Project Assignment 03 Report

CS415

3-29-2017

By: Jervyn Suguitan

Table of Contents:

Description of Assignment

Project 03's main purpose was to teach students about bucket sort and how to parallelize the code that creates the bucket sort. This report will only talk about running bucket sort sequentially and how to do it and discussion based on several variables. The timing function that was used to time calculation for this project was MPI_Wtime.

Part 1: Sequential

The first part of the project tasks students in figuring out how to code bucket sort sequentially. The first thing to do however is to create an input file with a specified amount of numbers, this is done using the createValues.cpp. After creating the input file with the set amount of numbers that will be sorted we sort the numbers in the sequential.cpp code. The code reads in the input file and places those numbers into a holder/buffer. This buffer puts all the numbers into buckets. The number of buckets is predetermined to 10 and each bucket has a size of the max amount of possible integers. I do this for the case where all the numbers that need to be sorted are all on the same bucket. When they are initially put into the buckets they are not sorted. Once the numbers are in the buckets I get the numbers from each bucket and sort them using a sort function in the algorithm library. After the buckets are sorted, I put them back into a large array that contains all the values in order. The sorting function I use is bubble sort in which in which I created. The run time of bubble sort $O(n^2)$.

Figure 1 and Figure 2 showcase the run time of Bucket Sort with multiple amount of integers. The only difference between Fig. 1 and Fig. 2 is that Fig. 1 includes the trials with 1500000 and 1000000 integers sorted while Fig. 2 does not. These numbers of integers sorted (100000, 250000, 500000, 1000000, 1500000) were chosen to show times over a very small, small, medium, large, and very large amount of sorted integers. These graphs show that as more numbers that are needed to be sorted, the longer it takes to sort. To specify all of the trials in Fig. 1 and Fig.2 all use a number range of 0 to 10000. Thus the numbers that are sorted will only be between that range.

