CS 4504 Project Part 2
Spring 2024

Gavin Frey/01, Cameron Lowry/01, Shaun Teague/01, Tanner Velzy/01, John Sheffield/01, Joss Hufnagel/01

Abstract

Our task is to send an array of unorganized numbers from the client side to a server using the same network from part 1 of the project. The server will then organize the array of numbers with multiple tasks before displaying it for the user. To achieve this, we decided to use a merge sort algorithm so that we could utilize multiple tasks. Each task will act as a part of the merge sort and eventually merge back into a root thread/core that will output the result.

When making our code, we used Java threads to simulate having multiple threads with each getting a piece of the array and doing its own merge sort on the piece. The thread combines its piece with other threads to do merge sort, eventually ending up within the root thread, which outputs the result. The time, efficiency, and speed up are calculated after the algorithm is done running, giving us our data.

In our findings, we saw that the run times got worse the larger the array. With fewer threads, the speed up was worse at larger array sizes, while higher thread counts proved to be marginally beneficial as the array size increased (i.e. with lower efficiency but better speedup nonetheless) but experienced diminishing returns at the lower array sizes. From that, we concluded that while more tasks being used does decrease the runtime and helps with speed up on larger arrays, you lose out on efficiency due to all that work being divided up.

Introduction

In this project, we use the merge sort algorithm with multithreading in the server program to sort arrays of numbers from the client program that come in sizes of 10000, 100000, 200000, and 300000. We found merge sort to be the most efficient sorting algorithm, especially for the larger inputs we were using. We designed the code to divide up the data inputted with tasks that would continue until the data is ready to be sorted. Each thread would complete its task and hand it back up to the parent until the entire sorted array is pieced back together.

Our code has three parts: the mergesort, mergeSortTask, and server. The mergesort is the merge sort algorithm that would sort its given arrays and return the sorted result. This mergesort code would be used in the mergeSortTask, where each task or thread needed would go through to return its sorted part of the array. Finally, the server would get the array from the client before using the previous two code segments to find the run time, speed up, and efficiency of each run.

When running the code, we generate randomized numbers in arrays of sizes 10000, 100000, 200000, and 300000. The code runs through each array a hundred times to gather enough data to give an average for run time, speed up, and efficiency. It will then output tables for each type of data and display the thread counts with their appropriate averages.

In the data we gathered, we found that the run time, speed up, and efficiency were affected significantly by the amount of threads used and the array size they had to handle. For the most part, the more threads there were, the better they did on larger array sizes, while the single thread runs did better on smaller arrays.

Program design is focused around several different paradigms with two of the most important being efficiency and speed. This is crucial when looking at sorting algorithms that want to work as quickly as possible without sacrificing the computer's ability to run multiple different tasks. This report will be a look into testing and examining how different numbers of threads affect both speed-up and efficiency during a merge sort for different array sizes.

Design Architecture

In order to sort numbers quickly and efficiently, our team decided to implement a form of merge sort. Merge sort is one of many algorithms designed for sorting numbers. The reason it is so effective is because of its divide-and-conquer methodology. Before sorting even begins, it divides the data into two sublists. It then divides those sublists into two more sublists each. The algorithm will continue to do this until each sublist cannot be divided up anymore (when each sublist is reduced to one element). Once this happens, the merging portion of the merge sort begins. Each of these divided sublists are merged together one by one. This process of merging is when the data is sorted. Each new set of larger sublists are all in the correct order. Once these merges are completed, the process is repeated until the ordered sublists are combined into the full list of sorted numbers.

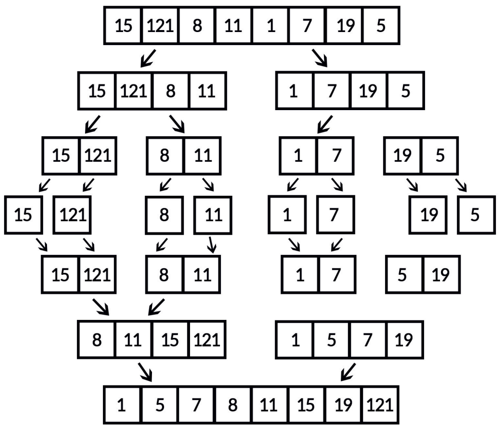


Fig 1. Merge Sort Example

The image above is an example of merge sort in practice. We can see that the original list of 8 numbers is out of order. When we first apply merge sort to this list, the data is divided into two sublists of 4 numbers each. These two sublists are then divided further down into 4 sublists with a length of 2. Once we have divided up this data, we can start combining it back together. Each time the lists are combined, they are put into the correct order. This process continues until the original list is restored with the numbers arranged in the correct order.

