PA 3

CS 615

By Samson Haile

March 29, 2017

Assignment Description

The objective of this assignment is to determine differences in computation time when executing the bucket sort algorithm. The computation involved organizes the data into one of several buckets representing a specific range of data. The execution time is evaluated on different sizes of data, utilizing subsets of a single set of an unorganized data set to 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 divided by partitioning the data set into n sets, where n represents the number of cores used. The sequential algorithm will be plotted in a three dimensional array number of array size versus number of buckets used versus time graph. The parallel algorithms will be plotted in a three dimensional array 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 bucket sort. This includes sequential and parallel computation of the bucket sort algorithm. The sequential code simply executes the computation code on a single core, iterating across the different array sizes and sorting subsets of a large data, increasing the subset size as the execution progresses. 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 code into a vector data structure. This operation has no impact on the execution time since it is done before any sorting is performed. An array is then initialized to handle receiving the sorted array for each subset iteration. Several individual arrays are also specified at the start of each sort iteration to represent the buckets for the bucket sort. The bucket sort algorithm then operates, beginning by determining the largest value in the data set. This largest value will then implicitly define the numerical range for which each bucket represents. Each data value is then placed into the bucket matching the data value's respective numerical range. After this is complete, each bucket is sorted sequentially using an insertion sort algorithm. Upon sorting all of the buckets, they are inserted into the result array in increasing order of the arranged buckets, resulting in the sorted array.

As for the parallel algorithm, a few more additional communication steps are added in between some steps of the sequential algorithm. The first thing done is that the master process determines the max value of the data set. Following this, the master process splits the array to be sorted into n different sections, where n represents the number of processes running, and distributes them evenly amongst all running processes. In addition, the master process sends the max value it calculated to each process. After all parts of the data are distributed, the master process starts the timer. The first timed operation is each process's organization of their assigned data into a bucket matching the data value's respective numerical range. After all processes categorize all of their assigned data, they send each data bucket to the corresponding process that matches via its task id. The manner of which this is done is by each process taking turns performing an MPI_Send to all other processes, while other processes that aren't sending receive from the appropriate process. After the circulation of data between processes is complete, each process insertion sorts their resulting bucket and the timing finishes.

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 three batch files can be run in the form of the command sbatch <file_name>. The batch file seqSort.sh measures the time it takes to bucket sort different sized data sets in a sequential manner. The batch file parSort.sh measures the time it takes to bucket sort different sized data sets in a parallel manner.

