Parallel Algorithmic Patterns in Java

Xiaozhou Li

Master of Science

School of Informatics

University of Edinburgh

2023

Abstract

Parallel computing technology enhances performance by executing multiple computations simultaneously, and it currently finds significant applications in various fields. Based on the characteristics of algorithms, parallel algorithms can be categorized into different parallel patterns. Java offers parallel programming through its Thread library and thread pool abstraction. Implementations that involve manually managing threads are referred to as hand-threaded implementations, while implementations using thread pools for thread management are termed thread pool implementations. Thread pool implementation has a better programmability, meaning it needs less programming effort to implement, but it may not always result in better performance. The existing gap lies in the lack of research investigating the trade-off between performance and programmability of Java thread pool abstraction in implementing specific parallel pattern algorithms. This project evaluated three parallel patterns: Divide-and-Conquer, Branch-and-Bound, and Wavefront. For each parallel pattern, two different algorithms were selected and implemented in sequential, manual-threaded, and thread-pool versions. By comparing the performance and programmability of different implementations of the same algorithms, this project eventually derived insights into the suitability of various implementation methods for specific patterns, thereby providing guidance for developers in implementing parallel algorithms for particular patterns. Additionally, this project implemented a parallel algorithmic skeleton for Wavefront pattern. This skeleton hides threading and synchronization details, requiring only the logic of the Wavefront algorithm to achieve parallelism. Compared to hand-threaded and thread pool implementations, the skeleton implementation gives improvements in both performance and programmability.

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.

(*Xiaozhou Li*)

Acknowledgements

My sincerest thanks go to those who have supported me in the process of writing my dissertation.

I would like to express my deepest gratitude to Professor Murray Cole, my supervisor, for his guidance throughout the research process. Professor Cole provided me with a tremendous amount of guidance and assistance during these months. We had weekly offline meetings to discuss my progress, and the difficulties I encountered in the project as well. Professor Cole also gave me a lot of feedback during the writing phase, which kept my paper being refined.

I would also like to thank my friends from the School of Informatics. We often study together at the Appleton Tower. Such a strong learning atmosphere makes me focus on my project.

Finally, I would like to thank my family and my girlfriend for their support and encouragement.

Table of Contents

1 Introduction **1**

2 Background **3**

2.1 Parallel Programming 3

2.2 Race Condition 3

2.3 Java Parallel Abstraction 4

2.4 Parallel Pattern 4

2.5 Parallel Algorithmic Skeleton 5

3 Goals and Methodology **6**

3.1 Goals 6

3.2 Methods 6

3.3 Algorithms Selections 7

3.4 Parallel Algorithmic Skeleton 8

4 Implementations of Six Algorithms **9**

4.1 Mergesort 9

4.2 Quadrature Algorithm 10

4.3 Cocke–Younger–Kasami Algorithm 12

4.4 Longest Common Subsequence Problem 14

4.5 0/1 Knapsack Problem 15

4.6 Travelling salesman problem 16

5 Parallel Algorithmic Skeleton for Wavefront **17**

5.1 Wavefront Pattern in General 17

5.2 Requirements and Design 19

5.3 Interface 20

5.4 Implementation 21

5.4.1 WaveFrontExecutor 21
5.4.2 WaveFrontSkeleton 22
5.4.3 WaveFrontTaskFactory 23
5.4.4 Synchronization 24
5.5 Examples 27

6 Evaluation **28**
6.1 Evaluation Setup 28
6.2 Evaluation Metrics 29
6.2.1 Performance 29
6.2.2 Programmability 29
6.3 Results 30
6.3.1 Divide and Conquer 30
6.3.2 Branch and Bound 33
6.3.3 Wavefront 35

7 Conclusions and Future Work **39**

Bibliography **41**

A Supplementary Materials **44**
A.1 Skeleton Implementation of the CYK Algorithm 44
A.2 Skeleton Implementation of the LCS Algorithm 46
A.3 Class Diagram of WaveFrontSkeleton 47
A.4 Speedup for CYK Algorithm 48
A.5 Speedup for LCS Algorithm 52

Chapter 1

Introduction

Parallel computing refers to the type of computation where multiple calculations are performed simultaneously [1]. The emergence of multi-core processors introduced the concept of parallel computing [2]. In multi-core processors, each core operates independently and can execute different instructions concurrently. This significantly improves program performance and provides the possibility to address complex problem. Given the demand for high-performance computing and the rapid development of multi-core processors, parallel computing has become increasingly important in recent years. Parallel patterns[3] are introduced to define a set of algorithms with the similar problem-solving strategy. There are a lot of well-known patterns such as Divide-and-Conquer (D&C) [4], Branch-and-Bound (B&B) [5], All-Pairs [6], Wavefront [7], and so on.

Java [8], as a popular programming language over the years [9], offers excellent support for parallel programming. Java Threads [10] is a multi-thread programming library based on a shared-memory architecture, allowing developers to manually create, start, and destroy threads. This approach is known as the hand-threaded parallel implementation. Almost all the parallel algorithms can be implemented using the hand-threaded approach, but it also introduces programming complexity for developers. Therefore, Java provides several thread pool abstractions, such as ThreadPoolExecutor and ForkJoinPool. Through these thread pool abstractions, developers can submit tasks to the thread pool without the need for thread management. This approach is known as the thread pool implementation. Within these thread pool abstractions, certain abstractions are better suited for implementing algorithms that belongs to the specific parallel pattern. For instance, the fork method in ForkJoinPool corresponds to the divide phase of the Divide and Conquer (D&C) algorithm, while join corresponds to the combine phase after conquering, making it very straightforward to implement

D&C algorithms using ForkJoinPool. However, increased programmability doesn't necessarily imply improved performance. In hand-threaded implementations, manually managing threads provides developers with more control over the program, enabling them to optimize performance for specific algorithms. Therefore, it's a trade-off between performance and programmability. While the Java Thread library and thread pools have been widely adopted, there remains a lack of comprehensive research evaluating the trade-offs between performance and programmability when applying hand-threaded and thread pool methods to specific parallel patterns.

This project addresses this gap by focusing on the three patterns. We chose the mergesort and quadrature algorithms of Divide-and-Conquer, the traveling salesman problem and the 0/1 Knapsack problems of Branch-and-Bound, and the CYK algorithm and the Longest Common Substring algorithm of Wavefront pattern. The project initially analyzed the characteristics of these algorithms and provided solutions to conceptual issues encountered during parallelization. Subsequently, for each algorithm, the project implemented sequential, hand-threaded, and thread pool versions in Java. Quantitative comparisons were then made among the three implementations regarding performance and programmability. The project provided explanations and analyses of the results, eventually concluding which implementation is best suited for a specific pattern.

Another contribution of this project is the design and development of the wavefront parallel algorithmic skeleton. This skeleton is used to implement a parallel wavefront algorithm, which is based on a further encapsulation of the Java thread pool. It hides all parallel details from the developer, making it easy-to-use. We address the conceptual problems of the wavefront pattern, then abstract the wavefront algorithm and achieve relatively efficient synchronization. The evaluations demonstrate that the wavefront skeleton outperforms hand-threaded and thread pool implementations, both in terms of performance and programmability. This means that developers can achieve efficient parallelization without much programming effort.

The remainder of this paper is organized as follows. Chapter 2 describes the background and related works.. Chapter 3 gives the methodology of the project, including how we are going to achieve our project goals. Chapter 4 provides the sequential and parallel implementation details for six algorithms. Our parallel algorithmic skeleton for wavefront pattern is presented in Chapter 5, where the implementation details are explained. Chapter 6 gives performance and programmability evaluations for different implementations of the six algorithms, as well as an evaluation of the skeleton we proposed. Finally, conclusions and future works are presented in Chapter 7.

Chapter 2

Background

This chapter gives the background knowledge of the key concepts and technologies involved in this project, and summarizes the related work.

2.1 Parallel Programming

A great number of parallel programming languages, application programming interfaces (APIs), or libraries have been developed for parallel computing. Based on the memory architecture they adopt, these parallel programming languages can be classified into two categories. The first type is message passing based parallel programming. The Message Passing Interface (MPI) [11] is a widely used API for message passing parallel programming. The second type is shared memory-based parallel programming languages, where communication between processes or threads occurs through the manipulation of shared variables in shared memory. For example, Pthreads [12] and OpenMP [13] are two well-known shared memory parallel programming libraries. Java Thread is a form of multi-thread programming library based on shared memory architecture.

2.2 Race Condition

If two threads access shared data simultaneously, it can potentially lead to a problem known as a race condition [14], which refers to the issue of data inconsistency. However, simultaneous access to shared data doesn't necessarily result in a race condition. A race condition may occur if one thread is reading data while another thread is writing to it, or if both threads are writing data at the same time. However, if two threads are reading the

same shared data concurrently without involving any write operations, there won't be a race condition. One way to address race conditions is through synchronization, which ensures that only one thread access on shared data at any time. The extra complexity and difficulty of parallel program often come from synchronization.

2.3 Java Parallel Abstraction

The most common approaches to create threads in Java involve extending the Thread class or implementing the Runnable interface [10]. The run() method needs to be implemented as it serves as the starting point for thread execution. Java also provides multiple techniques for thread synchronization. The synchronized keyword, for example, is utilized to synchronize code blocks or methods, thereby allowing only one thread to execute the code block or method at a time. The raw use of Java Thread, which involve manually managing threads as well as implementing thread synchronization, is referred to as hand-threaded implementation. On the other hand, the method of using Java thread pool abstraction to manage threads is referred to as thread pool implementation.

A thread pool is a pre-created collection of threads that can be reused to execute multiple tasks. It provides a framework for managing threads and concurrent task execution. In Java, thread pools enhance performance and reduce overhead by avoiding creating and destroying threads for each task. Compared to manual threading, thread pooling implementations eliminate the need for developers to manually manage threads and tasks. This reduces programming complexity and also ,may improve overall performance.

2.4 Parallel Pattern

A parallel pattern[3] is a set of algorithms that follow the the same or similar problem-solving strategies. This section introduces three common parallel patterns, which are also the three patterns that need to be evaluated in this project: Divide-and-Conquer, Branch-and-bound, and Wavefront.

Divide-and-Conquer (D&C) [4] is one of the most commonly used parallel patterns. In this pattern, a problem is recursively divided into smaller subproblems until the subproblems can be solved directly. The solutions to the subproblems are then collected and combined to generate a solution to the original problem.

Branch-and-bound (B&B) [5] pattern is also a well-known parallel pattern used in solving searching and optimization problems. The solution space is divided into a tree structure, where each node represents a sub-problem that needs to be solved. B&B pattern checks the boundary to eliminate sub-problems that cannot contain the optimal result when trying to solve the sub-problems.

Wavefront [7] is a typical parallel pattern, generally considered as a 2-dimensional dynamic programming problem. In most cases, algorithms that belong to the wavefront pattern start from one corner of a 2-dimensional array and scan the elements following a diagonal order, eventually reaching the opposite corner of the array. In the wavefront pattern, the computation of each element in the array relies on the results of previously computed elements. Such data dependencies necessitate the algorithm to scan the 2D array in a diagonal direction.

2.5 Parallel Algorithmic Skeleton

It takes more effort to implement parallel program than sequential one. Programmers are required to manually manage threads, handle synchronization between threads, and deal with potential issues like deadlocks. To tackle these challenges, the concept of parallel algorithmic skeletons has been introduced in research [15] [16]. These skeletons serve as elevated programming frameworks that offer a methodical approach for constructing parallel programs. By utilizing them, developers can concentrate on the algorithm's overarching logic, abstracted away from intricate parallelism intricacies. This methodology streamlines the process of crafting parallel programs.

There is a lot of related work on algorithmic skeletons. The DAC parallel template [17] is an algorithm template that supports three runtime environments: OpenMP [13], Intel TBB [18], and FastFlows [19]. Martínez et al. proposed the parallel_stack_recursion skeleton [20]. This is a C++ based algorithmic skeleton for the D&C pattern, built on top of parallel_recursion [21]. QUAFF [22] is an algorithmic skeleton that uses C++ template to achieve high performances. Murray Cole proposed a distributed memory based parallel skeleton called ESkel [16]. MUESLI [23] is a C++ skeleton library built on the basis of MPI and OpenMP. Lithium [24] is a Java library that use macro data flow to implement the parallel skeleton. Skandium [25] is an algorithmic skeleton in the Java environment. However, there are fewer skeletons for some patterns, such as wavefront and all-pairs patterns.

Chapter 3

Goals and Methodology

This chapter describes the goals and methodology of the project, including the problems to be addressed in this project, how to achieve the project goals, the choice of parallel patterns and algorithms, and the reasons for making the choices.

3.1 Goals

There is a lack of comprehensive investigation of the trade-off between performance and programmability when using raw Java thread and thread pool to implement algorithms that belong to specific parallel patterns. This gap makes it challenging for developers to decide which approach to use when programming specific patterns. Therefore, our first goal is to address this problem by conducting empirical research to compare thread pool and hand-threaded implementations, and deriving conclusions about the suitability (performance and programmability) of different implementations for specific parallel patterns. Furthermore, the existing parallel algorithmic skeleton in the Java do not fully support certain patterns, posing challenges for developers unfamiliar with parallel computing to implement those pattern. Our second goal is to abstract a specific parallel pattern, design and develop a efficient parallel algorithmic skeleton that hide low-level threading details and thus reduce the programming effort required.

3.2 Methods

For the first goal of this project, we have chosen the Divide-and-Conquer (D&C), Branch-and-Bound(B&B), and Wavefront patterns, which are common in parallel computing. The D&C pattern involves breaking down a problem into smaller subproblems

and solving them in parallel. B&B enhances efficiency by constraining the search space in searching problems. Wavefront involves solving a problem stage by stage. By studying these three typical patterns, we can gain a deep understanding of parallel approaches for different types of problems and provide guidance for broader applications. For each algorithm, as described in Section 3.3, we will implement sequential, hand-threaded, and thread pool versions. We will then evaluate the performance and programmability of the three implementations using various metrics. By considering the results of both performance and programmability, we can summarize whether each implementation approach is suitable for a specific parallel pattern.