As stated before, the reason why our team picked this sorting algorithm is due to its performance. If we scale our performance using Big-O notation, we can see that other sorting algorithms such as bubble and selection sort have a time complexity of (n^2), where n represents the length of the list entered into the algorithm. This means that these algorithms take exponentially longer to perform when more data is present. However, merge sort has a time complexity of (n*log(n)). This may not seem important at first, but the difference in performance can be shown visually through a Big-O Complexity chart.

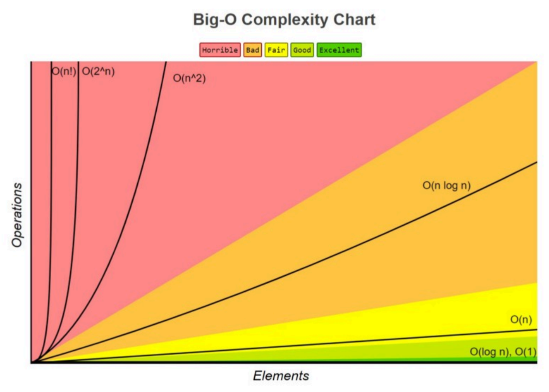


Fig 2. Big-O Complexity Chart

The chart above shows the difference in time complexity by graphing the operations in relation to the number of elements being processed. The (n^2) line is the third line in the red zone and the (n*log(n)) line is located in the orange zone. If we look at how each of these lines grow over time, we can see that the n^2 line grows at a much quicker rate than the n*log(n) one. This graph shows the performance increases over certain other sorting algorithms and why we chose this one over others such as bubble and selection sort. These performance gains become extremely important as datasets grow exponentially large, such as the arrays we sorted this project.

In order to design this algorithm with threading implemented, several things had to be taken into account. The most important of these was how we would divide up our data in the most even and efficient way possible.

To do this, we first manually set the number of threads we wanted to use. Once that value is initialized, we set the depth of the mergesort to the rounded-down log base 2 of the given threads count. This is so that when we get to the bottom level, there will be the same amount of sublists for each thread specified. The merge sort algorithm begins by splitting the data using recursion. Each time the data is split, child processes are created for each split. When the bottom level is reached, the number of threads specified will be running. Once each thread has sorted its corresponding sublist, it is returned to its parent thread's list where both its child lists are merged. This process continues until the full list is sorted.

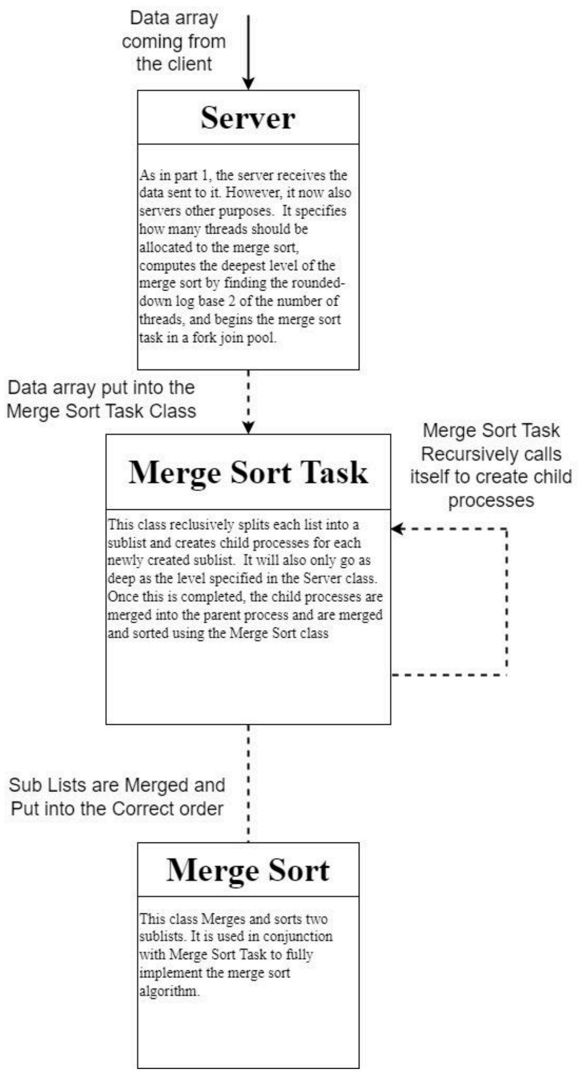
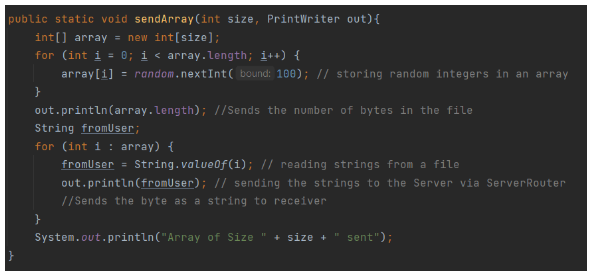