Data and Analysis

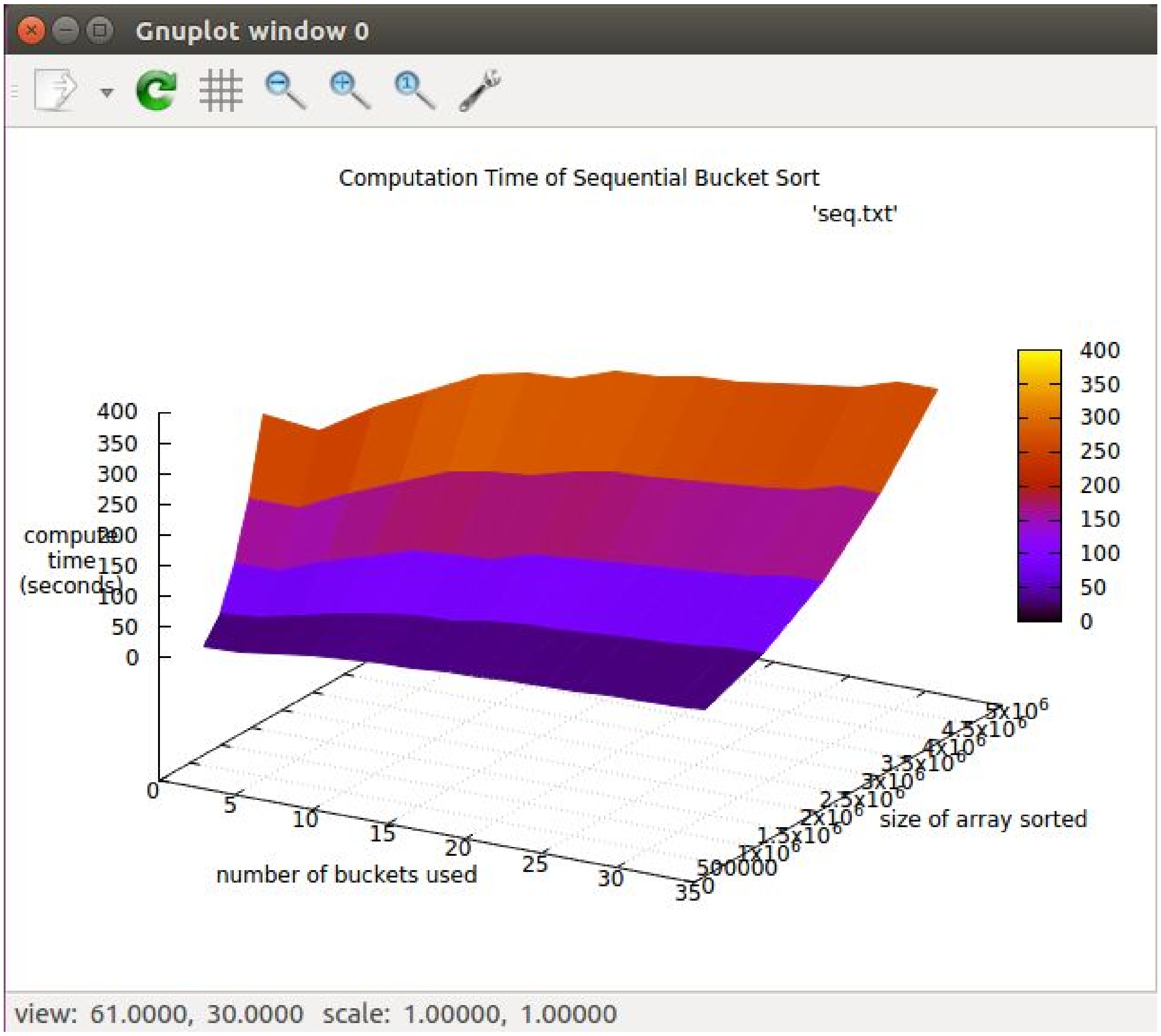

Figure 1:
The graph shows the execution time of a sequential bucket sort algorithm over varying sizes of arrays and bucket sizes. As more buckets are used, the program is able to sort larger sized arrays within five minutes. The legend denotes the file representing the data.

