Pattern Design and Parallelism

HPC–ASSIGNMENT1

Student Name: Adam Alcorn

Student ID: 40211628

Abstract

Background

Part A of this report looks at pattern matching in C. It uses a simple algorithm to retrieve patterns from various sizes of texts. The methods used to derive the patterns from the text are simple, however, they demonstrate the complexity of programs and how menial tasks can cause large overheads.
Part B examines parallelism through the OpenMP library. OpenMP is a programming interface that provides a simple form of parallelism for C and C++ programs.

Methods

The sequential searching algorithm used two different files. The pattern file stored the pattern and the text file which held all text portions, including an occurrence of the pattern. A C program has been written to generate these files, followed by a concatenation via a shell script. The searching part of the process then identifies patterns throughout the text files. Once completed the times are recorded and plotted in the graphs as shown in Section A.

Section B uses the OpenMP library. This library provides different parallel constructs for C language programs. Examples of these include parallel for loops, used to create concurrency between threads whilst iterating over the contents of the loop; parallel regions and sections, used to separately identify areas of parallel interest; and lastly, critical sections.

Results

Based on the data extracted in part A and from running the searching sequential program, the derived times were varied and felt random at first. However, after visualising the data through graphs and charts it was clear there were different trends relating to the size and amount of data in the patterns and texts.
Part B shows how concurrency can be brought to a simple program such as pattern retrieval. Many of the solutions highlight key concepts used in OpenMP and utilise threads to quicken programs.

Conclusions

One conclusion that was realised from the data visualised in part A was that the greater the amount of text versus the size of the pattern impacted the time greatly. One scenario proposed that the text be 100,000 times larger than the pattern. This resulted in a lengthy time of 10 seconds as was expected, as more comparisons were carried out, therefore, prolonging computation time. One result which proved surprising was the compiler-based computation times. Whilst these were mostly predictable for O0 and O2, O3 proved that there is very little speedup to be gained from running the program using this and, in some cases, 02 may prove to be faster.
Part B gave an insight into how using OpenMP can be used to transform a sequential searching pattern into a parallel program. An activity such as adding parallel functionality can lead to many advantages and mainly improve the overall execution time. Whilst race conditions and infinitive loops can be easily created through using these, when implemented correctly, the outcome vastly outweighs the problem statements.

Part A – Pattern Design for a Worst-Case Scenario

Introduction

Pattern matching in large text files has always been an issue of speed. Many different algorithms have been developed to quickly find the correct outcome. The one used in this part is a simple searching algorithm and uses two differing text files. The text file is searched for the pattern and the position of the pattern, number of comparisons, and time, are all recorded. We can use this numerical data to plot charts and derive certain conclusions. Experimentation played a large role in this project as the time taken for the overall searching relied on the generation of a text file containing the worst-case scenario.

The Worst-Case Scenario

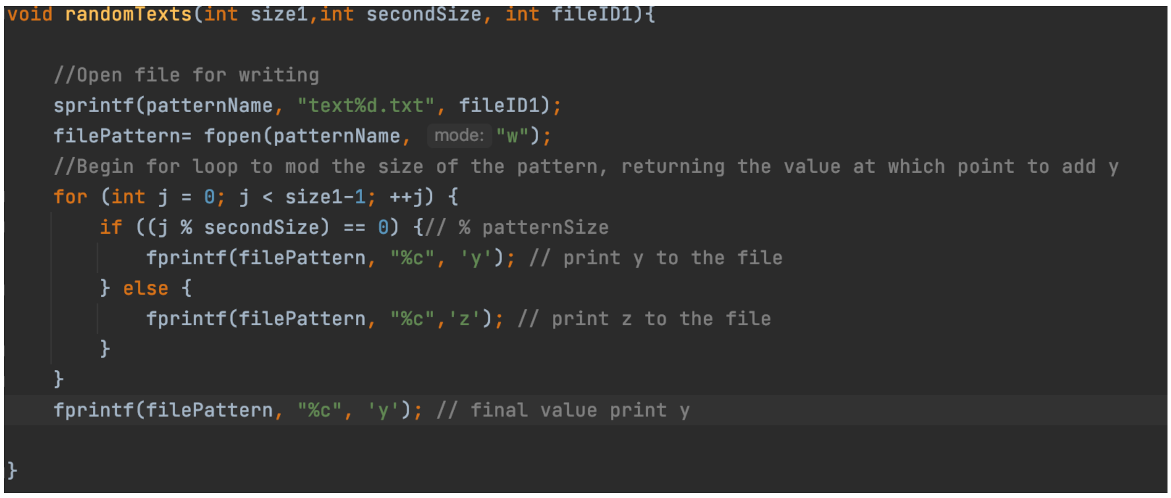
The worst-case scenario used is one that consists of the pattern being at the end of the text. The text also needs to contain a repetitive sequence where the first N-1 characters of the pattern are repeated, until the last sequence where the full pattern is added (example below).

```
SET textCounter to zero
FOR each of the elements in size of text
  WHILE textCounter is less than sizeOfText DIVIDED by patternSize
  INCREMENT sizeOfText
        DECLARE patternCounter SET to 0
        WHILE patternCounter is less than patternSize
                WRITE to file value in pattern at index patternCounter
        END WHILE
END FOR
```

The practical implementation of this was much simpler and involved creating a pattern of z's and y's separated to reflect the above pseudocode. For example, if my pattern file were to include three z's the corresponding text file of size nine would look like, zzyzzyzzz.

The code that performs this can be seen in figure 1.0. One factor unaccounted for was the final addition of the separation letter before the pattern. This is due to the referencing used where our pattern value mods (%) the pattern size. To achieve a worst-case scenario, it is required to add an extra character at the end, as this results in the correct number of comparisons. The outcome of this is one full extra check of the pattern resulting in a slight increase in time.

A piece of functionality not shown in figure 1.0 is the concatenation. This has been excluded as it was possible to join the two files using a shell script. This form of concatenation creates a new file and places the pattern at the end of the text. Objectively it resulted in an easier process as many of the files were being moved and ran this way.



```c
void randomTexts(int size1,int secondSize, int fileID1){

    //Open file for writing
    sprintf(patternName, "text%d.txt", fileID1);
    filePattern= fopen(patternName,  mode: "w");
    //Begin for loop to mod the size of the pattern, returning the value at which point to add y
    for (int j = 0; j < size1-1; ++j) {
        if ((j % secondSize) == 0) {// % patternSize
            fprintf(filePattern, "%c", 'y'); // print y to the file
        } else {
            fprintf(filePattern, "%c",'z'); // print z to the file
        }
    }
    fprintf(filePattern, "%c", 'y'); // final value print y

}
```

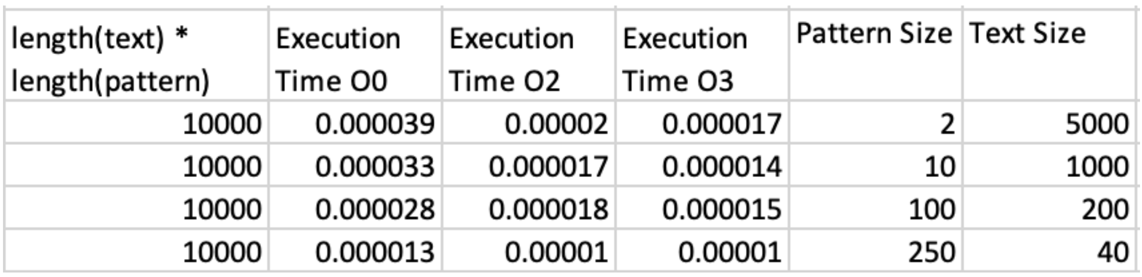
Figure 1.0 – Code to generate random text files based on pattern sizes.

