CSC4005 | Assignment 1 | 40006161

Introduction

In this assignment I shall evaluate the CPU execution time of a string matching algorithm ran over 20 test inputs using two different C compilers with various optimisations. The string matching algorithm I will be using is contained with in a file called searching_sequential.c. It will search a given string for a given pattern reporting the starting index of the first occurrence of the pattern if it is in the text and a 'pattern not found' message otherwise. The code for this is shown in Figure 1 below.

```
int hostMatch(long *comparisons)
{
    int i,j,k, lastI;

    i=0;
    j=0;
    k=0;
    lastI = textLength-patternLength;
        *comparisons=0;

    while (i<=lastI && j<patternLength)
    {
                (*comparisons)++;
      if (textData[k] == patternData[j])
      {
            k++;
            j++;
      }
      else
      {
            i++;
            k=i;
            j=0;
      }
    }
    if (j == patternLength)
      return i;
    else
      return -1;
}
```

Figure 1 String Matching Algorithm

In order to ensure I was getting an accurate result I needed to give the algorithm a number of pattern and text combinations to be run on, 20 was deemed to be acceptable. To create this I generated a program called test_gen.c that would

create the file and directory structures required and populate them with the correct pattern and text. The assignment required that we analyse the worst case performance of the string searching algorithm. This is produced when the pattern is not found in the text and at each position every character of the pattern is matched apart from the last one.

I achieved this by creating a function which would create a text file with n 'A's and a pattern file which would contain m-1 'A's followed by a 'B'. Where n is the desired length of the text and m is the desired length of the pattern. The pattern creation method is shown in the following code snippet.

```
write_pattern(FILE* file, char body, char end,
                unsigned long size){

  int k = 0;

  for(k; k<size-1; k++)
    {
      putc(body, file);
    }

  putc(end, file);
}
```

Figure 2 Method used to populate pattern file of required size with specified characters for body and end.

I created inputs for 5 different input sizes calculated by taking the product of the length of the text and the pattern. The different class where: 10^2, 10^4, 10^6, 10^8 and 10^10. For each class I specified four different pattern lengths and worked out the text length by dividing the desired product i.e. 10^10 by the defined pattern length. I have included the defined pattern lengths for the 10^10 class below.

```
// 10^10
pat_length[16]= 1000;
pat_length[17]= 2000;
pat_length[18]= 5000;
pat_length[19]= 10000;
```

Figure 3 Pattern lengths defined for 10^10 input class

I then ran these 20 inputs in the following conditions:

1. Using the Intel Compiler Collection (icc) with an optimisation code of 2 (Maximize speed. Default setting. Enables many optimizations, including vectorization. Creates faster code than -O1 in most cases.)
2. Using the icc with no optimisation.
3. Using the GNU Compiler Collection (gcc) C compiler with an optimisation code of 2 (-O2 turns on all optimization flags specified by –O)
4. Using the gcc compiler with no optimisation.

I will now present the results of each case:

ICC with optimisation

Input Product	CPU Execution Time (s)
100	0
100	0
100	0
100	0
10000	0
10000	0
10000	0
10000	0
1000000	0
1000000	0
1000000	0
1000000	0
100000000	0.26
100000000	0.26
100000000	0.25
100000000	0.25
10000000000	16.99
10000000000	17
10000000000	16.879999
10000000000	16.74

Figure 4 Table to show icc optimised run of inputs product against CPU execution time in seconds

