that both parallel strategies performed similarly, with significant variation (Figure **3.4**, Appendix **A.2**). This was expected, as the low degree to which QuickSelect divides tasks means the number of subtasks grows linearly with the number of divisions rather than exponentially, as is common in most D&C algorithms. As a result, there is little extra scheduling work even with a small parallelism cut-off.

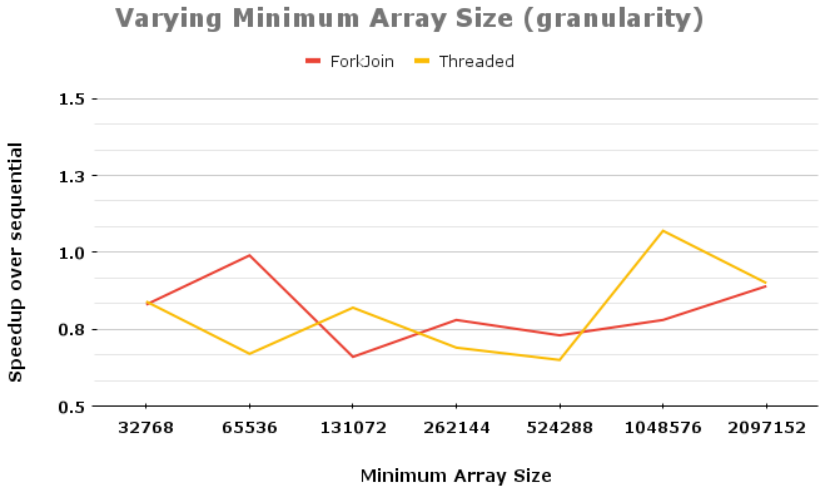

Figure 3.4: Comparing speed-up of parallel QuickSelect strategies when varying parallelism cut-off

3.3 MergeSort

MergeSort takes an unordered array of values as input. It will then recursively divide the original input array into two smaller subarrays until the length of the subarrays falls below a preset value, in which case they can be sorted by some non-recursive approach. These sorted subarrays are then merged into progressively larger sorted arrays until the original input array is fully sorted (visualization in Figure **2.3**).

3.3.1 Implementation

The implementation of the algorithm was straightforward for all strategies. In contrast to how MergeSort is commonly described in literature, the **Sequential** strategy makes use of a simple InsertionSort to sort all subarrays that fall below the recursion cut-off. Furthermore, additional care had to be taken when implementing the parallel strategies to ensure there were no concurrent read/writes to array elements.

3.3.2 Results

3.3.2.1 Parallelism

All tests were carried out with an input array size of $8,388,608$. The recursion cut-off value of **Sequential** was set to 8.

The results of the testing indicated that both parallel strategies offered an appreciable speed-up over the sequential strategy. Furthermore, both **Threaded** and **ForkJoin** achieved roughly similar speed-ups at all levels of parallelism tested (Figure **3.5**, Appendix **A.3**). The only notable deviation where **Threaded** outperformed **ForkJoin** occurred at a parallelism value of 32. However, the overall results were indicative of **ForkJoin's** ability to essentially match the speed-up of **Threaded** for MergeSort.

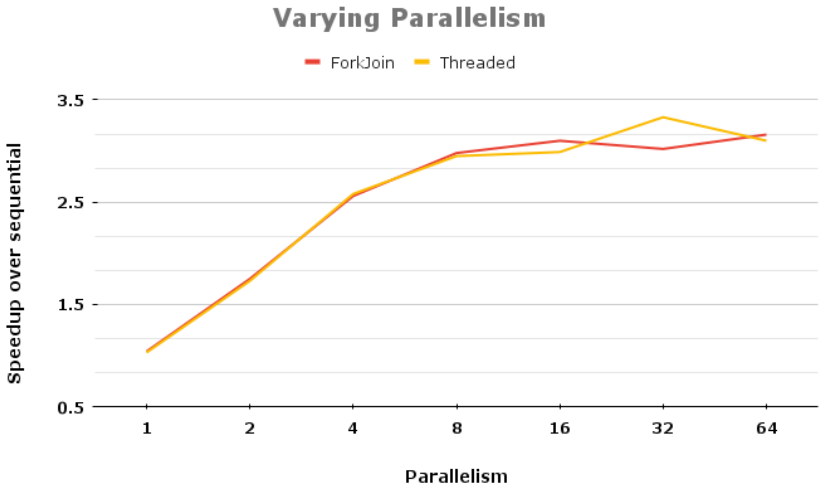

Figure 3.5: Comparing speed-up of parallel MergeSort strategies when varying parallelism

It can also be observed that **Threaded** and **ForkJoin** offered similar efficiencies, albeit quite lacking (Figure **3.6**). This is likely due to the naive implementation of parallel MergeSort opting to use a sequential merge procedure rather than a more performant parallelized procedure [Cormen et al. (2009), Chapter 27.3].

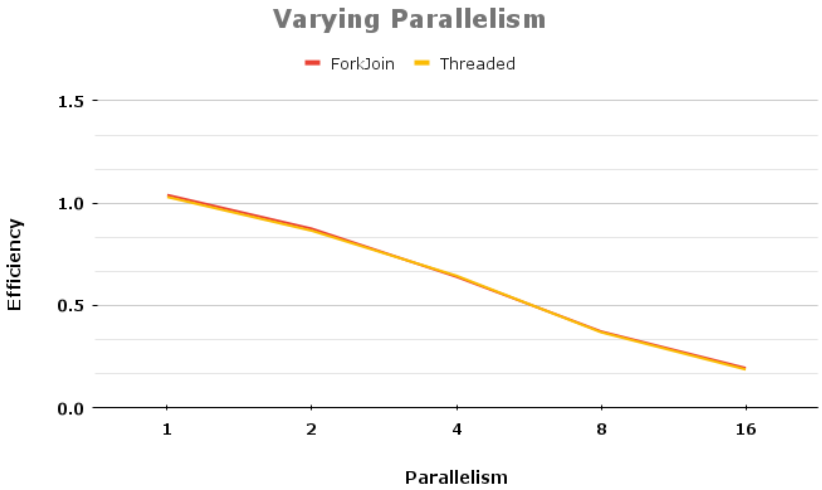

Figure 3.6: Comparing the efficiency of both parallel MergeSort strategies

3.3.2.2 Parallelism Cut-off

All tests were carried out with an input array size of $8,388,608$ and a fixed parallelism of 2048. The recursion cut-off value of **Sequential** was set to 8.

The results of the testing indicated that **ForkJoin** introduced less overhead when compared to **Threaded** (Figure **3.7**, Appendix **A.4**). This difference in overhead was incredibly significant at the lowest parallelism cut-off value, and gradually improved as the value was increased. Both strategies appeared to reach parity in performance around a cut-off value of $65,536$. The number of base-level sub-tasks can be calculated as:

$$T_{base} = \left\lceil \frac{InputSize}{ParallelismCutOff} \right\rceil$$

where T_{base} = total base-level sub-tasks

Given the input size, this means they reached parity when only 128 base-level sub-tasks were created, compared to the 2048 (**16x** more) created for the initial cut-off value of 4096. When considering the scheduling method of the **Threaded** strategy, this means only 255 threads were created for the former compared to 4095 for the latter ($\sim$**16x** more). The relative speed-up between these two cut-off values for **Threaded** was $\sim$**3.48x** which suggests that the number of threads created contributed significantly to the overhead observed.

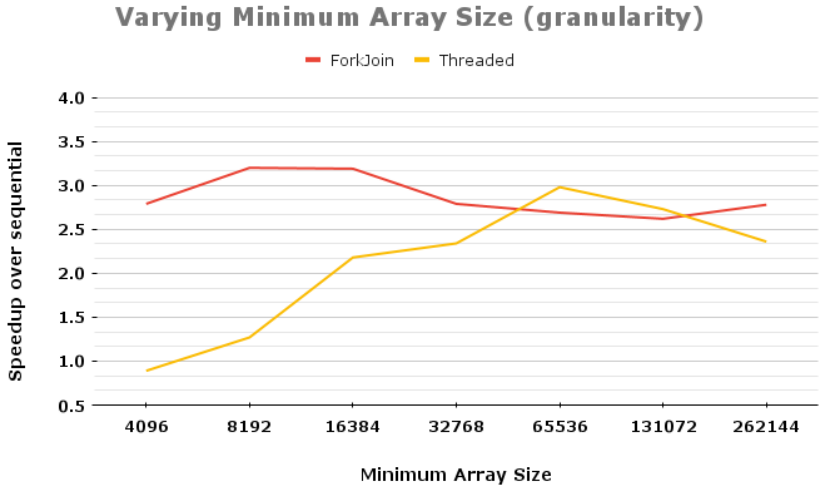

Figure 3.7: Comparing speed-up of parallel MergeSort strategies when varying parallelism cut-off

3.4 Fast Fourier Transform

The Fast Fourier Transform (FFT) algorithm [Cormen et al. (2009), Chapter 30.2] takes an even length sequence of complex values as input. It will then recursively

divide the input sequence into smaller subsequences of even-indexed and odd-indexed terms, until the length of the subsequences fall below a preset value. The Discrete Fourier Transform (DFT) of these subsequences is then computed directly, by some non-recursive approach, before they are combined into progressively larger sequences until the DFT of the original input sequence is computed.

3.4.1 Implementation

The implementation of the algorithm was slightly more involved than previous implementations. The **Sequential** strategy makes use of a naive DFT algorithm for subsequences that fall below the recursion cut-off. The naive DFT algorithm has an inferior worst case time complexity of $O(n^2)$ compared to $O(n \log n)$ for the FFT algorithm. However, this is acceptable for small values of n and even preferable as it saves on additional superfluous recursive calls. Additionally, it required the creation of a **Complex** class to represent complex values and associated operations. Furthermore, along with the necessary boilerplate code, some care was taken when implementing the parallel strategies to ensure no concurrent read/writes to array elements.

3.4.2 Results

3.4.2.1 Parallelism

All tests were carried out with an input sequence size of $8,388,608$. The recursion cut-off value of **Sequential** was set to 8.

The results of the testing indicated that both parallel strategies offered a modest speed-up over the sequential strategy. Furthermore, **ForkJoin** appeared to outperform **Threaded** consistently at parallelism values above 4 (Figure **3.8**, Appendix **A.5**). This may be indicative of **ForkJoin** having less overhead, or perhaps better work scheduling for this specific workload.

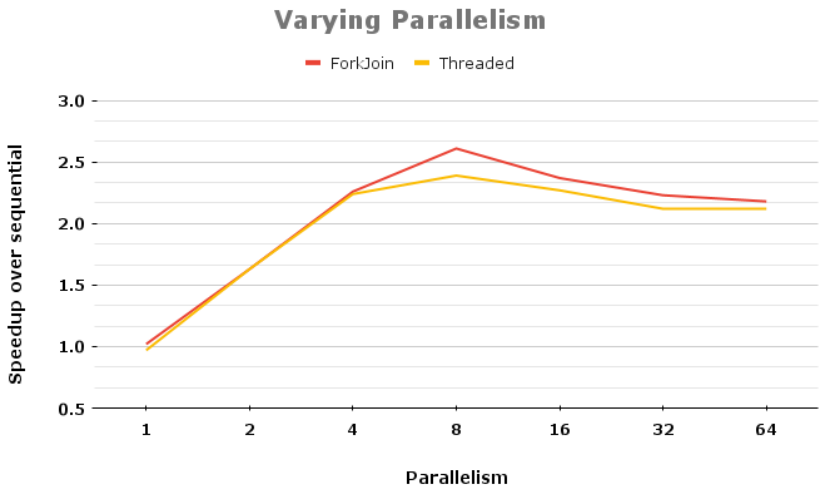

Figure 3.8: Comparing speed-up of parallel FFT strategies when varying parallelism

It can also be observed that **ForkJoin** and **Threaded** offered similar efficiencies, although they were rather poor (Figure **3.9**). This is likely due to the naive implementation of the sequential FFT [Frigo and Johnson (1998)] paired with an inefficient parallel implementation, not making use of various optimizations to improve the speed-up at scale [Gupta and Kumar (1993), Pippig (2013)].

