Novice-Friendly Parallel Programming Exercises

Haoshi Wang

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

2023

Abstract

Parallel programming has become increasingly important as the demand for computational power continues to grow, and single-core performance improvements have slowed down. To address the need for skilled parallel programmers, it is essential to introduce parallel programming concepts early in computer science education. This project focuses on the design, implementation, and evaluation of a set of novice-friendly parallel programming exercises intended to facilitate the learning process for students new to parallelism. Six exercises were developed, each targeting different aspects of parallelism and varying in difficulty. These exercises were implemented using the Java programming language and the Java Thread class for parallelism. To assess the effectiveness of the exercises, a comprehensive evaluation methodology was established, including criteria such as speedup, scalability, style of parallelism, difficulty, and ease of verification. The results of the evaluation provide valuable insights into the performance and educational value of the exercises, enabling educators to select the most appropriate exercises for their students. This project contributes to the development of accessible and engaging parallel programming education materials, with the potential to enhance the learning experience for novice programmers and better prepare them for future challenges in the field.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 24985
Date when approval was obtained: 2021-09-24
The participants' information sheet and a consent form are included in the appendix.

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.

(*Haoshi Wang*)

Acknowledgements

I want to say a big thank you to my supervisor, Professor Murray Cole, for all the help, support, and encouragement they gave me throughout this project. His knowledge and advice really helped me understand parallel programming concepts and create the exercises in this report.

I want to recognize the participants in the study, too. Their feedback and experiences were very important in evaluating and improving the exercises I developed. Your time and effort made a real difference in the quality of my work.

Finally, I am truly grateful to my family and friends for always supporting me, loving me, and encouraging me during my time at school. Their belief in me and constant motivation has helped me succeed in this project.

Table of Contents

Chapter 1

Introduction

In 1965, Gordon Moore posited that roughly every two years, the number of transistors on microchips will double, which is known as Moore Law. For over five decades, the development speed of hardware follows what the law suggests. Hardware performance has improved at an exponential rate, an impressive achievement resulting from the continuous reduction in the size of transistor components. Since transistors form the circuits within a computer's central processing unit (CPU) that are used to execute computer instructions, smaller transistors made it possible to create more powerful cores, in terms of both capability and speed [2]. However, we are approaching the time that Moore's Law ends [10], as we are facing the physical challenge of increasing the number of transistors in the small chip. Thus, the performance of the single-core processor is approaching its upper limit and the industry began to encounter physical limitations on the speed of a single-core. Since then, computer performance has improved mainly due to the increase in the number of cores per computer rather than the speed of a single core.

Over the past decade, computational power has increased tremendously, primarily due to advancements in multi-core processors and distributed computing environments. This shift has led to a growing need for parallel programming skills among computer science students and professionals. As the demand for high-performance computing in various industries, such as finance, scientific research, and artificial intelligence, continues to rise, parallel programming becomes more relevant than ever [13].

Universities, as major educational institutions for various scientific fields, face the challenge of teaching parallel programming to undergraduate computer science students. Most undergraduate computer science programs mainly focus on sequential programming, with parallel programming concepts introduced only in the later stages [13]. Postponing parallel programming education hinders students from applying parallelism in other subjects and reduces their ability to naturally adopt parallel solutions as future professionals [4]. To address this issue, universities should introduce parallel programming courses early in the curriculum, allowing students to get a taste of "parallel thinking" early, and they could incorporate the parallel paradigm naturally.

The main research question of this project is: "What are effective parallel programming

exercises that can be used to teach parallel programming concepts to novice learners?" the following objectives can be used to address this question:

1. Identify key concepts and techniques in parallel programming that should be covered in the exercises.
2. Design and develop a set of parallel programming exercises that are accessible and engaging for novice learners.
3. Establish evaluation criteria and metrics to assess the quality and effectiveness of the exercises.
4. Implement, test, and validate the exercises using the chosen programming language and parallelism technique.
5. Evaluate the exercises based on the established criteria and metrics and analyze the results.
6. Provide further possible approaches that can be taken to improve the novice students' learning experience

This report will provide an overview of related research that has been done. The report will then present a detailed methodology for designing, implementing, and evaluating the exercises, including the testing and validation procedures employed. The implemented example code of exercises will be uploaded with the report together.

The report will also present the results and analysis of the evaluation, including performance metrics for each exercise, and a comparison of the results from different platforms (PC with AMD CPU and MacBook laptop with Intel CPU). Any interesting or unexpected findings will be discussed, along with the strengths and limitations of the project, and potential improvements or modifications for future research.

By providing a set of well-designed parallel programming exercises, a complete evaluation of their effectiveness, and insights into the challenges and opportunities in teaching parallel programming to novice learners, this report aims to contribute to the advancement of parallel programming education. Educators can use the exercises and analysis presented in the report to design engaging and effective parallel programming courses for beginner university students, ultimately helping them build a strong foundation in this increasingly important field.

In this project, I have made the following contributions:

1. Design a set of parallel programming exercises that are suitable for beginners.
2. Design evaluation criteria and metrics that can be used to analyse the exercises.
3. Design a survey to collect student feedback on subjective criteria, like interesting.
4. Implement, test, and validate the exercises using Java, Java Thread class and Junit test.
5. Gather and organise the data from the survey and the testing result, and draw graphs with the data.

6. Evaluate the exercises using the designed criteria and the data gathered from testing the exercises as well as the survey.
7. Provide the overall conclusion of how the exercise set can benefit students as well as teachers.
8. Provide further possible work that can be done to improve this project.

Chapter 2

Background

Much research has been done for investigating how to develop good programming exercises, teaching parallel programming, and how evaluate parallel program performance. The following paragraphs will introduce some related research.

Teaching Parallel Programming for Beginners in Computer Science[4]: This paper reported several experiments that have been done to find the parallel programming learning outcome from different groups of students, with prior knowledge of computer science or without (see Figure 1). Three of the experiments taken in the research show that for students with and without prior knowledge, the learning outcome is generally the same, and positive.

Experiments 2 and 3 are experiments taken for groups of students mixed with students that have prior knowledge and students without prior knowledge. The group of students study parallel programming content together (OpenMP for experiment 2, Pthread for experiment 3), class content provided to students is all same. At the end of the experiment, students took tests for the contest they learnt in class. Results show for students with prior knowledge, their average test scores do not have a significant difference from the scores of students without prior knowledge(93.8% to 88.9% for Exp 2, 98.8% to 94.6% for Exp 3).

Another important experiment is Experiment 4, which only takes students that do not have any prior knowledge. Pre and Post-tests are taken by students to compare the evolution of the knowledge in parallel programming (see Figure 2). The result shows most of the students can learn the parallel programming concept well, and most of the

(Exp II) Extension Course of OpenMP/2018	93.8%	88.9%
(Exp III) Extension Course of Pthreads/2018	98.8%	94.6%
(Exp IV) Extension Course of OpenMP/2019 (only comp first-year students)	-	94.1%

Figure 2.1: Experiment results, figure taken from [4]

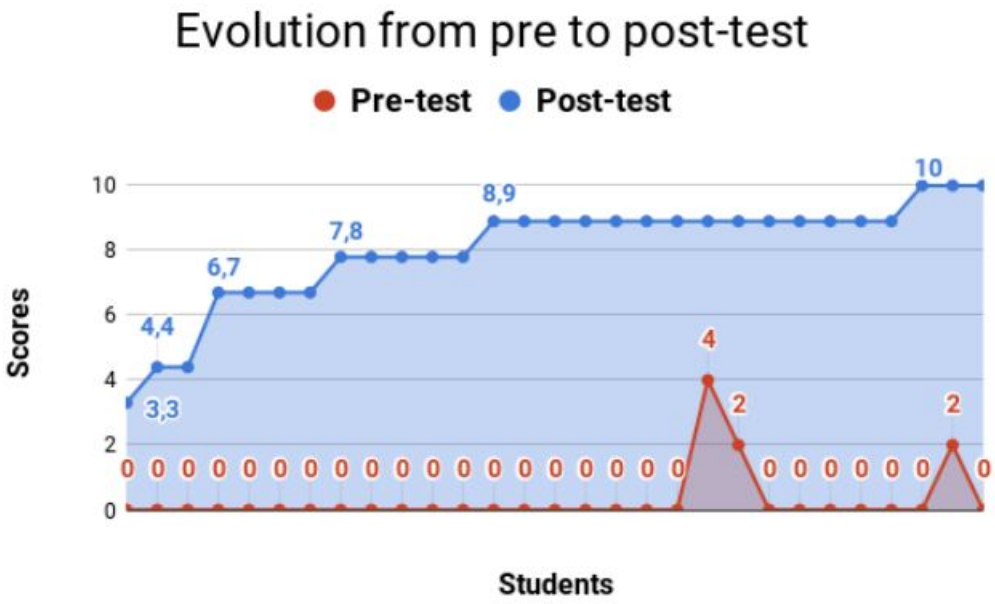

Figure 2.2: Pre and Post tests results, figure taken from [4]

students achieve 70% grades.

The results of this research show students that who do not have previous knowledge of programming can also achieve scores the same as the students who have studied computer science for years in university. For 1st or 2nd-year computer science students, they have the ability to understand and apply parallelism.

Teaching Parallel Programming with Active Learning [5]: This research shows how students progress when they study parallel programming with Active Learning. Active Learning means students are assigned into groups and are given tasks to work together to learn the content. Many studies have already shown that active learning improves students' performance. The research divides students into 2 groups, one group's students are split into teams, and another group's students study individually. Both of the groups are given the same content and tasks for them to complete, for the first group, tasks are given to teams and when students complete the task, teams have to present their solution to other teams, while for the second group, the teacher will provide the solution directly.

Students are asked to complete Pre and Post-tests (result see figure 3 section 1 represents students doing the individual study, section 2 represents students doing active learning), where the result shows before the course, students in 2 sections have a similar level of understanding of the course content. However, after they completed the course in two different ways, students who participated in the active learning course had a higher average test score, indicating active learning does bring benefit to students' studies.