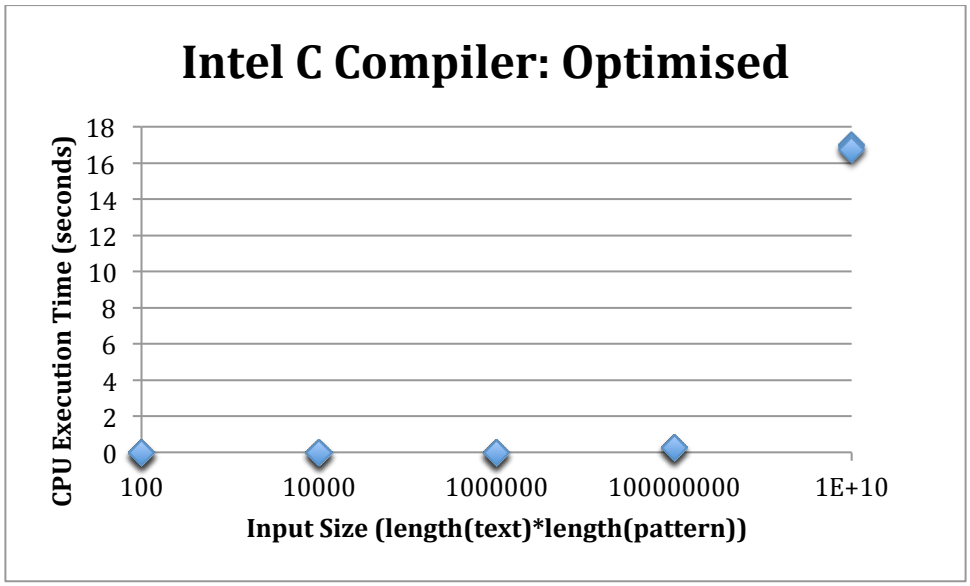

Figure 5: Results from running 20 test inputs on Intel's C Compiler with optimisation

Above we can see that when we compile the searching_sequential.c algorithm using the Intel Compiler Collection with the optimised flag set the majority of our inputs take less than 0.000001 seconds to run. When the product of the inputs reaches 10^8 we notice an increase of the time taken by ~0.25 seconds. We can deduce from this that the processor is able to handle the number of comparisons required to do the worst case search of inputs with a product of less than 10^6 without a noticeable decrease in performance in terms of seconds taken by the CPU. We see that as the product of the input sizes continue to increase to 10^10 the execution time of the CPU reaches as high as 17 seconds. Suggesting that the number of comparisons now required is having a significant impact on the processors performance at this level of optimisation using the Intel compiler.

The varying combinations of pattern length and text length can account for the variation of results within an input product class. A shorter pattern will have to be compared against the text more times to determine if it is there than a long one.

ICC without optimisation

Input Product	CPU Execution Time
100	0
100	0
100	0
100	0
10000	0
10000	0
10000	0
10000	0
1000000	0.01
1000000	0.01
1000000	0.02
1000000	0.01
100000000	0.73
100000000	0.72
100000000	0.71
100000000	0.64
10000000000	64
10000000000	65.019997
10000000000	64.589996
10000000000	65.709999

Figure 6: Table to show icc non-optimised run of inputs product against CPU execution time in seconds