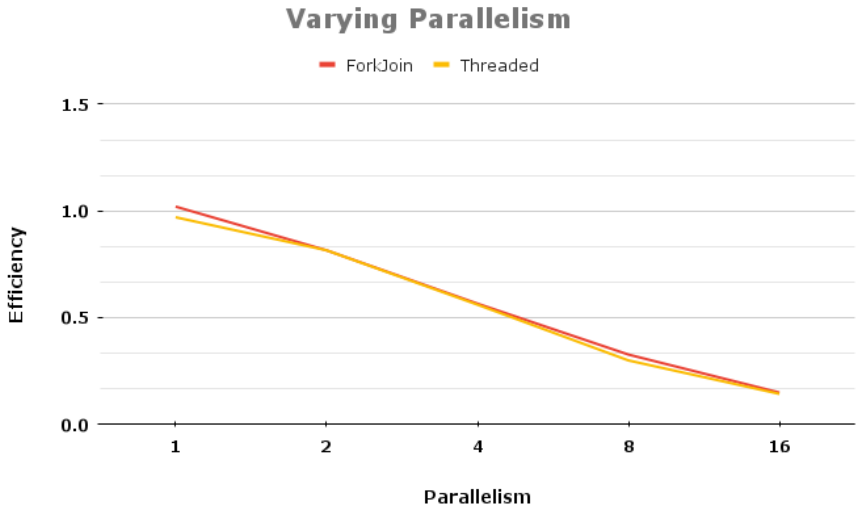

Figure 3.9: Comparing the efficiency of both parallel FFT strategies

3.4.2.2 Parallelism Cut-off

All tests were carried out with an input sequence size of $8,388,608$ and a fixed parallelism of 2048. The recursion cut-off value of **Sequential** was set to 8.

The results of the testing indicated that **ForkJoin** introduced less overhead when compared to **Threaded** (Figure **3.10**, Appendix **A.6**). The difference in overhead was incredibly significant at the lowest parallelism cut-off, with **Threaded** offering notably diminished performance even compared to **Sequential**. This gradually improved as the parallelism cut-off was raised. However, **ForkJoin** outperformed **Threaded** for almost all parallelism cut-off values tested except the largest value $(262,144)$ where they reached performance parity and **Threaded** marginally outperformed it.

Given the input size, this means they reached parity when only 32 base-level subtasks were created, compared to the 2048 (**64x** more) created for the initial cut-off value of 4096. When considering the scheduling method of the **Threaded** strategy, this means only 63 threads were created for the former compared to 4095 for the latter ($\sim$**77x** more). The relative speed-up between these two cut-off values was $\sim$**2.06x** which suggests the number of threads created contributed significantly to the overhead observed.

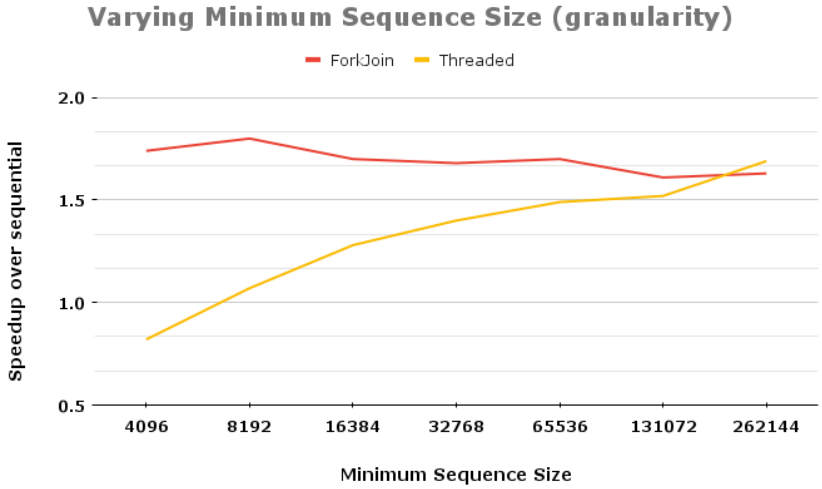

Figure 3.10: Comparing speed-up of parallel FFT strategies when varying parallelism cut-off

3.5 Strassen's Matrix Multiplication

Strassen's Matrix Multiplication algorithm [Strassen (1969)] takes two square matrices with matching dimension, NxN, as input. It then splits these matrices into smaller square submatrices with dimension $\frac{N}{2}$x$\frac{N}{2}$. It performs a series of additions and subtractions on these submatrices before performing 7 multiplications via recursive calls. The resulting matrices from these multiplications can then be recombined via further additions and subtractions to return the product of the original input matrices. The base case is invoked when the dimension of the input matrices falls below a preset value, in which case the multiplication is performed directly via a naive matrix multiplication algorithm.

3.5.1 Implementation

The implementation of the algorithm was the most complex out of all the algorithms tested. Implementing a naive parallel Strassen's algorithm required a large amount of working memory and was slow due to the large number of Matrix object allocations. As such, abstractions were implemented to aid the management of working memory, reduce the number of costly allocations, and keep the code clean.

- **Matrix** - An abstract class defining the operations that all subclasses must implement, for example: add, subtract, multiply, and helper methods to support interaction between subclasses.
- **ConcreteMatrix** - Serves as a wrapper around a 2-dimensional array of integer values that represents a matrix. This type of **Matrix** is the most costly to instantiate and backs all instantiated **MatrixView** objects.
- **MatrixView** - Serves as a wrapper around any subclass which extends the **Matrix** class, for example: a **ConcreteMatrix** or even another **MatrixView**. It allows

the programmer to specify a submatrix of an existing **Matrix** in memory then interact with it as if it were an independent **Matrix**.

If instantiated from another **MatrixView**, it will backtrack and follow references in order to find and store a direct reference to the **ConcreteMatrix** backing it. This hugely boosts performance and ensures the call chain for reads/writes to the backing matrix is as short as possible, hence reducing the computational cost of the abstraction.

Another optimization, in order to reduce the memory/compute required, was the use of Winograd's form [Winograd (1978), Knuth (2014)] instead of the form described by Strassen. This alternative form reduces the number of addition/subtraction operations required whilst still requiring 7 multiplications. This reduced the amount of 'scratch' working memory required for the sequential implementation and more significantly for parallel implementations, as each parallel thread requires its own non-shared working memory. As a result, the final implementation of each strategy is more memory efficient whilst also being non-destructive to the original input matrices.

3.5.2 Results

3.5.2.1 Parallelism

All tests were carried out with input matrix dimensions of 2048x2048. The recursion cut-off value of **Sequential** was set to 32.

The results of the testing indicated that both parallel strategies offered a significant speed-up over the sequential strategy. However, **ForkJoin** appeared to almost consistently outperform **Threaded** at parallelism values above 2 (Figure **3.11**, Appendix **A.7**). This may be indicative of **ForkJoin** having slightly less overhead. Furthermore, a large difference in the speed-up at parallelism values of 16 and 32 can be observed which likely signals a work scheduling deficiency of **Threaded** creating a performance bottleneck amongst worker nodes.

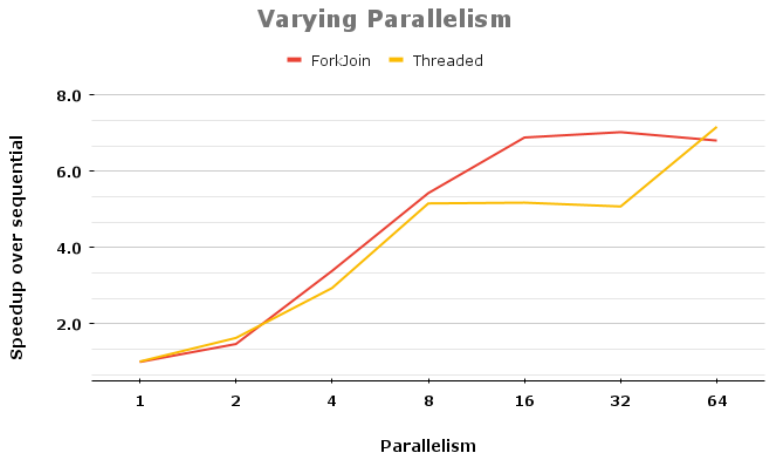

Figure 3.11: Comparing speed-up of Strassen strategies when varying parallelism

When inspecting the efficiency of both parallel strategies, it can be observed that **ForkJoin** offered consistently better efficiency than **Threaded** for parallelism values above 2 (Figure **3.12**). Furthermore, both strategies offered a reasonably good efficiency compared to the other algorithms tested, especially when considering the 8-core configuration of the test system.

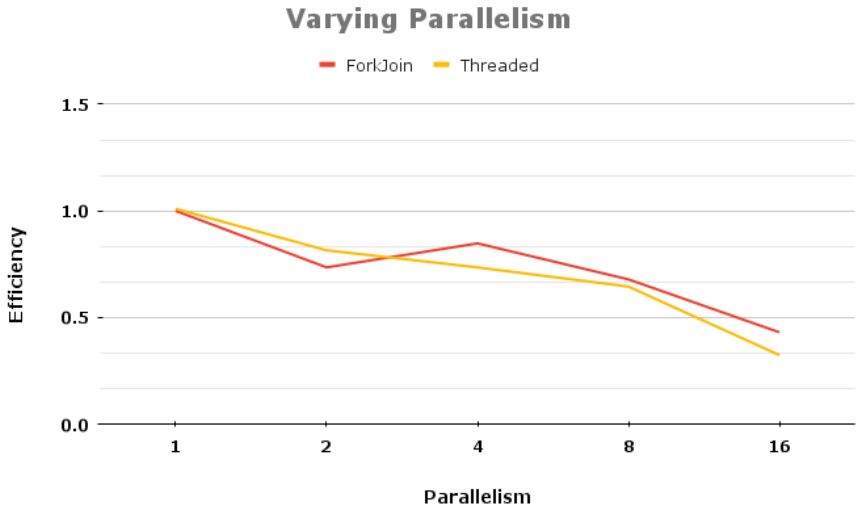

Figure 3.12: Comparing the efficiency of both parallel Strassen strategies

3.5.2.2 Parallelism Cut-off

All tests were carried out with input matrix dimensions of 2048x2048 and a fixed parallelism of 128. The recursion cut-off value of **Sequential** was set to 16.

The results of the testing indicated both strategies performed similarly. However, **Threaded** appeared to slightly outperform **ForkJoin** at lower parallelism cut-off values, with **ForkJoin** performing better at higher values (Figure **3.13**, Appendix **A.8**). This may be attributable to the slightly lower fixed value of parallelism compared to other tests, hence resulting in the parallel strategies not being stressed enough to expose significant differences in overhead.

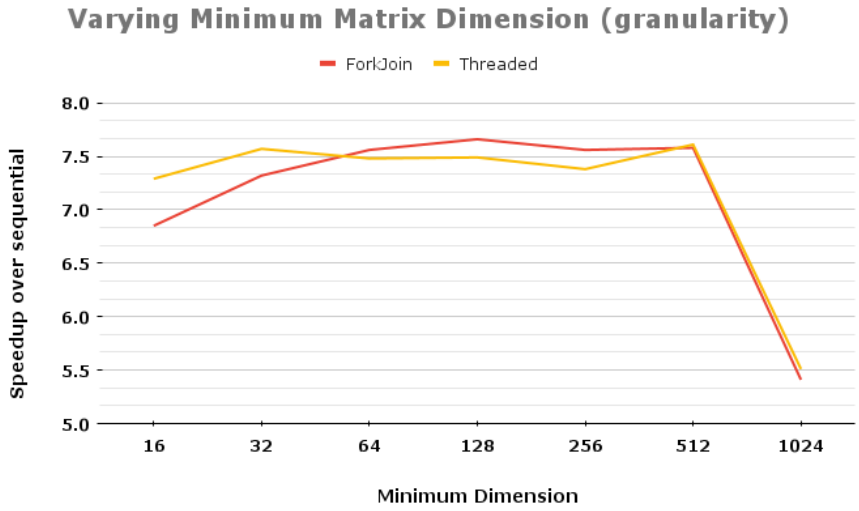

Figure 3.13: Comparing speed-up of parallel Strassen strategies when varying parallelism cut-off

3.6 Conclusions

From the experimental results, it can be concluded that **ForkJoin** is an effective, performant strategy closely matching or exceeding the performance of the **Threaded** strategy. Therefore, in most scenarios, it would be preferable for the programmer to use the Fork/Join framework rather than manually hand-threading, as it has proven a resilient abstraction requiring substantially less implementation effort for essentially the same or better performance.

