Parallel Optimization of Quicksort and Merge Sort

CS 442/542

Jeremy Middleman

Andrei Popa-Simil

December 11th, 2023

1 Introduction

Sorting is intrinsic to various applications in computer science, from database management to data analysis and scientific computing. As data sizes increase, finding efficient sorting algorithms becomes increasingly important and challenging. One consideration when choosing a sorting algorithm is what resources are available. Certain algorithms, such as mergesort, can be parallelized on CPUs and accelerated further on GPUs. We aim to use multiple CPUs and GPUs to parallelize and improve the performance of two commonly-used sorting algorithms: Merge Sort and Quicksort.

2 Data Collection Techniques

Data collection is vital for identifying differences between algorithms so we ran all of our algorithms through a tester file that would generate the arrays to be sorted, timed the sorting and validated each run. We ran our tests on the Xena cluster provided by UNM's Center for Advanced Research Computing (CARC). This super-computer was chosen because it contains many GPU nodes which may run one of our implementations. To standardize our tests we made and followed several standards. First, all arrays to be sorted are of type float. Second, we tested on arrays of size 2^0 to 2^{20}. We collected a minimum, maximum, and average time for every test and only when a test was valid. We validated tests after sorting by checking if the resultant list is in ascending order. We ran 10 and 1000 tests per array size for Quicksort and Merge Sort respectively, to get accurate measurements. Additionally, for any MPI implementations we ran on 1, 2, 4, 8, 16, 32, and 64 processors. Xena's singleGPU partition has 16 CPUs per node which means our 32 and 64 processor tests were using 2 and 4 nodes respectively. Xena's singleGPU partition has 61G of memory per node as the upper limit for high memory processes. With the above testing standards we collected plenty of analyzable data. All code is in our github repo *[1].*

Algorithm 2.1 Tester Pseudocode:

```
Get command line arguments
MPI_Init
Initialize timer and arrays.
For (array size = 2^0 to 2^20{
        Randomize array
        Initialize stats
        For (runs=0 to number_of_runs){
                MPI_Barrier
                Start Timer
                Sort Array with Selected Method
                End Timer
                if( Validation of Run){
                        Update Stats}
```

```
            Reset Stats}
        Output results
    Cleanup Memory
    MPI_Finalize
```

3 Quicksort

Quicksort is a divide-and-conquer sorting algorithm that efficiently sorts an unsorted array of numbers by recursively partitioning it into smaller subarrays based on a chosen pivot element. Quicksort has a time complexity of $O(nlog(n))$ for the best and average cases and $O(n^2)$ for the worst case. In the average case, Quicksort takes $O(n)$ time to sort the subarray about the pivot, and $O(log(n))$ time to recursively run Quicksort on the left and right halves of the array. We hypothesize that using MPI and multiple processors will improve the performance of Quicksort as the number of processes increases. We predict that this effect will taper as the number of processes increases and the parallelizable code approaches optimization.

3.1 Naive MPI Implementation

An existing serial implementation of the Quicksort algorithm was used (pseudocode shown in appendix Algorithm 8.1).*[2]* We also made and tested a naive version of Quicksort on multiple processors. The algorithm in Algorithm 4.1 works by calling MPI_Scatter on the array and then running Quicksort on the resulting subarrays found in each processor. The subarrays are then gathered using an MPI_Gather call. The serial merge function merges the subarray in a process with the array containing all previous subarrays. Similar to the theoretical serial algorithm of Quicksort, this sorting mechanism should give us a time complexity of $O(nlog(n))$. In reality, our implementation only performs in $O(nlog(n))$ in the best case and performs in $O(n^2)$ in the average and worst cases (Figure 8.1.1). Squaring the number of processes resulted in an $O(log(n))$ improvement in performance (Figure 3.1). However, the number of unsuccessful sorts doubled as the number of processes doubled. Additionally, not all processes returned any successful sorts for certain larger array sizes (Figure 8.1.2).

Algorithm 3.1: Naive MPI Quicksort Pseudocode:

```
nrunQuicksort(array, size){
    Scatter the array on the root process to subarrays in the n processors
    Quicksort on buffer array
    Gather the subarrays on the processes to the original array in the root process
    For(i =1 to log(num_processes)){
        Step = size of buffer * i
        For( j = 0 to array - 2 * step){
```

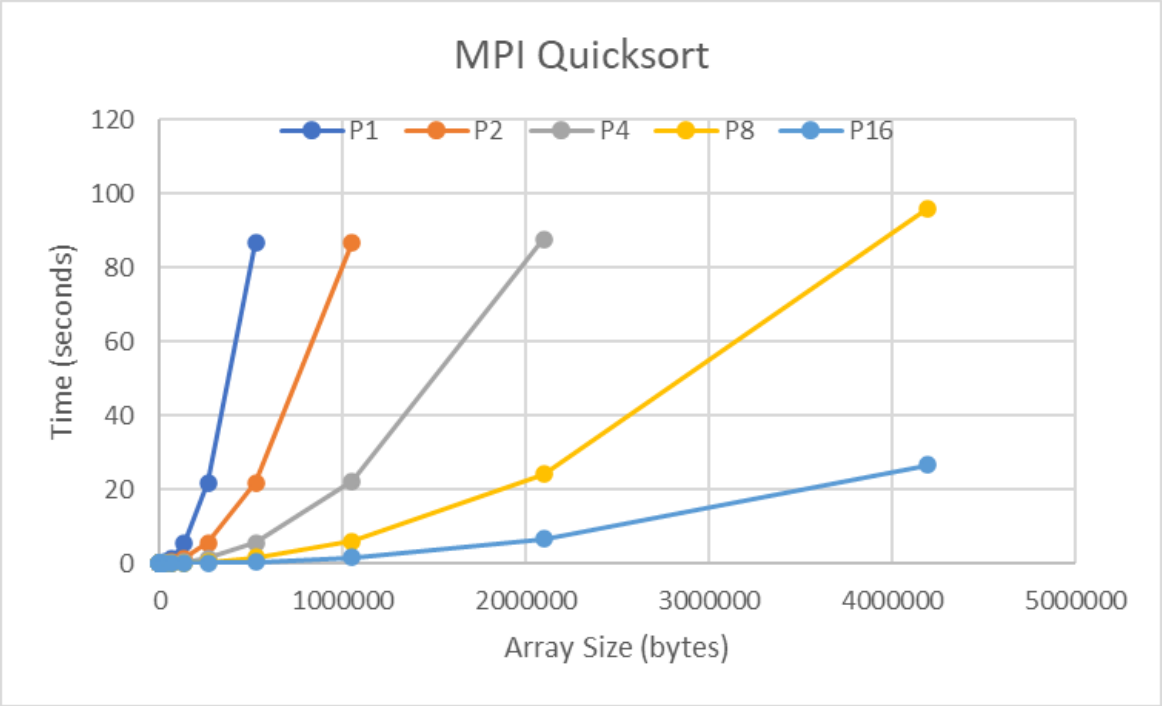

Figure 3.1: MPI improves performance of the Quicksort algorithm. Performance analysis is shown for 1 (dark blue), 2 (orange), 4 (grey), 8 (yellow), and 16 (light blue) processors. Tests with one, two, and four processors failed to fully sort arrays larger than 2^{17}, 2^{18}, and 2^{19}, respectively, within one hour.