Data justification and Graphical Modelling

Graph 1 and Supporting Data

The graph and supporting information shown in figures 2.0 and 2.1 have been derived from running the worst-case scenario for a product size of 10000 (10^4). The code generated was timed using the following command $time searching_sequential.

The first observation we can see is that the larger the pattern size the quicker the program runs. This is expected as once the program finds the beginning of the last pattern it will no longer have any more incorrect matches. This results in a faster time. Another factor that plays an important part is the size of the file. When a larger pattern is specified, the product of pattern and pattern size is taken to formulate a text file that matches. The outcome of this is a sizable difference between a program with a pattern size of 2 versus 250. Figure 2 clearly shows this in the pattern and text size fields.



length(text) * length(pattern)	Execution Time O0	Execution Time O2	Execution Time O3	Pattern Size	Text Size
10000	0.000039	0.00002	0.000017	2	5000
10000	0.000033	0.000017	0.000014	10	1000
10000	0.000028	0.000018	0.000015	100	200
10000	0.000013	0.00001	0.00001	250	40

Figure 2.0 – Code to generate random text files based on pattern sizes.

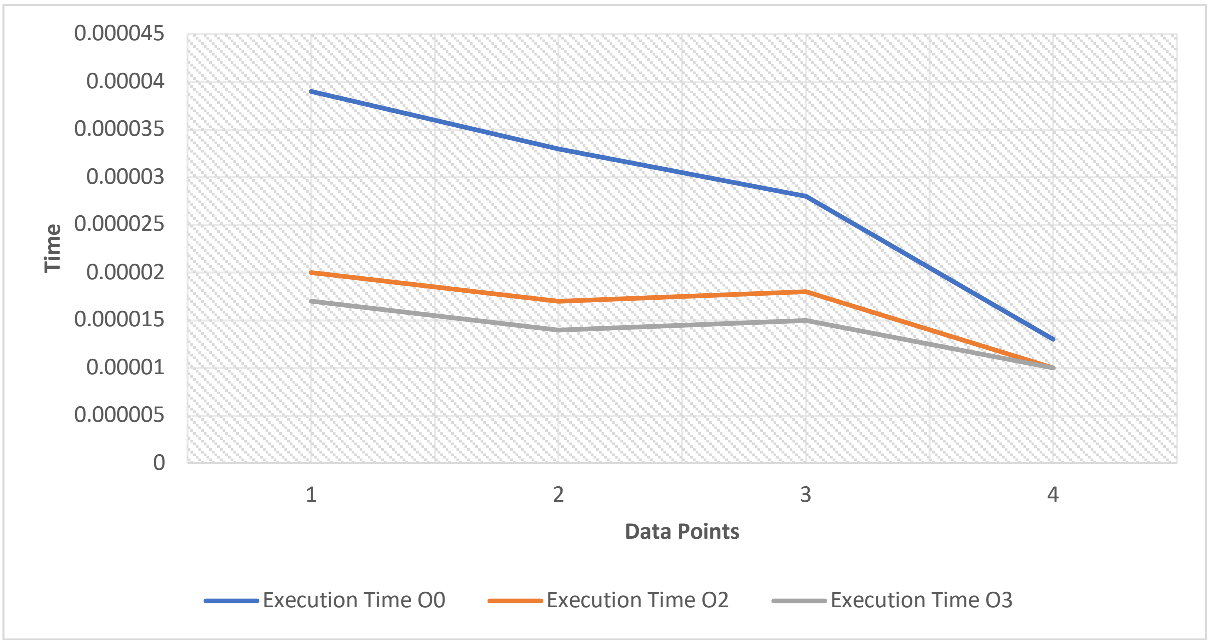


Figure 2.1 – Code to generate random text files based on pattern sizes.

Graph 2 and Supporting Data

The graph and supporting information shown in figures 3.0 and 3.1 have been derived from running the worst-case scenario for a product size of 10000 (10^6). The code generated was timed using the following command $time searching_sequential.

The first test looked at differentiating large text sizes. Graph two looks at how large pattern sizes can impact a program. From point two onwards we can see the expected scalar, however, there is one anomaly at data point one. There is very little reasoning as to why this value should be taking an unexplainable amount of time for an equal pattern and size length. However, from point two onward it can be verified that these results are correct.

From the experiments and analysis carried out, it is known that small text size and large pattern size (and vice versa) will take the longest. Continual positive growth of one data set and negative growth of the other will result in latencies in timing and increased computation, especially when implementing a worst-case scenario, such as the one identified above.

length(text) * length(pattern)	Execution Time O0	Execution Time O2	Execution Time O3	Pattern Size	Text Size
1000000	0.001838	0.000823	0.000693	1000	1000
1000000	0.000053	0.000034	0.000028	10000	100
1000000	0.000297	0.000139	0.000136	100000	10
1000000	0.002075	0.000961	0.001063	1000000	2

Figure 3.0 – Code to generate random text files based on pattern sizes.

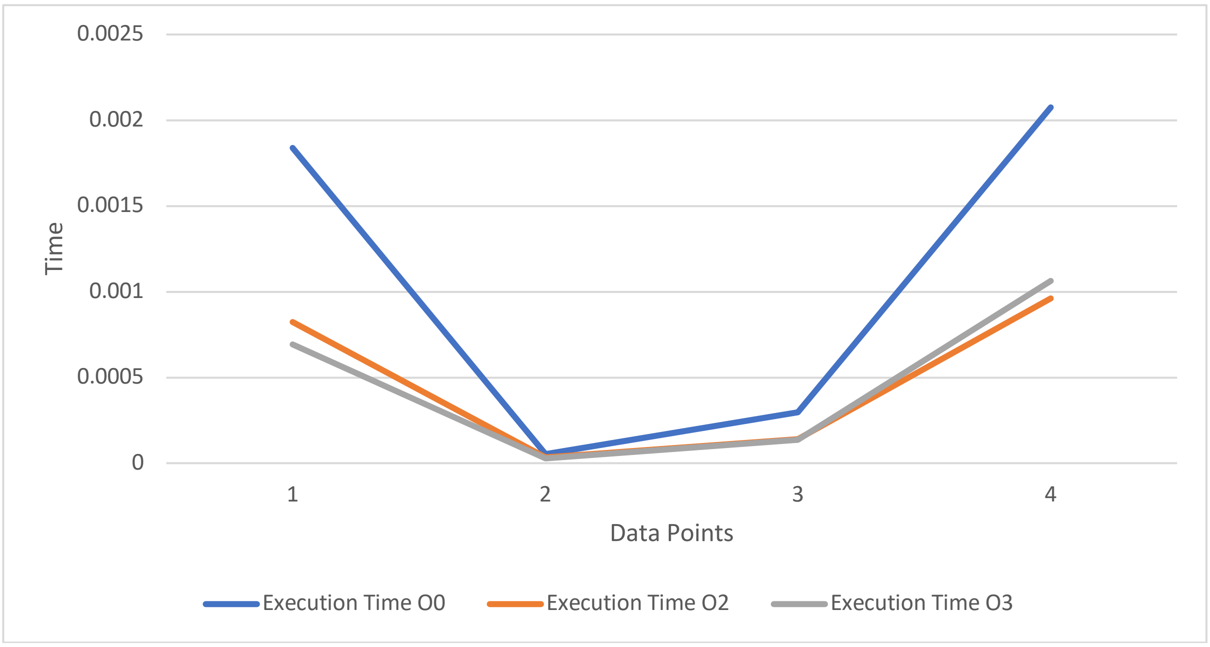


Figure 3.1 – Code to generate random text files based on pattern sizes.

