N

Parallel computing

Performance analysis - Adam Stock

University of the West of England

UFCFFL-15-M - Parallel computing

# Introduction

In the interests of keeping all tests and analysis of tests as fair as possible all potential variables will be disclosed and discussed below.

## Environment and background

For this assignment work was completed on a laptop containing a mobile Intel I 7700hq, 8GB of ddr4 am using a USB 3.0 (type c) Samsung SSD. The work was done inside an Ubuntu 18.04.1 LTS virtual machine.   
This is bought to attention as the nature of this system gives serious limitations to the testing of this software.

### Hardware

The I7 7700hq CPU used inside this laptop is a mobile variant and therefore contains features designed to keep power consumption to a minimum, these include scaling of core frequency on a core by core basis, depending on workload and background tasks .This could easily effect the outcome of tests being performed. Furthermore the operating environment is located on an external drive, due to the polling nature of external USB drives this can again introduce unwanted variance in testing (though in practice this should be minimal due to the program running from Ram).

### Software

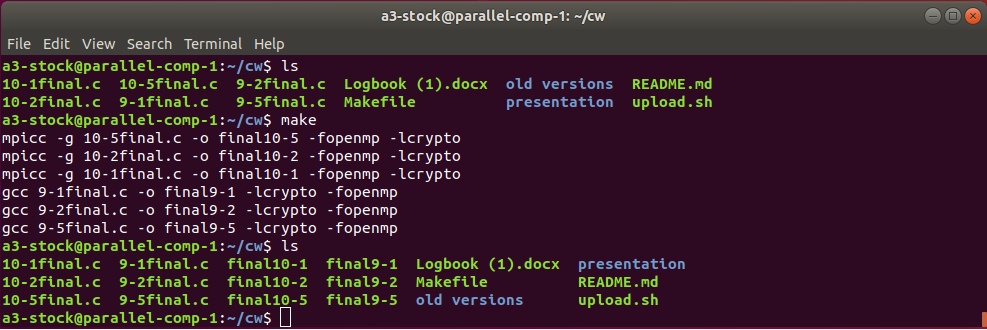
As mentioned the operating environment was Ubuntu 18.04.1 LTS running in a virtualized environment in this case the host machine is running Windows 10. This is noteworthy as even using Intel’s VT technology there will be some disconnect between threads inside the virtual machine and their mapping to hardware threads, this is due to potential blocking from the host operating system.

### The Cluster

For these reasons the system the software was developed on is not ideally suited to performance testing, luckily we have a solution for this – the cluster.

Using the commands ‘lscpu’ and ‘lshw’ from the terminal (at this point we have used ssh to connect to the cluster computer) we can view hardware information for the system. The system is outfitted with an Intel Xeon E3-1220 CPU and 64GB of ram. This CPU is in some ways slower than that of the development system as it has only 4 available cores without the Hyper-Threading available on the 7700hq, but in practice is likely to give better results as its power saving features are less aggressive and its thermal solution will allow running at high frequencies for long periods of time without thermal throttling. The RAM used will also help results as though it is likely to work at a lower frequency (this information is not easily available) it is likely running in dual channel more providing basically twice the available bandwidth, this in turn should prevent the program having to wait for RAM access.

The software used on the cluster is a native version of Ubuntu Linux and as such should provide an even testing environment for all tests. All versions of the program were copied to the cluster using a script containing an ‘rsync’ command and then compiled using the included makefile giving the following result –



# Testing methodology

Each of the programs will be tested on the cluster multiple times and an average will be taken to give a result. Each of the programs use a different method to compute the workload these are as follows –

9-1final.c - serial (single thread OpenMP set to single thread)

9-2final.c - two threads using OpenMP to parallelize

9-5final.c - five threads using OpenMP to parallelize

10-1final.c - single thread workload but run 5 times in parallel

10-2final.c - two threads using MPI set to 2 threads

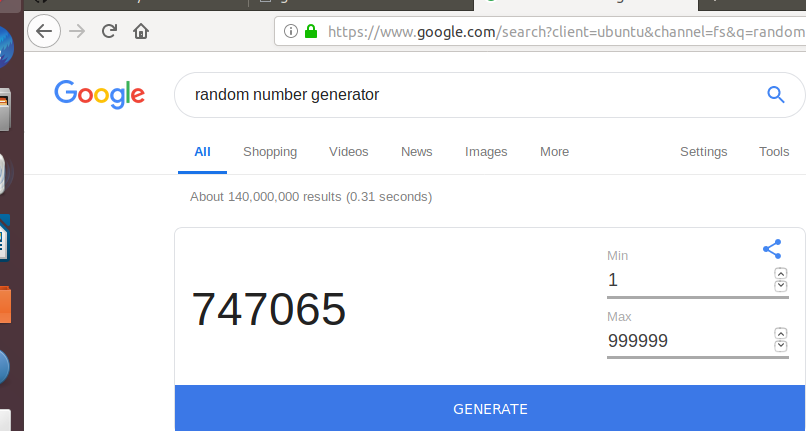
10-5final.c - five threads using MPI set to 5 threads

Each of these tests will be run 10 times the results added together and divided by 10 to give a final result from which conclusions will be drawn.