3.2 Optimized MPI Implementation

An optimized algorithm was incompletely implemented. This algorithm, visualized abstractly in figure 3.2, should perform Quicksort without needing a merge function at the end, as the array in the i+1th process may be appended to the array in the ith process to produce a combined sorted subarray. At this point, each element is sorted both within its subarray and process such that the first half of processes have smaller elements and the second half have larger elements.

Each process starts with a chunk of the original array of equivalent size. A pivot is selected at random from the subarray on the root process and that pivot is then communicated to each other process. Each process divides its unsorted subarray into two subarrays: one containing elements smaller than or equal to the pivot and the other containing elements greater than the pivot. Processes in the upper half of the processes send the smaller subarrays to the lower half and receive the larger subarrays from the lower processes. This pattern repeats by recursion until the upper-half processes exclusively hold values greater than the pivot, while the lower-half processes exclusively hold values smaller than the pivot. After it is no longer to halve the number of processes, serial Quicksort is run on the remaining subarray and the subarrays are gathered back into the original array.

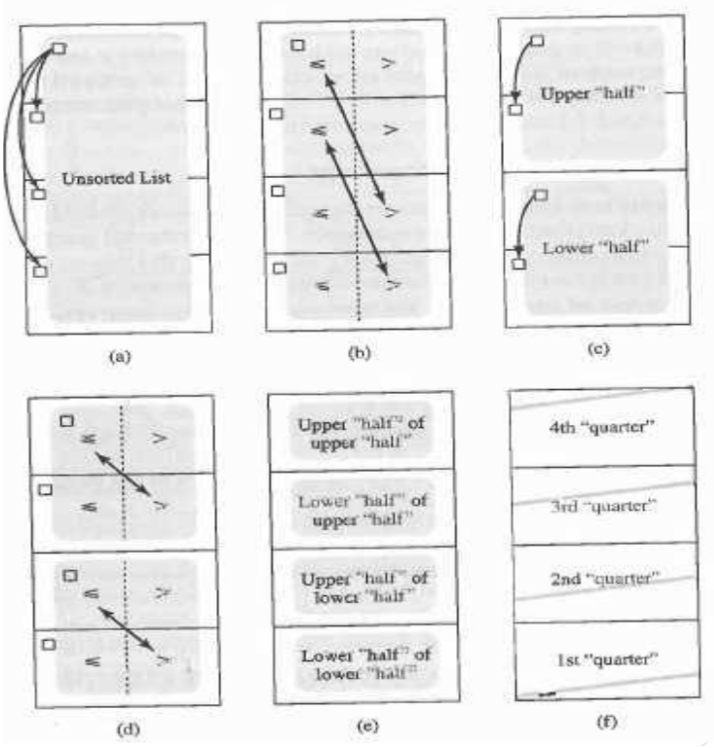

Figure 3.2: Abstract Depiction of a More Optimized Parallel Quicksort Algorithm. [3]

4 Merge Sort

Merge Sort is one of the most basic and well taught algorithms for sorting an array. In general, Merge Sort is an optimal algorithm that has a time complexity of $O(nlog(n))$for best, worst, and average cases. Merge Sort takes $O(log(n))$ time to divide the array into subarrays, and $O(n)$ time to merge it back together.

4.1 Hypothesis

As the biggest time complexity for Merge Sort comes from merging all the subarrays together, we focus our approach on the merging process and testing various implementations to see if we can reduce the merge time. We plan to utilize the fact that two subarrays being merged are already in sorted order. We hope to show that using MPI and multiple processors will parallelize Merge Sort consistently as more processors are added to sort. Additionally, we plan to test an implementation on the GPU that uses n^2 memory, but merges two lists in constant time if there are n^2 running threads. Additionally we would like to see if there are any optimizations that can be done to this GPU implementation to speed it up and optimize it for the case of fewer threads.

4.2 Serial/Classical Implementation

The classical serial code implementation that we used was described above. $O(nlog(n))$ best, worst and average solve time complexity, generated by $O(log(n))$ calls to merge which is an $O(n)$ operation. The pseudo code below is how our implementation of serial Merge Sort works. Figure 4.2.1 shows how an array is split into subarrays and then sorted.

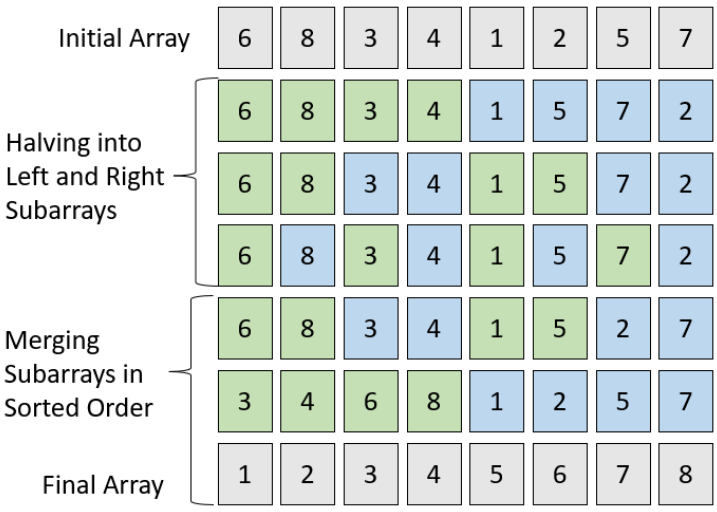

Figure 4.2.1: Classical Merge Sort splitting array into left (green) and right (blue) subarrays followed by sorting merges.

The following algorithm was provided by Geeks for Geeks *[4]*.

Algorithm 4.1: Classic Merge Sort Pseudocode:

```
Merge(array, left, middle, right){
        Initialize a left and right array based on the left, middle, and right indices.
        For( length of array){
                If( left < right)(
                        Place item from left into array}
                Else{
                        Place item from right into array}}

MergeSort(array, left, right)
        if( left < right){
                Find middle
                Call MergeSort(array, left, middle)
                Call MergeSort(array, middle+1, right)
                Call Merge(array, left, middle, right)}}
```

The following graph, figure 4.2.2, shows the minimum, average, and maximum times to sort an array. As the graphic cannott show the average as it is very close to the minimum. Additionally, in figure 4.2.3 we can see that the maximum, average and minimum time complexity is the same as our loglog plot shows about the same slope for each data set.

