Evaluating Java Parallel Libraries on Divide and Conquer Algorithms

Chenzhi Wu

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

This project investigated the performance of five different Java libraries (Thread, FixedThreadPool, CachedThreadPool, ForkJoin and Skandium) for implementing D&C algorithms as well as their ease of programming. Focusing on six different problems, We implemented the sequential version and five different parallel versions of solutions with the selected libraries for each of them. We experiment with the implementations by tuning the granularity (the size of no-further-split cases) and collect data of performance and ease of programming. A comparative analysis of the libraries has been conducted based upon the experiment results. With the aid of quantitative metrics and additional comments, a thorough evaluation of the five Java parallel libraries from aspects of performance and ease of programming has been given on D&C algorithms. We have concluded that there is a trade-off between performance and ease of programming. As a result, in future work, we propose to develop skeletons which adjust the scheduling of underlying threads automatically so that they can not only hide the complexity of parallelism from users but also achieve the optimal performance.

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.

(*Chenzhi Wu*)

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Murray Cole. Thanks for offering me such a valuable opportunity to study with him. I am very grateful to him for showing me the way when I was lost in my project, and for his patience and attentiveness when I was in trouble. I am grateful to him for always being able to take care of my emotions in a thoughtful way. Thanks for all the help and encouragement during my project.

I would then like to thank my mother and my aunt, for taking good care of my father and my grandmother so that I can focus on my project. Thanks for always offering me support and being my backup.

I would also like to thank my girlfriend, who always keeps me accompany and encourages me when I was upset.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Divide and Conquer, hence denoted D&C, is a classic algorithmic paradigm, which divides a problem into sub-problems of smaller sizes recursively until finding base cases, applying the same strategy to solve the sub-problems and combine their results to obtain the final solution. It has been found an efficient problem-solving technique in a variety of areas. For example, it is the basis of efficient sorting and searching algorithms, which are the most commonly needed algorithms in programming [1]. Nevertheless, with the size and complexity of problems growing exponentially, the need for faster and more scalable algorithms becomes imperative.

One of the most efficient ways is to parallelize them. In recent decades, the development of multi-core systems has led to the evolution of computing power, where each core can work and process independently. Parallelism is a mechanism that splits large computation into smaller partitions and process them simultaneously. Due to the independence between sub-problems in D&C, it is natural to exploit parallelism for D&C algorithms so that one can make the most of computational power. A great number of research has been conducted on parallelized D&C algorithms, some of which have been proved to decrease time consumption sharply comparing to their sequential version [2].

However, it is more complex to implement parallel algorithms with regard to their sequential versions. Specifically, besides implementing solution strategies for sub-problems, programmers also need to create worker threads for tasks, execute them correctly to avoid errors or deadlocks, and then to achieve a good balance of work to the available cores. To help programmers take the benefits of parallelism in increasing

throughput and improving performance in software development, Java, as one of the most popular programming languages, has provided pre-built libraries. They have different abstraction levels and thus aid parallel programming in different ways[3, 4].

Considering the wide use of D&C algorithms, parallelizing them will be of great significance, especially for practical applications with high throughput. We hope that this project will provide a reference for the development of parallel D&C algorithms and help programmers make better use of these libraries to fully utilize the computational resources to optimize the performance of their programs.

1.2 Problem Statement

When implementing parallel D&C algorithms in Java, there are multiple parallel libraries available for use. They have different abstraction levels, i.e., they vary in functions and interfaces they provide, with which programmers can either create and manipulate threads directly or manage threads with thread pool automatically. There are also programming abstractions that hide the complexity of parallelism from users. In addition, there are algorithmic skeletons that completely hide underlying parallelism from users. For example, when programming with algorithmic skeletons, such as Skandium [5, 6, 7], programmers only need to specify sequential code blocks for splitting, merging and solving problems. These blocks will be invoked and parallelized automatically. As a result, a lot of code can be omitted. Nevertheless, in contrast to direct threading libraries, it also leads to less control on low-level parallelism execution in the mean time.

Furthermore, these libraries are also distinguished by the work scheduling strategies they adopt. Scheduling tasks in Java consist of two parts, which are work scheduling (schedule tasks to virtual threads) and thread scheduling (schedule threads to be executed by physical CPU cores). While thread scheduling strategies are determined by JVM, work scheduling strategies can vary across libraries. For example, if a First-In-First-Out pattern is applied to schedule waiting tasks, because of dependency between tasks in D&C algorithms, deadlocks will occur. In this case, more virtual threads will be required, which increases the cost for thread scheduling and may potentially consume more time. Moreover, work scheduling strategies can affect the ease of tuning granularity (the size of non-split tasks during execution). Specifically, with the decrease of granularity, more tasks will be assigned to threads, which in the mean time increase the difficulty of balancing workload for worker threads. As argued above, a balanced workload benefits

the performance a lot. To achieve the optimal performance, a lot of experiments will be required for tuning the granularity.

To sum up, different libraries differ in the aspects of interfaces provided and work scheduling strategies adopted, which can affect both the performance and ease of programming. Therefore, it is important to evaluate how they aid the implementation of parallel D&C algorithms in software development.

1.3 Aims and Objectives

This project aims to measure Java parallel libraries' ability to aid the utilization of computation resources (which is reflected by the performance) and the ease of programming for D&C algorithms. Our research is based upon the hypothesis that Java parallelism libraries with the different levels of abstraction provide a useful range of trade-offs between performance and programmability, i.e., more abstract libraries may have a compromise on performance, which make them suitable for different applications.

The objectives of this project are:

- Find a collection of specific problems that can be solved by D&C algorithms. Make sure these problems have different depths of recursion, degrees of balance and quantity of computation in base cases.
- For each problem, implement its sequential solution as the baseline. After that, implement different versions of its parallel solution with each of the libraries we aim to evaluate.
- Experiment with the solutions by tuning the granularity that can optimize the performance and collect the results.
- Analyse their performance and evaluate the ease of programming.
- Discuss how features of the libraries aid the implementations of D&C algorithms, which can be indicative for future work.

1.4 Achieved Result

In this project, we have implemented six versions of D&C solutions (including one sequential version and five parallel versions with five different Java parallel libraries)

to six classic problems. We successfully collected data of different versions about performance by tuning the granularity (size of not further split cases) and ease of programming. We have discussed these experiment results and given a thorough analysis of the selected Java libraries from aspects of performance and ease of programming with the aid of quantitative metrics and comments. It has been concluded that their could be a trade-off between performance and ease of programming.

1.5 Dissertation Outline

This dissertation will be divided into six chapters. Chapter 2 will present the related work about parallel libraries. Programming abstractions and algorithmic skeletons designed for D&C algorithms, such as TBB, ForkJoin and Skandium will be introduced as they are valuable for further evaluation. Methodology for experiments and evaluation in the project will be described in Chapter 3, including the metrics for analysing the performance and ease of programming. The results of experiments will be exhibited in Chapter 4. Chapter 5 will discuss and analyse the results. Finally, Chapter 6 will conclude the project and discuss future work for implementing D&C algorithms more efficiently.

Chapter 2

Related Work

This chapter will discuss the related work to our project. To be specific, we will introduce the Java parallel libraries that we will evaluate in our project and thread scheduling strategies in Java that will aid us to discuss experiment results. Moreover, we will also mention parallel libraries designed for D&C algorithms in other languages.

2.1 Java parallel libraries

2.1.1 Thread

When Java was first introduced in mid-1990s, one of its standout features was built-in support for multi-threaded programming, core of which was the "**Thread**" class. Java allows programmers to create multiple threads in a single program. To parallelize tasks, a new thread can be created by instantiating a new "Thread" object and overriding its "**run()**" method, which is the code block to be executed.

The life-cycle of a "**Thread**" includes several states: **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING** and **TERMINATED**. "**Thread**" class provides methods to manage and monitor these states, such as "**start()**", "**sleep()**" and "**join()**". A thread can be started from external by calling its **start()** method, which will invoke its **run()** function, while **join()** is for waiting for its termination.

As the most basic execution unit of java parallel computing. "**Thread**" provides control over underlying parallelism. Specifically, each instance of **Thread** has a assigned priority which helps the scheduler to decide the order of thread execution. By default, a thread's priority equals the priority of the thread from which it is created. Programmers can also query and change the priority of a thread through "**getPriority()**"

and "**setPriority**()" methods [8].

However, "**Thread**" has several limitations. Firstly, the direct manipulation **Thread** objects can be complex, especially when managing a large number of threads, including the spawning, execution and killing of each thread. Secondly, writing and handling multi-threading can be complex and error-prone. Improper coordinating can cause issues such as deadlocks, race conditions and memory inconsistencies.

