PA 4

CS 615

By Samson Haile

May 4, 2017

Changes

- Added input for matrices A & B and output for result matrix
- Implemented two variations of parallel multiplication
- Since original sequential multiplication ran slowly, parallel program was adapted to do sequential multiplication on 1 core

Assignment Description

The objective of this assignment is to determine differences in computation time when executing matrix multiplication. The computation involved multiplies two square matrices of same dimensions and times how long it takes to calculate the resultant matrix. The execution time is evaluated on different sizes of data, utilizing subsets of a single set of matrices represent the different sizes of data that are timed. In the case of a parallelized algorithm, the computation time is expected to decrease with an increase in the number of cores utilized. With respect to parallel computation, subtasks of the computation are determined by dividing the number of elements to be computed equally over the number of cores utilized. Additional changes might be needed to the parallel program to ensure memory constraints don't interfere with the program's execution. The sequential algorithm will be plotted in a two dimensional matrix dimension size versus time graph. The parallel algorithms will be plotted in a three dimensional matrix dimension size versus cores used versus time graph. An assessment will then be made to explain trends and differences between the graphs.

Assignment Methodology

The assignment was implemented in the form of two c files and two batch files, as well as a makefile. Each c file and batch file match to designate one of the two implementation types of the matrix multiplication. This includes sequential and parallel computation of the matrix multiplication algorithm. The sequential code simply executes the computation code on a single core, iterating across the different matrix sizes. The parallel programs use the general heuristic the sequential does, but involves extra components for properly sending and receiving work between the master and slave nodes. The number of cores the parallel programs run on is modified through the -n and -N parameters of the batch scripts used to run the programs. This is to ensure that the programs request only as much resources as it uses.

The sequential code operates by first reading the data into a two dimensional vector data structure. This operation has no impact on the execution time since it is done before any sorting

is performed. After all data is read into their container objects, timing starts. The sequential algorithm then uses three nested for loops in order to calculate the resultant matrix, computing each element at a time. Timing is then stopped upon having calculated every element of the resultant matrix.

There are two variations of the parallel code implemented in a single file. The first variation is canon, of which matrices are split into smaller square submatrices which are distributed amongst all processes. The processes then swap and multiply submatrices in such a manner that each process is able to calculate a complete submatrix of the resultant matrix. The second variation takes a set of rows instead of submatrices and performs an action similar to cannon, swapping and multiplying rows of matrices in such a manner that each process computes a set of rows from the resultant matrix. The benefit of using the second method over canon is that you are not restricted by the matrix dimension divisibility.

The c code is compiled through the usage of makefile which is capable of producing an executable to run the code, as well as the ability to remove the produced executable from the directory. Once the executable is produced, one of two batch files can be run in the form of the command sbatch <file_name>. The batch file seqMat.sh measures the time it takes to multiply two matrices by each other in a sequential manner.The batch file parMat.sh measures the time it takes to multiply two matrices by each other in a parallel manner.

Data and Analysis

*numerical data is contained in .txt files in repository

The files denoted ‘parc’ as a prefix indicate data files for canon multiplication. The prefix is then followed by a number to indicate the number of cores used for the test run. This is then followed by a dash number to indicate the number trial. The suffix of each file is a .txt.

Alternative to the prefix ‘parc’ are files denoted with prefix ‘par’ to show they are data files for the alternative parallel program implementation.

Remaining files denoted with additional tags ‘avg’ indicate average of data across trials. A denotation of ‘All’ indicates all compiled average data in a single file. A denotation of ‘speed’ indicates speedup of averaged data. Lastly, a denotation of ‘eff’ indicates efficiency of averaged data