For the second goal of the project, we will select a specific parallel pattern and design a parallel algorithmic skeleton for it. We will refer to existing skeletons, considering how to implement task partitioning and thread synchronization to balance performance and programmability. Then, we will use this skeleton to implement the chosen algorithms again and test their performance and programmability. By comparing the results with other three implementations, we will analyze whether our skeleton has any improvements in terms of either performance or programmability.

3.3 Algorithms Selections

For each parallel pattern, we have selected two different algorithms. Our goal is to explore the suitability of thread pool abstraction and hand-threadeding to various parallel patterns. Thus, we want these two algorithms to be representative enough to generalize the results of performance and programmability to common patterns. We aim to select algorithms that belong to the same pattern but differ in details. By studying two variants of a pattern, we can deduce properties that apply to the entire parallel pattern.

For the D&C pattern, we have chosen the mergesort and quadrature algorithms. Mergesort divides an array into smaller subarrays, sorts each subarray, and then merges the sorted subarrays to obtain a sorted original array. Quadrature algorithm recursively applies two different rules to estimate the integral of the subintervals of an interval, until the difference between the estimations of the two rules is small enough. The integral estimations of the sub-intervals are summed to obtain the accurate estimation of the original interval. The sub-tasks of mergesort are load-balanced as the subarrays have equal sizes. However, the sub-tasks of the Quadrature algorithm are not balanced due to potentially different depths in the recursion of subintervals. Furthermore, the merging subarrays can only be done by a single thread in mergesort algorithm, which leads to

the lower parallelism compared to quadrature algorithm.

For the B&B pattern, we have selected the 0/1 Knapsack problem and the Traveling Salesman Problem (TSP). Given a weight constraint, the Knapsack problem aims to maximize the total value of items in a knapsack. TSP involves finding the shortest route for a salesman to visit all cities and return to the starting city. Both problems construct a search tree and employ pruning strategies to discard certain nodes and their subtrees. The difference lies in the fact that the Knapsack problem has only two child nodes for each node, while the TSP's number of child nodes depends on the remaining unvisited cities. This results in deeper trees for Knapsack and wider trees for TSP for problems of the same size.

For the Wavefront pattern, we have chosen the Cocke-Younger-Kasami (CYK) algorithm and the Longest Common Subsequence (LCS) algorithm. The CYK algorithm is used for parsing, determining whether a sentence is in a given language. The LCS problem involves finding the longest common subsequence between two sequences. They are both 2D dynamic programming problems, but CYK's direction is from bottom-left to top-right, while LCS's direction is from top-left to bottom-right. CYK starts from the diagonal of the 2D array, while LCS starts from a corner of the 2D array. Additionally, the computational complexity of each cell in CYK algorithm increases as the program goes on, while cell computations in LCS algorithm are load-balanced.

3.4 Parallel Algorithmic Skeleton

We have chosen to develop a parallel algorithmic skeleton for the Wavefront pattern. As one of the most common parallel patterns, there are already numerous Java parallel skeletons that support the D&C pattern, such as Skandium, Muskel, and Lithium. There have also been research providing Java-based parallel frameworks for the B&B pattern, like Grid'BnB. However, there is relatively limited research on Java algorithmic skeletons for the Wavefront pattern. CO2P3S is a Java-based multi-pattern parallel programming framework. However it was introduced quite early and lacks comprehensive support for the Wavefront pattern, for instance, it only supports the top-left to bottom-right direction. Our aim is to design a more versatile Wavefront algorithmic skeleton that can be applied to various problems. The Wavefront parallel pattern has widespread application in practical problems such as image processing and graph algorithms. Developing an efficient and easy-to-use parallel algorithmic skeleton can assist developers in easily implementing parallel algorithms for Wavefront pattern.

Chapter 4

Implementations of Six Algorithms

This chapter mainly describes 6 different algorithms belong to 3 parallel patterns that we need to implement. For each algorithm, we fist briefly describe what the algorithm does and the logic of it. Then we talk about the conceptual problems we met for each algorithm, such as synchronization difficulties, followed by our solution to it. Finally we explain in detail how we implemented the sequential, hand-threaded, and thread pool versions of the algorithm.

4.1 Mergesort

We borrow the mergesort algorithm from [26] to implement our sequential mergesort. We split the array recursively untilthere's only one element requiring sorting in the subarray. Then the merge method is called to merge the sorted subarrays into a single sorted array. The merge method modifies the original array rather than returning a new one. In the merge method, two temporary arrays that copy the left and right sub-arrays are initialized. Then we traverse both left and right array at the same time, and write the smaller elements back to the original array. When both left and right are traversed, the original array has also been sorted.

We utilize the ForkJoinPool to implement the threaded version. As mentioned before, the ForkJoinPool is well-suited for divide-and-conquer algorithms. In the hand-threaded version, we introduce a Task class that represents the tasks submitted to the thread pool. This class inherits from the RecursiveAction class and can be submitted to the ForkJoinPool. The Task class has three member variables: the array representing the array, and the index bounds low and high indicating the range to be sorted. Inheriting from RecursiveAction requires implementing the compute() method, which closely

follows the approach of the sequential merge sort. We first determine if the array can be further divided based on variables low and high. Then we create two Task instances to handle the left and right parts of the array. Next, we call the invokeAll method to submit both Task instances to the thread pool. As invokeAll is a blocking operation that waits until both tasks have completed execution, no additional thread synchronization is needed. Finally, we call the same merge method as in the sequential version to merge the two sorted subarrays into a single sorted array.

In the hand-threaded version, we have a Task class that implements the Runnable interface, allowing us to create new threads on tasks. Implementing the Runnable interface requires implementing the run method. In addition to the array, low, and high member variables, the hand-threaded Task also includes a variable called availableThreads, indicating the number of threads available for sorting this array segment. In the run method, apart from checking whether the task's array can be further divided, we also need to check the availableThreads. If availableThreads equals 1, it means that only one thread is available to handle this array segment. In this case, even if further division is possible, there are no additional threads. Thus, we directly call the sequential merge sort algorithm to sort this portion of the array. If availableThreads is greater than 1, we create and start two threads to handle the divided subarrays. Each of the two child tasks should have availableThreads set to half of the original availableThreads. Finally, we use the join method to wait for both threads to return. Afterward, we call the same merge function as in the sequential version to merge the two sorted subarrays into a single sorted array.

4.2 Quadrature Algorithm

The sequential version of quadrature algorithm uses a recursive implementation of the trapezoidal rule. It first computes the integral estimate of the entire interval using the trapezoidal rule, and divides the interval into two subintervals. Next, it applies the trapezoidal rule to each of the two subintervals separately and adds the results up to get the second integral estimates for the entire interval. If the difference between the two estimates is smaller than the tolerance, or the interval itself is smaller than the threshold, the function returns the second estimate as the final result. Otherwise, it recursively divides the subinterval into two even smaller subintervals to compute the integral estimates. The final result is obtained by summing up the sufficiently accurate estimates of the interval integrals.

We still choose to implement the thread pool version using ForkJoinPool. However, the Task class now inherits from RecursiveTask¡Double¿ instead of RecursiveAction, as the quadrature algorithm's compute method has a return value, whereas RecursiveAction cannot have a return value. In fact, we could maintain a shared variable and accumulate the values whenever the integration estimate is accurate enough. However, this approach would require frequent locking and unlocking to ensure data consistency, so we do not adopt this method. Since we need to obtain return values, the program uses fork method to execute task and join method to get the return value, rather than the calling invokeAll method. In the end, when both subintervals have been computed, the program adds the integration estimate values of the sub-intervals and returns it as the integral estimate for the entire interval.

Algorithm 1 Bag-of-tasks: Task Retrieval Algorithm

lock.lock();
while empty queue **do**
    if counter.get() == 0 **then**
        cond.signalAll();
        lock.unlock();
        return;
    cond.await();
// retrieve task
counter.incrementAndGet();
lock.unlock();

Algorithm 2 Bag-of-tasks: Task Submitting Algorithm

if submit task **then**
    lock.lock();
    // submit tasks
    counter.decrementAndGet();
    cond.signalAll();
    lock.unlock();
else
    counter.decrementAndGet();

For the hand-threaded version, we cannot adopt the same approach as in the merge sort, as the quadrature algorithm doesn't ensure load balancing for each interval, even

if the interval sizes are the same. To prevent threads from being idle, we utilize the Bag-of-tasks strategy. Each Task instance represents a computation task for a specific interval. We maintain a shared double ended queue named tasks to store the tasks, using a Lock lock and a Condition cond to implement thread synchronization when accessing the shared variable. The termination condition for threads is not solely based on an empty queue, as new tasks might still be enqueued by other working threads. Hence, we introduce an AtomicInteger counter to keep track of the number of working threads. Threads can exit only when the queue is empty and the counter is 0. The pseudo code for thread task retrieval is shown in Algorithm 1, and the pseudo code for submitting tasks is shown in Algorithm 2. We use a thread-local variable named result to accumulate the integral values computed by each thread. Finally, in the main function, we sum up the result values from all threads to obtain the final result. Using thread-local variables avoids the use of a critical section and improves performance.

4.3 Cocke–Younger–Kasami Algorithm

The CYK algorithm differs from other wavefront algorithms 2.4 in that it starts its computation from the diagonals of the 2D array, rather than corner. We use a 2D array of Maps to store whether each substring has matched non-terminal symbols and their combination counts. The number of rows and columns in the chart equals the length of the input string. During the initialization phase, the diagonal elements are initialized with the terminal symbols of the string. In the chart filling phase, in a sequential implementation, each cell is filled in the order shown in the Figure 4.1. The algorithm starts by checking substrings of length 2 and proceeds to the last cell, which corresponds to the input string itself. For each cell, the algorithm considers every possible partition of the substring into two parts and checks whether there is a rule A ::= BC in the grammar, where B matches the first part and C matches the second part. If such a rule exists, the combination count of B and C is multiplied, and added to the combination count of A in the Map of that cell. Take Figure 4.2 as an example, for the red cell, with a substring length of 4, there are three possible partitions: 13, 22, and 31. Thus, we need to sequentially check the two blue, two green, and two yellow cells. When the computation of the last cell,which is the upper-right corner, is completed, if there is a matching with the starting symbol, it indicates that the string belongs to the grammar. The sequential CYK strictly follows the above process, filling each cell in the order given in the Figure 4.1 until the last cell.

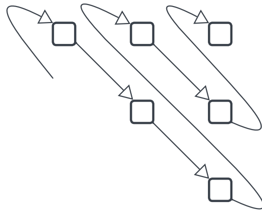

Figure 4.1: CYK Order

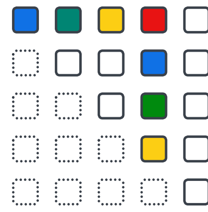

Figure 4.2: CYK Filling

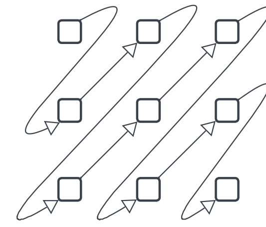

Figure 4.3: LCS Order

We still use ThreadPoolExecutor to create a fixed-size thread pool. Although the CYK algorithm can be expressed as a divide-and-conquer approach starting from the last cell, additional measures are needed to avoid duplicate cell calculations. Therefore, using ForkJoinPool wouldn't provide extra convenience for us. We still use the Task class that implements the Runnable interface. Each task has the starting indices i and j of the cell, as well as the task size size. Each task needs to consecutively fill size cells starting from i and j. Size is influenced by the CHUNK_SIZE specified by the developer and is used to adjust the granularity of tasks. We can found that all cells have data dependencies on the two cells to their left and below. This dependency can be transformed into dependencies between waves, as shown in the Section 5.1. We introduce a 2D AtomicInteger array to track cell computation status, where each AtomicInteger corresponds to a cell in the chart. Whenever a cell completes computation, it increments the corresponding AtomicInteger of the cells above and to the right. If the value of an AtomicInteger is 2, it means the cells it depends on have finished computation, and that cell can start computation. Whenever all cells of a task are computed, it checks if the corresponding task of the next wave can start computation. If so, it submits the task to the thread pool. In summary, the order of chart filling in the thread pool version differs from the sequential one, but the logic for cell filling remains the same.

For the hand-threaded implementation, we adopt the same synchronization strategy as the thread pool. We continue to use Task as the task, and each task still needs to computes size cells. Similarly, we use a 2D AtomicInteger array for thread synchronization, tracking cell execution process and submitting new tasks. Since there's no thread pool, we need to manually manage tasks, which is why we still use the Bag-of-tasks strategy. The logic for retrieving and submitting tasks remains the same with Algorithms 1 and 2. The run method of the Task is an infinite loop, where each iteration involves obtaining a task, filling all the cells in the task, and then attempting to submit new tasks.

4.4 Longest Common Subsequence Problem

The Longest Common Subsequence (LCS) problem is also a 2D dynamic programming problem. We declare a 2D array named chart of type short and ensure that the lengths of both strings do not exceed the maximum value of short. This is to prevent potential memory overflow issues. The core of the LCS algorithm involves comparing all subsequences of two strings. If the i-th element of subsequence x_i matches the j-th element of subsequence y_j, a common element has been found, so the length of the LCS should be increased by 1 based on the previous cell. Otherwise, we take the maximum value of the lengths of the LCS from the two previous subsequences. In the sequential version, due to the data dependencies with the left and upper cells, we perform the chart filling operation in the order shown in Figure 4.3. For each cell, we compute according to the transition equation as shown in Equation 4.1. The value in the last cell represents the final result, which is the length of the longest common subsequence.

$$chart[i][j] = \begin{cases} 0 & ij = 0 \\ chart[i-1][j-1]+1 & ij \neq 0 \quad and \quad x_i = y_j \\ max\{chart[i-1][j], chart[i][j-1]\} & ij \neq 0 \quad and \quad x_i \neq y_j \end{cases} \tag{4.1}$$

We continue to use the ThreadPoolExecutor with a fixed-size thread pool for thread pool version. We also use the Task class to implement the Runnable interface, where each task is responsible for computing a sequence of contiguous cells within the same wave, as explained in section 5.1. However, due to the potentially large scale of the LCS problem, creating an AtomicInteger for each individual cell might lead to memory overflow. To address this, we use a single AtomicInteger, namely counter, to track the progress of the current wave. Before tasks for a wave are submitted, the counter is initialized to the number of cells in that wave. The wave is then divided into tasks of size CHUNK_SIZE as specified by the developer, and all these tasks are submitted to the thread pool. When a task completes the computation for all its cells, it decrements the counter by the number of cells it processed. Based on the previously discussed data dependency relationship, when the counter reaches zero for the current wave, it implies that all elements of that wave have completed their computations. At this point, we can initiate the calculation for the next wave. We then reset the value of the counter to the number of cells in the next wave and submit tasks for all cells in the next wave to the thread pool. This synchronization mechanism ensures that wave-by-wave computation progresses in the correct order while utilizing the thread pool effectively.