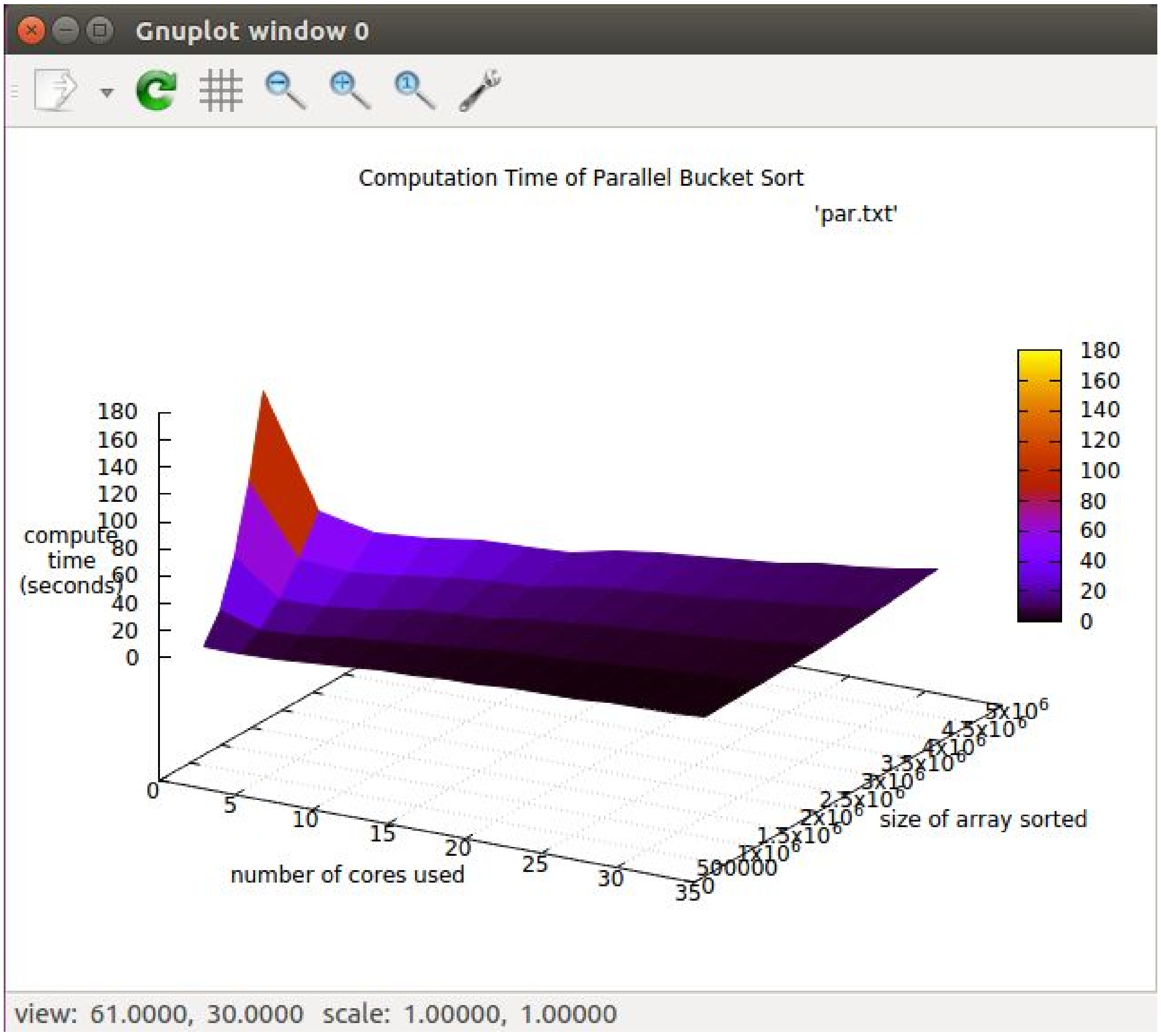

Figure 2:
The graph shows the execution time of a parallel bucket sort algorithm over varying sizes of arrays and bucket sizes. The execution time is shown to decrease with an increase of cores, despite any increase in array size. The legend denotes the file representing the data.