2.1.2 ThreadPoolExecutor

Besides single threads, **Thread pool** is also an important concurrency mechanism, introduced as a part of "**java.util.concurrent**" package in Java 5. **Thread pool** is a pool of worker threads, enabling programmers to manage them as a whole. [3]

"**java.util.concurrent**" package also offers higher-level concurrency tools, one of which is "**ThreadPoolExecutor**" class. "**ThreadPoolExecutor**" is a threading framework that helps reduce the complications and inefficiencies of manual thread managements. By facilitating thread pooling and task scheduling, it allows for better resource utilization and simplifies concurrent programming, which is a compensation for drawbacks of "**Thread**" class. Specifically, the key difference between direct threading and thread pooling is that there is no need to care about task scheduling. Programmers can focus more on tasks that need to be executed, and thread pools will schedule them automatically. Moreover, thread pools also control the lifecycle of threads inside them, which help release the computation resources in a timely manner. Methods such as "**getPoolSize**()", "**getActiveCount**()" and "**getCompletedTaskCount**()" have provided insights into the thread pool's status.

To create a thread pool, programmers can use the constructor of **ThreadPoolExecutor**. There are various configurable parameters controlling behaviors of the thread pool:

- Core pool size: Specifies the minimum number of threads in the pool. The pool will keep threads alive no fewer than this value.
- Maximum pool size: Specifies the maximum number of alive threads in the pool.
- Keep-live time: Controls how long an idle thread can keep alive in the pool. **ThreadPoolExecutor** will kill threads that have been idle for longer time than this value to save resources.
- Work queue: Specifies the data structure containing tasks waiting to be executed.

- Thread factory function: Specifies the function for creating new threads.

Moreover, the "**Executors**" class provide interfaces for creating pre-configured **ThreadPoolExecutor**s. Programmers can use the factory functions in this class directly without specifying the parameters in **ThreadPoolExecutor**, such as fixed thread pool and cached thread pool [3, 9].

2.1.3 Fork and Join

Fork/Join is another Java's parallel library, introduced in Java 7. It is a threading framework designed with the idea of divide and conquer, by decomposing large tasks to achieve fine-grained parallelism. As the name suggests, **fork()** splits a large task into smaller ones, while **join()** gets the result of the execution of the smaller ones.

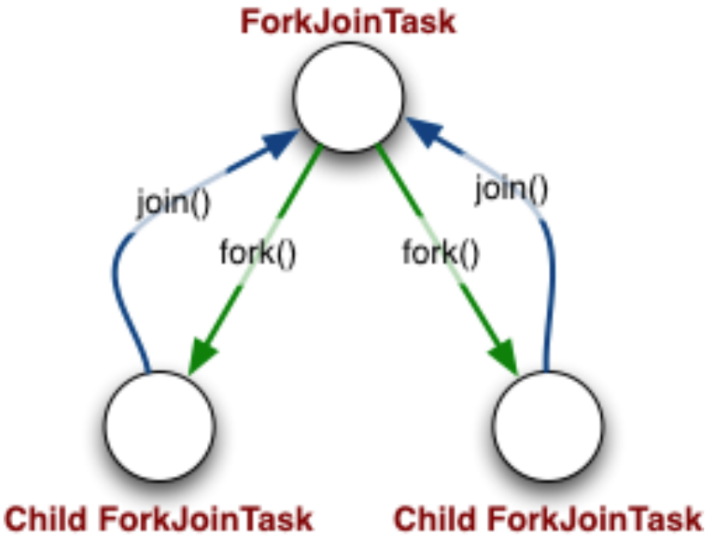

Figure 2.1: ForkJoin Framework

There are two primary components in the framework, which are **ForkJoinPool** and **ForkJoinTask**. **ForkJoinPool** is an implementation of executor service (same as ThreadPoolExecutor) and designed to work with ForkJoinTasks, including "**RecursiveTask**" (tasks that return a result) and "**RecursiveAction**" (tasks that do not return a result). Each task is represented with a corresponding object and gets executed automatically by ForkJoinPool after started.

As a framework designed especially for fine-grained parallelism, Fork/Join framework handles a vast number of small tasks particularly efficiently, which fits right in with the characteristics of the D&C algorithm. This is because ForkJoinPool employs

a work-stealing algorithm where idle threads ”steal” tasks queued up in other busy threads, ensuring balanced workload and reduced thread contention [10, 11]. Another strength of this framework is that it scales naturally with the number of available cores, thus optimizing resource optimization [4]. However, the usage of this framework is relatively limited since D&C might not be intuitive for all algorithms.

2.1.4 Algorithmic skeletons and Skandium

Algorithmic skeletons were first introduced by Cole in [5] in 1991. Originating from the recognition that many parallel algorithms share common structures, skeletons are designed as reusable patterns of parallel computation and communication. These skeletons contain sequential code blocks, which are responsible for actual computations, and encapsulate parallelism details. All programmers need to do is to implement the code blocks and combine the skeletons, and they will be invoked and parallelized automatically. To sum up, algorithmic skeletons abstract away the complexity of low-level parallelism and offer a high-level view. Moreover, they can be combined to implement different algorithms.

Several skeleton frameworks targeting at D&C algorithms have also been produced. A typical example is Skandium, which was firstly introduced in [7]. Skandium firstly defines four ”muscle” functions, including *condition*, *split*, *execution* and *merge*, which are sequential code blocks programmers need to implement. These ”muscle” functions can be combined to constitute different skeletons. For instance, in the structure of a ”DaC” skeleton, (1) *condition* specifies whether to split the problem, (2) *split* is invoked to split the problem, (3) *execution* solves the base cases, (4) and *merge* merges results from sub-tasks. Secondly, Skandium designs and implements nine skeletons. Different skeletons consist of different ”muscle” functions. They can be also combined following a pipeline model to solve various problems, as guided in [7].

$$\begin{aligned}\triangle ::= \ & \text{seq}(f_e) \mid \text{farm}(\triangle) \mid \text{pipe}(\triangle_1, \triangle_2) \mid \text{while}(f_c, \triangle) \mid \\ & \text{if}(f_c, \triangle_{true}, \triangle_{false}) \mid \text{for}(i, \triangle) \mid \text{map}(f_s, \triangle, f_c) \mid \\ & \text{fork}(f_s, \{\triangle_i\}, f_m) \mid \text{d\&c}(f_c, f_s, \triangle, f_m)\end{aligned}$$

Figure 2.2: Skandium Skeletons

2.2 Thread scheduling strategies

Thread scheduling strategy is the fundamental concept that determines the order in which threads get executed. Strictly speaking, Java has no mechanism for scheduling threads, because Java threads are backed by native threads, which are scheduled by the operating system. In general, the thread scheduling strategy adopted by Java multi-thread programs is **Time-slicing (Round-Robin) Preemptive, priority-based scheduling**. (However, the exact scheduling behavior may depend on the implementation of Java virtual machine and underlying operating system.)

As mentioned above, each thread in Java is assigned a priority. The OS will choose one thread at one time and give a fixed slice of time to execute based on their priorities. A thread with higher priority will get more time slices to executed while lower-priority threads get fewer. ”Preemptive” means when a thread with a higher priority becomes runnable, a lower-priority thread will be preempted, i.e., interrupted. It is notable that this strategy may lead to starvation for threads with lower priority.

2.3 Libraries in other Languages

Intel developed TBB (Threading Building Blocks), which is a C++ library and provides parallel patterns such as for, *reduce*, *scan* and *pipeline*. Although it is described as a task-based threading library, TBB actually provides a low-level abstraction. Specifically, TBB hides all functions to parallelize from users. When programming with TBB, all users need to do is to use the template and specify the input and the splitting threshold, i.e., the granularity. In the meantime, it enables users to have more control on parallelism, such as combining other parallel libraries or accessing the scheduler [12].

In 2021, Millan et al. proposed a C++ skeleton library which is designed for D&C algorithms. It is worthy to mention that this project puts more focus on those problems with large degree of imbalance and depth of recursion. As a result, instead of basing the skeletons on TBB scheduler, it implements its own scheduler with the strategy ”work stealing”. Moreover, to deal with problems which requires deep recursion, it implements a new data structure to contain the tasks, basing on heap instead of stack, to prevent they from crashing down. As shown by its evaluation, it achieves almost linear speedup on the benchmark $T3XXL$, which is a benchmark to traverse a deep and highly unbalanced binomial tree, where the programming abstractions provided by TBB fail with stack overflow [6, 13].