Graph 3 and 4 with Supporting Data

The graph and supporting information shown in figures 4.0,4.1,5.0,5.1 have been derived from running our worst-case scenario for a product size of 10000 (10^8) and 10000 (10^10). The code generated was timed using the following command $time searching_sequential.

With the knowledge gained from the first two graphs, 10^8 and 10^10 could now be plotted to produce worst-case scenarios with significant data values. These values, for simplicity, have been constants which we are familiar with i.e. values to the power of 10. When running the program with these larger data sets, it became clear that better trends were emerging when looking at similar pattern sizes. Before we had encountered an anomaly in the same pattern sizes, yet now the data showed the expected outcome, a value that followed the trend lines shown in time and complexity.

One piece of notable information gained from looking directly at the graphs is the large drop-in time on the final data point (4). This is in both iterations i.e. 10^8 and 10^10. Once again this is due to what had been identified earlier where if the pattern size was much larger than the text size then the comparisons are subsequently lower, resulting in less time spent searching for the wrong pattern as there are fewer data locations to iterate through.

length(text) * length(pattern)	Execution Time O0	Execution Time O2	Execution Time O3	Pattern Size	Text Size
100000000	0.114528	0.054929	0.059037	100	1000000
100000000	0.105318	0.051081	0.055012	1000	100000
100000000	0.108597	0.054011	0.0521	10000	10000
100000000	0.001292	0.000634	0.000777	100000	1000

Figure 4.0 – Code to generate random text files based on pattern sizes.

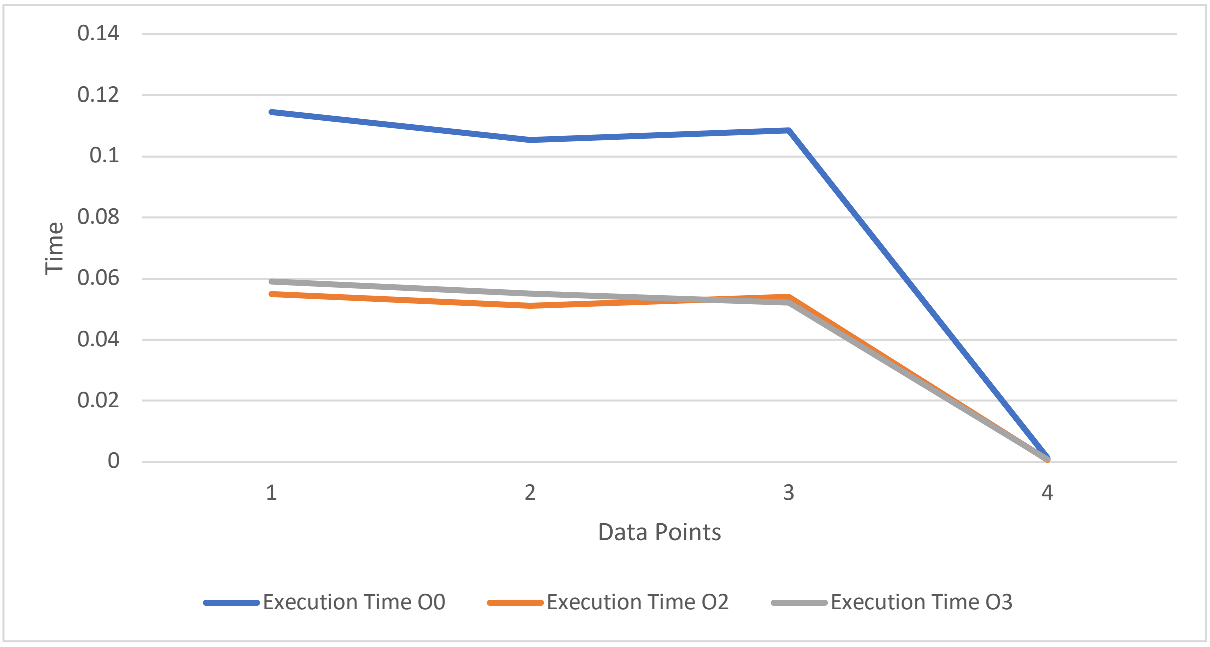


Figure 4.1 – Code to generate random text files based on pattern sizes.

length(text) * length(pattern)	Execution Time O0	Execution Time O2	Execution Time O3	Pattern Size	Text Size
10000000000	10.56074	4.760431	4.922138	10000	1000000
10000000000	10.508987	5.136894	4.876634	100000	100000
10000000000	10.46952	5.214827	4.814659	1000000	10000
10000000000	0.105694	0.053513	0.050387	10000000	1000

Figure 5.0 – Code to generate random text files based on pattern sizes.

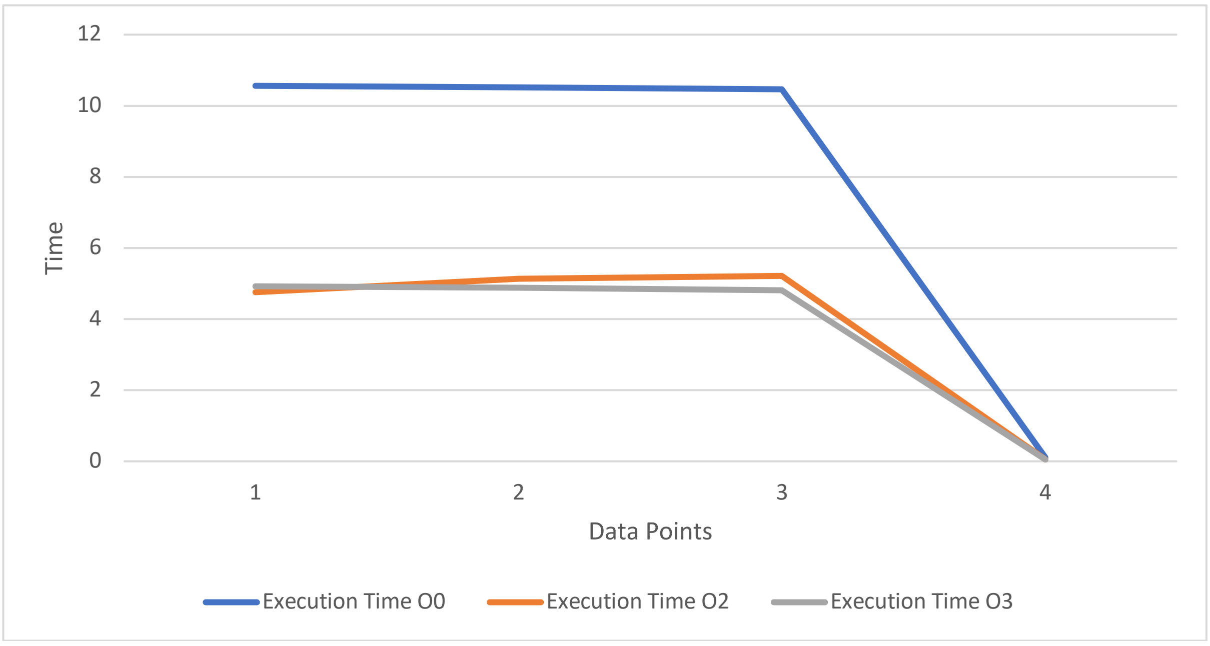


Figure 5.1 – Code to generate random text files based on pattern sizes.