During the experiment in this research, the researcher designed the course with a combined strategy, including visualizations, role play, and practical exercises. These ingredients of the course combined with active learning, improve the learning efficiency significantly. In addition, with the hands-on practical exercises, students also get

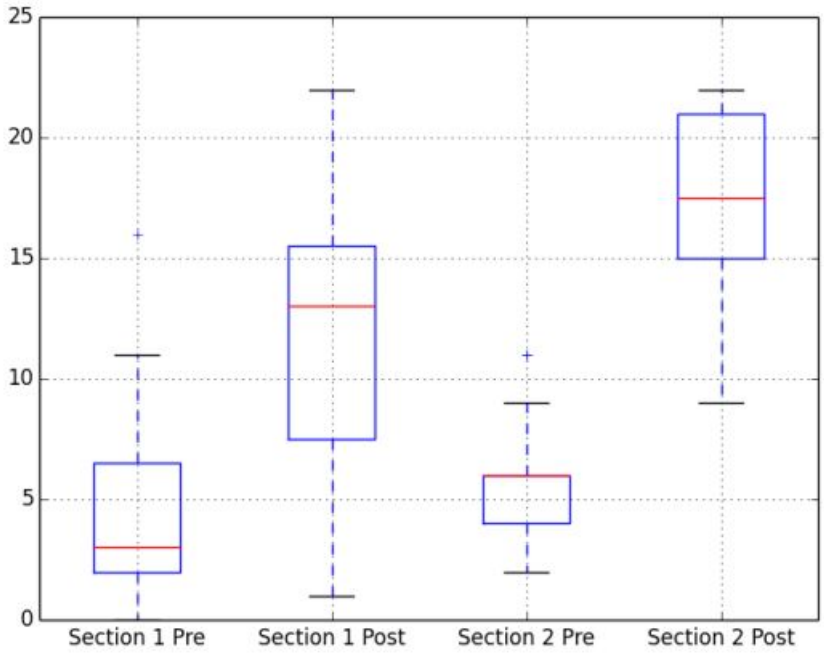

Figure 2.3: Pre and Post tests results for different groups, figure taken from [5]

experiences of using parallelism to improve the speed of traditional algorithms (like merge sort used in experiments) and also start to consider using parallelism for later course problems.

Principles for Designing Programming Exercises to Minimise Poor Learning Behaviours in Students [3]: The research investigated how improperly designed programming exercises will cause students to get poor learning behaviours. The research collected data from computer science students' courses at Monash University by interviewing students, observation from tutors, and student cases. The research concludes 3 major poor learning behaviours that students showed when they studied the course, and also provided principles for designing programming exercises that can reduce poor learning behaviours.

The first poor learning behaviour is Superficial Attention, meaning that students skim over the content, not attempting to dig inside the exercises. In this case, students may complete the exercise with acceptable accuracy, but actually do not understand how to approach the result. This habit is always led by tasks that require students to modify or reproduce the code. Students may directly copy and paste the code rather than study and understand the concept. Principles suggested by the author to avoid this habit when designing exercises are: rewording for understanding not completing; smaller coding questions; not always coding and outlining a method of attack.

The second habit is Impulsive Attention, meaning a student may not focus on the content that the designer wants them to, resulting in students spending a lot of time on the unnecessary part of the exercise, cannot complete the exercise in the end. In this case, students may consider the exercises difficult and lose enthusiasm to complete the exercise. The reasons leading to this problem are the tasks do not emphasise the

key concept and the tasks contain too many unfamiliar contents. The solution to this problem is to emphasise the key concept and provide suitable resources for unknown materials.

The last problem is Staying Stuck, meaning students lack of strategy to deal with being stuck, they will not try to revisit the materials or instructions or try to analyze what they have done and thought of new approaches. The reason causing students to have this problem is either students do not know how to start, or they do not know how to use their knowledge to build the solution to the task, or they are stuck in a debugging session. To improve the exercises, the designer should follow strategies like providing guidelines for writing and testing the code, providing useful resources and references, provide graded helps on how to start the task.

The research provided principles that a programming exercise designer should follow. Although the author did not take the experiments with teaching parallel programming, it is reasonable to assume these principles work not only on sequential programming exercises but also on parallel programming. Some of the principles are very important considerations when designing parallel programming. For example, tasks that require reproducing or modifying code often do not lead students into discussions to think deeply about the material, thus, making exercises for parallel programming should ask students to first step into the sequential version of the exercise, implement the sequential version first, then modify their own code into parallel version, rather than provide them with sequential version and ask them to modify into parallel version.

Chapter 3

Methodology

This section outlines the methodology employed in designing, implementing, and evaluating the parallel programming exercises for novice learners. The methodology consists of the following steps:

1. Design
2. Implementation
3. Testing and Validation

3.1 Design

3.1.1 Evaluation criteria and metrics

We designed seven criteria defined to evaluate the exercises, the criteria include the basic concepts of parallel programming, as well as the subjective opinions of the exercises, for example: interesting. All seven criteria are listed below:

Speedup [6]: Speedup is simply how much quicker the parallel program's run-time is than its sequential version's run-time. If we define the run-time of the sequential version as T_S, the run-time of the parallel version as T_P, the formula of speedup S is:

$$S = \frac{T_S}{T_P} \tag{3.1}$$

This criterion is checking whether a parallel program can gain speed with a certain amount of data size and thread number.

Speedup is a fundamental metric in parallel programming. By emphasizing exercises that demonstrate significant speedup, students will realize the potential performance improvements offered by parallelism.

Scalability [6]: The scalability of a program is that a program is scalable if it can obtain speedups when using it on a larger system. Running on a larger system includes running with more threads, running with a larger data size or running with a larger data size and

more threads. With this concept, the scalability of a program can be defined as three types:

1. **Strongly scalable**: How speedup changes as you add more threads for some fixed problem size.
2. **Weakly scalable**: How speedup changes when you increase the problem size along with an increase in thread count.
3. **Not scalable**: By increasing the problem size and the number of threads, the program can not show speedup.

Scalability is an essential characteristic of successful parallel programs. By using scalability as one of the evaluation criteria, it allows students to think about how their solutions can be adapted to different problem sizes and hardware configurations.

Style of parallelism [6]: The style of parallelism can be defined into two types:

1. **Task parallelism**: Programs in which each thread executes a different task
2. **Data parallelism**: Programs in which the data are split among the threads, and each data part is processed with the same block of code.

An easily understandable example to explain the style of parallelism is doing kitchen work: Imagine you are cooking in the kitchen, and you need to cook 100 dishes. Data parallelism is you split dishes into several parts, and ask your friends to cook dishes, each of your friends takes a part of 100 dishes.

Task parallelism, in another way, you do not split the dishes into parts, but each of your friends is doing a different job which helps cook the dishes, for example, one friend is boiling the water, one friend is cutting the raw materials and etc.

Introducing both task parallelism and data parallelism exposes students to different strategies for parallel programming. Understanding the nuances of these approaches will help them develop a more comprehensive skill set and better prepare them for real-world parallel programming challenges.

Speedup Visualization: Is the problem able to show the speedup dynamically, rather than just output speedup as numbers?

Dynamically presenting speedup, such as graphs or animations, can help beginners understand the benefits of parallel programming more intuitively. It is direct feedback to students rather than looking at a static output. This can motivate students to learn more about the subject and make it easier for them to understand the concept of performance improvement through parallelization.

Difficulty: The difficulty to complete the exercise.

Ensuring that the exercises have various difficulties for beginners is important. If there are only super easy exercises, students may not develop the skills required for parallel programming. If there are only difficult exercises, students may become discouraged and lose interest. This criterion is used to ensure the exercise set covers exercises with a range of difficulty, but not extremely difficult for the beginner.

Interesting: is the question interesting?

Engaging in interesting exercises can capture students' attention and stimulate their curiosity, it also can encourage students to explore the topic further, leading to a better understanding of the concepts.

Difficulty to Verify: Can students easily verify that their implementation of the problem is correct? i.e. can students come out with some test cases easily that can be used to test their solution's correctness?

Providing exercises that are easy to verify allows students to build confidence in their abilities and learn from their mistakes. When students can quickly test their solutions, they can iterate and refine their understanding, leading to better overall comprehension of parallel programming.

For the first four criteria: Speedup, Scalability, style of Parallelism, and Speedup Visualization, Speedup and Scalability are evaluated using data collected during the testing and validation phases. Style of Parallelism and Speedup Visualization, being characteristics of the exercise, can be evaluated directly based on the exercise's description.

The remaining three criteria: Difficulty, Interesting, and Difficulty to Verify—are subjective. To obtain a more impartial assessment of these criteria, rather than relying solely on the author's perspective, a survey is designed to gather feedback from university students. The data collected will provide a compelling understanding of these subjective criteria.

The survey is structured in a straightforward way, consisting of nine sections. The first section presents the survey's background information and relevant contacts. The second section shows the Participant Information Sheet and the Participant Consent Form. Participants must review the Participant Information Sheet and agree to the statements in the Participant Consent Form to proceed. The third section is an introduction to the survey, detailing the structure of the subsequent sections, the meaning of the questions, and guidance on answering them. Sections four to nine contain a description of the exercise and corresponding questions. All questions in the final six sections are the same, there are three questions in each section, they are:

1. Is the exercise interesting?
2. What is the difficulty of the exercise?
3. What is the difficulty to verify the exercise?

Each question asks the participant to select a number between 0 and 10, where 0 means it is easy/ not interesting/ easy to verify, and 10 means hard/ very interesting/ and hard to verify. We consider the average score lies on 1-3 to be easy/ not interesting/ easy to verify, score lies on 4-6 to be medium-level difficult, interesting and difficult to verify, and score lies on 7-10 to be hard, very interesting and hard to verify.

3.1.2 Overview of parallel programming exercises

There are six exercises developed in this project, This subsection will provide a brief description of each exercise, as well as their characteristics. For all of the exercises, students are asked to write both sequential and parallel versions of the exercise.

Scrabble: This is a game where players get pieces with letters that they use to form words in a crossword-like puzzle. Each letter is worth a different amount of points. A word is worth the number of points that its letters add up to. Students are asked to design and implement an algorithm that takes a list of words as input and outputs the number of points that each word is worth.

The Scrabble exercise is an interesting and easy exercise as our consideration, at the same time, it should not be hard to verify. It is related to a real-world game, students are implementing a program that can solve a real-world problem which makes it interesting. This exercise does not have a special way to visualize the speedup, it is designed to output the speedup directly as a number. Scrabble is intended to be implemented in the data parallel style.