There are many possible reasons for the **ForkJoin** strategy's improved performance when compared to the **Threaded** strategy. One reason may be the difference in overhead between creating threads for execution and creating tasks to add to the ForkJoinPool. In the official documentation, it is stated that each ForkJoinTask is 'much lighter weight than a normal thread', which likely contributes significantly to the reduced overhead [Oracle (2024)]. Another reason could be the fairer work scheduling method employed by the Fork/Join framework, opting to use work-stealing to more evenly distribute work amongst the ForkJoinPool's threads. This is in contrast to the **Threaded** strategy's work scheduling where work may only be shared by a worker thread with its children. This can lead to performance bottlenecks, particularly for algorithms with a high-degree of division for tasks, for example: Strassen's Matrix Multiplication (Figure **3.14**).

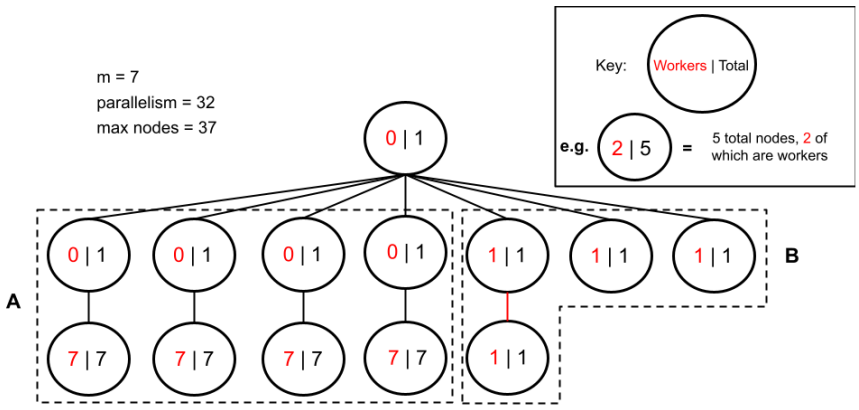

Figure 3.14: An example of how the **Threaded** strategy would handle an execution of Strassen's algorithm with a parallelism value of 32. This is an abbreviated notation, with each circle conceptually representing multiple nodes (threads).

Referring to Figure **3.14**, it can be observed that there are 37 nodes in total with 32 of these being worker nodes carrying out heavy computational work, in line with the degree of division and parallelism value set. In the group labelled **A**, all the worker nodes are leaf nodes as their parents successfully requested an allocation of 7 threads from the scheduling entity, hence they can delegate all the work to their children. However, in the group labelled **B** we see that one of the nodes received a partial allocation of 1 thread from the scheduling entity, whereas the others have been left with none. This creates a large bottleneck in performance, as it is evident that nodes in group **A** will finish their work before those in group **B** — the former has 28 worker nodes, whereas the latter only has 4. Furthermore, there currently exists no mechanism for nodes in group **B** to request additional worker resources once nodes in group **A** finish their work and are freed. Overall, this serves as a prime example of one of the many nuanced scheduling considerations that the Fork/Join framework effectively shields the programmer from.

After due consideration of the experimental results, and the conclusions reached above, it is clear that the Fork/Join framework would provide a suitable foundation on which to build a parallel algorithmic skeleton. Using the Fork/Join framework as the execution engine of the skeleton would not only provide adequate performance, but drastically simplify the implementation of the skeleton. In turn, this would allow more development time to be spent on the addition of useful features and the design of a robust user-facing interface.

Chapter 4

Divide-and-Conquer Skeleton

In this chapter '**Spooky D&C**', a parallel algorithmic skeleton built atop the Fork/Join framework is introduced. Subsequently, the design and implementation of the skeleton is presented. This is followed by a series of experiments, similar to those in Chapter **3**, designed to compare the performance of the skeleton to the previously presented pure **ForkJoin** strategy.

4.1 Spooky D&C

4.1.1 Overview

The primary goal of the skeleton is to hide the management of ForkJoinTasks and the ForkJoinPool from the programmer. This was decided as, during the implementation for Chapter **3**, a common pattern emerged between the **Threaded** and **ForkJoin** strategies. Specifically, the code provided by the programmer to implement the Runnable interface for Java Threads and extend the ForkJoinTask template for the Fork/Join framework bore striking similarities. Referring to Figure **4.1** it can be observed that, aside from obvious differences caused by their respective interfaces, the general structure of thread/task creation and completion is essentially the same.

```
FFTTask evenTask = new FFTTask(f_even, currLevel: CURR_LEVEL + 1);
FFTTask oddTask = new FFTTask(f_odd, currLevel: CURR_LEVEL + 1);

ArrayList<FFTTask> tasks = new ArrayList<>(List.of(evenTask, oddTask));
ArrayList<Thread> runningThreads = new ArrayList<>();
for (int i = 0; i < numThreads; i++) { // begin parallel threads
    Thread thread = new Thread(tasks.get(i));
    thread.start();
    runningThreads.add(thread);
}
for (int i = numThreads; i < tasks.size(); i++) { // run remaining sequentially
    tasks.get(i).run();
}
for (int i = 0; i < numThreads; i++) { // wait for parallel threads to finish
    try {
        runningThreads.get(i).join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

(a) Thread creation/completion for FFT **Threaded**

