

CSC4005 – High Performance Computing: Principles of Parallel Programming

Assignment 1: Sequential Searching in C and Parallel Searching Using OpenMP

Adam Coyle

40178464

September – October 2020

Table of Contents

Abstract.....	3
Introduction	3
Part A – Sequential Searching.....	3
Creating Test Cases	3
Character Sequence Selection for test cases.....	3
Selection of Text and Pattern lengths.....	4
Generation of test cases	5
Worst Case performance of the sequential searching algorithm.....	5
Compiling and running the sequential searching algorithm.....	5
Modifying the timing method of the program	5
Results.....	6
Part B – Parallel Searching Using OpenMP	10
Test 0.....	10
Modifying the main function	10
Searching OMP 0 implementation.....	10
Results.....	12
Test 1.....	15
Evaluating of Test 1 on the Searching Sequential and Searching OMP 0 programs.....	15
Searching OMP 1 implementation.....	15
Results.....	17
Test 2.....	18
Evaluation of Test 2 on the Sequential Search and Searching OMP 1.....	18
Searching OMP 2 implementation.....	18
Results.....	21
Conclusion.....	22
References	23

Abstract

The straightforward sequential searching algorithm is a well-known method of locating a pattern within a text, with performance ranging from $O(n)$ to $O(nm)$. Its limitations become embarrassingly clear whenever performance leans towards the worst-case, while its best-case performance is impressive. In this report, we will compare the performance of a sequential searching algorithm across 20 test cases, for differing compiler optimisations for Part A. In this experiment, we will observe any performance impact compiler optimisation flags might have on the algorithm. In Part B we will compare the algorithm to three variations of a parallelised searching algorithm using OpenMP, across 3 test cases, to weigh up the advantages and disadvantages of parallelisation. As it turns out, performance varies per test case, with some tests clearly showing the benefits of parallelising the sequential search, and others highlighting its drawbacks.

Introduction

Parallelisation of computer programs has become increasingly important in recent years. Upon reaching an upper limit in the power provided by single core chips, manufacturers have moved to designing processing units with multiple processing cores as a workaround. Software engineers now need to develop computer programs to take advantage of the increased core count of commercial chips. In theory, this should allow for a speedup proportional to the number of cores used to divide a given workload, but the use of multiple processors gives rise to new issues: optimal partitioning of the workload, to be carried out by the programmer or (at the risk of reduced efficiency) the compiler; scheduling of tasks to minimise stalling, and synchronisation of multiple cores, which brings additional overheads from the need to keep caches and memory coherent.

Part A – Sequential Searching

This section of the report will cover the results obtained from running the sequential searching algorithm using 2 different optimisation flags at compilation, on 20 tests generated using a bespoke C program.

Creating Test Cases

Before testing the performance of sequential searching algorithm, 20 test cases needed to be created within an inputs folder relative to the C program. These test cases were to be divided into 5 groups representing various $\text{length}(\text{text}) * \text{length}(\text{pattern})$ products, from 10^2 up to 10^{10} , each group increasing by a power of 2.

To create the required test cases, there were some prerequisite questions to be answered:

1. What sequence of characters would be contained within the text and pattern files?
2. Which combinations of text and pattern length should be used for each group?

Character Sequence Selection for test cases

When choosing the contents of the test files, I refrained from using random character strings, since there was no way to ensure control over whether a given test would obtain the worst-case performance. The simplest way was to use a traditional text/pattern for the algorithm [] consisting of repeating 'A' characters, with a single 'B' character at the end. This would ensure that the pattern is always found at the end of the text, and that the algorithm would make an extra P comparisons for every character in the text, where P = pattern length, since $P - 1$ characters would match the text.

Selection of Text and Pattern lengths

Since the contents of each text and pattern file could be known beforehand, as well as the how the algorithm searched for a pattern in a text, it became possible to calculate the number of comparisons that the algorithm would make on a given text-pattern combination.

The formula for calculating the number of comparisons for a given test could be given as:

$$\text{Comparisons} = \text{Product} - (P - 1) * P$$

Where P is the length of the pattern for that test case. Knowing the number of comparisons in advance for a given text-pattern combination was an important piece of the puzzle in selecting appropriate combinations to run the algorithm on.

We could also make assumptions about possible text-pattern combinations using the algorithms code; since the last character index to search in a text is given by:

$$\text{Index} = T - P$$

Where T is the text length and P is the pattern length, we can assume that $T \geq P$ since it would otherwise result in an out of bounds error. This greatly reduced the number of possible text-pattern combinations, particularly for the larger product lengths.

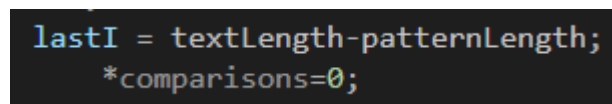


Figure 1 - Calculation of the Last Index to search in the Searching Sequential Algorithm

It was decided to use brute force generation to enumerate all possible text-pattern combinations for all product lengths, given the above constraints. Initially, pen and paper working out was used to select combinations, which was useful for evaluating the algorithm.

However, once I had spent more time on the problem, I considered that brute force would be more thorough in providing combinations.

From the output of the brute force script, I chose combinations across a range of comparisons where possible, since some products such as 10^2 have only 5 choices or other products such as 10^{10} had combinations impossible to select given the file size constraints i.e. a text length of 10^{10} and a pattern length of 1 would result in a file size some Gigabytes large. Initially I had opted to choose combinations which resulted in the maximum number of comparisons, but I ran into the file size issue making such a selection process unviable. Fortunately, for larger products, there were several choices of combination which would result in similar comparison numbers.