Mandelbrot Graph Rendering [12]: The Mandelbrot Set is a collection of complex numbers. We iterate function (3.2) with initial z = 0 and input c, the iteration will terminate when the z value becomes greater than 2, which means the number diverges. The interaction will also terminate when the number of iterations reaches the arbitrary limit, in this case, the number c converges and will be considered as a number in the Mandelbrot set. Figure 3.1 demonstrate an example of the Mandelbrot set graph, the graph is an x, y plane and x, y values are from -2 to 2. The dark part of the graph indicates the numbers in the plane belonging to the Mandelbrot set, whereas the white, highlighted part means the boundary of the Mandelbrot set. Other parts mean the numbers are not in the set.

$$Z_{k+1} = Z_k^2 + c \tag{3.2}$$

The reason that the numbers not in the Mandelbrot set are coloured in various ways is that numbers are taking a different number of iterations to be proven as not in the set. the colour of points that is not in the set depends on the number of iteration it needs to prove to diverge.

The Mandelbrot Graph Rendering exercise requires students to write a parallel version program that renders a Mandelbrot graph, skeleton code will be provided allowing the student to visualize the graph. The skeleton will also contain functions like live rendering and different threads with different colouring schemes. The skeleton can be modified to adopt different difficulties and can be used to demonstrate parallel programming concepts as well.

We believe the Mandelbrot Graph Rendering exercise is a hard exercise, novice parallel programming may found challenging to do it. It is also interesting and easy to verify. It has strong speedup visualization, students can see how each part of the graph gets rendered with different threads. It is intended to demonstrate the concept of data parallelism.

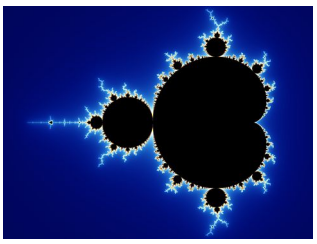

Figure 3.1: Mandelbrot graph example, figure taken from [14]

Student Selection: StudentInfo is a Java class that contains the information of the student, including first name, second name, course score, etc. Students need to write a program that takes a list of instances of StudentInfo class and filters the list with certain criteria. For example: find out all students that fail the course, meaning that the program should output all instances that have scores lower than 40 %.

We think this exercise is an easy one. It looks like the filter function of a database or Excel, which can be used in the real world, thus we believe this question should be interesting. The difficulty to verify depends on the complexity of the filter, if there are many criteria applied to the filter, it may take extra effort to construct test cases to cover all situations. This exercise does not have a special way to visualize the speedup, it is designed to output the speedup directly as a number and is intended to be implemented in the data parallelism style.

Matrix Multiplication: This exercise requires students to write a program that multiplies two N * N matrices and outputs the result N * N matrix, the multiplication of matrices A and B with result C follows function (3.3) [12].

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} B_{k,j} \tag{3.3}$$

This exercise is designed to be a medium-difficulty exercise, as an exercise that only does the calculation, it may be not interesting enough. However, this exercise should be easy to verify, as students can make up 2 matrices and do the multiplication themselves. This exercise demonstrates that parallel programming is not only used to increase a program's performance but also can be used to solve large problems that are beyond the capacity of a single processor.

Morse code Encryption: This exercise involves creating a program that takes a paragraph of English text as input, and outputs the respected Morse code cypher text of the input paragraph. Spaces are used to separate Morse code that represents letters and slashes are used to separate the words. For example, the text "hello world" will be translated to ".... . .-.. .-.. — / .– — .-. .-.. -..".

We consider this exercise moderate difficulty, interesting and easy to verify, the main challenge of this exercise is how to merge the output of different threads. Students need to carefully consider spaces and slashes when merging the threads' outputs. This exercise does not have a special way to visualize the speedup, it is designed to output the speedup directly as a number. It is intended to be implemented in the data parallel style.

Array Processing: This task is simply finding the max, min and sum of a list of numbers.

We designed this exercise to be an easy task, suitable for being a warm-up. This exercise differs from other exercises by demonstrating the concept of task parallelism. Students are required to write the task paralleled form of the exercise, which creates three threads while threads 1, 2 and 3 are responsible for calculating the max, min and sum of the array respectively. This exercise does not have a special way to visualize the speedup, it is designed to output the speedup directly as a number.

3.2 Implementation

The exercises have been developed using the Java programming language with Java Development Kit (JDK) version 11, encompassing both sequential and parallel variants. Java offers numerous benefits, such as advanced programming concepts, and enhanced compile-time and runtime checks, which in turn facilitate quicker issue identification and debugging. In addition, The JDK offers an extensive collection of libraries that developers can reuse for accelerated application development. Another argument in favor of Java is the large community of developers, mainly because it is the primary language taught in numerous universities worldwide. A strong advantage of Java-based applications is their portability across diverse hardware and operating systems, as long as a Java Virtual Machine (JVM) is available for the respective system.[12].

To implement the parallel version of the exercise, we decided to use Java Thread Class. It is one way to create a thread in Java, other ways like Fork/Join framework and implementing runnable are also options to create threads in Java programs. Using the Java Thread Class for parallel versions has several advantages: it is simple and straightforward to implement, grants students the ability to manage the number of threads they want to use, and allows them to explore the exercise's potential features by changing the thread count.

Figure 3.2 shows an example of creating a thread using the Java Thread class. To do so, you have to create a new class that extends the Thread class and implement the run() method, which is done by code in line 2 to line 5. The code block in the method run() is what you are expecting the thread to do, in the example, it is print text "thread is running...". Codes in line 7 and 8 demonstrate how to run a thread, first, you have to create an instance of the class that extends the Thread class, then call the "start" method of that instance. In the example code, the console will print out "thread is running..." after running the main.

The main difference between exercises that use data parallelism style is the way of splitting the input and the way of merging the output. For exercises like Student selection, Morse code Encryption and Scrabble, they take either an array or a string as input, and the string can consider as an array of chars. Thus, the input for these exercises is directly divided into n chunks, where n is the number of threads used to run the program, and each thread takes one of the input chunks and does the same computation. To handle the situation that the input size is not evenly divisible by the thread number, we write a helper function to ensure that the remainder data is allocated

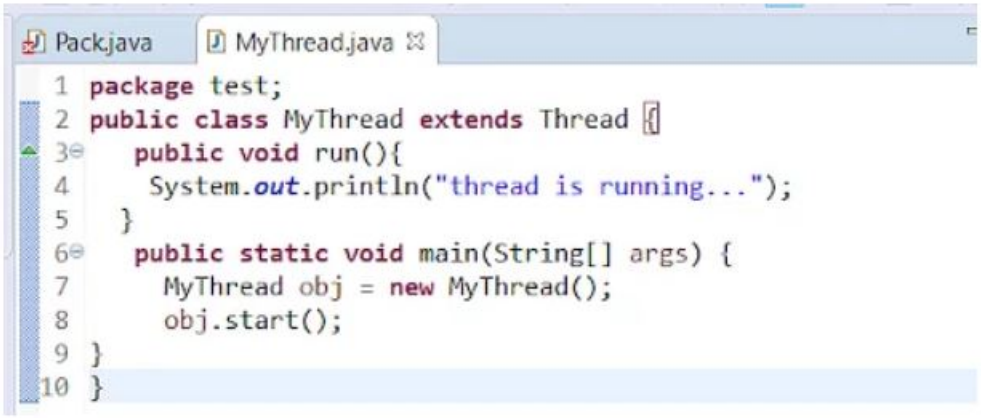

Figure 3.2: Example of Using Java Thread Class, figure taken from [11]

to the last data chunk. After the threads complete their job, the result is merged by either concatenating the arrays or adding up the scores that are output by the threads.

In the Morse code Encryption exercise, the merge process presents unique challenges. Directly concatenating output strings from different threads is not feasible. Firstly, the strings must be concatenated in sequence to preserve the paragraph's meaning. Secondly, since spaces separate letters, directly concatenating the output strings may result in the loss of spaces. For instance, while translating "hello world" with two threads, the first thread processes "hello" and the second one processes " world". The first thread's output is ".... . .-.. .-.. —", and the second thread's output is "/ .– — .-. .-.. -..". Direct concatenation would lead to the loss of space before the slash, deviating from the expected output. To address this issue, extra space is added during concatenation except for concatenating the result from the last thread.

In the Matrix Multiplication exercise, the input consists of two N x N matrices rather than an array. Here, we treat the matrix as a length N array and divide it into x parts, where x denotes the number of threads employed. As a result, each thread computes a segment of the resulting matrix. For instance, when multiplying two 8 x 8 matrices A and B using 4 threads, each thread calculates a 2 x 8 matrix by multiplying a 2 x 8 segment of matrix A and the entire 8 x 8 matrix B. This produces a 2 x 8 matrix, which is a part of the resulting matrix C. Ultimately, matrix C is composed of four 2 x 8 matrices, corresponding to the outputs from the 4 threads, as illustrated in Figure 3.3.

The Mandelbrot Graph Rendering exercise is implemented based on the pseudo-code provided in Figure 3.4. The program uses nested for loops with i and j as iterators to traverse each pixel on the screen. The x and y values represent the scaled x and y coordinates of the pixel. The while loop checks if the complex number formed by x and y belongs to the Mandelbrot set and returns the number of iterations required to confirm this. The graph is then plotted using the iteration count of each pixel to determine the corresponding color.

In the parallel version of the exercise, the graph is divided into n sections, where n

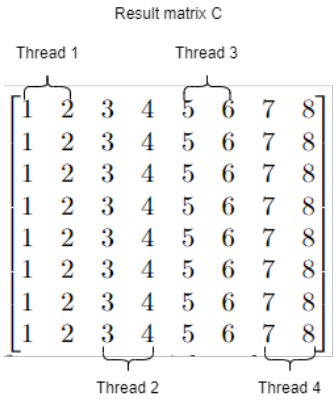

Figure 3.3: Example of result matrix c by multiplying 2 8*8 matrices

represents the number of threads used for rendering. This approach employs a data parallelism style. Figure 3.5 provides an example of how the graph is divided.

To achieve dynamic visualization, the algorithm has been modified to enable live rendering. Unlike the original algorithm that processes all pixels before plotting the graph, our approach renders each pixel immediately after completing the required iterations [7]. This allows the graph to load progressively from top to bottom.

Since the graph rendering is performed by multiple threads, a new color scheme has been introduced to distinguish the output from different threads. A random color generator, using the thread number as a seed, produces unique Red Green Blue values to ensure color of the graph output from each thread is unique. Using the seed can also ensure the color consistency within each thread. The result section will display an example of the Mandelbrot Graph Rendering exercise output.

```
1 for(int i=0; i<N; i++)
2   for(int j=0; j<N; j++) {
3     double x=step * i - 2.0; //   -2<=x<=2
4     double y=step * j - 2.0; //   -2<=y<=2
5     complex c=(x, y), z=c;
6     int k = 0;
7     while (k<CUTOFF && abs(z)<2.0) {
8       z=c + z * z;
9       k++ ;
10    }
11    set[i][j]=k;
12  }
```

Figure 3.4: Pseudo-code for sequential Mandelbrot, graph taken from [12]

In the exercises that have been implemented in the task parallelism style, the number of threads using is not changeable. For example, in the Array Processing exercise, there can only be three threads, each thread calculating the max, min and sum respectively. In this case, threads are doing different jobs with the same input and they will have independent output.

