**Spike:** Task 7.P

**Title:** Performance Measurement

**Author:** Mitchell Wright, 100595153

**Goals / deliverables:**

Demonstrate the ability to collect and analyse software performance data, applying said knowledge to the comparison and evaluation of functions, IDE settings and compiler settings.

Items created during task:

* Code, see: \07 - Spike – Performance Measurement\t7-performance\

**Technologies, Tools, and Resources used:**

List of information needed by someone trying to reproduce this work

* Visual Studio 2022
* SourceTree
* GitHub
* Lecture 3.1 – Code Performance

**Tasks undertaken:**

* Single Tests
* Ramp-up Test
* Repeatability
* Function Comparison
* IDE Settings
* Compiler Settings
* Commit to Git

**What we found out:**

1. Single Tests:

For the tests within this section, a simple bubble sort was used on a vector containing 10 random elements. The bubble sort algorithm is shown below. Please note bubble sort was chosen due to it being simple to code, as well as leaving the vector passed through to it unsorted at the end (for repeatability’s sake). Please also note this was a Chat GPT special. Since a sorting algorithm is a sorting algorithm and nothing fancy was needed here, I opted for the quick and dirty option so I could focus on the important parts of this task.

A screen shot of a computer code

Description automatically generated

To fill the vector with random numbers, the below code is used.

A computer screen shot of text

Description automatically generated

The random number generation has been lifted from task 2 and performs the same basic function, though now in conjunction with a vector rather than an array. Numbers generated are between 0 and 100.

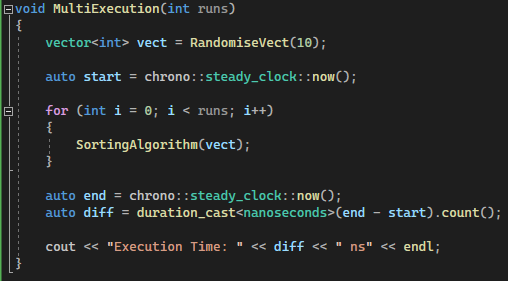
For the single execution test, a start time is taken, the sorting algorithm then runs, then the end time is taken.

A computer screen shot of code

Description automatically generated

Following the sample code provided in the lectures, steady\_clock::now() is used to get the current time, then the execution time is calculated by casting the difference between start and end into nanosecond format. After which, the results are printed to console.

Multi-execution testing is performed essentially the same way, though with one extra for loop.



The sorting algorithm is executed multiple times now, rather than just once. Note that the vector does not need to be reset in this case, as the changes to the values only occur on a copy of the vector, not the original vector.

The results for both tests are shown below.

A black screen with white text

Description automatically generated

1. Ramp-up Test:

The ramping tests were done using the Single Execution from before, placed inside a for loop, with the loop start and end points, as well as step size, being determined by parameters. In the case of the exponential test, this was done with a scale parameter and a power parameter (with scale^power being used to find the end point). It should also be noted that the size of the vector being sorted is what is being ramped here.

A computer screen with white text

Description automatically generated

Based solely on numbers, the linear ramp-up test seems to show steady numbers. Please note that the results for both this and the ramp-down test start/end at 10.

|  |  |
| --- | --- |
| Number of Elements | Execution Time (Nanoseconds) |
| 1 | 1600 |
| 10 | 500 |
| 20 | 3100 |
| 30 | 2300 |
| 40 | 3600 |
| 50 | 4800 |
| 60 | 6900 |
| 70 | 8300 |
| 80 | 9200 |
| 90 | 12100 |
| 100 | 21000 |

Towards the end of the chart, it becomes obvious that there is a bit of exponential growth in execution time occurring, with the numbers starting to double.

The chart for while looks like this:

A graph showing a number of objects

Description automatically generated

The chart highlights the exponential growth of execution time much more effectively. It should also be noted that during the 20-element test, there seems to be some sort of spike. This was consistent across runs.

The ramp-down test is essentially the ramp-up linear but in reverse, as shown below.

A screen shot of a computer program

Description automatically generated

For which the results look like the below.

A screen shot of a black screen

Description automatically generated

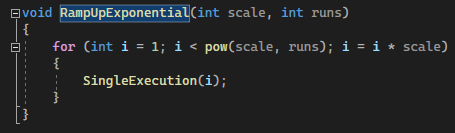
These results start to highlight a pattern with my laptop suddenly spiking in execution time upon the second run of the test, which is very interesting.