As a side note, when looking at complexity, I took it upon myself to also run the differentiating compiler options. Running O0, O2, and O3 resulted in our three different data lines on the graph. O2 and O3 were very similar, which after receiving a decrease in time by nearly 50% was very surprising. Upon further research, it was discovered that using 03 can “increase the code size by performing excessive inlining, leading to increased instruction cache pressure from the increased code size and thus minor performance improvements” [1]. This may be the reason for the slight increase at data point one, Execution time 03, in graph 5.1.

Conclusion

The above graphs and data represent the findings of a worst-case scenario. The process taken to retrieve these results was analytical and experimental, however, the outcome achieved was appropriate and provides an insight into how pattern matching is fundamentally complex. Much of the code written and data visualised demonstrates how the smallest changes can return the largest of outcomes and create prolonged times due to similar patterns in text.

Part B – Parallelised Solutions for Tests 0-2

Introduction

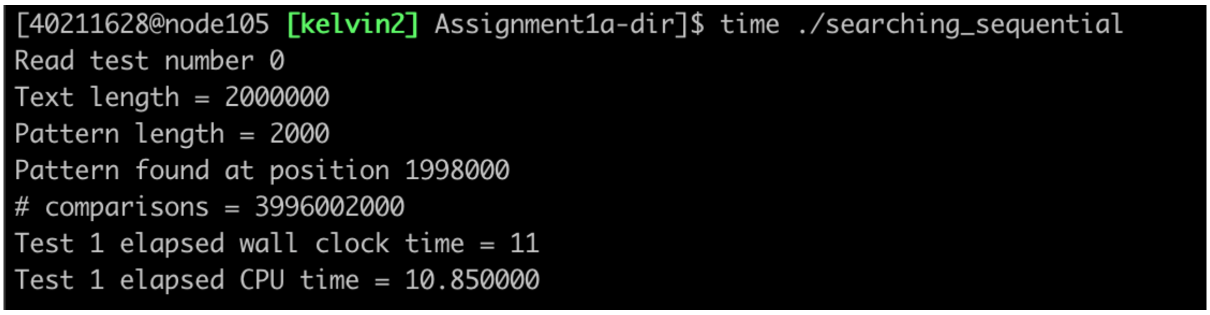
Parallelism is a topic that is at the forefront of technology. It uses high-level concepts to speed up programs and create algorithms that are optimised. OpenMP is a library that allows for simple concurrency throughout programs. The library is straightforward to implement, and whilst many of the concepts can be confusing, OpenMP has provided detailed software, showing practical examples and detailed explanations.

The following assignment looks at the execution of a parallel searching algorithm. The program used in part A has been adapted to run with many different threads. The parallelism used was taken from examples online and my own knowledge of the issue.

Test0

Part 1: Running the Sequential Program

The first step in this process was to run the sequential searching algorithm, as defined in part A, and note the elapsed (i.e. wall-clock) time taken and the number of comparisons reported. An example screenshot of this can be found below.
As shown below the pattern was found at position 199800 and the comparisons equalled 3996002000. One important factor here is the CPU time. This process only took 10.85 seconds to the find the pattern.



```
[40211628@node105 [kelvin2] Assignment1a-dir]$ time ./searching_sequential
Read test number 0
Text length = 2000000
Pattern length = 2000
Pattern found at position 1998000
# comparisons = 3996002000
Test 1 elapsed wall clock time = 11
Test 1 elapsed CPU time = 10.850000
```

Figure 1.0 – Sequential Program on Test0

Part 2: Creating a parallel program and experimenting with different threads and scheduling strategies

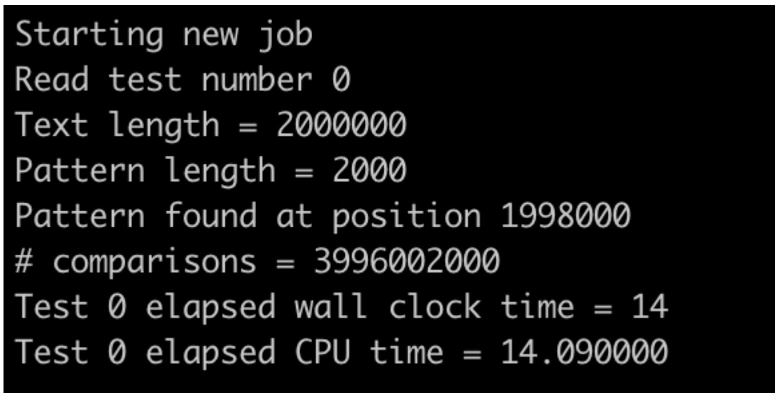
When beginning to design the parallel program the approach taken involved building the solution to work for one thread. Once a solution was devised for one thread it could then be manipulated for the use of two threads. To begin this process, all variables and their scope were considered. Variables that were set based on other parameters (such as using the size of a field), were made "first private", whilst values that were set from within the loop were initialised to zero and set to "private". Shared variables included a counter for tracking the comparisons and an index for the point at which the pattern was found.

Shown below is the pseudocode for the problem. This code outlines how the parallel loop works and how keeping track of the variables can be done. The "for loop" begins and continues running until the check within our “while loop” is achieved. This check is to test whether the element that the program is looking at in the text file is equal to the element in the pattern. From within this loop, an increment is performed on the pattern element and data element to check if the pattern is found. Once we know that this has finished, the loop is broken. If the pattern was found and the length matches, then the program will invoke a critical section and note the index at which the code was found. If the pattern was not found, then the loop keeps rolling until the end.