For the hand-threaded implementation of the LCS algorithm, we employ the same synchronization strategy as the thread pool version, using an AtomicInteger to track the progress of the current wave. Similar to the CYK algorithm, since we need to manage tasks manually, we continue to use the Bag-of-tasks strategy. The logic for retrieving and submitting tasks aligns with the approach described in algorithms 1 and 2.

4.5 0/1 Knapsack Problem

The brute-force approach for solving the 0/1 knapsack problem has a time complexity of O(2^n). In this approach, a binary tree of height equal to the number of items is constructed, with each level representing an item, and each node having two children representing whether the item is included or not. By traversing the leaf nodes of this tree, we can find the optimal solution. To optimize this brute-force approach, we use a technique called Branch-and-Bound proposed in [27], which involves pruning the search space. When a node's corresponding subtree's best solution is worse than the current optimal solution, we can safely ignore that node and its subtree. Since updating the optimal solution requires reaching a leaf node, the sequential algorithm actually employs a depth-first search, traversing the entire tree until all possibilities are explored. At that point, the current optimal solution becomes the global optimal solution.

For the thread pool version, we use ThreadPoolExecutor to create a fixed-size thread pool. In a multi-threaded environment, we use an AtomicInteger variable bestValue as the global optimal solution. When updates are needed, we call bestValue.getAndAccumulate(newValue, Math::max); to update the optimal solution. Using AtomicInteger helps avoid using locks and thus improves performance. Each task in the thread pool corresponds to a node in the search tree. The first step is to check whether the node is a leaf node. If it is a leaf node, we need to update the optimal solution. Otherwise, the node generates two child nodes representing whether to include the current item or not. Next, we check whether these two child nodes can be pruned based on the Branch-and-Bound rule. If they cannot be pruned, we submit the node as a task to the thread pool.The algorithm continues until there are no more working threads in the thread pool, indicating that all possibilities have been explored.

For the hand-threaded version, we continue to adopt the Bag-of-tasks strategy, which maintain a task queue using a lock and a condition variable. The logic for extracting tasks and submitting tasks remains the same as described in algorithms 1 and 2. However, there is a difference in how tasks are submitted and retrieved compared

to other algorithms. When submitting tasks, they are added to the front of the double-ended queue, and when retrieving tasks, they are obtained from the front of the queue too. This is done to ensure that, even in a multi-threaded environment, the traversal of the tree still follows a depth-first approach. This is important because if the traversal becomes closer to breadth-first. Since updates to the optimal solution can only occur at leaf nodes, a breadth-first search would degrade into a brute-force search. Other than this difference, the hand-threaded version shares the same algorithm logic as the thread pool version.

4.6 Travelling salesman problem

The brute-force algorithm for TSP involves considering all possible permutations of cities, resulting in a time complexity of O(n!), where n is the number of cities. Our sequential version borrows the original algorithm proposed in [28], which builds upon the brute-force approach by introducing the Branch-and-Bound. The borrowed algorithm employs a depth-first search approach. When reaching a leaf node, we update the current best solution. Otherwise, we calculate the lower bound and prune branches, continuing until we traverse the entire tree. At that point, the current optimal solution becomes the global optimal solution.

For the thread pool version of the TSP, we still use the ThreadPoolExecutor to create a fixed-size thread pool. Since travel costs are represented as floating-point, and the concurrent package does not provide an AtomicDouble class, we are limited to using primitive double type. To ensure thread safety when updating the minimal travel cost bestValue, we use a read-write lock, which enhances performance. The first step of a task is to check whether the corresponding node is a leaf node. If it is a leaf node, the task updates the best solution. Otherwise, the task creates child nodes corresponding to the unvisited cities. Then it calculates the lower bound of the subtree rooted at each child node and compares it with the bestValue to determine whether to prune or submit the task for execution. The algorithm continues until the thread pool has no working threads, meaning that all routes have been considered.

In the hand-threaded version of TSP, we continue to use the Bag-of-tasks strategy, as described in algorithms 1 and 2. Similar to the knapsack problem, tasks are submitted and retrieved from the front of the double-ended queue. This approach helps prevent the algorithm from degrading into breadth-first search and ensures that the Branch-and-Bound technique is effective.

Chapter 5

Parallel Algorithmic Skeleton for Wavefront

This chapter mainly describes the implementation of parallel algorithmic skeleton for Wavefront pattern. This chapter first extract the core of Wavefront pattern, and abstract the pattern in a genral way. Then it describes the requirements and design of the skeleton, followed by explanations of the interface and usage of the skeleton. Finally the chapter explains about the conceptual work and how the skeleton is actually implemented, including the functional modules and how to achieve synchronization.

5.1 Wavefront Pattern in General

We have abstracted and simplified the typical wavefront problem, as shown in the Figure 5.1. There are four possibilities of the wavefront direction. The Figure illustrates a wavefront algorithm from the top-left corner to the bottom-right corner, where each box represents an element or cell in a 2D array. We refer to this 2D array as the chart. In the diagram, we can observe solid and dashed boxes, the reason for this representation is due to the facgt that not every element in the chart participating in calculations. Certain wavefront algorithms initialize specific rows and columns (typically the row zero and column zero) of the chart with certain values, which are then directly used in subsequent computations. As a result, we use dashed boxes to represent elements that are not involved in actual calculations, while solid boxes are used to represent elements that are computed within the wavefront algorithm.

It's noteworthy that all the solid boxes combined still form a 2-dimentional array, referred to as the subchart, which is a subset of the big chart. We denote the number of

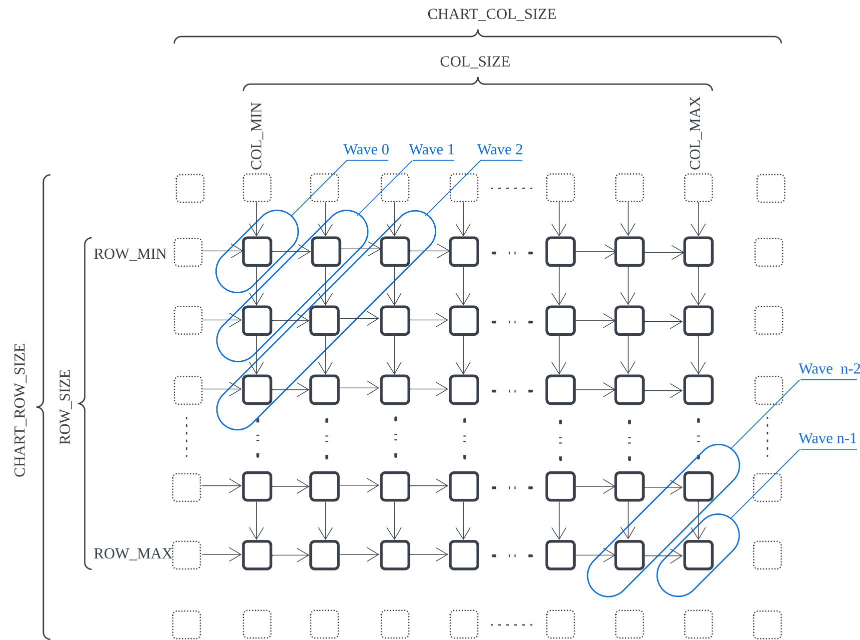

Figure 5.1: Abstraction of the Wavefront Pattern

rows in the entire chart as CHART_ROW_SIZE, the number of columns in the entire chart as CHART_COL_SIZE, the number of rows in the subchart as ROW_SIZE, and the number of columns in the subchart as COL_SIZE. The relationship ROW_SIZE $\leq$ CHART_ROW_SIZE and COL_SIZE $\leq$ CHART_COL_SIZE always holds. ROW_MIN and ROW_MAX respectively represent the minimum and maximum value of the row indices of the subchart. COL_MIN and COL_MAX represent the minimum and maximum column indices of the subchart.

The arrows in the Figure 5.1 represent the data dependency relationships between elements in the chart. Typically, in a wavefront algorithm that progresses from the top-left to the bottom-right, the computation of an element depends on the elements above and to the left of it, as shown in the Figure 5.1. That is, chart[i][j] depends on chart[i-1][j] and chart[i][j-1]. Due to these dependencies, the wavefront algorithm follows a specific execution order. We refer to elements located on the same diagonal as a ”wave”, as indicated by the blue circles in the Figure 5.1. Each wave can be regarded as a stage of the algorithm. Only when elements in the previous wave have completed their calculations can the computation of elements in the current wave start. Sometimes, the computation of a wave N may not solely depend on wave N-1, but may also rely on even earlier waves such as wave N-2. For example, chart[i][j] may depend not only on chart[i-1][j] and chart[i][j-1] but also on chart[i-1][j-1]. However, since the dependency

relationships exhibit transitivity, which means, the condition that wave N depends on wave N-1 implicitly includes the fact that wave N-1 depends on wave N-2, we do not need to explicitly state that wave N depends on wave N-2. As a result, we can conclude that, in the wavefront pattern, wave N can start its computation once wave N-1 has completed its calculations.

5.2 Requirements and Design

The primary objective of designing and developing the parallel algorithmic skeleton for wavefront pattern is to reduce the complexity of programming for programmers while introduce parallelism to enhance the efficiency of wavefront algorithms. Ideally, we aim for developers to use this skeleton to implement parallel algorithms with only a comprehension of the wavefront algorithm itself, and without any prior knowledge of parallel computing. To achieve this goal, our algorithm skeleton must meet the following requirements: (1) The algorithmic skeleton should hide developers from threading. Developers should not need to explicitly manage threads or synchronization. (2) The algorithmic skeleton should provide a easy-to-use interface that enables developers to define a wavefront algorithm easily. (3) The algorithmic skeleton should make its hyper parameter configurable, which ensure the generality while achieving higher parallel performance based on the specificity of different algorithms.

To fulfill these requirements, we have decomposed the skeleton based on functionality and defined three distinct functional modules: WaveFrontSkeleton, WaveFrontExecutor, and WaveFrontTaskFactory. The WaveFrontSkeleton focuses on defining the wavefront algorithm. The WaveFrontExecutor is responsible for creating and managing the thread pool. The WaveFrontTaskFactory manages how tasks are processed and how new tasks are generated as well. This module is hidden from developers. This modular approach enables a clear separation of concerns and encapsulates the underlying complexities of parallel execution, enabling developers to focus on the wavefront algorithm's logic without being distracted by the intricacies of parallelism and thread management.

There can be significant variations among different wavefront algorithms. Each element in the chart which we refer to as a cell, may have a different computation strategy. In certain algorithms, all cells are load balanced, and the computational workload of each cell is minimal. In such cases, if tasks are assigned to threads at the level of individual cells, a substantial amount of time can be consumed in thread

synchronization, such as competing for the task queue. Conversely, some algorithms may have imbalanced workloads between cells, with some cells requiring significantly more computation time than others. In such cases, if task granularity is not fine enough, thread idle time may occur, which is detrimental to the performance of the parallel program. For this reason our skeleton also supports a hyperparameter called CHUNK_SIZE, which need to be specified by developers. This parameter remains constant throughout the skeleton's execution. CHUNK_SIZE determines the maximum number of cells contained within a chunk. All elements within a chunk belong to a single wave and are contiguous. Situations where the number of cells in a wave is not evenly divisible by CHUNK_SIZE may result in tasks with cell counts smaller than CHUNK_SIZE. Each chunk corresponds to a single task, and when a thread obtains a task, it computes all elements within the chunk in a loop. This approach provides flexibility to adopt the task granularity, which provide possibility for developers to optimize program performance.

5.3 Interface

The full version of example code can be found in 5.5. To utilize the wavefront skeleton for parallel computation, developers need to initialize a thread pool with a specified number of threads through the WaveFrontExecutor module. WaveFrontExecutor is implemented as a singleton to manage the thread pool, which can be referred to in section 5.4. Developers need to call the `public void newExecutor(int maxThreads)` method, where the maxThreads represents the desired number of threads in the thread pool.

After initializing the thread pool, developers need to define a wavefront algorithm using the WaveFrontSkeleton. Depending on the data type of the chart, developers need to choose an appropriate abstract class to inherit from. Four abstract classes are provided for this purpose, which can be found in section 5.4. Take `WaveFrontSkeleton<P, R>` as an example, it requires developers to provide the data type P of the chart and the return data type R. It would also require the developer to implement the constructor method and other two abstract mathods. Their method signatures are as follows:

- `public WaveFrontSkeleton(P[][] chart, int CHART_ROW_SIZE, int CHART_COL_SIZE, int ROW_MIN, int ROW_MAX, int COL_MIN, int COL_MAX, int CHUNK_SIZE, WaveFrontDirection direction)`: The constructor method, where chart is an initialized two-dimensional array of type P.

Developers should also provide the following parameters: CHART_ROW_SIZE, CHART_COL_SIZE, ROW_MIN, ROW_MAX, COL_MIN, and COL_MAX. These parameters define the size of the algorithm, as discussed in the previous section 5.1. Additionally, developers need to specify the CHUNK_SIZE, which determines the granularity of tasks. The final parameter, direction, should be an enum type WaveFrontDirection, which has four values: LeftBottom2RightTop, LeftTop2RightBottom, RightBottom2LeftTop, and RightTop2LeftBottom, corresponding to the four directions.

- `protected void fillCell(P[][] chart, int i, int j)`: The core method of the wavefront algorithms, as it tells the algorithmic skeleton how to compute elements in the chart. Given a chart of type P[][] and row index i and column index j, developers need to define how to compute the value of chart[i][j].
- `protected R genResult(P[][] chart)`: The method used to compute the result. In some wavefront algorithms, the final result is not simply the value of the last element in the chart; it may involve additional calculations. This method specifies how to retrieve the final result from the completed chart after the wavefront algorithm has finished.

