

Assignment 1: Parallel Programming

ANALYSIS OF MEHTOD A AND METHOD B WITH PROPSOSED
PARALLEL SOLUTIONS AND RESULTS

JAMES CASSIDY | 40267110

Table of Contents

Introduction	2
Method A	2
Method B	4
Method A Results (Sequential and OpenMP)	6
Sequential Method A Wall Time	7
OpenMP Method A Wall Time	7
Method A Parallelization	8
Method A Results (Sequential and OpenMP)	9
Sequential Method B Wall Times.....	10
OpenMP Method B Wall Processing Times.	10
Method B Parallel Efficiency	11

Introduction

Parallel solutions will be planned, tested, implemented and evaluated on sequential methods A and B using OpenMP.

Each method will be run and evaluated on my own computer before my final methods are tested on Kelvin 2. The specs include:

- Operating System: Linux Mint 21
- CPU: Intel Core i7 1065G7 4 Core, Frequency – 3.90Ghz
- Memory: 16GB DDR4 SODIMM @2666Mhz

As the assignment will be marked using an unseen data file at 1GB in size, all tests run locally and on Kelvin2 will use 'data-1GB.txt' and 'pattern1.txt'.

Method A

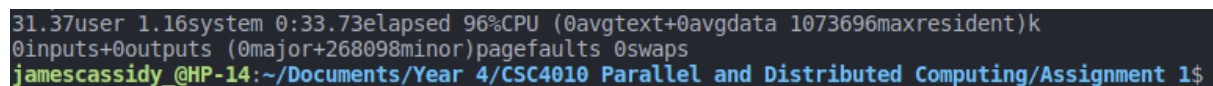
For method a, the source data has been broken down into chunks of a fixed length. This forms a data structure of `char[][]` with the first index being chunk number and the second index being the character of that chunk.

For example, if the pattern 'parent' is found in chunk 1046750 starting at index 274, the output will look like this:



```
** ,parent, 1046750, 273
```

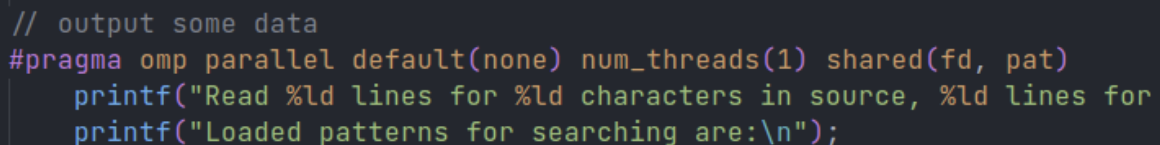
When running the sequential Method A, CPU time and wall time were 31.37 seconds and 33.73 seconds respectively.



```
31.37user 1.16system 0:33.73elapsed 96%CPU (0avgtext+0avgdata 1073696maxresident)k
0inputs+0outputs (0major+268098minor)pagefaults 0swaps
jamescassidy_@HP-14:~/Documents/Year 4/CSC4010 Parallel and Distributed Computing/Assignment 1$
```

To parallelize this program, some initial changes were made. `Num_threads()` was set to 1. Doing so sets the maximum number of threads in this region to only 1. As `printf` statements only need to run once in method, limiting them ensures system performance is utilised more for the more intensive segments of the program.

Shared uses `fd`(data file) and `pat`(pattern file). This declares the scope of these two variables as shared across all threads of Method A.



```
// output some data
#pragma omp parallel default(none) num_threads(1) shared(fd, pat)
    printf("Read %ld lines for %ld characters in source, %ld lines for
    printf("Loaded patterns for searching are:\n");
```

The following two for loops and nested for loop when combined inside this OpenMP Statement:

The variables `fd` and `pat` have been shared here for the same reason stated earlier; they are shared across all threads of Method A. Other parallel segments in my implementation of Method A will use the shared clause for the very same reason. `Num_threads` has been utilised again but has been set to 16 threads. The final Method A will be run on Kelvin2 from 1 core all the way to 16 cores. When Method A runs on 16 cores, 16 threads stated in this method will give the best result.

The first for loop in this parallel region uses `schedule(auto)` and `firstprivate(pat)`. Setting the scheduling to `auto` gives total freedom to the compiler to decide how to map iterations to the threads. Static and Dynamic scheduling were also considered. Dynamic Scheduling incurs a high overhead that immediately removed it from my testing. Static scheduling decides at the beginning what thread will use what value. With further testing taking place on multiple cores, automatic scheduling was the best option to use for both for loops.

`Firstprivate(pat)` has been used for the first for loop while `firstprivate(fd)` has been used for the second. These have been implemented in order to avoid false sharing. With the parallelisation of comes false sharing. This is a hardware limitation that arises within parallelised program as the processor tries to speed up the program. As multiple processors are updating data within the same cache line with high frequency, there can be a massive impact on performance. In order to combat this and avoid it, private data should be used whenever possible.