Pseudocode for Concurrent Algorithm:

```
SET i to zero
FOR each of the elements in lasti
  WHILE i is less than or equal to the lasti
  INCREMENT i
      DECLARE pattern element  SET to 0 and DECLARE data element  SET to i
      WHILE element within text data is equal to element within pattern data
            INCREASE the pattern element
            INCREASE the amount of comparisons
            INCREASE the data element
      END WHILE
      IF pattern element is equal to the pattern length
            CRITICAL SECTION
            SET pattern index to i
            END critical section
      ELSE element is NOT equal to the pattern length
            INCREASE the amount of comparisons
END FOR
```

Figure 2.0 shows the parallel program running on 1 thread. This test was conducted to prove that the data was retained whilst running for one thread. As shown in figure 3.0 the data gathered was also correct in generating the required pattern index and number of comparisons. This is outlined in the field with the tick. The data itself is very arbitrary and requires some study to work out the trends. However, the graphs provide a graphical representation



```
Starting new job
Read test number 0
Text length = 2000000
Pattern length = 2000
Pattern found at position 1998000
# comparisons = 3996002000
Test 0 elapsed wall clock time = 14
Test 0 elapsed CPU time = 14.090000
```

Figure 2.0 – Sequential Program Running Test 0 on 1 Thread

	PATTERN FOUND AND NUMER OF COMPARISONS CORRECT	WALL CLOCK TIME	PS (PARALLEL SPEEDUP)	PE (PARALLEL EFFICIENCY)
STATIC 1T	✓	12	1	1
STATIC 2T	✓	5	2.4	1.2
STATIC 4T	✓	2	6	1.5
STATIC 8T	✓	2	6	0.75
STATIC 16T	✓	1	12	0.75
STATIC 32T	✓	0	12	0.375
STATIC 64T	✓	0	12	0.1875
DYNAMIC 1T	✓	12	1	1
DYNAMIC 2T	✓	6	2	1
DYNAMIC 4T	✓	3	4	1
DYNAMIC 8T	✓	3	4	0.5
DYNAMIC 16T	✓	1	12	0.75
DYNAMIC 32T	✓	1	12	0.375
DYNAMIC 64T	✓	1	12	0.1875
GUIDED 1T	✓	11	1	1
GUIDED 2T	✓	6	1.83333333	0.91666667
GUIDED 4T	✓	3	3.66666667	0.91666667
GUIDED 8T	✓	2	5.5	0.6875
GUIDED 16T	✓	1	11	0.6875
GUIDED 32T	✓	1	11	0.34375
GUIDED 64T	✓	1	11	0.171875
AUTO 1T	✓	11	1	1
AUTO 2T	✓	6	1.83333333	0.91666667
AUTO 4T	✓	3	3.66666667	0.91666667
AUTO 8T	✓	3	3.66666667	0.45833333
AUTO 16T	✓	2	5.5	0.34375
AUTO 32T	✓	1	11	0.34375
AUTO 64T	✓	1	11	0.171875

Figure 3.0 – Data gathered from running different threads counts and different schedules

Key points extracted from visually analysing Figure 4.0:

- The Static schedule achieves a greater optimisation quicker. As shown in the diagram the speedup of 12 is achieved at a greater rate by using a static scheduler. Whilst the dynamic schedule still reaches this point, the path which it takes and speedups achieved for each individual thread, are less than the static schedule.
- Another point that should be noted is that the maximum speedup which the guided and auto schedule could achieve was 11.
- From looking at the data we can see the auto schedule had the worst performance overall. The results show that it took an additional 16 threads for the auto schedule to achieve the maximum speedup whilst the others only required the original 16.

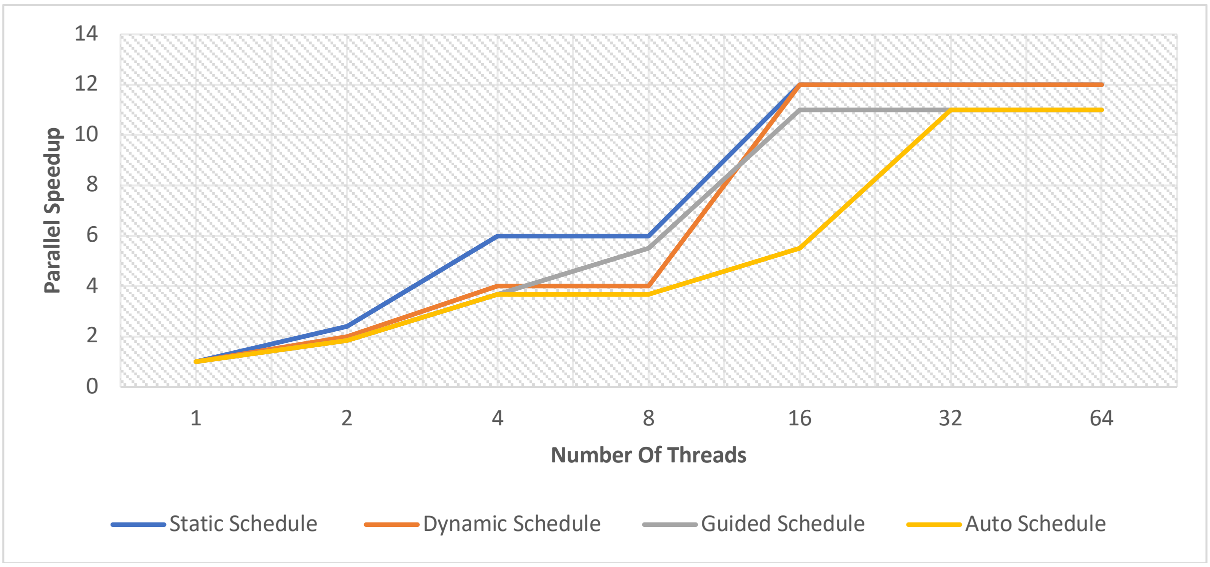


Figure 4.0 – Graph Depicting the Parallel Speedup

Key points extracted from visually analysing Figure 5.0:

- When referring to a speedup of parallel efficiency we look at how efficient our parallel speedup was compared to the number of threads. For example, when referring to 1 this means there was a speed up of half in correspondence to the number of threads used.
- The static schedule achieved a speedup of 1.2 which then increased to 1.5. This was the greatest speedup shown and is also reflected in figure 4.0. This eventually levelled out; however, it shows that by increasing the number threads it will not exponentially increase the efficiency.
- The other 3 schedules ran at a value of 1 or less than. Whilst compared to the static schedule, this may seem worse, it remains that the wall clock time was still being reduced at a rate of half or below, eventually becoming aligned with the static schedule.

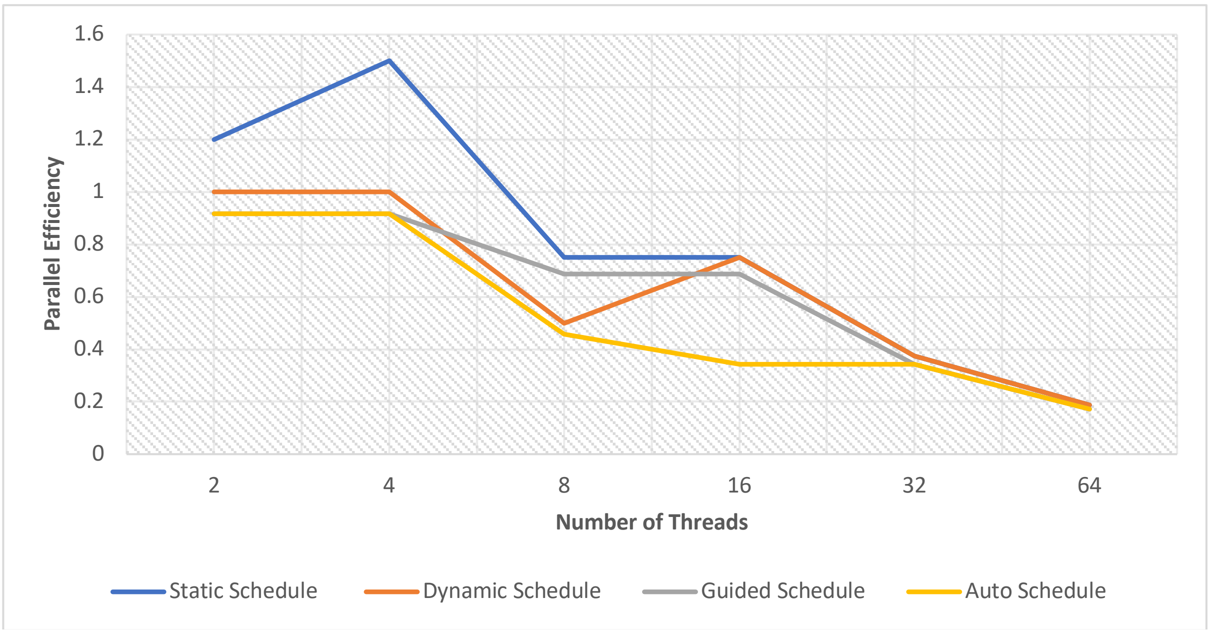
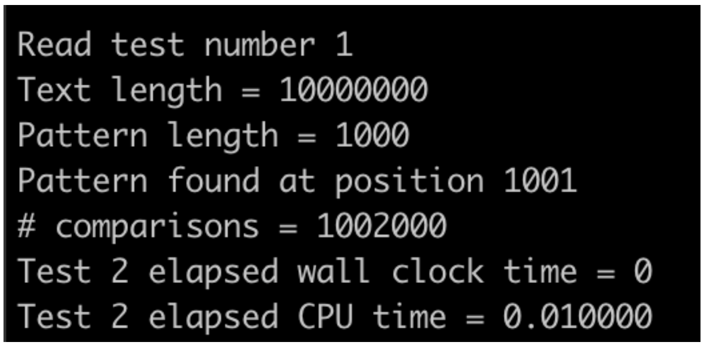