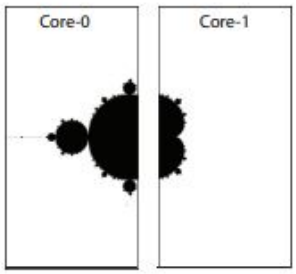

Figure 3.5: Mandelbrot graph partition, graph taken from [12]

3.3 Test and Validation

In this section, we will describe the testing and validation procedures used to evaluate the parallel programming exercises developed in this project. This includes a discussion of the testing methods, test cases, and validation techniques employed. By ensuring the correctness and effectiveness of the exercises, we aim to provide a valuable learning experience for students and contribute to the advancement of parallel programming education.

The testing and validation are done using the Java Junit Test, with Junit version 4.13. The test is done on two different platforms, they are a desktop computer with AMD 8 cores 16 threads CPU, Windows 10 operating system. A MacBook Pro laptop, with Intel 8 cores 16 threads CPU, MacOS operating system. Testing on different platforms can ensure the portability of the exercise and collect speedup data to compare the difference when running the exercises on different platforms.

Each exercise will be tested with several test cases, including normal input, randomly generated input and extreme value inputs. Both sequential and parallel implementations are tested with these test cases. In addition, the parallel version is tested with different numbers of threads used (from 1 to 8) to ensure general correctness.

Except for the Mandelbrot graph rendering exercise, the other five exercises follow the same steps of testing and validation. The Mandelbrot exercise is tested by comparing the shape of the output graph with the one shown in Figure 3.1. The test is also run with the different number of threads used, as well as different sizes of the input.

To collect data that can be used to analyse the speedup and scalability, a program that contains two nested loops is used to get the performance data of the exercises. The outer loop loops eight times represent the number of threads used in the parallel version of the exercises. The inner loop loops six times, which controls the data size of the input that passes to both sequential and parallel versions of the exercises. Each time the program loops, the sequential version and parallel will execute 100 times and record their run time using Java System.nanotime to ensure accuracy. After the run time has been recorded, the run time will be used to calculate the speedup of the parallel version of the exercises, using the formula (3.1).

The result of the program will be recorded as a table, an example of a table is shown in Figure 3.6, where all zeros in the centre represent the speedup of the exercise under

different input sizes and numbers of threads used. In the result and analysis section, the table will be plotted as a graph, and the table itself will be inserted in Appendix A, in case readers want to check the exact data.

Speedup Using Thread PC						
	Data Size					
Threads Used	10	100	1000	10000	100000	1000000
1 Thread	0	0	0	0	0	0
2 Threads	0	0	0	0	0	0
3 Threads	0	0	0	0	0	0
4 Threads	0	0	0	0	0	0
5 Threads	0	0	0	0	0	0
6 Threads	0	0	0	0	0	0
7 Threads	0	0	0	0	0	0
8 Threads	0	0	0	0	0	0

Figure 3.6: Example of the table contains speedup data

The scalability analysis is based on the speedup data, if the speedup table shows an increasing trend of speedup, that means the exercise is scalable. If the exercise is scalable, an additional data collection script will be used to gain the weak scaling data (strong scaling data already included in the table). We will pick the smallest data size that shows a speedup greater than 1 as the base data size (with any number of threads). With this base data size, we run the exercise program with scaled data sizes and corresponding thread counts. For example, if the exercise program begins to exhibit a speedup greater than 1 when the data size reaches 10,000, we will run the exercise program with 1 thread and 10,000 data size and 2 threads with 20,000 data size, up until 8 threads and 80,000 data size. We will obtain the speedup data from this data collection script and plot the graph of the speedup data, the graph will be used to do the weak scaling analysis of the exercise.

Chapter 4

Results and Analysis

This section presents a comprehensive evaluation of the parallel programming exercises developed in this project. This evaluation aims to assess the correctness, performance, and educational effectiveness of the exercises, ensuring that they provide a valuable learning experience for students who are new to parallel programming.

This section will present the performance metrics for each exercise, including speedup, scalability, style of parallelism, speedup visualization, as well as the results of survey feedback. This section will also provide an evaluation based on the performance metrics and the survey feedback. This analysis will highlight the strengths and weaknesses of each exercise, as well as any interesting or unexpected findings that emerged during the evaluation process.

4.1 Scrabble

Figures 4.1 and 4.2 show the weak scaling data and speedup data for the Scrabble exercise on the PC and the MacBook laptop. Figure 4.2 is plotted with data from Figures A.1 and A.2. Figure 4.1 is plotted with the weak scaling data (the way to collect data is introduced in section 3.3). The horizontal axis of Figure 4.1 means the number of threads used with corresponding data size (1 means 1 thread with 10000 words)

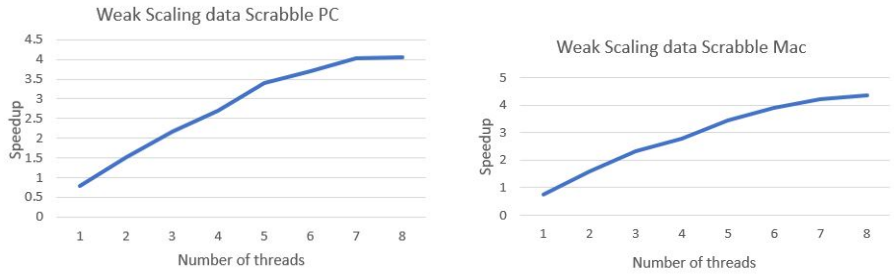

Figure 4.1: Weak scaling data Scrabble

Speedup: From Figure 4.2, it is obvious that when the data size increases to 10000 and the program is running with 2 or more threads, the Scrabble exercise achieves a speedup

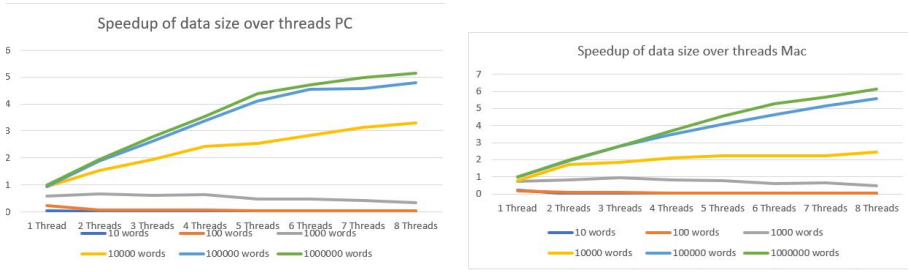

Figure 4.2: Speedup data Scrabble

greater than 1 on both platforms, which means the parallel version gains a performance improvement.

Scalability: According to Figure 4.2, we can find when increasing the number of threads used and the data size, the speedup will increase as well, which indicates that the exercise is scalable.

Figure 4.1 reveals that for up to seven threads, the speedup is increasing linearly, suggesting that the weak scaling performance of the parallel solution is generally satisfactory for up to seven threads.

Figure 4.2 presents the speedup data graph, highlighting that when the data size is 10,000 or greater, increasing the number of threads employed results in a higher speedup. This implies that for data sizes of 10,000 or more, the exercise exhibits favourable strong scaling performance.

Style of parallelism: This excise is implemented by splitting input data into smaller chunks and each thread takes a chunk to compute after all of the threads finished their work, their results are merged together to get the final result. This is a typical data parallelism.

Speedup Visualization: This exercise show speedup by printing out the calculated numerical speedup.

Interesting: According to Table 4.1, the average score of interesting in the Scrabble exercise is 6.70 out of 10, suggesting that students found the Scrabble exercise to be engaging and enjoyable which matches our expectations. This can motivate students to learn and explore parallel programming concepts more effectively.

Difficulty: From table 4.1, the difficulty score of 5.94 out of 10 suggests that the Scrabble exercise presents a moderate level of challenge for students. However, we consider this exercise to be an easy one. One possible reason causing the students thinks this exercise is a moderate difficulty is that student with no programming experience does not know how to use the dictionary to solve this exercise. They may think it is hard to check every letter's score.

Difficulty to Verify: The difficulty to verify score is 3.47 out of 10 from Table 4.1, implying that students find it relatively easy to come up with test cases to check their solutions' correctness. We consider this exercise easy to verify as well.

Interesting	Difficulty	Difficulty to Verify
6.70	5.941	3.47

Table 4.1: Survey result of the Scrabble exercise

4.2 Mandelbrot Graph Rendering

Figures 4.3 and 4.4 show the weak scaling graph and speedup graph for the Mandelbrot Graph Rendering exercise on the PC and the MacBook laptop. Figure 4.4 is plotted with data from Figures A.3 and A.4. Figure 4.3 is plotted with the weak scaling data (the way to collect data is introduced in section 3.3). The horizontal axis of Figure 4.3 means the number of threads used with corresponding data size (1 means 1 thread with 1000 max-iterations)

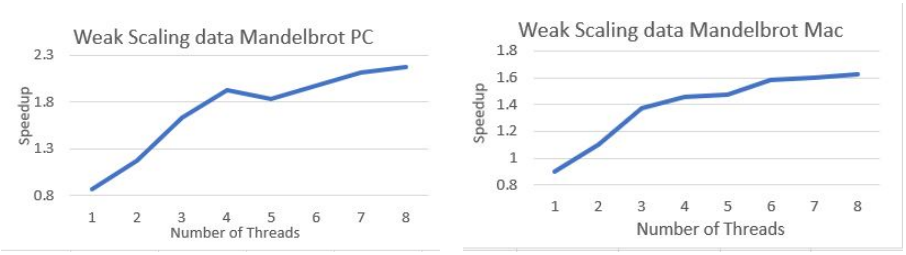

Figure 4.3: Weak scaling graph Mandelbrot

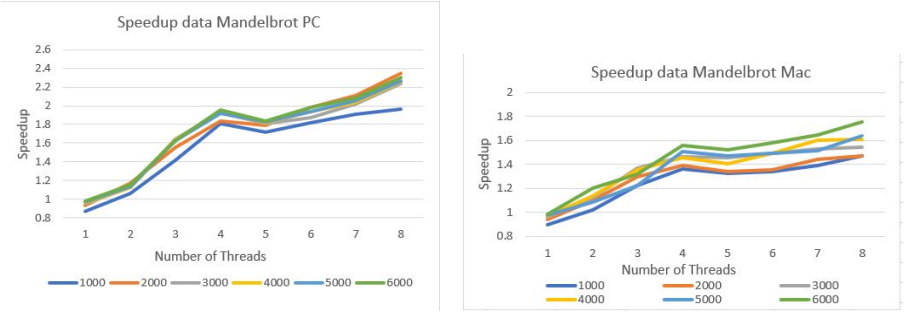

Figure 4.4: Speedup graph Mandelbrot

Speedup: Figure 4.4 illustrates that the Mandelbrot Graph Rendering exercise starts with data size equal to 1000 iterations, and with 1000 iterations, the exercise can show a speedup greater than 1. This exercise can demonstrate performance improvement brought by parallelism.