Graphs

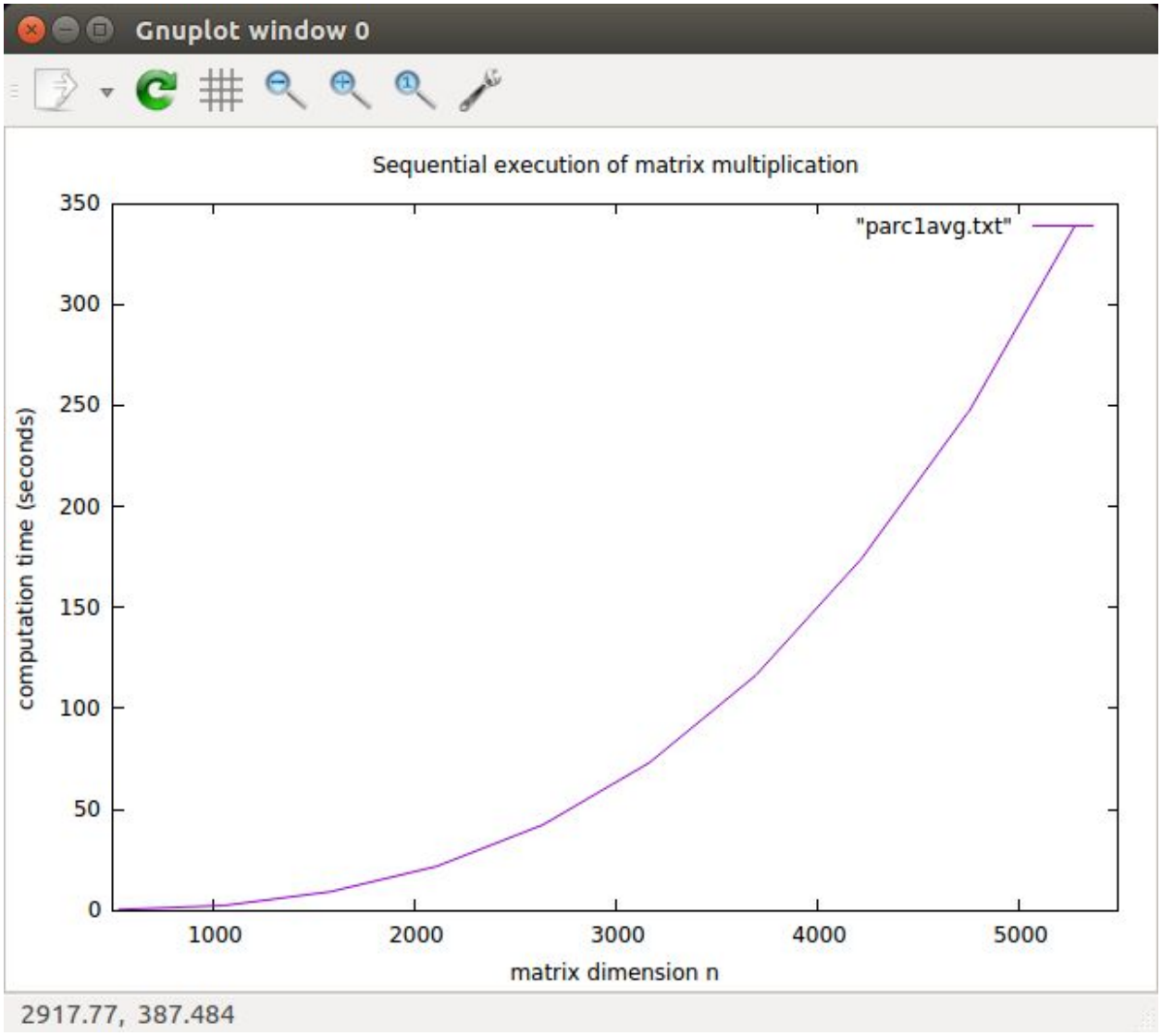

Figure 1:

The graph shows the execution time of sequential matrix multiplication over varying sizes of matrices. The graph displays the polynomial relation between the number of dimensions of the matrices multiplied and computation time. The legend denotes the file representing the average of 5 sets of sequential runtime data.