After inheriting from, and implementing WaveFrontSkeleton, as well as initializing a thread pool, developers can instantiate such a skeleton in the main function using its constructor. Then, they can call the start method of the skeleton to initialize tasks and initiate the computation process. The method signature of it is `public void start( int waveID)`, where waveID represents the wave from which the wavefront algorithm should start executing. This method divides the specified wave into multiple chunks and submits them to the thread pool. Developers can then use the `getResult()` method to obtain the final result. This method is a blocking operation that only returns the result once the skeleton has completed the wavefront algorithm.

5.4 Implementation

5.4.1 WaveFrontExecutor

WaveFrontExecutor is a singleton pattern that encapsulates ThreadPoolExecutor and controls the thread pool for the algorithmic skeleton. We chose the singleton pattern because developers may have more than one wavefront problem to solve. Using a

singleton pattern avoids additional creation and destruction of thread pools, which can be relatively time-consuming. Furthermore, the singleton pattern ensures a single global access point, allowing other classes or instances to easily access the singleton without passing the thread pool as a parameter. Our singleton pattern is implemented using the double-checked locking approach to ensure safety and high performance in a multi-threaded environment. Developers need to call `newExecutor(threadNum)` to instantiate a ThreadPoolExecutor with a fixed number of threads. Otherwise, it will be instantiated using the number of available core. Currently, WaveFrontExecutor only supports fixed-size thread pools, meaning the number of threads does not change during the execution. In future work, we plan to provide developers with more thread pool options.

5.4.2 WaveFrontSkeleton

`WaveFrontSkeleton` is an abstract class used to define a wavefront algorithm. As mentioned earlier, we provide four abstract classes, and developers need to choose the appropriate one based on their data type. Due to the significant differences among wavefront algorithms, we cannot assume a specific data type for the chart. Therefore, we utilize generics to pass the data type as a parameter to the abstract class. `WaveFrontSkeleton<P, R>` is the most generic abstract class, where P represents the data type of the chart, and R represents the data type of the result. The data type passed through generics can be any class, but not a primitive data type since Java is not supporting this. There are two methods to address this in the skeleton. The first approach is to use wrapper classes in place of primitive types, such as using Integer instead of int and Double instead of double. However, using wrapper classes introduces additional overhead, as wrapper class sizes are larger than primitive data types, and data access time can be impacted due to data locality. The second approach, which we have adopted, involves creating separate abstract classes for primitive data types, fixing the data type of the chart instead of passing it as a parameter. To achieve this, we first extract type-independent member variables and methods to form a separate abstract class, namely `AbstractWaveFrontSkeleton<R>`. Then, we define four new abstract classes inheriting from it. They are the generic abstract class `WaveFrontSkeleton<P, R>`, and the specialized classes `WaveFrontSkeletonShort<R>`, `WaveFrontSkeletonInt<R>`, and `WaveFrontSkeletonDouble<R>`.

The Figure A.1 illustrates their inheritance relationships. Type-independent member

variables and methods are defined in `AbstractWaveFrontSkeleton<R>`, while type-dependent elements are defined in its four subclasses for developer use. In addition to the chart, each abstract class corresponds to a specific type of `WaveFrontTaskFactory`. These four abstract classes also provide two abstract methods, `fillCell` and `genResult`, which are the core of the wavefront algorithm and need to be implemented by developers.

5.4.3 WaveFrontTaskFactory

`WaveFrontTaskFactory` is a factory pattern that includes an inner class Task which implements the `Runnable` interface, allowing it to be submitted as a task to the thread pool. The wavefront algorithm uses the `getNewTask` method provided by `WaveFrontTaskFactory` to create new tasks. By encapsulating the logic and task creation of the `Runnable` interface using the factory pattern, it helps to reduce code coupling, making it easier to maintain and extend. There is a one-to-one correspondence between `WaveFrontSkeleton` and `WaveFrontTaskFactory`, as the `run` method implemented by the factory's Task class invokes the `fillCell` and `genResult` methods implemented by the developer in the skeleton. Similar to `WaveFrontSkeleton`, the task factory is also tightly coupled with the data type, different factory classes need to be created for different data types. The logic of the `Task` that implements the `Runnable` interface is straightforward, as shown in the pseudocode 3. First, it needs to calculate all the cells in the assigned chunk by iteratively calling the `fillCell` method implemented by the developer. Then, it checks if it is the last task. If it is, it calls the `genResult` method to generate the return value and exits. Otherwise, it attempts to create and submit new tasks to the thread pool. The process of submitting tasks is slightly complex, including tracking global execution progress and synchronization among threads. Further details on this process are discussed in 5.4.4.

Algorithm 3 Method run() implemented for the Runnable Interface

```
for each cell in chunk do
    fillCell(cell);
if the last task then
    genResult();
    return;
submitTask();
```

5.4.4 Synchronization

In the algorithmic skeleton for wavefront pattern, the shared data among threads includes the singleton instance of the WaveFrontExecutor and the global 2D array chart. For the WaveFrontExecutor, we have implemented it as a singleton, ensuring only one instance of the thread pool exists. For each thread, this thread pool instance acts as a shared variable whenever a thread is trying to submit tasks to the thread pool. However, the ThreadPoolExecutor in Java uses a thread safe blocking queue to store tasks, which makes its task submission methods submit() also thread-safe. Thus, when threads submit tasks, no additional locking is required to achieve mutual exclusion, and thread synchronization issues are avoided. As for the chart, we can tell from Section 5.1, that even though threads need both read and write on the chart, due to the nature of the wavefront pattern algorithm, two threads won't simultaneously read and write the same part of the chart. For instance, when a thread is calculating elements in wave N, although it needs to read elements from wave N-1, there is no overlap between the elements of wave N and wave N-1. Consequently, data consistency is guaranteed. Therefore, if it is ensured that the algorithm strictly follows the order of waves during computation. Specifically, WaveFrontTaskFactory generates tasks for wave N only when all computations for wave N-1 are completed. As a result, there is no need for locking or other mutual exclusion tools when accessing the chart variable. This ensures both the consistency and correctness of the data.

With the idea mentioned above, a natural synchronization approach was proposed. When allocating tasks for wave N, we first calculate the total number of elements in that wave using its wave index, and determine how many chunks will be based on the number of elements and `CHUNK_SIZE`. Then each chunk is submitted as a task to the thread pool. To track the progress of wave N, we introduce an AtomicInteger as a counter to count how many chunks of that wave have been completed. Whenever a thread finishes computing a chunk, it invokes the incrementAndGet() method on the counter and compares the result with the total number of chunks for that wave. When the counter reaches the chunk number, it indicates that all chunks of that wave have been computed. Then the thread can allocate and submit tasks for wave N+1 to the thread pool. This synchronization method rigorously ensures that wave N always starts computing after wave N-1 has finished. However, this can also significantly impact the performance of the parallel program. This is because new tasks are submitted to the thread pool only after the final task of a wave is completed, causing most threads to

remain idle after finishing their respective tasks.

To address this challenge, we have introduced an new approach for achieving efficient thread synchronization. As discussed in Section 5.2, we not only divide the chart into waves indexed from 0 to N-1 along the diagonals, but we also further partition the elements within each wave into distinct chunks based on the CHUNK_SIZE provided by the developer. The Figure 5.2 illustrates the data dependency relationships between different chunks in adjacent waves when CHUNK_SIZE is set to 2. We can conclude from the figure that, in the left-top to right-bottom wavefront pattern, each chunk doesn't depend on all chunks from the previous wave. For example, chunk k only relies on chunks j and chunk j+1, while chunk k+1 depends on chunks j+1 and chunk j+2. This feature makes the synchronization method we previously proposed redundant. If the chunks that a particular chunk depends on have completed their computations, that specific chunk can be immediately submitted to the thread pool for computation, without waiting for the other chunks of the preceding wave to finish. Consequently, our new approach employs a two-dimensional AtomicInteger array instead of a single AtomicInteger for thread synchronization when submitting tasks.

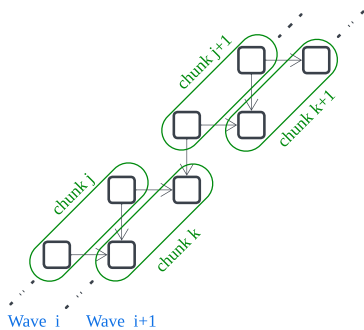

Figure 5.2: Data Dependencies Between Chunks

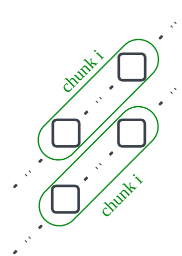

Figure 5.3: Position Case A

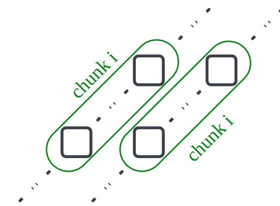

Figure 5.4: Position Case B

Since the size of the chart is specified by the developer, once we know the values of ROW_SIZE and COL_SIZE, we can calculate the total number of waves in the wavefront problem and the number of elements in each wave. We can further determine the maximum number of chunks in a single wave. Let N represent the maximum number of waves, and M represent the maximum number of chunks. We then allocate a two-dimensional AtomicInteger array of size N*M, which we'll name condition. We use the condition array to track whether the chunks that a particular chunk depends on have completed their computations. For instance, if the chunk j of the wave i, referred to as chunk A, has a data dependency on chunk B, when the task corresponding to

chunk B is completed by a thread, the thread call the incrementAndGet() method on condition[i][j] and checks whether its return value equals the number of chunks that chunk A depends on. If it is true, then chunk A can be submitted as a new task to the thread pool. By adopting this synchronization approach, our wavefront skeleton accelerates the submission of chunks to the thread pool as soon as their dependency conditions are met. This approach reduces thread idle time and ultimately enhances the performance of parallel programs.

To implement the synchronization method described above, we also need to understand the dependency relationships between different chunks in adjacent waves. For ease of calculation, we establish the convention that wave indices always increase in the direction of wavefront propagation, and chunk indices increase as rows move closer to the top of the chart. With this convention, we can deduce the relative positions of chunks with the same index in adjacent waves. Their relative positions fall into two cases, as shown in the Figure 5.3 and Figure 5.4. Furthermore, the absolute difference in the number of elements between adjacent waves can only be 0 or 1. This implies that the number of chunks between adjacent waves can only fall in three cases: 1) the number of chunks in the later wave is one less than that in the preceding wave; 2) the number of chunks in the later wave is one more than that in the preceding wave; 3) the number of chunks is the same in both waves.

For the first case, we have observed that regardless of the direction of the wavefront pattern or the size of the chart, the data dependency relationships between chunks in adjacent waves remain consistent, as depicted in the Figure 5.5. As a result, for a chunk j in wave i, the thread needs to check whether condition[i+1][j].incrementAndGet() and condition[i+1][j-1].incrementAndGet() is equals to 2 (excluding chunk 0 and chunk m in wave i, which only need to check one condition each).

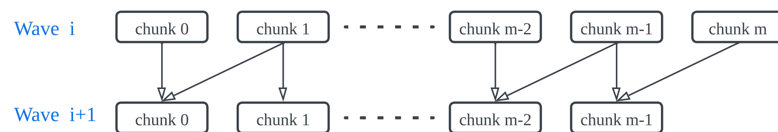

Figure 5.5: Data Dependencies Between Chunks

For the second case, we have also observed that similar to the first case, the data dependency relationships between chunks in adjacent waves are consistent, as shown in the Figure 5.6. Therefore, for a chunk j in wave i, the thread needs to check whether

condition[i+1][j].incrementAndGet() and condition[i+1][j+1].incrementAndGet() is equals to 2 (excluding chunk 0 and chunk m in wave i+1, which only depend on one chunk each).

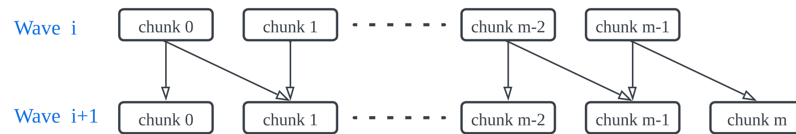

Figure 5.6: Data Dependencies Between Chunks

The third case is more complex. We have found that the dependency relationships between chunks are influenced by both the direction of the wavefront pattern and their positions within the chart. In one algorithm, chunk j might depend on chunk j and chunk j+1 from the preceding wave. In another algorithm, chunk j might depend on chunk j-1 and chunk j from the preceding wave. While it is feasible to calculate the precise dependencies, performing such checking for each thread every time can impact the performance of the algorithm skeleton. We have therefore simplified the case with the following assumption: chunk j in wave i+1 depends on chunk j-1, chunk j, and chunk j+1 from wave i. As a result, for chunk j in wave i, the thread needs to check whether condition[i+1][j-1].incrementAndGet(), condition[i+1][j].incrementAndGet(), and condition[i+1][j+1].incrementAndGet() is equals to 3 (excluding chunk 0 and chunk m in wave i, which only need to check one condition each; excluding chunk 0 and chunk m in wave i+1, which only depend on two chunks each). Once the conditions are satisfied, the chunks corresponded can be submitted to the thread pool.

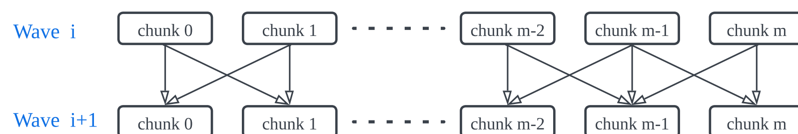

Figure 5.7: Data Dependencies Between Chunks

5.5 Examples

Example code for the CYK and LCS algorithms can be found in A.1 and A.2.

Chapter 6

Evaluation

6.1 Evaluation Setup

All the experiments are performed on a Windows machine with a 2.9 GHz 8-Core AMD CPU, and 2x8GB RAM at 3200 MHz. All experiments are run under the 64bit Java Runtime Environment of version 17.0.7+8. The JVM is specified with the -Xmx10g option, configuring the maximum heap size to 10G.