Scalability: According to Figure 4.4, we can find when increasing the number of threads used and the data size, the speedup will increase as well, which indicates that the exercise is scalable.

Figure 4.3 shows that for up to 4 threads, the speedup goes up linearly, suggesting that the program shows a good weak scaling until the number of threads used reaches 4. After adding more threads, the rate of increase in speedup slows down.

Figure 4.4 indicates that when the data size is greater or equal to 1000 max-iterations, adding more threads will bring an increase in speedup. This shows that when the exercise takes data size that is larger than 1000, it will show good strong scaling performance.

One reason that might cause this exercise to show bad weak scaling is the max-iterations do not fully represent the data size of this exercise, as increases the max-iterations will only result in the pixel in the Mandelbrot set having higher run time, for pixel not in the set, there is not much effect. Another possible way to increase the data size is to increase the Mandelbrot graph resolution, i.e. increase the number of pixels.

Style of parallelism: This exercise applies data parallelism, as the exercise is intended to render a graph, so the graph is split into n parts vertically (as shown in the design section), where n is the number of threads used to render the graph. Each thread renders a part of the graph, which match the concept of data parallelism.

Speedup visualization: `https://youtu.be/j_N0mostvzQ` This video provides a demonstration of the Mandelbrot Graph Rendering exercise, especially highlighting the live rendering function and multi-coloring function.

Figure 4.5 shows what the graph looks like during the rendering process. The white part is part of the graph that has not been rendered, and it will be filled with color while the program is running. This provides a feeling of ”Loading”, which is dynamic visual feedback of how fast the program is running. Students can change parameters like max iterations or the number of threads used to see the program running speed under different levels of resources used. This function can provide direct feedback on how can parallelism bring performance improvement.

Figure 4.6 shows the completed Mandelbrot graph, rendered by 16 threads. This figure demonstrates the multi-coloring function where the part of the graph rendered by a different thread will have a different color. This function can show students how each thread contributes to the final result, and give them a clear illustration of how data parallelism works.

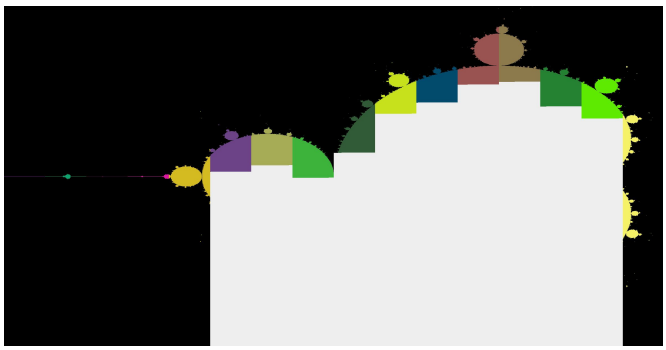

Figure 4.5: The Mandelbrot graph during rendering

Interesting: According to Table 4.2, the average score of interesting in the Scrabble exercise is 7.05 out of 10, meaning that students think the Mandelbrot Graph Rendering exercise is interesting. The survey result matches our opinion.

Figure 4.6: Finished Mandelbrot graph

Difficulty: From table 4.2, the difficulty score of 6.90 out of 10 suggests that the Mandelbrot Graph Rendering exercise is considered as a challenge. We have the same opinion on the level of difficulty this exercise should have. However, the level of difficulty of this exercise can vary depending on how much detail is provided in the skeleton.

Difficulty to Verify: The difficulty to verify score is 7.70 out of 10 from Table 4.2, implying that students find it difficult easy to verify their solutions' correctness. This differs from our expectation, as we are expecting this exercise should be easy to verify. Students may be stuck on how to write proper test cases to prove the correctness of their solution, but this exercise can be verified by simply checking the output graph with the standard Mandelbrot graph.

Interesting	Difficulty	Difficulty to Verify
7.05	6.90	7.70

Table 4.2: Survey result of the Mandelbrot Graph Rendering exercise

In addition to the above criteria, the Mandelbrot Graph Rendering exercise highlights the importance of load balancing in parallel programming. As illustrated in Figure 4.6, the left portion of the graph has been rendered, while the right portion is still being processed. This imbalance occurs because the workload for each thread is not evenly distributed. The left part of the graph contains fewer pixels within the Mandelbrot set, resulting in less time required for the threads rendering this area. Consequently, although the threads responsible for the left part have completed their tasks, they must wait for the other threads to finish. This imbalance prevents the program from achieving optimal speedup and also reduces the improvement of adding extra threads. This is also the reason why in Figure 4,4, the speedup trends have a spike when the number of threads used is 4, as using 4 threads will result in a generally evenly partitioned graph.

4.3 Student Selection

Figures 4.7 and 4.8 show the weak scaling graph and speedup graph for the Student Selection exercise on the PC and the MacBook laptop. Figure 4.4 is plotted with data

from Figures A.5 and A.6. Figure 4.7 is plotted with the weak scaling data (the way to collect data is introduced in section 3.3). The horizontal axis of Figure 4.7 means the number of threads used with corresponding data size (1 means 1 thread with 100000 Student instances)

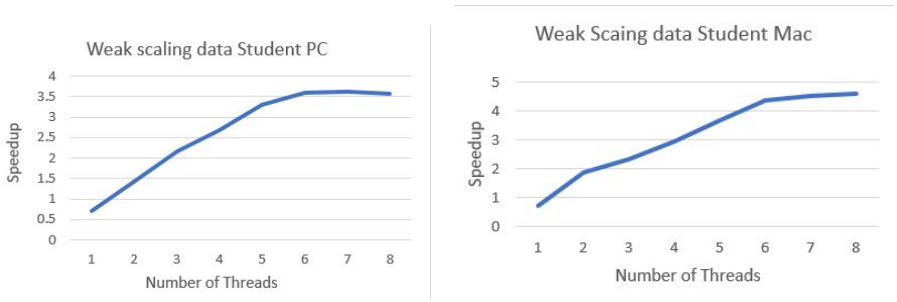

Figure 4.7: Weak scaling graph Student Selection

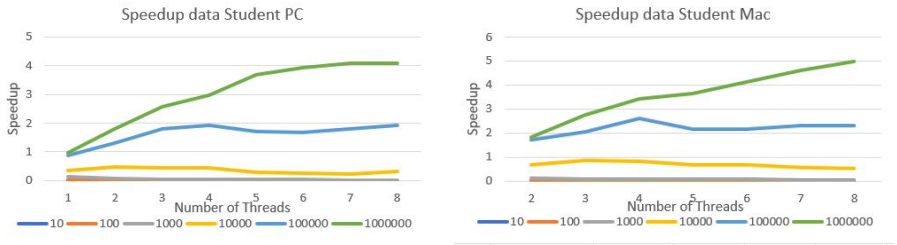

Figure 4.8: Speedup graph Student Selection

Speedup: From Figure 4.8 we can observe that the program starts to gain performance improvement when the data size is 100000. Though this data set is pretty large and may not be applicable in the real world, it still shows the potential improvement of applying parallelism.

Scalability: According to Figure 4.8, we can find when increasing the number of threads used and the data size, the speedup will increase as well, which indicates that the exercise is scalable.

Figure 4.7 shows that for up to 6 threads, the speedup increases linearly. After the number of threads used exceeds 6, the speed up increases at a much slower rate, meaning the exercise shows good weak scaling until the number of threads used is larger than 6.

Figure 4.8 indicates that when the data size is greater or equal to 100000 words, adding more threads will bring an increase in speedup. This shows that when the exercise takes data size that is larger than 100000 words, it will show good strong scaling performance.

Style of parallelism: This exercise applies data parallelism, as data is distributed among threads, each thread takes different data and does the same work on their input.

Speedup Visualization: This exercise shows speedup by printing out the calculated numerical speedup.

Interesting: According to Table 4.3, the average score of interesting in the Scrabble exercise is 5.65 out of 10, proving that students do not think this is a boring exercise, but not interesting enough at the same time. The score of 5,65 is close to 6, which is the level we consider as interesting, as we design this exercise to be an interesting one, students' opinion does not differ from ours a lot.

Difficulty: From table 4.3, the difficulty score of 4.35 out of 10 suggests that students think it is relatively easy to find a solution for this exercise, which matches our expected level of difficulty when designing this exercise.

Difficulty to Verify: The difficulty to verify score is 5.29 out of 10 from Table 4.3, implying that students find it moderate difficulty to verify their implementation. This differs from our expectation, as we are expecting this exercise should be easy to verify. Students may think it is not easy to build a test set that covers all the cases of the input, as input may contain special characters rather than just numbers and letters.