```
FFTTask evenTask = new FFTTask(f_even);
FFTTask oddTask = new FFTTask(f_odd);
invokeAll(evenTask, oddTask);
Complex[] F_even = (Complex[]) evenTask.join();
Complex[] F_odd = (Complex[]) oddTask.join();
```

(b) Task creation/completion for FFT **ForkJoin**

Figure 4.1: Comparing parallel strategies of FFT algorithm

4.1.2 Design

The design of **Spooky D&C** is rather straightforward, aiming to split up the D&C pattern into multiple logical chunks that can be easily implemented sequentially by the programmer. These logical chunks are then used by the skeleton to orchestrate a parallel execution of the algorithm on the Fork/Join framework. The only consideration of parallelism required by the programmer is that these logical chunks can and will be run in parallel, so any shared state/memory must be considered and synchronized within the chunks. These logical chunks are named and described at a high-level below:

- **Divider** - This is responsible for the task division logic of the algorithm. The programmer must provide: the input type of the algorithm, a function to determine when an input can be divided, and a function defining the actual division procedure that returns a collection of divided inputs.
- **Executor** - This is responsible for the base-level computation logic to solve subtasks. The programmer must provide: the input and output type of the algorithm, and a function that performs the base-level computation itself. Optionally, the programmer may provide an instance of another skeleton[1] to provide the base-level computation logic.
- **Conquerer** - This is responsible for the recombination logic for subtask results. The programmer must provide: the output type of the algorithm, and a function that returns a single output from a collection of outputs.

Furthermore, **Spooky D&C** ensures that the ordering of divided inputs from the **Divider** is maintained when they are subsequently returned as outputs to the **Conquerer**.

4.1.3 Implementation

All relevant implementation code can be found in the following GitHub repository: **https://github.com/barbourja/dac-skeleton**.

Following on from the design, the actual implementation of **Spooky D&C** is relatively simple so as to be accessible to more experienced programmers who may want to dig into its internals (Figure **4.2**). Key classes are detailed below:

- **GenericDivider<I>** - This is an abstract class implementing the **IDivider<I>** interface that the programmer must extend and provide a concrete implementation of. It requires the programmer to provide the following methods: **canDivide** and **divisionProcedure**.
- **GenericExecutor<I, O>** - This is an abstract class implementing the **IExecutor<I, O>** interface that the programmer must extend and provide a concrete implementation of. It requires the programmer to provide an **execute** method.
- **GenericConquerer<O>** - This is an abstract class implementing the **IConquerer<O>** interface that the programmer must extend and provide a concrete implementation of. It requires the programmer to provide a **conquer** method.

[1] referred to as 'nesting'

- **GenericDaCTask<I, O>** - This class extends the **RecursiveTask<O>** abstract class provided by the Fork/Join framework. The constructor accepts an input value along with instantiated objects that implement the **IDivider**, **IConquerer**, and **IExecutor** interfaces. It provides an implementation of the **compute()** method that relies on the aforementioned logical chunks provided in the constructor. This method will then either directly solve the input or recursively spawn more **GenericDaCTasks** to be executed by the associated ForkJoinPool, effectively implementing the parallel D&C pattern.

  DaCSkeleton<I, O> - This is a concrete class that implements the **IExecutor<I, O>** interface. The constructor accepts a parallelism value along with instantiated objects that implement the **IDivider**, **IConquerer**, and **IExecutor** interfaces. The primary purpose of this class is to instantiate the ForkJoinPool required for execution and submit the initial **GenericDaCTask** required for an input to be executed on the pool. Furthermore, it is worth noting that if a skeleton is nested, the execution will take place on the ForkJoinPool of the 'root' skeleton.

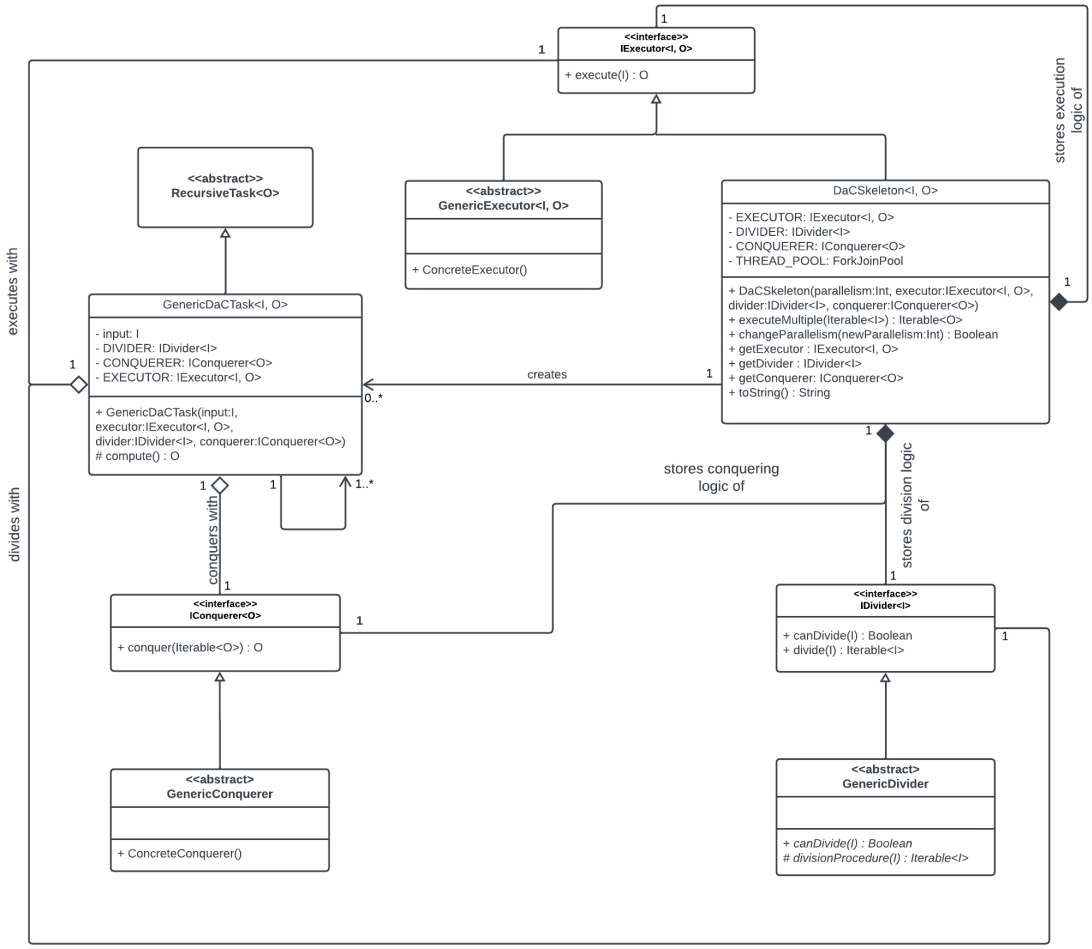

Figure 4.2: UML class diagram of **Spooky D&C**

4.1.4 Testing

Although the testing implementation had to be adapted to work for **Spooky D&C** skeletons, via a new **GenericSkeletonTest** abstract class, the testing methodology was identical to that detailed in Section **3.1.3**. Furthermore, the same testing configuration specified in Section **3.1.4** was utilized.

It is difficult to quantify the relative difficulty of implementing the **Skeleton** strategy when compared to the previous **ForkJoin** strategy. Therefore, this is done via a relatively simplistic 'lines of code' (LoC) metric, for which the definition necessarily varies between the strategies:

N.B. Multi-line comments are excluded for all line counts

- **Skeleton** - The number of lines of code required to extend the three generic classes passed into the skeleton's constructor.
- **ForkJoin** - A sum of the following:
  - The number of lines of code contained within the **ForkJoin** and **Sequential** classes, from the class declaration line to the end of the **execute()** method before the other @Override functions begin. This is done as **Sequential** essentially serves as an executor analogue for **ForkJoin**.
  - The number of lines of any methods, contained within static utility classes, that are called by either **Sequential** or **ForkJoin**. This is done since all the required utility methods for the **Skeleton** strategy are declared within the code extending the three generic classes.

It is worth noting, however, that the purpose of the skeleton isn't necessarily to significantly reduce the amount of code the programmer must write. Rather the primary benefit is the reduction in complexity provided by the effective separation of concerns, allowing the programmer to primarily focus on their D&C logic and data instead of parallel programming constructs. This allows for the fast prototyping and implementation of efficient algorithms leveraging parallelism; any reduction in the lines of code written should therefore be considered an auxiliary benefit.

4.2 MergeSort

Algorithm as described previously in Section **3.3**.

4.2.1 Implementation

The implementation was straightforward aside from the creation of a new class, **ArrayView**, designed to essentially bundle together the necessary inputs (backing array and start/end indices). The implementation required no consideration or direct reference to the Fork/Join framework. The LoC required for the **Skeleton** strategy was **106** compared to **132** for **ForkJoin**.

4.2.2 Results

4.2.2.1 Parallelism

All tests were carried out with an input array size of 8,388,608. The recursion cut-off value of the **SequentialMergeSortExecutor** and **Sequential** was set to 8.

The results of the testing indicated that the **Skeleton** strategy offered an appreciable speed-up over the sequential strategy. Furthermore, the **Skeleton** strategy broadly matched the performance of the **ForkJoin** strategy across all values of parallelism tested, even marginally outperforming it for values of parallelism up to 8 (Figure **4.3**, Appendix **A.9**). This was in line with expectations as a decent amount of the work of the skeleton, incurred by the use of generic types, takes place during compilation. Therefore, these results imply that the additional overhead of the skeleton during runtime is relatively low for reasonable values of parallelism and parallelism cut-off.

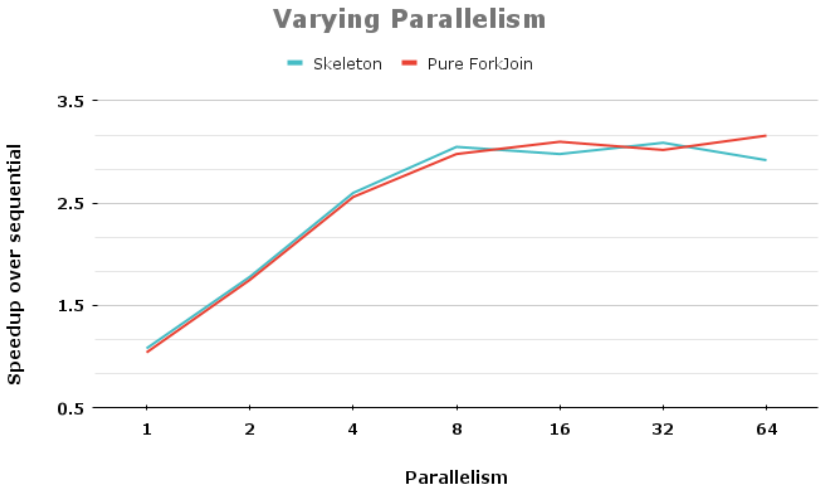

Figure 4.3: Comparing speed-up of Skeleton and ForkJoin MergeSort strategies when varying parallelism

4.2.2.2 Parallelism Cut-off

All tests were carried out with an input array size of 8,388,608 and a fixed parallelism of 2048. The recursion cut-off value of the **SequentialMergeSortExecutor** and **Sequential** was set to 8.

The results of the testing indicated that **ForkJoin** introduced slightly less overhead when compared to the **Skeleton** strategy (Figure **4.4**, Appendix **A.10**). The difference in overhead was more significant at lower parallelism cut-off values. However, **Skeleton** appeared to reach parity in performance and slightly outperform **ForkJoin** at parallelism cut-off values above 65,536.

Given the input size, this means they reached parity when only 128 base-level subtasks were created, compared to the 2048 (**16x** more) created for the initial cut-off value of