The number of comparisons for a given product could range from the square root of the product (minimum) since that was the largest length the pattern could take within the given constraints, up to the product itself (maximum) using a pattern length of only 1. The table below illustrates the chosen combination choices:

Table 1 - The selected text-pattern combinations to run the algorithm on

10^2		10^4		10^6		10^8		10^{10}	
T	P	T	P	T	P	T	P	T	P
10	10	100	100	1000	1000	10000	10000	100000	100000
20	5	125	80	1600	625	15625	6400	128000	78125
25	4	200	50	4000	250	80000	1250	250000	40000
100	1	10000	1	1000000	1	1000000	100	1000000	10000

The broad range of text-pattern combinations allows for the program to obtain performances ranging from best- to worst-case. The justification for this choice was to allow for a better evaluation of the algorithm by observing its performance for varying combinations.

Generation of test cases

Test cases are generated in the `test_maker.c` program, which creates the required files and their associated directories. The main function of the program does the following operations:

1. 20 directories are created in the inputs folder for the sequential program to search, with each folder following the naming convention of `testi`, where `i` is the i^{th} directory created by the loop, starting from 0 up to and including 19.
2. The chosen text-pattern combinations are read from five different files into an array of `TextPatternSize` structs. These structs contain the text and pattern lengths for a single test case. The files are in a data folder, with each file corresponding to one of the product lengths to investigate.
3. For every element in the array of `TextPatternSizes`, the program locates the relevant test directory (created in step 1) and writes a text and pattern `txt` file to that location.
4. The program writes the character 'A' from index 0 up to, but not including index `N-1`, where `N` is either the text or pattern length. A 'B' character is inserted as the final element of the sequence at index `N`, and the program terminates.

Worst Case performance of the sequential searching algorithm

The worst-case performance of the algorithm is $O(P*(T-P+1))$ where `T` is the text length and `P` is the pattern length [1].

Since the test files consisted only 'A' characters and a final 'B' character at the end, a pattern emerged around the number of comparisons for a given test. Therefore, it was possible to correctly calculate the number of comparisons for a given text-pattern test, assuming they followed the same character structure.

Compiling and running the sequential searching algorithm

Modifying the timing method of the program

Observation of the output file revealed several tests report both a wall time and elapsed CPU time of 0. This might have been the result of the clock and time functions having limited accuracy in their estimations of program time [2]. The functions were modified slightly to observe any changes in reported times, including formatting the results to report up to 12 decimal places (up from 6), but the output was the same.

One solution would have been to estimate the elapsed time from test cases which had an output not equal to zero, divide the time by the number of comparisons to find the time for 1 comparison, and finally estimate the time for those tests reporting 0 time.

The chosen solution to this issue was to use the `timespec_get` function and retrieve the result from the `timespec` struct [3]. The benefit of this method was that it allowed elapsed times to be report in nanoseconds (10^{-9}) over the original microsecond accuracy of the program.

```
long get_nanos(void)
{
    struct timespec ts;
    timespec_get(&ts, TIME_UTC);
    return (long)ts.tv_sec * 1000000000L + ts.tv_nsec;
}
```

Figure 2 - The getNanos function allowed for improved accuracy in record elapsed CPU time.

One issue with this method was that while it ran on an Ubuntu virtual machine, it would result in errors when running on the Kelvin cluster. This occurred due to the default C compiler not recognising the TIME_UTC const, which is declared in the C11 standard. Therefore, the compile instruction in the run scripts (and future run scripts) needed to be appended:

```
gcc -O2 searching_sequential.c -o searching_sequential_2 -std=c11
```

```
gcc -O0 searching_sequential.c -o searching_sequential_0 -std=c11
```

Figure 3 - The modified compiler instructions, using the C11 standard.

Once the C standard had been specified in the instruction, the program could be compiled and executed on Kelvin successfully. Another added benefit of using this method was that it needed only 2 function calls in contrast to the original method which made 2 calls to the clock function, and 2 to the time function.

```
while (readData (testNumber))
{
    long timeStart, timeEnd;

    timeStart = getNanos();
    processData();
    timeEnd = getNanos();

    long elapsedNsec = (timeEnd - timeStart);

    printf("\nTest %d elapsed wall clock time = %ld\n", testNumber,
(long) (elapsedNsec / 1.0e9));
    printf("Test %d elapsed CPU time = %.09f\n\n", testNumber,
(double) elapsedNsec / 1.0e9);
    testNumber++;
}
```

Figure 4 - The modified while loop in the program's main function.