Chapter 3

Methodology

We aim to evaluate and analyse Java's parallel libraries' performance on Divide and Conquer algorithms as well as their ease of programming. This will be based on experiments with implementations of algorithms with different libraries. This chapter will introduce specific algorithms used to evaluate the libraries and how we implement the solutions with libraries we aim to evaluate in the project. Besides, metrics to evaluate performance and ease of programming will also be stated.

3.1 Algorithms

We focus on a set of classic problems which can be solved with Divide and Conquer algorithms. These problems vary from each other in the following aspects:

1. Total number of tasks.

2. Largest possible size of tasks. In D&C algorithms, the size of tasks mainly depend on the quantity of computation to split and merge, which changes with the recursion level in some cases, while not in others. This factor refers to the largest size among all tasks, which is commonly the size of the root task.

3. Degree of imbalance of splits. This measures the imbalance between the size of subtasks generated by the same tasks. The imbalance can accumulate when the recursion goes deeper, which can cause a great difference between the sizes of tasks on the same recursion level.

4. Difference between sizes of all tasks. Size of tasks may change with the recursion level, polynomially in some cases while geometricly in others, e.g., FFT.

These factors affect the scheduling of tasks and further affect the performance of programs. As a result, selecting enough problems varing in these features can guarantee a thorough analysis of performance of Java parallel libraries. Table 3.1 gives a summary of algorithms in our project.

Problem	**Number of tasks**	**Size of largest tasks**	**Degree of imbalance of splits**	**Difference in size of all tasks**
Fibonacci	2^n	Small	Low	Low
N-Queens	n^n	Medium	Low	Low
Quick Sort	Unpredictable	Large	Large	Large
0-1 Knapsack	2^n (n is number of items)	Small	Medium	Small
FFT	n	Large	Very Low	Large
Adaptive Quadrature	Unpredictable	Small	Medium	Very Low

Table 3.1: Summary of Problems

3.1.1 Fibonacci

Fibonacci problem is to calculate the n-th number in a Fibonacci sequence with divide and conquer. It follows the formula:

$$F(n) = F(n-1) + F(n-2),\ n \geq 2$$

which indicates that each task is split into two approximate subtasks and base cases are F(0) and F(1). This indicates that the time difference consumed by synchronization between subtasks. Moreover, the computation required to merge results is simple, which means the size of each task is small, leading to the flexibility of scheduling. However, the number of tasks in this problem explodes with the increase of n (almost 2^n tasks) [14].

3.1.2 N-Queens

N-Queens problem is to find all the solutions to a N-Queens problem, where n queens are placed on an $n \times n$ chessboard, ensured that they cannot attack each other. The algorithm fixes one queen and find the position of the next queen in residual board

Algorithm 1 Fibonacci Algorithm

```
function FIB(n)
    if n ≤ 1 then
        return n
    end if
    return fib(n-1)+fib(n-2)
end function
```

recursively. As a result, it searches through the state space of size n^n. Since each task is responsible for checking the validity of each position, both the number and size of tasks are large, which challenges the program [15].

Algorithm 2 N-Queens Algorithm

```
function SOLVENQUEENS(board, col, n, solutions)
    if col ≥ n then
        solutions.append(board)
    end if
    for row from 1 to n do
        if queen can be placed on (row, col) then
            board[row][col] = 1
            solveNQueens(board, col+, n, solutions)
            board[row][col] = 0
        end if
    end for
end function
```

3.1.3 QuickSort

QuickSort algorithm is to sort an array of length n by partitioning it into two parts recursively, smaller and larger than a pivot. Due to the randomness of the selection of pivots, the sizes of two parts of the partitioned array may vary a lot, which causes imbalance between subtasks. Meanwhile, partitioning an array with a pivot requires $O(2n)$ operations. In this case, improper scheduling of tasks can lead to a decrease in performance [16].

Algorithm 3 Quick Sort Algorithm

```
function QUICKSORT(array, left, right)
    if left < right then
        pivotIndex = partition(array, left, right)
        quickSort(array, left, pivotIndex-1)
        quickSort(array, pivotIndex+1, right)
    end if
end function
function PARTITION(array, left, right)
    pivot = array[right]
    i = left - 1
    for index from left to right do
        if array[index] ≥ pivot then
            i++
            swap array[i] and array[index]
        end if
    end for
    swap array[i+1] and array[right]
    return i+1
end function
```

3.1.4 0-1 Knapsack

In 0-1 Knapsack problem, given a knapsack with a capacity and a collection of items, each of which has corresponding value and weight, we need to find the maximum value that can be put in the knapsack without exceeding the capacity where each item can be used up to one time. Each problem is split into two cases, i.e., for each item, either including or excluding it. As including the current item will occupy part of capacity, imbalance is naturally caused between two parts of recursion. Moreover, the number of tasks can be up to 2^n (where n is number of items) [17].

3.1.4.1 Fast Fourier Transformation

Fast Fourier Transformation algorithm is to compute the discrete Fourier transform of a sequence of n signals, where each signal has a real and imaginary part. The

Algorithm 4 Knapsack Algorithm

```
function KNAPSACK(values, weights, capacity, index)
    if index < 0 or capacity = 0 then
        return 0
    end if
    if weights[index] > capacity then
        return knapsack(values, weights, capacity, index-1)
    else
        includeItem=values[index]+knapsack(values, weights, capacity-
weights[index], index-1)
        excludeItem=knapsack(values, weights, capacity, index-1)
        return Max(includeItem, excludeItem)
    end if
end function
```

transformation follows the formula:

$$X_k = \Sigma_{n=0}^{N-1} e^{-i2\pi kn/N}, k = 0, ..., N-1$$

As derived and proved in [**?**], the transformation for the sequence can be computed by splitting it into odd-indexed and even-indexed part recursively and merge their results by:

$$X_k = E_k + e^{-(2\pi i/N)k} O_k, k = 0, ..., N/2-1$$

$$X_{k+N/2} = E_k - e^{-(2\pi i/N)k} O_k, k = 0, ..., N/2-1$$

This algorithm is a typical example where subtasks are well-balanced because the sequences are always divided equally, making the size the tasks half as the recursion goes deeper. It is notable that considering both splitting and merging sequences require the traversal of them, sizes of tasks on different recursion levels can vary greatly, while tasks on the same recursion level have approximately same size. The total number of tasks is n [18, 19].

3.1.4.2 Adaptive Quadrature

Adaptive Quadrature algorithm is to calculate the area of the graph below a function. The algorithm splits the current graph into two trapezoids, left and right, calculate and sum their areas to approximate its area, repeat this process recursively, until the error is

Algorithm 5 FFT Algorithm

```
function FFT(x)
    if x.length = 1 then
        return sequence
    end if
    odd = sequence of odd-indexed elements in x
    even = sequence of even-indexed elements in x
    oddResult = fft(odd)
    evenResult = fft(even)
    result = merge oddResult and evenResult with above formula
    return result
end function
```

less than a threshold. Because of the irregularity of the graph, it is unpredictable when the program terminates, leading to unpredictable number of tasks. However, the size of tasks is relatively small in this problem, each containing a few addition operations [20].

Algorithm 6 AQ Algorithm

```
Epsilon = 0.001
function AQ(func, left, right, lrarea)
    mid = (left + right)/2
    fleft = func(left)
    fright = func(right)
    fmid = func(mid)
    larea = (fleft + fmid) × (mid - left) / 2
    rarea = (fright + fmid) × (right - mid) / 2
    if Abs(larea + rarea - lrarea) > Epsilon then
        larea = AQ(func, left, mid, larea)
        rarea = AQ(func, mid, right, rarea)
    end if
    return larea+rarea
end function
```

3.2 Implementations

This section will introduce the implementations of above algorithms in our project, with libraries of abstraction level from lower to higher. For each of the above algorithms, all following versions are implemented for further experiments.

1. Sequential: A sequential version is implemented as a baseline. Moreover, the sequential solution is responsible for solving base cases in parallel versions, to make further analysis fairer.

2. Threading: The lowest abstraction level, where we are required to thread tasks directly. In this version, every task is represented with an object extending **Thread** class. Each creation of a task is also a creation of a thread. Similarly, starting and ending a task are direct operations on a thread. Because the result is encapsulated in the Thread object, extra code is required to fetch results from tasks. Because of direct threading, attention should be paid to synchronization among threads to make sure the right results have been prepared. Moreover, we have to handle exception in the execution of threads, such InterruptedException.

