

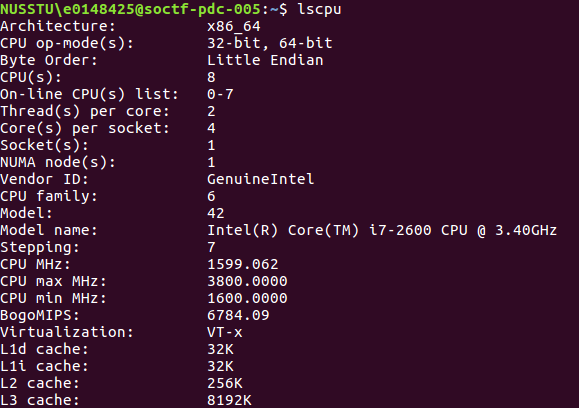
CS3221 Project 1

NAME: GOH KA HIAN

Student ID: A0164383B

|  |
| --- |
| 1. Hardware and lab results |

This part of the report will discuss the hardware used in the laboratory. The hardware for the laboratory experiment consist of the lab machine SOCTF-PDC-005 and the Tembusu Cluster.



*Figure 1.1 – Hardware specification of lab machine, obtained from lscpu command*

The laboratory machine has a quad core Intel i7-2600 processor(4 physical core) capable of simultaneous multithreading running on the Ubuntu operating system. While the Tembusu Cluster consist of 5 access nodes and 17 compute nodes designated for non-interactive compute jobs. Each of the compute node has a dual Quad-Core Xeons.

If the executing program has more thread beyond the number of core available to it, there will not be any visible speedup. In that case, it is likely that there will be more context switches occurring in a serial manner as well as having more overhead possibly from thread scheduling and cache contention.

Parallel matrix multiplication using OpenMP are used for the experiment for both of the machine. For the laboratory experiment, execution of the program using different matrix size with the numbers of thread scaling up for both the Tembusu Cluster and Lab machine are recorded 3 times each. The minimum time taken should be considered instead, it will be better to analyze the best possible time that is least affected by other events on the running machine. While the average case is not feasible due to the long running time of the program and it requires large sample size for better accuracy. It can also be observed that timing can vary dramatically which could be due the machine being affected by other events.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MatrixSize  Thread | **128** | **256** | **512** | **1024** | **2048** |
| 1 | 0.0295  0.0296  0.0290 | 0.1657  0.1690  0.1731 | 1.0060  1.0818  1.0054 | 8.9966  9.1823  9.5223 | 105.0188  109.4460  105.7258 |
| 2 | 0.0154  0.0154  0.0153 | 0.0981  0.0807  0.0995 | 0.5253  0.5396  0.4808 | 5.9763  4.6665  4.9701 | 53.5300  54.0157  62.3277 |
| 4 | 0.0079  0.0082  0.0071 | 0.0514  0.0457  0.0503 | 0.3245  0.2527  0.3008 | 2.4491  2.4205  2.8403 | 27.4418  27.9058  27.8533 |
| 6 | 0.0102  0.0077  0.0097 | 0.0658  0.0541  0.0638 | 0.3102  0.2785  0.3512 | 2.7477  2.8335  2.6712 | 24.7601  25.4885  26.0227 |
| 8 | 0.0104  0.0154  0.0074 | 0.0380  0.0547  0.0564 | 0.2475  0.2511  0.2818 | 2.1780  2.1581  2.2481 | 21.8444  22.3136  22.3563 |

*Figure 1.2 - Result of Matrix Multiplication using lab machine (in seconds), lowest timing is underlined*

Based on figure 1.2, a good improvement in speedup can be observed from 1 to 4 thread for a particular problem size as expected. It is observed that beyond 4 threads, beyond the number of core available to the lab machine, there is still slight improvement in speedup.

Simultaneous multithreading is a technique that permits multiple independent threads of execution to better utilize resources. The lab machine has an Intel i7-2600 capable of performing simultaneous multithreading within each processor core, up to two threads per core (total of 8 threads in parallel) which explains the speed up observed beyond 4 threads even though it has only 4 physical cores.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MatrixSize  Thread | **128** | **256** | **512** | **1024** | **2048** |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1.83 | 2.05 | 2.09 | 1.93 | 1.96 |
| 4 | 4.08 | 3.63 | 3.98 | 3.72 | 3.83 |
| 6 | 3.64 | 3.06 | 3.61 | 3.37 | 4.24 |
| 8 | 3.78 | 4.36 | 4.06 | 4.17 | 4.81 |