For both D&C and B&B pattern, there are three different implementations for each: sequential, thread pool, and hand-threaded implementation. We conducted performance tests on these implementations with thread number set to 2, 4, 6, and 8. For the two algorithms in the wavefront pattern (Longest Common Subsequence and CYK algorithm), apart from the three implementations mentioned above, there is also a parallel skeleton implementation. Furthermore, the parallel implementation of the wavefront pattern algorithms allows for specifying different CHUNK_SIZE values to set the granularity of thread tasks, as described in section 5.2. Hence, in our performance tests, we also explore the impact of CHUNK_SIZE on different algorithms. For the LCS algorithm, the computational cost of each cell is minimal, so theoretically CHUNK_SIZE should not be set too small to avoid frequently synchronizing. Therefore, for each thread count, we tested the performance of the LCS algorithm with CHUNK_SIZE set to 32, 64, 128, 256, 512, 1024 and 2048. On the other hand, the computational cost of each cell in the CYK algorithm increases with the size of the wave, so theoretically, CHUNK_SIZE should not be set too large to avoid thread idle. Therefore, for each thread number, we tested the performance of the CYK algorithm with CHUNK_SIZE set to 1, 2, 4, 8, 16, 32 and 64. Finally, we evaluate the programmability of each implementation to analyse its suitability to each pattern.

6.2 Evaluation Metrics

We consider both performance and programmability as indicators of suitability. The speedup is computed from the execution time, but is more intuitive, so we show the speedup more often in our results and analysis. The higher the speedup, the better the performances of the parallel program. The three programmability metrics represent the complexity of the program. The higher the complexity, the worse the programmability, meaning that the program requires more programming effort.

6.2.1 Performance

- **Execution time:** The System.currentTimeMillis() method in Java will be employed to record the timestamp in milliseconds. We call this method at the beginning and the end of each program, and calculate the difference as the execution time. The execution time exclude the time spent in the preparation phases, such as reading data from files or initializing variables. However, any additional overhead incurred by parallel programs is included, such as initializing locks, condition variables, and other synchronization tools, as well as creating thread. To obtain a more precise result, we execute each program 10 times and derive the accepted time by calculating the average execution time:

$$\bar{t} = (1/10) * \sum_{i=1}^{10} t_i$$

- **Speedup:** The speedup refers to the ratio between the execution time of a sequential program and that of a multi-threaded program. Let T_s and T_n represents the execution time of the sequential version, and parallel version with n threads. The speedup can be obtained as:

$$Speedup_n = T_s/T_n$$

6.2.2 Programmability

- **Source Lines of Code (SLOC):** SLOC is the total number of lines of code in the source code after removing comments and blank lines. It represents an estimate of the effort required to complete the program. Since all implementations are done by a single developer, SLOC can still accurately reflect the difference in programmability of different implementations. We use SonarQube [29] to

complete the calculation of SLOC. SonarQube is an open source code quality analysis software, we can find the SLOC metric in the size-of-code section of the static testing provided by it.

- **Cyclomatic Complexity:** cyclomatic complexity is a measurement used to indicate the complexity of a program proposed by McCabe[30]. Since our program has only one entry point and one exit point. Let P be the number of decision points, such as if statements or conditional loops. The cyclomatic complexity can be calculated as below. We can find the cyclomatic complexity metric in the complexity section of the static testing provided by SonarQube [29].

$$V = P + 1$$

- **Halstead Effort:** Halstead effort[31] is one of the measurements developed by Maurice Halstead to measure the complexity of a program. It is more accurate than the SLOC since it takes operands and operators into account. The formula for Halstead Effort is as follows:

$$E = D * V$$

Where D refers to difficulty and V refers to Volume, calculated as:

$$D = (n_1/2) * (N_2/n_2)$$

$$V = (N_1 + N_2) * \log_2(n_1 + n_2)$$

Where n_1 and N_1 represent the count of unique operators and all operators. n_2 and N_2 represent the count of unique operands all operands. We use an open source tool called Java-Code-Analyzer to generate the Halstead effort.

6.3 Results

6.3.1 Divide and Conquer

For the quadrature algorithm, we can observe that the thread pool implantation's speedup, as a function of the number of thread, nearly follows linear growth and closely approaches the ideal speedup, as shown in the Figure 6.1. On the other hand, the hand-threaded version of the quadrature algorithm shows decent performance and an increase speedup as the number of threads increases. It reaches its peak at four threads,

followed by a slight decline in speedup. This results match our expectations since our hand-threaded version use the Bag-of-tasks strategy and manually maintains a shared task queue. Every time a thread attempts to retrieve or submit a task, it needs to acquire a lock. Theoretically, the algorithm generates new tasks frequently in the initial stages of execution. With an increasing number of threads, the frequent access to the shared task queue results in more time being consumed in meaningless busy waiting, eventually leading to a decrease in performance. In contrast, the thread pool version use the ForkJoinPool, which is well-suited for the D&C pattern. It employs the work-stealing scheduling approach. As a result, each thread has its own task queue, and when one thread's queue is empty, it can steal tasks from other threads. This scheduling approach, compared to the hand-threaded version's single task queue, provides significant advantages by minimizing the difficulty of inter-thread synchronization. This results in a speedup that is very close to the ideal speedup.

For the mergesort algorithm, we can observe that the trends in speedup for both the hand-threaded and thread pool versions are very similar, as shown in the Figure 6.1. They both increase as the number of threads increases, but the rate of increase becomes very slow when the thread number goes beyond 4. This leads to a growing gap between their speedup and the ideal speedup. Additionally, the performance of the hand-threaded version is slightly better than that of the thread pool version across all thread settings. This result is due to the nature of mergesort. During the splitting phase, the parallel algorithm divides a large array into smaller arrays and assigns them to different threads for processing, which increase parallelism. However, during the merging phase, each merge task is completed within a single thread, and the merging involves in accessing memory, which is relatively slow. Therefore, while lower-level merging can be parallelized, the overall parallelism decreases as the merging goes on. This eventually results in an insignificant increase in speedup with an increasing number of threads. The fact that the hand-threaded version outperforms the thread pool version is also meets our expectations. The hand-threaded version directly calls the sequential mergesort when the available threads are insufficient, whereas the thread pool version continues the splitting until the array length reaches 1, as explained in 4.1. The granularity of tasks in the thread pool version is finer, leading to frequently task allocation and thread scheduling overhead.

Table 6.1 presents the percentage increase in programmability metrics of the parallel implementations of the two D&C algorithms over their sequential counterparts. We can observe that all three metrics show a smaller increase in complexity and programming

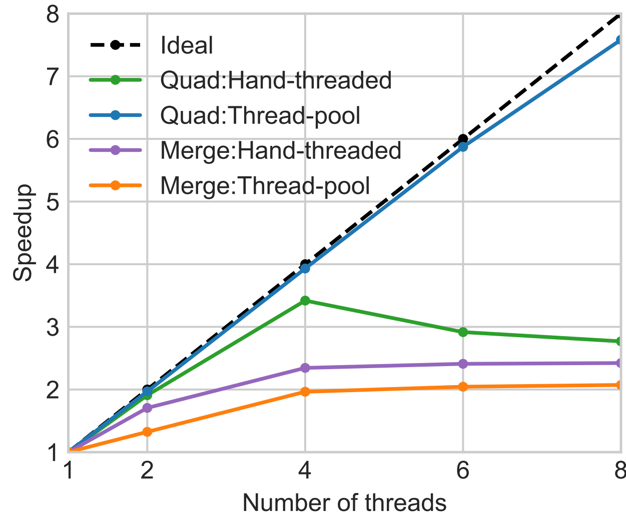

Figure 6.1: Speedup for D&C Algorithms

Table 6.1: Programmability Increase over Sequential of D&C Algorithms

Algorithm	Implementation	SLOC	Halstead Effort	Cyclomatic Complexity
Mergesort	Thread pool	23%	61%	9%
	Hand-threaded	81%	174%	36%
Quadrature	Thread pool	140%	290%	33%
	Hand-threaded	507%	2233%	200%

effort for the thread pool versions, indicating that using a thread pool is easier and requires less effort to achieve parallelism. For the mergesort algorithm, the thread pool version only needs to modify the recursion to submit tasks to the thread pool based on the sequential version. The hand-threaded version, on the other hand, requires additional programming effort to determine the availability of threads and to call the sequential mergesort algorithm when necessary, which are the sources of extra complexity. As for the quadrature algorithm, the hand-threaded version exhibits a significant increase in complexity. For example, the Halstead effort metric increases by 2233% compared to the sequential version. This is mainly due to the adoption of the Bag-of-tasks strategy, which adds additional complexity to task retrieval and submission in the program.

Considering both performance and programmability, we conclude that ForkJoinPool is well-suited for implementing algorithms of D&C pattern. The methods it provides align with the logic of Divide-and-Conquer algorithms, making it relatively straightforward to implement parallel versions of algorithms based on their serial counterparts. Additionally, for D&C algorithms with good parallelism, such as the quadrature algo-

rithm, using ForkJoinPool makes it easier to achieve better performance. For D&C algorithms with poor parallelism, like mergesort, although the thread pool abstraction offers better programmability, better performance may not be guaranteed. Developers need to take issues such as task granularity into consideration to achieve better speedup.

6.3.2 Branch and Bound

For the Knapsack algorithm, we did not draw the speedup lines in the Figure 6.2. This is because both parallel implementations of the Knapsack algorithm exceeded the runtime limit of 500 seconds for all thread numbers, resulting in no execution time and speedup data.That is to say, both hand-threaded and thread pool implementations resulted in negative optimization, with performance even worse than the sequential one. However, this outcome is not unexpected. As mentioned in section 4.5, the Branch-and-Bound pattern utilizes pruning techniques to reduce the search space. When the current node and its subtree are not possible to produce a better solution than the current global optimum, they will be disregarded. In the context of the Knapsack problem, the global optimum can only be updated when the search reaches a leaf node in the tree. The sequential algorithm employs depth-first searching, enabling the rapid updating of the global optimum. This allows it to quickly eliminate a large number of nodes and their subtrees, significantly reducing the runtime. In contrast, in parallel algorithms, tasks are submitted at the node level. By the time the program reaches the first leaf node, there are already numerous tasks waiting for execution, many of which could have potentially been disregarded. Although we attempted to enforce depth-first search by using a LIFO task queue in the hand-threaded implementation, this becomes unavoidable in a multi-threaded environment. Additionally, the depth of the search tree for the Knapsack algorithm is equal to the number of the items, which implies that the tree depth for large-scale Knapsack problems can be extremely large, making it less favorable for parallel program execution. These issues prevent our parallel implementations from effectively leveraging the benefits of the Branch-and-Bound approach, causing the algorithm to degrade into a brute-force. As a result, the runtime exceeds the given limit.

As for the Traveling Salesman Problem, the situation is similar to the Knapsack problem. From the Figure 6.2, we can observe that both the thread pool implementation and the hand-threaded implementation have speedup less than 1, indicating that their performance is worse than the sequential version. Additionally, the speedup of the two parallel versions show little variation as the thread number increases, with the

thread pool version outperforming the hand-threaded version. The reason for this phenomenon is similar to the Knapsack algorithm. In both cases, the parallel algorithms prematurely submit many tasks before the global optimum is updated, leading to a significant amount of meaningless computation. The better performance of the thread pool implementation is due to our use of the invoke method of ForkJoinPool instead of the submit method. The invoke method is a blocking method, which means that tasks won't continue executing until their subtasks return. This makes the execution order of the parallel program more similar to depth-first search. In contrast, in the hand-threaded implementation, even though we use a LIFO queue, we cannot predict the execution order of tasks. This uncertainty in execution order contributes to the lower performance of the hand-threaded version compared to the thread pool implementation.

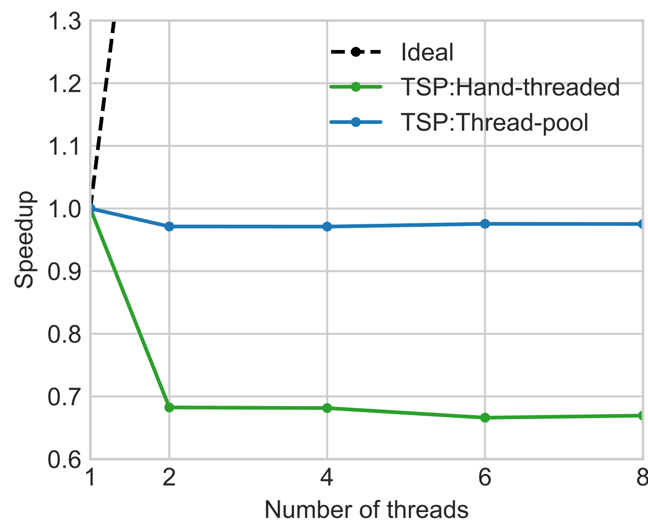

Figure 6.2: Speedup for B&B Algorithms

Table 6.2 presents the percentage increase in programmability metrics of the parallel implementations of the two B&B algorithms over their sequential counterparts. We can observe that all three metrics show a smaller increase in complexity for the thread pool version. In B&B algorithms, the nodes can be easily programmed into tasks in a parallel implementations. Therefore, both the thread pool and hand-threaded implementations require relatively little extra effort, resulting in increases of most of the programmability metrics by less than 100%. The additional thread synchronization operations, required for the hand-threaded version, are the source of the extra programming effort compared to thread pool version.

However, despite the fact that it is relatively easy to implement parallel B&B

Table 6.2: Programmability Increase over Sequential of B&B Algorithms

Algorithm	Implementation	SLOC	Halstead Effort	Cyclomatic Complexity
Knapsack	Thread pool	21%	39%	0%
	Hand-threaded	77%	137%	57%
TSP	Thread pool	26%	37%	8%
	Hand-threaded	54%	81%	25%

algorithms, their poor performance makes both the hand-threaded implementation with a Bag-of-tasks strategy and the thread pool implementation unsuitable for implementing parallel algorithms that belongs to the B&B pattern. B&B algorithms take advantages of depth-first search, while thread pools and Bag-of-tasks strategies are more likely to be breadth-first search. This mismatch results in the pruning techniques of branch-and-bound not being effective in a multi-threaded environment. Developers need to explore more efficient methods to parallelize B&B algorithms.

