CMPT 431

Assignment1

Report

Collaborators: Brian Pak Rafay Tanzeel

Student ID: 301168028 301243667

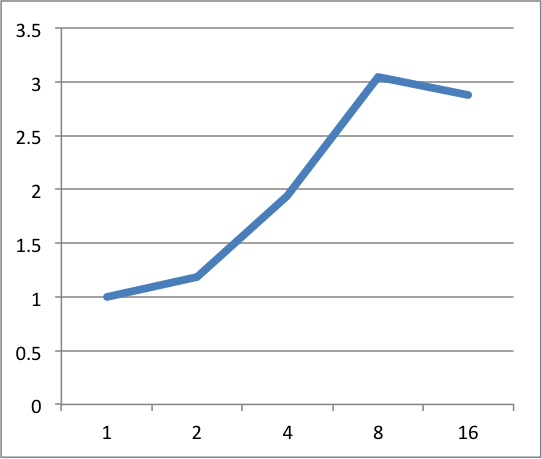
SFU ID: [bpak@sfu.ca](mailto:bpak@sfu.ca) [rtanzeel@sfu.ca](mailto:rtanzeel@sfu.ca)

Part 1. Pre-assignment:

Data used for the parallel version of SOR program

Number of Processors and Elapsed time - x-axis: Number of processors

in each trial (measured in sec) y-axis: Speedup

Input: i=100 m=4000 n=500

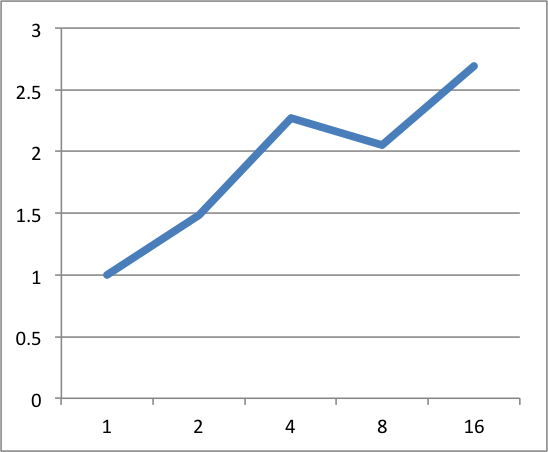
p1 1.04 0.99 1.05

p2 0.83 0.86 0.91

p4 0.60 0.49 0.50

p8 0.31 0.33 0.37

p16 0.32 0.36 0.39

Input: i=200 m=6000 n=500

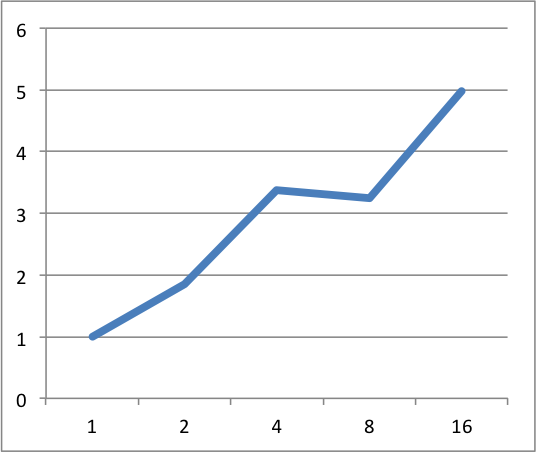
p1 3.18 3.09 2.99

p2 2.20 2.02 2.01

p4 1.38 1.25 1.45

p8 1.45 1.51 1.55

p16 1.05 1.17 1.22

Input: i=700 m=6000 n=500

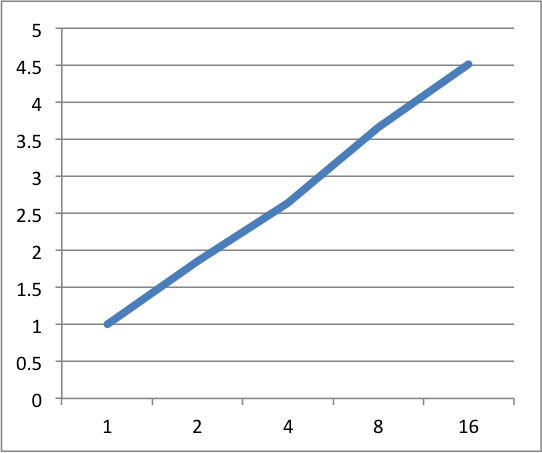
p1 10.89 11.11 10.50

p2 6.02 5.75 5.83

p4 3.07 3.50 3.07

p8 3.33 3.45 3.24

p16 2.22 2.04 2.27



Input: i=1000 m=7000 n=500

p1 17.85 18.21 17.92

p2 9.89 9.54 9.65

p4 6.73 6.72 7.05

p8 4.91 4.74 5.08

p16 3.54 3.70 4.70

The machine used to test the parallel version of SOR program is amoeba-n3. Different inputs (various iterations and matrix sizes) as well as different number of CPU cores are utilized to see and record the behaviors of the parallelized program and the speedup in performance.

As we can see from the charts, the speedup increases, as more numbers of CPU cores are available. However, from diagrams 2 and 3, when we increase the numbers of cores from 4 to 8, we actually see a decrease in speedup. This is because, the threads are currently being assigned to the CPU cores randomly; they are not assigned to specific cores. This could be a problem if the system has multiple sockets and each socket has its own cores. And the CPU of amoeba-n3 is this case.

Illustration:

Amoeba-n3

|  |  |
| --- | --- |
| P0, P2, P4, P6  P8, P10, P12, P14  P16, P18, P20, P22 | P1, P3, P5, P7  P9, P11, P13, P15  P17, P19, P21, P23 |