Data array coming from the client
Server
As in part 1, the server receives the data sent to it. However, it now also servers other purposes. It specifies how many threads should be allocated to the merge sort, computes the deepest level of the merge sort by finding the rounded-down log base 2 of the number of threads, and begins the merge sort task in a fork join pool.
Data array put into the Merge Sort Task Class
Merge Sort Task
Merge Sort Task Recursively calls itself to create child processes
This class reclusively splits each list into a sublist and creates child processes for each newly created sublist. It will also only go as deep as the level specified in the Server class. Once this is completed, the child processes are merged into the parent process and are merged and sorted using the Merge Sort class
Sub Lists are Merged and Put into the Correct order
Merge Sort
This class Merges and sorts two sublists. It is used in conjunction with Merge Sort Task to fully implement the merge sort algorithm.

Fig 3. Design Diagram

This can be seen in the design diagram above. The server class is sent the data array that will be used during the merge sort. In the class, it also specifies how many threads there will be as well as finding the deepest level of the mergesort. Once this has been completed, the Merge Sort Task class is called. Each time it splits the data, it recursively calls itself to create the child processes. These Merge Sort Tasks are then combined and put in order using the Merge Sort class. By dividing up the individual processes between these classes, it allows for the implementation of Mergesort to be clear and organized.

Implementation

Once a connection between the client and the server is made, the client starts transmitting the 4 arrays of sizes 10000, 100000, 200000, and 300000.

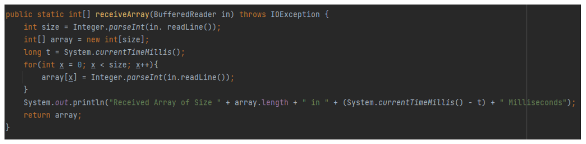


```
public static void sendArray(int size, PrintWriter out){
    int[] array = new int[size];
    for (int i = 0; i < array.length; i++) {
        array[i] = random.nextInt( bound: 100); // storing random integers in an array
    }
    out.println(array.length); //Sends the number of bytes in the file
    String fromUser;
    for (int i : array) {
        fromUser = String.valueOf(i); // reading strings from a file
        out.println(fromUser); // sending the strings to the Server via ServerRouter
        //Sends the byte as a string to receiver
    }
    System.out.println("Array of Size " + size + " sent");
}
```

Fig 4. sendArray code snippet

It accomplishes this by calling the sendArray method with the desired size of the array to send and the socket's printWriter. The sendArray method starts by making an array of the desired size and filling it with random values from 0 - 99 inclusively. Then the method sends the length of the array to the server so it knows how many messages to expect. The method finishes by sending each of the entries to the server.

On the server side, the 4 original arrays are received by calling the receiveArray method 4 times.