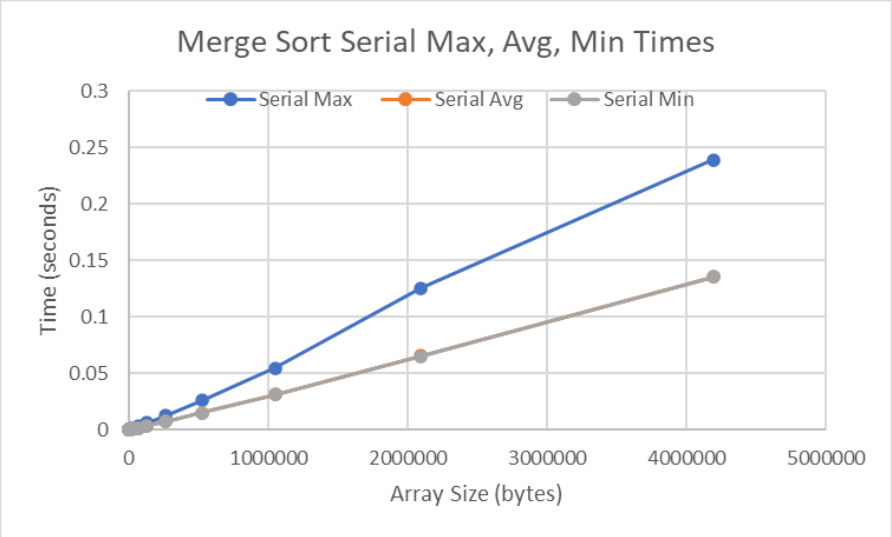

Figure 4.2.2: Maximum, Average, and Minimum test times of serial implementation of Merge Sort

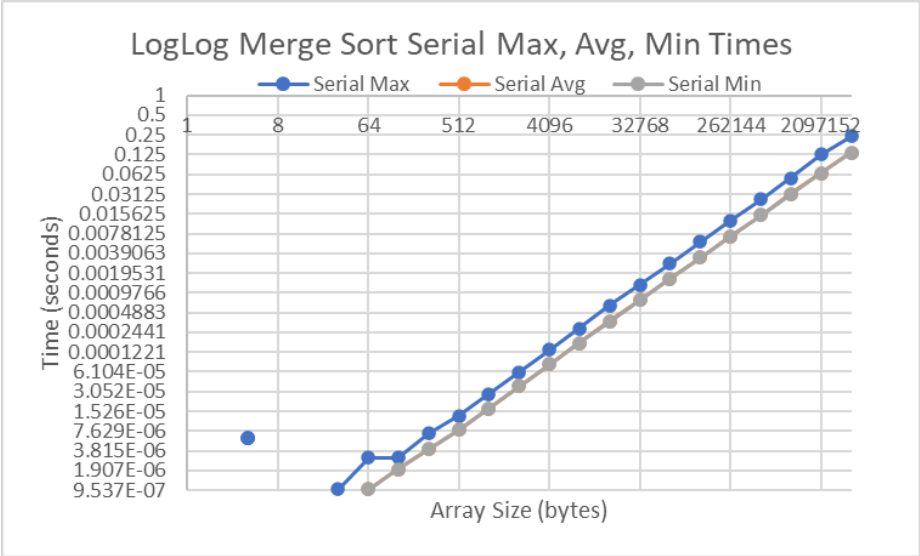

Figure 4.2.3: Loglog plot of serial Merge Sort with maximum, average, and minimum.

4.3 MPI Implementation

We also made and tested a naive version of Merge Sort on multiple processors.This algorithm works by calling MPI_Scatter on the array and then running Merge Sort on each processor for their portion of the array The array are then brought together with an MPI_Gather call and then merged $log(num\ processors)$ amount of times to stitch the sections together. This sorting mechanism should still give us a time complexity of $O(nlog(n))$ however we have reduced the constant by num_processsors, resulting in $O(nlog(n)/num\ processors)$. Below is figure 4.3.1 showing how the implementation of mergesort is parallelized. Additionally, the pseudocode to implement such mergesort is thereafter.

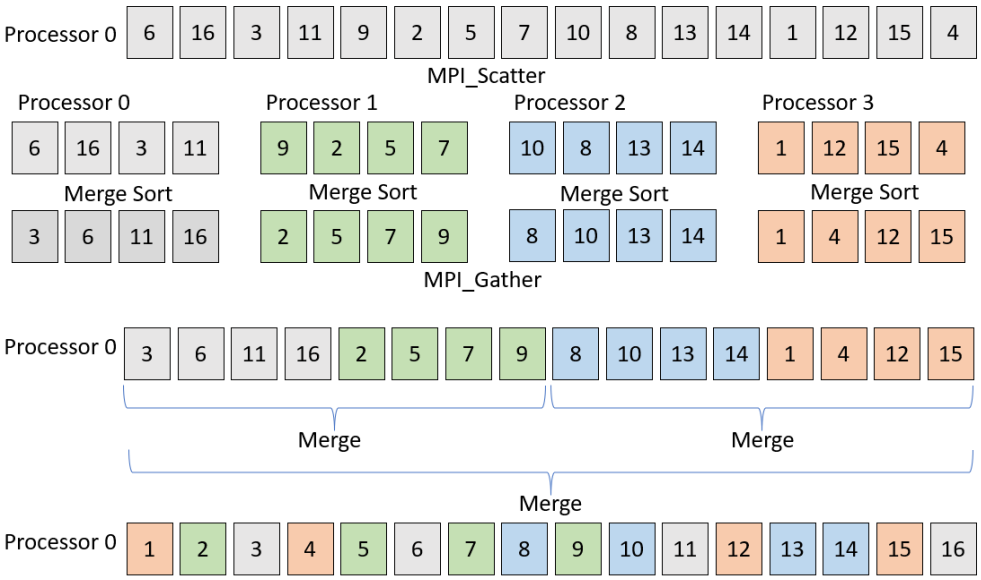

Figure 4.3.1: Implementation of Parallelized Merge Sort with 4 processors.

Algorithm 4.2: Naive MPI Merge Sort Pseudocode:

```
MPI_Mergesort(array, size){
        Initialize MPI variables rank and size
        Initialize a buffer array to hold MPI message data
        MPI_Scatter from root process 0 from array to buffer
        Mergesort on buffer array
        MPI_Gather to root process 0 from buffer to array
        For( i =1 to log(num_processes)){
                Step = size of buffer * i
                For( j = 0 to array - 2 * step){
                        merge(array, j, j + step, j + 2 * step)}}}
```

This implementation is very short and sweet. It can be optimized a bit more by using binomial tree messaging to stitch together the smaller sorted arrays. This implementation has very minimal communication between processors and is almost embarrassingly parallel as only the last few merges aren't done equally on all processors. We would expect more processors to be less and less effective at improving performance of the sort as communication is increased with more processors and as processor 0 has to do more and more merging at the end of the sort. In figure 4.3.2 we can see just how parallelizable this implementation actually is. Take note at how using eight processors (P8 - yellow) appears to be optimal as it is consistently the fastest sorting time. Additionally, there appears to be a spike in times for eight or more processors.