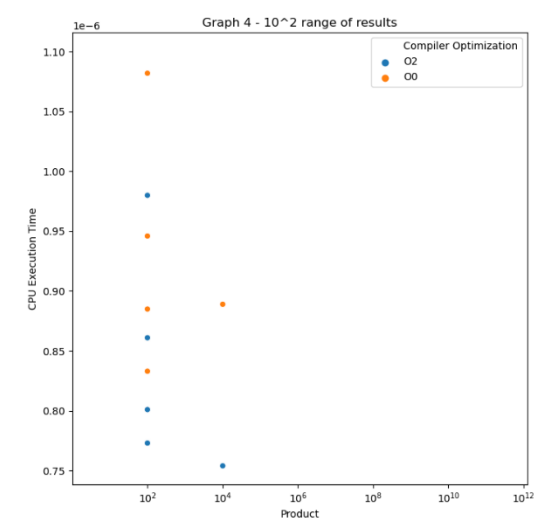
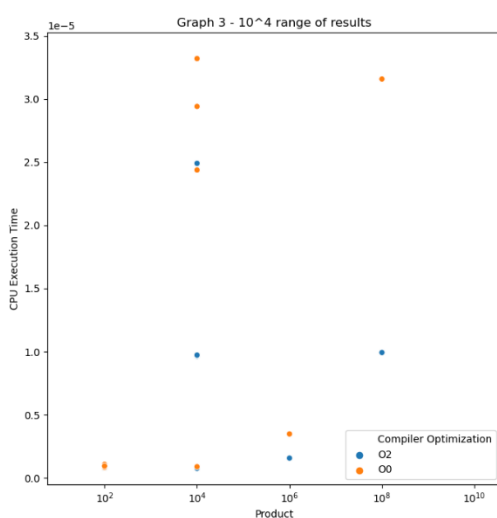
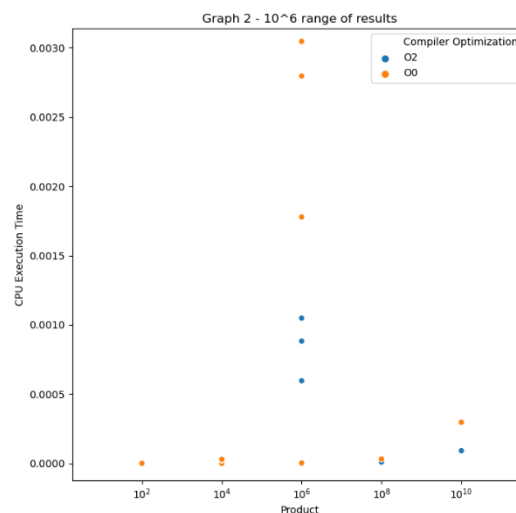
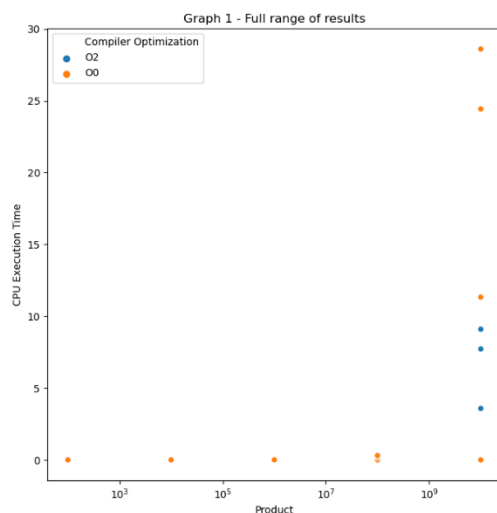
Results

A single run script was used to execute both variations of the program, with different compiler optimisation for each. The output file contained the results of both programs and the results were parsed in a Jupyter Notebook. The decision to use python and Jupyter was that it was relatively easy to read through the output file and parse the relevant data and also because of the capabilities of libraries such as Pandas and Seaborn for tabulating and plotting the data. While less thought would be required to manually type the results into an excel table, the benefit of using a Python script to read the data was that it would minimise human error that might occur from manual typing. More

importantly, the sequential program was run on Kelvin several times before settling on a final set of results, so it was more practical to have the output read automatically.

The results of both programs showed that using compiler optimisation has a marked effect on performance, for example when using text and pattern lengths of 1000000 and 10000 respectively, the program using O2 compiler optimisation ran the test in 9.1 seconds. When using no compiler optimisation, the same test took 28.6 seconds to complete. However, these results are unsurprising given that setting the O0 flag disables most compiler optimisations [4], while O2 applies even more optimisations (than O0 and O1). The trade-off of using compiler optimisation is the increased compilation time, which was not the case with this program, which compiled in under 1 second for both optimisation flags. This can be attributed to the relatively short amount of source code to compile.

Visualising the results was difficult considering the range of the execution times reported. Tests in the smaller product range for example reported run times only some hundreds of nanoseconds long, while the longest test took 28.6 seconds.



Using just the above figures, it is difficult to discern a relationship between the product of text-pattern lengths and the execution time for a test case. This is due to the variation in timings within product groups, for example tests 16 and 19 took 92 microseconds (10^{-6}) and 9.09 seconds, respectively. This pattern occurs across all product groups since the text-pattern lengths were chosen to represent the range of possible combinations from $T = P$ up to $T = TP$, where T is the text length and P is the pattern length.

Unfortunately, it was not feasible to plot the times for the product group 10^8 due to the limited number of graphs that could be plotted. Therefore, those results were left out on the basis that they provided less insight than the others:

1. Graph 1 displays the full range of elapsed CPU times for all product lengths and highlights the sheer difference in magnitude between execution times for smaller product lengths and larger product lengths.
2. Graph 2 shows that some tests using considerably larger product lengths performed better than most tests in this product group.
3. Graph 3 again highlights the above observation; the difference in magnitude is emphasised when comparing the times for 10^4 in Graph 2, where they almost look to be sharing the same point, and in Graph 3 where they are fairly distributed within their own range.
4. Graph 4 has perhaps the most interesting observation; a product of 10^4 obtained the shortest execution time with a narrow lead over the shortest time of 10^2 (a mere 19 nanoseconds), despite making 10 times as many comparisons. However, this result might just reflect the difficulty in accurately measuring execution time for trivial programs, and highlights that even the use of the `timespec_get` function is not perfect for this scenario.

While the elapsed CPU time for a test has been observed to be inconsistent with the product for that test, a relationship might be established between the *number of comparisons* and the elapsed time.