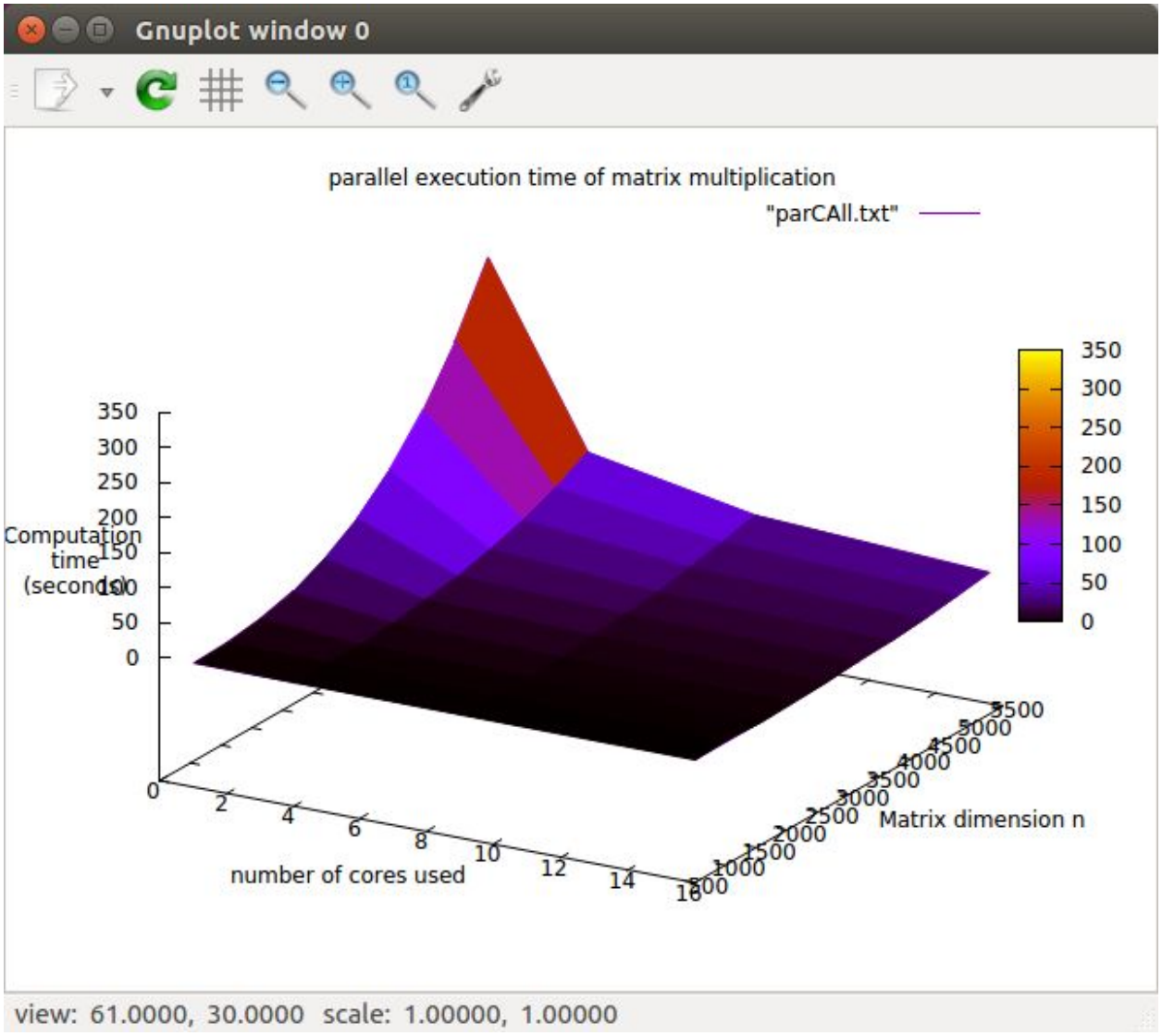

Figure 2:

Graph shows the parallel execution time of matrix multiplication using canon's algorithm. The graph illustrates the pattern of execution times using data testing at 1,4,9, and 16 cores.

