

# **Parallel Quicksort**

*CMP202 2023/24 Term 2*

*Mini-Project: Mini-Project 1, CPU*

**Jack Laundon**

**2202274**

## **Contents**

---

1	Introduction .....	1
2	Application Overview .....	1
3	Parallelization Strategy and Implementation .....	1
3.1	Thread Management .....	1
3.2	Implementation Specifics .....	2
4	Performance Evaluation .....	2
5	Conclusion.....	4

# 1 INTRODUCTION

---

The task chosen for this project was to parallelise a quicksort, a widely used sorting algorithm. The quicksort algorithm is an effective method of sorting data but could be more efficient if it benefits from parallelisation. This program will also demonstrate some ineffective methods of parallelisation, to convey the adverse effects of parallelisation, such as thread overheads, so that the increase in speed gained from an efficient parallel quicksort can be fully exhibited.

## 2 APPLICATION OVERVIEW

---

The quicksort was chosen to be parallelised as the quicksort lends itself to “task based” parallelism and “divide and conquer” parallelism, as the data to be sorted can be easily split into chunks (divide and conquer) and each chunk can be sorted in parallel (task based). This is what the parallel quicksort function does. A sequential, parallel, “thread” (where there is no parallelism except creating a new thread for each function call), and a “parallel\_for” (where the parallel sort is in a parallel\_for loop further creating threads) are used. The “thread” and “parallel\_for” sorts are used to demonstrate that the overheads created by threads can negate the benefits of parallelism. The “thread” sort is also run in a parallel\_for loop to further demonstrate the inefficiency. The features include:

- Five concurrent threads
  - Async launches as many threads as necessary and std::launch ensures each task runs on a new thread
- Two thread functions
  - When the task farm is running, another thread running at the same time is used to monitor and update the status of the farm using a condition variable to communicate with the thread
- Modification from lab materials
  - The quicksort is not from a lab, so this requirement does not apply
- Resource sharing
  - Mutexes are used in the task farm, a future is used in the parallel quicksort, a condition variable is used to synchronise access to the progress function, threads.join() in the task farm synchronises threads, an atomic variable is used in the partition function
- Thread communication
  - Condition variable in the task farm signals to update the progress counter
- Compare to sequential
  - The results from the sequential and parallel sort are output to the console
- Vary thread number/thread groups
  - The chunk size can be modified

## 3 PARALLELIZATION STRATEGY AND IMPLEMENTATION

---

### 3.1 Thread Management

The parallel quicksort uses std::async to execute tasks concurrently and uses “launch” with async to ensure each task is run on a new thread. Mutexes are used in the task farm to ensure safe access to shared resources for each thread and avoid race conditions. A condition variable is also used in the task farm to safely communicate with and update the progress function, running in another thread. The “count” variable in the partition function is atomic to avoid unpredictable behavior as, without it being atomic, the sorting function finished extremely prematurely and thus gave incorrect results. A future is used in the parallel quicksort function to safely collect the

results. The `thread.join()` function is used in the task farm and the parallel quicksort to ensure all threads complete their tasks before continuing.

### 3.2 Implementation Specifics

The following implementation specifics are used in the program:

- **Mutexes**
  - Unique locks are used to ensure thread safety – each time a shared resource is accessed, the mutex is locked to avoid race conditions.
- **Condition Variables**
  - A condition variable is used to signal and notify the progress function in the task farm.
- **Atomic Operations**
  - The “count” variable in the partition function is atomic. This ensures that the variable updates correctly and safely so multiple threads can access it at once.

## 4 PERFORMANCE EVALUATION

---

The specifications in the program were:

- A data size ranging from 100 to 10,000,000, increasing by 100 each time.
- A chunk size that stayed at 100 for each test

See Figure 1 for a graph of the median average results, and Figure 2 for a graph of the mean average results.

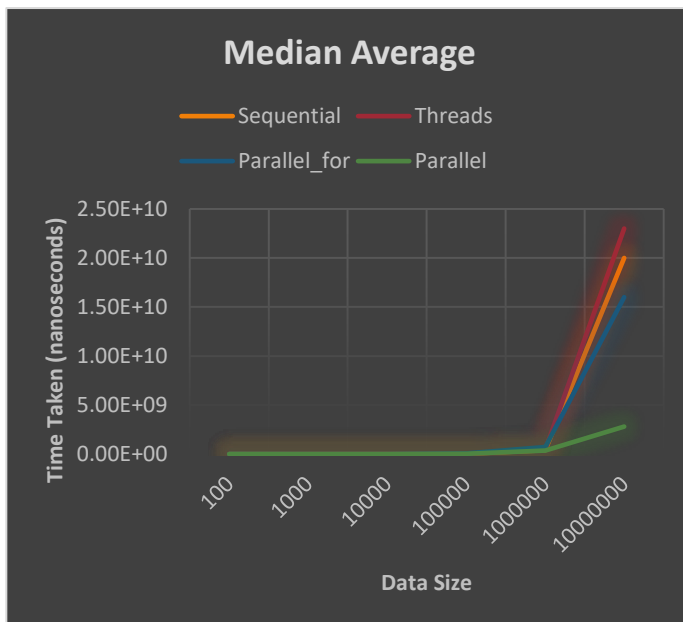


Figure 1 - The median average

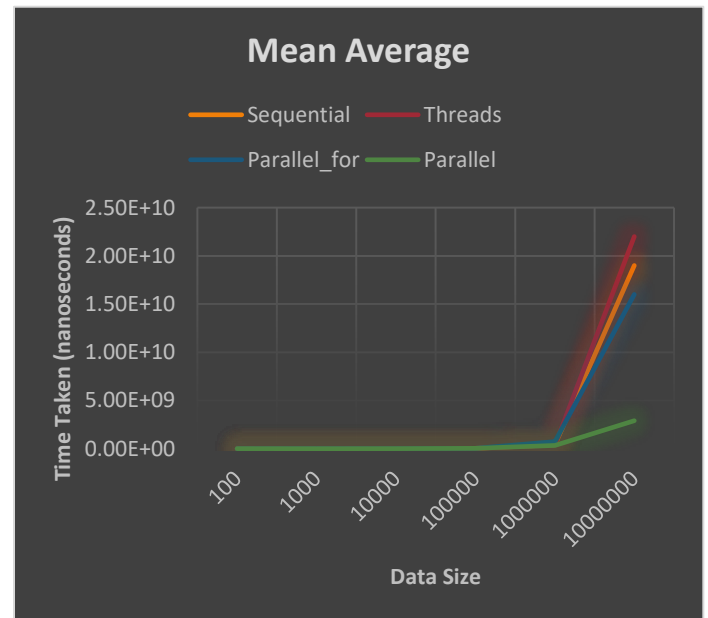


Figure 2 - The mean average

As shown in the graphs, the time taken for each sort stays very similar until the data size reaches one million. This is due to the overheads created outweighing the benefits of parallelism at smaller data sizes. When the size reaches one million, the benefits of parallelism begin to show as they are no longer being cancelled out by the overheads created by the threads. For the same reason, the “thread” and “parallel\_for” sort become dramatically slower when the data size reaches one million as the thread overheads significantly slow down the sorts. When

the data size reaches 10 million, the benefits of parallelism are clearly demonstrated, with an average speed up of 7x (median) and 6.5x (mean) for parallel to sequential. See Figure 3 for a graph of the median speed ups, and Figure 4 for the mean speed ups.