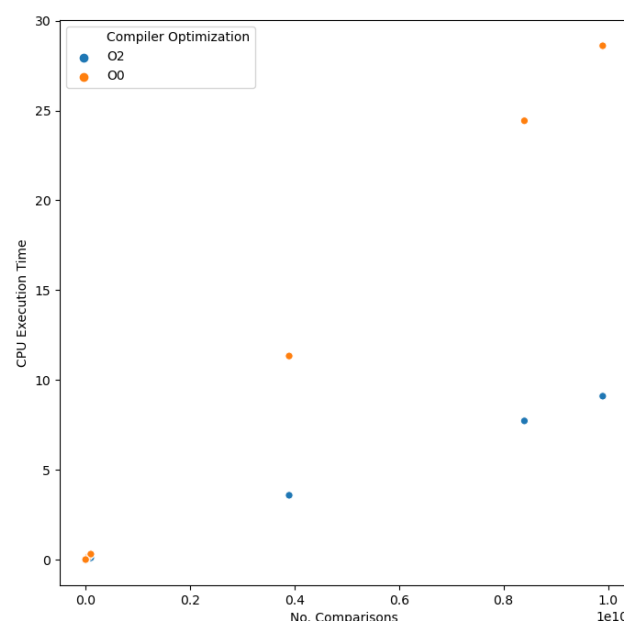


Figure 5 - The linear relationship between the number of comparisons and elapsed CPU time.

The above figure shows a very clear correlation between the number of comparisons and the elapsed CPU time, with the optimisation flag having a marked effect on the execution time for the same number of comparisons of a test. However, these results are rather unsurprising since a comparison is supposed to represent the program instructions which require actual processing power. The key takeaway for these results is that the elapsed CPU time does not scale with the product length of a test, since a test can make a broad range of comparison counts for a given product.

Part B – Parallel Searching Using OpenMP

This section will focus on the creation of an increasingly modified searching algorithm parallelised using OpenMP constructs and will compare its performance against the original sequential searching algorithm across 3 provided test cases. The parallelised program will be compiled to use a varying number of threads as well as different scheduling strategies. The objective of these comparisons is to highlight cases where parallelism of the searching algorithm will yield either an improvement or deterioration in performance compared to the sequential search.

Test 0

In this test, both programs search for a single occurrence of a pattern of length 2000 in a text of length 2 million (2×10^6). A parallelised version of the sequential search was implemented using a thread count ranging from 1 to 64, with 4 different scheduling strategies: static, dynamic, guided and auto.

Modifying the main function

Modifications were made to the original program such that the test number could be passed into the main function when executing the program from the command line. This removed the issue where the program would run all tests in succession, and therefore it became easier to search the output file for results.

```
int main(int argc, char **argv)
{
    if (argc < 1)
    {
        printf("\nRead Default - not enough arguments.\n");
        readDefault();
    }
    else
    {
        int testNumber = atoi(argv[1]);
        runTest(testNumber);
    }
}
```

Figure 6 - The modified main function in searching_sequential.c

The main function within the parallel searching program was tweaked slightly such that it took in 2 additional parameters when executing from the command line: the minimum and maximum number of threads. This flexibility allowed for different iterations to vary the number of threads used as per the requirements of each test.

```
int testNumber = atoi(argv[1]);
int minThreads = atoi(argv[2]);
int maxThreads = atoi(argv[3]);
runTest(testNumber, minThreads, maxThreads);
```

Figure 7 - Modified runTest function to take the minimum/maximum threads as arguments.

Searching OMP 0 implementation

The implementation for the parallel searching algorithm assumed that the number of text characters to loop through would be distributed evenly among the number of threads. Therefore, the algorithm effectively acts as several smaller sequential searches running in parallel.

```

long localComparisons = 0;

int patternLoc = -1;
int breakSearch = 0; // used to stop comparing text-patterns and move to
next text character.

#pragma omp parallel for reduction(+: localComparisons) \
shared(patternLoc) private(j,k) firstprivate(breakSearch) \
num_threads(thread_count) schedule(runtime)
for (i = 0; i <= lastI; i++)
{
    k = i;
    j = 0;
    while (j < patternLength && breakSearch == 0)
    {
        localComparisons++;
        if (textData[k] == patternData[j])
        {
            k++;
            j++;
        }
        else
        {
            breakSearch = 1;
        }
    }

    // condition can only be true once assuming only one occurrence of
    pattern in text.
    if (j == patternLength && patternLoc == -1)
    {
        patternLoc = i;
    }
    else
    {
        breakSearch = 0;
    }
}
(*comparisons) = localComparisons;
return patternLoc; // returns -1 if not found.

```

Figure 8 - The parallel searching algorithm implemented in the hostMatch function

The parallel search uses several key OpenMP directives in the implementation to find the pattern while producing the same number of comparisons as the sequential search:

- The reduction clause allows each thread to have its own comparison count, all of which are then combined upon exiting the loop. This was a performant alternative to flagging the comparisons variable as shared, which resulted in worse execution time than the sequential search.
- The patternLoc variable must be flagged as shared since it is not known which thread will find the pattern (if it exists), therefore all threads need the option to write to it.
- Since the variable i is autoincremented every iteration, an additional variable needed to be introduced to break out of the while condition. The alternative behaviour would be that variable j would be reset to 0 when a mismatch occurred, and the program would be unable

to move to the next iteration because the while condition could not be broken. The breakSearch variable ensures another condition can be met to break out of the while loop to ensure the program continues searching.

- Variables j and k are set to private since they are reset for every iteration of the loop, and therefore their default values would be immediately overwritten if declared in the first private construct. Additionally, k needs to be set to j, making its initial value of 0 redundant.
- Since the test is experimenting with different thread counts for the program, we declare the num_threads construct to allow for this alteration.