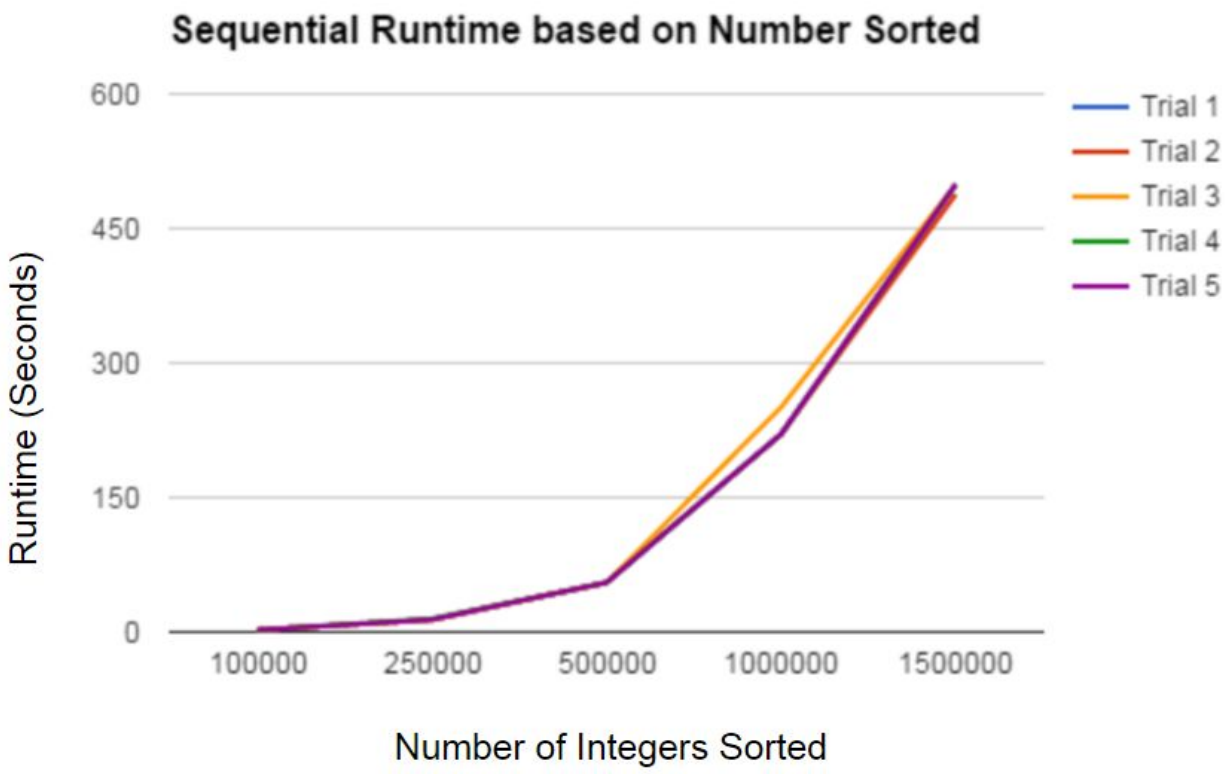

Fig. 1: Graphs that showcase the sequential times for different amount of integers sorted for Bucket sort. The Y axis shows the time in seconds and the x-axis shows the for that specific number of integers sorted. This graph includes times for 1.5 million integers sorted. The times do not vary large amounts since everything is sequential thus there can not be a large speed up. Speed up or slow down may occur due to the processor speed which could be different for every trial. Some of the trials are hard to see as they are behind other lines.

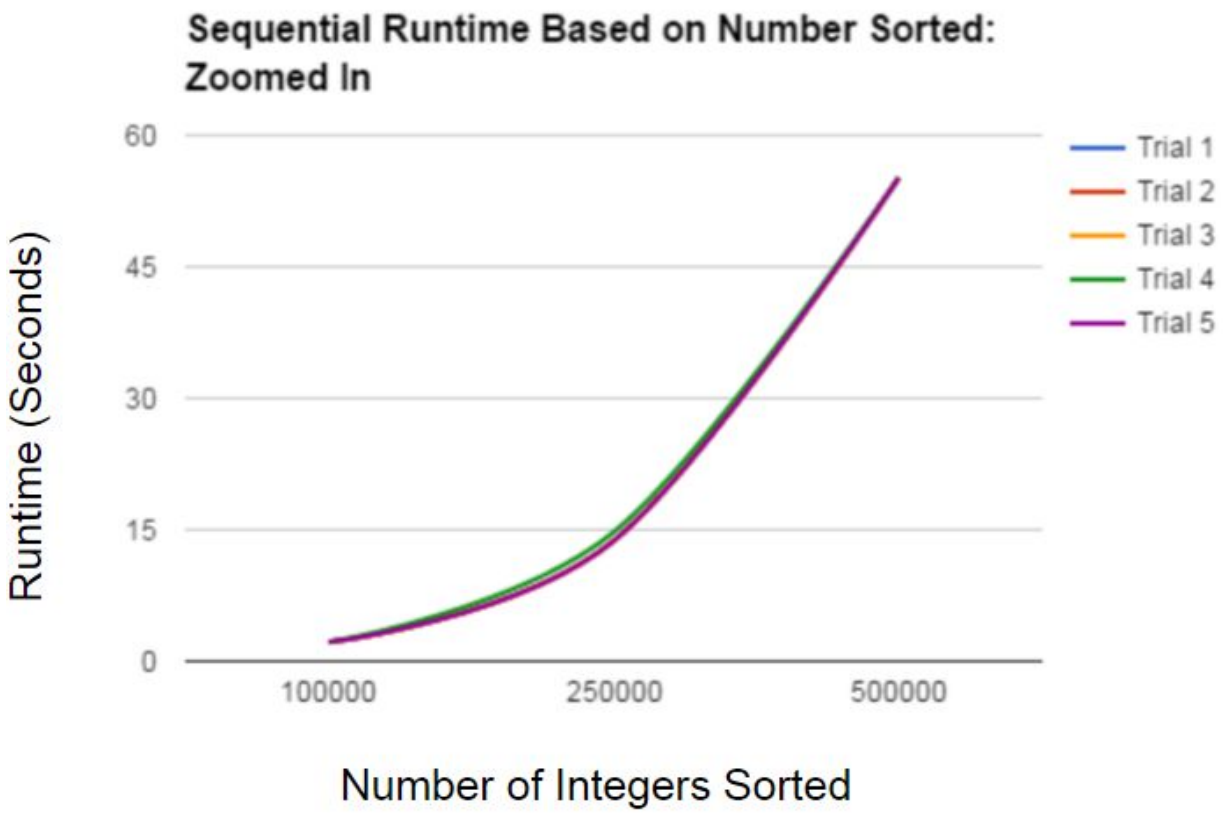

Fig. 2: Graphs that showcase the sequential times for different amount of integers sorted for Bucket sort. The Y axis shows the runtime in seconds and the x-axis shows the number of integers sorted. This graph shows similar data to Fig. 1 but only showcases the trials that sort 100000, 250000, and 500000 integers. The times do not vary large amounts since everything is sequential thus there can not be a large speed up. Speed up or slow down may occur due to the processor speed which could be different for every trial.