Interesting	Difficulty	Difficulty to Verify
5.65	4.35	5.29

Table 4.3: Survey result of the Student Selection exercise

4.4 Matrix Multiplication

Figures 4.9 and 4.10 show the weak scaling graph and speedup graph for the Mandelbrot Graph Rendering exercise on the PC and the MacBook laptop. Figure 4.10 is plotted with data from Figures A.7 and A.8. Figure 4.9 is plotted with the weak scaling data (the way to collect data is introduced in section 3.3). The horizontal axis of Figure 4.9 means the number of threads used with corresponding data size (1 means 1 thread with 1000 max-iterations)

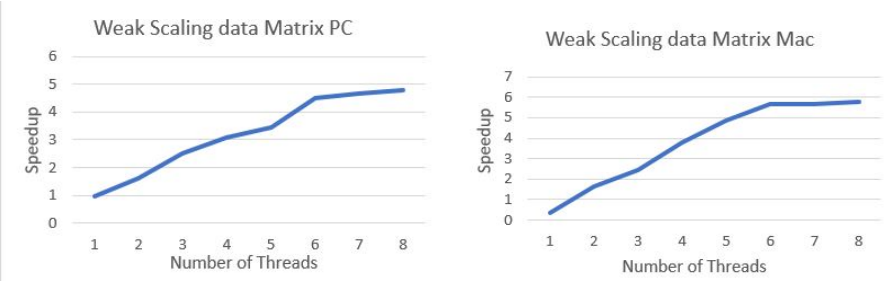

Figure 4.9: Weak scaling graph Matrix Multiplication

Speedup: From Figure 4.10, we can find out that on both platforms, the program started to gain speedup greater than 1 when the data size is greater or equal to 128 (128 * 128 matrix). This exercise shows performance improvement by applying parallelism with relatively small data size.

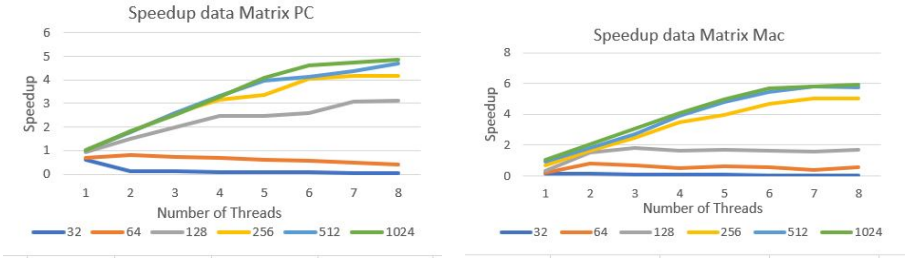

Figure 4.10: Speedup graph Matrix Multiplication

Scalability: According to Figure 4.10, we can find when increasing the number of threads used and the data size, the speedup will increase as well, which indicates that the exercise is scalable.

Figure 4.9 shows that for up to 6 threads, the speedup increases linearly. After the number of threads used exceeds 6, the speed up increases at a much slower rate, meaning the exercise shows good weak scaling until the number of threads used is larger than 6.

Figure 4.10 illustrates that when the data size is greater or equal to 128 * 128 Matrix, adding more threads will bring an increase in speedup. This shows that when the exercise multiplying 128 * 128 or larger Matrices, it will show good strong scaling performance.

Style of parallelism: When multiplying matrix A and B with size N * N, the program will split the matrix A into n parts (n is the number of threads), and each thread will multiply an (N/n) * N matrix with B to get a part of the result matrix. In this situation, input data is distributed among threads and different threads are doing the same work, meaning this program is following the data parallelism style.

Speedup Visualization: This exercise shows speedup by printing out the calculated numerical speedup.

Interesting: According to Table 4.4 the average score of interesting in the Scrabble exercise is 5.82 out of 10, proving that students found the Matrix Multiplication exercise not very interesting. This matches our expectations mentioned in the design section. Most university computer science students should be familiar with matrix multiplication as they should have studied this during math lessons in high school. This familiarity could make the exercise seem less novel and interesting.

Difficulty: From table 4.4, the difficulty score of 5.47 out of 10 suggests that students consider the difficulty of this exercise to be medium-level. This matches our expectations mentioned in the design section, students may have encountered matrix multiplication in previous courses which could reduce their feeling of the level of difficulty.

Difficulty to Verify: The difficulty to verify score is 4.17 out of 10 from Table 4.4, meaning that students believe this exercise is generally easy to verify which is the same as our opinion. Students can verify this exercise by simply coming out with 2 matrices and multiplying the matrices themselves then comparing the result with their solution's output.

Interesting	Difficulty	Difficulty to Verify
5.82	5.47	4.17

Table 4.4: Survey result of the Matrix Multiplication exercise

4.5 Morse code Encryption

Figures 4.11 and 4.12 show the weak scaling graph and speedup graph for the Mandelbrot Graph Rendering exercise on the PC and the MacBook laptop. Figure 4.12 is plotted with data from Figures A.9 and A.10. Figure 4.11 is plotted with the weak scaling data (the way to collect data is introduced in section 3.3). The horizontal axis of Figure 4.11 means the number of threads used with corresponding data size (1 means 1 thread with 10000 words)

Figure 4.11: Weak scaling graph Morse code Encryption

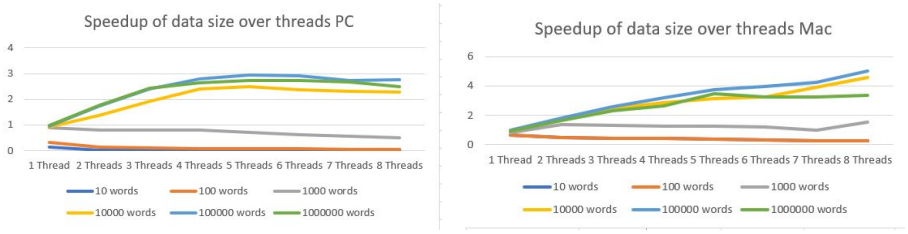

Figure 4.12: Speedup graph Morse code Encryption

Speedup: From Figure 4.12, when the data size is 10000 and the program runs with 2 or more threads, the Morse code Encryption exercise achieves a speedup greater than 1 on PC. When the data size is 1000 and running on 2 or more threads, it also gains speedup greater than 1 on MacBook. The result shows a possible improvement brought by parallelism. It also shows that the same program can start gaining speedup with different data sizes on different platforms. The reasons why Macbook can achieve speedup with smaller data size are listed:

1. The CPU on the different platforms have different cache sizes. The cache size of MacBook CPU may be smaller so it takes more time to compute 1000 words data size, and because the parallel version split the data into smaller chunks, it utilizes the cache size.

2. The 2 platforms have different operating systems, and the way that the operating systems allocate resources may be different.

Scalbility: According to Figure 4.12, we can find when increasing the number of threads used and the data size, the speedup will increase as well, which indicates that the exercise is scalable.

Figure 4.11 reveals that for up to 6 threads, the speedup is increasing linearly, suggesting that the weak scaling performance of the parallel solution is generally satisfactory for up to 6 threads.

Figure 4.12 presents the speedup data graph, on the PC, when the data size is greater or equal to 10000 words, adding more threads will increase speedup. On the MacBook laptop, when the data size is greater or equal to 10000 words, adding more threads will increase speedup. Possible reasons that cause the data from 2 platforms to be different are explained in the Speedup criteria analysis paragraph. When the data size is 1000 words, the rate of increase in speedup on MacBook is slow and has not achieved a speedup greater than 2. We consider the exercise starts to show good strong scaling performance when the data size is greater or equal to 10000.

Style of parallelism: This excise is similar to the Scrabble exercise, input data is divided into smaller chunks and each thread is responsible for computing a chunk of data. All threads are doing the same task which is data parallelism.

Speedup Visualization: This exercise shows speedup by printing out the calculated numerical speedup.

Interesting: According to Table 4.3 the average score of interesting of the Scrabble exercise is 7.52 out of 10, proving that students found the Morse code Encryption exercise attractive. The survey result matches our opinion: this exercise is an interesting exercise. This exercise is not only a real-world application but also provides a taste of cryptography to students which can make them feel interested.

Difficulty: From table 4.3, the difficulty score of 6.58 out of 10 suggests that students believe this exercise has a moderate level of difficulty. We have the same opinion on the level of difficulty this exercise should have. Students have to learn how to use the dictionary data structure to solve this exercise and they may not be familiar with that. In addition, They have to consider how to merge the result properly, all these can become challenges when they implement their solution, causing them to rate this exercise as 6.5/10.

Difficulty to Verify: The difficulty to verify score is 5.88 out of 10 from Table 4.3, implying that students find it not very easy to come up with test cases to check their solutions' correctness. As we mentioned in the design section of this question, the difficulty to verify this question highly depends on the complexity of the filter. Some students may realize this situation and rate a high score.

Interesting	Difficulty	Difficulty to Verify
7.52	6.58	5.88

Table 4.5: Survey result of the Morse code Encryption exercise

4.6 Array Processing

Figure 4.13 show the speedup data of the Array Processing exercise with different input size. Figure 4.13 is plotted with data from Figures A.9 and A.10. As this exercise applies task parallelism, the number of threads used is fixed to three (each thread responsible for calculating max, min and sum respectively).

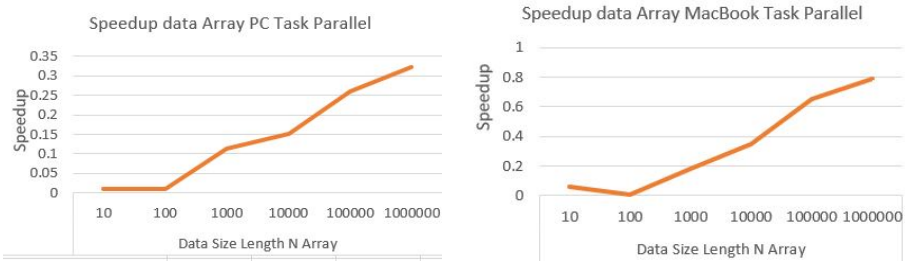

Figure 4.13: Speedup graph Array Processing

Speedup: From Figure 4.13, the program cannot gain a speedup greater than 1 on both platforms. Though the trend shows that increasing data size will increase the speedup, an array of size 1000000 is a really large data size in a real-world application, continue to increase the data size may finally bring a performance improvement but the data size may become billions or more, which is not worth to do so.

In addition, the possible reason cause the exercise does not show a speedup greater than 1 with a significantly large amount of data is the simplicity of calculation. Calculating the max, min and the sum of a list of numbers is simple, so even if we add more threads and increase the data size, the time saved by applying parallelism can not overcome the parallel overhead.

Scalbility: As the exercise is running with a fixed number of threads, it is not able to talk about the strong scaling and weak scaling of this exercise. However, according to Figure 4.13, we can see that increasing the data size will bring an increase in speedup, meaning this exercise is salable.

Style of parallelism: This exercise differs from other exercises, in this exercise, threads all take the same data and do different jobs (threads responsible for calculating max, min and sum respectively), this matches the concept of task parallelism.

Speedup Visualization: This exercise shows speedup by printing out the calculated numerical speedup.

Interesting: According to Table 4.6 the average score of interesting of the exercise is 3 out of 10, considering this exercise is an easy one, which means lacks challenge and engagement. However, this exercise is designed to be a warm-up, and a low score of interest is acceptable.