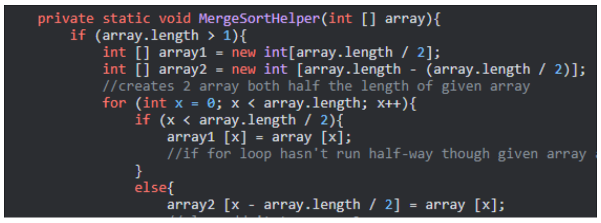


```
public static int[] receiveArray(BufferedReader in) throws IOException {
    int size = Integer.parseInt(in. readLine());
    int[] array = new int[size];
    long t = System.currentTimeMillis();
    for(int x = 0; x < size; x++){
        array[x] = Integer.parseInt(in.readLine());
    }
    System.out.println("Received Array of Size " + array.length + " in " + (System.currentTimeMillis() - t) + " Milliseconds");
    return array;
}
```

Fig 5. receiveArray code snippet

The recieveArray starts by creating the array of the right size by receiving the first message from the client which contains the size of the array that's going to be sent. The method then takes the current time in milliseconds. Finally the method receives the entries in the array one by one until they have all been received by the client. The method finally ends by displaying the size of the received array and the time it took in milliseconds before returning the int array.

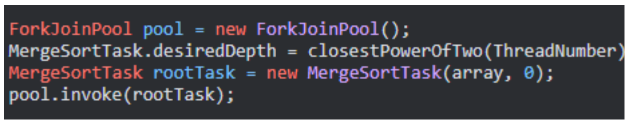
To begin implementing the merge sort into our code, we had a merge sort algorithm made that would be used in other parts of our code.



```
private static void MergeSortHelper(int [] array){
    if (array.length > 1){
        int [] array1 = new int[array.length / 2];
        int [] array2 = new int [array.length - (array.length / 2)];
        //creates 2 array both half the length of given array
        for (int x = 0; x < array.length; x++){
            if (x < array.length / 2){
                array1 [x] = array [x];
                //if for loop hasn't run half-way though given array
            }
            else{
                array2 [x - array.length / 2] = array [x];
```

Fig 6. Merge Sort code snippet

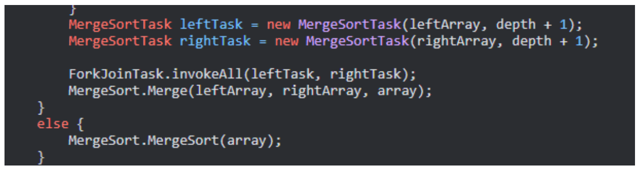
In figure 6 is the main workhorse of the merge sort code. It takes in an array, splits it, and puts the correct half of the integers in the two arrays. This method will continue splitting the array until it reaches a point where it cannot be split any further. At this point, the algorithm will remerge the now-sorted arrays together back into a parent array. With the merge sort code ready, we now had to help the server easily use the code by creating a task for it.



```
ForkJoinPool pool = new ForkJoinPool();
MergeSortTask.desiredDepth = closestPowerOfTwo(ThreadNumber)
MergeSortTask rootTask = new MergeSortTask(array, 0);
pool.invoke(rootTask);
```

Fig 7. TCPServer code snippet

We start by creating the thread pool that will be used to implement our program as seen in figure 5. In this project, we decided to use the Java ForkJoinPool class. The ForkJoinPool class is a child of the ExecutorService class. This is important since the overhead from creating a traditional class in Java is massive and had a major negative impact on the performance of our algorithm. After creating the ForkJoinPool, we then set the static integer variable called desiredDepth to the output of the closetPowerOfTwo method when given the current number of threads. We did this because, with the way we implemented our algorithm, the number of active threads must be 2 to a power of n. We then need to create a rootTask with the array we want to sort and a given depth of 0. After all of this is done, we call the invoke method which tells the main method to sleep until the root task is done.



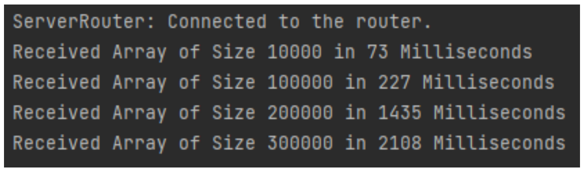
```
        }
        MergeSortTask leftTask = new MergeSortTask(leftArray, depth + 1);
        MergeSortTask rightTask = new MergeSortTask(rightArray, depth + 1);

        ForkJoinTask.invokeAll(leftTask, rightTask);
        MergeSort.Merge(leftArray, rightArray, array);
    }
    else {
        MergeSort.MergeSort(array);
    }
```