The times for each of the sorting do not vary large amounts, one reason this occurs is that there is no message passing between master and slave which takes up time. There is no message passing since it is sequential there is no need to use multiple processors thus the program uses a " -n1 " to show that we will only use one core. Also since everything is sorted one integer at a time due to it being bubble sort.

Table 1 shows the average for sequential time shown as a chart. In the fifth column in the chart, we see the average time of 500 million integers however the range changed with the numbers being sorted could be between 0 - 1000000. This increased the average by around 7 whole seconds. The slowdown is most likely occurred due to the sort function in the algorithm library as the speed of the entire bucket sort is reliant on the speed of that sort. The ranges of the numbers are from 0 to 1 million. Having the range go from 0 to 1 million increase the runtime as numbers that are equal in bubble sort do not need to swap. The highest average time was around 497 seconds which about 8.2 minutes. This runtime is seen when sorting 1.5 million numbers. The reason why sorting only 1.5 million numbers takes a long time is due to the secondary sort, bubble sort. Although bucket sort is $O(n + k)$, bubble sort is $O(n^2)$ which is very slow. **Bubble sort runtime grows exponentially as there are more integers sorted.**

Table that Shows the Average Runtimes of Different amount of Integers: Sequentially

# of Ints	100000	250000	500000	1000000	1500000
1 Processor	2.19119	13.8908	55.4943	225.623	497.537

Table 1: Chart that showcases the average times of different amount of sorted integers. The ranges of the numbers are from 0 to 1 million

Problems Encountered in Sequential

Problems that I have encountered during this part of the project is that I didn't know that I still had to use MPI_Init even though there was no sending to slave functions at all. I needed it for the MPI_Wtime functions. I had to think of a way to use bucket sort without using a queue data structure or a vector. I also had to figure out a way to put the numbers in the buckets correctly. Also finding a set amount of numbers to reach the 5 minute

Part 2: Parallel Bucket Sort

The next part of the projects tasks students to parallelize their sequential code. The idea for the parallel code is somewhat similar to the sequential code. My code reads in a file which takes in how many integers will be sorted and also takes in those integers into a buffer/holder. Essentially the dynamic code breaks up the read in integers that need to be sorted, N, into P, number of processors/buckets. Thus each processor gets around N/P work. The master will send from what part of the buffer each slave will be working on. MPI_Barrier will be called once every processor knows what part of the buffer/holder array they will be working on and know which range of numbers each processor can work on.

The ranges of the numbers that will be sorted range from 0 to 1000000. So for example if there are two processors, processor 1 will work from range 0 to 499999 and processor 2 will work from range 500000 to 10000000.

Once all processors reach the MPI_Barrier, the processors will now put all the numbers into sub buckets. Once all the numbers are put into the correct sub bucket, each processor MPI_Isends their sub bucket into the correct processor/ normal bucket. Once every processor/bucket gets the correct numbers they can work on, they put all the numbers into a very large array. This array is then sorted using bubble sort. Once all buckets finish with their bubble sort a MPI_Barrier is then called. Once all reach the MPI_Barrier, that is when the computation is completed and the timer ends.

Table 2: Table that shows the average run times of different number of integers sorted and the number of processors on the number of integers sorted in **Seconds**

Size	100000	250000	500000	1000000	1500000
1	**2.19119**	**13.8908**	**55.4943**	**225.623**	**497.537**
8	0.36018	2.74381	8.90924	35.4396	235.841
10	0.20208	1.41859	5.64387	22.4735	52.306
12	0.20347	0.98086	3.909	15.9356	41.6392
14	0.10076	0.75722	2.96192	11.493	26.157
16	0.09822	0.55499	2.19599	8.77595	26.0745
18	0.26129	0.45038	2.56128	7.79231	16.6009
20	1.0014	0.41317	1.76869	6.42041	13.5607
22	0.06534	0.37876	1.72468	5.65322	11.8245
24	0.07873	0.30693	1.57496	4.90074	9.99457
30	0.04714	0.23361	0.9039	3.51152	8.10783
32	0.04862	0.21091	0.7838	4.28071	7.28116

Figure 3 is a compilation of all the run times with x-axis showing the Number of Cores used, y-axis is the number of integers sorted and the z-axis the runtimes. From this we can again clearly see that using 1 core (sequential) is the slowest. And adding many cores increase speed exponentially. Eventually there will be a diminishing returns as we add more processors. The run times are seen in Table 2 which showcase the average times of processors on different number of integers sorted. The speed up which I used S(p) = Time of Sequential / Time of Parallel can be seen on Table 3. The speed up varies from greatly from numbers of integers sorted to number of processors.