Figure 5.0 – Graph Depicting the Parallel Efficiency

Test1

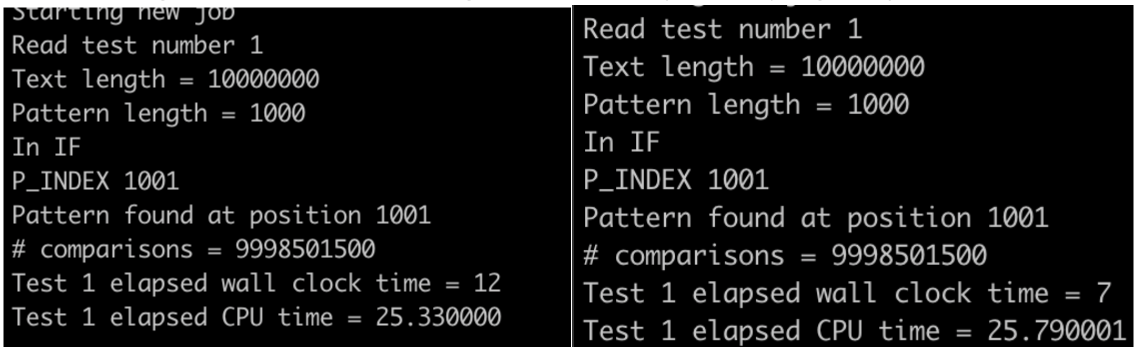
As shown below the sequential searching algorithm finds the pattern at position 1001. This is correct and I have confirmed this within the file. Figures 1.1 and 1.2. show the printed out the values of “i” and “IN IF”. This helps when tracking the data flow through the program. The statement “IN IF” provided a flag to correctly check if the loop was entering this statement.

When running the concurrent programs for the first time it can be noted the number of comparisons is much higher, resulting in longer running times. Whilst the program is still scaling based on the number of threads, the number of comparisons must compliment those shown in figure 1.0. To achieve this, a boolean value was implemented which is set when the pattern is found. The result after implementation is visible in figure 1.3. The code for this is also available in appendix A.



```
Read test number 1
Text length = 10000000
Pattern length = 1000
Pattern found at position 1001
# comparisons = 1002000
Test 2 elapsed wall clock time = 0
Test 2 elapsed CPU time = 0.010000
```

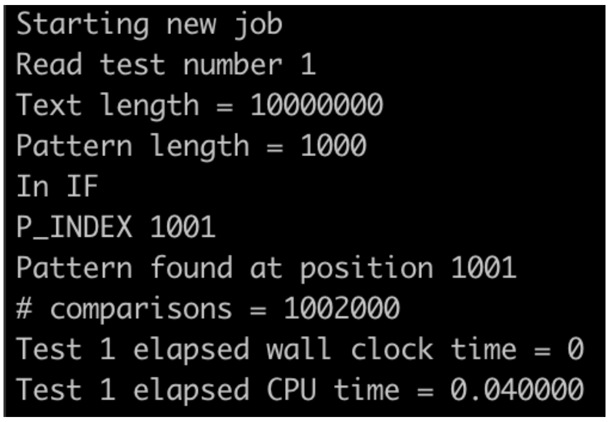
Figure 1.0 – Kelvin running the search sequential program for test1



```
Starting new job
Read test number 1
Text length = 10000000
Pattern length = 1000
In IF
P_INDEX 1001
Pattern found at position 1001
# comparisons = 9998501500
Test 1 elapsed wall clock time = 12
Test 1 elapsed CPU time = 25.330000
```

```
Read test number 1
Text length = 10000000
Pattern length = 1000
In IF
P_INDEX 1001
Pattern found at position 1001
# comparisons = 9998501500
Test 1 elapsed wall clock time = 7
Test 1 elapsed CPU time = 25.790001
```

Figure 1.1 – Kelvin running the Searching OMP program for test1 on 2 threads (left) and test1 on 4 threads (right)



```
Starting new job
Read test number 1
Text length = 10000000
Pattern length = 1000
In IF
P_INDEX 1001
Pattern found at position 1001
# comparisons = 1002000
Test 1 elapsed wall clock time = 0
Test 1 elapsed CPU time = 0.040000
```

Figure 1.2 – Addition of the Boolean Flag and New Values

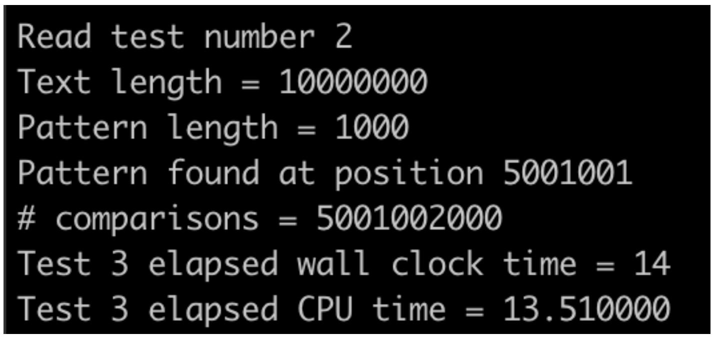
Test2

Test two data was loaded into the sequential program and ran to completion. The results of this test can be seen in figure 1.0. One assumption which was made (test case 2) was that you may not assume that there is only one occurrence of the pattern in the text. This resulted in multiple patterns spread throughout the text patterns. As shown in figure 1.0 we only return one pattern. I believe this to be the first location at which the pattern is found. Figure 1.1 demonstrates the data being ran against 2 and four threads. Note that the pattern found is the same in each. This confirms the first location of pattern, however, fails to identify any of the remaining locations.