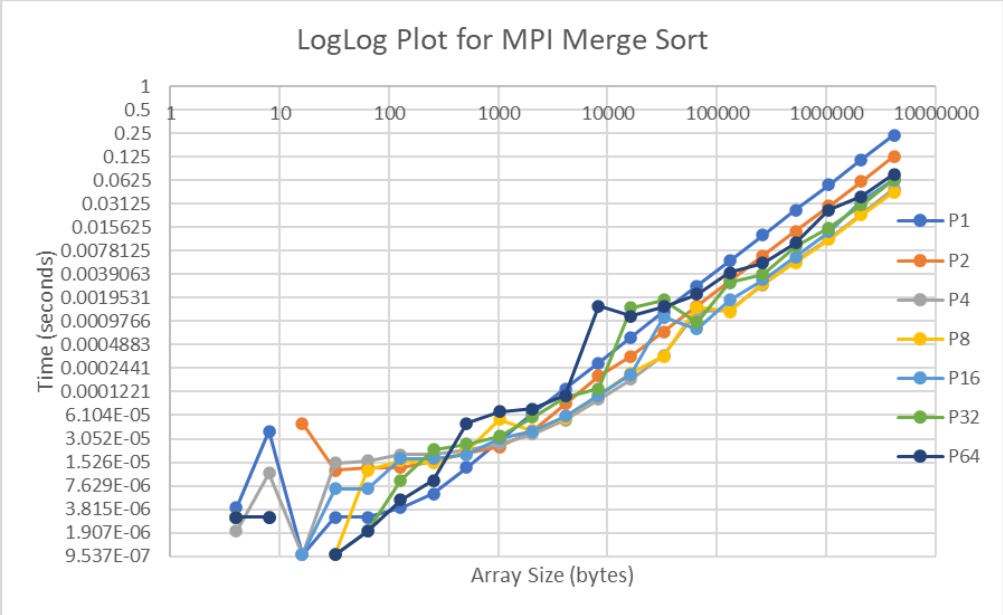

Figure 4.3.2: Loglog plot for MPI Merge Sort using 1, 2, 4, 8, 16, 32 and 64 processes.

4.4 GPU Implementation

The implementation used for GPU is a personal solution to an exam problem that asked “design an algorithm that uses infinite processors to merge two sorted lists in constant time. This solution uses $O(n^2)$ memory to achieve a time complexity of $O(1)$ when there are n^2 processors. This is possible because the merging algorithm makes a grid of n by m size where n is the length of the left subarray and m is the length of the right subarray. Since all the test cases are powers of 2 the left and right subarrays will be the same size.
The process proceeds as follows:

Step 1: With the created grid, each cell compares the value of its x and y index to the same index in the left and right subarray respectively. The comparison will result in a 1 or a 0 in each cell This is represented in figure 4.4.1.

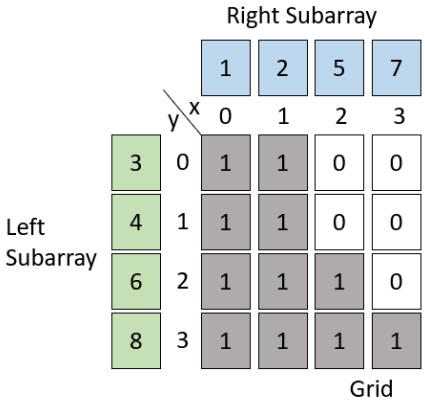

Figure 4.4.1: First step of matrix merge that places 1s and 0s in the grid.

Step 2 is to find each instance of a 1 and check if the cell above is a zero. If this is the case, place the represented item from the right array into its position in a buffer array based on the x and y indexes. This check places all of the right array into the sorted position in the buffer. The formula is shown in the pseudocode below and the visual representation is shown in figure 4.4.2.

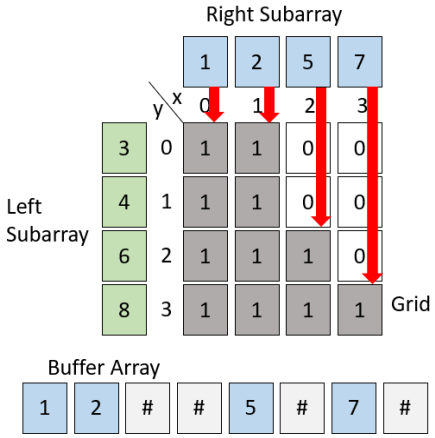

Figure 4.4.2: Second step, identifying the positions of elements in the right subarray.

This will be repeated but for the cell to the right to place the entire left array into the sorted position. Afterwards this buffer array will be returned as the sorted array. The formula will be shown in the following pseudocode additionally the visual representation is shown in figure 4.4.3.

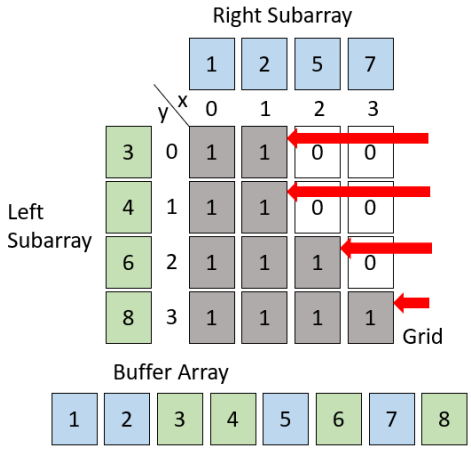

Figure 4.4.3: Third step, identifying the positions of elements in the left subarray.

Steps 2 and 3 can be run simultaneously when a 1 is hit in the grid scan. Additionally, edge cases when columns and rows are all 1s or 0s a 1 or change is assumed giving the index of searched for point.

The pseudocode in Algorithm 4.3 represents the serial implementation of this algorithm without optimizations and the combined index search steps.

As one can tell at a glance, this code is far more complicated and resource intensive than the previous two pseudocodes. Figure 4.4.4 shows the loglog plot comparing the matrix, serial, and MPI with one processor Merge Sort algorithms. The matrix merge is $O(n^2)$ while the other two are $O(nlog(n))$ and as such the matrix merge should not be used on a CPU implementation. This implementation of Merge Sort is made to validate the solution implementation in terms of sorting correctness.

