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

Tables

Table 1: Sequential execution time of Bucket Sort

cores used	size of array	computation time (seconds)
2	240000	13.340779
2	480000	53.540959
2	720000	120.724385
2	960000	214.935031
2	1200000	336.372528
4	320000	11.877993
4	640000	47.472415
4	960000	106.863436
4	1280000	190.546209
4	1600000	298.251136
6	400000	12.813664
6	800000	51.242876
6	1200000	115.456904
6	1600000	205.737117
6	2000000	322.065796
8	480000	13.349994
8	960000	53.343131
8	1440000	120.052958
8	1920000	213.891826
8	2400000	334.850439
10	550000	14.000187
10	1100000	56.100074
10	1650000	126.132832
10	2200000	224.54372
10	2750000	351.305928
12	600000	13.886957

12	1200000	55.504981
12	1800000	121.352477
12	2400000	222.306201
12	3000000	348.020097
14	640000	13.543764
14	1280000	53.842277
14	1920000	116.125731
14	2560000	216.791738
14	3200000	338.943429
16	690000	13.781446
16	1380000	55.084869
16	2070000	123.981423
16	2760000	220.37722
16	3450000	344.68806
18	730000	13.711067
18	1460000	54.804359
18	2190000	123.275869
18	2920000	219.21524
18	3650000	335.060203
20	760000	13.371493
20	1520000	53.457553
20	2280000	120.150557
20	3040000	213.584593
20	3800000	334.032731
22	790000	13.128983
22	1580000	52.53095
22	2370000	118.099156
22	3160000	209.980854
22	3950000	328.156155
24	820000	12.977724
24	1640000	51.808276

24	2460000	116.561339
24	3280000	207.21782
24	4100000	323.915322
26	850000	12.857435
26	1700000	51.403363
26	2550000	115.638114
26	3400000	205.576065
26	4250000	321.340495
28	880000	12.799124
28	1760000	51.148297
28	2640000	115.13166
28	3520000	204.5607
28	4400000	319.674623
30	910000	13.266621
30	1820000	53.020631
30	2730000	119.196635
30	3640000	211.997257
30	4550000	331.234049
32	940000	12.796706
32	1880000	50.787179
32	2820000	114.938303
32	3760000	202.672279
32	4700000	319.039974

Table 2: Parallel computation time of Bucket sort

cores used	size of array	computation time (seconds)
2	240000	6.690409
2	480000	26.805969
2	720000	60.374979
2	960000	107.641156
2	1200000	168.571151

4	320000	2.978596
4	640000	11.942402
4	960000	26.827003
4	1280000	47.813476
4	1600000	74.917193
6	400000	2.075969
6	800000	8.271151
6	1200000	18.653293
6	1600000	33.215822
6	2000000	52.047341
8	480000	1.688843
8	960000	6.733292
8	1440000	15.110572
8	1920000	27.061497
8	2400000	42.380514
10	550000	1.44375
10	1100000	5.723964
10	1650000	12.863138
10	2200000	22.917659
10	2750000	36.003281
12	600000	1.191319
12	1200000	4.696621
12	1800000	10.559679
12	2400000	18.855183
12	3000000	29.383931
14	640000	1.013003
14	1280000	3.942368
14	1920000	8.840624
14	2560000	15.755947
14	3200000	24.60466
16	690000	0.942805

16	1380000	3.637073
16	2070000	8.174081
16	2760000	14.555904
16	3450000	22.751238
18	730000	0.875101
18	1460000	3.313974
18	2190000	7.234727
18	2920000	13.189122
18	3650000	20.691343
20	760000	0.774634
20	1520000	2.913592
20	2280000	6.54054
20	3040000	11.641057
20	3800000	18.224693
22	790000	0.664084
22	1580000	2.553352
22	2370000	5.727777
22	3160000	10.181157
22	3950000	15.852324
24	820000	0.635973
24	1640000	2.369123
24	2460000	5.312073
24	3280000	9.428673
24	4100000	14.684986
26	850000	0.647917
26	1700000	2.280884
26	2550000	5.103092
26	3400000	9.026287
26	4250000	14.101973
28	880000	0.590972
28	1760000	2.000731