Fig 8. MergeTask code snippet

Our task starts with an if statement that compares its current depth to the desired depth. If the current depth is greater than or equal to the desired depth, the task simply calls the MergeSort method with the array assigned to it. Otherwise, the task takes its assigned array and splits it into 2 arrays, a left array and right array. The task then creates 2 child tasks, one with the left array and one with the right array. Each of the children are given a depth 1 higher then the parent task's depth. The parent task then calls the invokeAll method with the 2 child tasks, causing the parent task to sleep until the 2 child tasks have both completed running. Once the 2 child tasks are done, the parent task will call the Merge method from our MergeSort class in order to combine the 2 now-sorted left and right arrays into the parent task's array, resulting in the parent task's array becoming sorted.

Simulation Method

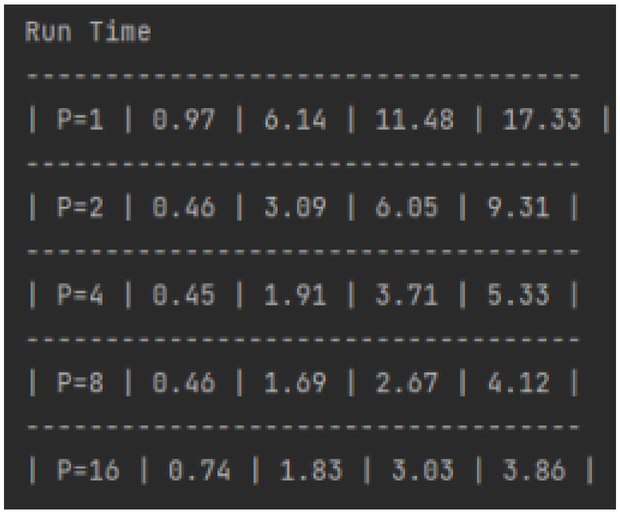
In this simulation, we decided we would test thread counts of 1, 2 ,4, 8, and 16. The simulation first created 4 Int arrays of different sizes (10000, 100000, 200000, 300000), filling them with random numbers. The arrays would be created and filled with random arrays on a simulated client process that would later be sent to the server to be tested on.



```
ServerRouter: Connected to the router.
Received Array of Size 10000 in 73 Milliseconds
Received Array of Size 100000 in 227 Milliseconds
Received Array of Size 200000 in 1435 Milliseconds
Received Array of Size 300000 in 2108 Milliseconds
```

Fig 9. Received Array Output

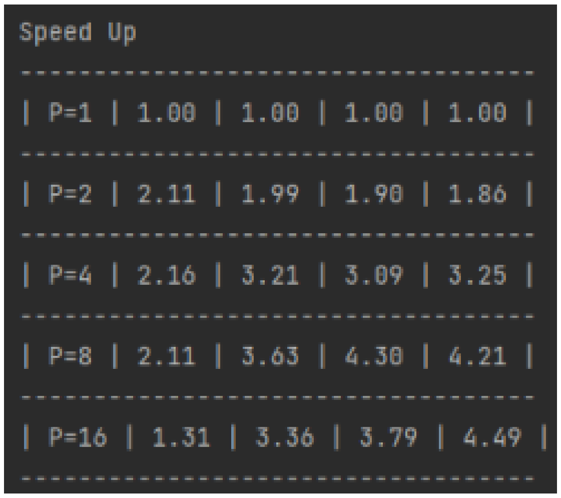
In figure 9, you can see that once an array has been completely received by the client, the size of the received array and the time it took for it to be sent is printed out for the user to see. The simulation runs through each array size 100 times on 1 thread then takes the average sort time for each array size. Afterwards, the simulation moves on to 2 threads and performs the same action of sorting each array size 100 times and taking the average. This continues until all thread counts have been run and their Run Times have been obtained. The Speed Up is then calculated by taking each number of threads' average Run Time and dividing it by the Run Time of 1 thread and outputting the result. To calculate efficiency, the program takes the Speed Up and divides it by the number of threads for each.



```
Run Time
-----------------------------------
| P=1 | 0.97 | 6.14 | 11.48 | 17.33 |
-----------------------------------
| P=2 | 0.46 | 3.09 | 6.05 | 9.31 |
-----------------------------------
| P=4 | 0.45 | 1.91 | 3.71 | 5.33 |
-----------------------------------
| P=8 | 0.46 | 1.69 | 2.67 | 4.12 |
-----------------------------------
| P=16 | 0.74 | 1.83 | 3.03 | 3.86 |
```