Algorithm 4.3: Serial Matrix Merge Sort Pseudocode:

```
MatrixMerge(array, left, middle, right){
        Initialize variables, grid, and buffer
        //Step 1
        //Comparisons
        For(row = 0 to size of left array){
                For (col = 0 to size of right array){
                        if( left[row] >right[col]){
                                Grid[row][col] = 1}
                        Else{ Grid [row][col] = 0}}}

        //Step 2 and 3
        //Row wise and Column wise index finding.
        //Edge cases are ignored in pseudocode :D
        For(row = 0 to size of left array){
                For (col = 0 to size of right array){
                        if(Grid[row][col] == 1){
                                If(Grid[row-1][col]==0){
                                        buffer[row+col]=Right[col]}
                                If(Grid[row][col+1]==0){
                                        buffer[row+col+1]=Left[row]}}}}

        Copy elements from buffer into array
        Cleanup allocated buffer and grid}

MatrixMergeSort(array, size){
        Initialize variables
        //Apply a binomial tree to split the matrix and merge properly
        For(step = 1 to log(size)){
                For(start = 0 to size - 2 * step){
                        MatrixMerge(array, start, start+step, start+ 2 * step)}}}
```

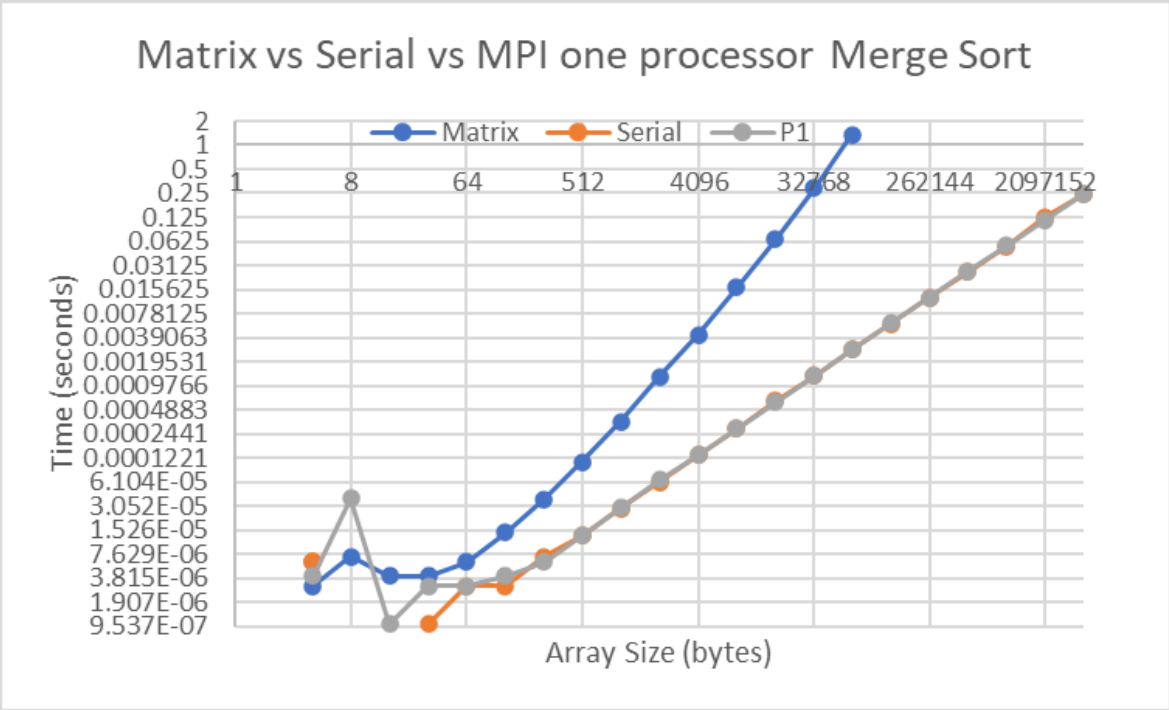

Figure 4.4.4: Loglog plot of Matrix, Serial, and MPI with one processor Merge Sort.

5 Conclusions and Future Work

For Quicksort, the naive MPI Quicksort exhibited comparable performance to the serial algorithm when employed on a single process, but demonstrated increasing efficiency as the number of processes scaled. The observed time complexity was closer to $O(n^2)$ for both algorithms, which underscores the inherent limitations of the serial implementation that was also used for the naive MPI version. The failure of the parallel quicksort algorithm to function with 32 and 64 processes raises questions about the algorithm's scalability. Additionally, the occurrence of unsuccessful sorts was consistently a maximum of half of the number of processors used in testing. These are issues that require attention.

For Merge Sort, the three implementations functioned mostly as predicted. The MPI parallelized implementation had a surprising optimal CPU count of 8, even though nodes on the Xena singleGPU partition have 16 CPUs per node. The MPI implementation also suffers from a lack of parallelization on its last set of merges which could be improved with a binomial tree merge to better use the additional CPUs. Upon testing the serial implementation with the MPI implementation with one processor, we saw no noticeable difference. It is possible that this may be caused by the tester file itself. In the serial and MPI implementations the average appeared to stay closer or on top of the minimum, while in the matrix implementation the average was perfectly balanced between the minimum and maximum. For the future, many things must be done to improve the quality of testing and the efficiency of the algorithms. Making a more strict validator and tester would ensure all tests work every time. Most importantly, a GPU version of the matrix sort should be implemented and can itself be optimized and error bound, as the implementation can work for non-square arrays as well. Possible optimizations for a GPU

version include blocking which will allow for less memory usage and the sorting of very large arrays. Other well known Merge Sort implementations should be tested on the GPU to compare performance such as Bitonic Merge Sort and Odd Even Merge Sort. This study is a preliminary research into optimization techniques applicable to sorting methods and alternative sorting methods for parallel machines.

6 Acknowledgements

We would like to thank the UNM Center for Advanced Research Computing, supported in part by the National Science Foundation, for providing the high performance computing resources used in this work.
We would like to thank Dr.Shaun Luan from UNM Computer Science Department, for providing puzzling exam questions which motivated this project.
Additionally we would like to thank ChatGPT for providing coding assistance throughout this project.