In the case of Method A, `pat` has been made `firstprivate` in the first for loop and `fd` has been made `firstprivate` in the second. With `firstprivate`, the variable is initialised with the value it has before the parallel region. Modifications made to any of the variables `fd` and `pat` for their respective for loops are not visible after the parallel region has run.

```

47 #pragma omp parallel default(none) shared(fd, pat) num_threads(16)
48 {
49
50     #pragma omp for schedule(auto) firstprivate(pat)
51     for (long p = 0; p < pat.lines; ++p)
52         printf("[%ld] %s\n", p, pat.data[p]);
53
54     // loop through the patterns
55     #pragma omp for schedule(auto) collapse(2) firstprivate(fd)
56     for (long p = 0; p < pat.lines; ++p)
57     {
58         // loop through the lines
59         for (long l = 0; l < fd.lines; ++l)
60         {
61             // do a search for the pattern on the line
62             long s = Search(fd.data[l], fd.linesize, pat.data[p], pat.linesize, SEARCH_MODE_FIRST);
63             // if found output the data format
64             if (s >= 0)
65                 printf("**,%s,%ld,%ld\n", pat.data[p], l, s);
66         }
67     }
68 }
69
70 return 0;

```

In the second for loop, there exists a nested for loop that searches for a pattern on the line of the data file. To parallelize this region of code, `collapse(2)` has been added to the second for loop in this parallel region. This clause will collapse both the for loop and nested loop into one loop.

With Method A parallelised, the code was able to compile and run. Results show a significant decrease in wall time to 10.64 seconds, an improvement of over 68%. CPU Usage has also increased from 96% to 430% showing that the parallel code is running and utilising all cores of the system. Patterns were also checked against the original results of the sequential method and showed no difference between them.

```

44.60user 1.22system 0:10.64elapsed 430%CPU (0avgtext+0avgdata 1074196maxresident)k
0inputs+0outputs (0major+268144minor)pagefaults 0swaps
jamescassidy_@HP-14:~/Documents/Year 4/CSC4010 Parallel and Distributed Computing/Assignment 1$

```

This new implementation of Method A will be tested on Kelvin2 for my research alongside the original sequential implementation.

Method B

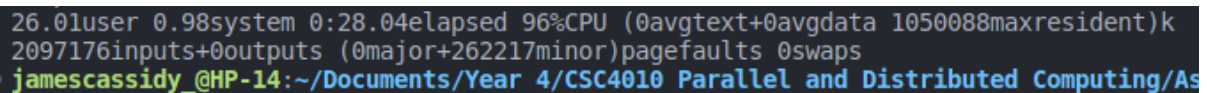
For Method B, the source data is loaded as a single chunk of full length.

Each pattern the source data is searched and the number of instances of a pattern is found is returned. 0 is returned if a pattern is not found by the method.

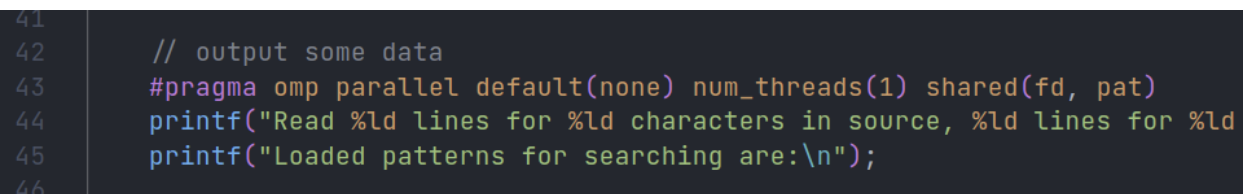
For example, if the pattern 'Market' is found 1674 times in the data file, then the output will resemble:



The first initial run of Method B showed a total CPU time of 26.01 seconds and a total wall time of 28.04 seconds.

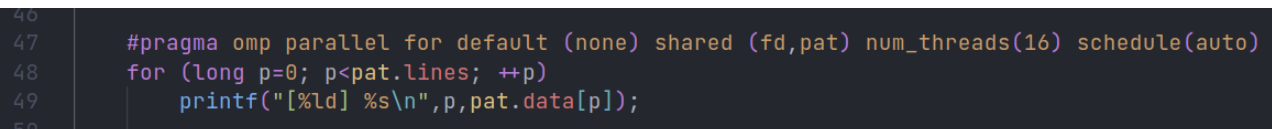


To parallelize this program, some initial changes were made. The same clauses were applied to both printf statements at the beginning of Method B that were also applied to Method A. Num_threads() was set to 1. Doing so sets the maximum number of threads in this region to only one. Printf statements in this context only need to run once in this method. Limiting them ensures system performance is utilised more for the more intensive regions of the program. The shared clause has also been implemented for variables fd and pat again in various parallel segments in Method B as they are shared across all threads.