6.3.3 Wavefront

Due to the Wavefront algorithms' implementations offering an additional parameter CHUNK_SIZE, we aim to explore how CHUNK_SIZE affects the performance and whether there exists an best CHUNK_SIZE that often gives the best performance. We make the number of threads fixed and change the CHUNK_SIZE , then calculate the speedup of the CYK algorithm. Figures A.2 to A.5 correspond to the speedup for thread number of 2, 4, 6, and 8, respectively. We can observe that the speedup of all three implementations follow a pattern of increasing followed by decreasing. For both the thread pool and the skeleton implementations, the highest speedup is achieved when CHUNK_SIZE is set to 4 under all thread number settings. As CHUNK_SIZE further increases, their speedup drop sharply. For thread number of 2, 4, and 6, the speedup of hand-threaded version remains relatively stable when CHUNK_SIZE is small. However, when CHUNK_SIZE surpasses 4, the speedup also declines rapidly. When the thread count is 8, the hand-threaded version achieves the highest speedup with CHUNK_SIZE set to 1, which then decreases as CHUNK_SIZE increases. Although in very few instances other configurations might surpass the speedup achieved with CHUNK_SIZE set to 4, we still consider 4 to be the best CHUNK_SIZE setting for the CYK algorithm. This conclusion aligns with our theoretical hypothesis, which suggests

that the CYK algorithm performs better with smaller CHUNK_SIZE values. This is because the tasks of the CYK algorithm are not load balanced, and larger waves result in longer cell calculation times. Consequently, if CHUNK_SIZE is too large, threads are frequently idle, waiting for other tasks to complete, eventually leading to performance degradation.

Furthermore, we generated figure depicting how the speedup changes with varying thread counts with the best CHUNK_SIZE setting. From Figure 6.3, we can see that the speedup of the three implementations are very close. Among them, the hand-threaded version achieves the highest performance when the thread number is 6 or below, while the skeleton version excels at a thread number of 8. We believe this is due to the similar synchronization mechanisms employed by all three implementations. The difference is that, the thread pool and hand-threaded versions track cell execution, while the skeleton tracks chunk execution. This distinction becomes more evident as the thread count increases, where the speedup of the skeleton implementation rises faster than the other two as the thread count increases from 6 to 8. Moreover, we also observed that the speedup growth rate of all three implementations decreases as the thread count increases. This is because with an increase in the number of threads, more time is spent on thread synchronization. Additionally, due to the nature of the wavefront pattern, which typically ends at one corner of a two-dimensional array, the final stages of the algorithm lack substantial parallelism.

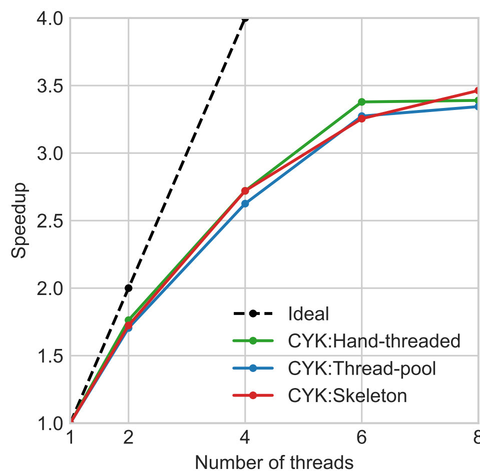

Figure 6.3: Speedup for CYK Algorithm (CHUNK_SIZE = 4)

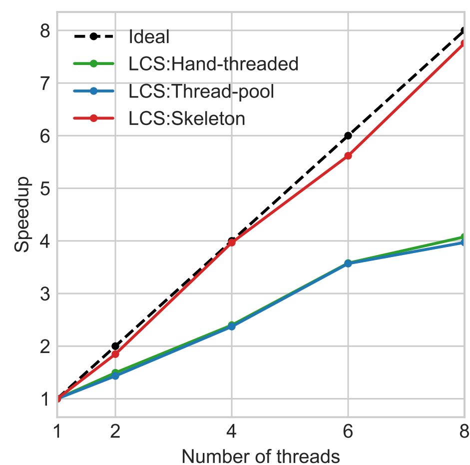

Figure 6.4: Speedup for LCS Algorithm (CHUNK_SIZE = 1024)

For the LCS algorithm, we also fix the number of threads and explore the effect

of different CHUNK_SIZE on the performance. Figures A.6 to A.9 correspond to the speedup of the three implementations when the number of threads is 2, 4, 6, and 8, respectively. We can observe that the speedup increases and then decreases in all cases except for the skeleton implementation with thread number set to 2 where the speedup increases monotonically. The speedup for both the hand-threaded and thread pool implementations typically peak at a CHUNK_SIZE of 512 and then slowly decrease as CHUNK_SIZE continues to increase. The skeleton version, on the other hand, often reaches its best performance at a CHUNK_SIZE of 1024. Since the hand-threaded and thread pool implementations have very little difference in speedup ratios at CHUNK_SIZE of 512 and 1024, the skeleton version far outperforms the other two versions. We consider that 1024 is the best CHUNK_SIZE configuration for the LCS algorithm. This conclusion aligns with our theoretical hypothesis, which suggests that the LCS algorithm performs better with larger CHUNK_SIZE values. Otherwise threads will spend more time synchronizing.

The speedup relative to the number of threads for the best CHUNK_SIZE setting is shown in Figure 6.4. The performance of hand-threaded and thread pool version are close. The skeleton implementation, on the other hand, has a speedup that very close to the ideal one, far exceeding that of hand-threaded and thread pool versions. We believe this is due to the difference in synchronization strategies. The skeleton version tracks the execution of chunks and commits tasks as soon as a chunk is ready for computation. The hand-threaded and thread pool track the execution of waves, and the task for the next wave is submitted only when all cells of a wave have been computed, which leads to thread idling. However, the skeleton implementation's speedup is way too close to the ideal speedup which is beyond expectation. The synchronization method in the skeleton is quite complicated and time-consuming, and its upward trend of speedup should slow down as the number of threads increases. We believe this is due to data locality [32]. As the number of cores increases, the total cache size increases, which allows more data in cache at the same time. It reduces the time spent on accessing memory. Sequential program cannot take advantage of this, which results in extra speedups for parallel programs. So theoretically if under certain hardware settings, we may even observe the super-linear speedup, that is, the speed of N threads exceeds N.

Table 6.3 presents the percentage increase in programmability metrics of the parallel implementations of the two Wavefront algorithms over their sequential counterparts. Among all implementations, the skeleton implementation provides the smallest increase in complexity, and the cyclomatic complexity even decreased compared to the sequential

one. This is because our Wavefront algorithmic skeleton abstracts the wavefront problem at a higher level, eliminating the need for developers to explicitly loop through the chart. Additionally, the increase in SLOC and Halstead Effort metrics for the skeleton implementation is minimal, indicating that it offers the best programmability by requiring the least programming effort. In contrast, both the thread pool and hand-threaded implementations of two algorithms exhibit higher complexity and programming effort, which comes from the Bag-of-tasks strategy and thread synchronization.

Table 6.3: Programmability Increase over Sequential of Wavefront Algorithms

Algorithm	Implementation	SLOC	Halstead Effort	Cyclomatic Complexity
CYK	Thread pool	100%	235%	75%
	Hand-threaded	164%	397%	115%
	Skeleton	10%	42%	-5%
LCS	Thread pool	115%	223%	30%
	Hand-threaded	241%	558%	90%
	Skeleton	15%	16%	-30%

We can conclude that the wavefront algorithm skeleton demonstrates the best programmability among the three different parallel implementations. It requires minimal programming effort to implement parallel wavefront algorithms based on their sequential counterparts. Furthermore, if a properly configured CHUNK_SIZE is employed for the wavefront algorithm, the skeleton implementation can also yield the best or near-best performance. Although hand-threaded implementations occasionally offer slightly better performance than the skeleton version, the difference is marginal. In addition, hand-threaded version exhibits the poorest programmability, indicating that developers might have to invest several times more programming effort for negligible performance gains.

In summary, the parallel algorithmic skeleton we provide outperforms both hand-threaded and thread pool implementations in terms of performance and programmability, making it very suitable for the wavefront pattern. We found that the trend of the speedup with CHUNK_SIZE for each implementation is very close. Developers can thus apply the algorithm to smaller-scale problems to identify the best CHUNK_SIZE for maximum performance, which can then be used as the general CHUNK_SIZE for the algorithm. Moreover, we do not find that the performance decreases with the increase of the number of threads, so we believe that as long as the number of threads does not exceed the number of available cores, a higher number of threads implies better performance.

Chapter 7

Conclusions and Future Work

This project addresses a gap in research by investigating the performance and programmability of different implementations of specific parallel pattern. We selected three common parallel patterns for evaluation: Divide-and-Conquer, Branch-and-Bound, and Wavefront. For the D&C pattern, we chose the mergesort and quadrature algorithms. For B&B pattern, we selected the 0/1 Knapsack problem and the Traveling Salesman Problem. For the Wavefront pattern, we selected the CYK algorithm and the Longest Common Subsequence algorithm. For each algorithm, we implemented sequential, hand-threaded, and thread pool versions. Additionally, we designed and developed a parallel algorithmic skeleton for the Wavefront pattern. This skeleton hides all the threading and synchronization details, allowing developers to focus solely on the core logic of the Wavefront algorithm. The algorithmic skeleton supports wavefront pattern and their variants. The developer can choose the direction of the wavefront as well as the location where the wavefront algorithm begins, making our skeleton very generalizable. Therefore, for the CYK algorithm and the LCS algorithm, we also implemented skeleton versions with the algorithmic skeleton.

After conducting a thorough evaluation of performance and programmability, we have arrived at our conclusions. The thread pool implementation using ForkJoinPool is exceptionally well-suited for implementing the D&C pattern. ForkJoinPool's design aligns seamlessly with the principles of Divide-and-Conquer, requiring minimal additional programming effort for parallelization. For algorithms with good parallelism, using ForkJoinPool makes it easier to achieve better performance. However, it may not necessarily yield better performance for algorithms with poor parallelism. Regarding the B&B pattern, even though both hand-threaded and thread pool implementations exhibit good programmability, their performance is not satisfying. Both of our parallel

implementations act like breadth-first search, while Branch-and-Bound requires timely exclusion of unnecessary nodes, which is more effectively achieved with depth-first search. As a result, we conclude that neither of these implementations is suitable for the Branch and Bound algorithm, necessitating the exploration of alternative parallelization strategies. In the case of the Wavefront pattern, our findings indicate that the Wavefront algorithmic skeleton outperforms both hand-threaded and thread pool implementations in terms of both performance and programmability. This implies that the algorithmic skeleton is well suited for solving Wavefront problems. This success underscores the effective design and implementation of an efficient Wavefront algorithmic skeleton. Moreover, the skeleton significantly reduces the programming complexity for developers, who can easily implement parallelized wavefront algorithms even without knowing anything about parallel computing.

There are several potential research directions for future work. This project focused on investigating the suitability of different Java implementation methods to specific parallel patterns in terms of performance and programmability. Firstly, this project evaluated only three parallel patterns: Divide-and-Conquer, Branch-and-Bound, and Wavefront. To provide a more comprehensive assessment, future work could investigate other common parallel patterns, such as the All-pairs pattern. Additionally, for each parallel pattern, we only selected two algorithms to implement. To enhance the generalizability of our conclusions, it would be beneficial to increase the number of algorithms implemented for each pattern. Moreover, these algorithms should cover different variants within the same pattern make the conclusions more persuasive.

For the algorithmic skeleton we developed for the Wavefront pattern, there's room for further improvement. Currently, the CHUNK_SIZE in the skeleton is a constant and specified by the developer. In future work, we can offer more CHUNK_SIZE options to accommodate different algorithms. For instance, we can provide an option for variable CHUNK_SIZE during algorithm execution. Developers could provide both a minimum and a maximum value for CHUNK_SIZE. When the Wavefront pattern starts, CHUNK_SIZE is set to the minimum value, allowing more tasks to enter the thread pool to prevent thread idleness. As the Wavefront pattern goes on, CHUNK_SIZE gradually increases until reaching the maximum value. Towards the end of the Wavefront pattern, CHUNK_SIZE decreases gradually back to the minimum value to avoid cases with fewer tasks. In addition to variable CHUNK_SIZE, future work could involve providing an adaptive CHUNK_SIZE option, allowing the skeleton to make decisions about CHUNK_SIZE based on the algorithm behavior.

Bibliography

[1] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., USA, 1989.

[2] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[3] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[4] Collin Jefferson. Bibliography. 5. av aho, je hopcroft and jd ullman, the design and analysis oe computer algorithms, addison-wesley, 1974.

[5] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[6] Christopher Moretti, Jared Bulosan, Douglas Thain, and Patrick J Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *2008 IEEE international symposium on parallel and distributed processing*, pages 1–11. IEEE, 2008.

[7] Naraig Manjikian and Tarek S Abdelrahman. Scheduling of wavefront parallelism on scalable shared-memory multiprocessors. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, volume 3, pages 122–131. IEEE, 1996.

[8] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[9] TIOBE Index - TIOBE — tiobe.com. `https://www.tiobe.com/tiobe-index/java/`, 2023.

[10] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[11] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[12] Dick Buttlar, Jacqueline Farrell, and Bradford Nichols. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[13] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[14] Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[15] Kevin Hammond and Greg Michelson. Research directions in parallel functional programming. 2000.

[16] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[17] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. A divide-and-conquer parallel pattern implementation for multicores. In *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*, pages 10–19, 2016.

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

[19] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: High-level and efficient streaming on multicore. *Programming multi-core and many-core computing systems*, pages 261–280, 2017.

[20] Millán A Martínez, Basilio B Fraguela, and José C Cabaleiro. A parallel skeleton for divide-and-conquer unbalanced and deep problems. *International Journal of Parallel Programming*, 49(6):820–845, 2021.

[21] 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. IEEE, 2010.

[22] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Jean-Thierry Lapresté. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[23] Philipp Ciechanowicz and Herbert Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 108–113. IEEE, 2010.

[24] Marco Danelutto and Paolo Teti. Lithium: A structured parallel programming environment inja va. In *Computational Science—ICCS 2002: International Conference Amsterdam, The Netherlands, April 21–24, 2002 Proceedings, Part II 2*, pages 844–853. Springer, 2002.