*Figure 1.3– Speed up of lab machine using the lowest timing, Speedup = time(1 thread)/time(n threads)*

It is noticed from figure 1.3, the speedup is smaller beyond 4 threads. Even though there are 8 running threads in parallel, the lab machine only has 4 physical processing core, meaning that it has a maximum of four jobs. However, if one of the jobs stall or idle due to reasons such as memory access, another thread can start executing on the free processing core with little penalty. Thus, for highly cache efficient and CPU-bound codes, there will hardly be any speedup and explains why the speed up from 4 to 8 cores is less noticeable.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MatrixSize  Thread | **128** | **256** | **512** | **1024** | **2048** |
| 1 | 0.0325  0.0319  0.0325 | 0.1863  0.1812  0.1813 | 1.2426  1.0003  1.0270 | 9.0087  8.8380  8.8452 | 111.0298  97.9535  109.3421 |
| 2 | 0.1070  0.0612  0.0171 | 0.3749  0.1071  0.1108 | 2.6264  2.7664  0.5384 | 25.9093  4.4837  4.4699 | 58.1249  204.4176  159.0158 |
| 4 | 0.0741  0.0089  0.0090 | 0.0615  0.3924  0.0598 | 1.3158  0.3766  0.3836 | 2.2773  2.2590  2.4205 | 27.8055  116.7861  28.4859 |
| 6 | 0.0518  0.0066  0.0062 | 0.0427  0.0442  0.0439 | 0.3066  0.2649  0.2616 | 1.8410  1.7834  5.6492 | 19.6650  19.9580  20.2109 |
| 8 | 0.0380  0.0045  0.0282 | 0.0322  0.1520  0.1459 | 0.2050  0.2061  0.2038 | 1.4001  1.7331  9.9427 | 15.3875  65.8483  15.7047 |
| 16 | 0.0046  0.0375  0.0046 | 0.1285  0.1317  0.0328 | 0.1892  0.1909  0.1914 | 1.1417  7.2340  1.0788 | 10.7480  10.7007  44.1143 |
| 24 | 0.0035  0.0034  0.0198 | 0.0226  0.0252  0.0238 | 0.1441  0.1446  0.1465 | 5.0674  4.1594  0.9610 | 8.8483  8.6387  35.9054 |
| 32 | 0.0045  0.0045  0.0045 | 0.0313  0.0237  0.0329 | 0.1651  0.1617  0.1685 | 1.0681  1.0523  3.9088 | 9.4105  9.8029  28.6050 |
| 40 | 0.0039  0.0038  0.0041 | 0.0957  0.0250  0.0845 | 0.1442  0.1414  0.1432 | 3.7651  0.9869  0.9909 | 9.2966  9.3598  9.2718 |

*Figure 1.4 - Result of Matrix Multiplication using Tembusu Cluster (in seconds), lowest timing is underlined*

Number of threads up to 40 are recorded, as unlike the lab machine, it has much more processing core available to it.

It is noticed that for the Tembusu Cluster, unlike the result for the lab machine where the speedup keeps increasing with more threads, as number of thread increases, the amount of speed up achieved decreases until a limit for has been reached. Notice that speedup from 32 to 40 threads are almost non-existential. This suggest that it is not always the case when more parallelism equates to more speedup, more thread comes with more overhead such as context switches.

|  |
| --- |
| 1. Speed up |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MatrixSize  Thread | **128** | **256** | **512** | **1024** | **2048** |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1.87 | 1.70 | 1.86 | 1.98 | 1.69 |
| 4 | 3.58 | 3.03 | 2.66 | 3.91 | 3.52 |
| 6 | 5.15 | 4.24 | 3.82 | 4.96 | 4.98 |
| 8 | 7.09 | 5.63 | 4.91 | 6.31 | 6.37 |
| 16 | 6.93 | 5.52 | 5.29 | 8.19 | 9.15 |
| 24 | 9.38 | 8.02 | 6.94 | 9.20 | 11.33 |
| 32 | 7.08 | 7.65 | 6.19 | 8.40 | 10.41 |
| 40 | 8.40 | 7.25 | 7.07 | 8.96 | 10.56 |

*Figure 2.1 – Speed up for Tembusu Cluster using the lowest timing, Speedup = time(1 thread)/time(n threads)*

Figure 2.1 has shown that with more thread, the speed up will increase. However, another observation can be made, when the matrix size increases, the speed up using the same number of thread also increases.