Table 3: Table that shows the speed up of different number of integers sorted and the number of processors on the number of Integers sorted in **Seconds**

	Speed Up				
	100000	**250000**	**500000**	**1000000**	**1500000**
8	6.0836	5.06261	6.22885	6.36641	2.10963
10	10.8431	9.79201	9.83267	10.0396	9.51204
12	10.7692	14.1618	14.1965	14.1584	11.9487
14	21.7459	18.3445	18.7359	19.6314	19.0212
16	22.3098	25.0291	25.2708	25.7093	19.0813
18	8.38612	30.8426	21.6666	28.9546	29.9705
20	2.18812	33.6204	31.3759	35.1416	36.6896
22	33.5357	36.6742	32.1766	39.9106	42.0769
24	27.8309	45.2574	35.2354	46.0386	49.7807
30	46.4853	59.4626	61.394	64.2523	61.3649
32	45.069	65.8606	70.8016	52.707	68.332

Part 3: Speed Up Analysis **Important Section**

As seen above in Table 3 above the speed up varies from 2 to around 70. Although this is an outrageous claim for other sorts (i.e quicksort, mergesort), this data makes sense due to the fact that this data uses **bubble sort** as its secondary sort. To reiterate bubble sort is $O(n^2)$ a very slow runtime. Since bubble sort is $O(n^2)$ that means that the runtime increases exponentially as more numbers are needed to be sorted. We can see this exponential runtime growth using Table 2 and Fig. 3. The runtime increases exponentially 200 seconds from adding 500000 integers going from 1 million to 1.5 million.

To reiterate that $O(n^2)$ a very slow runtime, Table 4 shows 4 different numbers of Integers sorted with bubble sort. Table 4 shows that the gap from 1000 integers to 50000 integers takes only half of a second while the gap from 150000 integers to 187500 integers takes around 2 minutes to sort. This table also shows us again that as more numbers need to be sorted, the run time grows **exponentially**. Thus as big numbers gain more numbers to sort, there will be a very large runtime increase.

Table 4: Table that shows the runtimes of different number of Integers Sorted on Sequential Bucket Sort(Bubble Sort) Time