28	2640000	4.465859
28	3520000	7.863376
28	4400000	12.264902
30	910000	0.61609
30	1820000	1.934295
30	2730000	4.31614
30	3640000	7.629445
30	4550000	11.854993
32	940000	0.627652
32	1880000	1.890083
32	2820000	4.213942
32	3760000	7.453163
32	4700000	11.579104

Table 3: Speedup of Parallel Bucket Sort

cores used	size of array	Speedup
2	240000	1.99402
2	480000	1.99735
2	720000	1.99958
2	960000	1.99677
2	1200000	1.99543
4	320000	3.98778
4	640000	3.97511
4	960000	3.98343
4	1280000	3.9852
4	1600000	3.98108
6	400000	6.17238
6	800000	6.19537
6	1200000	6.18963
6	1600000	6.19395
6	2000000	6.18794

8	480000	7.90482
8	960000	7.9223
8	1440000	7.94496
8	1920000	7.90392
8	2400000	7.90105
10	550000	9.6971
10	1100000	9.80091
10	1650000	9.80576
10	2200000	9.79785
10	2750000	9.75761
12	600000	11.6568
12	1200000	11.8181
12	1800000	11.4921
12	2400000	11.7902
12	3000000	11.8439
14	640000	13.3699
14	1280000	13.6573
14	1920000	13.1355
14	2560000	13.7594
14	3200000	13.7756
16	690000	14.6175
16	1380000	15.1454
16	2070000	15.1676
16	2760000	15.1401
16	3450000	15.1503
18	730000	15.668
18	1460000	16.5374
18	2190000	17.0395
18	2920000	16.6209
18	3650000	16.1933
20	760000	17.2617

20	1520000	18.3476
20	2280000	18.3701
20	3040000	18.3475
20	3800000	18.3286
22	790000	19.7701
22	1580000	20.5733
22	2370000	20.6187
22	3160000	20.6245
22	3950000	20.7008
24	820000	20.4061
24	1640000	21.8681
24	2460000	21.9427
24	3280000	21.9774
24	4100000	22.0576
26	850000	19.8443
26	1700000	22.5366
26	2550000	22.6604
26	3400000	22.7753
26	4250000	22.7869
28	880000	21.6578
28	1760000	25.5648
28	2640000	25.7804
28	3520000	26.0144
28	4400000	26.0642
30	910000	21.5336
30	1820000	27.4108
30	2730000	27.6165
30	3640000	27.7867
30	4550000	27.9405
32	940000	20.3882
32	1880000	26.8703

32	2820000	27.2757
32	3760000	27.1928
32	4700000	27.5531
32	4700000	27.5531

Table 4: Efficiency of Parallel Bucket Sort

cores used	size of array	Efficiency
2	240000	0.997008
2	480000	0.998676
2	720000	0.999788
2	960000	0.998387
2	1200000	0.997717
4	320000	0.996946
4	640000	0.993779
4	960000	0.995857
4	1280000	0.9963
4	1600000	0.995269
6	400000	1.02873
6	800000	1.03256
6	1200000	1.0316
6	1600000	1.03232
6	2000000	1.03132
8	480000	0.988102
8	960000	0.990287
8	1440000	0.993121
8	1920000	0.98799
8	2400000	0.987631
10	550000	0.96971
10	1100000	0.980091
10	1650000	0.980576
10	2200000	0.979785

10	2750000	0.975761
12	600000	0.971399
12	1200000	0.984839
12	1800000	0.957672
12	2400000	0.982516
12	3000000	0.986991
14	640000	0.954994
14	1280000	0.975525
14	1920000	0.938248
14	2560000	0.982811
14	3200000	0.98397
16	690000	0.913593
16	1380000	0.946587
16	2070000	0.947977
16	2760000	0.946254
16	3450000	0.946894
18	730000	0.870443
18	1460000	0.918742
18	2190000	0.946637
18	2920000	0.923384
18	3650000	0.899625
20	760000	0.863085
20	1520000	0.917382
20	2280000	0.918506
20	3040000	0.917376
20	3800000	0.916429
22	790000	0.898639
22	1580000	0.935151
22	2370000	0.937212
22	3160000	0.937475
22	3950000	0.940947