This for loop will display the only patterns that Method B will look for. In this region of parallel code, the number of threads has been set to 16 and scheduling set to auto. The pattern contains 10 different words that Method B attempts to look for in the data file and with number of threads set to 10, this ensures there is a thread for each word. Number of threads was initially set to 10 for this segment of parallel code, however when further tests were carried out between 10 and 16, speed gains of up to half a second could be seen on my own machine. To ensure the best possible results from Kelvin, num_threads has been left at 16 to best utilise the high-performance computing available.

As seen before in Method A, Scheduling has been set for auto for all instances of this parallelised Method B. This gives the compiler total freedom to decide how to map iterations to the threads.



In the next region of parallelised code, an array patterncount has been created and initialised to 0. Scheduling has been set to auto again along with the array patterncount using the

lastprivate clause. Making this variable private helps to prevent false sharing when Method B executes, and in turn can increase performance. The lastprivate clause will keep the last value of x after the parallel region is concluded.

```
// array for the results of the pattern counting, initialise to zero
long patterncount[pat.lines];

#pragma omp parallel for default(none) shared(pat,fd) schedule(auto) lastprivate(patterncount)
for(long i=0; i<pat.lines; ++i)
{
    patterncount[i]=0;
}
```

The last parallel region will loop through the patterns and do a search for a pattern on the line using counter mode. Several clauses were used in order to parallelize this segment efficiently. Scheduling has been set to auto to give the compiler total freedom to decide how it maps iterations to threads. The num_threads clause has been used again and set to 16. As each method will be tested on a 16 threaded machine, it was best to utilise this number of threads. Other tests were carried out with lower thread counts including 10, the number of patterns available. These values however saw a slight decrease in wall time.

Inside the for loop, the following statement has been used:

- #pragma omp reduction(+:patterncount)

The use of reduction will help to absolve a race condition that might occur without parallelism. A race condition may occur when 2 or more threads have access to the same memory location and at least one of them is writing it. The variable patterncount may do this. While the loop runs, the reduction clause will tell each thread to keep track of its own patterncount[p] variable and add it up at the end when the loop has finished running. The only overhead will be at the end of the loop.

```
// loop through the patterns
#pragma omp parallel for default(none) schedule(auto) num_threads(16) shared(pat,fd,patterncount)
for(long p=0; p<pat.lines; ++p)
{
    // do a search for the pattern on the line using counter mode, incrementing the pattern counter
    // we only have one line in this mode fd.data[0]
    #pragma omp reduction(+:patterncount)
    patterncount[p] += Search(fd.data[0],fd.linesize,pat.data[p],pat.linesize,SEARCH_MODE_COUNT);
}
```

With Method B parallelised, the code was able to compile and run. Results show a significant decrease in elapsed time to 6.60 seconds, an improvement of over 76%. CPU Usage has also increased from 96% to 705% showing that the parallel code is running and utilising all cores of the system. Patterns were also checked against the original results of the sequential method and showed no difference between them.

```
46.21user 0.38system 0:06.60elapsed 705%CPU (0avgtext+0avgdata 1050740maxresident)k
0inputs+0outputs (0major+262274minor)pagefaults 0swaps
jamescassidy@HP-14:~/Documents/Year 4/CSC4010 Parallel and Distributed Computing/Assignment 1$
```

This new implementation of Method B will be tested on Kelvin2 for my research alongside the original sequential implementation.

Method A Results (Sequential and OpenMP)

In order to obtain accurate results for both the sequential and OpenMP method, a batch file was created that compiled and ran each program 100 times. As there can be a variety of results when this method runs on Kelvin2, maximizing the number of results for each core run ensures more accurate results are obtained.

Kelvin2 ran this batch file 32 different times (16 for sequential Method A and 16 for OpenMP Method A) with each batch file requesting cores from 1 to 16. Several other batch files were used to obtain results 24, 32, 64 and 128 cores. These results were not used in the graphs that follow but can be seen in the table below. Upon completion of sequential 1 core batch files, the results Kelvin2 returned are much faster than other sequential results, including 128. Because of this, I chose to exclude all 1 core results from my analysis for Method A.