Size	1000	50000	150000	187500
Sequential	0.0027	0.55893	4.97509	7.72465

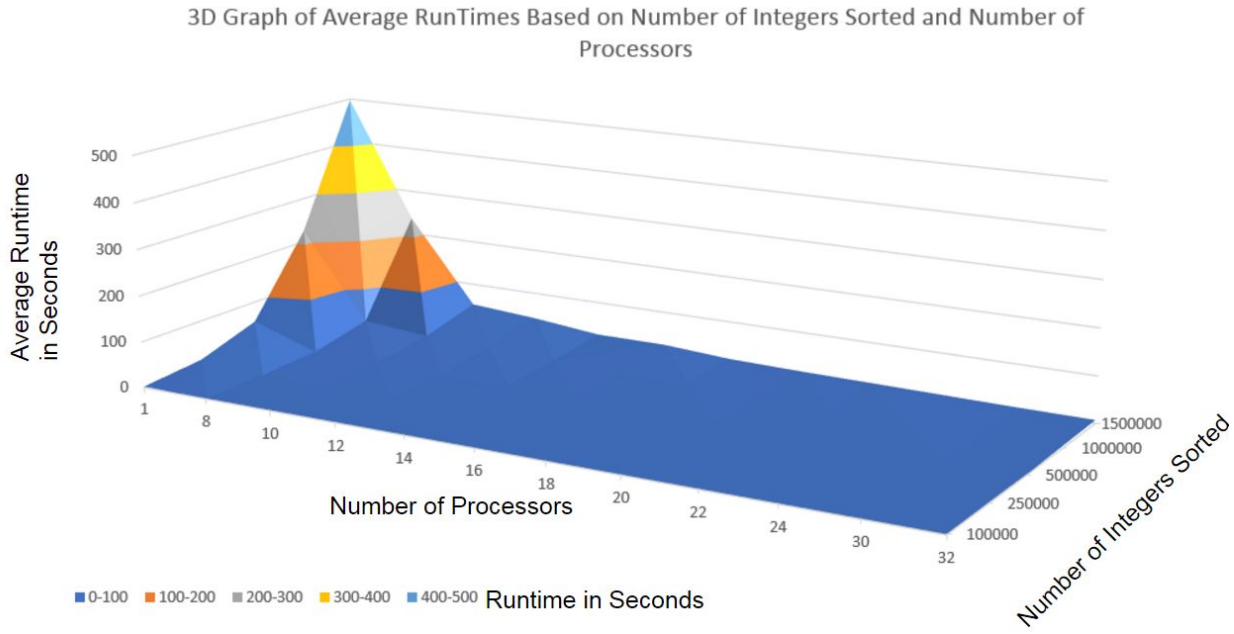

Figure 3. 3D graph that represent the Run Times based on number of integers sorted and number of Cores used. As more number of integers grows there is more slow down. Having 1 Core is slower in all integer amounts being around 4 times slower which is seen in Table 2.

Part 3: Speed Up Analysis **Important Section** Continued

Now that we have proved that bucket sort sorts very slowly with large amount of numbers, we can explain Table 2 and Fig. 4 more clearly. Both of these images show that generally as there are more processors there is more speed up. Since this parallel bucket sort algorithm splits up the number of integers by the number of processors there will be N/P work for each processor. For example I used 1.5 million integers for my biggest sort and I use 32 as my max processors.

If we divide 1.5 million integers by 32 processors each processor gets around 50000 integers to sort (N/P = 50000).

Using Table 3 we know that 50000 integers takes around 5 seconds to sort per processor. Since all the sorting takes place in parallel all 1.5 million numbers will be sorted in about 5 seconds due to the 32 processors each working on 50000 integers each. **The reason again why this is so fast is due to the fact that bubble sort is exponential thus when we break up how much work each processor has to do their speed increases exponentially.** Due to the explanation above this is the why the Speedup graph, Fig. 4 makes sense logically. I also I want to point out that that the speed up dips a little after every 8 cores specifically after the 14-18 core range.

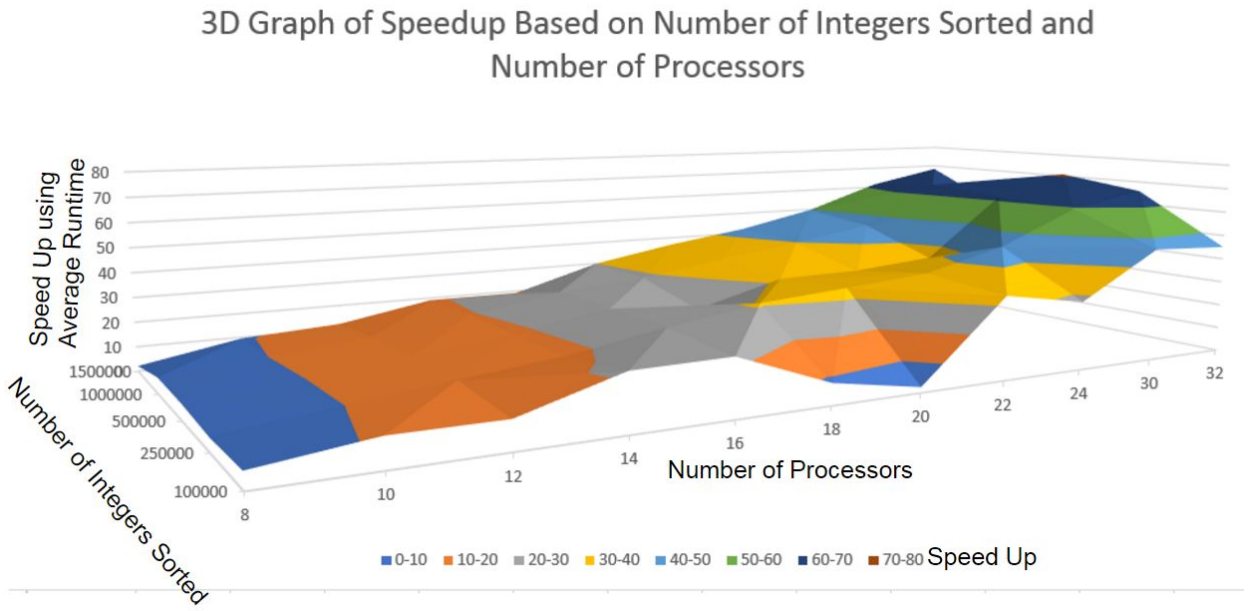

Figure 4. 3D graph that represent the Run Times based on number of integers sorted and number of Cores used. The speed up increases as there are more processors.