24	820000	0.850254
24	1640000	0.911172
24	2460000	0.91428
24	3280000	0.915725
24	4100000	0.919066
26	850000	0.763241
26	1700000	0.866792
26	2550000	0.871554
26	3400000	0.875972
26	4250000	0.87642
28	880000	0.773491
28	1760000	0.913029
28	2640000	0.920729
28	3520000	0.929084
28	4400000	0.930864
30	910000	0.717786
30	1820000	0.913694
30	2730000	0.92055
30	3640000	0.926224
30	4550000	0.931349
32	940000	0.637132
32	1880000	0.839698
32	2820000	0.852366
32	3760000	0.849775
32	4700000	0.861034
32	4700000	0.861034

Graphs

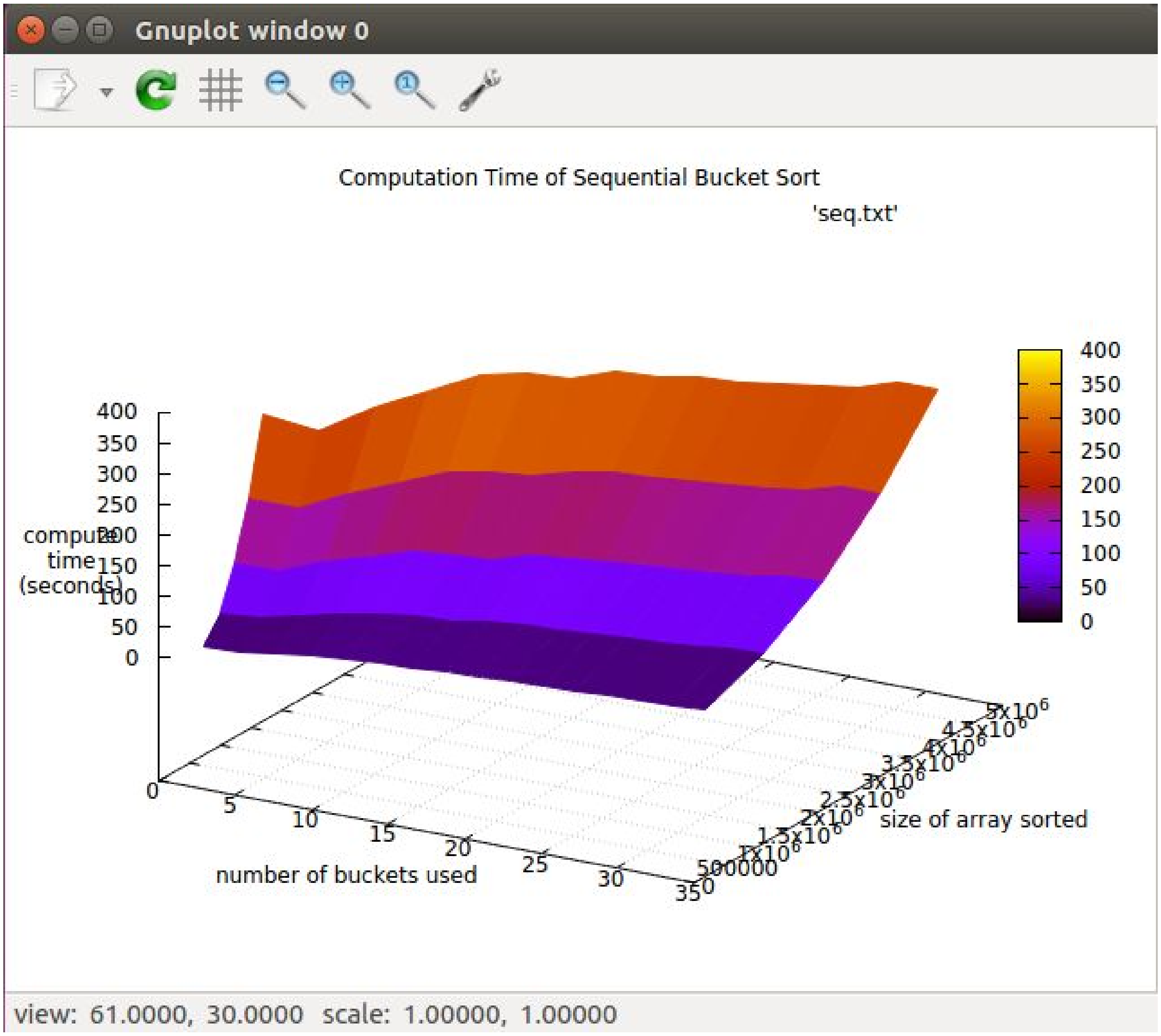

Figure 1:

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.