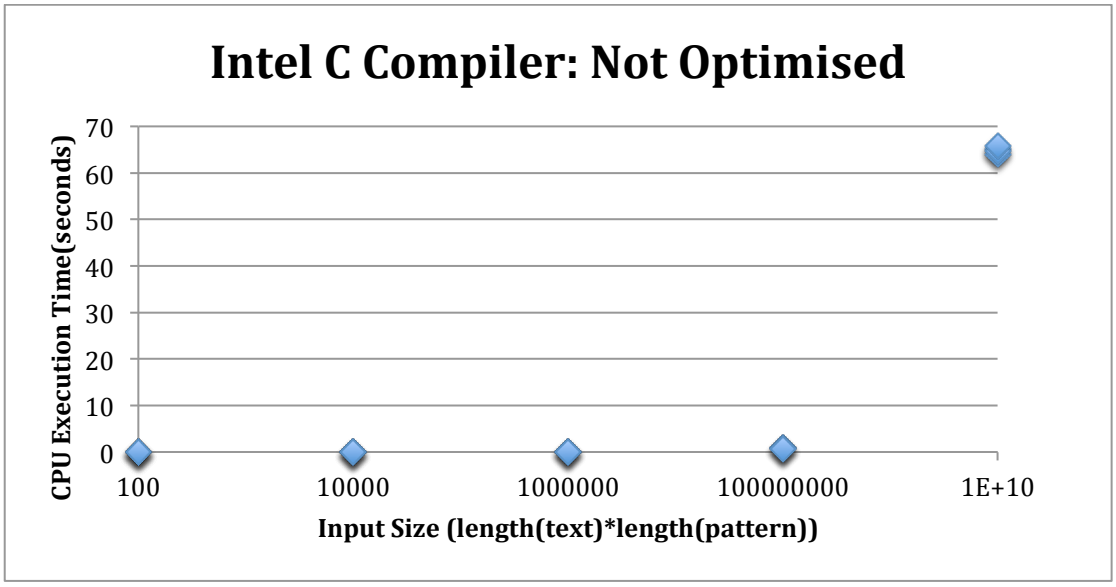

Figure 7 Graph showing the results of running 20 test inputs on Intel's C++ Compiler without optimisation

The first thing we notice about this set of results is that no longer are the majority of CPU execution times less than 0.000001 seconds. Instead we notice the CPU taking up to 0.02 seconds to carry out the search on input products of 10^6 and this increase of time is a trend which is seen across all of the input product classes. The icc without optimisation takes over twice as long to run the search on input products in the 10^8 class as the optimised version did and nearly 4 times the amount of time for input products in the 10^10 class.

From this analysis the non-optimised version of the compiled code seems to take exponentially longer than the optimised version.

GCC with optimisation

Input Product	CPU Execution Time
100	0
100	0
100	0
100	0
10000	0
10000	0
10000	0
10000	0
1000000	0
1000000	0.01
1000000	0
1000000	0
100000000	0.31
100000000	0.3
100000000	0.28
100000000	0.19
10000000000	16.92
10000000000	16.77
10000000000	19.15
10000000000	18.43

Figure 8 Table showing the results of running the tests on a program compiled with the gcc compiler using optimisation

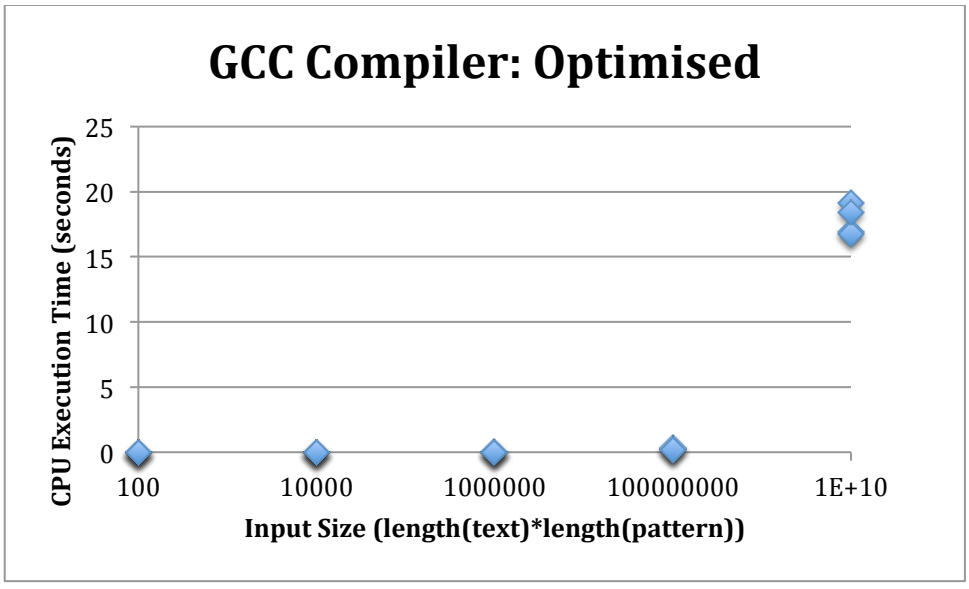

Figure 9 Graph showing the results of running 20 tests on a program compiled with gcc compiler without optimisation

As we can see as with the optimised gcc program the majority of the tests don't use more than 0.000001 seconds of CPU time to compute the search algorithm. We see one of the 10^6 inputs using 0.01 seconds but its not until we reach the

10^8 that we see the CPU execution time consistently reach over 0.1 seconds. These are much faster the non-optimised icc program runs and are more comparable to the icc optimised results. Although the largest gcc time (19.15 seconds) is greater than the largest icc time (17 seconds) this is consistent with all inputs. For some inputs the gcc is marginally faster e.g. for the firist input of product 10^10 the optimised gcc takes 16.92 seconds while the optimised icc equivalent takes 16.99.