[25] Mario Leyton and José M Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 289–296. IEEE, 2010.

[26] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[27] Peter J Kolesar. A branch and bound algorithm for the knapsack problem. *Management science*, 13(9):723–735, 1967.

[28] Egon Balas and Paolo Toth. *Branch and bound methods for the traveling salesman problem*. Carnegie-Mellon University, Design Research Center, 1983.

[29] Code Quality Tool Secure Analysis with SonarQube — sonarsource.com. `https://www.sonarsource.com/products/sonarqube/`. [Accessed 08-2023].

[30] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

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

[32] Yuanzhe Li and Loren Schwiebert. Memory-optimized wavefront parallelism on gpus. *International Journal of Parallel Programming*, 48:1008–1031, 2020.

Appendix A

Supplementary Materials

A.1 Skeleton Implementation of the CYK Algorithm

```
class CYKWaveFrontSkeleton extends WaveFrontSkeleton
    <Map<Integer, Long>,Long>{
        public CYKWaveFrontSkeleton(Map<Integer, Long>[][] chart,
        int CHART_ROW_SIZE, int CHART_COL_SIZE,
        int ROW_MIN, int ROW_MAX,
        int COL_MIN, int COL_MAX,
        int CHUNK_SIZE, WaveFrontDirection direction) {
            super(chart, CHART_ROW_SIZE, CHART_COL_SIZE,
            ROW_MIN, ROW_MAX, COL_MIN, COL_MAX,
            CHUNK_SIZE, direction);
        }

        @Override
        protected void fillCell(Map<Integer, Long>[][] chart,
        int i, int j) {
            Map<Integer, Long> cell = chart[i][j];
            for(int k=i; k<j; k++){
                Map<Integer, Long> cell1 = chart[i][k];
                Map<Integer, Long> cell2 = chart[k+1][j];
                for(Map.Entry<Integer, Long> entry1: cell1.entrySet()){
                    for(Map.Entry<Integer, Long> entry2: cell2.entrySet()){
                        addToCell(cell, entry1, entry2);
```

```
                }
            }
        }
    }

    void addToCell(Map<Integer, Long> cell,
    Map.Entry<Integer, Long> entry1, Map.Entry<Integer, Long> entry2){
        List<Integer> tmpList = new ArrayList<>();
        tmpList.add(entry1.getKey());
        tmpList.add(entry2.getKey());
        if(nonTerminalRule.containsKey(tmpList)){
            Set<Integer> nonTerminalSet = nonTerminalRule.get(tmpList);
            for(Integer nonT: nonTerminalSet){
                long num = entry1.getValue() * entry2.getValue();
                cell.compute(nonT, (key, value) -> (value == null) ?
                num : value + num);
            }
        }
    }

    @Override
    protected Long genResult(Map<Integer, Long>[][] chart) {
        return chart[0][strLength-1].get(0);
    }
}
}
```

A.2 Skeleton Implementation of the LCS Algorithm

```
class LCSWaveFrontSkeleton extends
    WaveFrontSkeletonShort<Integer> {
        public LCSWaveFrontSkeleton(short[][] chart,
        int CHART_ROW_SIZE, int CHART_COL_SIZE,
        int ROW_MIN, int ROW_MAX, int COL_MIN,
        int COL_MAX, int CHUNK_SIZE, WaveFrontDirection direction) {
            super(chart, CHART_ROW_SIZE, CHART_COL_SIZE,
            ROW_MIN, ROW_MAX, COL_MIN, COL_MAX,
            CHUNK_SIZE, direction);
        }

        @Override
        protected void fillCell(short[][] chart, int i, int j){
            if(text1.charAt(i-1) == text2.charAt(j-1)){
                chart[i][j] = (short) (1 + chart[i-1][j-1]);
            }
            else {
                chart[i][j] = (short) Math.max(chart[i-1][j], chart[i][j-1]);
            }
        }

        @Override
        protected Integer genResult(short[][] chart){
            return (int) chart[text1Length][text2Length];
        }
    }
```