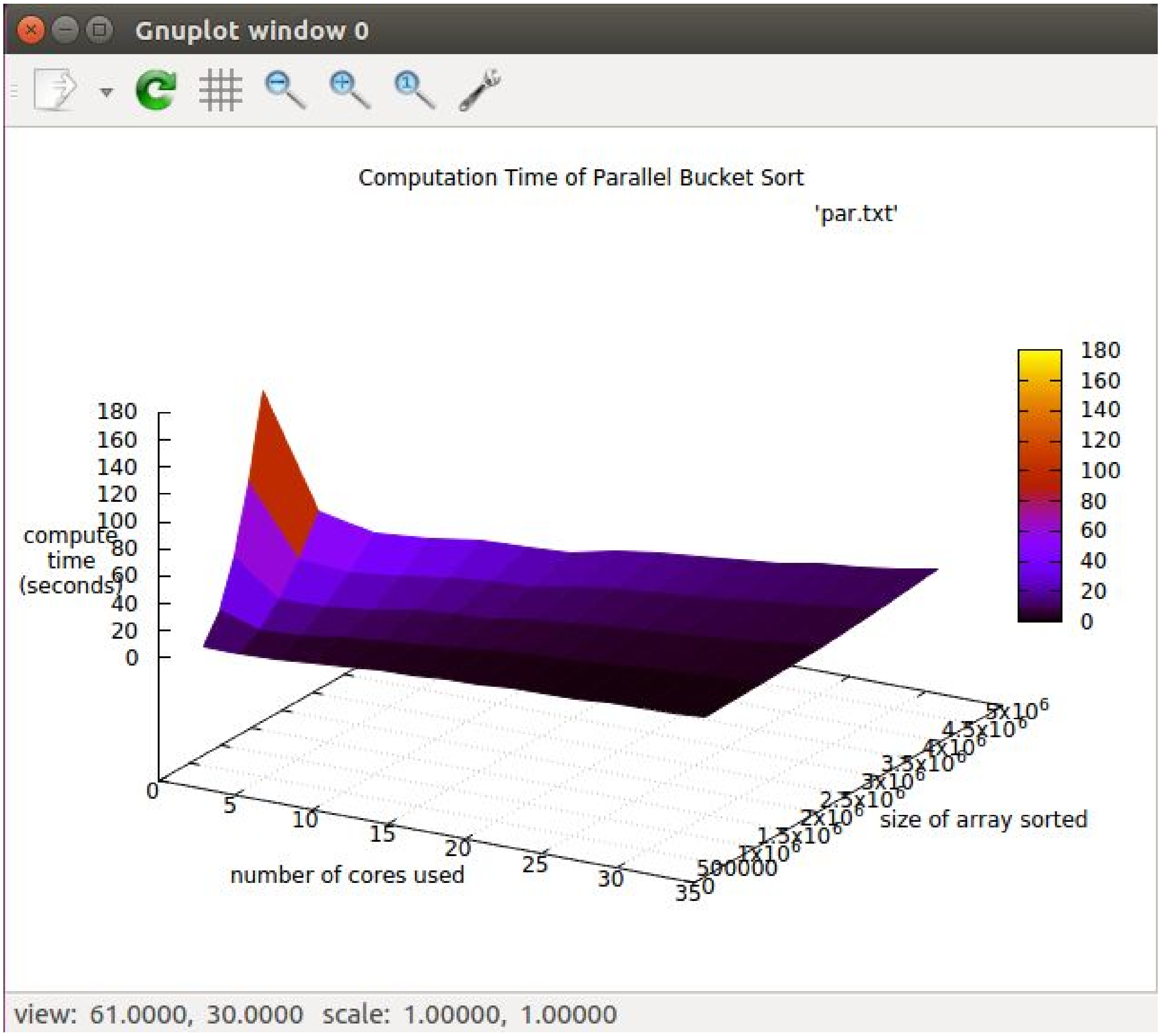

Figure 2:
The graph shows the execution time of a parallel bucket sort algorithm over varying sizes of arrays and bucket sizes. The execution time is shown to decrease with an increase of cores, despite any increase in array size. The legend denotes the file representing the data.

