**Performance Analysis of a Serial Pattern Matching Algorithm**

**Jordan McDonald – 40063974**

**1. Introduction**

The context of this study involves the application of a straight forward pattern matching algorithm developed in C to a set of test cases which are composed or patterns (what we are seeking) and texts (the file which may or may not contain the pattern. The program attempts to determine if the entire pattern occurs contiguously as a subsequence of the text and will report back if the pattern is found and the location of the patterns first character. The end goal is to measure the performance of the pattern matching algorithm which executes on a dell cluster – the metrics of interest are #comparisons, wall clock speed & elapsed CPU time. A secondary part of the study is a performance comparison while using different types of compiler (ICC or GCC) and two different permutations for each compiler (optimized or not optimized). The main requirement involves determining a ‘worst case’ for the straight forward algorithm for each of the test case (see the relevant section for further detail on this).

**1.2 Characteristics of a Worst Case Pattern and Text**

For this particular algorithm a worst case occurs when the corresponding text for a given pattern matches the characters of the pattern exactly except the last character in the pattern. If this sequence continues repeatedly in the text block it will lead to a worst case as the algorithm will perform the maximum possible amount of comparisons required in order to find the required pattern. This is confounded by the process of evaluating each text character one at a time in sequence no matter how close the subsequence comparison (only the last value is different) which leads to the algorithm ‘cursor’ moving forward once and repeating the process. See figure one for an example text/pattern pair which will illustrate this.

Text = AAAAAAAAAB;

Pattern = AB;

Number of Total Comparisons = 9

*Figure 1 – Worst Case Example*

**1.3 Experiment setup**

The protocol for the experiment dictates the creation of twenty pattern/test pairs, each of which implements the worst case performance identified previously. The twenty pairs will then be ordered into five ‘size’ samples of 10^2, 10^4, 10^6, 10^8 and 10^10 which is calculated by calculating the length (pattern) \* length (text) which will lead to four pairs for each sample. Restrictions include the total file size of all the test cases should be below one hundred megabytes and the execution for a 10^10 test case sample should be below ten seconds. Each test case will then be input to the pattern matching algorithm in order to determine the values of the required output metrics for analysis.

**1.4 Hypotheses**

**HP1** – The smaller the pattern length the worse the performance (more comparisons)

**HP2** – As sample size increases the performance will decrease (fairly obvious)

**2. Pattern/Text Generation Strategy**

To satisfy the initial 10^2 sample of four test cases the patterns chosen = {1, 2, 4, 5} and text = {100, 50, 25, 20} which exactly matches the required product size for each instance. If we consider the 10^4 sample the pattern = {10, 20, 40, 50} and text = {1000, 500, 250, 50} – this process is repeated across each of the five samples in order to generate text and pattern variations that fit the product size requirements. An additional advantage to using this process is it allows a consistent division of the test case patterns and text for each sample which will allow any comparisons made between samples to be fair and structured (using random distributions of pattern and text for each of the twenty test case would disrupt the validity of the experiment). In addition to this for each sample set of four test cases the pattern size will increase as the corresponding text file decreases, this will allow and analysis on the effects of varying pattern/text file lengths have on the algorithm. At this stage it is expected that the smallest pattern and largest text lengths will yield the worst performance as more comparisons will be required, this experiment will verify that this theory is true.

**3. Results**

**3.1 ICC Compiler – O0 (no optimization)**

|  |  |  |
| --- | --- | --- |
| **Test Number** | **# Comparisons** | **Elapsed Wall Clock Time** |
| **1** | **100** | **0** |
| **2** | **98** | **0** |
| **3** | **88** | **0** |
| **4** | **80** | **0** |
| **5** | **9910** | **0** |
| **6** | **9620** | **0** |
| **7** | **8440** | **0** |
| **8** | **7550** | **0** |
| **9** | **990100** | **0** |
| **10** | **960200** | **0** |
| **11** | **840400** | **0** |
| **12** | **750500** | **0** |
| **13** | **99001000** | **1** |
| **14** | **96002000** | **1** |
| **15** | **84004000** | **0** |
| **16** | **75005000** | **0** |
| **17** | **9900010000** | **67** |
| **18** | **9600020000** | **61** |
| **19** | **8400040000** | **56** |
| **20** | **7500050000** | **51** |

