Parallel Bubble, Merge, and Quick Sort with MPI

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

1. Introduction
   1. Background and Overview
2. Merge Sort
   1. Serial Analysis
   2. Parallel Implementation
   3. Results
3. Radix
   1. Serial Analysis
   2. Parallel Implementation
   3. Results
4. Quick Sort
   1. Serial Analysis
   2. Parallel Design and Implementation
   3. Results
5. References

**1 Introduction**

1.1 Background and Overview

The goal of the final project was to investigate an interesting application of a super-computer using the knowledge we have accumulated throughout the semester.

With a strong interest in algorithms from previous class work and with the realization that all prestigious companies that hire software engineers use algorithm based interviewing, our group set out to dive deeper into our favorite three sorting algorithms: Bubble, Radix, and Quick sort. We chose Bubble and Merge for it’s simplicity to us as we understood those the best while in our algorithms course, and Quick because it is generally considered to be the fastest sorting algorithm. To be efficient with the actual goal of the lab (to parallelize these algorithms) we adapted our designs from serial versions provided from [GeeksforGeeks](https://www.geeksforgeeks.org/). Additional resources from the web allowed us to gain an understanding for how these algorithms might work in parallel and at times provided us our “ah ha” moments. All references are cited at the end of this document.

**2 Bubble Sort**

2.1 Serial Analysis

Bubble sort is an extremely basic sorting algorithm that work under one piece of logic. If given an array, bubble sort will go through the entire array, looking at every element. It looks at two elements at a time, element n and n+1. If element n is greater than element n+1, it swaps them. It continues until the array index size -1. If bubble sort were to continue until size, there would be an array out of bounds exception. In addition, bubble sort will then make a second pass, if there are no elements to be swapped in this second pass then it will terminate and return the sorted array. This makes bubble sort a really poor sorting algorithm, the average time complexity is O(n^2) meaning that is in most cases it bubble sort has to traverse then entire array twice.

2.2 Parallel Implementation

In order for the serial implementation of bubble sort to work, all one needs to give the function is the array to be sorted and the number of elements to sort. Not a lot of work needs to be done in setup for the algorithm. This make turning the bubble sort parallel not much work. First, MPI needs to be initialized. Next, rank 0 gathers the number of elements that need to be sorted, which is stored in n, and given to the program, and the program creates an array of random elements. From there, rank 0 broadcasts the number of elements, n, to each rank. Each rank needs to know how many elements they need to sort, so the number of elements is determined based off if n is evenly divisible into the number of ranks. If they are, then the size is equal to n/numranks. Otherwise the size is equal to n/numranks+1. Next, a the data array is then scattered into a newly formed temp array, sub, which has the size of the number of elements each rank needs to compute. Armed with their piece of the work, bubble Sort is then called using the temp array, sub, as the first parameter and the number of elements each rank needs to compute, or size. Once each rank finishes its sorting, rank 0 then gathers each ranks chunk of the work. From there, if the number of ranks is greater than one, bubble Sort is called one more additional time on the entire newly gathered dataset. This is to sort all of the pieces that have been gathered. Once that last bubble sort is complete, the array is sorted.

2.3 Results

As bubble sort is a notoriously bad sorting algorithm in serial, using it in parallel was not any better. In order to have a sorted array at the end of the program, bubble sort needed to be called on the gathered dataset. Odds are this ran through the entire length of the data twice, making sure there were no mistakes. Due to the thoroughness of this algorithm, the programs’ speed and input capabilities suffered. In serial, the maximum input size for the program was around one hundred thousand. Other programs which have been implemented have been able to reach a million. Was able to handle more, due to the help each additional rank offered. Figure 1 shows the strong scaling plot which includes the average timings for each of the ranks. Each average is taken out of five sample times to ensure accuracy.

**Figure 1.**

The first jump from one to two ranks is fairly normal, the time is almost split in half when another rank takes half of the work. From there, there is not much speed up occurs. A very good speedup would have more curve to the graph and not level off so quickly. Unfortunately due to how bubble sort works in serial, more speedup could not be achieved.

**3 Merge Sort**

3.1 Serial Analysis

3.2 Parallel Implementation

3.3 Results

**4 Quick Sort**

4.1 Serial Analysis

Quick Sort is a Divide and Conquer algorithm much like Merge Sort. Broken down simply, the algorithm operates by picking an element as a “pivot” and compares the current looked at element to that pivot. If that element is smaller/higher than the pivot then swap the elements until all elements are in their correct sorted order. What this achieves is a linear time for sorting which is why in serial, Quick Sort is generally considered to be the fastest sorting algorithm for large enough data sets.

4.2 Parallel Design and Implementation

The main driver of the serial code for quicksort is the partition section which relies heavily on picking the pivot element. An interesting piece of information that was discovered what that in general, it’s least efficient to pick a pivot by picking an element somewhere in the middle of the array. In parallel however, it is not only the correct way to pick the pivot but it also in theory could be the most efficient.

The goal of this project was to implement the above methods in parallel to be run on the super computer using multiple nodes/ranks. In order to achieve this goal, MPI was utilized using a variety of methods. For example, in class we had assignments that would only use a send/receive method like the ring and ping-pong algorithms, while other assignments like the matrix-vector product would use a scatter call. Another method that was used was broadcasting a number to all ranks in order for the main driver of the program to operate. This algorithm uses all of these methods in order to achieve its goal.

A basic outline of the algorithm can be broken down with MPI calls as the main focus:

1. Broadcast size of original array
   1. All ranks needs this in order to calculate sub array size.
2. Scatter original array data evenly to all ranks
3. Calculate appropriate sub size and quicksort each respective rank’s sub array
4. Merge back sub arrays on each “step” by sending the lower half sub arrays to the lower ranks and the higher half sub arrays to the higher ranks. Do this up to log (numranks) times.

The below flowchart illustrates the above method.

A close up of a map

Description automatically generated

4.3 Results

The following graphs show the results of the algorithm. In order to no go over the page limit too much some general take away points was that the algorithm speed up time from serial wasn’t that intense and as you increase ranks/nodes while you do see an increase it is also not that intense. That being said, there is a noticeable trend of speed up for sorting 1 million and 10 million integers.

The above figure shows a graph that illustrates the times taken for the program to run on a variety of nodes/ranks on 5 different runs for each node/rank combination with an input size of 1 million integers

The above figure shows the times taken to sort 10 million integers on 5 runs using ranks/nodes 1-10. A key take away from this graph shows that as the number of integers increase on input, the more closely aligned the data points are.

**5 References**

# References

GeeksforGeeks. *Bubble Sort*. 02 02 2014. Web. 07 08 2020.

—. *Quick Sort*. 07 01 2014. Web. 07 08 2020.

—. *Radix Sort*. 09 02 2013. Web. 07 08 2020.

Karleigh Moore, 刚 王, Jimin Khim. "Bubble sort." 1 Jan 2014. *Brilliant.* Web. 10 August 2020.