3. Thread pools: Instead of directly threading, thread pools are created to manage the threads. We implement the solutions with the aid of pre-configured thread pools provided in **Executors** class [9]. Specifically, we experiment with **FixedThreadPool** and **CachedThreadPool**, which apply different configurations and thus cause different resource consumption and ease of programming. In thread pools, we submit tasks as lambda functions, which return **CompletableFuture**s. We use interfaces provided by **CompletableFuture** class to achieve synchronization between tasks and fetch results [21, 22]. However, the two types of thread pools differ in the following ways:

   (a) FixedThreadPool: We create fixed thread pools to manage the threads, where the number of active threads is always fixed. It is notable that if all tasks are executed in the same fixed thread pool, because of dependency between tasks, a deadlock can be caused (when number of blocked tasks exceeds the number of threads). As a result, we create a new fixed thread pool in each task to get its subtasks to be executed. Careful configuration is also needed for fixed thread pool to make sure we prepare enough worker threads so that we can not only prevent deadlocks but also make the most of parallelism.

(b) CachedThreadPool: It is another version of thread pools. Unlike fixed thread pool, cached thread pool does not specify the number of threads (thus there is no need for configuration when creating). It ensures there is at least one active thread in the pool and sets no upper limit on the number of threads. When a new task come in, it firstly considers reusing an existing thread that is available. If there are no such threads, it will create a new thread to execute it. As a result, in this version of solution, we execute all tasks in a single cached thread pool.

4. ForkJoin: **ForkJoinPool** is a programming abstraction library that is designed for D&C algorithms. In this version, tasks are represented as objects extending **RecursiveAction** or **RecursiveTask**, where **RecursiveTask** specifies the data type of execution results. When invoking the "**start()**" function of a task, we are actually submitting it to the fork/join pool. After that, we can use the "**join()**" function to wait for the termination of the task and fetch results from them computed by their "**run()**" function in the meantime. Because fork/join pool automatically scale its parallelism to the number of available processors, it is simple to configure a fork/join pool.

5. Skandium: With the highest abstraction level, Skandium allows us to write sequential code to implement skeletons and achieve parallel computing. Specifically, to use the D&C skeleton, we need to implement four "muscle" functions, which are (1) Condition to decide whether to split the task, (2) Split to specify how to split the task into subtasks, (3) Execute to solve the base cases and (4) Merge to merge results from subtasks. Each "muscle" function corresponds to a class. To create a D&C skeleton, we pass new instances of the "muscle" functions into the constructor of "**dac**" skeleton. After that, we can invoke the skeleton in a skandium pool.

3.3 Metrics of Performance

This section will introduce the metrics that we apply to measure the performance of the programs. Generally, the performance of parallel algorithms mainly refers to the execution time, which can be quantified through the following metrics:

1. Execution time: In each run, we firstly record the execution time. The execution time equals the difference between the time of end and start of an execution, which

are fetched through the interface "**System.nanoTime()**". This interface returns a long integer representing the current value of time source in the running JVM, in nanoseconds, thus providing a high precision and resolution in time estimation [23]. Moreover, in order to avoid and eliminate the errors and oscillations that could occur during execution, we repeat every execution for several times and calculate an average value as the final execution time.

2. Speedup: Speedup reflects how parallel algorithms accelerate the resolving of problems. Specifically, taking sequential solutions as the baseline, speedup equals the time taken by the baseline divided by the execution time of parallel versions. In our implementations, all base cases, i.e., no longer split cases, in parallel solutions are solved by the corresponding sequential solution. As a result, speedup is completely resulted from parallelism, ensuring the fairness of speedup.

$$Speedup = \frac{Time\ cost\ by\ sequential\ version}{Time\ cost\ by\ parallel\ version}$$

3. Efficiency: Efficiency quantifies the ability of parallel programs to make use of available computing resource. It is calculated by the following formula:

$$Efficiency = \frac{Speedup}{Number\ of\ Processors}$$

However, CPUs today tend to support more hardware threads than physical cores [24, 25]. For example, CPU of the computer used to conduct the experiments has 8 physical cores, while each core has 2 processors which task turn to access the physical core. As a result, there are at most 8 processors working at the same time. Although time can be saved by this pattern, we still set the number of processors as 8 in our project. Moreover, as JVM does not have access to control underlying hardware, we cannot limit the number of cores. Therefore, we only experiment with 8 cores in our project.

To ensure the fairness of the data collected, we set a benchmark for each problem. Benchmarks are sensible inputs for the corresponding problem, so that they are not only solved correctly without causing memory overflow or taking too long time, but also able to show the benefits of parallelism. For example, if the input size is too small, sequential solution will take a little time, thus leading to a very low speedup. In the experiments, we firstly execute the sequential solution to each problem on the benchmark and record the execution time as the baseline. Secondly, we experiment with all parallel solutions by tuning the granularity (size of no-further-split cases) and collect the data. Values

of granularity cover the range where performance increases to peak and then drops, depending on how much parallelism is applied.

Problem	Benchmark (problem size)
Fibonacci	44-th fib number
N-Queens	15$\times$ 15 board
Quick Sort	Array with $10^8 - 1$ integers
Knapsack	40 items in total and capacity is 75
Fast Fourier Transformation	Sequence with $2^2$4signals
Adaptive Quadrature	$f(x) = \frac{(x-50)^2}{1000} + 20,\ x \in [0,\ 10^{10} - 1]$

Table 3.2: Summary of Benchmark

3.4 Metrics of Ease of Programming

This section will introduce the metrics applied to measure the ease of programming with different libraries. As base cases are solved by sequential solutions in our project, the difference of ease of programming completely comes from the code required for parallelism and synchronization.

1. SLOC: SLOC stands for ”Source Lines of Code”, which is a classic metric of size of a program and used to estimate the effort required to develop it. This is probably the most intuitive way to reflect the difficulty of programming. However, a typical drawback of SLOC is that it is easily affected by personal programming habits.

2. Halstead efforts: Maurice Halstead introduced a set of metrics introduced in 1977 to measure the complexity of a program. He treats software development as an information processing activity and use principles from information theory to derive the metrics. These metrics are based on the number of operators and operands in a program, among which is Halstead effort, aiming to represent the amount of mental effort for developing, understanding and maintaining a program [26]. It can be calculated following the steps:

   (a) Determine the number of **distinct operators** (n_1) and **distinct operands** (n_2).

(b) Count the total number of **operator occurrences (N_1)** and **operand occurrences (N_2)**

(c) Calculate the **program vocabulary n**: $n = n_1 + n_2$ and **program length N**: $N = N_1 + N_2$

(d) Calculate the **Volume V**: $V = N \times log_2 n$ and **Difficulty D**: $D = (\frac{n_1}{2}) \times \frac{N_2}{n_2}$

(e) Halstead E equals $D \times V$

In our project, Halstead effort is calculated for every parallel program with a self-written Java script. (As shown in Appendix A.1)

3. Comments: In addition to the above quantitative metrics, there are several issues that we cannot describe with a specific numeric value in parallel programming. As a result, we provide extra comments on complexity and pitfalls that can bother programmers when implementing D&C algorithms with these libraries, such synchronization issues.

Chapter 4

Experiment and Results

This chapter will introduce the conduct of our experiments, including the environment where the experiments are performed and their design. After that, we will present sample results of the experiments.

4.1 Experiment Environment

In our project, all experiments are performed on a single computer with 16 GB of memory and one 2.30 GHz Intel Core i7-11800 CPU with 8 cores, each of which has 2 threads (processors). The Operating System is Windows 11 of version 22H2. When running, programs are guaranteed to access all 8 cores. Source codes are interpreted with Java JDK 20 (Oracle OpenJDK Version 20.0.1).

4.2 Experiment Settings

This section will introduce the design and settings of our experiments. First of all, to prevent errors resulted from performance oscillations, we repeat every run five times in the experiment. Secondly, in this section, we will discuss the granularity we experiment with in our project, where granularity is the size of base cases. The setting of granularity determines the degree of applying parallelism, thus affecting the performance sharply. We try to experiment with a range of granularity as wide as possible so that the change of performance can be clearly observed from the results. Results have shown that the settings of granularity in our project have witnessed the performance raise to peak and then decrease with the further increase of granularity. Lastly, when recording execution time, we make sure all the time consumption we time in the experiment completely

comes from the algorithm instead of the whole program, i.e., excluding the time for loading benchmarks or outputting the results.

1. Fibonacci: The benchmark is to calculate the 44^{th} Fibonacci number. The first n Fibonacci numbers are calculated with sequential solution when n is less than granularity. Our range of granularity is from 23 to 39, and step is 2, where the performance peaks around 31.

2. N-Queens: The benchmark is to solve n-queens problem on a 15×15 chessboard. When the number of unoccupied columns is less than granularity, the placement of queens will be found by sequential solution on the residual boards. The range of granularity is from 10 to 14 and step is 1. The performance of all libraries peaks around 12.