Figure 1.2,1.3 show the creation of an array. The array will hold the locations of the different patterns spread throughout the text. As seen below the pattern found index is also incremented and the value set to the corresponding location of within the text. When setting up the array I attempted to make it dynamic using malloc statements and the allocation/freeing of memory, however, I found that it was easier to use the "if statement" shown in figure 1.4. An example of the code generated using two threads can be found in appendix B.

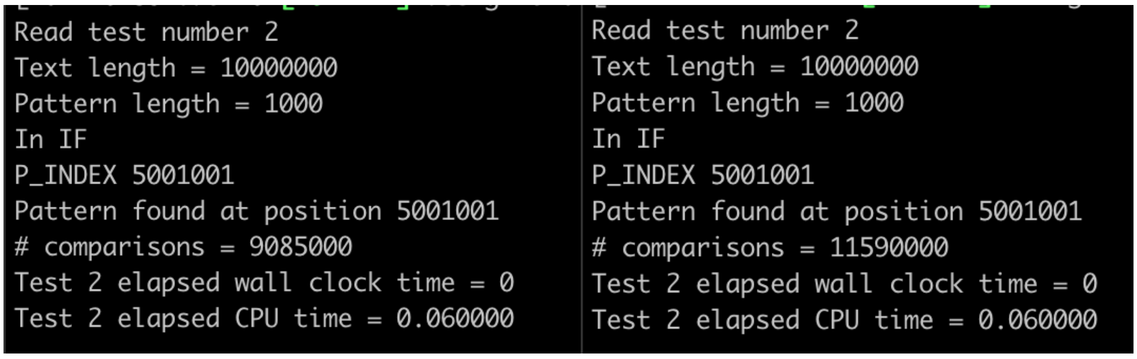
As shown below, the concurrency through threads is still being achieved, and whilst it results in a much larger overall wall clock (as opposed to the previous test) test2 requires the program to iterate over all the text, not just to the location of the first occurrence of the pattern.

	Wall Clock Time	Parallel Speedup	Parallel Efficiency
1 Thread	23	1	1
2Threads	12	6	3
4 Threads	6	1.5	0.375
8 Threads	5	0.625	0.078125
16 Threads	4	0.25	0.015625
32 Threads	4	0.125	0.00390625
64 Threads	4	0.0625	0.00097656



```
Read test number 2
Text length = 10000000
Pattern length = 1000
Pattern found at position 5001001
# comparisons = 5001002000
Test 3 elapsed wall clock time = 14
Test 3 elapsed CPU time = 13.510000
```

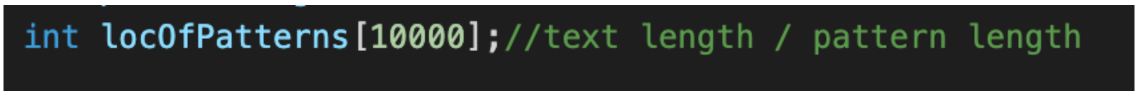
Figure 1.0 – Kelvin running the search sequential program for test1



```
Read test number 2
Text length = 10000000
Pattern length = 1000
In IF
P_INDEX 5001001
Pattern found at position 5001001
# comparisons = 9085000
Test 2 elapsed wall clock time = 0
Test 2 elapsed CPU time = 0.060000
```

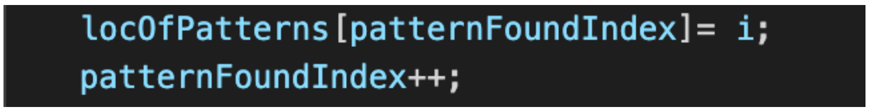
```
Read test number 2
Text length = 10000000
Pattern length = 1000
In IF
P_INDEX 5001001
Pattern found at position 5001001
# comparisons = 11590000
Test 2 elapsed wall clock time = 0
Test 2 elapsed CPU time = 0.060000
```

Figure 1.1 – Kelvin running the Searching OMP program for test1 on 2 threads (left) and test1 on 4 threads (right)



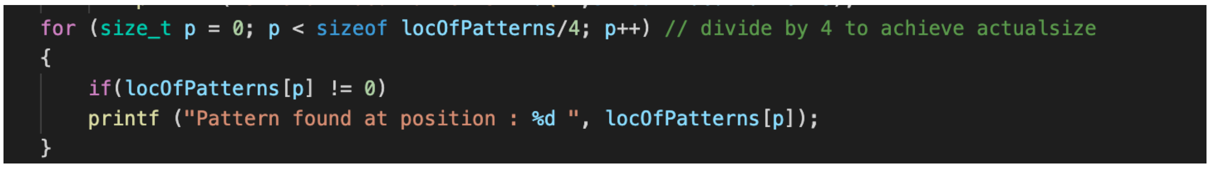
```
int locOfPatterns[10000];//text length / pattern length
```

Figure 1.2 – Initialisation of the array to store pattern locations



```
    locOfPatterns[patternFoundIndex]= i;
    patternFoundIndex++;
```

Figure 1.3 – Initialisation array to value of I and increment of the index



```
for (size_t p = 0; p < sizeof locOfPatterns/4; p++) // divide by 4 to achieve actualsize
{
    if(locOfPatterns[p] != 0)
    printf ("Pattern found at position : %d ", locOfPatterns[p]);
}
```

Figure 1.4 – Print statement used to now print the positions where the patterns are found

