CS 4504
PARALLEL & DISTRIBUTED COMPUTING

PROJECT REPORT – PART 2

Spring 2024

Instructor - Dr. Patrick Bobbie

Names: Eli Headley/02, Charlie McLarty/02, Sam Bostian/02, Ernesto Perez/02, Daron Pracharn/02, Michael Rizig/02, and Jonathan Turner/W01

***Abstract*—Distributed computing offers the advantage of dividing and sharing network resources among interconnected processes, mitigating bottlenecks and minimizing wasted potential caused by idle computing nodes. These network resources encompass not only individual files but also hardware components such as processors. Furthermore, distributed computing can be complemented by concurrent and parallel computing methods to tackle complex and sizable problems through a divide-and-conquer approach. Given the substantial computational demands of such problems, parallelizing and distributing computing tasks across a network is often more cost-effective than relying on a single powerful computer. However, one drawback of distributed computing lies in the complexity of coordinating network resources. The objective of this project is to leverage parallelization within distributed computing to sort data using an implementation of Merge Sort. The approach for this project involved establishing a Client-Server network utilizing the Single Program Multiple Data (SPMD) model. This model facilitated the intake of datasets from clients and the distribution of workload across multiple cores on the server side. Specifically, a Merge Sort algorithm was implemented to organize arrays of increasing sizes, dispersing the computational load across multiple distributed threads on the server side. Comparisons were made with the speedup, efficiency, and runtimes achieved by increasing the number of distributed cores across different array sizes against the metrics of a single-core processor.**

Keywords—Computer Networks, Distributed Computing, Merge sort, Parallel Computing, Socket Computing

INTRODUCTION

For this report on the client-server paradigm was used to transmit and receive data from a client node and is then for processed by a server that simulates multicore processing using threads. Each node in the network can act as a client or server, depending on which class is used, with an additional node acting as the server/router to facilitate handshakes and communication between these server and client roles. [4]

The client-server architecture as enhanced to simulate parallel programming while keeping the basic functionalities of the server-client paradigm intact, with a server/router used to act as an intermediary communicator.

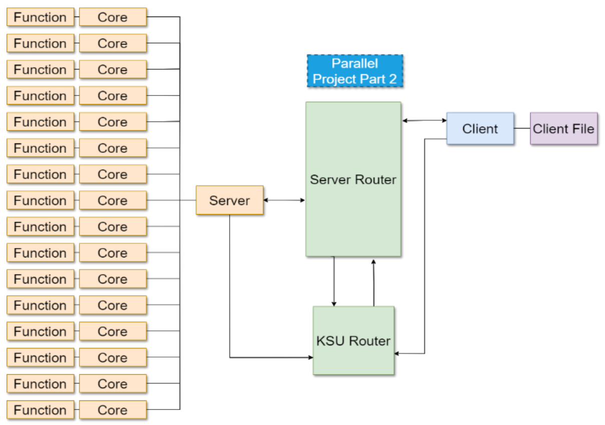

Figure 1: Client-Server Architecture with Multithreaded Server

A file containing a list of integers can be transmitted from a client node to a server node using parallelization, TCP, data parallelism, sockets, busy waiting, and event synchronization to facilitate reliable data transfers. [1]

The focus of this paper is the use of parallelism to increase the speed and efficiency of completing a task. A splitting algorithm was designed and used to split the data the Server receives from the Client and divide the set into computationally comparable sets depending on the size of the input and the number of threads. Each thread is then called to run the sorting algorithm on the subset it was provided, and the values are returned to the control thread that will complete the last merge. This approach ensures that a list can always be sorted regardless of whether there are more threads than values in the set or a large list has only one available thread. By designing and implementing the program in a parallelized-centric approach, the performance gains were simulated to mimic a real-world sorting algorithm when provided with a variable number of cores and values.

With the ability to simulate the effects of thread count and set size on the performance gain/loss of parallel computing, two experiments were conducted. This experiment is aimed at establishing a relationship between the number of threads and input size. These tests were run on lists consisting of sizes 10, 100, 200, and 300. By doing this, each of the four input sizes were run with five different counts of threads. 1, 2, 4, 8, 16. This was done to find the correlation between increasing the number of threads and worsening performance on smaller data sets. However, the significance of these exponential performance gains with increasing the data set was slightly apparent, so a second experiment was conducted.