*Figure 2.2 – Shows the speed up for figure 2.1 represented in graph for each matrix size.*

For Amdahl’s law in which the size of the problem is held constant, it is shown that from figure 2.2, the speedup is not linear when the core scales up and it is similar to a logarithmic function. Thus, this shows that when solving the same sized problem with greater computing power, the speed up achieved will gradually get lessen until an upper bound limit. This suggest an upper bound limit as to how much speedup we are able to achieve by holding the problem size constant while increasing the number of threads.

*Figure 2.3 – Shows the speed up for figure 2.1 represented in graph with the scaling of problem size.*

Figure 2.3 shows the result of holding time constant instead of the problem size. As shown from the graph, it is noted that as the problem scales up along with the number of thread count, the speed up also scales up linearly. It can be viewed as having the amount of problem solved scales up along with the problem size. This implies that with more thread count, larger amount of work can be completed in the same time, exploiting available computing power unlike Amdahl’s Law.

|  |
| --- |
| 1. Memory Effects |

Most modern computer has a two or three levels cache, this experiment shows an effect that is based on these caches.

1. **for** (i = 0; i < steps; i++)
2. {
3. arr[(i \* 16) & lengthMod]++; // (x & lengthMod) is equal to (x % arr.Length)
4. }

*Figure 2.1 – testmem.c showing array access*

From figure 2.1, it is observed that the program traverses the array by incrementing integer i by 16, which is a way to modify cache line. Upon reaching the last value, the program loops back to the start.

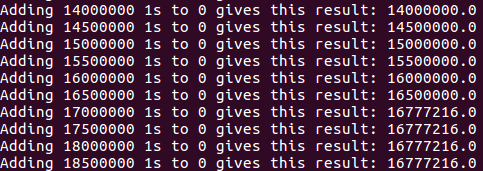
*Figure 3.1 – Graph on average time taken (in seconds) for executions of testmem.c for different array size.*

From figure 3.1, n = 5, 8, 12, 13 are the points at before the time taken increases noticeably. These points also correspond to the L1, L2 and L3 cache size of the lab machine (as shown on Figure 1.1).

The most distinct increase in time taken is when the array size is of 8192kB, which is also the size of the L3 cache. With 8192kB array size, it is still possible to have cache hits. However, when the size double, cache misses will become very frequent, as the L3 cache will only be able to hold half the array size at best, resulting in significantly longer time due to overhead from cache handling.

As the source code from Figure 2.1 suggested, the program has poor temporal and spatial locality resulting in frequent cache miss if the array size gets bigger than the machine cache size.

|  |
| --- |
| 4. Accuracy |

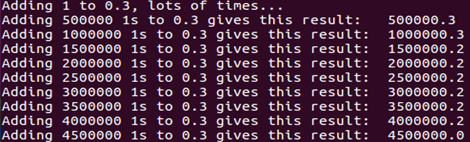


*Figure 4.1 – Result from executing fpadd1.c*

fpadd1.c is a program that adds 1 repeatedly to a floating point number.

[(1 sign bit)] [8 exponent bit] [23 mantissa bit] is the format of IEEE 754 32-bit floating point number. The mantissa bits represent binary digits after the decimal separator.

16777216.0 is represented as 0 10010111 00000000000000000000000 in this format. Thus when incrementing to 16777217.0 = (+1) \* \* (1.0 +) which is not representable in this format (as it requires 24 mantissa instead of 23), it is rounded to 16777216.0 instead, a representable value. (Rounding error)



*Figure 4.2 – Result from executing fpadd2.c*

fpadd2.c is a similar program to fpadd1.c except that the floating point number starts at 0.3

The same rounding errors as fpadd1.c occurs again for value 1500000.3 and 4500000.2, which could not be expressed exactly as a floating-point number and are rounded to nearest presentable value.

|  |  |  |
| --- | --- | --- |
| Execution no. | **Sum from 1 to 20** | **Sum from 20 to 1** |
| 1 | 631.012695 | 631.012817 |
| 2 | 678.601257 | 678.601318 |
| 3 | 457.345184 | 457.345154 |
| 4 | 597.020813 | 597.020874 |

*Figure 4.3 – Result from executing fporder.c 4 times.*

fporder.c is a program that generate random floating point number sum them up in a sequence, and in a backward sequence.