***Figure 2 – ICC O0 comparisons and wall clock time***

***Figure 3 – ICC O0 product v execution time***

**3.2 ICC Compiler - O2 (optimized)**

|  |  |  |
| --- | --- | --- |
| **Test Number** | **# Comparisons** | **Elapsed Wall Clock Time** |
| **1** | **100** | **0** |
| **2** | **50** | **0** |
| **3** | **25** | **0** |
| **4** | **20** | **0** |
| **5** | **1000** | **0** |
| **6** | **500** | **0** |
| **7** | **250** | **0** |
| **8** | **200** | **0** |
| **9** | **10000** | **0** |
| **10** | **5000** | **0** |
| **11** | **2500** | **0** |
| **12** | **2000** | **0** |
| **13** | **100000** | **0** |
| **14** | **50000** | **0** |
| **15** | **25000** | **0** |
| **16** | **20000** | **1** |
| **17** | **1000000** | **17** |
| **18** | **500000** | **16** |
| **19** | **250000** | **14** |
| **20** | **200000** | **12** |

***Figure 4 – ICC O2 wall clock time and comparisons***

***Figure 5– ICC O2 Product v CPU execution time***

**3.3 GCC Compiler - O0 (no optimization)**

|  |  |  |
| --- | --- | --- |
| **Test Number** | **# Comparisons** | **Elapsed Wall Clock Time** |
| **1** | **100** | **0** |
| **2** | **98** | **0** |
| **3** | **88** | **0** |
| **4** | **80** | **0** |
| **5** | **9910** | **0** |
| **6** | **9620** | **0** |
| **7** | **8440** | **0** |
| **8** | **7550** | **0** |
| **9** | **990100** | **0** |
| **10** | **960200** | **0** |
| **11** | **840400** | **0** |
| **12** | **750500** | **1** |
| **13** | **99001000** | **1** |
| **14** | **96002000** | **1** |
| **15** | **84004000** | **0** |
| **16** | **75005000** | **0** |
| **17** | **9900010000** | **66** |
| **18** | **9600020000** | **63** |
| **19** | **8400040000** | **58** |
| **20** | **7500050000** | **50** |

***Figure 6 – GCC O0 Comparisons and wall Clock time***

***Figure 7– GCC O0 product v CPU execution time***

**3.4 GCC Compiler - O2 (optimized)**

|  |  |  |
| --- | --- | --- |
| **Test Number** | **# Comparisons** | **Elapsed Wall Clock Time** |
| **1** | **100** | **0** |
| **2** | **98** | **0** |
| **3** | **88** | **0** |
| **4** | **80** | **0** |
| **5** | **9910** | **0** |
| **6** | **9620** | **0** |
| **7** | **8440** | **0** |
| **8** | **7550** | **0** |
| **9** | **990100** | **0** |
| **10** | **960200** | **0** |
| **11** | **840400** | **0** |
| **12** | **750500** | **0** |
| **13** | **99001000** | **0** |
| **14** | **96002000** | **0** |
| **15** | **84004000** | **1** |
| **16** | **75005000** | **0** |
| **17** | **9900010000** | **28** |
| **18** | **9600020000** | **27** |
| **19** | **8400040000** | **23** |
| **20** | **7500050000** | **20** |

***Figure 8– GCC O2 comparisons and wall clock***

***Figure 9– GCC 02 CPU v Product***

**3.5 Overall Comparisons/Discussion/Explanation**

\*some research was performed (google) to try and assist the discussion – see references where pointed out.

Initially it would be prudent to examine the 10^10 CPU execution times, it is clear that in every permutation of compiler this sample takes significantly longer to execute then the other samples, therefore the growth is not linear as would initially but sudden. Contributing to this includes … . If we continue to observe the scatter graphs we can determine that the CPU execution time for the 10^2 and 10^4 set of test pairs will be 0 potentially due to the power of the dell cluster (see below for more analysis on comparisons for supporting coverage for these samples). The 10^6 and 10^8 cup execution times will however register values in some instances (where the most comparisons are made) which will remain below 1 second.