4096. The relative speed-up between these two cut-off values for **Skeleton** was ∼**1.19x**, which is small and doesn't necessarily indicate a large overhead due to the number of sub-tasks created. However, a slow-down can also be observed between the same cut-off values for **ForkJoin** which perhaps signals the need for further testing under this configuration.

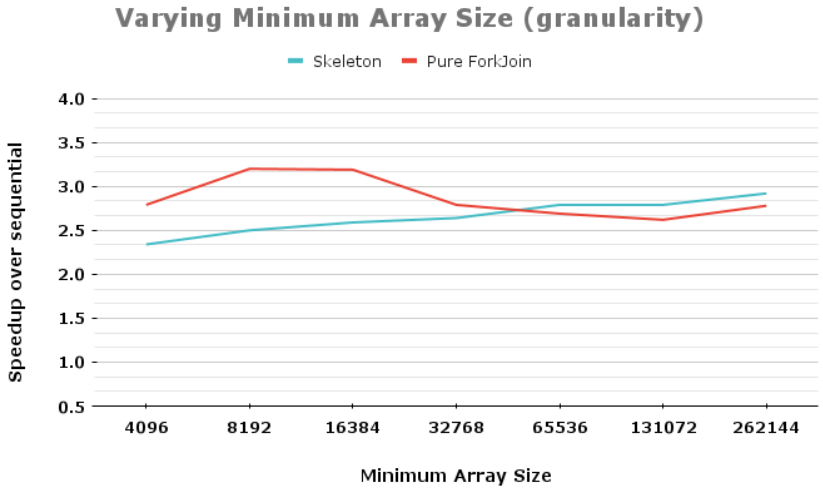

Figure 4.4: Comparing speed-up of Skeleton and ForkJoin MergeSort strategies when varying parallelism cut-off

4.3 Strassen's Matrix Multiplication

Algorithm as described previously in Section **3.5**.

4.3.1 Implementation

The implementation was straightforward, aside from having to add a new method to the **Matrix** class in order to overcome a variable scoping issue with the **StrassensConquerer** implementation. Furthermore, the implementation required the introduction of a new **StrassensInput** class to effectively bundle together the required inputs (two input matrices and the result matrix). The implementation required no consideration or direct reference to the Fork/Join framework. The LoC required for the **Skeleton** strategy was **180** compared to **187** for **ForkJoin**.

4.3.2 Results

4.3.2.1 Parallelism

All tests were carried out with input matrix dimensions of 2048x2048. The recursion cut-off value of the **SequentialStrassensExecutor** and **Sequential** was set to 32.

The results of the testing indicated that the **Skeleton** strategy offered a significant

speed-up over the sequential strategy. Furthermore, **Skeleton** appeared to outperform **ForkJoin** by a reasonable margin for all values of parallelism tested (Figure **4.5**, Appendix **A.11**). There are many possible reasons for this. For example: the more prescriptive structure of computation enforced by the skeleton could have resulted in more optimizations being applied during compilation. Regardless, these results are promising evidence of the potential performance benefits of the skeleton approach.

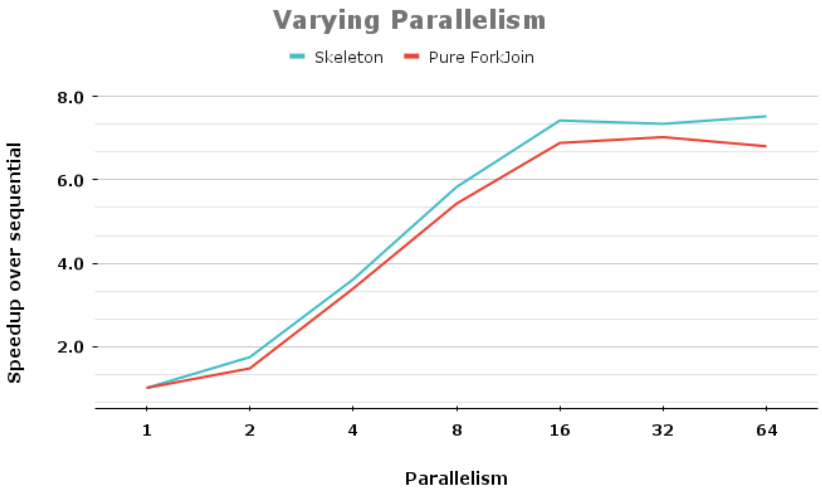

Figure 4.5: Comparing speed-up of Skeleton and ForkJoin Strassen strategies when varying parallelism

4.3.2.2 Parallelism Cut-off

All tests were carried out with input matrix dimensions of 2048x2048. The recursion cut-off value of the **SequentialStrassensExecutor** and **Sequential** was set to 16.

The results of the testing indicated that the **ForkJoin** strategy introduced slightly less overhead when compared to the **Skeleton** strategy at lower parallelism cut-off values (Figure **4.6**, Appendix **A.12**). However, **Skeleton** appeared to reach parity in performance and slightly outperform **ForkJoin** at parallelism cut-off values above 256, despite a large drop-off in performance for both. Furthermore, it can be observed that both **Skeleton** and **ForkJoin** exhibited a similar rate of speed-up for increasing parallelism cut-off values below 128. This signals the ability of **Skeleton** to keep up with, and broadly match the performance, of **ForkJoin** in this test.

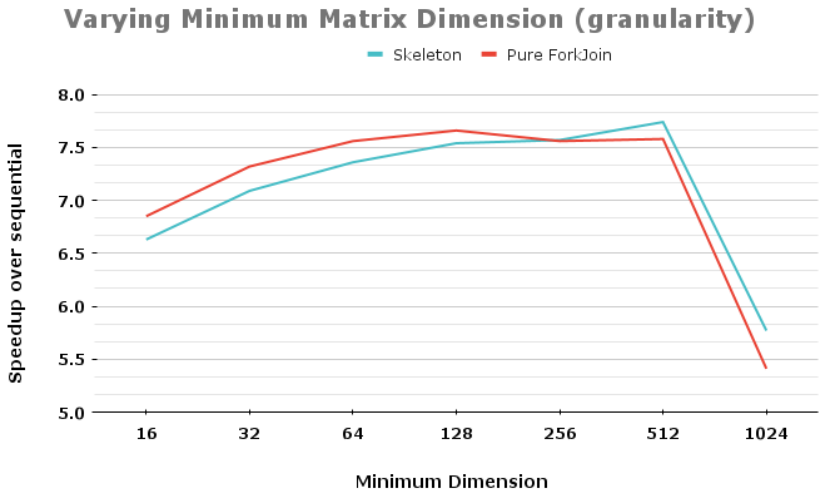

Figure 4.6: Comparing speed-up of Skeleton and ForkJoin Strassen strategies when varying parallelism cut-off

4.4 Conclusions

From the experimental results it appears that **Spooky D&C** is capable of matching, or even improving upon, the performance of the pure **ForkJoin** strategy under a wide range of conditions. This was expected, as it is built directly on top of the Fork/Join framework and designed to introduce a minimal amount of overhead during runtime.

The implementation code for the **Skeleton** strategies provided convincing evidence of **Spooky D&C** being effective as an abstraction. None of the **Skeleton** strategies required consideration or direct reference to the Fork/Join framework, nor other related parallel programming constructs. Furthermore, each **Skeleton** strategy required fewer lines of code than the alternative **ForkJoin** strategy for both algorithms tested. Notably, the reduction in line count for Strassen's algorithm was not very significant. That is not to say that the skeleton is perfect; as touched on before, the programmer must consider using manual synchronization for shared state where applicable. However, the splitting of each algorithm by the skeleton into pre-defined logical chunks made it easier to write modular, clear code focussed entirely on the flow of data and core logic rather than the orchestration of parallel computing activities.

One of the primary benefits of **Spooky D&C** lies in the simplicity of the skeleton itself, provided by using the Fork/Join framework as a foundation. This, in combination with the logical mapping of D&C constructs to components of the skeleton, leaves it easily accessible to programmers who wish to delve further into its internals. As a result, programmers may extend the generic templates or otherwise alter the skeleton to better suit their needs, hence addressing the problem of limited flexibility faced by other skeletons. This would hopefully prompt further adoption of **Spooky D&C** as this alleviates concerns about the conceptual overhead that would be incurred should the needs of the programmer eventually outgrow the functionality offered by the skeleton.

A slightly less obvious benefit of **Spooky D&C** to the programmer is that it decouples the client code from the underlying method of execution. Therefore, with a few tweaks, **Spooky D&C** could be modified to support a variety of different execution methods in future. For example: if a new, faster alternative to the Fork/Join framework was released, then **Spooky D&C** could be modified to offer this with minimal changes required to the client code. Additionally, the design of **Spooky D&C** leaves it open to further extension, for example: adding the ability for the skeleton to automatically control task granularity, similar to the **auto_partitioner** in Intel's TBB [Robison et al. (2008)].

Chapter 5

Conclusion

In this chapter, the main body of the report is summarized and conclusions are drawn about the results obtained (Section **5.1**). The work carried out is then critically evaluated, with a focus on the lessons learned whilst carrying the project through to completion (Section **5.2**). Finally, deficiencies in the presented solution are identified when compared to existing parallel algorithmic skeletons, which paves the way for future work to be carried out (Section **5.3**).

5.1 Summary

In Chapter **2**, the challenges faced by programmers due to the advent of multicore general purpose microprocessors were introduced. Subsequently, a brief overview of parallelism and the parallel programming paradigm introduced to overcome these challenges was presented. Then a more specialized view of parallel programming in Java was detailed, including an introduction to Java Threads and the Fork/Join framework. Finally, the concept of algorithmic patterns was elaborated upon, with a focus on the divide-and-conquer (D&C) pattern, followed by the natural progression of these into parallel algorithmic skeletons.

In Chapter **3**, a series of experiments on a collection of D&C algorithms were presented. These experiments were designed to compare the efficacy of implementations based on the Fork/Join framework against manually hand-threaded variants. The overarching goal was to determine the best way to proceed in building a D&C focussed parallel algorithmic skeleton in Java, presented in Chapter **4**. A myriad of conceptual problems were solved in order to carry this work out. Notably, this included the design/implementation of: a thread scheduling mechanism built on Java Threads, an abstraction to clearly and efficiently carry out recursive matrix operations, and a generalized testing framework to ensure consistency between tests of implementation variants. Overall, the results of the experimentation were quite conclusive and pointed towards the Fork/Join framework being a remarkably capable solution. It managed to beat the hand-threaded alternative in both implementation effort and, in many cases, performance.