## Random number

All programs contain the same hard coded key to find, this key has been limited as mentioned in the logbook. The key is 16 characters long but to save on compute time the first and last five characters are set to the ‘#’ character and the center 6 characters to number 0-9 , this gives us 10^6 or one million possibilities. To provide a fair test a random key was generated using Googles random number generator with a scope of 0 – 999999.

The number as chosen by Google was 747065 as seen below –



# Results

With all tests completed the data has been put into the following table –   
Notes all results are in seconds.

## Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Run** | **final9-1** | **final9-2** | **final9-5** | **final10-1** | **final10-2** | **final10-5** |
| **1** | 0.161577 | 0.076907 | 0.053055 | 0.168064 | 0.08547 | 0.074754 |
| **2** | 0.179854 | 0.08005 | 0.05121 | 0.166443 | 0.08931 | 0.071252 |
| **3** | 0.164947 | 0.079515 | 0.048836 | 0.169693 | 0.091191 | 0.061263 |
| **4** | 0.159618 | 0.07502 | 0.054747 | 0.162189 | 0.091917 | 0.063275 |
| **5** | 0.159514 | 0.079501 | 0.08425 | 0.16643 | 0.088457 | 0.059442 |
| **6** | 0.164828 | 0.079147 | 0.057307 | 0.16601 | 0.096991 | 0.061397 |
| **7** | 0.169674 | 0.0792 | 0.065354 | 0.164099 | 0.08717 | 0.049007 |
| **8** | 0.16229 | 0.075648 | 0.059411 | 0.162889 | 0.0882 | 0.058999 |
| **9** | 0.163526 | 0.082157 | 0.052497 | 0.16341 | 0.099483 | 0.068069 |
| **10** | 0.161017 | 0.077258 | 0.078637 | 0.164831 | 0.08391 | 0.062097 |
| Total | 1.646845 | 0.784403 | 0.605304 | 1.654058 | 0.902099 | 0.629555 |
| **Average** | **0.164685** | **0.07844** | **0.06053** | **0.165406** | **0.09021** | **0.062956** |
| Max | 0.179854 | 0.082157 | 0.08425 | 0.169693 | 0.099483 | 0.074754 |
| Min | 0.159514 | 0.07502 | 0.048836 | 0.162189 | 0.08391 | 0.049007 |
| **Variance** | **0.02034** | **0.007137** | **0.035414** | **0.007504** | **0.015573** | **0.025747** |

## Speedup

From the above results table we can work out the speed up of the parallelization, this is found using –

So in our case for the OpenMP code –

For two threads

For five threads.

And for the MPI code –

For two threads

For five threads.

## Algorithm cost

We can figure out the Algorithm cost using –

Cost = Parallel running time x #processors

In our case this gives us -

0.15688 = 0.07844 x 2 for OpenMP with two threads

0.30265 = 0.06053 x 5 for OpenMP with five threads

0.18042 = 0.09021 x 2 for MPI with two threads

0.31478 = 0.062956 x5 for MPI with five threads

## Efficiency

With the available numbers we can also find out efficiency using one of the following –

Efficiency = sequential running time / (processors x parallel running time)

Efficiency = speedup / processors

Efficiency = sequential running time / cost

However doing this provides inconsistent results so only one example is included for OpenMP with five threads –

0.165406 / (5 x 0.062596) = 0.31298

2.720716999834793 / 5 = 0.54414

0.165406 / 0.32065 = 0.54652

# Analysis / Conclusion

From the above results we can conclude that adding processors to a given problem provides diminished returns. This is due to two main reasons, firstly overheads from communication with many processors eventually use more CPU time than the function being solved meaning that eventually more processors could actually increase computation time (this was discussed further in the logbook) and secondly Amdahl’s law dictates that the theoretical speedup from adding processors can never exceed the proportion of the program that can be parallelized. This also highlights why performance of modern computing systems is often a function of both the speed/frequency and the number of available cores/threads/CPU’s available and as an example why the leaderboard of benchmarking systems such as 3dMark by Futuremark are not dominated strictly by systems with multiple CPU’s or CPU’s with very high core counts such as the AMD Ryzen Threadripper series. The same can be said for the Top500 supercomputer list where peak TFlops/s is measured of the fastest supercomputers in the world.