# PARALLEL PRACTICAL 1

WLSGRA012 – Grant Wilson

# Introduction

The aim of the project is to test a parallel sorting algorithm versus a sequential algorithm. The program would run the program on 2 threads, 4 threads, 8 threads and 16 threads and compare this parallel timing with the sequential timing. We would expect a speed up of about on a duel core machine and a speed up of 4 on a quad core machine. We would expect the parallel algorithm to run the fastest on nightmare which has 16 cores.

# Methods

## Merge sort

I create two methods for merge sort. I created the compute() method that would recursively begin the parallelism. I also created the sequentialMerge() method which would join two ordered lists together. The compute() method was the branching method that created more and more threads until a sequantial thresh hold was reached. From this step it would sort each individual branch (list) until the entire sub list was sorted. SequentialMerge() would then be called to join these branches together to for a larger branch. This would happen until there is only a single sorted array.

I had lots of trouble making my code faster. This was when I realised that I could branch the array recursively and not create 2 new branching arrays. I could fix this by sorting the arrays and swapping values around the main array instead of creating new arrays. This created high computation overhead.

Another huge problem I had was that my parallel code ran about 200 times slower than my sequential code. I restarted my program. I created a sequential helper instead of using Arrays.sort(). This was very frustrating as I had to re plan all my unit testing as well.

I know my code works as I tested the merge function with odd and even array sizes. I also created a unit test for testing if my output was equal to arrays.sort output. I also created a new junit test that tested that my parallel code ran faster than my sequential code on a duel core machine.

## Quick Sort

For the quick sort method I created one algorithm. This is compute(). In this method I partitioned the main array into two smaller sub arrays via a pivot. I would then invoke these two sub pivots to do the same and create 2 more sub arrays. This would carry on until all sub arrays are sorted via a pivot until a sequential cut off is reached. From this point the array would sort sequentially. Once all sub arrays are sorted, the main array would be sorted due to the constant sorting of the smaller segments.

The problems I encountered with my quicksort algorithm at first was that I created new arrays for each half of the pivot. I realised that I did not have to do this and rather just pass new “start and end values”, for a segment of the array that I wanted to sort. This speed up my time about 250 times.

I tested my partition method using unit test. I then used this test multiple times by creating a new test that would sort the array. The out put array was equal to arrays.sort(). I can conclude that my code runs fine.

## Record

I created this helper class to help me record, save and write my answers to a text file. I took these results and analysed them for my discussion.