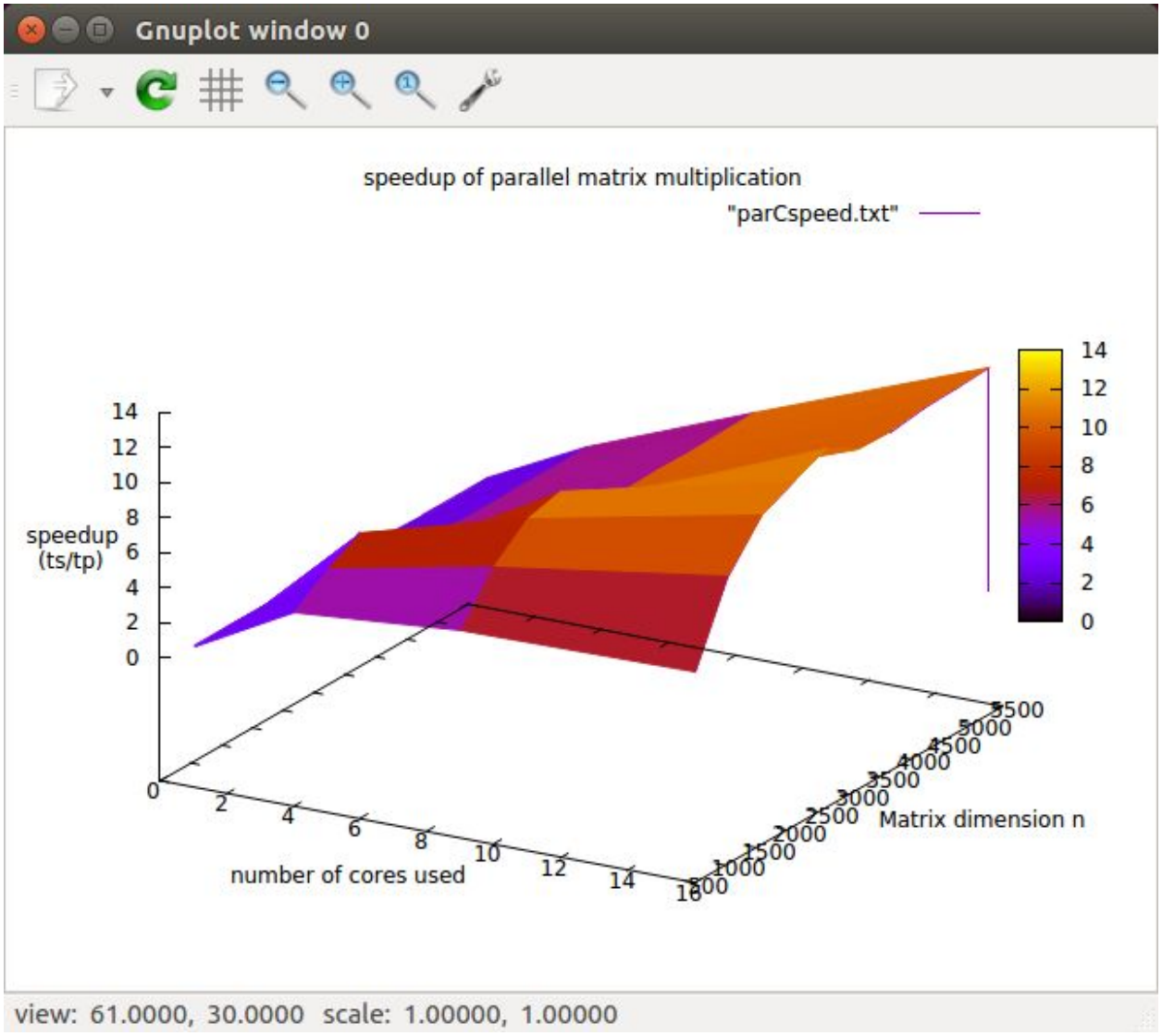

Figure 3:

The graph illustrates the speedup of the program over a varying number of cores. The graph shows a steady linear increase in speedup as the number of cores used increases