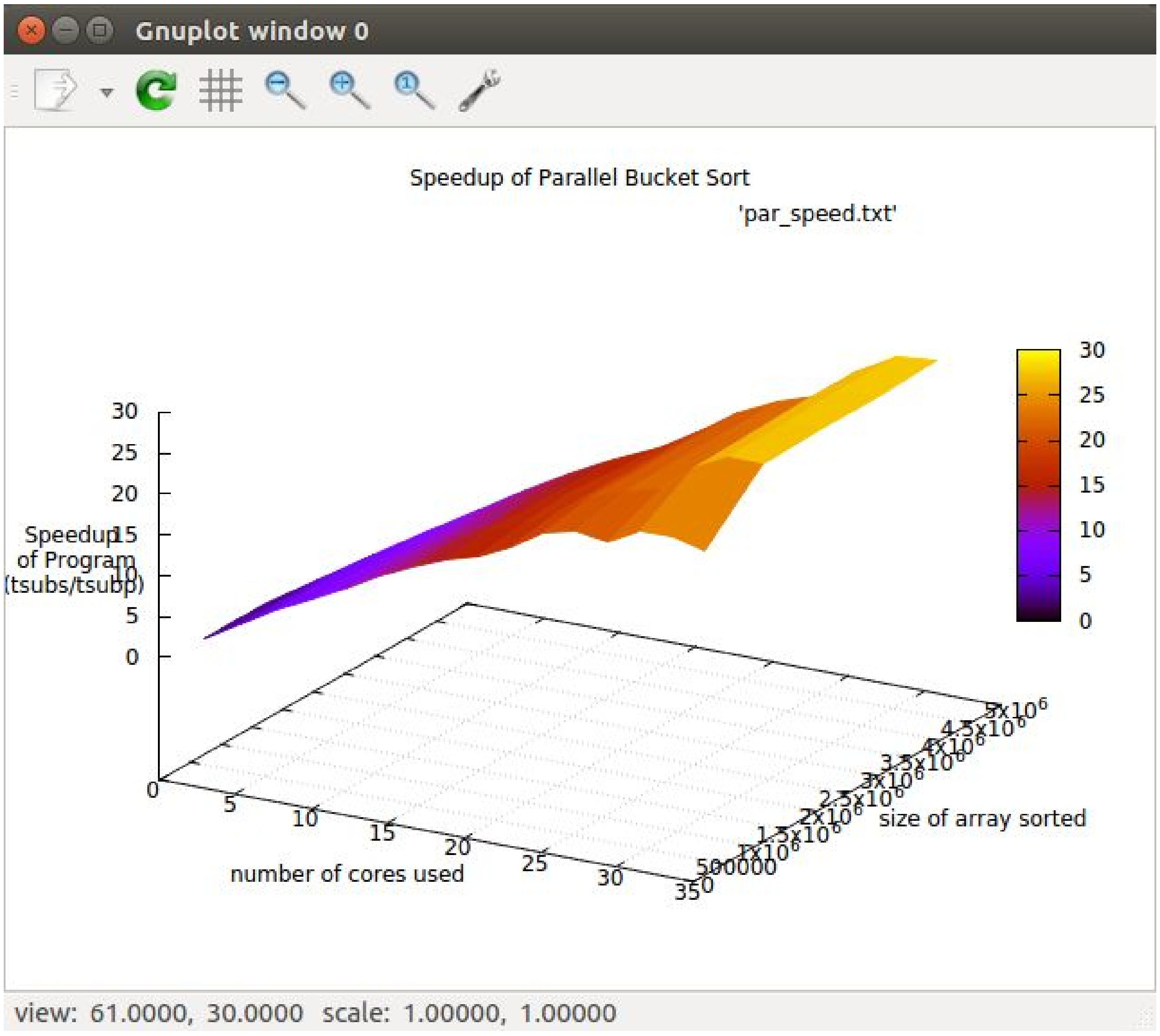

Figure 3:
The graph shows the speedup of a parallel bucket sort algorithm over varying sizes of arrays and bucket sizes. The graph indicates a linear speedup with respect to the number of cores used. The legend denotes the file representing the data.