7 References

[1] Jeremy Middleman and Andrei Popa-Simil. cs442Sorting github repo, https://github.com/apsUNM/cs442sorting

[2] GeeksforGeeks. (2023, October 16). *Quicksort - data structure and algorithm tutorials*. GeeksforGeeks. https://www.geeksforgeeks.org/quick-sort/

[3] Ramkumar, D. R. (2014). *Implementation of parallel quick sort using MPI.* https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ramkumar-Spring-2014-CSE633.pdf

[4] GeeksforGeeks. (2023, November 28). *Merge sort - data structure and algorithms tutorials*. GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/#

8 Appendix

8.1 Quicksort

Algorithm 8.1: Serial Quicksort Pseudocode:

```
partition(array, idx1, idx2)
        Select the array[idx2] as the pivot
        Iterate through the array.
                The front of the array will hold the values larger than the pivot.
                If the current element is less than the pivot, move the element to the front
                section of the array.

sQuicksort(array, size){
        Base case: size = 1
        Recursive case:
                Find the pivot and call partition to partition the array
                Run sQuicksort on the left and right halves of the current array
```

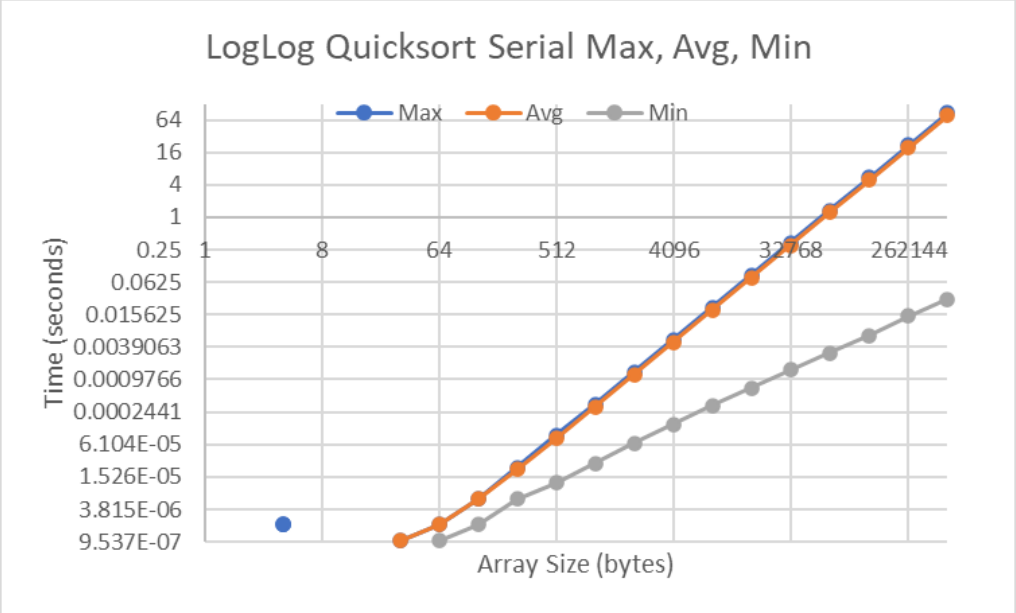

Figure 8.1.1: This implementation of Quicksort achieves a performance of $O(nlog(n))$ in the best case, and $O(n^2)$ in the average and worst cases. Successful sorts lasting less than 9.537E-07 are not visualized.

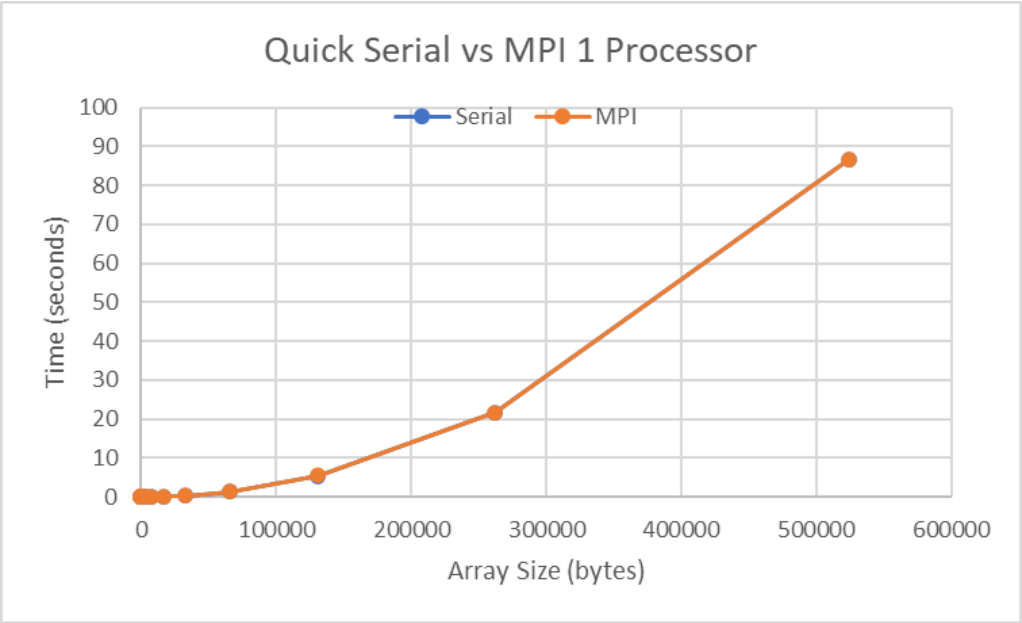

Figure 8.1.2: Serial Quicksort has identical performance to parallel quicksort on a single process. Serial and parallel MPI. Shown are the performances of the serial implementation (blue) and the parallel implementation (orange), run on a single processor.