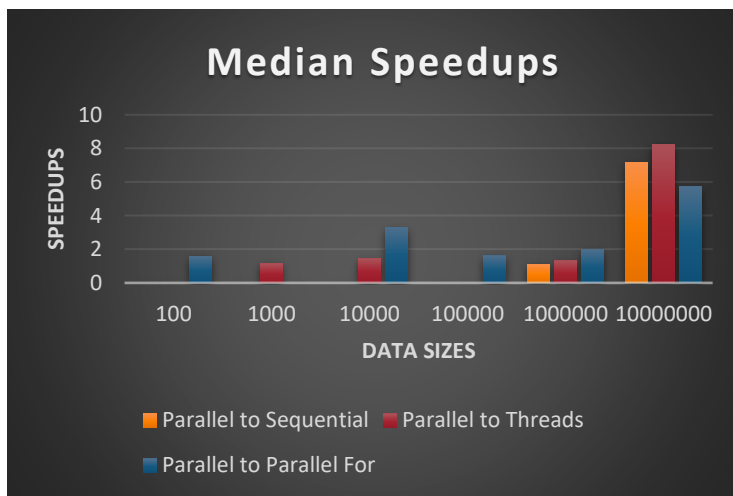


Figure 3 - Median speedups

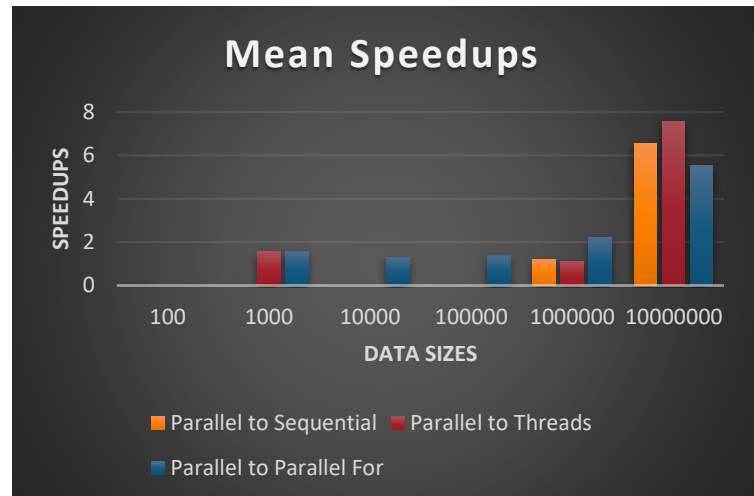


Figure 4 - Mean Speedups

The effectiveness of the parallel quicksort is good but could be better. The reason for this is because, although each chunk is sorted, the chunks are not in order. To rectify this, the chunks are merged by creating a vector of vectors and passing each chunk into the vector. The elements of the vector (the chunks) are then merged using `std::merge`. This decreases the speed of the sort and brings what was a 13x speedup down to the 6-7x. If there was a way that did not require merging the sort would be even faster, but at present the merge is necessary.

A challenge that was faced during the development of this program was the task farm getting stuck on an arbitrary task. This was addressed by adding mutexes before any work was carried out on the vectors, as the shared resources were causing the issues. Another challenge was, at seemingly random moments, the program would go out of bounds. Through use of the debugger and stepping through, this was found to be that the vectors were not getting cleared after each run of each sort. To rectify this, vectors are cleared at the end of each run and, for the parallel for loops, a new vector was initialised at the beginning of the loop.

If this project was repeated, some aspects could be done differently to improve efficiency. Firstly, a method to perform a parallel quicksort without needing to merge the chunks would increase the speed of the sort even more. Secondly, the optimal data size and chunk size was challenging to find, and this was done through trial and error. A more efficient method may exist and should be utilised if this project were to be repeated. Finally, a threshold could be introduced to limit the number of threads created in the parallel quicksort. This would perform the sort in parallel until a certain number of threads were created and the sort could then be done sequentially. This would combine the benefits of parallelism with managing the thread overheads to result in the most efficient solution possible.

## 5 CONCLUSION

---

In conclusion, this program successfully parallelizes a quicksort algorithm and demonstrates the benefits of parallelism over sequential programming. The program demonstrates a solution for parallelism as well as two solutions that would not work, to explain how thread overheads can cancel out the benefits of parallelism if not managed correctly, and outputs the results to be read and analysed by the user. With the aforementioned future work implemented, the program could be even more efficient in parallelising a quicksort algorithm.