The second experiment conducted aims to answer the question: "How significant would the performance increase for if the datasets become exponentially larger for each increase in the number of threads used for processing?" An attempt to answer this question by increasing each of the data sets by a factor of 1000. By doing this, the increase in performance, due to the number of threads, became increasingly apparent and significant.

This paper is organized in the following order: design architecture, implementation approach, simulation method, data analysis, and conclusion. The design section lays out the architecture used for the project. The implementation approach provides the specifics for how the Java programming language was implemented for parallel processing. The simulation method presents the metrics used to test the implementation techniques used for this project. The data collected from the simulations is broken down in the data analytics section. The final section, the Conclusion, summarizes the results obtained from the simulation and the experiences that were gained from these experiments.

DESIGN ARCHTIECTURE

The network architecture follows the Client-Server architecture with a third node called Server/Router to act as a handshaking intermediary. The process starts with the server router listening on socket port '5555' for connection requests and builds a routing table in the process. For each client-server pair the server/router creates an SThread object to facilitate the communication between each client-server pair. Once the connection has been established, the client creates a text file and generates a list of random unique integers of size n to send to the server for sorting.

Because a Single Program Multiple Data (SPMD) implementation is used, the data is split between the simulated cores. A merge sort was used because it lends itself easily to parallelization: each core receives a subset of the array and sorts it before merging with the sorted subsets from the other threads, as shown in Figure 1. With this fork-join style of parallelism that uses the divide and conquer approach larger datasets are divided into smaller more easily solved tasks.[5]

The file created on the client side, that was received on the socket port '5555' on the server side, is only passed to a single node instead of multiple nodes. The server processes the data received from the client side and divides the task of sorting the data between the simulated cores, as shown in figure one. Once the data is sorted and merged it is not sent back to the client, only the time taken to accomplish the sort is displayed in milliseconds.

Implementation Approach

This implementation of the client-server paradigm utilizes 5 unique classes, TCPServerRouter, TCPServer., TCPClient, SThread, and MergeSort.

TCPClient:

After the connection has been established between the client and server, the client will generate a random distinct array of the size n given by an argument to the function shown in the Random Distinct Number Generator code snippet. The function begins by creating a hash set and hashing each value randomly generated into it. It then checks each additional value to ensure that it is unique by checking for its existence in the hash set. If it is unique, meaning that it has not yet been hashed, it is added to the string followed by a comma to be used as a delimiter when the file is read at the server node. This function will return a string with n characters; where n represents the size of the array that will be sorted. The file can then be easily parsed once it is received by the server.

Random Distinct Number Generator

```
public static String randomDistinct(int n) {
    /* Used to track what ints have been used
*/
    HashSet<Integer> isUsed = new HashSet<>();
    String newList = "";
    Random random = new Random();

    /* Loops 1024 times and assigns a new
unique value each time. */
    for (int i = 0; i < n; i++) {
        int newValue = -1;

        /* Ensures that there is no duplicate
values */
        do {
            newValue = random.nextInt(999999)
+ 1;
        } while (isUsed.contains(newValue));

        /* Adds the value to the known list
and also to the generated list. */
        newList+= newValue + ",";
        isUsed.add(newValue);
    }
    newList = newList.substring(0,
newList.length()-1);
    /* Sorts and assigns the values generated
to the field. */
    return newList;
}
```

After this array is instantiated by the client, a file named 'file.txt' is created and a PrintWriter object is utilized to pass the generated array into this newly created file. This process is the equivalent to data marshalling in a real-world environment as the data is 'marshalled' into a format that both the client and server can understand, a text file is used in this project's implementation. The line "Bye." is appended to the end of the file to signal to the server that it has reached the end of the file. The file is then sent over the network to the server using a TCP socket. To do this, the client creates a socket connecting it to the server router on port 5555 and passes the server router to the intended destination by passing it the IP address of the server. The output stream from this socket is then passed into the constructor of a DataOutputStream allowing it to take advantage of its ability to send raw bytes through the socket. For the implementation of this project, the client sends the array as a stream of bytes utilizing a buffer of 8kb. A FileInputStream object is utilized to loop through the file, reading in 8kb into the buffer with each iteration, and sending those bytes through the DataOutputStream, and repeating until all bytes are sent. This process is demonstrated in the File Creation and Writing code snippet. Finally, once all the data is sent, the objects are closed, and the connection is terminated. The client is no longer needed as the server will not return the sorted array after it's processed. [4]