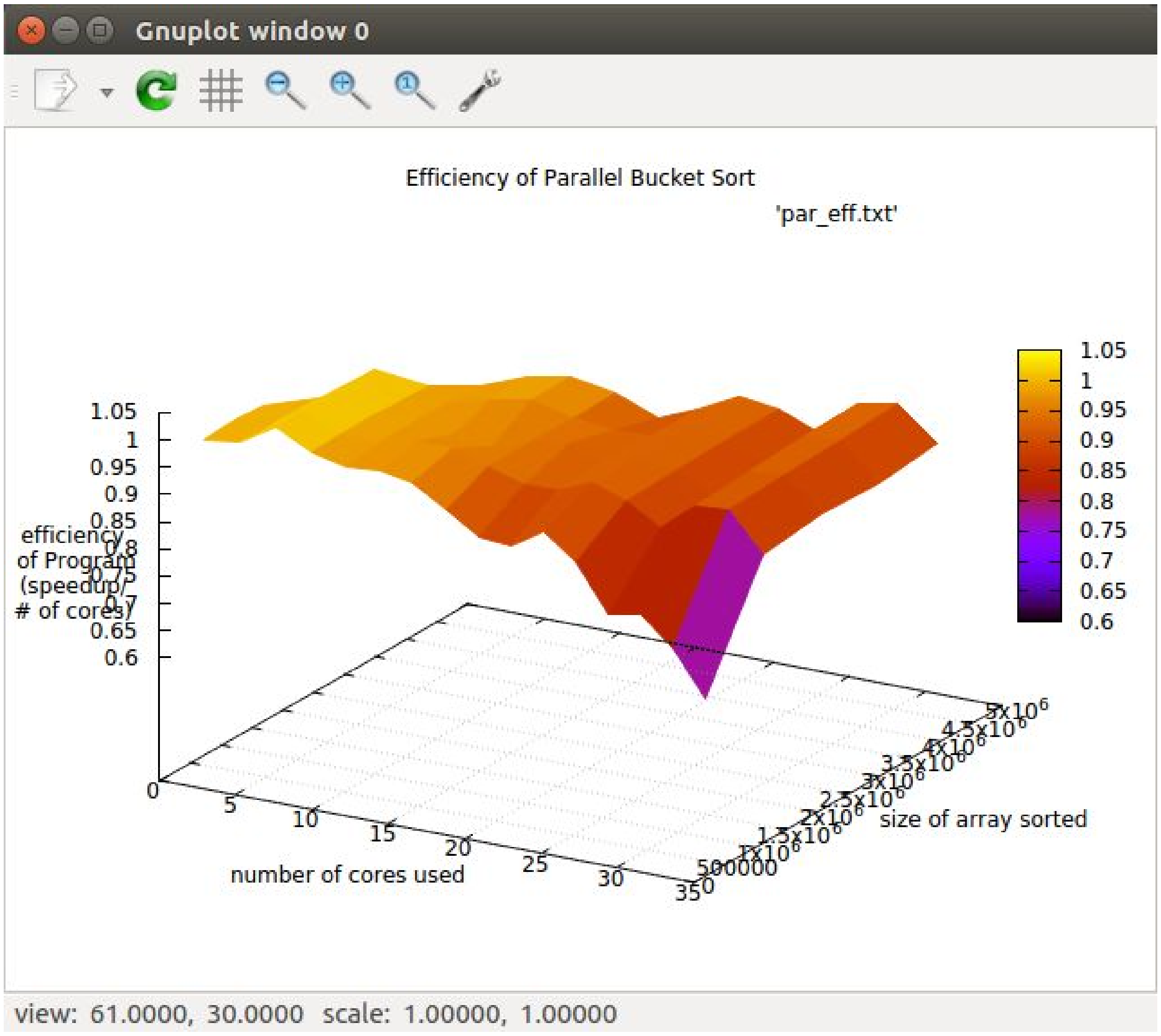

Figure 4:
The graph shows the efficiency of a parallel bucket sort algorithm over varying sizes of arrays and bucket sizes. The graph shows efficiency to be greatest when the size of the array is maximized. When the size of the array is minimized, efficiency decreases with an increase in cores used. The legend denotes the file representing the data.

Examining the sequential computation graph, the computation time is shown to be consistent across all buckets by increasing the size of the array with an increase in the bucket size. The same can be said for the parallel computation time, except the parallelization divides the computation time by the number of cores. The behavior of both graphs is expected as dividing the sorting work for an algorithm that takes O(n^2) using only one bucket would cause a shorter computation time with an increase of buckets as the execution time becomes O(k * (n/k)^2) or O(n^2 / k), where k is the number of buckets and n is the array size.

Looking at the speedup of the graph, the pattern observed is also expected as the speedup is shown to be near linear. The only exception is the slight superlinear speedup made more apparent in the efficiency graph where the efficiency is near 1.03. This occurs when there are six cores used for the parallel program. The speedup can best be explained by the effect of cache, where since each process contains a smaller subset of the data, they are able to fit more of the data in their cache, resulting in faster computations. This coupled with the reduced communication time between processes at 6 cores leads to a slightly over 100% efficiency. The reduced communication time at 6 cores also explains why efficiency starts to drop after using 6 cores.