Fig 10. Run Time Output

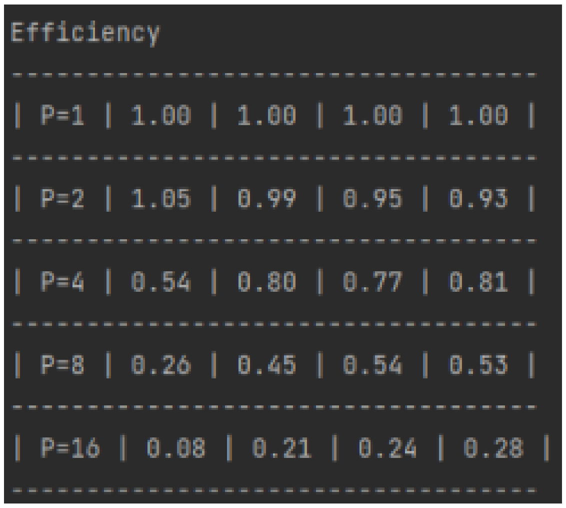
In figure 10, the average runtimes of the tests in milliseconds are displayed to the user via an output table. We got this data by using the System.currentTimeMillis method before and after the test is ran then storing the difference in a local array. Once all 100 of the tests have been ran, we take the average of all times in the array and store it in a global array. We then move on to the next set of tests. Once all the tests have been run, we display the obtained runtime value in the global array to the user in a table format as shown above.



```
Speed Up
-----------------------------------
| P=1 | 1.00 | 1.00 | 1.00 | 1.00 |
-----------------------------------
| P=2 | 2.11 | 1.99 | 1.90 | 1.86 |
-----------------------------------
| P=4 | 2.16 | 3.21 | 3.09 | 3.25 |
-----------------------------------
| P=8 | 2.11 | 3.63 | 4.30 | 4.21 |
-----------------------------------
| P=16 | 1.31 | 3.36 | 3.79 | 4.49 |
-----------------------------------
```

Fig 11. Speed Up Output

In figure 11, the speedup for each run is displayed to the user in a table-like format similar to the run times after being calculated by the program. Speedup is calculated by the program by taking the serial (p=1) runtime for each array size and dividing it by the runtime for the current amount of threads.



```
Efficiency
-----------------------------------
| P=1 | 1.00 | 1.00 | 1.00 | 1.00 |
-----------------------------------
| P=2 | 1.05 | 0.99 | 0.95 | 0.93 |
-----------------------------------
| P=4 | 0.54 | 0.80 | 0.77 | 0.81 |
-----------------------------------
| P=8 | 0.26 | 0.45 | 0.54 | 0.53 |
-----------------------------------
| P=16 | 0.08 | 0.21 | 0.24 | 0.28 |
-----------------------------------
```

Fig 12. Efficiency Output

In figure 12, the program outputs efficiency in a third table. Efficiency is calculated by the program by dividing the speedup by the number of threads used in each test. The efficiency statistic is useful in seeing what the optimal amount of threads to use is for each array size.

Data Discussion

Size (numbers)	Time (Milliseconds)	Average Transmission Speed
10,000	73	136.99
100,000	227	440.52
200,000	1435	139.37
300,000	2108	142.31

Table 1. Transmission Time Table

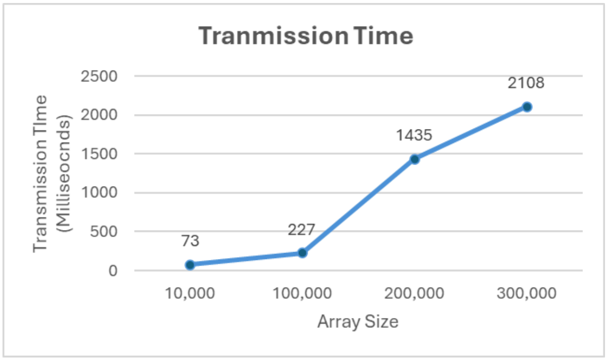


Fig 13. Transmission Time Graph

In Table 1 and figure 13, you can see the time it took to transmit arrays of varying sizes in milliseconds. By looking at the data, you can see a general trend that for the most part as the array size grows the time it takes to transmit also grows proportionally. The number of entries per millisecond stayed around 140 entries per millisecond for each array size except for array size 100,000, which had an average of 440 entries per second.