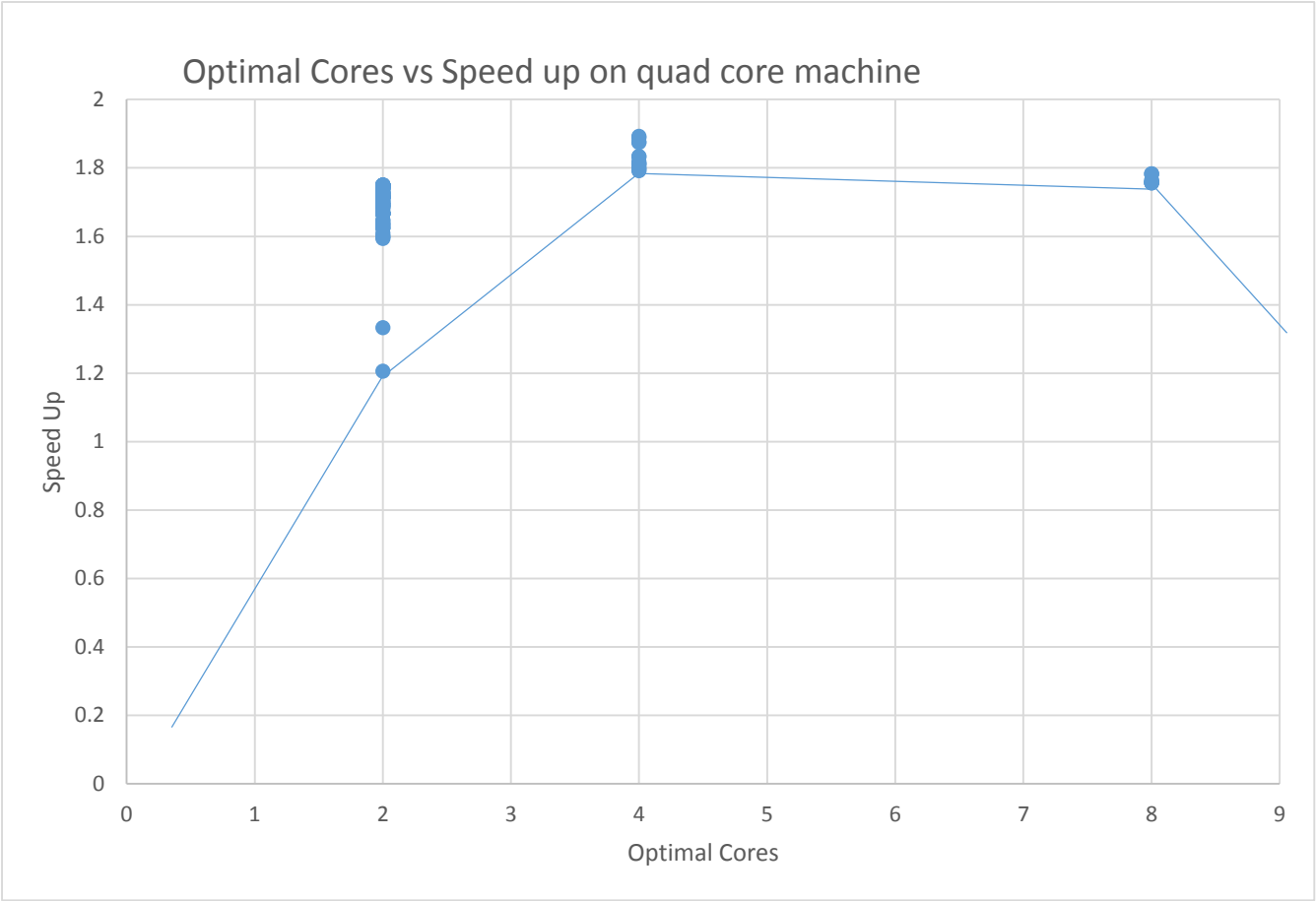
## Driver Sort

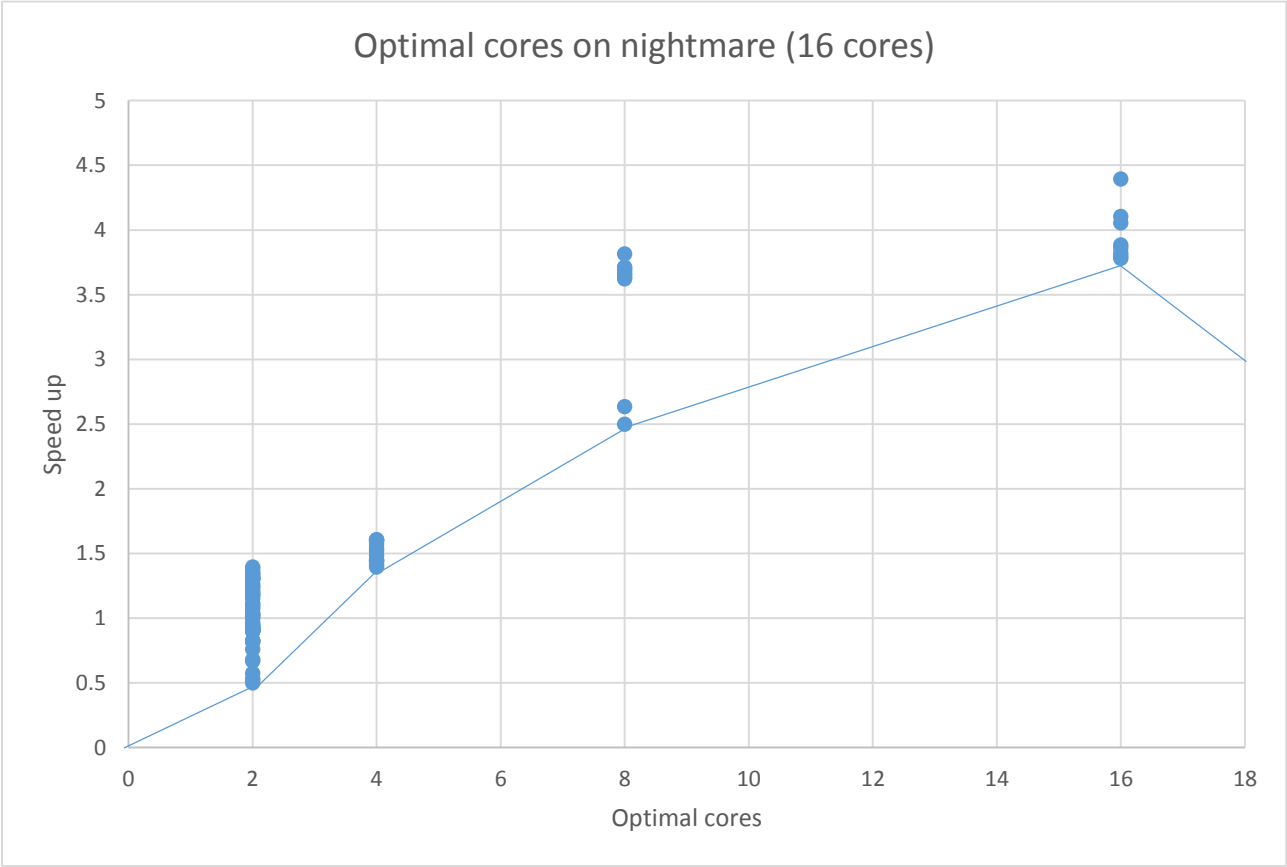
For the Driver sort I created one big algorithm in main to record all my recordings in one shot. I created a new array for every array increment. For a single array size my code ran 5 times (once for 2 threads, 4 threads, 8 threads, 16 threads and 32 threads). I ran this 5 times as well. I only had to run driver sort twice. Once for quicksort and once for merge sort.

