# Analysis of Parallel Merge and Bubble sort, along with Sequential Merge Sort

Dylan Bonsell

February 28, 2014

## 1 Introduction

Applying parallel computing tecniques to traditional sorting algorithms is a great was to demonstrate the usefulness of parallel programs, as well as a stable way to compute speedups. In this paper, c is chosen as the programming language, and MPI is used to pass data from each of the threads.

## 2 Analysis

### 2.1 Abstract

For this assignment, I programmed a Merge sort using MPI. The way that it works is by creating separate arrays of size $n/comm\_sz$, sending them to each process, having them sort the array, then sending it back op the tree until the main core has a sorted array.

For reference, I used qsort to get a sequential time for mergesort. In reality, the implementation of qsort is typically quicksort, but the complexity is still $O(nlgn)$, making the difference negligable.

I attempted the extra credit, but took an algorithm for Bubble sort using MPI from the web. The link is below, in my references.

All of my times are on an i7 @ 1.8GHz x 8, 4GB of ram, and a hybrid ssd.

I ran each of the tests 5 times, averaged the time, and then did my comparisons. This way, the times are an accurate representation of the algorithm itself, rather than having an outlier because of background CPU usage.

### 2.2 Compilation & Running

The following is how to compile:

```
For MPI_P_Sort.c:
mpicc MPI_P_Sort.c -o MPI

For bsort.c:
```

```
mpicc -Wall -O -o bsort bsort.c
```

*MPI_P_Sort.c* takes in size n from the argument list, such as:

```
mpirun -np $num_procs$ ./MPI $n$
```

*bsort.c* takes in 2 text files, of which the first must contain n on the first line, and n lines after containing random integers. The second file outputs said numbers in sorted order.

```
mpirun -np $num_procs$ ./bsort $infile.txt$ $outfile.txt$
```

## 2.3 Data

Below is the data for the experiment:
For n = 1,000

| | Parallel | Sequential | Bubble |
|---|---|---|---|
| np = 8 Average | .000852 | .000474 | .0002419 |
| np = 16 Average | .00112 | .000434 | .001945 |
| np = 32 Average | .003054 | .000442 | .0025524 |

For n = 10,000

| | Parallel | Sequential | Bubble |
|---|---|---|---|
| np = 8 Average | .00286 | .005574 | .0165702 |
| np = 16 Average | .002762 | .005634 | .0072716 |
| np = 32 Average | .006568 | .011642 | .0072716 |

For n = 100,000

| | Parallel | Sequential | Bubble |
|---|---|---|---|
| np = 8 | | | |
| Average | .021594 | .056232 | .7420986 |
| np = 16 | | | |
| Average | .055086 | .11183 | .4340272 |
| np = 32 | | | |
| Average | .091362 | .1717404 | .2692352 |

### 2.4 Data Analysis

Overall, it seems that bubble sort is wildly inefficient, even when parallelized. This is because it is $O(n^2)$, compared to merge sort, which is $O(nlgn)$.
At lower values of n, sequential seemed to be on par with parallel, which seems normal. The time saved by not initializing threads, communicating, and recombining is an asset to sequential algorithms, but only with small values of n. Once we hit n = 10,000, we see a huge increase in performace. The average speedup of Parallel compared to Sequential was 1.92, which is a massive increase in performace. Again, bubble sort came in last.
At values of n=100,000, things get interesting. We start to see a huge gain in bubble sort, with higher values of np, and even coming close to a sequential sort time. With parallel on the other hand, increasing np seemed to actually cause a degradation of speed, which seems interesting. At a certain point, sending the data with more cores seems to take more time then actually sorting the data within them.

## Conclusion

Parallel programs can help us achieve faster and faster run times, by allowing us to utilize hardware in a modern world.

Changes in hardware must come accompanied by changes in software, to beter utilize their true potential.

One problem, though, is the actual creation, communication, and synchronozation of parallel programs can get in the way, at certain intervals. But in general, the larger the data that has to be computed, the better parallel items become, especially when the computation is linear.

I think it would be interesting to see how dynamic programming applied with parallel programming might turn out, combining the efficiency of dynamic structures with the massive performace gains of parallel programs.

## REFERENCES

## References

[1] http://www.sci.hkbu.edu.hk/tdgc/tutorial/ExpClusterComp/sorting/bubblesort.c,

"Bubble Sort with MPI".

[2] https://docs.google.com/spreadsheet/ccc?key=0AtGVZx-NN2eVdFJ3b2dGMFBKTUtrZkpIZTdDampMX0E&usp=sharing, "Data on Google Drive".