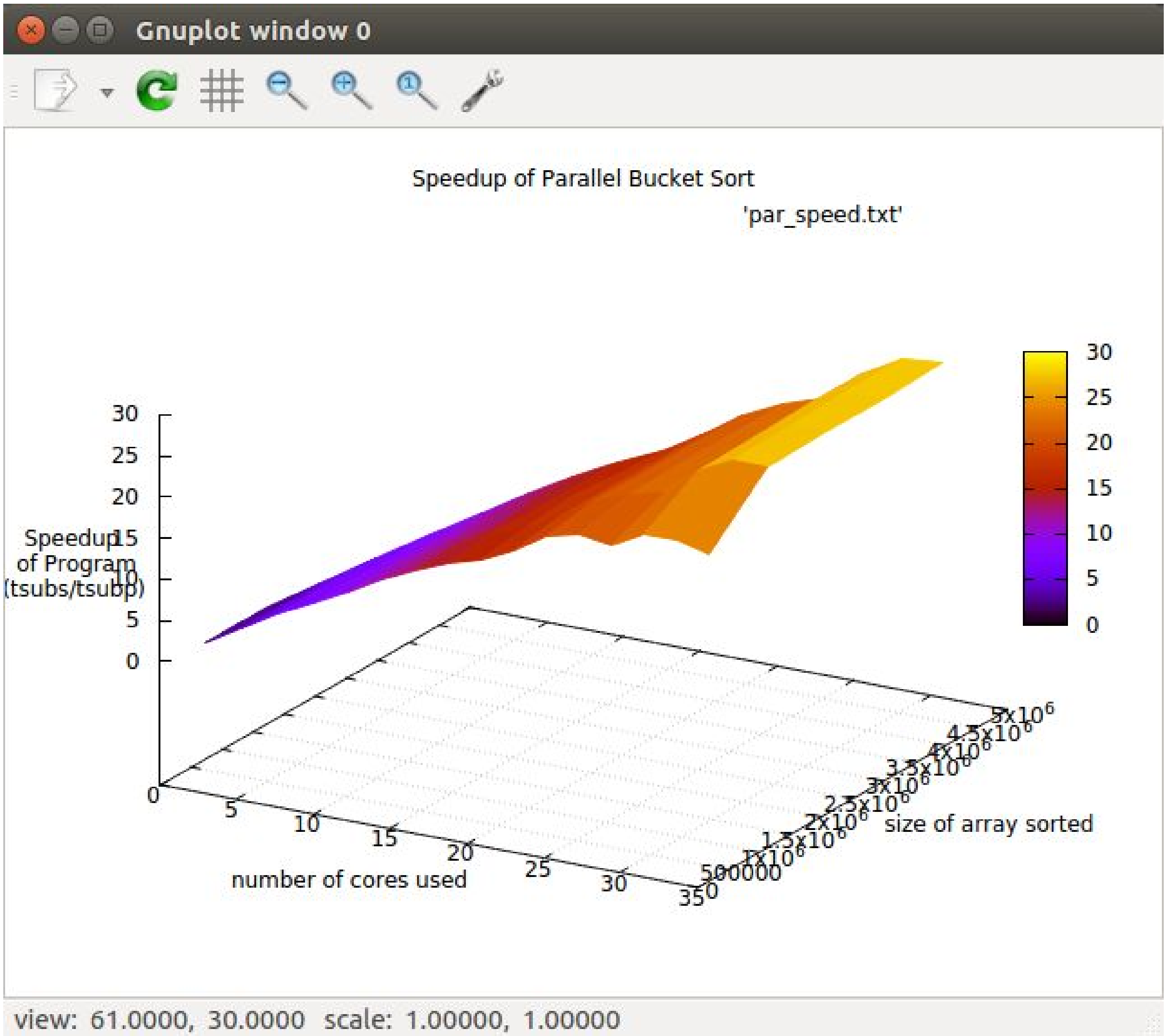

Figure 3:
The graph shows the speedup of a parallel bucket sort algorithm over varying sizes of arrays and bucket sizes. The graph indicates a linear speedup with respect to the number of cores used. The legend denotes the file representing the data.