I measured the time it took for both sequential and parallel times to run for both quick sort and merge sort. I measured my speed up by dividing my sequential times by my parallel times.

**SPEED UP = SEQUENTIAL/PARALLEL**

**Results and discussion**



## Questions

### For what range of data set sizes does each parallel sort perform well?

On a quad core machine, the parallel performed very well for array sizes 700 000 to a million. The speed up definitely increased as the array size increased. Results were not recordable for array sizes smaller than 50 000.

On nightmare, the code only started getting much faster at an array size of 900 000.
Higher cores performed poorly for low array sizes.

### What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?

My theoretical speed up on quad core machine was 2.5 using Amdahl's law. This is estimating that 65% of my codes is parallel. However I only got a max speed up of 1.89.
My theoretical speed up on nightmare was 6.6. However I got a max speed up of 4.49.
By these to results I can see that nightmare was much faster.

### How do the parallel sorting algorithms compare with each other?

Quicksort was much faster than merge sort. I was however unable to correct my quicksort. Looking at my friends results I could see that quick sort was considerably faster. This is probably due to that fact that extra arrays did not have to be created. Merge sort has a higher computation overhead due to the construction of new arrays.

### How do different architectures influence speedup?

The speed up was just about over double on nightmare even though it had 12 more cores. However, if I was expecting super linear speed up, I would expect a speed up of 7.56. The architecture with more cores yields higher speed ups. They higher speed up times in my graph on night mare were created using 16 cores where the higher speed up times on a quad core machine was created using 4 cores.

### What is an optimal number of threads on each architecture?

The optimal on night mare was 16 cores. The optimal on quad core machine was 4.

### Is it worth using parallelization (multithreading) to tackle this problem in Java?

I would say no. I found this simple problem very difficult to code and very errorful. I would say that it would be worth it if you had millions and millions of records to compute, but for small tasks it is very frustrating and not worth your time to sit, code and debug.

## Conclusions

I found parallel programming much more difficult to understand. I think that it would be costly on simple tasks as race conditions are sure to be difficult bugs to find. From my results I found that the optimal number of threads is proportional to the number of cores on your computer. I also found that parallel computing is really helpful for large array sizes. I feel you would be wasting your time if you were trying to parallel serial code to solve a very small problem.

Speed up times are proportional to the amount of data you want to compute. Small array speed ups are not noticeable. I should not parallel code if not absolutely needed.

I also found that getting my head around the assignment was much easier once I constructed a DAG. I realised where parallelism began and when it stopped. I also found that the fork/join frame work is much easier to use. I had too many problems using run() and start(). This method lead to too many race conditions.

All in all I found this assignment fairly challenging.