3. Quick Sort: The benchmark is to sort an array of length $10^8 - 1$. When the length of the array in a subtask is less than the granularity, it will be sorted with sequential quick sort. Settings of granularity in this problem includes $25, 50, 125, 250, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000 (\times 10^4)$, where performance peaks around 50×10^4.

4. Knapsack: We need to consider 40 items to put into a knapsack of capacity 75. As the solution to a knapsack problem with n items is derived from a sub-problem on the first n-1 items, the granularity in this problems specifies that when the number of items in a sub-problem is less than the granularity, this sub-problem will be solved with the sequential solution. Granularity from 26 to 36 has been experimented, where step is 1. All libraries achieve the best performance around 29.

5. Fast Fourier Transformation: We try to calculate Fourier transformation on a sequence with 2^{24} signals, and a subtask processes sequence having less signals than granularity with sequential solution. We experiment with granularity of 2^9 to 2^{23} (multiplying by 2) and results show that when granularity is approximately 2^{19}, libraries achieve the best performance.

6. Adaptive Quadrature: We aim to calculate the area of graph under $\frac{(x-50)^2}{1000} + 20$ where domain is $[0, 10^{10}]$. We try to approximate the area of each part of the graph and when the error is less than the granularity, we continue the calculate the area with sequential AQ algorithm. The granularity ranges from 10^8 to 10^{19} (multiplying by 10), and performance peaks when granularity is around 10^{10}.

4.3 Experiment Results

In this section, we will present sample results of our experiments. We will firstly draw a table to present the performance of baseline, i.e., the sequential solution, in the benchmark of each problem. Secondly, for each problem, we will post a line graph summarizing the performance of each library with the change of granularity. Besides, for each problem, we will include a table showing their speedup and efficiency at (1) the lowest granularity, (2) the best granularity and (3) the highest granularity. Moreover, to aid the analysis of ease of programming, we will attach a table showing values of SLOC and Halstead effort of different solutions in each problem.

To make it easier to mention them, we refer to **Threading**, **FixedThreadPool** and **CachedThreadPool** as **thread** solutions while we call **ForkJoin** and **Skandium** as **abstraction** solutions.

4.3.1 Performance of baseline

This table shows the execution time of the sequential solution in each problem, calculated from 5 repeated attempts. All further calculation of speedup and efficiency of parallel solutions will be based on this.

Problem	**Execution time(s)**
Fibonacci	2.36
N-Queens	32.31
Quick Sort	9.73
0-1 Knapsack	33.64
FFT	13.38
Adaptive Quadrature	2.47

Table 4.1: Performance of baseline

4.3.2 Fibonacci

Figure 4.1 shows the speedup of different parallel solutions to the benchmark of Fibonacci with different values of granularity. It can be observed that performance of thread solutions improves when granularity increases from 23 to 33 while abstraction solutions achieve a high speedup all along. The further increase of granularity causes

less application of parallelism, thus leading a decrease of performance. It can be concluded from the graph that thread solutions achieve a better peak performance while abstraction solutions perform well on a wider range of granularity. Table 4.1.1 has shown the values of speedup and efficiency with granularity 23, 33 and 39.

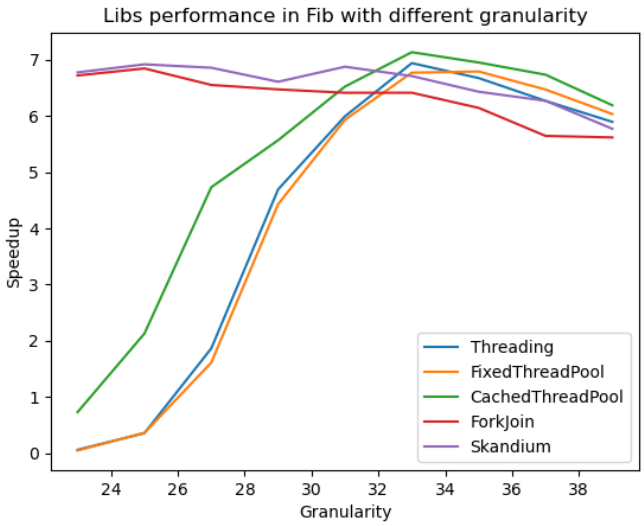

Figure 4.1: Summary of performance in Fibonacci

Granularity	23		33		39	
Solution	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**
Threading	0.06	0.0075	6.94	0.87	5.9	0.74
FixedThreadPool	0.05	0.0063	6.77	0.85	6.04	0.76
CachedThreadPool	0.73	0.091	7.13	0.89	6.19	0.77
ForkJoin	6.72	0.84	6.41	0.8	5.62	0.7
Skandium	6.77	0.85	6.71	0.84	5.77	0.72

Table 4.1.1: Performance with granularity 23, 33, 39

Table 4.2 shows the values of SLOC and Halstead effort of solutions to Fibonacci. The values are calculated based on the code to both implement and invoke the solutions. It is notable that the solution with Skandium includes the implementations of four muscle functions in four different classes.

Solution	SLOC	Halstead effort
Threading	29	11497.56
FixedThreadPool	10	12939.91
CachedThreadPool	10	8131.61
ForkJoin	16	14805.35
Skandium	24	15729.38

Table 4.2: SLOC and Halstead effort of solutions to fib

4.3.3 N-Queens

Figure 4.2 is the line graph summarizing the performance of solutions to N-Queens. It is worth mentioning that there is a lack of data when granularity is less than 10. This is because it takes too long for thread solutions to be executed. However, abstraction solutions can still achieve ideal performance, which reveals that abstractions ease the tuning when programming. When granularity is increased by 1, number of tasks created is decreased by n (the size of the chessboard, which is 15 here) times, leading the performance of thread solutions to increase sharply and peak when granularity is around 13. Table 4.2.1 has shown the values of speedup and efficiency when granularity is 10, 13 and 14.

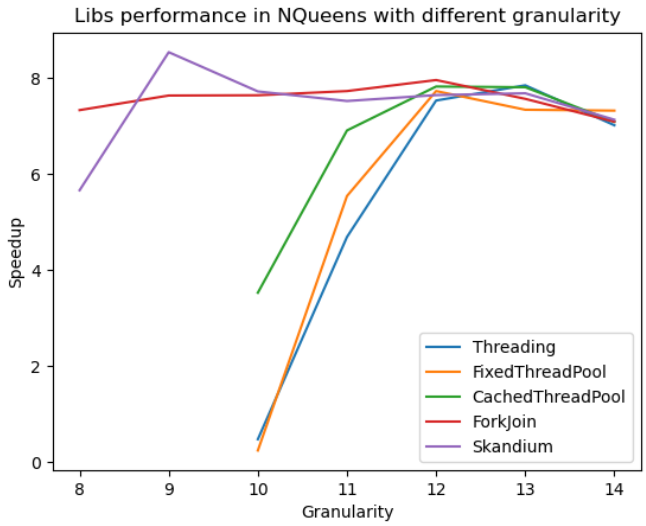

Figure 4.2: Summary of performance in N-Queens

Table 4.3 shows the values of metrics measuring the ease of implementations of solutions to N-Queens problem. In Skandium solution, besides four muscle functions,

Granularity	10		13		14	
Solution	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**
Threading	0.48	0.06	7.84	0.98	7.02	0.88
FixedThreadPool	0.25	0.032	7.34	0.92	7.32	0.92
CachedThreadPool	3.53	0.44	7.81	0.98	7.09	0.89
ForkJoin	7.63	0.95	7.57	0.95	7.09	0.89
Skandium	7.72	0.97	7.68	0.96	7.13	0.89

Table 4.2.1: Performance with granularity 10, 13, 14

we design an auxiliary class to describe the progress of solution and pass parameters, which is not required in other solutions.

Solution	**SLOC**	**Halstead effort**
Threading	52	138115.75
FixedThreadPool	37	140709.37
CachedThreadPool	37	129964.47
ForkJoin	46	147650.98
Skandium	60	202505.74

Table 4.3: SLOC and Halstead effort of solutions to N-Queens

4.3.4 Quick sort

Figure 4.3 is the summary of solutions' performance in Quick Sort. Three thread solutions no longer share similar performance as in previous cases. Specifically, direct threading achieves much better performance than thread pools. Thread pools perform best when granularity is around 1.25×10^6 while others peak when granularity is between 5×10^5 to 15×10^5. Table 4.3.1 presents the speedup and efficiency when granularity is 2.5×12.5^5, 5×10^5 and 400×10^5.

Table 4.4 shows the values of SLOC and Halstead effort of solutions to quick sort. As data passed between muscle functions in Skandium should be encapsulated in a single object to conform to the interfaces, we have an auxiliary class, which increase the quantity of source code.