In Chapter **4**, **Spooky D&C**, a lightweight parallel algorithmic skeleton built upon the Fork/Join framework, was introduced. The design and implementation of **Spooky D&C** was guided by the results and experienced gained from the experiments conducted in Chapter **3**. In order to achieve this, a series of conceptual engineering problems were solved. This involved: generalizing the expression of D&C algorithms such that they can be written sequentially but run in parallel; creating a modular, extensible skeleton architecture; and designing a cohesive interface for programmers to leverage. An overview of the design and implementation details were presented, followed by a series of experiments to compare the efficacy of implementations using **Spooky D&C** to the baseline pure Fork/Join implementations. The results of the testing indicated that **Spooky D&C** was capable of broadly matching, and in some cases exceeding, the performance of the pure Fork/Join implementations. Furthermore, **Spooky D&C** was found to achieve its primary goals of hiding the Fork/Join framework from the programmer and requiring less implementation effort than using the framework directly, from both a qualitative and quantitative perspective.

5.2 Lessons Learned

A fair amount of time was spent getting familiar with the Fork/Join framework and Java Threads, which in retrospect was unavoidable. Regardless, this resulted in slower progress than desired whilst completing the implementation work for Chapter **3**. Furthermore, this was compounded by the need for a fair testing framework, which resulted in significant refactoring and multiple rewrites to produce the required generalizations to achieve this. This could have been avoided by spending more time on the initial planning and placing a greater focus on the testability of code being written.

Whilst implementing the thread scheduling logic, used for the **Threaded** implementations, a series of design decisions had to be made. The primary guiding philosophy of these decisions was to provide a fair competitor, and comparison, to implementations based upon the Fork/Join framework. Initially, the scheduling logic was defined on the premise that the user-provided parallelism value would define the maximum number of threads that could exist at any one time. After running the entire test suite and collecting results, it was later determined that this was an unfair comparison. As a result, the concept of worker threads was introduced (see Figure **3.1**), which required a significant rewrite of the scheduling logic along with a re-run of the test suite. This produced a fairer comparison that could be considered more true to the definition of 'parallelism' in this context, as it now defined threads performing heavy computational work for both **Threaded** and **ForkJoin** implementations.

The implementation of **Spooky D&C** went smoothly, in part due to the lessons learned from the Chapter **3** implementation. It was built with more consideration to testability, hence useful generalizations were introduced at the start of development to be worked in with the implementation code written later. Additionally, the interface offered by the Fork/Join framework was well documented and intuitive to work with after getting to grips with its intricacies.

5.3 Future Work

Largely, the future work proposed is focussed around refactoring **Spooky D&C** and making improvements to it in order to bring it in line with other parallel algorithmic skeleton offerings. However, there were also a few opportunities for further experimentation left unexplored due to the time and content limitations imposed on this project:

- Refactor **Spooky D&C** to further simplify the design and improve extensibility. Specifically, introduce a generalized 'skeleton' interface and make further use of object-oriented design patterns to reduce coupling. For example, referring to Figure **4.2**, there is unnecessary coupling between **DaCSkeleton<I, O>** and the interfaces of the logical chunks that could be solved by using a creational pattern such as the factory[1] pattern.
- Implement generalizations in **Spooky D&C** that make the execution engine swappable based on the programmer's preference. This could allow for further experimentation to compare the centralized thread scheduling logic from Chapter **3** directly to the Fork/Join framework in the context of **Spooky D&C**.
- Implement automatic task granularity control in **Spooky D&C**, similar to the **auto_partitioner** offered in Intel's TBB [Robison et al. (2008)].
- Implement a diagnostic component for **Spooky D&C** to allow programmers to more easily identify and fix deficiencies in their code, similar to **Calcium's** *blame system* [Caromel and Leyton (2007)].
- Conduct experiments to compare the performance of **Spooky D&C** to the current D&C skeleton offering from **Skandium**.
- Conduct a user study with **Spooky D&C** to determine where improvements can be made and to gain more insight into its usability for programmers.
- Conduct experiments on the efficacy of the matrix abstraction introduced in Chapter **3** to determine the overhead introduced when compared to the naive approach of performing all operations directly on the underlying data structure.

[1] https://refactoring.guru/design-patterns/abstract-factory

Bibliography

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450378956. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architecturesnbsp; - SPAA '98*, page 119–129, Jun 1998. doi: 10.1145/277651.277678.

Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009. doi: 10.1109/MSP.2009.934110.

Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, pages 72–81, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74466-5.

Denis Caromel, Christian Delbé, Alexandre Di Costanzo, and Mario Leyton. Proactive: an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12:issue–1, 2006.

Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2003.12.002. URL https://www.sciencedirect.com/science/article/pii/S0167819104000080.

Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization*, pages 57–66, 2008. doi: 10.1109/IISWC.2008.4636091.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *In-*

troduction to Algorithms, Third Edition. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

Marco Danelutto, Gabriele Mencagli, Massimo Torquati, Horacio González-Vélez, and Peter Kilpatrick. Algorithmic skeletons and parallel design patterns in mainstream parallel programming. *International Journal of Parallel Programming*, 49:177–198, 2021.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi: 10.1109/JSSC.1974.1050511.

M. Frigo and S.G. Johnson. Fftw: an adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 3, pages 1381–1384 vol.3, 1998. doi: 10.1109/ICASSP.1998.681704.

Carlos H. González and Basilio B. Fraguela. A generic algorithm template for divide-and-conquer in multicore systems. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 79–88, 2010. doi: 10.1109/HPCC.2010.24.

Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010. doi: https://doi.org/10.1002/spe.1026. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1026`.

Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley Publishing, 1st edition, 2013. ISBN 1118290275.

A. Gupta and V. Kumar. The scalability of fft on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, 1993.

John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

Donald E Knuth. *The Art of Computer Programming: Seminumerical Algorithms, Volume 2*. Addison-Wesley Professional, 2014.

Alexey Kukanov and Michael J Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. doi: 10.1109/MC.2006.180.

Joeffrey Legaux, Frédéric Loulergue, and Sylvain Jubertie. Development effort and performance trade-off in high-level parallel programming. pages 162–169, 2014. doi: 10.1109/HPCSim.2014.6903682.

Mario Leyton and José Piquer. Skandium: Multi-core programming with algorithmic skeletons. pages 289–296, 02 2010. doi: 10.1109/PDP.2010.26.

Gordon Moore. Moore's law. *Electronics Magazine*, 38(8):114, 1965.

Gordon E Moore et al. Progress in digital integrated electronics. In *Electron devices meeting*, volume 21, pages 11–13. Washington, DC, 1975.

Oracle. ForkJoinTask (Java Platform SE 8). `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html`, 2024. [online; accessed 18-March-2024].

Jolan Philippe and Frédéric Loulergue. Pyske: Algorithmic skeletons for python. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 40–47, 2019. doi: 10.1109/HPCS48598.2019.9188151.

Michael Pippig. Pfft: An extension of fftw to massively parallel architectures. *SIAM Journal on Scientific Computing*, 35(3):C213–C236, 2013. doi: 10.1137/120885887.

James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

Arch Robison, Michael Voss, and Alexey Kukanov. Optimization via reflection on work stealing in tbb. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008. doi: 10.1109/IPDPS.2008.4536188.

Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4): 354–356, 1969.

Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in science & engineering*, 19(2):41–50, 2017.

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 392–403, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916980. doi: 10.1145/223982.224449. URL `https://doi.org/10.1145/223982.224449`.