Below are the mean results from all 4000 results obtained for both sequential and OpenMP Method A:

<u>Cores</u>	<u>Sequential Total Mean</u>	<u>Sequential CPU Mean</u>	<u>OpenMp Total Mean</u>	<u>OpenMp CPU Mean</u>	<u>OpenMP PS Mean</u>	<u>OpenMp PE Mean</u>
1	9.377	0.113	28.756	28.134	0.326	0.326
2	34.827	33.743	14.269	26.805	2.441	1.220
3	35.959	34.481	9.976	26.880	3.604	1.201
4	34.651	33.649	7.504	26.887	4.618	1.154
5	33.293	32.116	6.687	28.833	4.979	0.996
6	33.183	32.101	5.707	28.692	5.814	0.969
7	34.742	33.518	5.766	26.532	6.026	0.861
8	35.433	33.982	4.344	26.678	8.158	1.020
9	30.466	29.501	4.626	31.783	6.585	0.732
10	32.148	31.100	4.454	31.993	7.218	0.722
11	31.243	30.270	4.617	31.938	6.767	0.615
12	28.146	27.247	4.049	32.139	6.952	0.579
13	28.352	27.405	4.979	31.974	5.695	0.438
14	28.082	27.184	3.878	28.178	7.242	0.517
15	28.183	27.302	3.679	30.773	7.661	0.511
16	27.630	26.695	3.148	31.465	8.777	0.549
24	30.274	29.324	3.052	31.483	9.918	0.413
32	28.525	27.730	3.068	30.084	9.298	0.291
64	27.879	26.972	2.702	26.624	10.316	0.161
128	27.635	26.434	3.012	29.411	9.176	0.072

Table 1 Mean Times for Sequential and Open MP Method A

From Table 1, the most parallel efficient version of OpenMP Method A runs on 5 cores at 0.996. On 16 cores on Kelvin2, the best average result from Open MP Method A Wall Processing time was 2.84 seconds. This was taken from the best 8 results out of 100 that were collected from the batch file.

Sequential Method A Wall Time

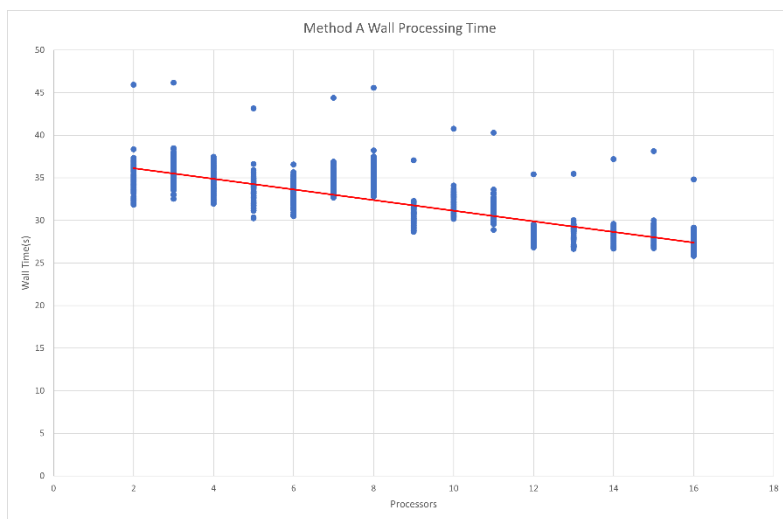


Figure 1 Method A Sequential Wall Time

From the wall time of Sequential Method A, the wall time decreases as the core count increases. This shouldn't be the case as this is a sequential program. However, as Kelvin2 is used by staff and students at QUB throughout the day, the exact same nodes and cores are not used, hence the variation.

Sequential CPU Time for Method A also tells a similar story to the Wall Processing Time as the file is not parallelised.

OpenMP Method A Wall Time

Looking at the results obtained from the OpenMP Method A, the wall processing time has drastically decreased compared to its sequential counterpart. A few outliers remain, however.

A closer look into this shows the most drastic decrease in wall time is from 2 cores to 3 cores. After this there are further decreases in the wall processing time though not as significant as before.

The parallel segments implemented in Method A have successfully improved performance. As the core count increases, the wall time decreases. From figure 3, the decrease in wall time between sequential and OpenMP Method A is stark. For the OpenMP implementation, there is a steeper decrease between cores 2 and 3. After this the decrease in wall time becomes less significant.