In our testing, we found that the run times got longer the fewer threads and bigger array sizes there were. In addition, the speed up was larger as the number of threads and the array size increased, and the efficiency got worse as the number of threads increased and the array size decreased.

No of Threads/ cores	10^4	10^5	$2*10^5$	$3*10^5$
p = 1	.97	6.14	11.48	17.33
p = 2	.46	3.09	6.05	9.31
p = 4	.45	1.91	3.71	5.33
p = 8	.46	1.69	2.67	4.12
p = 16	.74	1.83	3.03	3.86

Table 2. Run Time Table

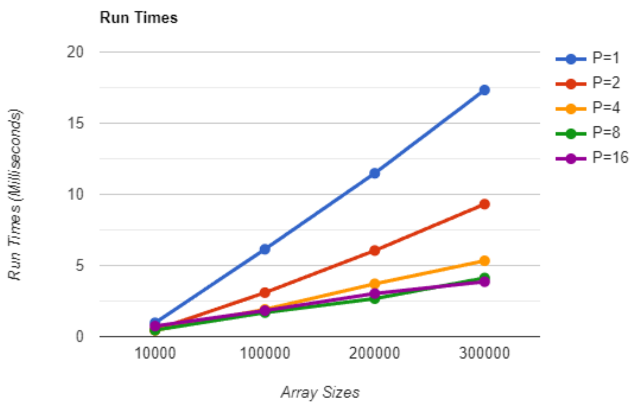


Fig 14. Run Time graph

In table 2 and figure 14, you can see that all thread counts begin at similar times for the lowest array size (10000), but the time it takes for thread 1 to sort higher array sizes quickly grows. This is in contrast to higher thread counts such as 8 and 16 rising at a much slower rate. For the single thread runs, only one thread at a time is working on the array, so the proportionally higher run times make sense with the multi-thread runs being able to divide up the work. This makes sorting larger sample sizes quicker for larger thread counts and smaller sample sizes more or less similar in execution time. However, this does not explain why 8 threads and 16 threads are close together in terms of their run-time. This may be because with more tasks working on a relatively small array, it overcomplicates that process and results in the code taking more time as more tasks than are needed are running together. This is supported by the fact that in the higher array sizes, 16 threads does better than 8 threads, which begin to take more time on those larger array sizes.

No of Threads/ cores	10^4	10^5	$2*10^5$	$3*10^5$
p = 1	1	1	1	1
p = 2	2.11	1.99	1.90	1.86
p = 4	2.16	3.21	3.09	3.25
p = 8	2.11	3.63	4.3	4.21
p = 16	1.31	3.36	3.79	4.49

Table 3. Speed up table

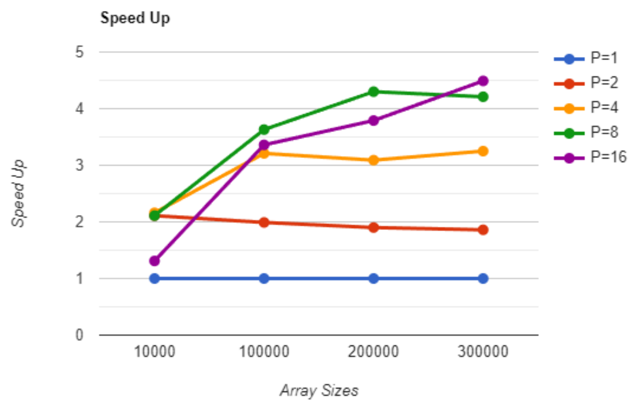


Fig 15. Speed up graph

In table 3 and figure 15, runs with 8 threads rise quickly but get beat out by 16 threads at the largest array size. Other thread counts stay at relatively constant and differing speed ups with 1 thread runs being the lowest and flattest. While 16 threads would seem to be the quickest as many threads are working on the same problem, it doesn't matter that much for smaller array sizes as there is less to divide up and work on. Therefore, other multithread runs have an easier time working on the arrays up until the larger sizes, where 16 thread runs start to work faster. Similar to the run times, it seems that the 16 task runs start low with the lower task runs starting higher on average and then, as the array gets larger, the speed up for the 16 task runs begins to overtake the other runs while those runs start to dwindle.