Part 4: Efficiency Analysis

Based on speed up analysis the above Fig. 3 continues the trend that the more processors increases efficiency solely due to the amount of work being lessened as more processors are added on. To add to this point, the efficiency we grow as the number of integers sorted grows as well. The number of integers sorted is probably the most important factor as we see around the 20 core area at the smallest number of integers sorted, that is the worst efficiency.

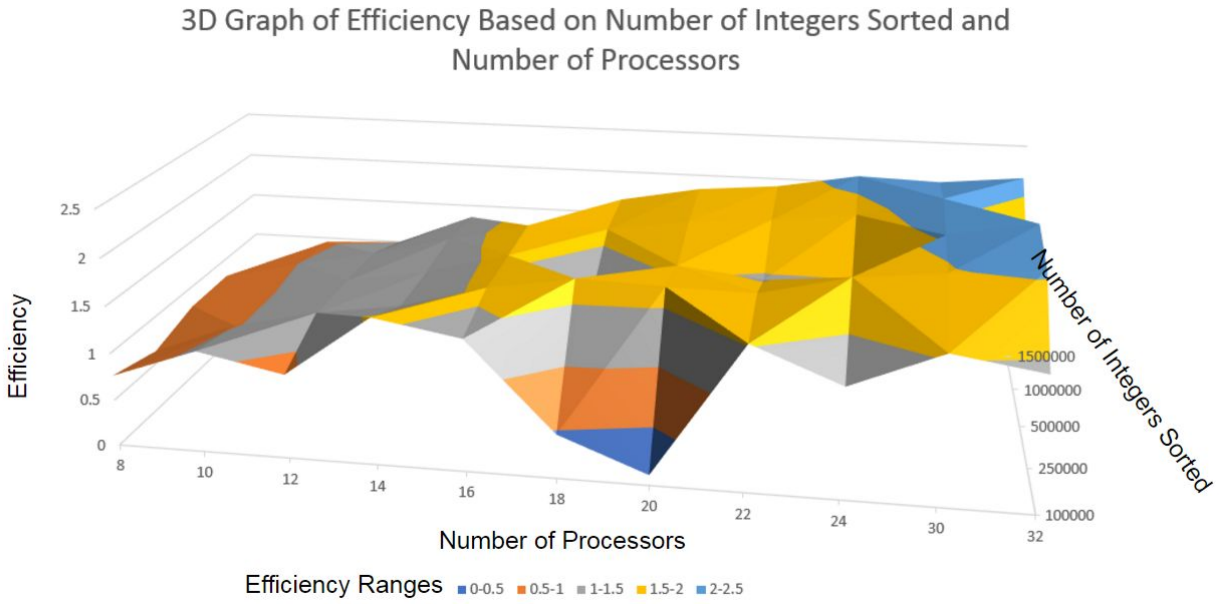

Figure 5. 3D graph that represent the Efficiency based on number of integers sorted and number of Cores used. The speed up increases as there are more processors.

Problems with Parallel

Runtimes did not make sense until further analysis. It was difficult to get more that 16 nodes at a time. Hard time understanding how to send the small sub buckets to other processors. How to print the sorted array all in the master.

Conclusion

This project teaches students about the Bucket Sort and how to parallelize the code that creates the bucket sort. Due to using bubble sort as the secondary sort there are many things that were learned/should be pointed out:

1) Bubble sort is exponentially run time thus, if you make each processor sort a thousand integers less, it will result with decrease in runtime and an increase in speed up.
2) More processors help the Bubble sort algorithm since each processor gets N/B work.
3) Other sorts will not have the speed up or efficiency of bubble sort in this case since most other sorts are around O(nLogn) which is pretty consistent even with the number of cores changing.

Future Works

Students should try this problem with multiple different types of sorts such as merge sort or quick sort. Students could also try recursive bucket sort. Students should try sorting algorithm with a Big O that is an exponent to see if the speed up is very large.