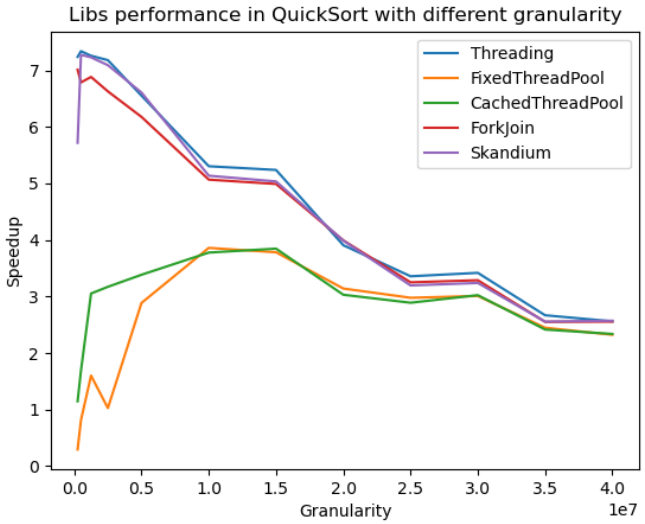

Figure 4.3: Summary of performance in Quick sort

Granularity	2.5 x10^5		12.5 x10^5		400 x10^5	
Solution	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**
Threading	7.24	0.91	7.25	0.91	2.56	0.32
FixedThreadPool	0.29	0.036	1.6	0.2	2.32	0.29
CachedThreadPool	1.15	0.144	3.05	0.38	2.34	0.29
ForkJoin	7.26	0.91	6.88	0.86	2.55	0.32
Skandium	7.33	0.92	7.23	0.9	2.57	0.32

Table 4.3.1: Performance with granularity 2.5, 12.5, 400 x10^5

Solution	**SLOC**	**Halstead effort**
Threading	38	11497.56
FixedThreadPool	26	12939.91
CachedThreadPool	27	8131.61
ForkJoin	34	14805.35
Skandium	48	94990.91

Table 4.4: SLOC and Halstead effort of solutions to Quick sort

4.3.5 Knapsack

Figure 4.5 presents the summary of performance of solutions to knapsack problem. It shows a clear trend that performance improves with the increase of granularity within 26 to 29 and starts to decrease after that. In this problem, the highest speedup is over 9,

which exceeds the number of physical cores in CPU, which is because every core has two processors. Table 4.4.1 demonstrates the concrete values of speedup and efficiency with granularity 26, 29 and 36.

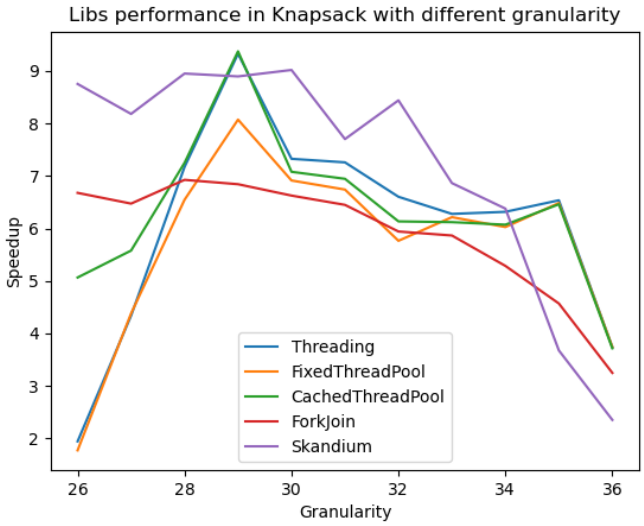

Figure 4.4: Summary of performance in Knapsack

Granularity	26		29		36	
Solution	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**
Threading	1.94	0.24	9.33	1.17	3.74	0.47
FixedThreadPool	1.77	0.22	8.07	1.01	3.74	0.47
CachedThreadPool	5.06	0.63	9.37	1.17	3.72	0.47
ForkJoin	6.67	0.83	6.84	0.9	3.25	0.41
Skandium	8.75	1.09	8.9	1.11	2.35	0.29

Table 4.4.1: Performance with granularity 26, 29, 36

Table 4.6 illustrates the values of metrics measuring ease of programming in Knapsack problem. Similarly, Skandium solution include the implementations of four muscle functions as well as an auxiliary class to encapsulate data. However, in contrast to previous cases, the Halstead effort of Threading solution exceeds the value of FixedThreadPool solution. This is because we consider two separate situations, i.e., including or excluding the current item, when splitting a problem. Threading requires us to handle the exception of execution in two branches separately, which increases the complexity of programming.

Solution	SLOC	Halstead effort
Threading	33	36334.72
FixedThreadPool	17	31441.49
CachedThreadPool	15	27996.07
ForkJoin	25	26604.87
Skandium	46	70763.33

Table 4.5: SLOC and Halstead effort of solutions to Knapsack

4.3.6 Fast Fourier Transformation

Figure 4.6 is a summary of performance of solutions to FFT. Although the general pattern of change in performance is similar to previous problems, the speedup achieved in this case is not nearly as high as in other problems. This is because for each task created during execution, we create a copy of large array of signals, which can saturate the memory bandwidth and lead to contention and slowing down the computation. We will show concrete speedup and efficiency for granularity 2^9, 2^{19} and 2^{23}. Moreover, In Table 4.6 are the values of SLOC and Halstead effort of solutions to FFT.

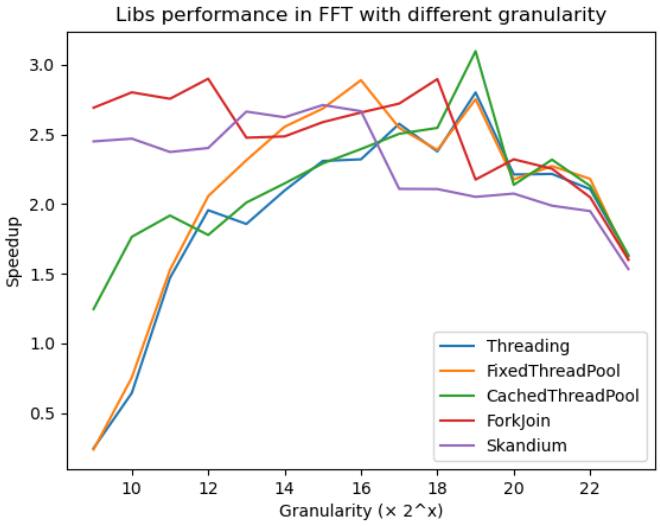

Figure 4.5: Summary of performance in FFT

Granularity	2^9		2^19		2^23	
Solution	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**
Threading	0.25	0.31	2.8	0.35	1.62	0.2
FixedThreadPool	0.24	0.03	2.75	0.34	1.6	0.2
CachedThreadPool	1.25	0.16	3.1	0.39	1.63	0.2
ForkJoin	2.69	0.34	2.18	0.27	1.6	0.2
Skandium	2.45	0.31	2.05	0.26	1.53	0.19

Table 4.5.1: Performance with granularity 2^9, 2^19, 2^23

Solution	**SLOC**	**Halstead effort**
Threading	37	54077.29
FixedThreadPool	24	57784.84
CachedThreadPool	24	55850.48
ForkJoin	30	51144.19
Skandium	48	101715.36

Table 4.6: SLOC and Halstead effort of solutions to FFT

4.3.7 Adaptive Quadrature

Figure 4.6 summarizes the performance of libraries in Adaptive quadrature. It can be clearly observed from the graph that the line is greatly similar to that in Fibonacci, which is because the degree of split and quantity of computation are similar in these problems. Performance improves when granularity increases from 10^8 to 10^{13}, and decreases then because of less parallelism. Table 4.6.1 shows speedup and efficiency of solutions when granularity is 10^8, 10^{13} and 10^{19}.

Values of SLOC and Halstead effort of solutions to Adaptive Quadrature are presented in Table 4.7. It can be observed the Halstead efforts of Threading and ForkJoin increase and both become higher than thread pool solutions. It is because we pass in six parameters in each task in this problem, which requires twice as much code to declare and assign variables in them as they follow the object-oriented programming pattern.