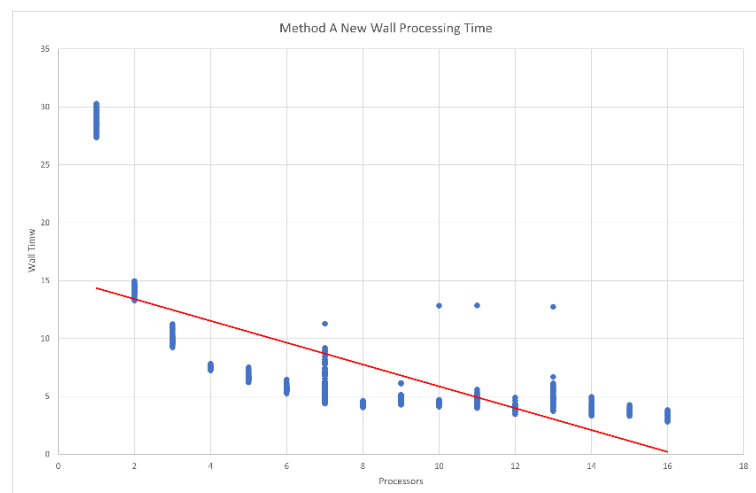


Figure 2 Method A OpenMP Wall Time

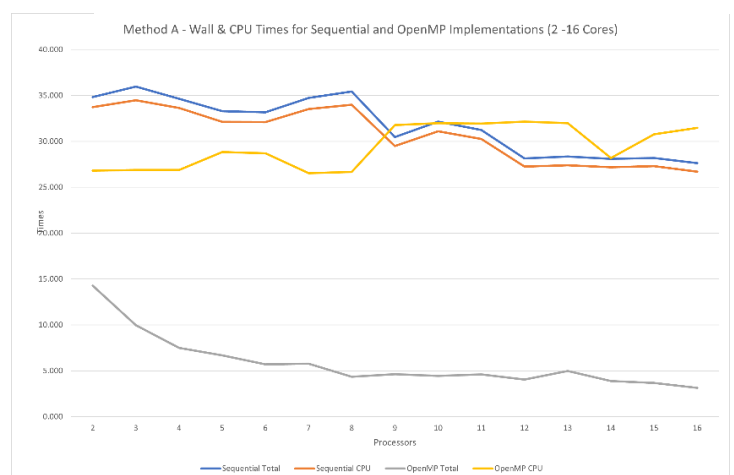


Figure 3 Method A Both Implemented Wall Times

Method A Parallelization

With the results obtained from all 1600 runs for Open MP Method A, parallel speedup unparallel efficiency can now be calculated.

Parallel Speedup is calculated by taking the time it takes the program to run on a single processor and dividing that by the time it takes to run that program on the same number of cores but in parallel. This will return the parallel speedup.

Once Parallel Speedup has been calculated, the Parallel Efficiency can be found. This takes the Parallel Speedup and divides it by the number of processors that were used in parallel for that program. A parallel efficiency of 1 means the efficiency of the parallel program is perfect.

Parallel speedup and efficiency have been calculated from the means of each core from 2 to 16. Each mean value has been calculated from 100 different results on each core. This ensures that the parallel speedup and efficiency are as accurate as possible.

From the graphs shown, Method A is most efficient between 5 cores at 0.996. This is almost perfectly efficient.

The cores before 5 and core 8 show a parallel efficiency of over 1. These results cannot be possible. The `num_threads` clause may cause this with fewer cores available to the parallel program while trying to work with 16 threads. Auto Scheduling may have played a role in parallel efficiency. Static scheduling and specific modifications made for each schedule on the specific core it runs on can be made to make the method more efficient.

While OpenMP Method A may run faster with more cores, it is not efficient in doing so. 16 cores have a parallel efficiency of 0.594 and wall time of 3.148 seconds while with 128 cores the wall time is 3.012 seconds and parallel efficiency of 0.072. Wall time has improved by 0.136 seconds on average but has become horribly inefficient.