A.3 Class Diagram of WaveFrontSkeleton

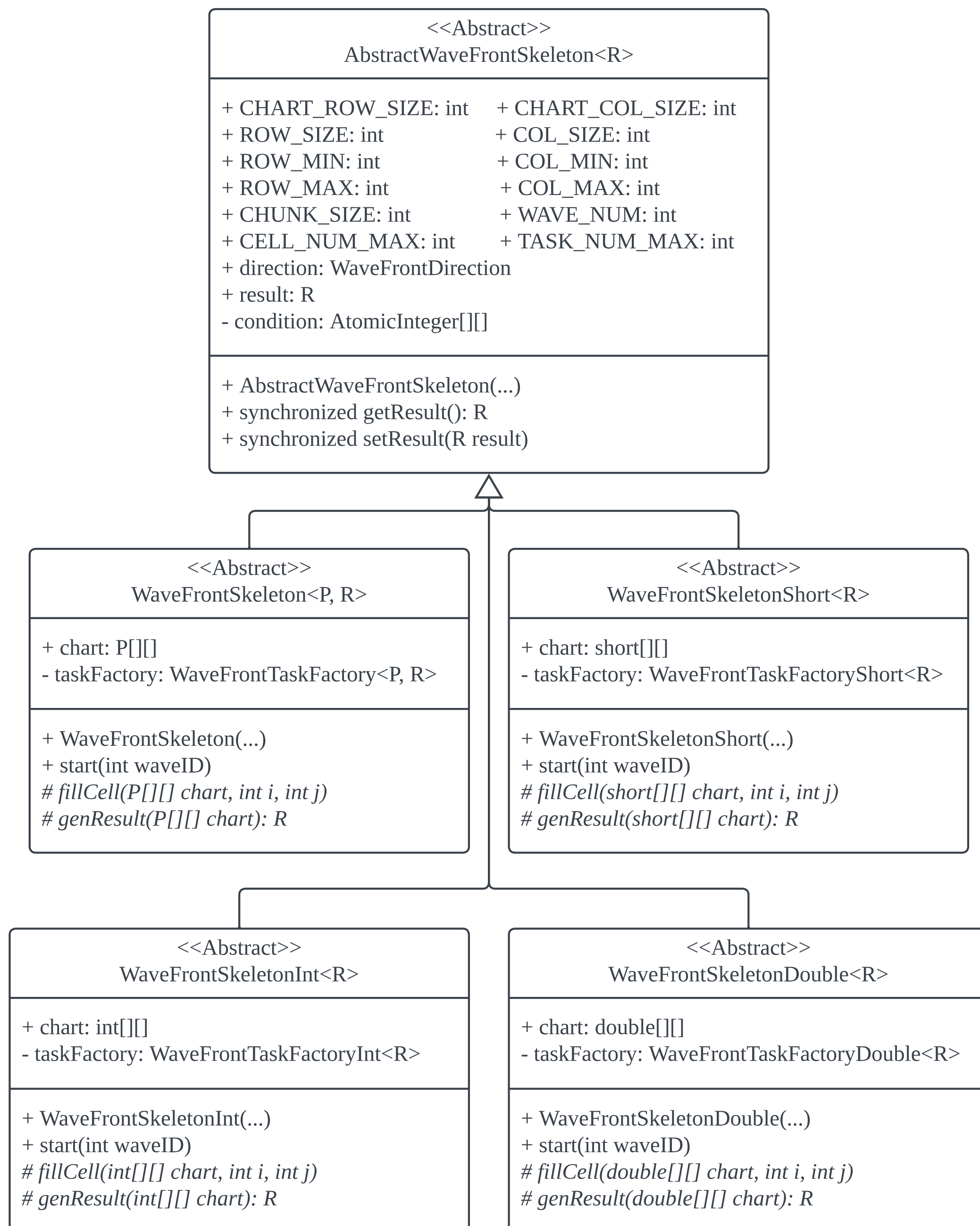

Figure A.1: Class Diagram of WaveFrontSkeleton

A.4 Speedup for CYK Algorithm

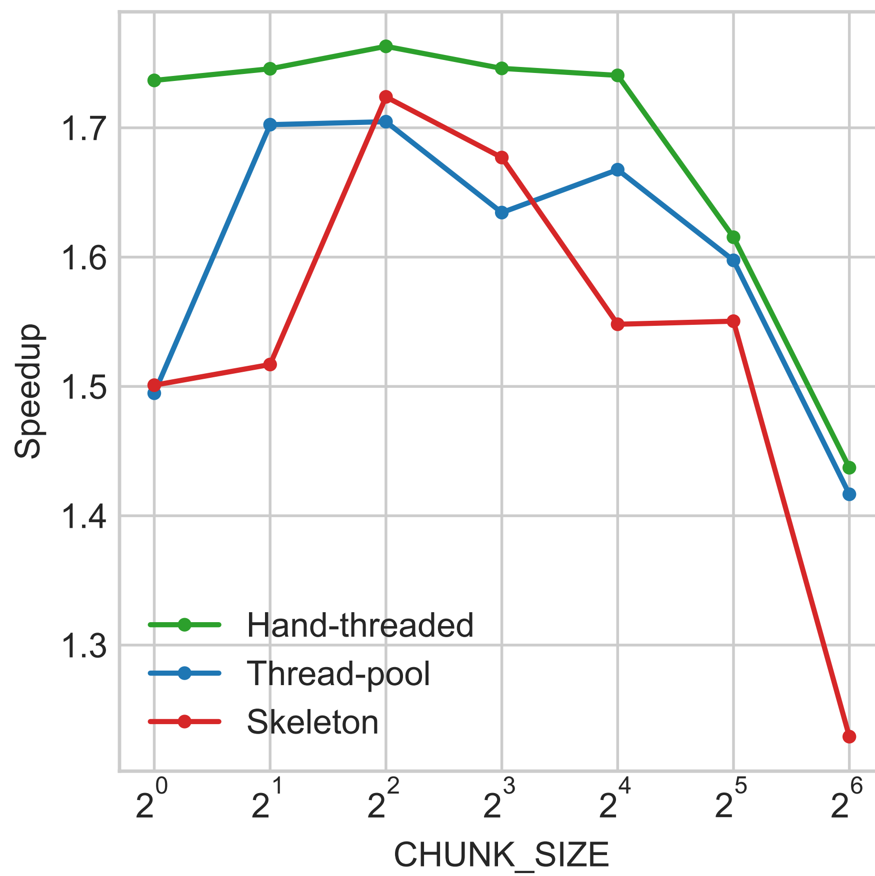

Figure A.2: Speedup for CYK Algorithm when number of threads is 2

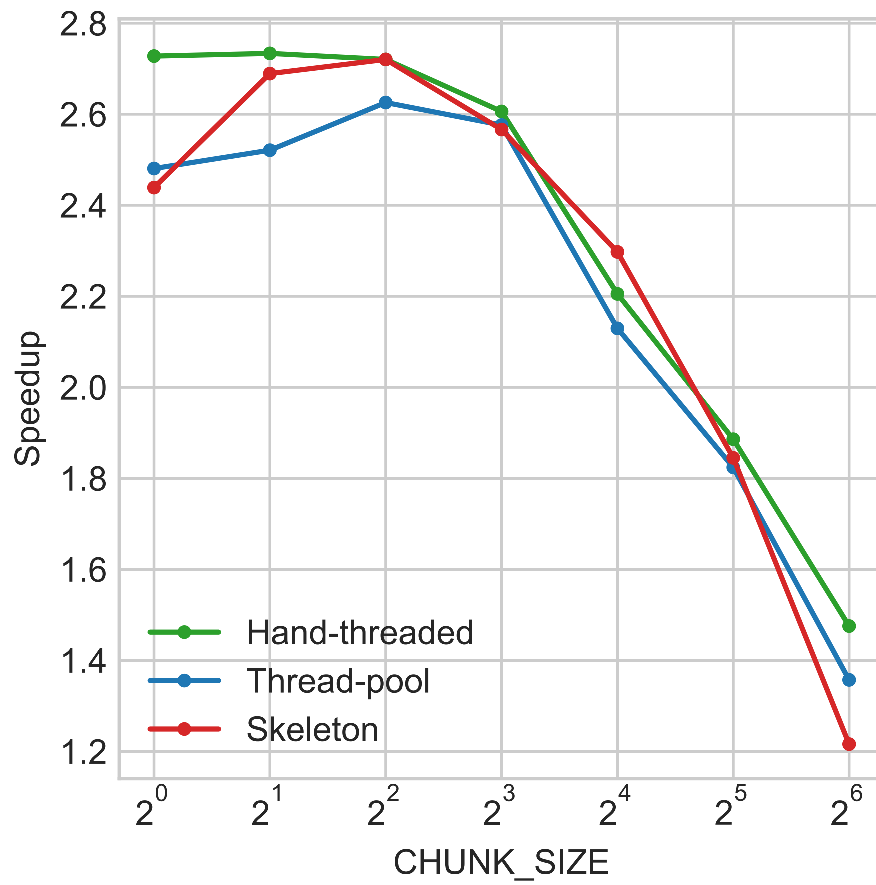

Figure A.3: Speedup for CYK Algorithm when number of threads is 4

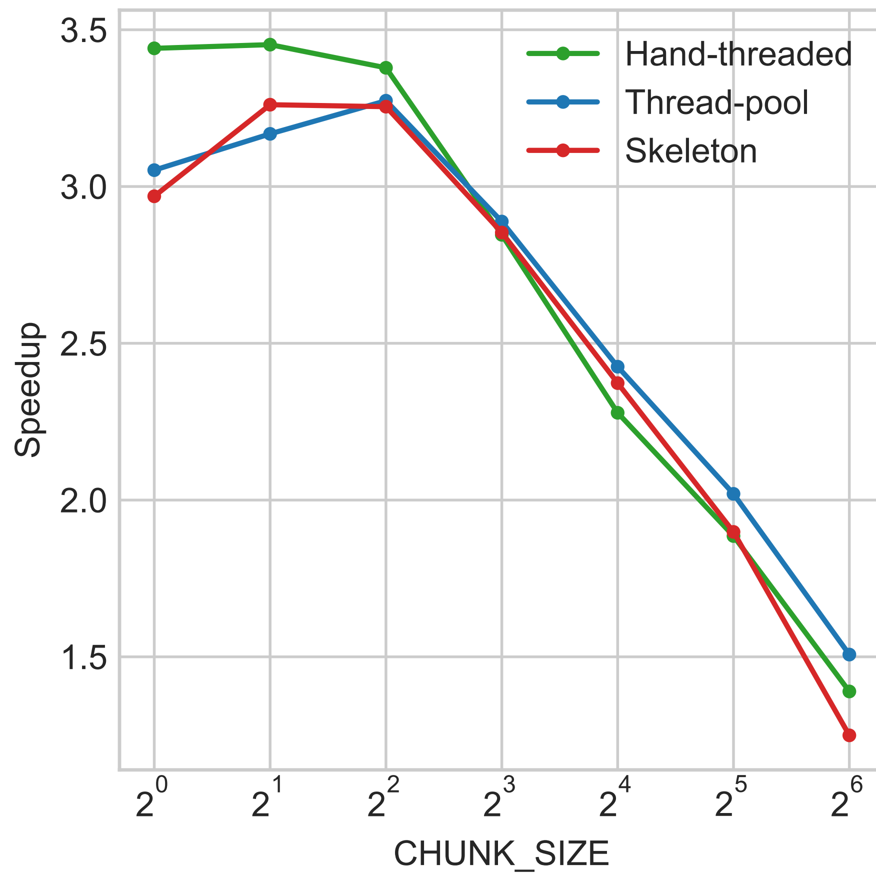

Figure A.4: Speedup for CYK Algorithm when number of threads is 6

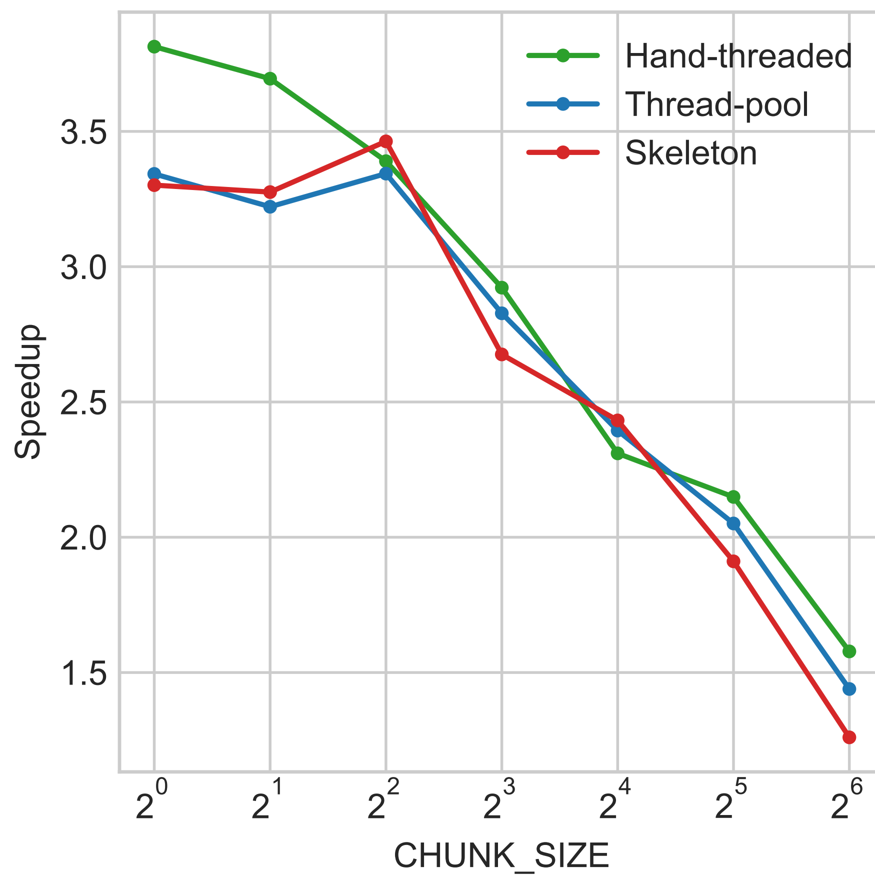

Figure A.5: Speedup for CYK Algorithm when number of threads is 8

A.5 Speedup for LCS Algorithm

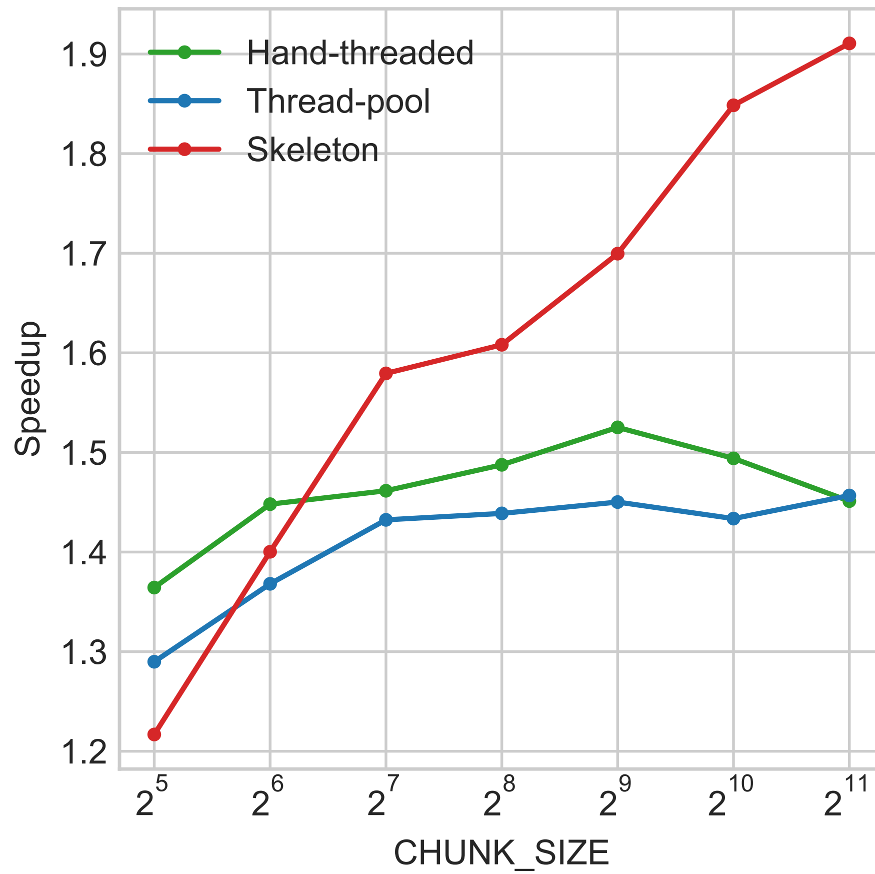

Figure A.6: Speedup for LCS Algorithm when number of threads is 2

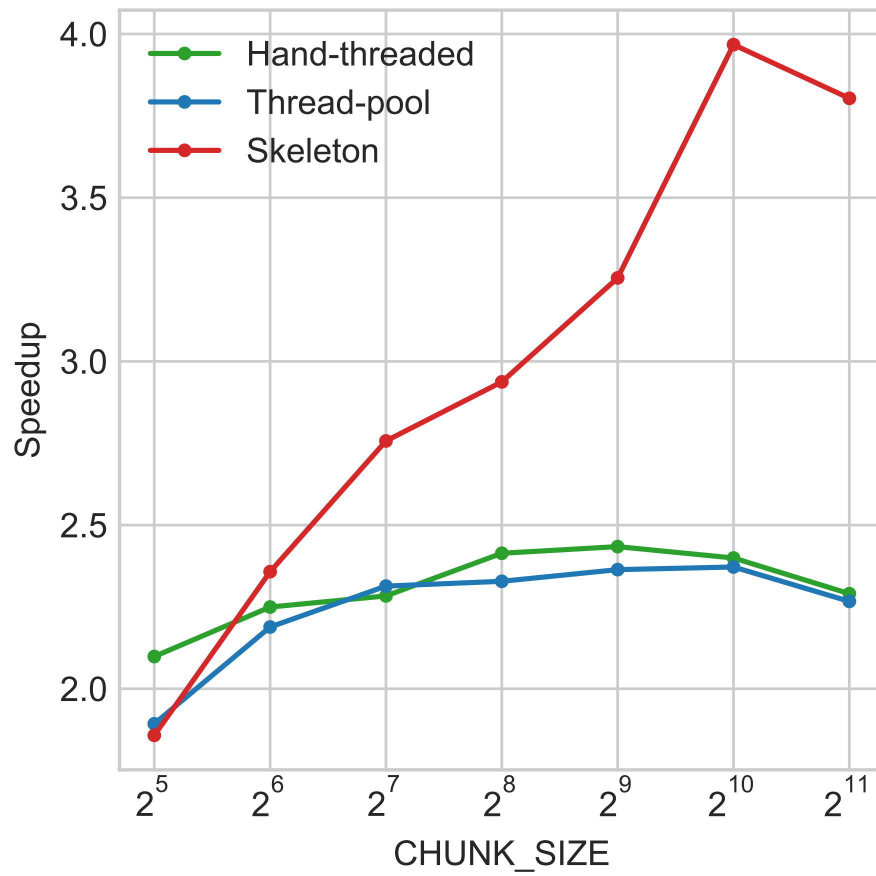

Figure A.7: Speedup for LCS Algorithm when number of threads is 4

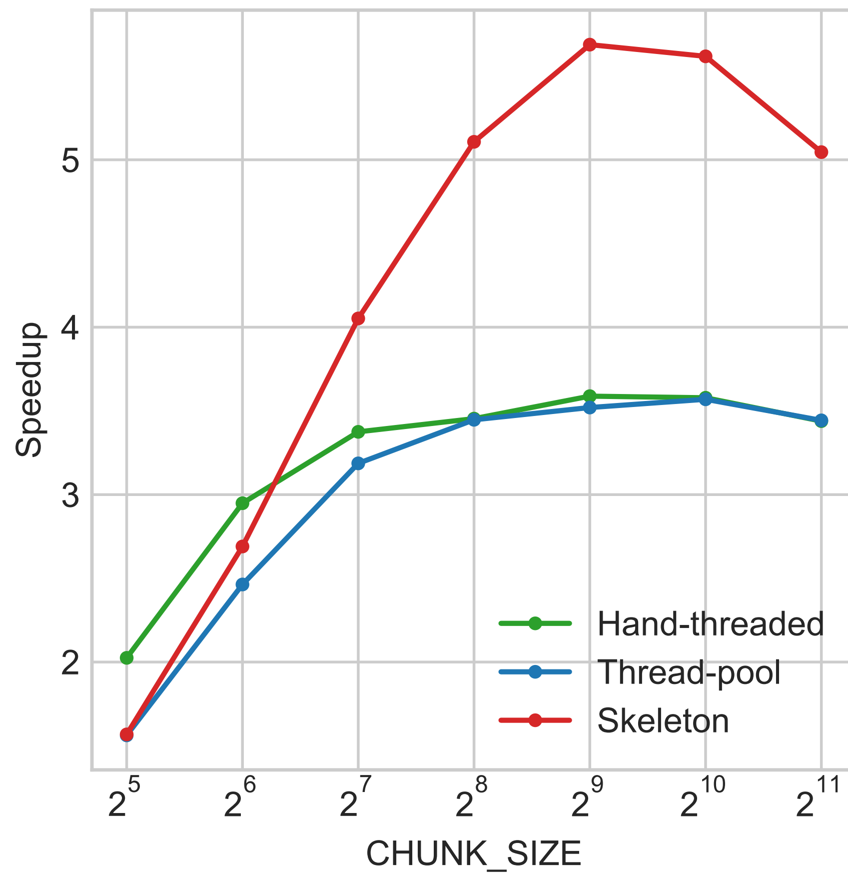

Figure A.8: Speedup for LCS Algorithm when number of threads is 6

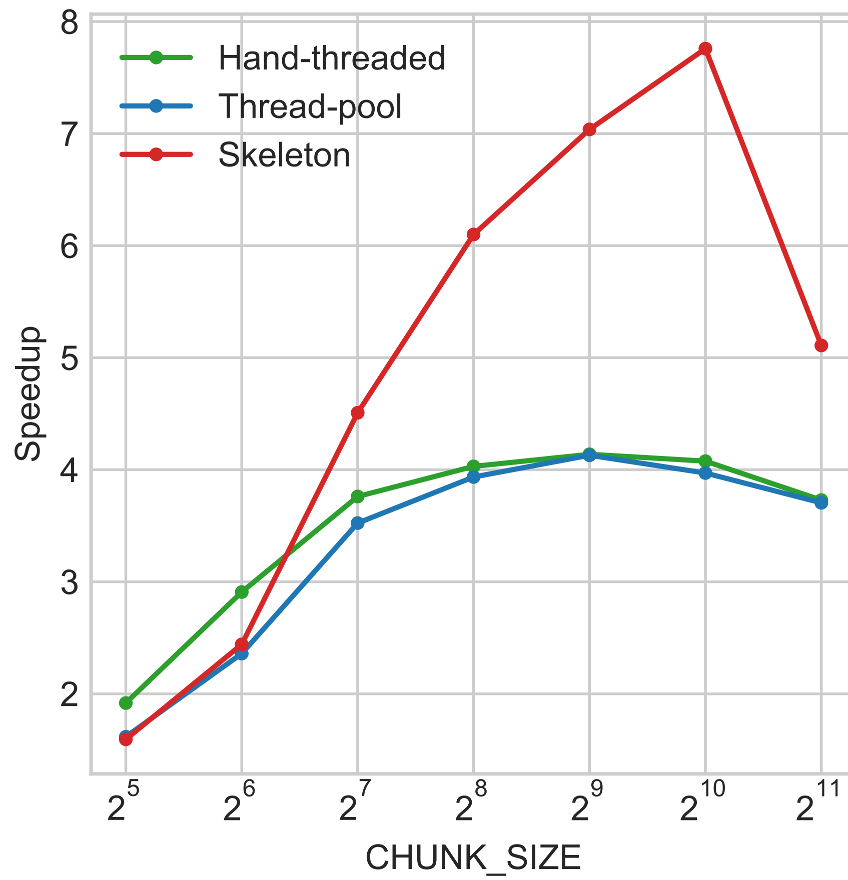

Figure A.9: Speedup for LCS Algorithm when number of threads is 8