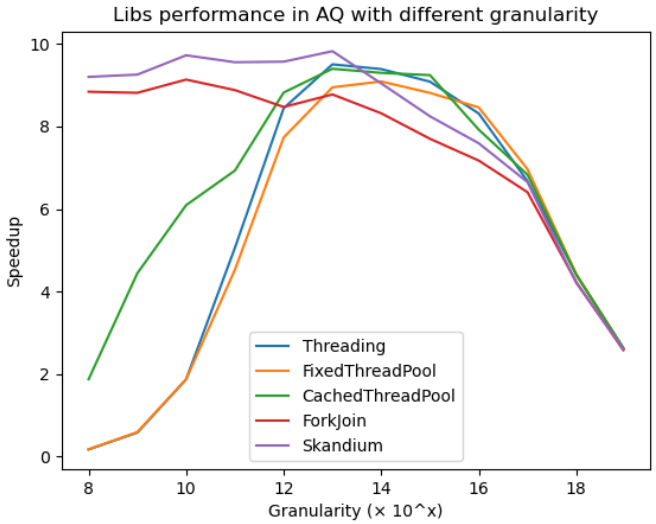

Figure 4.6: Summary of performance in AQ

Granularity	10^8		10^13		10^19	
Solution	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**
Threading	0.17	0.021	9.51	1.19	2.62	0.33
FixedThreadPool	0.18	0.023	8.95	1.12	2.62	0.33
CachedThreadPool	1.88	0.24	9.4	1.12	2.63	0.33
ForkJoin	8.84	1.11	8.78	1.1	2.58	0.32
Skandium	9.2	1.15	9.83	1.23	2.6	0.33

Table 4.5.1: Performance with granularity 10^8, 10^13, 10^19

Solution	**SLOC**	**Halstead effort**
Threading	36	37233.06
FixedThreadPool	15	28262.36
CachedThreadPool	15	26812.56
ForkJoin	30	33155.60
Skandium	42	94488.77

Table 4.7: SLOC and Halstead effort of solutions to Adaptive quadrature

Chapter 5

Discussion and Analysis

This chapter will discuss the experiment results presented in Chapter 4. Based on this, a thorough analysis will be given on the five Java libraries we selected from the aspects of performance and ease of programming. For convenience, solutions are also classified as **thread** solutions(including **Threading**, **FixedThreadPool** and **CachedThreadPool**) and **abstraction** solutions (including **ForkJoin** and **Skandium**) as discussed in Section 4.3.

5.1 Performance

In overall, five libraries can contribute a ideal speedup when their performance peaks, achieving efficiency around 1 which means they are making the most of 8 physical core (FFT is an exception because each task maintains a copy of array of signals, which leads to contention and slows down the computation). Although their peak performance varies, it can be observed from the results that thread solutions achieve a slightly higher speedup in most cases at the optimal granularity. However, in contrast to our research hypothesis, among thread solutions, CachedThreadPool seems to perform better than others while Skandium is ahead of ForkJoin. Neither of these is in the order of the degree of abstraction.

To understand why this happen, we will focus on how these libraries schedule tasks to threads. For a specific benchmark, the number of threads created by each of the five solutions is different. Specifically, among thread solutions, FixedThreadPool solution always creates the most threads as in each task, we maintain a thread pool with its size equal to the number of its possible branches (subtasks). Meanwhile, Threading solution creates a thread only when a subtask is actually forked. For example, in

N-Queens problem, we always maintain a thread pool with n threads (n is the size of chessboard) in a task while we only create a new thread when we find a position to place the queen in the current task in Threading solution. Moreover, CachedThreadPool may create threads even fewer than the number of tasks because it will firstly consider reusing a previous thread when a new task is submitted. To summarize, the number of threads created in each solution conforms to the following formula: FixedThreadPool $\geq$ Threading = Number of tasks $\geq$ CachedThreadPool. As Java is adopting a Time-Slicing scheduling strategy, which loops around and assigns a time slice to each thread no matter whether it is blocked or idle, fewer threads can save the time wasted on executing blocked and idle threads. Moreover, maintaining a new thread pool in every task require extra computation resources, which can slow down the execution in FixedThreadPool solution. Regarding to abstraction solutions, their specially designed working queue ensures subtasks to be executed first so that deadlocks can be avoided. Therefore, successful execution is guaranteed with a few threads in abstraction solutions (which by default is the number of available processors).

Furthermore, this also explains why abstraction solutions have a stable performance over a wider range of granularity. It is clear from the results that thread solutions and abstraction solutions follow two completely different changing patterns with the change of granularity. Specifically, thread solutions achieve their peak performance around the optimal granularity while abstraction solutions tolerate a much smaller granularity. A smaller granularity leads to more threads in thread solutions, making it much harder to schedule them. Nevertheless, in abstraction solutions, it will only cause more tasks to be executed in the same number of threads. With work-stealing strategy balancing the workload, abstraction solutions can still achieve a high speedup with a small granularity. When the granularity exceeds the optimal value, any further increase makes the solutions approach the sequential solution, naturally slowing down the execution.

Although thread solutions have been proved to perform well, it is notable that they provide interfaces to control underlying parallelism, which means programmers can trade ease of programming up for a better performance. For example, in our project, we experimented with **setPriority**() in thread solutions, to set the priority of threads so that subtasks can be executed in prior to their parents, which improves the performance slightly. Programmers are able the suspend those blocked threads and awake them from their children threads, while it requires them to pass more parameters and care more about synchronization issues.

To sum up, among thread solutions, CachedThreadPool performs better than others

while Skandium is better than ForkJoin in abstractions, which is different from the order of abstraction level. Comparing thread solutions and abstraction solutions, it can be concluded that thread solutions perform slightly better than abstraction solutions at the optimal granularity, while abstraction solutions perform stably over a wider range of granularity. However, it is notable that we can further improve the performance with thread solutions by trading off ease of programming.

5.2 Ease of Programming

We collected two metrics to evaluate the ease of programming. The values of SLOC is consistent across all problems that Skandium needs the most lines of source code and Threading requires the second most. FixedThreadPool and CachedThreadPool need similar number of lines of code, which specifically differ at the shutdown of thread pools. When counting the number of lines, we have excluded the lines that do not have any operations to eliminate the effect of personal programming habits.

However, SLOC is not consistent with Halstead effort, which focus more on the number of operators and operands. As the code for checking granularity, solving base cases and merging results are the same across all solutions for the same problem, the difference of Halstead effort mainly reflects the difference of synchronization and submitting new tasks. It can be seen from the results that FixedThreadPool always has a higher value for Halstead effort than CachedThreadPool as it requires more code to create new thread pools and shut it down. Threading tends to have a lower value than FixedThreadPool in most cases, whereas there are exceptions in Knapsack and Adaptive Quadrature. It is because Threading needs extra code to handle InterruptedException under two conditions separately in Knapsack, while the large number of parameters in Adaptive Quadrature affect the computation of Halstead effort according to the formula (as discussed in Secion 4.3.5 and Section 4.3.7).

The exception in Adaptive Quadrature reflect the drawbacks of the metric, because repetition of simple code does not essentially increase programming complexity. Actually, we cannot represent all difficulties with quantitative metric in parallel computing. Taking synchronization for an example, although synchronization can be achieved with a single interface ”join()” in Threading and ForkJoin, Threading needs extra code for handling exception of being interrupted while in ForkJoin ”join()” returns the result of execution automatically. However, in thread pools, synchronization is achieved in a different way, i.e., with the aid of ”CompletableFuture” class, which provides interfaces

to monitor the status of execution and fetch results. Moreover, Threading, thread pools and ForkJoin are adopting completely different programming patterns. Specifically, thread pools adopt functional programming while Threading and ForkJoin use object oriented programming, which causes a clearer structure and makes it easier to maintain the code. As a result, to start new tasks, methods of newly created objects can be invoked directly, whereas in thread pools programmers have to submit them as lambda functions. Given these features, it is not fair to determine which is easier to program with, but there are factors to consider about when programming such as the familiarity with lambda functions.

In the meantime, Skandium provide a completely different way for parallel programming, i.e., skeletons. In our project, the only involved skeleton is the **DaC** skeleton, which requires the implementation of four muscle functions. We only need simple sequential code to implement muscle functions. To invoke them, we can build the skeleton by passing in new instances of muscle functions. The muscle functions are invoked following a pipeline model and result of each function is streamed to the next one automatically. The execution of functions is parallelized by the skeleton. As a consequence, there is no need for users to know about utilization in Java about parallelism or synchronization. Everything programmers need to do is to follow a sequential programming pattern, input the problem and the result will be produced when implementing parallel D&C algorithms. It can be concluded that Skandium is the easiest way for a non-parallel programmer to implement parallel D&C algorithms.

5.3 Summary

When programming, the key idea is to achieve a balance between performance and ease of programming. As argued above, abstraction solutions, i.e., ForkJoin and Skandium, provides a stable speedup over a wide range of granularity, which not only ensures an ideal performance close to that of thread solutions, but also saves the effort for tuning granularity. However, if programmers aim to achieve better performance, utilizing interfaces provided by thread solutions is inevitable to control underlying parallelism and coordinate the schedule of threads manually. Hence, there is a trade-off between ease of programming and performance.