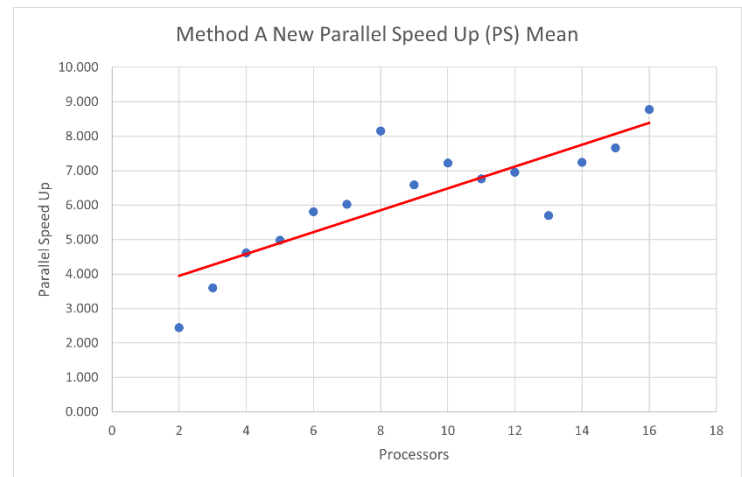


Figure 4 Method A Parallel Speedup Mean

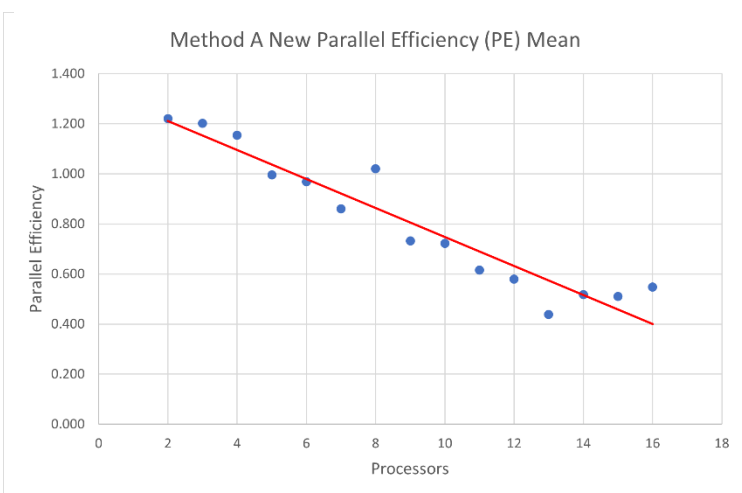


Figure 5 Method A Parallel Efficiency Mean

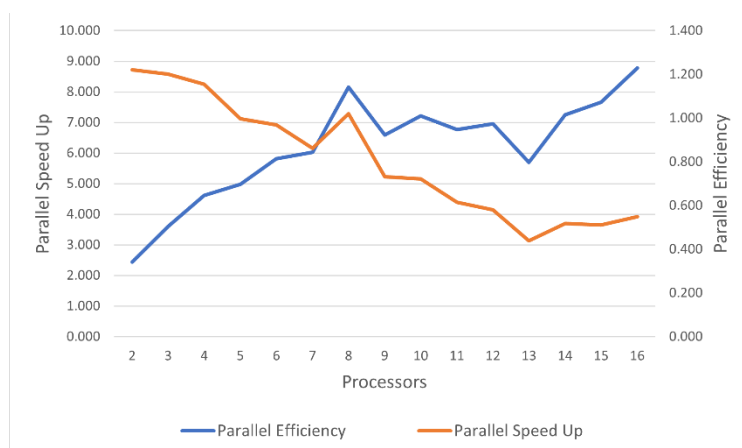


Figure 6 Method A OpenMP PE and PS Means

Method A Results (Sequential and OpenMP)

To obtain accurate results for both the Sequential and OpenMP method, a batch file was created that compiled and ran each program 100 times. As there can be a variety of results when this method runs on Kelvin2, maximizing the number of results for each core run ensures accurate results are obtained.

Kelvin2 ran this batch file 32 different times (16 for sequential Method B and 16 for OpenMP Method B) with each batch file requesting cores from 1 to 16. Several other batch files were used to obtain results 24, 32, 64 and 128 cores. These results were not used in the graphs that follow.

Below are the mean results from all 4000 values obtained for both sequential and OpenMP Method B:

Cores	Sequential Total	Sequential CPU	Open MP Total	Open MP CPU	OpenMP PS	Open MP PE
1	31.241	30.834	27.748	27.448	1.126	1.126
2	29.477	29.247	13.872	25.821	2.125	1.062
3	29.659	29.429	9.545	26.329	3.107	1.036
4	29.559	29.329	11.350	25.688	2.604	0.651
5	32.261	31.992	7.080	29.987	4.557	0.911
6	32.465	32.098	6.290	30.078	5.161	0.860
7	30.041	29.648	7.791	33.644	3.856	0.551
8	26.505	26.286	4.367	25.570	6.070	0.759
9	27.147	26.597	4.329	26.155	6.270	0.697
10	26.964	26.549	3.785	25.506	7.124	0.712
11	26.638	26.357	3.566	27.096	7.470	0.679
12	26.481	26.200	4.151	31.055	6.380	0.532
13	26.564	26.281	4.136	30.331	6.423	0.494
14	27.215	26.521	5.709	25.900	4.767	0.341
15	28.719	28.390	4.271	25.798	6.724	0.448
16	29.604	29.276	3.821	25.971	7.749	0.484
24	30.449	30.106	3.805	27.332	8.003	0.333
32	28.879	28.625	3.937	26.190	7.336	0.229
64	26.652	26.328	3.428	26.539	7.776	0.121
128	26.405	26.121	3.505	27.608	7.534	0.059

Table 2 Mean Times for Sequential and Open MP Method B

From the table above, the most parallel efficient implementation of OpenMP Method B runs at 5 cores with a mean parallel efficiency of 0.911. On 16 cores, the best average result from Open MP Method B Wall Processing time was 3.53 seconds. This was taken from the best 8 results out of 100 that were collected from the batch file.

Sequential Method B Wall Times

For the sequential implementation of Method B, the trendline for both shows there is a slight decrease in processing time as the core count increases. As batch files are ran throughout the day, it is unlikely that my results were obtained on the exact same cores and nodes.

Consideration must be taken for other members of staff and students at QUB that use Kelvin to process other data. It is likely the results were obtained whilst other, more intensive workloads were taking place. This is the why each batch file obtained 100 results for each core.

Along with this, there is a wide variation of times record for each core. In the wall processing time for Method B there are values from 32 seconds down to 27 seconds. These trends are seen throughout all cores.