Results

The parallel search reported the same number of comparisons as the sequential search, and successfully found the pattern at the correct index. Despite the similar comparison count, the parallel search was markedly faster.

Table 2 - Results of Test 0 using various thread and scheduling strategies, using 8 cores. Not all results are shown.

Threads	Wall Clock Time	Elapsed CPU Time	No. Comparisons	Scheduling	Parallel Speedup	Parallel Efficiency
1	10	10.34050896	3996002000	static	1.134036461	1.134036461
2	5	5.209357291	3996002000	static	2.251048168	1.125524084
4	2	2.875149359	3996002000	static	4.078575657	1.019643914
8	1	1.482056666	3996002000	static	7.91232512	0.98904064
1	10	10.75634842	3996002000	dynamic	1.090194714	1.090194714
2	5	5.448220323	3996002000	dynamic	2.152356823	1.076178412
4	2	2.998635025	3996002000	dynamic	3.910617361	0.97765434
8	1	1.55364359	3996002000	dynamic	7.547750502	0.943468813
1	10	10.34569141	3996002000	guided	1.13346839	1.13346839
2	5	5.250584545	3996002000	guided	2.233373082	1.116686541
4	2	2.882085218	3996002000	guided	4.068760394	1.017190098
8	1	1.484871716	3996002000	guided	7.897324773	0.987165597
1	10	10.33048892	3996002000	auto	1.135136418	1.135136418
2	5	5.200771516	3996002000	auto	2.254764346	1.127382173
4	2	2.86153855	3996002000	auto	4.097975261	1.024493815
8	1	1.480172886	3996002000	auto	7.922394943	0.990299368

Note that for each scheduling strategy, no chunk size was specified in the environment variable, since all threads would take on identical workloads with the same number of iterations/comparisons.

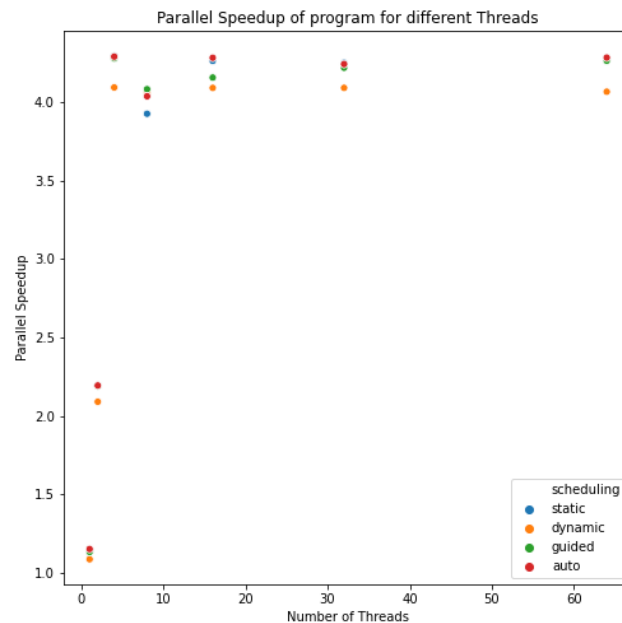


Figure 9 - The parallel speedup of the modified algorithm for increasing thread counts, using 4 cores.

From the results, several observations can be made about the parallel searching algorithm. Most importantly, there is a noticeable speedup when the number of threads is increased, which plateaus when using beyond 4 threads. This is a consequence of the number of threads exceeding the number of cores available, since the workload is distributed as 1 thread per core. Therefore, as the thread count increases, the number of threads per core exceeds 1, but the core only works on 1 thread at a time. This observation was confirmed when the run script was modified to increase the number of CPUs per task:

```
#SBATCH --cpus-per-task=4
#SBATCH --cpus-per-task=8
```

The increase in available CPU cores allowed the parallel speedup to increase again, until the number of threads exceeds 8, after which performance ceases to increase.

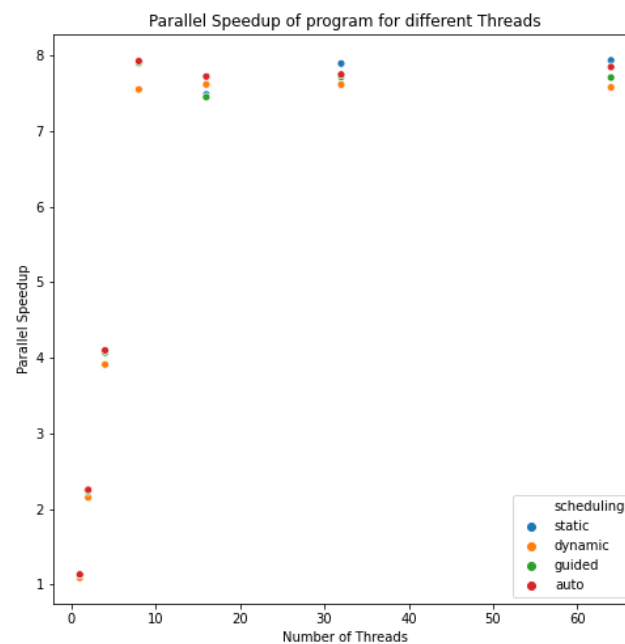


Figure 10 - The parallel speedup of the modified algorithm for increasing thread counts, using 8 cores.

Another observation to be made from the results is the effect of using various scheduling strategies on the program. Using dynamic scheduling appears to result in reduced performance across all thread counts, with exceptions when the number of threads is equal to double the number of cores. Other scheduling strategies report very similar results for each thread setting. The reduced performance can likely be attributed to the overhead required by using dynamic scheduling [5], and since the workload is evenly distributed among threads, its main purpose is redundant and it only served to degrade the performance of the algorithm.