Difficulty: From table 4.6, the difficulty score of 2.17 out of 10 suggests that students believe this exercise is very easy. The students' response matches our aspect, the simplicity of this exercise may cause the students to feel not interested. In another

way, as a warm-up exercise, a simple exercise can help students build confidence, and encourage them to overcome later challenges in the exercise set.

Difficulty to Verify: The difficulty to verify score is 2.34 out of 10 from Table 4.6, implying that students find it very easy to come up with test cases to check their solutions' correctness. This result matches our expectations, as students can make up some arrays and calculate the max, min and sum by themselves. Calculating the max, min and sum of a small-size array is easy, so this exercise should be easy to verify.

Interesting	Difficulty	Difficulty to Verify
3	2.17	2.34

Table 4.6: Survey result of the Morse code Encryption exercise

4.7 Extra Findings

This section will talk about some common characteristics of the speedup data for all exercises.

Speedup trend: From the speedup data graphs of both platforms, we can find out that when the data size is small, and the program cannot get a speedup greater than 1, by adding more and more threads, the speedup is decreasing. Because when the data size is small when adding more threads, extra effort will be taken to create and destroy the thread instances. As the data size is small, the time taken to create new threads is much larger than the time reduction bring by parallelism, thus, causing the overall runtime to become larger, reducing the speedup.

Super-linear speedup [9]: From the speedup data, we can observe some cases where when running the program with n threads, the speedup is greater than n. Intuitively, the maximum speedup that can be achieved when running with n threads is n, the case that we obtain a speedup greater than n is called super-linear speedup. The following points might be the reason cause the super-linear speedup:

1. Cache effects [9]: When a problem is divided across multiple processing units, each smaller subproblem may fit better into the processor's cache, leading to more efficient use of the cache. This can improve the overall processing speed and result in a super-linear speedup.
2. Improved resource utilization [8]: In some cases, parallelizing a problem can lead to more effective utilization of available resources, such as processors, memory, or I/O bandwidth. This can enable better performance than would be expected from a linear scaling of the problem.
3. Algorithmic improvements [1]: Parallelizawtion may lead to the discovery of more efficient algorithms or the opportunity to exploit problem-specific properties that were not apparent in the sequential version. These improvements can result in a super-linear speedup. Moreover, based on the finding from the parallel version, it can improve the sequential version algorithm as well.

Chapter 5

Conclusion and Reflection

5.1 Conclusion

In this project, our primary objective was to develop and evaluate parallel programming exercises for novice programmers, with a focus on the effectiveness of these exercises in teaching parallel programming concepts and their ability to engage students. Our research question centred around identifying the characteristics of effective and engaging parallel programming exercises for beginners. To achieve this goal, we developed a set of six parallel programming exercises and conducted a thorough evaluation, analyzing various criteria such as performance, scalability, and student feedback.

Based on tested data and survey results, we believe our exercise set covers most of the basic parallel programming concepts in a variety of ways. In addition, through the evaluation of the survey result, we can confirm that our exercise set has covered all range of difficulties and most of the exercises are considered as interesting by students' responses.

This set of parallel programming exercises offers significant benefits to both teachers and students. For teachers, the exercises provide a comprehensive and diverse collection of problems that can demonstrate essential parallel programming concepts. The exercises can serve as an effective assessment tool and save time in preparing course materials, while also providing opportunities to customize the content based on students' needs. For students, the exercises deliver a hands-on experience with parallel programming, helping them develop problem-solving skills and construct a "parallel thinking" mindset. Moreover, these exercises enhance motivation and engagement in the subject matter, build confidence in facing parallel programming challenges, and ultimately prepare students for future academic and professional achievements in the computer science field.

5.2 Reflection

Upon reflecting on the project, we can identify several strengths and limitations of this project.

5.2.1 Strengths

1. A diverse set of parallel programming exercises was developed, covering most of the basic parallel programming concepts and difficulty levels. This allowed the novice to have a good learning experience.
2. A comprehensive evaluation was employed, including performance metrics, scalability analysis, and student feedback. This provided a general view of the effectiveness of the exercises. Also, allowing teachers to pick the exercise that fits the class situation based on the evaluation of the exercises.
3. The student feedback allowed us to identify areas of improvement, tailor the exercises to better suit students' needs, and ensure that the exercises were engaging and interesting.
4. The exercise set contains exercises which demonstrate parallel concepts in a dynamic visible way, which can show students the parallel concepts in an intuitive way, allowing students to understand the concepts better.

5.2.2 Limitations

1. The evaluation was based on a limited sample of students, which might not be fully representative of the population of novice parallel programmers. This could affect the generalizability of our findings.
2. The project focused on Java and the Thread class for implementing parallelism. Other programming languages, frameworks, or libraries might offer different insights and results.
3. The exercises that the project designed only demonstrate the basic parallel concepts, and high-level parallel programming concepts like data races, locks and synchronization are not contained in the exercise set.
4. The size of the exercises is quite small, as there are only six exercises, and they may not cover all the cases of parallel concepts.

5.3 Future works

This section will list possible future works that could be done to improve the effectiveness of the exercise set.

1. Expand the sample size of students participating in the evaluation to improve the generalizability of the subjective criteria evaluation.
2. Introduce additional assessment methods, such as quizzes or tests, to further evaluate the learning outcomes and comprehension of students.
3. Evaluate the long-term impact of the exercises on students' parallel programming skills, including their ability to apply these concepts to real-world problems and projects.

4. Design and develop more exercises that can be added to the exercise set, achieving higher coverage of parallel concepts.
5. Develop a tool that could perform an input space search for the exercises, allowing teachers and students to discover interesting input points (e.g. the point that the exercise shows speedup greater than 1, ranges that the exercise shows good or bad scalability, etc.).

Bibliography

[1] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] Richard Brown, Elizabeth Shoop, Joel Adams, Curtis Clifton, Mark Gardner, Michael Haupt, and Peter Hinsbeeck. Strategies for preparing computer science students for the multicore world. In *Proceedings of the 2010 ITiCSE Working Group Reports*, ITiCSE-WGR '10, page 97–115, New York, NY, USA, 2010. Association for Computing Machinery.

[3] Angela Carbone, John Hurst, Ian Mitchell, and Dick Gunstone. Principles for designing programming exercises to minimise poor learning behaviours in students. In *Proceedings of the Australasian conference on Computing education*, pages 26–33, 2000.

[4] Davi Jose Conte, Paulo Sergio Lopes de Souza, Guilherme Martins, and Sarita Mazzini Bruschi. Teaching parallel programming for beginners in computer science. In *2020 IEEE frontiers in education conference (FIE)*, pages 1–9. IEEE, 2020.

[5] Mohammad Amin Kuhail, Spencer Cook, Joshua Neustrom, and Praveen Rao. Teaching parallel programming with active learning. pages 369–376, 05 2018.

[6] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[7] Gustavo Pinto. Simple mandelbrot demo.

[8] T Rauber and G Rünger. Parallel programming: For multicore and cluster systems.[sl]: Springer science & business media, 2013. 2013.

[9] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. Superlinear speedup in hpc systems: Why and when? In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898, 2016.

[10] David Rotman. We're not prepared for the end of moore's law, Apr 2021.

[11] Ravikiran A S. An introduction to thread in java.

[12] Aamir Shafi, Aleem Akhtar, Ansar Javed, and Bryan Carpenter. Teaching parallel programming using java. In *2014 Workshop on Education for High Performance Computing*, pages 56–63, 2014.

[13] Yuriy Sitsylitsyn. Methods and tools for teaching parallel and distributed computing in universities: a systematic review of the literature. In *ICHTML 2020: SHS Web of Conferences*, number 75. EDP Sciences, 2020.

[14] Wikipedia. Mandelbrot set.

Appendix A

Speedup data table

Speedup Using Thread PC						
	Data Size					
Thread Number	10 words	100 words	1000 words	10000 words	100000 words	1000000 words
1 Thread	0.0347	0.2193	0.5741	0.9326	0.9493	0.9893
2 Threads	0.0073	0.0737	0.6621	1.5385	1.8941	1.9439
3 Threads	0.0061	0.0631	0.6215	1.9362	2.6179	2.774
4 Threads	0.0046	0.0569	0.6275	2.435	3.3912	3.5497
5 Threads	0.0028	0.0399	0.4747	2.551	4.1106	4.3971
6 Threads	0.0037	0.0439	0.4672	2.83	4.5351	4.7228
7 Threads	0.0027	0.0352	0.4204	3.1489	4.5762	4.9733
8 Threads	0.0021	0.0287	0.3316	3.3088	4.799	5.132

Figure A.1: Speedup data of Scrabble from PC

Speedup Using Thread Mac						
	Data Size					
Thread Number	10 words	100 words	1000 words	10000 words	100000 words	1000000 words
1 Thread	0.2159	0.1618	0.711	0.7657	0.99	1
2 Threads	0.0054	0.0816	0.8125	1.7354	1.944	1.9597
3 Threads	0.0054	0.0764	0.9433	1.8404	2.7725	2.783
4 Threads	0.0051	0.0626	0.82	2.1177	3.4721	3.6973
5 Threads	0.0049	0.0476	0.7688	2.2332	4.0951	4.5661
6 Threads	0.004	0.0429	0.6044	2.2313	4.6391	5.2745
7 Threads	0.0036	0.0348	0.643	2.2442	5.1462	5.6762
8 Threads	0.0028	0.0301	0.4753	2.4347	5.5706	6.12

Figure A.2: Speedup data of Scrabble from MacBook

Speedup on PC						
	Max iterations					
Number of Threads	1000	2000	3000	4000	5000	6000
1	0.87	0.93	0.95	0.96	0.97	0.98
2	1.06	1.17	1.12	1.13	1.14	1.15
3	1.71	1.76	1.64	1.63	1.62	1.63
4	1.81	2.01	1.92	1.93	1.92	1.95
5	1.72	1.79	1.81	1.82	1.83	1.84
6	1.82	1.98	1.87	1.94	1.94	1.98
7	1.91	2.11	2.02	2.04	2.06	2.08
8	1.96	2.35	2.24	2.26	2.27	2.30