General observations about the data in the tables will now be performed, initially we will look at the ‘icc 02’ compiler to establish a base line for comparisons. In every case we can determine that as the pattern size increases, the amount of comparisons will decrease – this makes sense if we consider the logic of the algorithm in use which performs comparisons in sequence through the whole text. Therefore we can deduce that the largest amount of comparisons for each sample (10^8 etc.) will be the first of the four, as the pattern size is increased each time and this is evident when the tables are consulted. In addition to this wall clock time also is also associated with the number of comparisons – for when the most comparisons are made the wall clock speed will also be at its highest. How do these facts this relate to this algorithm? It indicates the performance bottle neck of the algorithm is the number of comparisons made – so the worst possible performing pair of text and patterns in the 10^10 sample reflects this and possibly the pattern could decreased further to reinforce this point.

At this point the performance of an optimized and un-optimized compiler will be taken into account for both GCC and ICC. If we consider the ICC tabulated results for an optimal compiler we can see that the amount of comparisons are indeed as optimal as possible, however for an un-optimized the amount of comparisons as the text/pattern files increase in size are exponentially larger. The default settings (O0) disables optimization during compilation whereas (O2) optimizes specifically for speed [5.1] so the performance difference is to be expected – see figure 10 for some possible optimizations. The same performance variation is evident when the GCC compiler is analyzed, in this case (O0) will Reduce compilation time and make debugging produce the expected results. O2 in comparison will perform nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code [5.3].

* removal of unreferenced variables
* constant propagation
* copy propagation
* dead-code elimination

Figure 10 – list of ICC O2 optimizations

* global register allocation
* tail recursions
* peephole optimizations
* structure assignment lowering
* and optimizations
* dead store elimination

It would also be prudent to loosely discuss ‘how’ the programmer chose to code the application and the effects of compilation on performance. The program utilizes the ‘realloc’ function of C to resize the memory block dynamically as the text or pattern file exceeds the initial memory size, if the compiler performs no optimization this command in particular will result in increased execution time as memory allocation will suffer. It should be stated that this is good practice when handling data of an unknown size and is used as well to handle the exception of no memory being available. As a side note the drawback of c loading in each file one character at a time will have a performance impact.

To complete the discussion the next point of discussion is the comparison of the ICC and GCC compilers and the effect each has on performance. In terms of the number of comparisons it is interesting that the un-optimized ICC and both version of GCC provide the same amount of comparisons for each test case – this would indicate that despite optimization the compiler cannot organize memory to reduce the number of comparisons. However the wall clock speed of the optimized GCC is far superior to the (O0) ICC compiler, despite when compared the CPU execution time of the optimized ICC compiler is the best performer in this particular algorithm and context. This will boil down to how both compilers translate the C code to assembly and the binary and the ways in which they vary will contribute to this outcome.

**4. Conclusions**

This report has made attempt to compare the effects of compiler permutations on a simple pattern matching algorithm using a structure distribution of pattern and text test cases. In terms of overall performance the two un-optimized permutations had quite similar performance when CPU execution time is considered, remaining on this measure however it was found the intel compiler had the best performance & and the wall clock speed reflects this as well. In addition to this patterns were identified such as the effect of the number of comparisons on the other measures, with high comparisons implying a high execution time (in this context). To further this point the amount of comparisons between three of the permutations for the test cases were exactly the same, the reasons behind this lay within the intrinsic compiler processes which is a good avenue for further research.

**5. References**

**5.1** [**https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler**](https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler)**, viewed on 18/02/2015**

**5.2** [**https://wiki.scinet.utoronto.ca/wiki/images/7/77/Snug\_techtalk\_compiler.pdf**](https://wiki.scinet.utoronto.ca/wiki/images/7/77/Snug_techtalk_compiler.pdf)**, viewed on 18/02/2015**

**5.3** [**https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html**](https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)**, viewed on 18/02/2015**