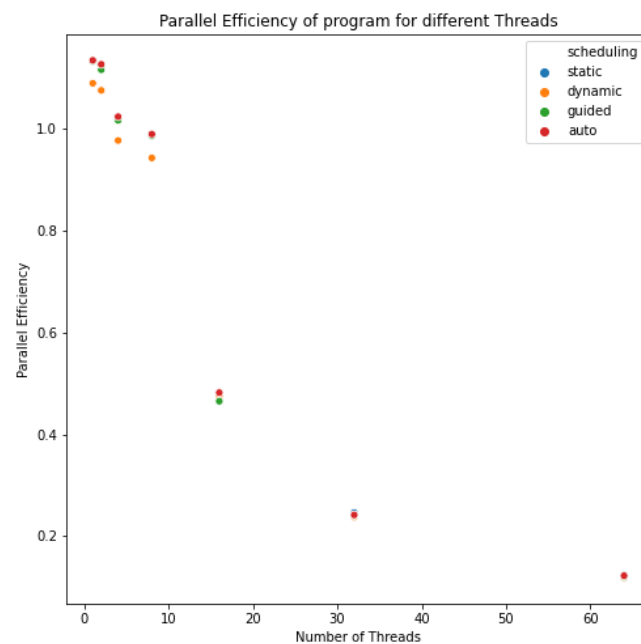


Figure 11 - The parallel efficiency of the program with increasing thread counts, using 8 cores.

The parallel efficiency of the program experiences a slight reduction as the thread count increases, until the number of threads exceeds the number of cores where efficiency converges towards 0 in an asymptotic fashion. Although this was to be expected since the speedup of the algorithm is being throttled by the CPU.

Test 1

This test involves searching a text of length 10 million (1×10^7) for a pattern of length 1000, where the pattern does not occur at the end of the text, unlike in the previous text. This test was carried out on three programs: the sequential search, the parallel search from test 0, `searching_OMP_0` and a modified parallel search based on the first, `searching_OMP_1`.

Evaluating of Test 1 on the Searching Sequential and Searching OMP 0 programs

The below comparison shows the considerable difference in comparison counts for the test, with the program used in the previous test demonstrating how ill-suited its algorithm was for this test. It would require some modification before it could be fairly compared to the sequential search.

Table 3 - Results of Test 1 for the sequential search and the parallel search (no. threads = 2).

Algorithm	No. Comparisons	Elapsed Time
Sequential Search	1002000	0.002994074
Parallel Search (<code>searching_OMP_0</code>)	9998501500	14.154749475

Searching OMP 1 implementation

Very little needed to be altered for this test case; the algorithm in test 0 searched every text character up until the last index `lastI`, for the pattern, regardless if it was found before searching all indexes. Therefore, the key consideration when adapting the algorithm to this test case, was that it should 'exit' the search as soon as the pattern was found. Unfortunately, OpenMP does not have a

mechanism to allow this [6]. Instead of adding an additional Boolean value, it was sufficient to use the patternLoc variable, where the initial value of -1 would denote that the pattern was not found, and any value of 0 or greater would indicate that the pattern was found.

```
for (i = 0; i <= lastI; i++)
{
    // if we find the pattern, stop making comparisons and loop until end
    if (patternLoc >= 0)
    {
        continue;
    }
    else
    {
        //printf("Thread %i at index %i\n", omp_get_thread_num(), i);
        k = i;
        j = 0;
        // since the pattern can be found in the middle of the loop
        // we check if pattern is found every loop to stop making com-
        parisons as soon as possible.
        while (j < patternLength && breakSearch == 0 && patternLoc == -
1)
        {
            localComparisons++;
            if (textData[k] == patternData[j])
            {
                k++;
                j++;
            }
            else
            {
                breakSearch = 1;
            }
        }
        // condition can only be true once assuming only one occurrence
        of pattern in text.
        if (j == patternLength && patternLoc == -1)
        {
            patternLoc = i;
        }
        else
        {
            breakSearch = 0;
        }
    }
}
```

Figure 12 - The modified parallel for loop in the hostMatch function of searching_OMP_1

The use of this additional check was sufficient to substantially reduce the number of comparisons made by the parallel search. When running the tests for varying different thread counts, no scheduling environment variable was set before running the programs. Instead, scheduling was determined at runtime by an internal control variable, run-sched-var [7]. It should be noted that increasing the thread count produced different comparison counts each time, this is likely due to the existence of a race condition between the thread that locates the pattern and the others searching before the breakout flag is set. Therefore, the reported results for using 2 threads were averaged from a sample size of 10.

Results

Table 4 - Results for the searching_OMP_1 algorithm, without a set schedule environment variable.

Number of Threads	Number of Comparisons	Wall time (s)	Elapsed CPU time (s)
1	1002000	0	0.143652578
2	1002288	1	0.439131130

The race condition resulting from threads trying to find the pattern first is a consequence of the pattern being located early in the text, combined with using runtime scheduling creating chunks for which there was no known method of handling this. Therefore, additional tests were carried out for the same scheduling strategies used in test 0. Each scheduling strategy ran 10 tests each, with the environment variable setting the number of chunks to 2 (except for auto) and using 2 threads. The aim of these experiments was to observe which scheduling strategies might provide an improvement in the number of comparisons relative to the sequential search.

Table 5 - Results from running 10 tests using different scheduling strategies.