Figure A.3: Speedup data of Mandelbrot PC

Speedup on Mac						
	Max iterations					
Number of Threads	1000	2000	3000	4000	5000	6000
1	0.901	0.943	0.962	0.971	0.977	0.982
2	1.017	1.103	1.122	1.135	1.088	1.201
3	1.225	1.299	1.372	1.352	1.225	1.319
4	1.365	1.329	1.465	1.458	1.505	1.555
5	1.324	1.281	1.458	1.403	1.473	1.521
6	1.340	1.356	1.491	1.495	1.495	1.580
7	1.393	1.440	1.526	1.599	1.518	1.642
8	1.471	1.473	1.547	1.606	1.635	1.756

Figure A.4: Speedup data of Mandelbrot Mac

Speed Up Using Thread PC						
	Data Size N Students					
Number of Threads	10	100	1000	10000	100000	1000000
1 Thread	0.0167	0.0230	0.1474	0.3455	0.8735	0.9723
2 Threads	0.0011	0.0072	0.0695	0.4712	1.3016	1.8070
3 Threads	0.0007	0.0053	0.0498	0.4264	1.7953	2.5672
4 Threads	0.0008	0.0041	0.0380	0.4306	1.9364	2.9774
5 Threads	0.0007	0.0034	0.0295	0.2736	1.7004	3.6879
6 Threads	0.0004	0.0032	0.0312	0.2703	1.6722	3.9206
7 Threads	0.0003	0.0029	0.0210	0.2124	1.7872	4.0979
8 Threads	0.0003	0.0027	0.0196	0.3153	1.9278	3.6936

Figure A.5: Speedup data of Student Selection PC

Speed Up Using Thread MacBook						
	Data Size N Students					
Number of Threads	10	100	1000	10000	100000	1000000
1 Thread	0.0487	0.1734	0.0591	0.6087	0.8637	0.9207
2 Threads	0.0027	0.0192	0.1227	0.6761	1.7245	1.8196
3 Threads	0.0020	0.0116	0.0861	0.8623	2.0421	2.7381
4 Threads	0.0021	0.0107	0.0904	0.8333	2.5984	3.4114
5 Threads	0.0014	0.0088	0.0688	0.6862	2.1598	3.6585
6 Threads	0.0012	0.0077	0.0728	0.6582	2.1750	4.1251
7 Threads	0.0013	0.0065	0.0597	0.5701	2.3180	4.6049
8 Threads	0.0011	0.0059	0.0478	0.5188	2.2892	4.9947

Figure A.6: Speedup data of Student Selection Mac

Speed Up Using Thread PC						
	Data Size (N*N) matrix					
Number of Threads	32	64	128	256	512	1024
1 Thread	0.6	0.67	0.93	1	1.01	1
2 Threads	0.13	0.8	1.51	1.8	1.79	1.83
3 Threads	0.11	0.75	1.99	2.59	2.6	2.52
4 Threads	0.1	0.67	2.48	3.15	3.33	3.27
5 Threads	0.08	0.62	2.49	3.35	3.96	4.09
6 Threads	0.08	0.57	2.6	4.06	4.15	4.63
7 Threads	0.06	0.5	3.08	4.16	4.36	4.72
8 Threads	0.06	0.41	3.11	4.16	4.7	4.85

Figure A.7: Speedup data of Matrix Multiplication PC

Speed Up Using Thread Mac						
	Data Size (N*N) matrix					
Number of Threads	32	64	128	256	512	1024
1 Thread	0.17	0.19	0.35	0.7	0.95	1.04
2 Threads	0.14	0.83	1.53	1.64	1.79	2.03
3 Threads	0.09	0.66	1.81	2.49	2.69	3.06
4 Threads	0.08	0.48	1.66	3.48	3.89	4.06
5 Threads	0.07	0.63	1.71	3.96	4.78	4.97
6 Threads	0.05	0.56	1.63	4.7	5.45	5.67
7 Threads	0.05	0.38	1.6	5.03	5.79	5.83
8 Threads	0.05	0.55	1.68	5.06	6.63	6.93

Figure A.8: Speedup data of Matrix Multiplication Mac

Speedup Using Thread PC						
	Data Size					
Thread Number	10 words	100 words	1000 words	10000 words	100000 words	1000000 words
1 Thread	0.139	0.33	0.881	0.921	0.964	0.973
2 Threads	0.011	0.147	0.808	1.387	1.722	1.752
3 Threads	0.008	0.103	0.795	1.912	2.39	2.423
4 Threads	0.007	0.084	0.81	2.397	2.801	2.641
5 Threads	0.006	0.069	0.725	2.49	2.936	2.736
6 Threads	0.005	0.075	0.63	2.355	2.897	2.716
7 Threads	0.005	0.055	0.563	2.302	2.726	2.66
8 Threads	0.006	0.049	0.497	2.285	2.756	2.489

Figure A.9: Speedup data of Morse code from PC

Speedup Using Thread Macbook						
		Data Size				
Thread Number	10 words	100 words	1000 words	10000 words	100000 words	1000000 words
1 Thread	0.6431	0.6431	0.7883	0.9411	0.9746	0.9364
2 Threads	0.4516	0.4516	1.3528	1.773	1.817	1.664
3 Threads	0.4446	0.4446	1.3004	2.4592	2.5753	2.3048
4 Threads	0.398	0.398	1.2348	2.8558	3.1661	2.6186
5 Threads	0.3456	0.3456	1	3.1416	3.7285	3.4525
6 Threads	0.3284	0.3284	1.1938	3.2321	3.958	3.2203
7 Threads	0.2629	0.2629	0.999	3.898	4.2166	3.2315
8 Threads	0.2458	0.2458	1.5278	4.5753	5.0069	3.3776

Figure A.10: Speedup data of Morse code from MacBook

Speed Up Using Thread PC Task Parallel						
	Data Size Length N Array					
Number of Threads	10	100	1000	10000	100000	1000000
3 Threads	0.0097	0.0095	0.113	0.1506	0.26	0.3216

Figure A.11: Speedup data of Array Processing from PC

Speed Up Using Thread MacBook Task Parallel						
	Data Size Length N Array					
Number of Threads	10	100	1000	10000	100000	1000000
3 Threads	0.0574	0.005071	0.178538	0.348883	0.651919	0.79213

Figure A.12: Speedup data of Array Processing from MacBook

Appendix B

Participants' information sheet

Participant Information Sheet Project title: Novice-Friendly Parallel Programming
Principal investigator: Professor Murray Cole
Researcher collecting data: Haoshi Wang
Funder (if applicable): N/A

This study was certified according to the Informatics Research Ethics Process, RT number 2021/24985. Please take time to read the following information carefully. You should keep this page for your records.

Who are the researchers?
Haoshi Wang is an Undergraduate/MSc student in the School of Informatics, undertaking this study as part of their Honours/MSc project. The project is supervised by Professor Murray Cole.

What is the purpose of the study?
The project involves the design and evaluation of a set of parallel programming examples, intended to be useful to novice parallel programmers.

Why have I been asked to take part?
As a programmer, we seek your views on various aspects of the designed examples, and how they might be perceived by learners.

Do I have to take part?
No – participation in this study is entirely up to you. You can withdraw from the study at any time without giving a reason, until the project has been submitted. After this point, personal data will be deleted and anonymised data will be combined such that it is impossible to remove individual information from the analysis. Your rights will not be affected. If you wish to withdraw, contact the PI. We will keep copies of your original consent, and of your withdrawal request.

What will happen if I decide to take part?
You will complete an electronic or paper questionnaire, asking for your opinions on properties of a collection of parallel programs (for example, attractiveness, difficulty, intuitiveness) as they might be perceived by novice parallel programmers. You will complete this in your own time, without supervision. You are not being asked to write

the programs, merely to assess them.

Are there any risks associated with taking part?
There are no significant risks associated with participation.

Are there any benefits associated with taking part?
No, though we hope you will find the programs interesting!

What will happen to the results of this study?
The results of this study will be summarised in the project report and presentation. Quotes or key findings will be anonymized: We will remove any information that could, in our assessment, allow anyone to identify you. With your consent, information can also be used for future research. Your data may be archived for a maximum of 1 year (in practice, only until the project has been submitted). All potentially identifiable data will be deleted within this timeframe if it has not already been deleted as part of anonymization.

Data protection and confidentiality.
Your data will be processed in accordance with Data Protection Law. All information collected about you will be kept strictly confidential. Your data will be referred to by a unique participant number rather than by name. Your data will only be viewed by the researcher/research team: Haoshi Wang and supervisor Professor Murray Cole. All electronic data will be stored on a password-protected encrypted computer, on the School of Informatics' secure file servers, or on the University's secure encrypted cloud storage services (DataShare, ownCloud, or Sharepoint) and all paper records will be stored in a locked filing cabinet in the PI's office. Your consent information will be kept separately from your responses in order to minimise risk.

What are my data protection rights?
The University of Edinburgh is a Data Controller for the information you provide.You have the right to access information held about you. Your right of access can be exercised in accordance Data Protection Law. You also have other rights including rights of correction, erasure and objection. For more details, including the right to lodge a complaint with the Information Commissioner's Office, please visit www.ico.org.uk. Questions, comments and requests about your personal data can also be sent to the University Data Protection Officer at dpo@ed.ac.uk.

Who can I contact?
If you have any further questions about the study, please contact the lead researcher, Professor Murray Cole (mic@inf.ed.ac.uk). If you wish to make a complaint about the study, please contact inf-ethics@inf.ed.ac.uk. When you contact us, please provide the study title and detail the nature of your complaint.

Updated information.
If the research project changes in any way, an updated Participant Information Sheet will be made available on http://web.inf.ed.ac.uk/infweb/research/study-updates.

Alternative formats.
To request this document in an alternative format, such as large print or on coloured paper, please contact Professor Murray Cole (mic@inf.ed.ac.uk).

General information.
For general information about how we use your data, go to: edin.ac/privacy-research

Appendix C

Participants' consent form

Project title: Novice-Friendly Parallel Programming
Principal investigator (PI): Professor Murray Cole
Researcher: Student Haoshi Wang
PI contact details: mic@inf.ed.ac.uk

By participating in the study you agree that:
I have read and understood the Participant Information Sheet for the above study, that I have had the opportunity to ask questions, and that any questions I had were answered to my satisfaction.
• My participation is voluntary, and that I can withdraw at any time without giving a reason. Withdrawing will not affect any of my rights.
• I consent to my anonymised data being used in academic publications and presentations.
• I understand that my anonymised data will be stored for the duration outlined in the Participant Information Sheet.

Please tick yes or no for each of these statements.
1. I agree to take part in this study.
Yes No

Name of person giving consent – Date – Signature
Name of person taking consent – Date – Signature
Haoshi Wang