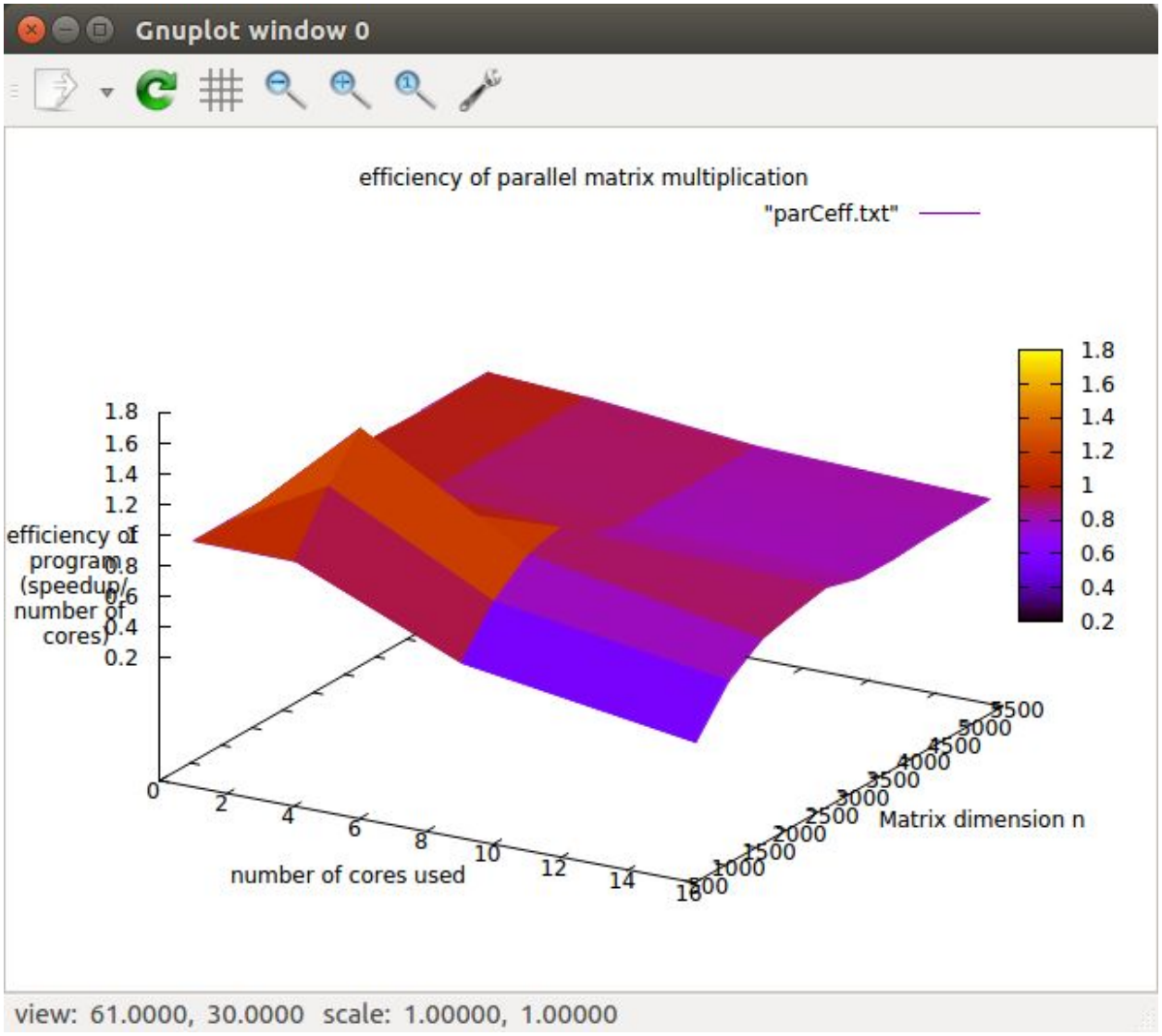

Figure 4:

The graph illustrates the efficiency of the program over a varying number of cores. The program demonstrates an initial jump in efficiency, likely due to caching effects on relatively small matrix sizes. The program then shows a steady decrease in efficiency with more cores added

Examining the sequential computation graph, the computation time is shown to increase parabolically, in accordance with the O(n^3) projected complexity of the algorithm. The parallel computation graph also displays expected results, quickly decreasing in execution time upon using 4 cores. The slope of the graph quickly flattens out by the time 16 cores of execution is reached, implying a decrease in efficiency with an increase in cores utilized. However, it can also be observed from the computation graphs that speedup is greatest when multitasking on a larger data set, likely due to a reduction of memory swapping when spreading data across multiple processes. In this case, since canon's algorithm is used, more execution time is expended communicating data between processes instead of looking up data in memory.

Observing the speedup graph, a steady increase can be observed from the graph as the number of cores is increased. It can also be observed that speedup is poor on small data sets, likely due to the the execution time being dominated by communication without making up for memory lookups that would occur in a sequential execution. Within the efficiency graph, hints of the speedup being superlinear are shown around when the program is run on 4 cores. Since this speedup occurs when data sizes are minimal, it can be said that the superlinear speedup is due to minimal communication time and minimal cache misses.