8.2 Merge Sort

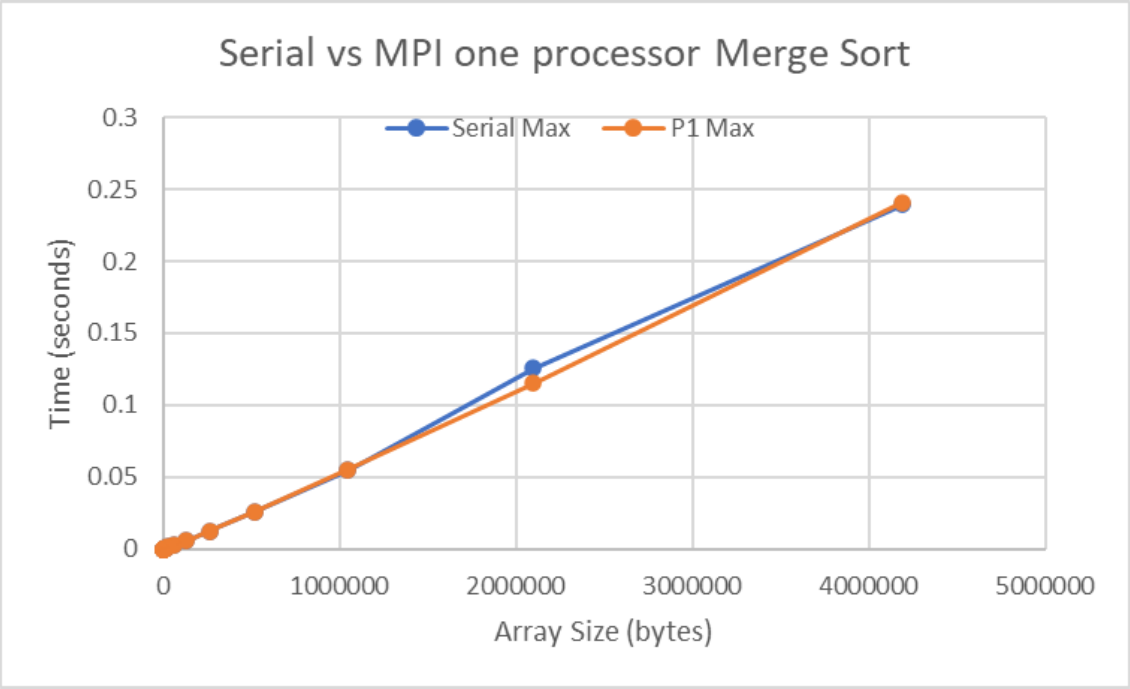

Figure 8.2.1: Serial vs MPI one processor merge sort

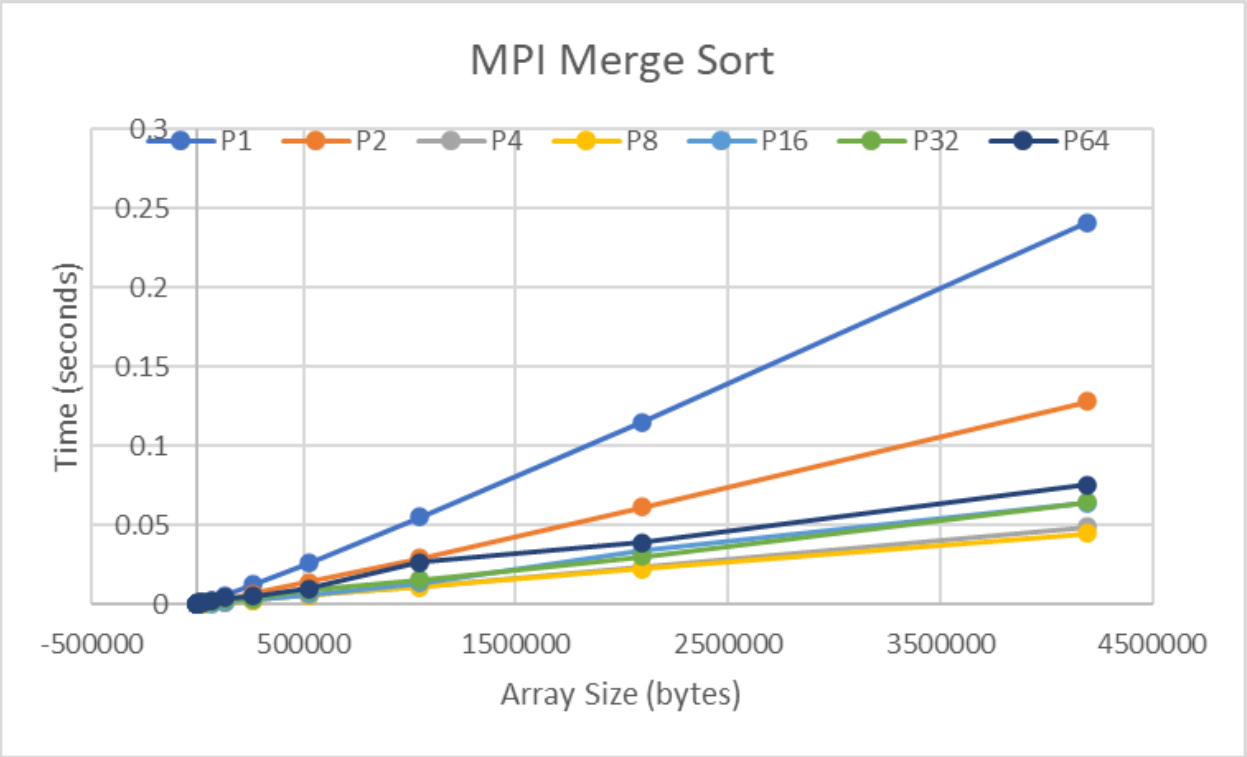

Figure 8.2.2: MPI Merge Sort with 1, 2, 4, 8, 16, 32, and 64 processors.

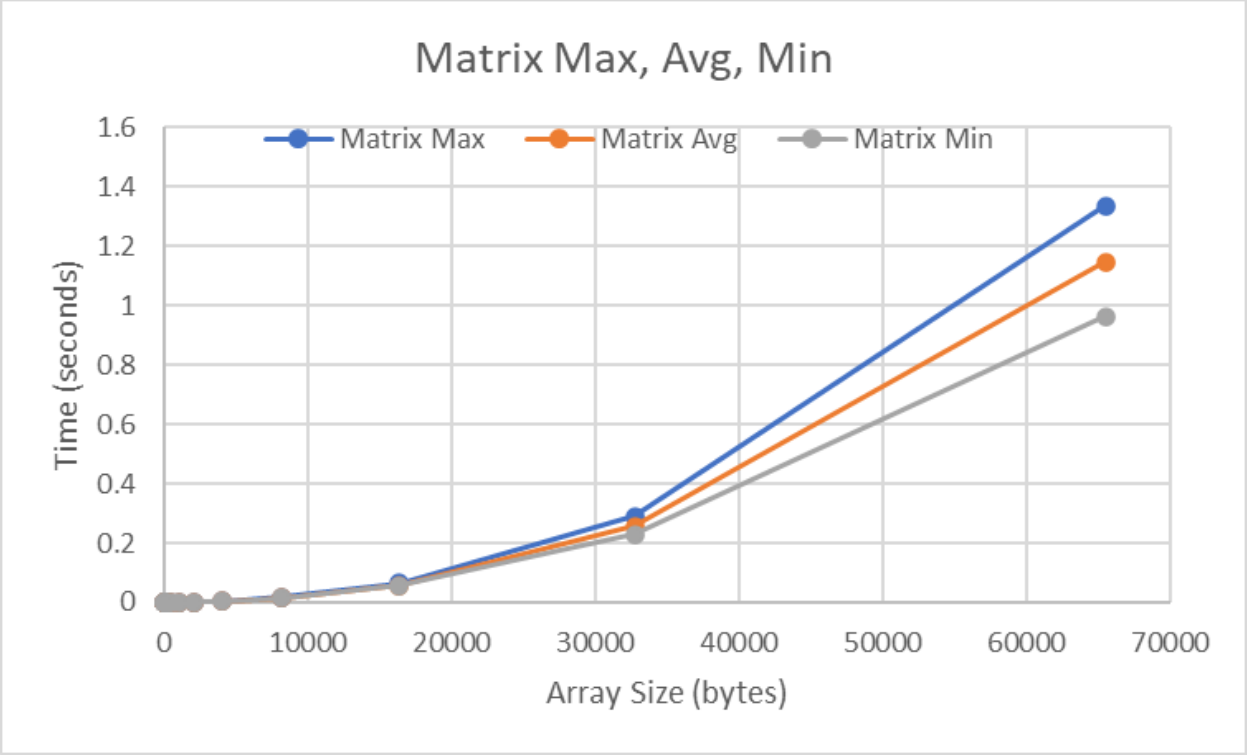

Figure 8.2.3: Serial Matrix Merge Sort with Maximums, Averages, and Minimums.