No of Threads/ cores	10^4	10^5	$2*10^5$	$3*10^5$
p = 1	1	1	1	1
p = 2	1.05	.99	.95	.93
p = 4	.54	.80	.77	.81
p = 8	.26	.45	.54	.53
p = 16	.08	.21	.24	.28

Table 4. Efficiency table

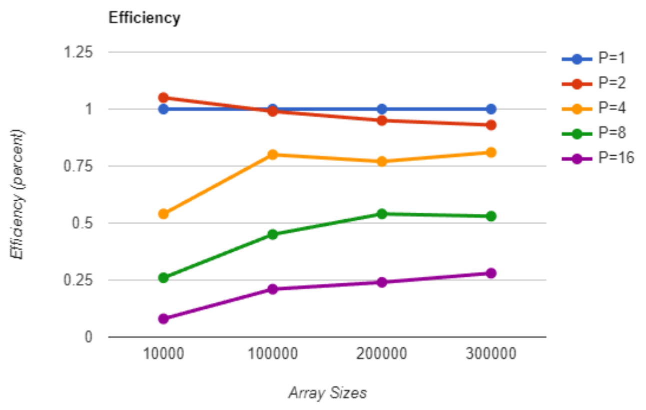


Fig 16. Efficiency graph

Finally, for table 4 and figure 16, you can see the single thread runs having the most consistent and higher efficiency. This is in contrast to higher thread counts, where efficiency tends to drop off. We found that when using 16 threads, we had the lowest amount of efficiency. With so many threads, the 16 thread runs usually have an efficiency issue, only getting better at larger array sizes where the threads can more easily divide up and work on the array. Unlike the other two tables, the 16 task runs stay consistently inefficient compared to the other runs, while the lower task runs are leagues more efficient. This is, as explained before, because with 16 tasks running at the same time, there is going to be loss of efficiency due to the fact the array must be divided up then put back together at the end whereas 1 task does all the work with no loss of data or work.

Overall, from the data we collected, we found that while more threads nets shorter run times overall, the speedup and efficiency begin to fluctuate between more and less threads. Depending on the implementer, the data can be interpreted in different ways. For a large corporation with essentially unlimited resources, using a higher amount of threads for that little bit of speedup could make more sense despite the efficiency loss. For our personal systems, which we found to be barely able to handle the 16 thread runs, higher efficiency would be more beneficial over the small amount of speedup.

Conclusion

Our goal for this project was to take a merge sort algorithm and convert it to use multiple threads. In addition, we wanted our program to automatically record the speed-up and efficiency when utilizing more threads on arrays of various sizes. We tested our algorithm when using 1, 2, 4, 8, 16 threads on arrays of size 10000, 100000, 200000, 300000 integers. We achieved this by utilizing the ForkJoinPool in Java and creating new tasks recursively until the program utilizes the number of threads we desire (as long as it is 2 to a power of n).

The program runs each test 100 times and takes the average of every test instead of running each test once. We do this to ensure our results are accurate but also more resilient to outliers. The program then uses the average times gathered to calculate the speed-up of using multiple threads and also the efficiency of using multiple threads on each of the possible array sizes (10000, 100000, 200000, 300000). The results of the runtimes show a clear trend of lower runtimes as more threads are utilized. The speed-up table shows a similar trend where typically as more threads are utilized, the speedup is greater. However, it also shows that the speed up gained by using multiple threads is typically greater on bigger array sizes. The efficiency table shows that while utilizing more threads leads to higher speed up, it also leads to much lower efficiency. Typically, the efficiency loss is lower with bigger arrays. This shows that our algorithm has aspects of weak scalability to it.

All of this data allows us to draw a conclusion about parallel systems, that conclusion being that utilizing multiple threads can allow us to greatly speed up the time it takes to solve large problems. These higher thread counts usually speed up smaller problems, too, but with much less of a benefit. It is very hard to make a highly effective parallel algorithm with strong scalability. For this reason, many of the algorithms that are made to utilize multiple threads are highly ineffective but still produce a significant speed-up.