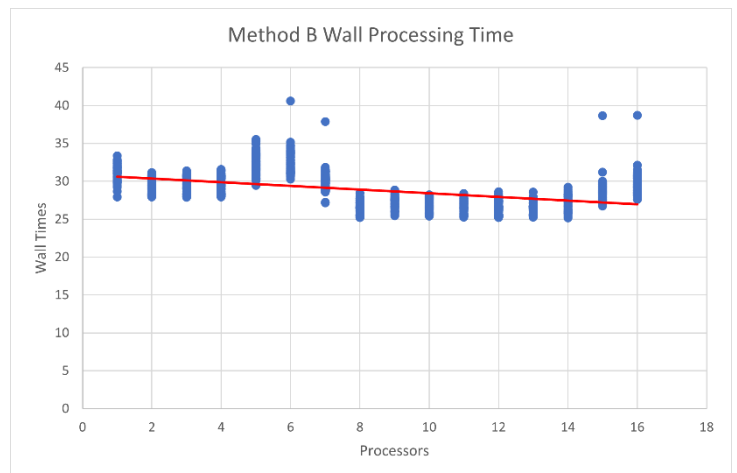


Figure 7 Sequential Method B Wall Time

OpenMP Method B Wall Processing Times.

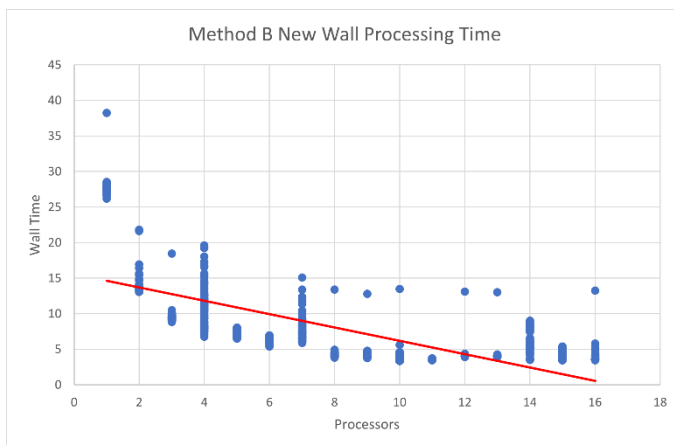


Figure 8 Method B OpenMP Wall Time

In figure 9, the averages for both sequential and Open MP CPU and wall times have been taken and plotted. It can be seen immediately as a second core is implemented for OpenMP Method B; the wall time decreases sharply. After this the decrease is less significant. When compared to the sequential Method B, wall time of Open MP is a fraction of the sequential implementation.

Implementation of parallelisation for Method B shows a massive decrease in wall time from 1 core to 7 cores.

From 8 cores onward, there is little variation in the wall processing time. The wide range of wall times in each core can again be attributed to Kelvin2.

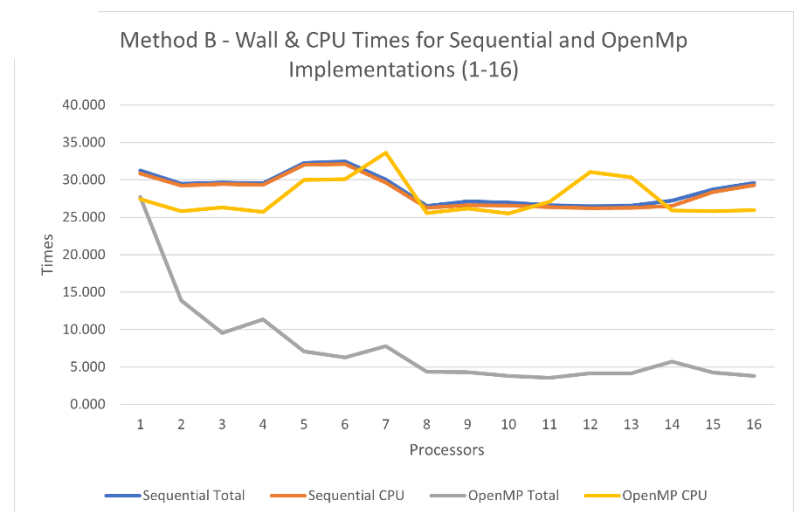


Figure 9 Method B Wall and CPU Time Both Implementations

Method B Parallel Efficiency

Having calculated wall processing times for OpenMP Method B, I can now see how efficiently the method is performing by calculating the parallel speedup and parallel efficiency.

Parallel speedup is the ratio of time that is required to complete a task with a single processor over the time it takes using multiple processors. The higher the number, the better the speedup.

From figure 10, the trendline points upwards to show parallel speedup increases as the number of processors used increases.

With parallel speedup calculated, the parallel efficiency for each variation of core count can now be calculated. This is a measure of how effectively the method uses multiple processors.