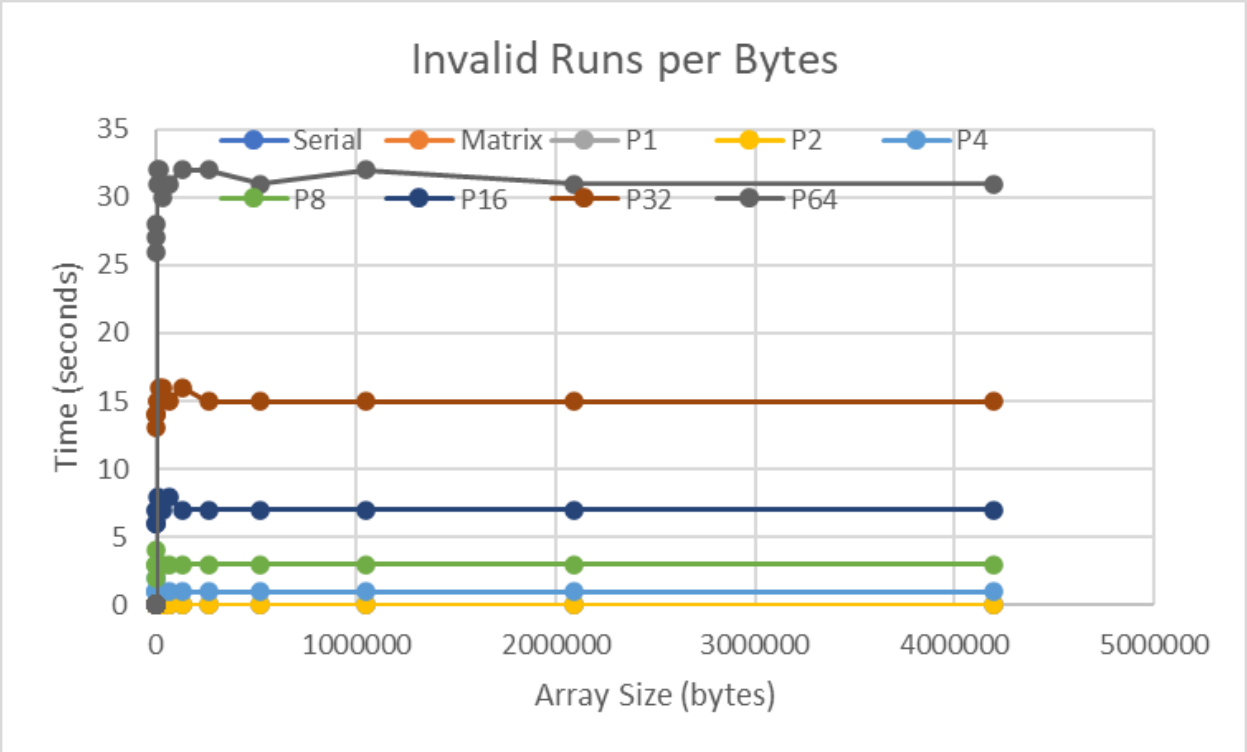

Figure 8.2.4: Invalid Runs Anomaly with tester file. Consistently, failed $num\ processes/2$ tests, regardless of number of iterations.

8.3 Quicksort vs Merge Sort

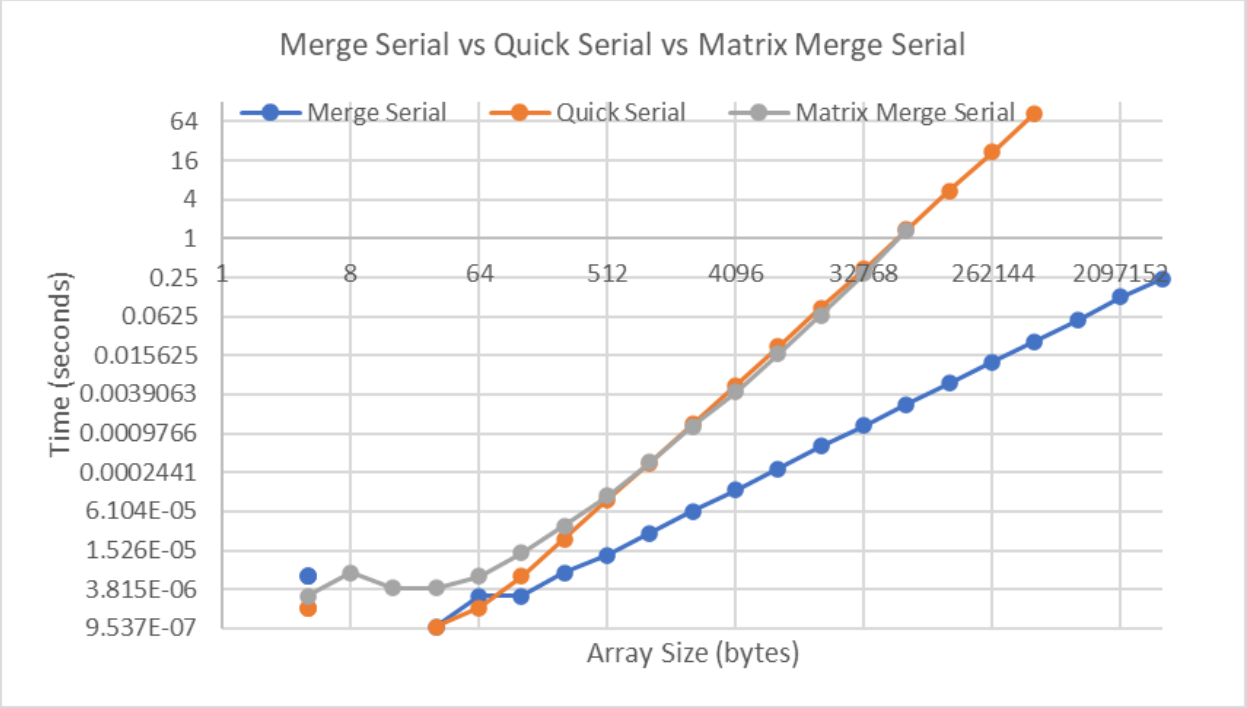

Figure 8.3.1: Loglog Quicksort Serial vs Merge Sort Classical Serial vs Matrix Merge Sort