Socket1 Socket2

When variables are cached in a core in one socket and are shared to a core in the other socket will create overhead because of locality. It simply takes more time to reach and talk to the cores in a different region. So, as the thread binding method is not used, it directly affects to the performance of the parallel SOR program.

Part 2. Main assignment:

Usage:

./gauss <matrixfile> <noOfThreads> <method>

-m1, -m2, -m3 are used as the arguments for the <method>

Parallelization Strategy (for –m2):

After initializing the N by N matrix in the main function, threads are created according to the argument passed by the user. Each thread, then calls a function called “work\_thread(void \*lp)” and the task\_id is passed.

First thing done in the method is to set the affinity of the pthreads to specific CPU cores so that the work is done efficiently and evenly divided to maximize parallelization. Then, we put our first barrier() to synchronize all of the pthreads before jumping into the main work.

The clock starts and the main loop operates. It runs from i=0 to N (iterating rows), and we get the pivot of row i. Getting the pivot value cannot be parallelized because it contains value comparisons, thus one thread runs the function and the rest awaits. Once the thread returns, the row i is now the pivot row, matrix[i][i] becomes 1 and the remaining values in the row are scaled in parallel. (e.g. if we have 4 threads and N = 10, T0 runs {1,5,9}, T1 runs {2,6}, T2 runs {3,7}, T3 runs {4,8}). Then, the rest of the matrix is factorized using a loop from j=i+1 to N (iterating rows), and the values in row j are also calculated in parallel (the same as above). Then, the main loop reaches the bottom, a barrier() call is made to synchronize the pthreads and starts again with i=i+1. After all is done, the clock stops.

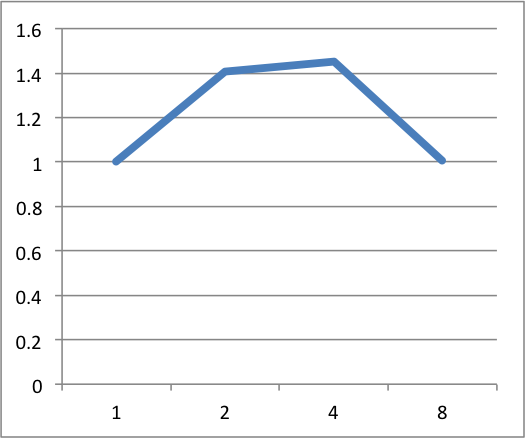
Lastly, the matrix is solved, and the elapsed time and error is displayed.

2-1. Data used for the parallel version of gauss elimination program

MACHINE USED: 9838n-e10 in Burnaby CSIL (8 core CPU)

Number of Processors and Elapsed time - x-axis: Number of processors

in each trial (measured in sec) y-axis: Speedup



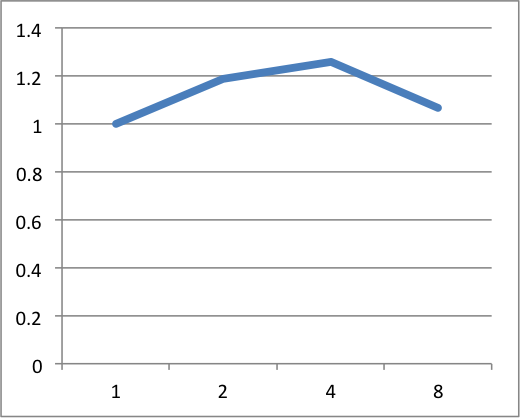
Input: jpwh\_991.dat

p1 0.307 0.306 0.305

p2 0.218 0.215 0.220

p4 0.212 0.210 0.211

p8 0.305 0.302 0.304



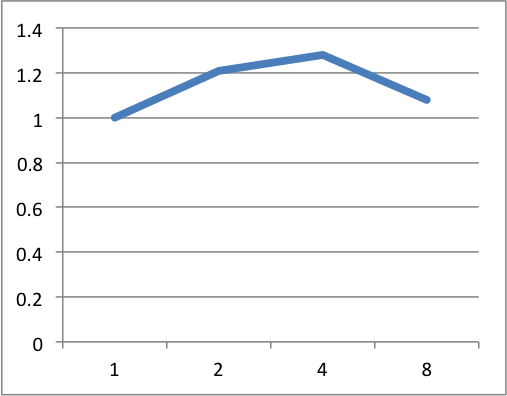
Input: matrix\_2000.dat

p1 2.75 2.74 2.73

p2 2.30 2.32 2.31

p4 2.17 2.18 2.19

p8 2.57 2.58 2.56



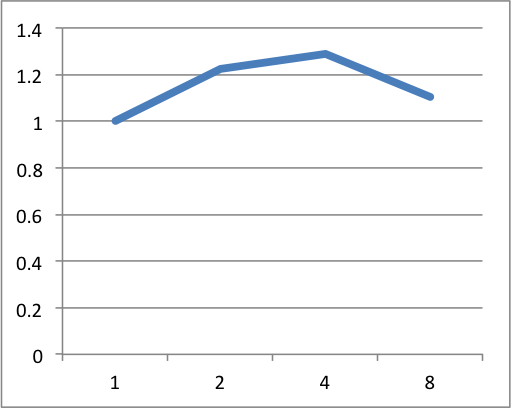
Input: sherman5.dat

p1 12.5 12.5 12.5

p2 10.3 10.4 10.3

p4 9.76 9.77 9.78

p8 11.4 11.6 11.7

Input: sherman3.dat

p1 43.1 43.0 43.0

p2 35.1 35.3 35.2

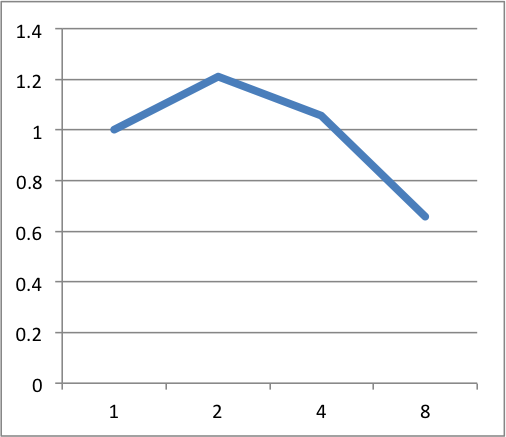
p4 33.3 33.6 33.4

p8 39.1 39.0 38.9

MACHINE USED: 9838n-e07 in Burnaby CSIL (8 core CPU)

Number of Processors and Elapsed time - x-axis: Number of processors

in each trial (measured in sec) y-axis: Speedup



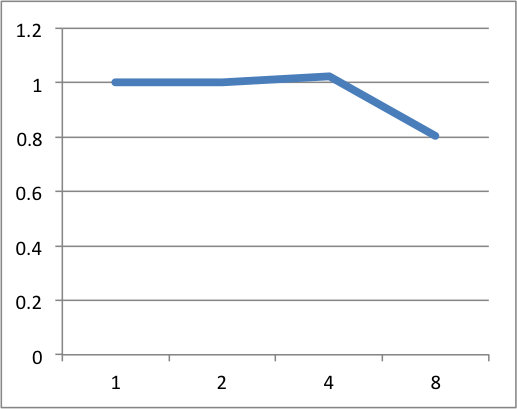
Input: jpwh\_991.dat

p1 0.208 0.205 0.206

p2 0.168 0.171 0.172

p4 0.195 0.196 0.195

p8 0.319 0.314 0.310



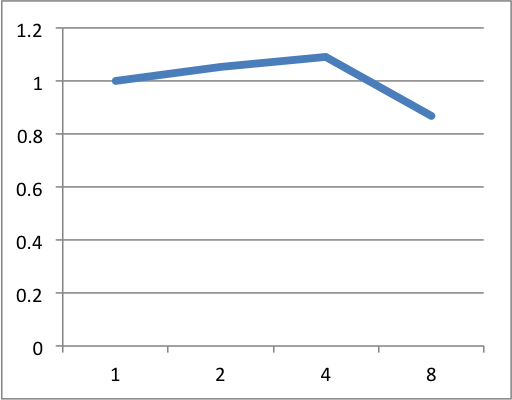
Input: matrix\_2000.dat

p1 2.10 2.08 2.05

p2 2.05 2.07 2.09

p4 2.03 2.03 2.02

p8 2.57 2.58 2.59



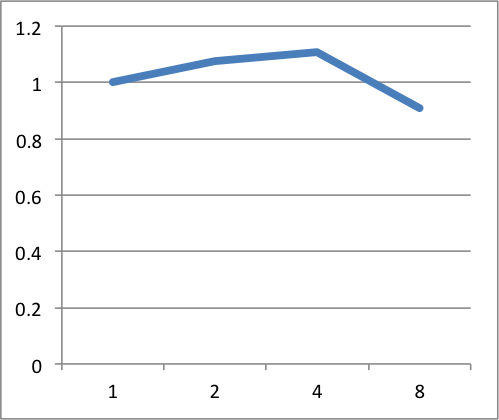
Input: sherman5.dat

p1 10.06 10.06 10.08

p2 9.55 9.55 9.57

p4 9.21 9.21 9.24

p8 11.5 11.6 11.7

Input: sherman3.dat

p1 35.3 34.7 35.5

p2 32.5 33.1 32.4

p4 31.7 31.8 31.7

p8 38.4 38.8 38.9

The machines used to test the parallel version of gauss elimination program are two 8-core machines in CSIL (e07 and e10). Different matrixes are used as inputs and different number of CPU cores is utilized to see and record the behaviors of the parallelized program and the speedup in performance.

As we can see from the charts, the speedup generally increases, as more numbers of CPU cores are available (up to 4 cores). However, when we increase the numbers of cores from 4 to 8, we actually see a decrease in speedup. This is because; the latency for synchronizing the threads becomes larger when more threads are assigned. In order to beat the bottleneck, we need a larger input data so that more parallelization can be done. The two machines produce similar performance graphs even though e07 machine has a higher single thread process.

2-2. Comparisons between different implementations for gauss elimination

The method we implemented (-m2) is a 1D Column Cyclic Layout. Each column of each row is assigned to and calculated by a thread cyclically (described in the parallelization strategy section). The two other methods are used to compare the performance between different implementations.

The first method, -m1 is 1D Column Blocked Layout. It is similar to –m2, but columns are grouped as blocks and each thread is responsible for one block. (e.g. if we have 4 threads and 8 columns, col1, 2 are grouped as block 1 and it is handled by thread 0 … col6, 7 are grouped as block 3 and it is handled by thread 3).

The other method, -m3 is 1D Column Block Cyclic Layout. It is a combination of –m1 and –m2 methods, it creates blocks depending on the variable b. So, there can be more number of blocks than the number of threads. (e.g. if we have 4 threads and 8 columns, b = sqrt(N / task\_num) = 1. There will be 8 blocks and thread 0 handles block 0,4 … thread 3 handles block 3, 7).

Comparison:

MACHINE USED: 9838n-e10 in Burnaby CSIL (8 core CPU)

Number of Processors and Elapsed time - x-axis: Number of processors

in each trial (measured in sec) y-axis: Speedup

Input: jpwh\_991.dat

Input: matrix\_2000.dat

Input: sherman5.dat

Input: sherman3.dat