From Figure 11, there are several issues that arise. Cores 1 – 3 all show a parallel efficiency of over 1. This is impossible. Reasoning behind this could perhaps be false sharing still existing in some aspect in the OpenMP implementation of Method B or a race condition not properly handled. A `num_threads` clause may also play a part in this. With a parallel segment of the program set to have 16 threads, the lower core count of Kelvin2 may not have processed this and correctly, causing multiple parallel efficiency values being greater than 1.

The most parallel efficient implementation of Open MP Method B ran on 5 cores with a parallel efficiency of 0.911.

While OpenMP Method B may run faster with more cores, it is not efficient in doing so. 16 cores have a parallel efficiency of 0.484 and wall time of 3.821 seconds while with 128 cores the wall time is 3.505 seconds and parallel efficiency of 0.059. Wall time has improved by 0.316 seconds on average but has become horribly inefficient.

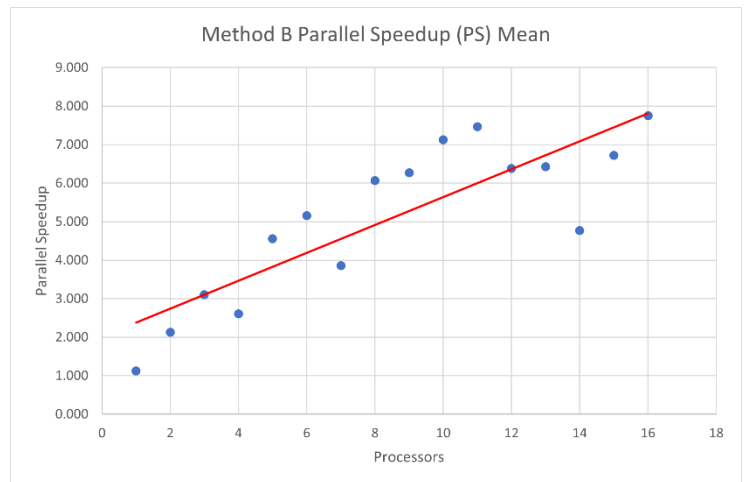


Figure 10 Method B Parallel Speedup Mean

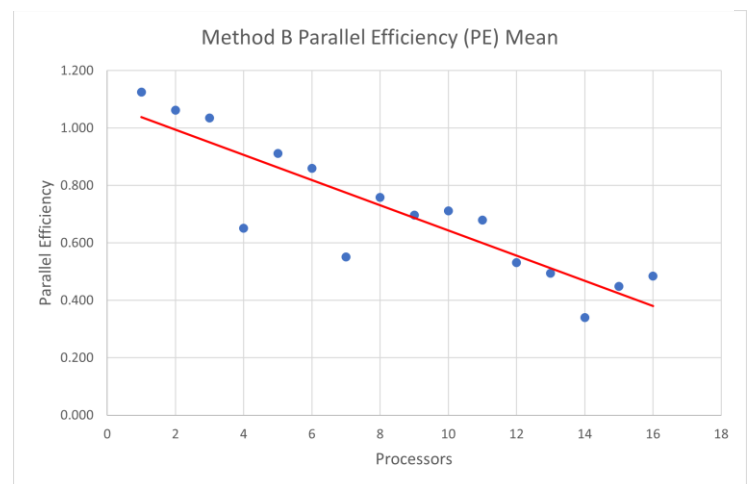


Figure 11 Method B Parallel Efficiency Mean

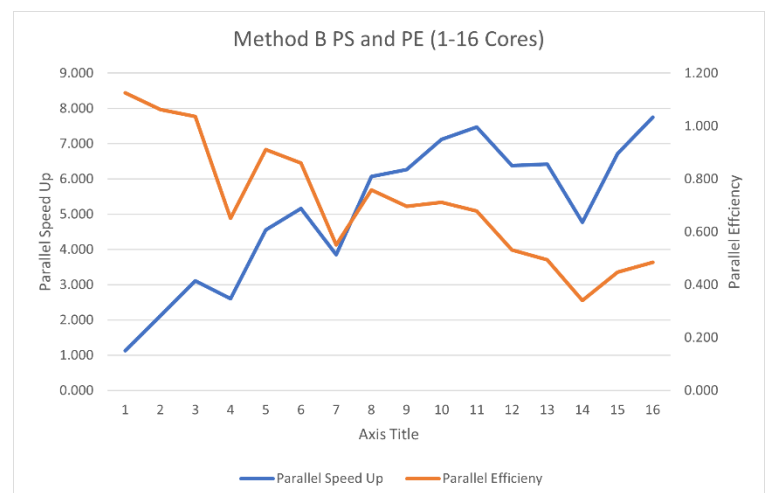


Figure 12 Method B PS and PE Means