A graph showing a number of elements

Description automatically generated

These results still do show a (skewed) exponential growth, though now with a representative value for a single element vector. Generally, it seems that ramp-up tests stabilise in terms of results when reaching the higher bounds of the testing, whereas ramp-down testing achieves the opposite. These types of testing used in conjunction can give a more accurate picture of how an algorithm reacts over a range of inputs.

In order to test an exponential growth in element size as an input, the ramp test was modified as below.



This produced the below results.

A screen shot of a computer

Description automatically generated

And charted, looks like the below.

A graph with a line going up

Description automatically generated

While more test points will highlight this better, there does seem to be the beginnings of an exponential curve forming. Please note that graph scales up 1 to 2^15. Unfortunately measuring using powers of 10 was not realistically feasible on a laptop.

1. Repeatability:

Repeatability can be quite difficult when it comes to getting exact results. Many variables can cause differences, such as Windows updates, background tasks, other programs, OS overhead, etc. Re-using the linear ramp test from before, the results on the chart below show a reasonable amount of variance, even from such a small test. Please note that the programs were run consecutively, with no new programs opened or closed between runs.

A graph with different colored lines

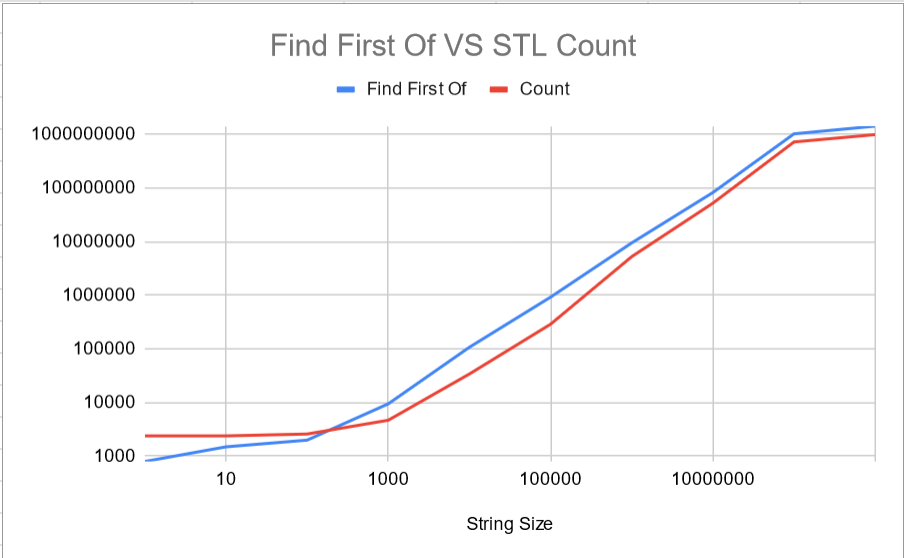
Description automatically generated

1. Function Comparison:

Comparing the 2 functions provided in the sample code, the one comment as slower seems to be performing a lot faster than the STL version. The below tests were done using a portion of the Bee Movie script.

|  |  |
| --- | --- |
| **Find First Of** | **Count** |
| 7700 | 13400 |
| 3800 | 5000 |
| 2900 | 4600 |

Testing this again with a ramp-up based on string length using a random string generator, these are the results.



Using an exponential ramp up test, it’s notable that the Find First Of function starts off much faster, but soon loses out to the Count when getting up to the larger strings. Please note that both axes have been log scaled for visibility.

1. IDE Settings:

When testing Debug vs Release modes with the multi-execution test from section 1, the results are as below.

|  |  |
| --- | --- |
| **Debug** | **Release** |
| 16100 | 1700 |
| 17000 | 1300 |
| 15100 | 1500 |

Clearly Release mode, running without all the debugging tools runs a heck of a bit faster, aye.

1. Compiler Settings:

In order to enable compiler optimization, follow the menus to the setting:

Project -> Properties -> Configuration Properties -> C/C++ -> Optimization.

These settings cannot be enabled during Debug mode without causing an error. However in Release mode they’re enabled by default, so the results will be similar to above.

1. Commit History:

