CMPT 431

Assignment1

Report

Collaborators: Brian Pak Rafay Tanzeel

Student ID: 301168028 301243667

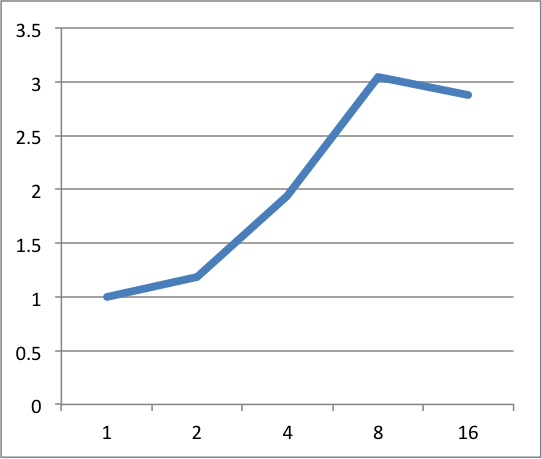
SFU ID: [bpak@sfu.ca](mailto:bpak@sfu.ca) [rtanzeel@sfu.ca](mailto:rtanzeel@sfu.ca)

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

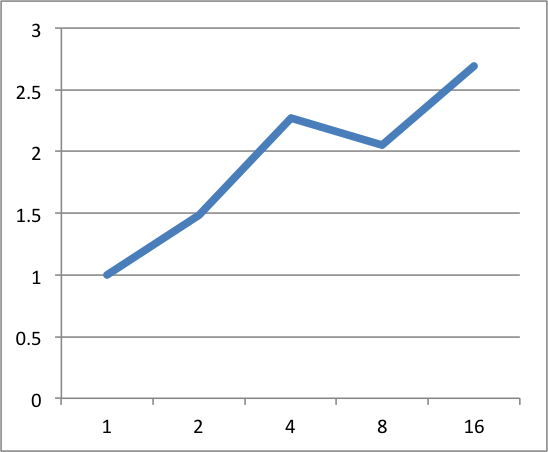
p1 1.04 0.99 1.05

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

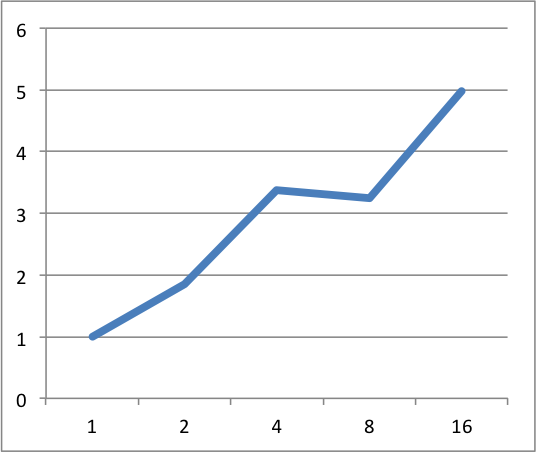
p1 3.18 3.09 2.99

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

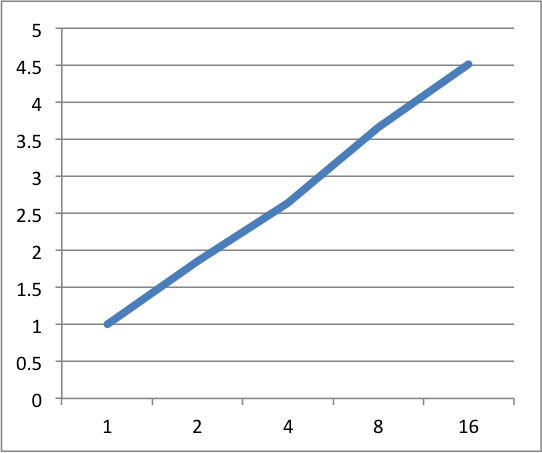
p1 10.89 11.11 10.50

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:

Parallelization Strategy:

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.