Schedule	Min. Comparisons	Avg. Comparisons	Max. Comparisons
static	997219	1001458	1003679
dynamic	1002076	1003026	1003980
guided	1997993	2001808	2004427
auto	1999683	2003288	2007543

The guided and auto schedules can safely be excluded from the comparison since they performed almost twice as many comparisons as the sequential search. This is likely due to guided scheduling being better suited to imbalanced workloads, since its chunk sizes are proportional to the number of unassigned iterations divided by the number of threads in the team [8]. The auto schedule uses a default strategy which is compiler dependent, in the case of gcc it defaults to static [9] without specifying the number of chunks.

Static and dynamic scheduling run the program on par with the sequential search, with static scheduling reporting 542 fewer comparisons on average. This might be attributed to how these strategies tend to use chunk sizes equal to the number of iterations divided by number of chunks. Therefore, it is possible the threads were assigned smaller chunk sizes and would make fewer comparisons since the pattern was found relatively early in the text.

Test 2

Similarly, to test 1, the programs must search for the pattern of length 1000 in a text of length 1×10^7 . However, in this case there are multiple instances of the pattern within the text. Therefore, the parallel search must be modified to detect all occurrences of the pattern within the text, and report the starting index of the first occurrence, in contrast to the sequential search and the parallel search from test 1, both of which stops searching at the first instance.

Evaluation of Test 2 on the Sequential Search and Searching OMP 1

When running the test using the sequential searching algorithm, the following results were obtained:

Table 6 - Results of the sequential search on Test 2.

Pattern Location	No. Comparisons	Wall Clock Time	Elapsed CPU Time
5001001	5001002000	14	14.614776586

The algorithm searched 5001001 out of a possible 9999000 text indexes, or about 50% of the text, before finding the pattern and stopping the search. Crucially, this shows that the algorithm does not meet the requirements of the sequential search, that is to find all instances of the pattern, since it left a further 99 pattern instances undetected in the second half of the text.

Table 7 - Results of the second parallel searching algorithm on Test 2.

Threads	No. Comparisons	Wall Clock Time	Elapsed CPU Time
1	5001002000	13	13.60369372
2	5001002990	7	7.006734862
4	5001002462	3	3.910661248
8	5001001023	4	4.076670193
16	5000997637	3	3.97225015
32	5000989919	4	4.050201237
64	5000976507	4	4.077213668

The second parallel searching algorithm, `searching_OMP_1`, also experiences the same issue of exiting the search after finding the first instance. It also contains a race condition wherein the thread processing the chunk with the first pattern is searching while other threads are searching beyond that index. However, this test case benefits from parallelisation, as demonstrated by the reduction in elapsed time as the thread count increases. This pattern continues until the number of threads exceeds the number of cores, which was set to 4 for this program. Like the sequential search, this algorithm also fails to detect the remaining 99 patterns.

Searching OMP 2 implementation

Therefore, with these results as a reference, the parallel search had to be modified for this test with two considerations in mind to avoid the above pitfalls:

1. It must be assumed that there exists more than one instance of the pattern, so the algorithm must be altered such that it returns the *first* instance.
2. The algorithm must detect all occurrences of the pattern. This means that no flags can be used to stop searching once a pattern has been found.

Therefore, the `hostMatch` function was modified to match these new requirements. Firstly, the `patternLoc` variable was set to `lastl + 1`, this allowed the program to check if a found patterns start

index was less than the current value of patternLoc, which would allow for the algorithm to detect the first occurrence.

A second reduction clause, localPatterns, was introduced in the OpenMP directive. This would ensure that each thread could count the number of pattern instances it detected in the text, and finally print the total result upon exiting the loop. Additionally, the if-else statement that was wrapped around the core search logic was removed, since it facilitated a loop exit upon finding a pattern.

It was necessary to wrap the setting of the patternLoc variable within an omp critical construct. This was to ensure only a single thread at a time could write to the variable, which was not needed in previous programs under the assumption that only 1 instance of the pattern existed, and therefore the patternLoc would only be written to once. In this test however, multiple threads might encounter a pattern instance concurrently, and attempt to write a value to patternLoc at the same time.

```
long localComparisons = 0;

int localPatterns = 0;
volatile int patternLoc = lastI+1;

int breakSearch = 0;

#pragma omp parallel for reduction(+: localComparisons) \
reduction(+: localPatterns) shared(patternLoc) \
private(j, k) firstprivate(breakSearch) \
num_threads(thread_count) schedule(runtime)
for (i = 0; i <= lastI; i++)
{
    k = i;
    j = 0;
    while (j < patternLength && breakSearch == 0)
    {
        localComparisons++;
        if (textData[k] == patternData[j])
        {
            k++;
            j++;
        }
        else
        {
            breakSearch = 1;
        }
    }

    if (j == patternLength)
    {
        printf("Found pattern at position:%i\n", i);
        localPatterns++;
        #pragma omp critical
        {
            if (i < patternLoc)
            {
                patternLoc = i;
            }
        }
    }
    else
    {
        breakSearch = 0;
    }
}
(*comparisons) = localComparisons;

printf("\n\nFound %i instances of pattern.\n\n", localPatterns);

if (patternLoc < lastI + 1)
{
    return patternLoc;
}
else
{
    return -1;
}
```

Figure 13 - The third parallel searching algorithm, engineered to detect all instances of a pattern and return the first occurrence.

Results

Unsurprisingly, the modified algorithm performed substantially more comparisons than the previous programs. However, the algorithm now meets those requirements of the sequential search: the tests performed using this algorithm successfully located all 100 instances of the pattern as well as reporting the first occurrence in the text. While it does perform approximately double the comparisons of those previous algorithms, the benefit of parallelism allows execution time to be reduced drastically. While limited by the 4 cores set in the run script, the algorithm searches the entire text in half the time that the sequential search took, while undertaking double the workload and detecting more patterns.