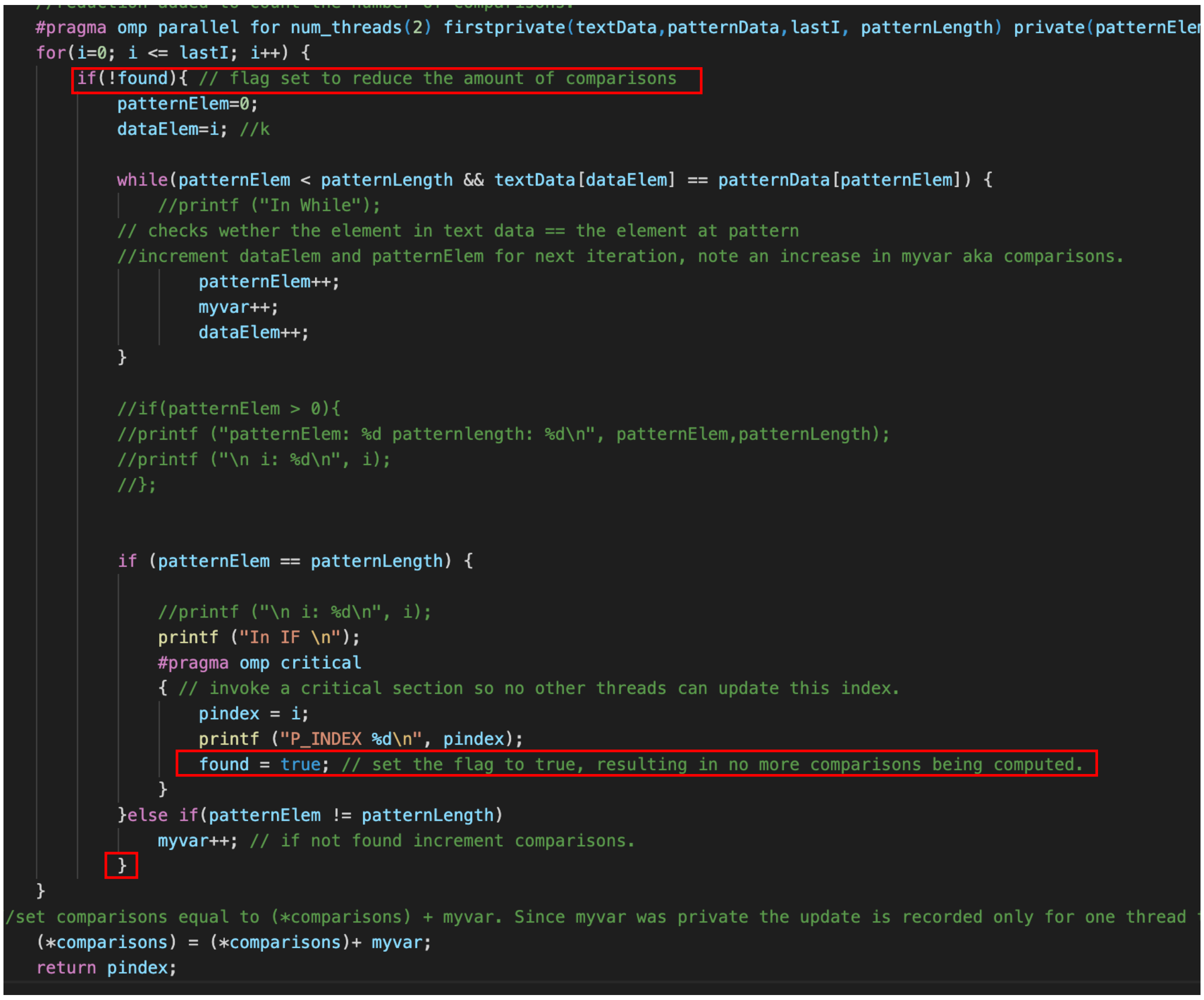
Conclusion

In conclusion the above tests demonstrate how a sequential program can be modified to run in parallel using thread manipulation. Test A provided the initial creation of the program and was used as a staple template for the extension of test1 and test2. The results show how overall performance of a program can be enhanced due to concurrency throughout the program.

References

[1] Gcc.gnu.org. 2021. *Optimization Levels (GNAT User's Guide for Native Platforms)*. [online] Available at: <https://gcc.gnu.org/onlinedocs/gnat_ugn/Optimization-Levels.html> [Accessed 10 October 2021].

Appendix



```
    #pragma omp parallel for num_threads(2) firstprivate(textData,patternData,lastI, patternLength) private(patternEle
    for(i=0; i <= lastI; i++) {
        if(!found){ // flag set to reduce the amount of comparisons
            patternElem=0;
            dataElem=i; //k

            while(patternElem < patternLength && textData[dataElem] == patternData[patternElem]) {
                //printf ("In While");
            // checks wether the element in text data == the element at pattern
            //increment dataElem and patternElem for next iteration, note an increase in myvar aka comparisons.
                    patternElem++;
                    myvar++;
                    dataElem++;
            }

            //if(patternElem > 0){
            //printf ("patternElem: %d patternlength: %d\n", patternElem,patternLength);
            //printf ("\n i: %d\n", i);
            //};


            if (patternElem == patternLength) {

                //printf ("\n i: %d\n", i);
                printf ("In IF \n");
                #pragma omp critical
                { // invoke a critical section so no other threads can update this index.
                    pindex = i;
                    printf ("P_INDEX %d\n", pindex);
                    found = true; // set the flag to true, resulting in no more comparisons being computed.
                }
            }else if(patternElem != patternLength)
                myvar++; // if not found increment comparisons.
            }
    }
/set comparisons equal to (*comparisons) + myvar. Since myvar was private the update is recorded only for one thread
    (*comparisons) = (*comparisons)+ myvar;
    return pindex;
```

Appendix A: Code showing implementation of Boolean flag

```
Read test number 2 Text length = 10000000 Pattern length = 1000
Pattern found at position : 5001001 Pattern found at position : 5002002 Pattern found at
position : 5004002 Pattern found at position : 5007001 Pattern found at position : 5010999
Pattern found at position : 5015996 Pattern found at position : 5021992 Pattern found at
position : 5028987 Pattern found at position : 5036981 Pattern found at position : 5045974
Pattern found at position : 5055966 Pattern found at position : 7554587 Pattern found at
position : 5066957 Pattern found at position : 5078947 Pattern found at position : 5091936
Pattern found at position : 5105924 Pattern found at position : 5120911 Pattern found at
position : 7626517 Pattern found at position : 5136897 Pattern found at position : 5153882
Pattern found at position : 5171866 Pattern found at position : 5190849 Pattern found at
position : 7699446 Pattern found at position : 5210831 Pattern found at position : 5231812
Pattern found at position : 5253792 Pattern found at position : 5276771 Pattern found at
position : 7773374 Pattern found at position : 5300749 Pattern found at position : 5325726
Pattern found at position : 5351702 Pattern found at position : 7848301 Pattern found at
position : 5378677 Pattern found at position : 5406651 Pattern found at position : 7924227
Pattern found at position : 5435624 Pattern found at position : 5465596 Pattern found at
position : 5496567 Pattern found at position : 8001152 Pattern found at position : 5528537
Pattern found at position : 5561506 Pattern found at position : 8079076 Pattern found at
position : 5595474 Pattern found at position : 5630441 Pattern found at position : 5666407
Pattern found at position : 8157999 Pattern found at position : 5703372 Pattern found at
position : 5741336 Pattern found at position : 8237921 Pattern found at position : 5780299
Pattern found at position : 5820261 Pattern found at position : 8318842 Pattern found at
position : 5861222 Pattern found at position : 5903182 Pattern found at position : 8400762
Pattern found at position : 5946141 Pattern found at position : 5990099 Pattern found at
position : 8483681 Pattern found at position : 6035056 Pattern found at position : 6081012
Pattern found at position : 8567599 Pattern found at position : 6127967 Pattern found at
position : 8652516 Pattern found at position : 6175921 Pattern found at position : 6224874
Pattern found at position : 8738432 Pattern found at position : 6274826 Pattern found at
position : 6325777 Pattern found at position : 8825347 Pattern found at position : 6377727
Pattern found at position : 8913261 Pattern found at position : 6430676 Pattern found at
position : 6484624 Pattern found at position : 9002174 Pattern found at position : 6539571
Pattern found at position : 6595517 Pattern found at position : 9092086 Pattern found at
position : 6652462 Pattern found at position : 9182997 Pattern found at position : 6710406
Pattern found at position : 6769349 Pattern found at position : 9274907 Pattern found at
position : 6829291 Pattern found at position : 9367816 Pattern found at position : 6890232
Pattern found at position : 6952172 Pattern found at position : 9461724 Pattern found at
position : 7015111 Pattern found at position : 9556631 Pattern found at position : 7079049
Pattern found at position : 7143986 Pattern found at position : 9652537 Pattern found at
position : 7209922 Pattern found at position : 9749442 Pattern found at position : 7276857
Pattern found at position : 7344791 Pattern found at position : 9847346 Pattern found at
position : 7413724 Pattern found at position : 9946249 Pattern found at position : 7483656
# comparisons = 9949051000
Test 2 elapsed wall clock time = 12
Test 2 elapsed CPU time = 32.939999
```

Appendix B: Results of where the pattern is found where T=2. After calculation 100 results were found.