Shmuel Winograd. On computing the discrete fourier transform. *Mathematics of computation*, 32(141):175–199, 1978.

Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

Appendix A

Raw Results and Additional Charts

A.1 QuickSelect Parallelism Testing Results

QuickSelect (Parallelism)

Input Size = 67108864		Raw Run Results												
Sequential														
Recursion Cut-off	Parallelism	Run Number										Average (ms)		
		1	2	3	4	5	6	7	8	9	10			
1	1	1805	1643	1777	1204	1177	4720	898	2792	2791	1479	2029		
ForkJoin														
Parallelism Cut-off	Parallelism	Run Number										Average (ms)	Speedup	Efficiency
		1	2	3	4	5	6	7	8	9	10			
67108864	1	3148	1336	766	1189	712	813	4261	3014	1117	1788	1814	1.12	1.12
33554432	2	3919	1906	1154	2761	986	2427	1338	3981	2986	1805	2326	0.87	0.44
16777216	4	1963	1807	1800	3777	1693	3445	2812	2456	1609	2878	2424	0.84	0.21
8388608	8	3394	2140	2384	4429	627	2255	1960	2924	3144	2133	2539	0.80	0.10
4194304	16	2176	2771	1577	2071	1237	2250	2784	2198	3558	1073	2170	0.94	0.06
2097152	32	3646	2710	2522	4488	1300	2017	1070	2756	1748	576	2283	0.89	0.03
1048576	64	3721	1573	1256	1831	3736	802	2683	2200	1745	2132	2168	0.94	0.01
Threaded														
Parallelism Cut-off	Parallelism	Run Number										Average (ms)	Speedup	Efficiency
		1	2	3	4	5	6	7	8	9	10			
67108864	1	2109	2101	1055	3071	1302	2202	2186	1916	1778	662	1838	1.10	1.10
33554432	2	3914	2027	2272	2878	2549	1455	1962	1693	2588	1610	2295	0.88	0.44
16777216	4	1684	2177	3378	2970	790	1522	1310	2282	3809	1082	2100	0.97	0.24
8388608	8	3129	1992	1597	3230	1000	1370	3190	2922	1759	682	2087	0.97	0.12
4194304	16	3524	1516	1953	3261	441	2727	3992	3697	3487	162	2476	0.82	0.05
2097152	32	2644	2073	644	5542	1885	2706	2444	2738	1795	4038	2651	0.77	0.02
1048576	64	2229	1761	5238	2408	1741	2070	3074	1612	3748	2105	2599	0.78	0.01

Figure A.1: Raw results of QuickSelect testing when varying parallelism

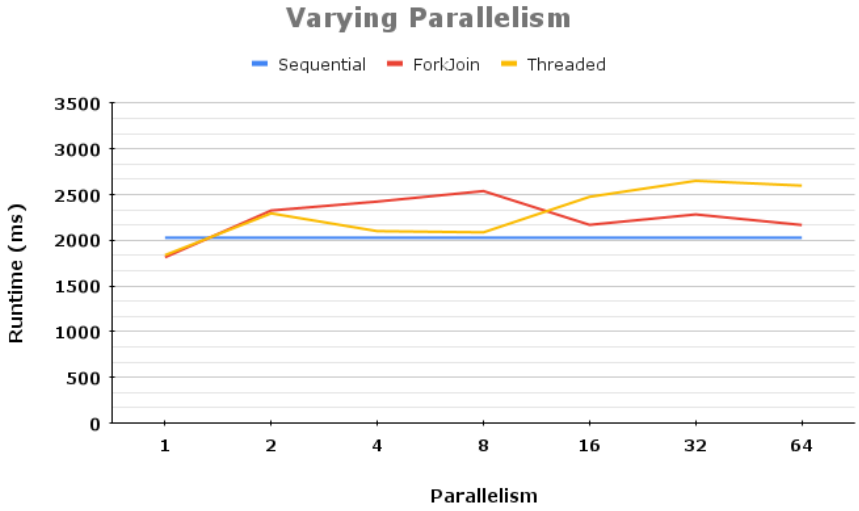

Figure A.2: Comparing runtime of QuickSelect strategies when varying parallelism

A.2 QuickSelect Parallelism Cut-Off Testing Results

QuickSelect (Min Size)

Input Size = 67108864		Raw Run Results											
Sequential													
Parallelism	Recursion Cut-off	Run Number										Average (ms)	
		1	2	3	4	5	6	7	8	9	10		
1	1	2211	1817	2157	1477	1130	1746	1666	4033	2121	940	1930	
ForkJoin													
Parallelism	Parallelism Cut-off	Run Number										Average (ms)	Speedup
		1	2	3	4	5	6	7	8	9	10		
2048	32768	3100	1983	2221	2070	2082	3184	1570	4424	1866	838	2334	0.83
2048	65536	2708	2523	2410	2031	1188	1461	2485	1744	2001	1013	1956	0.99
2048	131072	3671	1968	1926	2617	5338	852	1422	1910	4394	4951	2905	0.66
2048	262144	2133	2449	3128	3826	602	718	2969	3936	4557	325	2464	0.78
2048	524288	2592	2807	1929	2935	3786	5270	2123	1509	2418	1051	2642	0.73
2048	1048576	3879	2097	2842	2313	3835	791	3350	1639	3501	487	2473	0.78
2048	2097152	2285	1547	1956	3460	1152	1485	1279	3226	2455	2804	2165	0.89
Threaded													
Parallelism	Parallelism Cut-off	Run Number										Average (ms)	Speedup
		1	2	3	4	5	6	7	8	9	10		
2048	32768	1564	2257	3442	2798	1013	3676	3196	2125	2116	787	2297	0.84
2048	65536	3441	1688	1911	4803	1892	1199	6780	3958	1647	1415	2873	0.67
2048	131072	2236	2273	1452	3301	1225	1646	3177	1827	3540	2780	2346	0.82
2048	262144	1756	2037	2005	2609	2645	3412	2860	2542	5726	2569	2816	0.69
2048	524288	3463	1364	2072	4191	3204	3552	3050	4098	1513	3142	2965	0.65
2048	1048576	1616	2520	1714	2213	712	898	739	3583	2193	1913	1810	1.07
2048	2097152	2536	2744	1320	3450	1891	492	1748	1442	1646	4119	2139	0.90

Figure A.3: Raw results of QuickSelect testing when varying parallelism cut-off

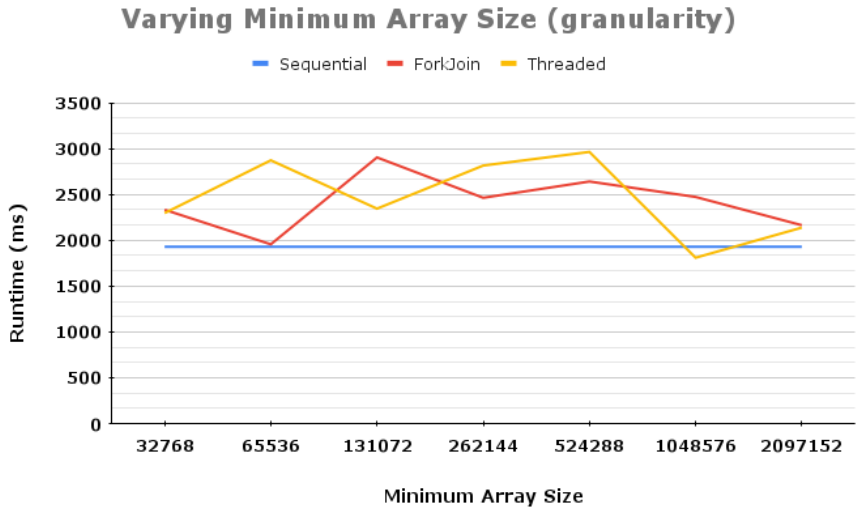

Figure A.4: Comparing runtime of QuickSelect strategies when varying parallelism cut-off

A.3 MergeSort Parallelism Testing Results

MergeSort (Parallelism)

Input Size = 8388608		Raw Run Results												
		Sequential												
Recursion Cut-off	Parallelism	Run Number										Average (ms)		
		1	2	3	4	5	6	7	8	9	10			
8	1	2639	2467	2511	2575	2502	2535	2535	2577	2560	2467	2537		
		ForkJoin												
Parallelism Cut-off	Parallelism	Run Number										Average (ms)	Speedup	Efficiency
		1	2	3	4	5	6	7	8	9	10			
8388608	1	2558	2443	2535	2256	2456	2321	2456	2430	2463	2434	2435	1.04	1.04
4194304	2	1532	1289	1346	1519	1461	1386	1495	1648	1397	1456	1453	1.75	0.88
2097152	4	868	1040	885	929	1084	1150	1196	937	862	955	991	2.56	0.64
1048576	8	806	916	884	971	919	785	665	806	800	957	851	2.98	0.37
524288	16	824	904	868	813	840	765	880	689	899	713	820	3.10	0.19
262144	32	863	891	787	840	714	785	680	850	1078	904	839	3.02	0.09
131072	64	851	1031	851	940	890	462	440	900	943	714	802	3.16	0.05
		Threaded												
Parallelism Cut-off	Parallelism	Run Number										Average (ms)	Speedup	Efficiency
		1	2	3	4	5	6	7	8	9	10			
8388608	1	2377	2669	2444	2455	2460	2433	2443	2432	2414	2460	2459	1.03	1.03
4194304	2	1385	1427	1425	1496	1559	1502	1711	1322	1478	1361	1467	1.73	0.87
2097152	4	956	1082	1113	889	1066	1100	907	841	934	942	983	2.58	0.65
1048576	8	788	994	675	863	653	920	831	930	1014	927	860	2.95	0.37
524288	16	876	949	866	926	602	888	866	876	844	781	847	2.99	0.19
262144	32	871	732	851	882	750	549	759	689	878	651	761	3.33	0.10
131072	64	739	723	623	672	759	840	1074	706	848	1208	819	3.10	0.05

Figure A.5: Raw results of MergeSort testing when varying parallelism

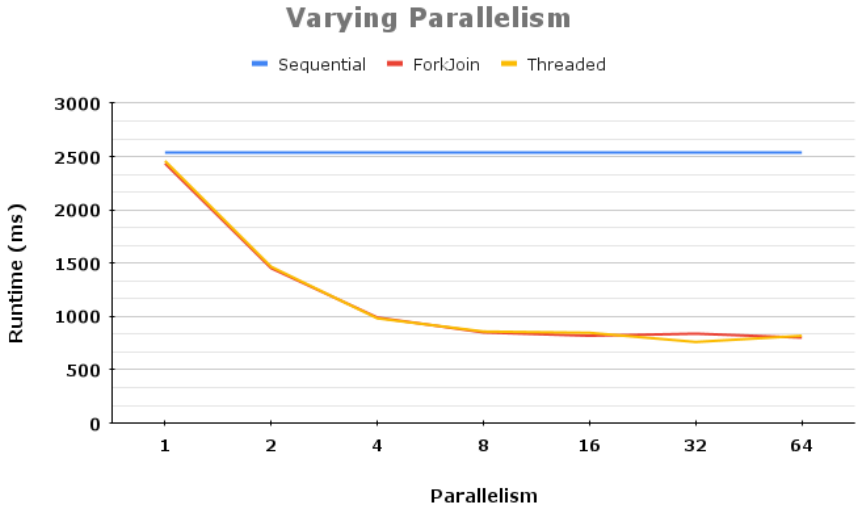

Figure A.6: Comparing runtime of MergeSort strategies when varying parallelism

A.4 MergeSort Parallelism Cut-Off Testing Results

MergeSort (Min Size)

Input Size = 8388608 — **Raw Run Results**

Sequential

Parallelism	Recursion Cut-off	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)
1	8	2617	2533	2475	2573	2539	2485	2523	2431	2488	2407	2507

ForkJoin

Parallelism	Parallelism Cut-off	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup
2048	4096	1018	1010	949	890	965	907	914	839	589	904	899	2.79
2048	8192	884	892	998	874	777	691	866	654	698	506	784	3.20
2048	16384	800	691	900	997	809	787	492	637	827	911	785	3.19
2048	32768	1321	971	887	726	661	907	830	914	812	967	900	2.79
2048	65536	805	893	916	1302	913	957	1020	776	826	911	932	2.69
2048	131072	898	1071	991	915	960	1153	558	894	992	1120	955	2.62
2048	262144	800	902	766	1019	903	929	812	957	1035	887	901	2.78

Threaded

Parallelism	Parallelism Cut-off	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup
2048	4096	3089	2768	2409	2673	2666	2701	2833	2878	3122	2920	2806	0.89
2048	8192	2004	1884	1976	2224	1914	2050	2001	1958	1786	1912	1971	1.27
2048	16384	1522	1026	1166	1345	1238	1251	1047	1063	994	846	1150	2.18
2048	32768	1283	1206	952	1139	1172	934	779	976	1328	945	1071	2.34
2048	65536	968	856	929	764	709	715	940	769	842	924	842	2.98
2048	131072	862	1057	912	1103	813	784	929	1025	782	915	918	2.73
2048	262144	1329	1004	971	1030	925	1391	919	1319	821	915	1062	2.36

Figure A.7: Raw results of MergeSort testing when varying parallelism cut-off

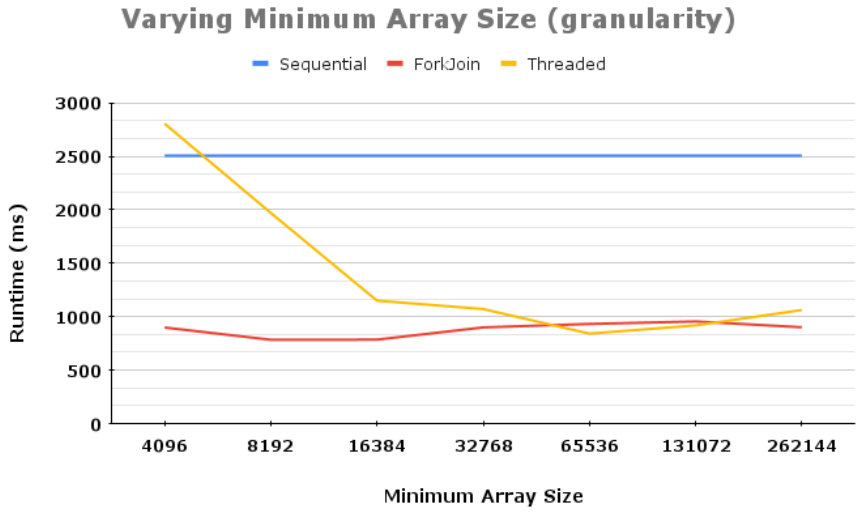

Figure A.8: Comparing runtime of MergeSort strategies when varying parallelism cut-off

A.5 FFT Parallelism Testing Results

FFT (Parallelism)

Input Size = 8388608 — **Raw Run Results**

Sequential

Recursion Cut-off	Parallelism	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)
8	1	5173	4937	4991	4676	5059	4812	4883	4637	4760	4954	4888

ForkJoin

Parallelism Cut-off	Parallelism	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup	Efficiency
8388608	1	4649	4983	4780	4987	4724	4825	4635	4914	4551	4753	4780	1.02	1.02
4194304	2	3085	3005	3046	2830	2953	3039	2916	2862	3098	3114	2995	1.63	0.82
2097152	4	2138	2145	2351	2423	2083	1937	2309	1991	2221	2038	2164	2.26	0.57
1048576	8	2026	2064	1796	1761	1990	1731	1722	1752	1887	1999	1873	2.61	0.33
524288	16	2066	2064	2111	2069	2038	2048	2039	2032	2117	2046	2063	2.37	0.15
262144	32	2091	2251	2380	2313	2191	2099	2263	2135	2071	2096	2189	2.23	0.07
131072	64	2219	2217	2218	2400	2238	2197	2345	2274	2222	2101	2243	2.18	0.03

Threaded

Parallelism Cut-off	Parallelism	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup	Efficiency
8388608	1	4981	5106	5058	5221	4982	4973	5064	4959	4978	5037	5036	0.97	0.97
4194304	2	3041	2951	3201	3064	2953	2973	3014	3155	2813	2845	3001	1.63	0.82
2097152	4	2244	2172	2453	2072	2042	1989	2469	1959	2018	2386	2180	2.24	0.56
1048576	8	1957	1970	1866	2018	1864	2013	2032	2083	2126	2485	2041	2.39	0.30
524288	16	2204	2149	2097	2071	2121	2203	2194	2104	2216	2151	2151	2.27	0.14
262144	32	2282	2398	2315	2329	2376	2242	2244	2235	2302	2335	2306	2.12	0.07
131072	64	2376	2239	2321	2208	2188	2519	2426	2299	2302	2207	2309	2.12	0.03

Figure A.9: Raw results of FFT testing when varying parallelism

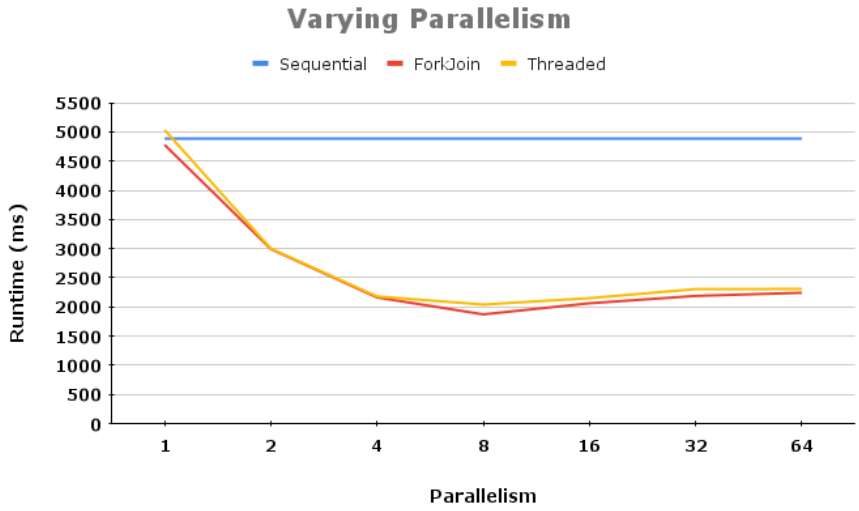