It is hard to say at this stage which compiler will give the best results when the optimised flag is enabled as there is not one which has performed better on every test.

GCC Not Optimised

Input Product	CPU Execution Time
100	0
100	0
100	0
100	0
10000	0
10000	0
10000	0
10000	0
1000000	0
1000000	0.01
1000000	0.01
1000000	0.01
100000000	0.93
100000000	0.74
100000000	0.63
100000000	0.62
10000000000	62.83
10000000000	62.759998
10000000000	66.410004
10000000000	63.200001

Figure 10 Table showing the results of running 20 tests on a program compiled with the gcc compiler without optimisation

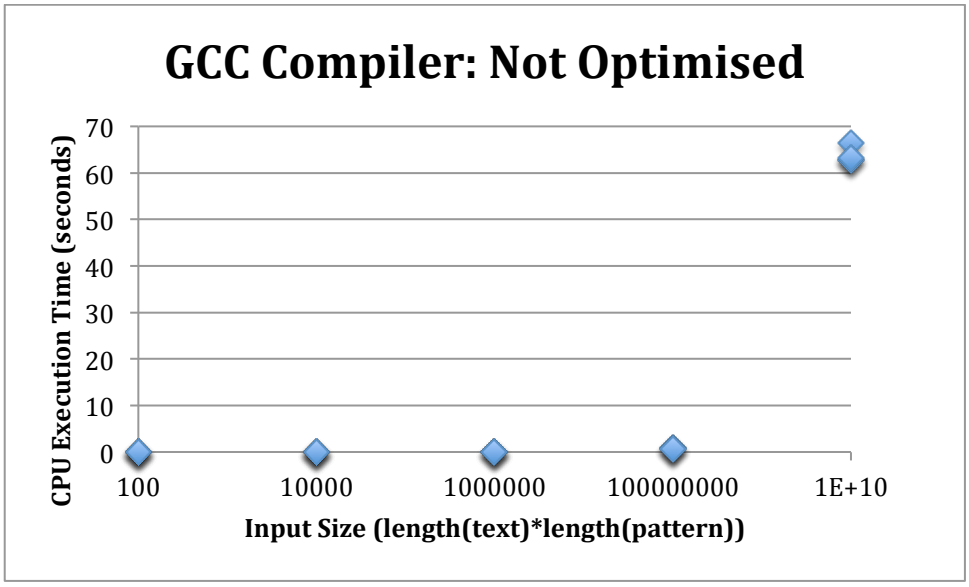

Figure 11 Graph showing the results of running 20 tests on a program compiled with gcc compiler without optimisation

As with the icc compiler the program compiled without optimisation takes a much longer time on all of the inputs than the one compiled with optimisation. The optimised version is up to 3.7 times faster when working in the 10^10 input product class. A more useful comparison would be with the program complied with icc without optimisation. Although the maximum time take by the gcc compiler without optimisation was 66.41 seconds compared to 65.71 on average the gcc compiled program performed the most tests quicker. Running 7 tests faster than icc compiled program while the icc compiled program ran only 3 tests faster. The size of the inputs also doesn't seem to effect which compiled program will run the tests fastest as both ran tests faster in each input product class.

Conclusions

In conclusion I don't feel that I can claim either compiler will produce faster executable C programs than the other as both have shown similar results. One thing is clear however; optimisation can have a significant impact on the amount of CPU time taken to run C programs over a variety of input sizes. With optimisation each compiler can produce programs that are up to 3 times faster than their non-optimised counterparts. In regard to test data, the lower values of input products are relatively insignificant when analysing results. If I were running this again I would only include tests of input products over 10^8 and ran a few more on higher inputs to see if that would have produced a more definite result.