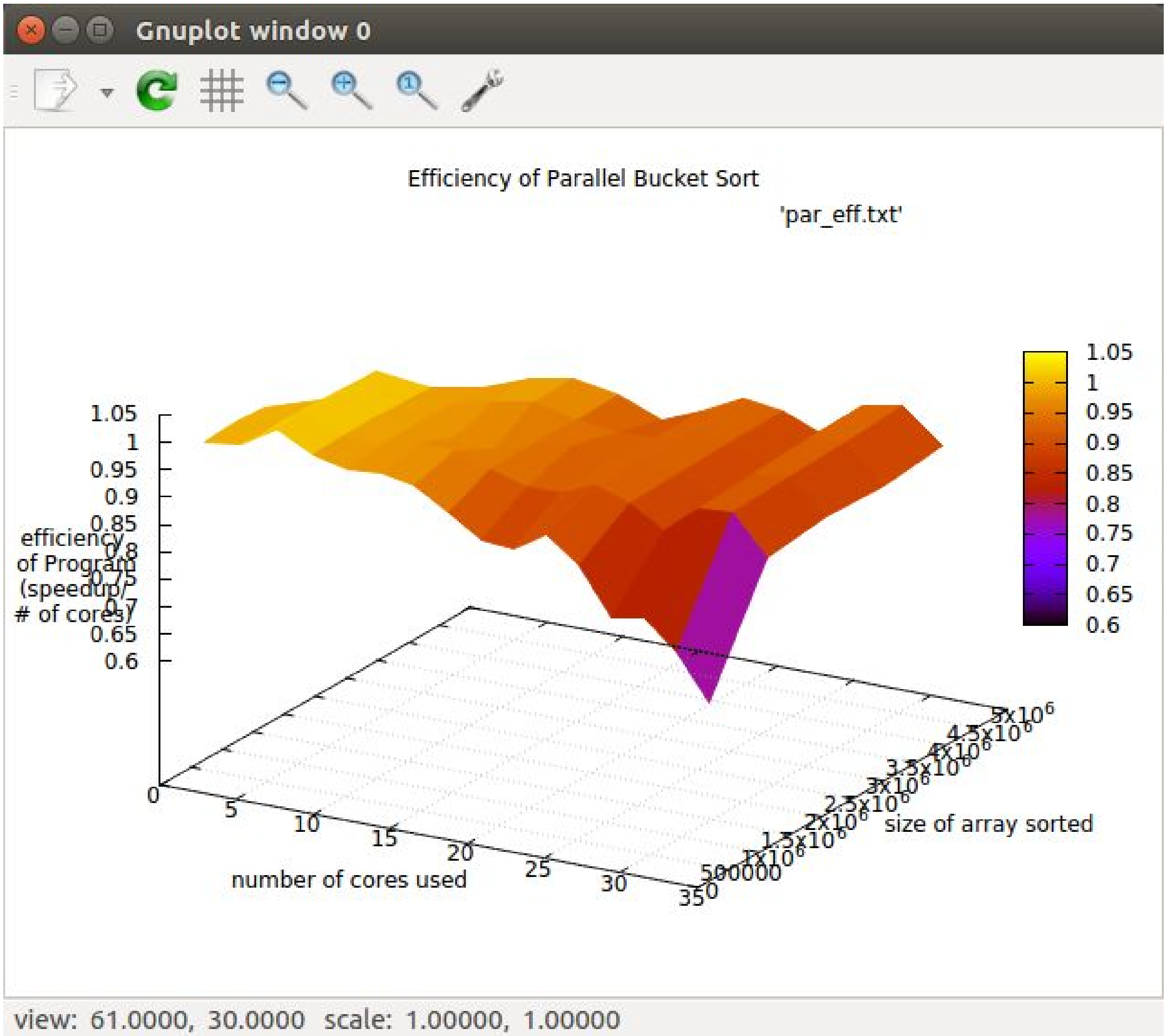

Figure 4:
The graph shows the efficiency of a parallel bucket sort algorithm over varying sizes of arrays and bucket sizes. The graph shows efficiency to be greatest when the size of the array is maximized. When the size of the array is minimized, efficiency decreases with an increase in cores used.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