Figure A.10: Comparing runtime of FFT strategies when varying parallelism

A.6 FFT Parallelism Cut-Off Testing Results

FFT (Min Size)

Input Size = 8388608						Raw Run Results									
						Sequential									
Parallelism	Recursion Cut-off	Run Number										Average (ms)			
		1	2	3	4	5	6	7	8	9	10				
1	8	5292	5027	5006	4651	4981	4946	4889	4551	4794	4655	4879			
						ForkJoin									
Parallelism	Parallelism Cut-off	Run Number										Average (ms)	Speedup		
		1	2	3	4	5	6	7	8	9	10				
2048	4096	2737	2621	2570	2596	2808	2938	2912	3017	3053	2837	2809	1.74		
2048	8192	2665	2650	2604	2709	2654	2765	2686	2779	2801	2804	2712	1.80		
2048	16384	2692	3295	2878	3097	2784	2791	2761	2767	2802	2757	2862	1.70		
2048	32768	3003	3199	2795	3326	2915	2889	2739	2676	2755	2814	2911	1.68		
2048	65536	2753	2736	2836	2890	2922	2842	2887	2852	2805	3109	2863	1.70		
2048	131072	2680	3340	2857	3198	2929	2937	3333	3119	3115	2802	3031	1.61		
2048	262144	2890	2907	3408	3133	2823	2767	2677	2924	2930	3537	3000	1.63		
						Threaded									
Parallelism	Parallelism Cut-off	Run Number										Average (ms)	Speedup		
		1	2	3	4	5	6	7	8	9	10				
2048	4096	5768	5630	6039	5736	6027	5864	5843	5990	6678	5956	5953	0.82		
2048	8192	4729	4708	4581	4537	4588	4557	4457	4397	4568	4388	4551	1.07		
2048	16384	3784	3923	3700	3816	3770	3803	3781	3761	3883	3889	3811	1.28		
2048	32768	3570	3283	3387	3368	3659	3605	3362	3587	3504	3583	3491	1.40		
2048	65536	3075	3441	3395	3198	3449	3270	3175	3365	3391	3040	3280	1.49		
2048	131072	3546	3208	3112	3458	3031	2930	3286	3057	3427	3035	3209	1.52		
2048	262144	2911	3126	3010	2851	2792	2854	2808	2844	2847	2817	2886	1.69		

Figure A.11: Raw results of FFT testing when varying parallelism cut-off

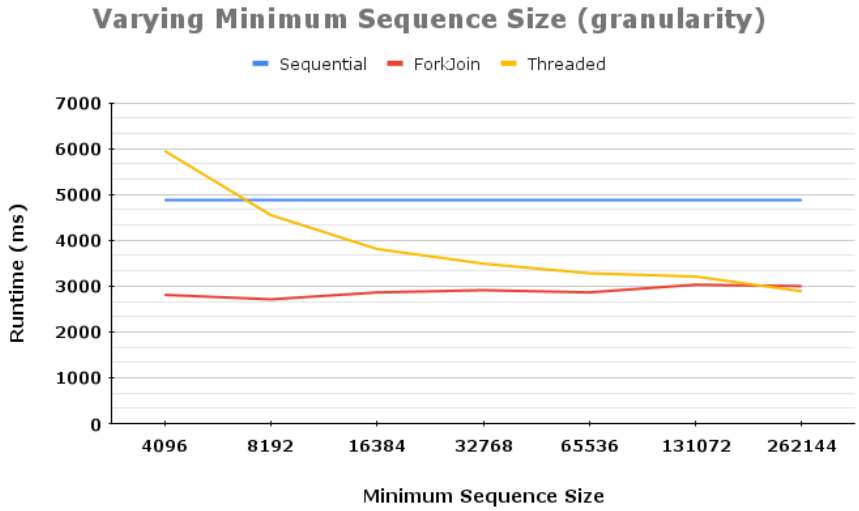

Figure A.12: Comparing runtime of FFT strategies when varying parallelism cut-off

A.7 Strassen Parallelism Testing Results

Strassens (Parallelism)

Input Size = 2048x2048 — **Raw Run Results**

Sequential

Recursion Cut-off	Parallelism	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)
32	1	4834	4576	4576	4578	4576	4642	4622	4583	4654	4641	4628

ForkJoin

Parallelism Cut-off	Parallelism	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup	Efficiency
2048	1	4650	4593	4637	4794	4687	4617	4595	4605	4619	4616	4641	1.00	1.00
1024	2	3360	2705	3335	2681	3332	2714	3379	3357	3342	3320	3153	1.47	0.74
512	4	1360	1302	1316	1331	1362	1308	1489	1346	1355	1493	1366	3.39	0.85
256	8	886	855	850	844	894	819	842	829	843	855	852	5.43	0.68
128	16	670	677	657	682	695	697	685	681	636	641	672	6.89	0.43
64	32	660	656	694	672	670	643	646	622	636	684	658	7.03	0.22
32	64	1537	577	593	592	607	621	571	574	571	551	679	6.81	0.11

Threaded

Parallelism Cut-off	Parallelism	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup	Efficiency
2048	1	4607	4592	4539	4605	4555	4514	4630	4668	4591	4584	4589	1.01	1.01
1024	2	2647	2654	3265	2653	2648	2664	3286	2640	3298	2654	2841	1.63	0.82
512	4	1436	1357	1367	1396	1387	2014	1992	1384	1371	2028	1573	2.94	0.74
256	8	904	931	911	878	883	854	883	910	908	916	898	5.16	0.65
128	16	985	958	891	966	899	862	824	838	858	862	894	5.18	0.32
64	32	890	916	892	967	936	927	947	890	879	868	911	5.08	0.16
32	64	637	611	650	640	654	624	643	673	665	661	646	7.17	0.11

Figure A.13: Raw results of Strassen testing when varying parallelism

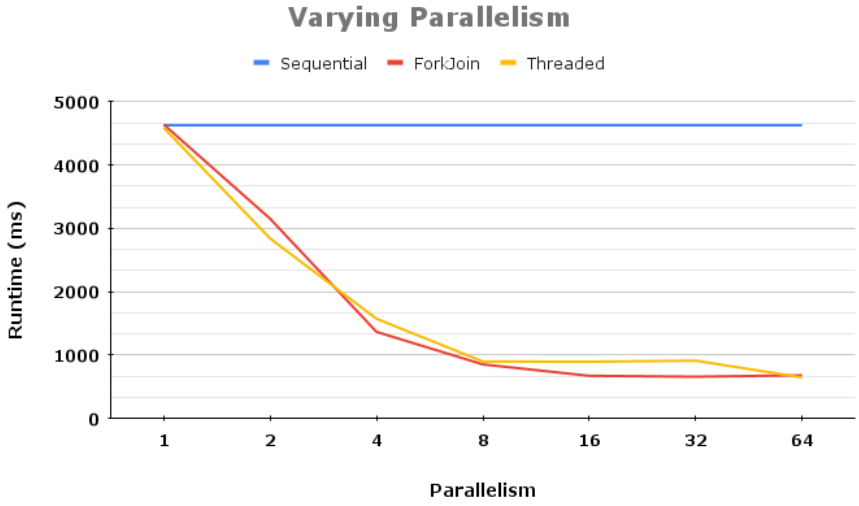

Figure A.14: Comparing runtime of Strassen strategies when varying parallelism

A.8 Strassen Parallelism Cut-Off Testing Results

Strassens (Min Size)

Input Size = 2048x2048 — **Raw Run Results**

Sequential

Parallelism	Recursion Cut-off	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)
1	16	6070	5812	5817	5783	5804	5780	5776	5844	5787	5754	5823

ForkJoin

Parallelism	Parallelism Cut-off	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup
128	16	992	814	799	835	943	819	838	816	838	811	851	6.85
128	32	795	771	798	793	807	815	787	771	793	824	795	7.32
128	64	747	727	782	772	758	811	819	774	774	739	770	7.56
128	128	774	728	736	774	724	768	768	773	773	786	760	7.66
128	256	813	760	752	822	760	755	744	757	795	745	770	7.56
128	512	751	757	760	782	794	766	769	747	769	782	768	7.58
128	1024	1144	1042	1034	1062	1075	1096	1075	1030	1094	1108	1076	5.41

Threaded

Parallelism	Parallelism Cut-off	Run Number 1	2	3	4	5	6	7	8	9	10	Average (ms)	Speedup
128	16	804	792	777	793	831	793	811	788	774	829	799	7.29
128	32	738	731	797	762	808	761	770	787	766	774	769	7.57
128	64	773	816	771	781	754	772	762	815	768	776	779	7.48
128	128	787	779	786	775	747	818	800	748	771	765	778	7.49
128	256	812	767	763	804	758	772	825	793	790	810	789	7.38
128	512	743	745	780	775	812	766	748	734	743	808	765	7.61
128	1024	1031	1092	1098	1077	1024	1064	1091	1052	1007	1036	1057	5.51

Figure A.15: Raw results of Strassen testing when varying parallelism cut-off

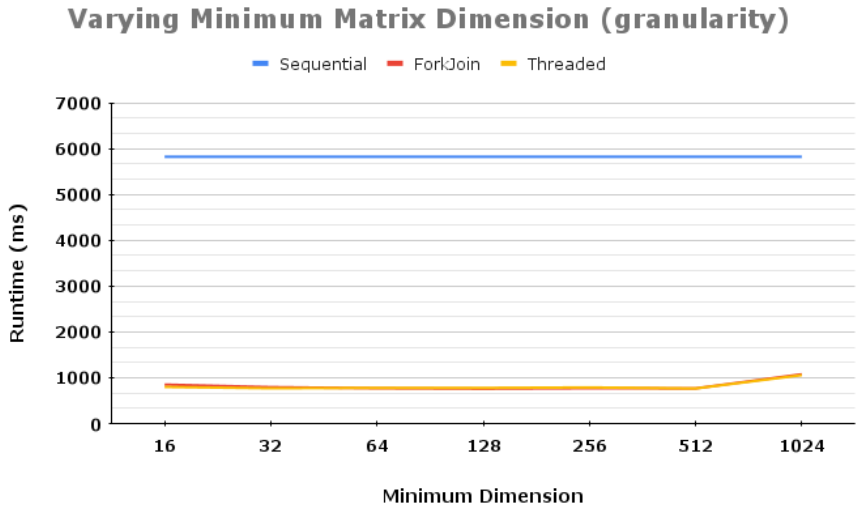

Figure A.16: Comparing runtime of Strassen strategies when varying parallelism cut-off

A.9 Skeleton MergeSort Parallelism Testing Results

Skeleton | MergeSort (Parallelism)

Input Size = 8388608		Raw Run Results											
Sequential Recursion Cut-off													
8													
MergeSort Skeleton													
Parallelism Cut-off	Parallelism	Run Number										Average (ms)	Speedup
		1	2	3	4	5	6	7	8	9	10		
8388608	1	2490	2215	2430	2384	2400	2253	2391	2268	2288	2376	2350	1.08
4194304	2	1445	1419	1395	1329	1530	1512	1313	1430	1556	1345	1427	1.78
2097152	4	976	1020	931	872	949	951	1065	1012	1024	954	975	2.60
1048576	8	841	745	893	790	837	950	750	952	824	728	831	3.05
524288	16	929	876	742	826	734	919	968	746	943	824	851	2.98
262144	32	768	947	888	846	889	804	759	725	829	759	821	3.09
131072	64	814	983	826	898	788	870	885	912	875	848	870	2.92

Figure A.17: Raw results of Skeleton MergeSort strategy testing when varying parallelism

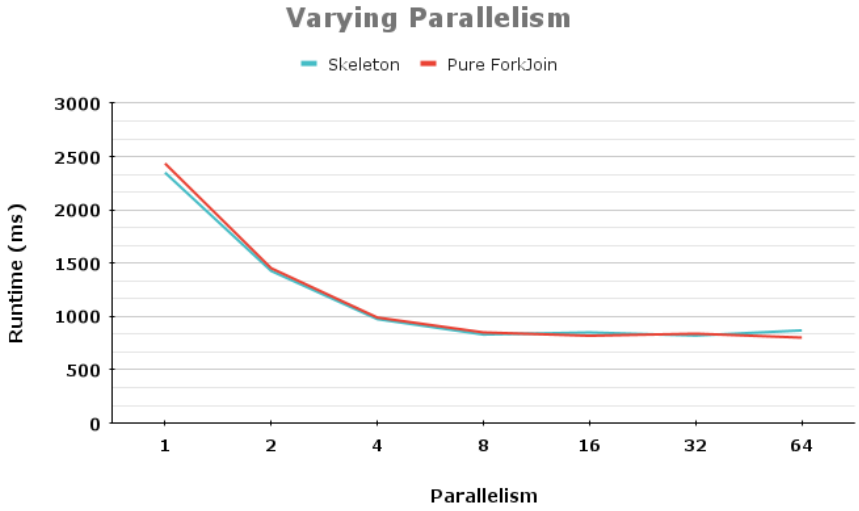

Figure A.18: Comparing runtime of Skeleton and ForkJoin MergeSort strategies when varying parallelism

A.10 Skeleton MergeSort Parallelism Cut-Off Testing Results

Skeleton | MergeSort (Min Size)

Input Size = 8388608 — Raw Run Results

Sequential Recursion Cut-off
8

MergeSort Skeleton

Parallelism	Parallelism Cut-off	Run Number										Average (ms)	Speedup
		1	2	3	4	5	6	7	8	9	10		
2048	4096	1157	1292	1173	857	1092	997	1012	1005	1107	1034	1073	2.34
2048	8192	981	859	1197	1058	973	975	927	1058	1000	1011	1004	2.50
2048	16384	1032	982	975	992	870	974	953	931	1010	950	967	2.59
2048	32768	969	791	920	926	933	956	936	942	1150	959	948	2.64
2048	65536	855	936	821	1057	885	825	917	927	841	914	898	2.79
2048	131072	921	897	877	910	829	1014	923	907	861	846	899	2.79
2048	262144	935	821	857	949	909	844	760	772	888	837	857	2.92

Figure A.19: Raw results of Skeleton MergeSort strategy testing when varying parallelism cut-off

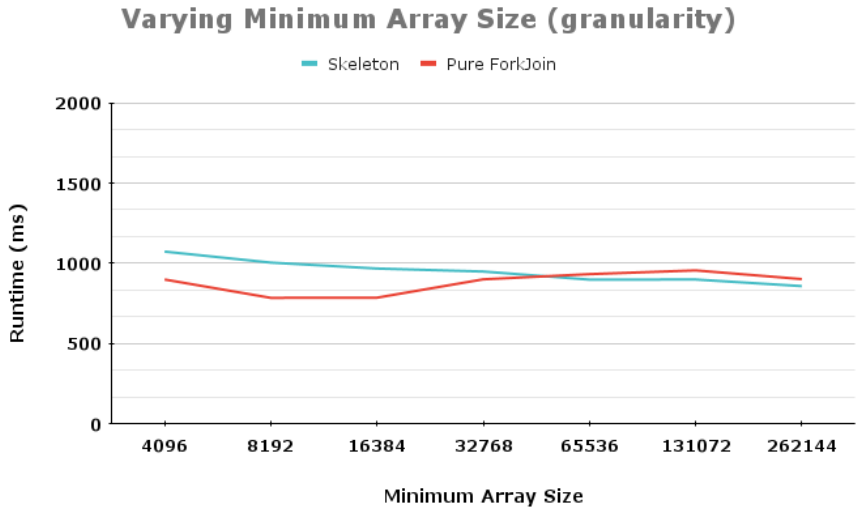

Figure A.20: Comparing runtime of Skeleton and ForkJoin MergeSort strategies when varying parallelism cut-off

A.11 Skeleton Strassen Parallelism Testing Results

Skeleton | Strassens (Parallelism)

Input Size = 2048x2048 — Raw Run Results

Sequential Recursion Cut-off
32

Strassens Skeleton

Parallelism Cut-off	Parallelism	Run Number										Average (ms)	Speedup
		1	2	3	4	5	6	7	8	9	10		
2048	1	4840	4587	4562	4560	4627	4633	4596	4605	4590	4583	4618	1.00
1024	2	2659	2672	2654	2653	2658	2651	2659	2664	2652	2683	2661	1.74
512	4	1278	1258	1261	1285	1306	1274	1307	1276	1279	1289	1281	3.61
256	8	789	798	859	841	777	764	755	812	798	751	794	5.83
128	16	613	629	637	585	611	644	676	601	617	615	623	7.43
64	32	622	638	651	642	616	633	627	616	621	632	630	7.35
32	64	745	582	611	600	575	578	619	603	618	614	615	7.53

Figure A.21: Raw results of Skeleton Strassen strategy testing when varying parallelism

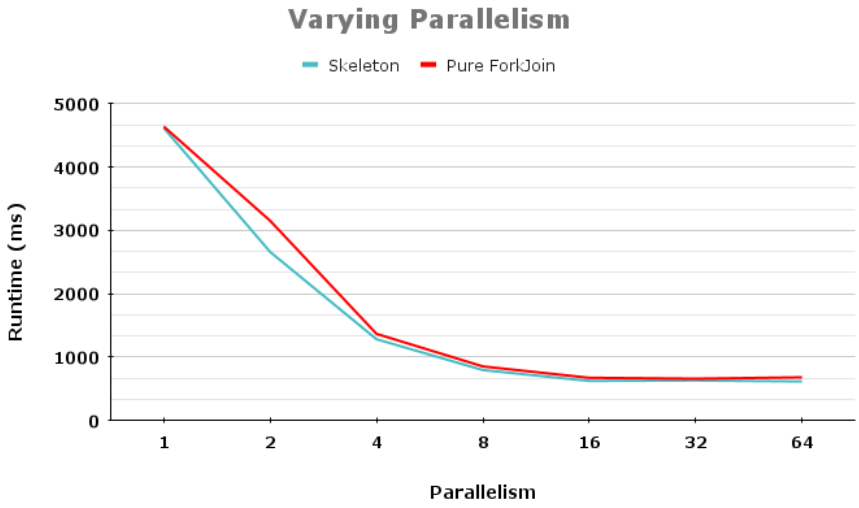

Figure A.22: Comparing runtime of Skeleton and ForkJoin Strassen strategies when varying parallelism

A.12 Skeleton Strassen Parallelism Cut-Off Testing Results

Skeleton | Strassens (Min Size)

Input Size = 2048x2048 — Raw Run Results

Sequential Recursion Cut-off
16

Strassens Skeleton

Parallelism	Parallelism Cut-off	Run Number										Average (ms)	Speedup
		1	2	3	4	5	6	7	8	9	10		
128	16	1589	808	814	794	788	803	804	821	794	769	878	6.63
128	32	878	824	809	804	811	827	818	819	811	817	822	7.09
128	64	798	781	778	804	803	771	785	783	798	806	791	7.36
128	128	768	766	777	779	776	770	770	763	776	774	772	7.54
128	256	774	760	769	766	775	766	763	778	775	770	770	7.57
128	512	768	762	740	749	754	743	745	766	745	748	752	7.74
128	1024	1004	998	1023	1026	1022	977	1015	1061	971	994	1009	5.77

Figure A.23: Raw results of Skeleton Strassen strategy testing when varying parallelism cut-off

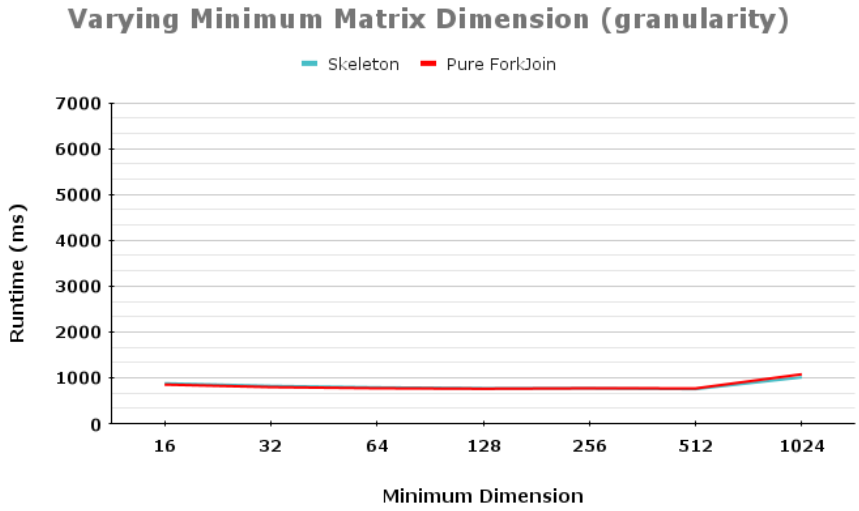

Figure A.24: Comparing runtime of Skeleton and ForkJoin Strassen strategies when varying parallelism cut-off