Chapter 6

Conclusions and Future Works

This chapter will conclude what we have done as well as our findings in the project. Based on this, we will further discuss future work for implementing parallel D&C algorithms in Java.

6.1 Conclusions

In this project, we evaluated five different Java parallel libraries over Divide and Conquer algorithms from the aspects of performance and ease of programming. Listed in ascending order of abstraction level, we focused on Threading, FixedThreadPool, CachedThreadPool (which three are called thread solutions), ForkJoin and Skandium (which two are called abstraction solutions). We implemented D&C solutions to six classic problems, including Fibonacci, N-Queens, Quick Sort, Knapsack, Fast Fourier Transformation and Adaptive Quadrature, each with five parallel versions of solutions as well as a sequential one as the baseline. By tuning the granularity (the size of not further split cases) in each problem, we experimented with the implementations and collected data about performance and ease of programming. According to the results, we have concluded that thread solutions aid a slightly higher speedup while abstraction solutions achieve an ideal performance over a wider range of granularity. Regarding to ease of programming, abstraction solutions eliminate the complexity of underlying parallelism. Specifically, there is no need to schedule tasks to threads manually or care about synchronization, which benefits non-parallel programmers greatly. Moreover, they save a lot of effort in tuning granularity. However, to pursue better performance, it is necessary for programmers to trade off between the ease of programming and intervening thread scheduling and execution through interfaces provided by thread

solutions.

In our project, there are several issues where we could make improvement. Firstly, as argued in Section 5.2, the two metrics of ease of programming do not comprehensively describe the complexity in parallel programming, while the comments are not exactly objective, which has limited our evaluation to some extent. As a result, a more comprehensive quantitative metric along with some comments might aid our evaluation on ease of programming. Secondly, the hardware environment have limited our experiment, including the memory space and the number of cores in CPU. For example, the performance of solutions to FFT is greatly limited to the size of the running memory, which led to a low efficiency. With a larger memory and a CPU with more cores, we are likely to be able to achieve more realistic performance of the libraries in more diverse environments in our project.

6.2 Future works

In the future, there are several areas for further investigation in. Specifically, there are several exceptions in the results of our experiments, which are not consistent with our theoretical expectations and analysis. For example, in contrast to other cases where thread solutions' performance are approximately the same, thread pools achieved much lower speedup in Quick Sort than direct threading. Further experiments need to be conducted to study the execution of underlying parallelism, which may help to explain the phenomenon and provide reference for programming with thread pools in the future.

Furthermore, our project has found skeletons powerful tools for programming parallel D&C algorithms. However, there is a gap of peak performance between skeletons and thread solutions. As a result, future work could be done to fill the gap on performance, to produce a new library of skeletons which not only hide the complexity of parallelism from users and ease the tuning of granularity, but also manipulate the scheduling of threads so that they can achieve equal or even better performance than thread solutions.

Bibliography

[1] Wikipedia, "Divide-and-conquer algorithm — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Divide-and-conquer_algorithm&oldid=1137028109, 2023. [Online; accessed 23-April-2023].

[2] A. Al-Adwan, R. Zaghloul, B. A. Mahafzah, and A. Sharieh, "Parallel quicksort algorithm on otis hyper hexa-cell optoelectronic architecture," *Journal of Parallel and Distributed Computing*, vol. 141, pp. 61–73, 7 2020.

[3] O. Java, "Threadpoolexecutor document." https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html. [Online; accessed 23-April-2023].

[4] O. Java, "Forkjoinpool document." https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/ForkJoinPool.html. [Online; accessed 23-April-2023].

[5] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[6] M. A. Martínez, B. B. Fraguela, and J. C. Cabaleiro, "A parallel skeleton for divide-and-conquer unbalanced and deep problems," *International Journal of Parallel Programming*, vol. 49, pp. 820–845, 12 2021.

[7] M. Leyton and J. M. Piquer, "Skandium: Multi-core programming with algorithmic skeletons," *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010*, pp. 289–296, 2010.

[8] O. Java, "Thread document." https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Thread.html. [Online; accessed 27-July-2023].

[9] O. Java, "Executors document." https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/Executors.html. [Online; accessed 27-July-2023].

[10] Baeldung, "Guide to work stealing in java." https://www.baeldung.com/java-work-stealing. [Online; accessed 29-July-2023].

[11] Wikipedia, "Work stealing." https://en.wikipedia.org/wiki/Work_stealing. [Online; accessed 29-July-2023].

[12] Intel, "TBB Document." http://www.threadingbuildingblocks.org. [Online; accessed 23-April-2023].

[13] G. Almási, C. Cascaval, and P. Wu, *Languages and Compilers for Parallel Computing: 19th International Workshop, LCPC 2006, New Orleans, LA, USA, November 2-4, 2006, Revised Papers*, vol. 4382. Springer, 2007.

[14] Wikipedia, "Fibonacci." https://en.wikipedia.org/wiki/Fibonacci. [Online; accessed 06-June-2023].

[15] Wikipedia, "N-queens problem." https://en.wikipedia.org/wiki/Eight_queens_puzzle. [Online; accessed 06-June-2023].

[16] Wikipedia, "Quicksort." https://en.wikipedia.org/wiki/Quicksort. [Online; accessed 06-June-2023].

[17] Wikipedia, "knapsack problem." https://en.wikipedia.org/wiki/Knapsack_problem. [Online; accessed 06-June-2023].

[18] Wikipedia, "Fast fourier transformation." https://en.wikipedia.org/wiki/Fast_Fourier_transform. [Online; accessed 06-June-2023].

[19] Wikipedia, "Cooley–tukey fft algorithm." https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm. [Online; accessed 06-June-2023].

[20] Wikipedia, "Adaptive quadrature." https://en.wikipedia.org/wiki/Adaptive_quadrature. [Online; accessed 06-June-2023].

[21] O. Java, "Future." https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/Future.html. [Online; accessed 02-August-2023].

[22] O. Java, "Completablefuture document." https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/CompletableFuture.html. [Online; accessed 02-August-2023].

[23] O. Java, "System.nanotime() document." https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/System.html#nanoTime(). [Online; accessed 16-July-2023].

[24] Intel, "What is hyper-threading?." https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html. [Online; accessed 30-July-2023].

[25] Wikipedia, "Hyper-threading." https://en.wikipedia.org/wiki/Hyper-threading. [Online; accessed 30-July-2023].

[26] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

Appendix A

First appendix

A.1 First section

Here is the script for calculating the Halstead effort. The code for our project has be uploaded to: https://github.com/isoland818/Parallel_Patterns

```
public static double calculateHalsteadEffort(String filePath) {
    int distinctOperators = 0;
    int distinctOperands = 0;
    int totalOperators = 0;
    int totalOperands = 0;

    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
        String line;
        Set<String> operators = new HashSet<>();
        Set<String> operands = new HashSet<>();

        while ((line = reader.readLine()) != null) {
            // Remove comments
            line = line.replaceAll( regex: "//.*|/\\*(.|\\R)*?\\*/",  replacement: "");

            // Extract operators
            String operatorPattern = "(\\+\\+|--|&&|\\|\\||\\+=|-=|\\*=|/=|%=|==|!=|<<|>>|<<=|>>=|>>>=|&=|\\|=|\\^=|<<|>>|>>>|\\(|\\)|\\[|\\]|\\{|\\}|,|\\.|;|\\?|:|=|\\+|-|\\*|/|%|<|>|!|~|&|\\||\\^)";
            Matcher operatorMatcher = Pattern.compile(operatorPattern).matcher(line);
            while (operatorMatcher.find()) {
                String operator = operatorMatcher.group(1);
                operators.add(operator);
                totalOperators++;
            }

            // Extract operands
            String operandPattern = "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b";
            Matcher operandMatcher = Pattern.compile(operandPattern).matcher(line);
            while (operandMatcher.find()) {
                String operand = operandMatcher.group();
                operands.add(operand);
                totalOperands++;
            }
        }
        distinctOperators = operators.size();
        distinctOperands = operands.size();

        System.out.println(filePath);
        System.out.println(totalOperands);
        System.out.println(totalOperators);

        System.out.println(distinctOperands);
        System.out.println(distinctOperators);

    } catch (IOException e) {
        e.printStackTrace();
    }

    double programLength = totalOperators + totalOperands;
    double programVocabulary = distinctOperators + distinctOperands;

    double V = programLength * Math.log(programVocabulary) / Math.log(2);
    double D = ((double) distinctOperators) / 2 * ((double) totalOperands) / ((double) distinctOperands);

    double effort =  V * D;

    return effort;
}
```

Figure A.1: Script for calculating Halstead effort