Table 8 - Results of the third parallel searching algorithm on Test 2.

Threads	Patterns Found	First Pattern Index	No. Comparisons	Wall Clock Time	Elapsed CPU Time
1	100	5001001	9949051000	26	26.82006845
2	100	5001001	9949051000	13	13.74160392
4	100	5001001	9949051000	7	7.522392383
8	100	5001001	9949051000	7	7.573625203
16	100	5001001	9949051000	7	7.509869528
32	100	5001001	9949051000	7	7.527940134
64	100	5001001	9949051000	7	7.547720764

To better assess the performance difference, performance can be measured in comparisons per second.

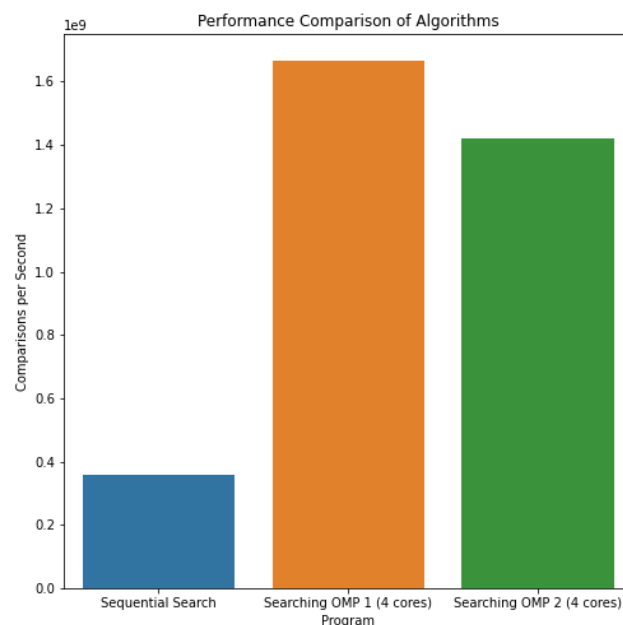


Figure 14 - Performance Comparison of the searching algorithms.

While the Searching OMP 1 program slightly outperforms the modified program, it still fails to meet the requirements of the algorithm. Additionally, due to the nature of the test pattern containing a 'B' character at the end, this reduces the number of comparisons to make by 1000 (the pattern length) for each occurrence of the pattern. Therefore, the modified parallel search made 100000 fewer comparisons, and the dip in performance relative to Searching OMP 1 is justified.

Conclusion

The performance of the sequential searching algorithm has been shown to vary for a given text-pattern product, with best- and worst-case performances differing greatly, in some cases by many several orders of magnitude. Performance has also been shown to be affected greatly by the chosen compiler optimisation, with no optimisation resulting in a threefold increase in execution time compared to level 2 optimisation.

Parallelisation of the sequential searching algorithm has been proven to have both benefits and detriments. A parallelised search algorithm yields considerable performance gains over a sequential search, most notably when the first instance of the pattern occurs in the second half of the text, as reported by the results of tests 0 and 2. Conversely, parallelisation was shown to have matched the sequential algorithm in terms of comparisons in test 2, at the expense of a marked increase in execution time resulting from the overheads of shared memory parallelisation. These results confirmed that while a parallel searching algorithm performs better when approaching worst-case performance, while struggling to match the sequential algorithm for best-case performance.

To conclude, the results of the various tests using different thread counts, scheduling strategies and source code modifications provides key information regarding the performance implications of parallelisation of a sequential program, which should be considered when deciding to implement such a modification.

References

- [1] K. Ka.patel, "The Naive String Matching Algorithm," 8th September 2019. [Online]. Available: https://medium.com/@krupa_110/the-naive-string-matching-algorithm-be7992ebbd1d.
- [2] "Why C clock() returns 0," Stack Overflow, 26th March 2012. [Online]. Available: <https://stackoverflow.com/questions/9871071/why-c-clock-returns-0>.
- [3] "How to measure time in milliseconds using ANSI C?," Stack Overflow, 31st October 2018. [Online]. Available: <https://stackoverflow.com/questions/361363/how-to-measure-time-in-milliseconds-using-ansi-c/36095407#36095407>.
- [4] "3.11 Options That Control Optimization," [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.
- [5] u. coincoin, "Is dynamic scheduling better or static scheduling (Parallel Programming)?," Stack Overflow, 4th June 2017. [Online]. Available: <https://stackoverflow.com/questions/29916732/is-dynamic-scheduling-better-or-static-scheduling-parallel-programming>.
- [6] "How to: Convert an OpenMP Loop that Uses Cancellation to Use the Concurrency Runtime," Microsoft, 11th April 2016. [Online]. Available: <https://docs.microsoft.com/en-us/cpp/parallel/concr/convert-an-openmp-loop-that-uses-cancellation?view=vs-2019>.
- [7] "Using OMP_SCHEDULE with #pragma omp for parallel schedule(runtime)," Stack Overflow, 19th March 2013. [Online]. Available: <https://stackoverflow.com/questions/15508128/using-omp-schedule-with-pragma-omp-for-parallel-scheduleruntime/15512872>.
- [8] "OpenMP Dynamic vs Guided Scheduling," Stack Overflow, 23rd May 2017. [Online]. Available: <https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling>.
- [9] "How is OpenMP "auto" schedule implemented in gcc?," Stack Overflow, 5th October 2017. [Online]. Available: <https://stackoverflow.com/questions/46580902/how-is-openmp-auto-schedule-implemented-in-gcc>.