Mathematically, A + B + C = C + B + A (Associative property), but for floating point number, this is not the case. From figure 4.3, we can overserve that the change in addition operation sequence resulted in different set of results. Because of floating point’s limited precision, as shown previously from fpadd1.c and fpadd2.c, the order in which operation are executed affects the accuracy of the result.

|  |  |  |  |
| --- | --- | --- | --- |
| **First Run** | **Second Run** | **Third Run** | **Fourth Run** |
| 3056318.000000 | 3062428.750000 | 3049300.000000 | 3062640.000000 |
| 3056318.000000 | 3062428.500000 | 3049299.750000 | 3062640.000000 |
| 3056318.250000 | 3062428.750000 | 3049299.750000 | 3062640.000000 |
| 3056318.000000 | 3062428.500000 | 3049299.750000 | 3062640.000000 |
| 3056318.000000 | 3062428.500000 | 3049299.750000 | 3062640.250000 |
| **80% identical** | **60% identical** | **80% identical** | **80% identical** |

*Figure 4.4 – Result from executing fpomp.c on lab machine with 8 threads*

|  |  |  |  |
| --- | --- | --- | --- |
| **First Run** | **Second Run** | **Third Run** | **Fourth Run** |
| 3050245.250000 | 3056767.250000 | 3056767.500000 | 3055063.250000 |
| 3050245.250000 | 3056767.500000 | 3056767.500000 | 3055063.500000 |
| 3050245.250000 | 3056767.000000 | 3056767.250000 | 3055064.000000 |
| 3050245.000000 | 3056767.250000 | 3056767.250000 | 3055064.250000 |
| 3050245.250000 | 3056767.500000 | 3056767.500000 | 3055063.750000 |
| **80% identical** | **40% identical** | **60% identical** | **0% identical** |

*Figure 4.5 – Result from executing fpomp.c on Tembusu Cluster with 24 threads*

fpomp.c is a program that sums up an array of floating randomly generated floating point numbers by partitioning the summing to separate different threads. Each thread will perform the summing for their partition before totaling up the results with all other threads. This is repeated another 4 times per execution.

From figure 4.4 and 4.5, it is observed that there are accuracy issue as the sum does not tally to be the sum most of the time. This is due to threads not being assigned to the same partition for performing the summing and the order of summing the combined result could be different. (As shown from figure 4.3, the order of operation for floating point number can affect the accuracy of result)

By comparing the execution of the program with 8 threads (figure 4.4) and 24 threads (figure 4.5), it can be observed that as there are more threads, the accuracy issue worsen. Even though, the computational speed is most likely faster due to more threads, this experiment has suggested that when it comes to computing floating-point numbers, accuracy can become an issue with greater parallelism. One way to resolve this issue is to use a higher precision floating-point format.

|  |
| --- |
| 1. Communication, Speedup |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **No. of process** | **Communication time for slave process** | | | **Computation time for slave process** | | | **Average total time per slave process** |
|
| **Minimum** | **Maximum** | **Average** | **Minimum** | **Maximum** | **Average** |
| **8** | 1.39 | 1.39 | 1.39 | 15.65 | 16.91 | 16.41286 | 17.80286 |
| **16** | 2.52 | 2.54 | 2.525333 | 7.57 | 8.05 | 7.761333 | 10.28667 |
| **24** | 3.66 | 3.67 | 3.663478 | 4.93 | 6.17 | 5.181304 | 8.844783 |
| **32** | 4.66 | 5.51 | 4.68871 | 3.72 | 4.65 | 3.899355 | **8.588065** |
| **40** | 5.79 | 6.45 | 5.818718 | 2.91 | 3.64 | 3.250513 | 9.069231 |
| **48** | 6.78 | 7.5 | 6.824468 | 2.4 | 3.11 | 2.809787 | 9.634255 |
| **56** | 7.93 | 8.48 | 7.957455 | 2.09 | 2.65 | 2.496182 | 10.45364 |
| **64** | 8.94 | 9.41 | 8.971429 | 1.72 | 2.32 | 2.172698 | 11.14413 |

*Figure 5.1 – Communication and computation time taken by slave processes when executing mm-mpi.c with scaling number of processes using multiple machines.*

mm-mpi.c is a MPI matrix multiplication program that attempt to measure communication and computation time for a matrix-multiply operation.

From figure 5.1, it is very clear that as number of processes increases, the communication time for slave process increases while the computation time decreases. With more processes, each slave process will be given a smaller partition of work, resulting in shorter computation time. However, with more processes, naturally there will be more message passing which would result in longer communication time.

From figure 5.1, it is also observed that having 32 threads incur the least average total time per slave process for this experiment, while having 40 and more threads increases the total time as communication time required lengthens despite the shorter computation time. This experiment has shown the communication overhead of having more processes when trying to lessen computation time through having more processes, a problem when attempting to improve speedup through parallelism.