File Creation and Writing

```
File vFile = new File("src/Client/file.txt");
//creates file
byte[] Bytes = new byte[8192];
//creates a byte buffer for bytes
InputStream dis = socket.getInputStream();
OutputStream dos = socket.getOutputStream();
int sentCount = 0;
FileInputStream fis = new
FileInputStream(vFile);
//write file to dos
while ((sentCount = fis.read(Bytes)) != -1) {
    dos.write(Bytes, 0, sentCount);
}
```

TCPServerRouter:

The server/router utilizes a ServerSocket object as the server/router node prepares to handle simultaneous incoming connections. Each connection is stored in a 'lookup' table to allow for incoming requests to be referenced against the table to serve each request to the correct and intended receiver. The lookup table houses each socket and that socket's host address. These connections are initially facilitated by the accept() method blocking the execution of the process until a connection is received. Once a connection is established, an SThread is created and passed to that connection in addition to the address of the lookup table. This allows the thread to service the connection while the original server router process can continue accepting additional connections. This can be considered as an 'asynchronous blocking' technique as shown in the Socket Creation and Connection code snippet 3. [2][6]

Socket Creation and Connection

```
clientSocket = serverSocket.accept();
            String [] ip =
(clientSocket.getRemoteSocketAddress() +
"").split(":");
            pw.println(ip[0].substring(1));
//this line updates the routing table with a
new line holding the clients ip in slot one
and a delimiter eg. [clients ip] , [slot for
servers ip]
            pw.flush();
            RoutingTable[ind][0] =
ip[0].substring(1);
            RoutingTable[ind][1]=clientSocket;
SThread t = new SThread(RoutingTable,
clientSocket, ind); // creates a thread with a
random port
t.start(); // starts the thread
ind++; // increments the index
```

SThread:

This class contains the thread object used to serve each connection and act as a middleman between the client-server pair. It starts by taking in the routing table and the client's socket passed through as an argument to the SThread constructor before parsing the lookup table until the intended destination is found as shown in the SThread connection code snippet. Once found, the socket takes the InputStream of the client and passes it to the OutputStream of the server, which facilitates the connection. The beauty of this structure is that the main server router process is completely unaffected by this process and continues accepting connections.

SThread Connection

```
// loops through the routing table to find the
destination
for (int i = 0; i < 10; i++) {
    if (destination.equals((String)
RTable[i][0])) {
        outSocket = (Socket) RTable[i][1]; //
gets the socket for communication from the
table
        System.out.println("Found destination:
" + destination);
        outToClient =
outSocket.getOutputStream(); // assigns a
writer
    }
}
```

TCPServer:

The server handles the main portion of this implementation as it does the sorting of arrays. This process can be divided into three subsections: receiving the array, dividing the work, and merging the results. The first operation the server needs to accomplish is receiving the file containing the list of integers array. Upon the server process beginning execution, a constant is defined for how many threads the server wants to utilize for sorting. This can be easily modified for testing purposes. The line below demonstrates this for an example run using 8 threads:

```
final int NUM_OF_THREADS = 8;
```

The server begins by connecting to the server router in the same fashion as the client. It then utilizes a DataInputStream as well as the 8kb size buffer to accept the sent bytes and parse them into strings. A vector is utilized to hold the lines. A vector was chosen since it is a dynamic data type, and can accept many lines without crashing, whereas a fixed array can only hold a fixed number of lines. Once the server receives the designated end term, "Bye.", the loop terminates stopping the parsing. As a result, an array is stored in a string entitled 'fromClient' containing the values of the full array. Finally, a thread pool is created to house the number of threads defined in the NUM_OF_THREADS constant. The receiving process is shown in the Array Receiving code snippet.

Array Receiving

```
fromClient = in.readLine();// initial receive
from router (verification of connection)
 System.out.println("ServerRouter: " +
fromClient);
 // Communication while loop TODO:: alter this
before running with client
 Vector <String> s = new Vector<>();
        while ((fromClient = in.readLine()) !=
null) {
            s.add(fromClient);
            //System.out.println("Client said:
" + fromClient);
            if (fromClient.equals("Bye.")) //
exit statement
                break;
        }
        fromClient=s.get(0);

 //array is now received by the server and a
thread pool is created
Thread[] threadpool = new
Thread[NUM_OF_THREADS];
```

The next step is to convert the string of integer values into an integer array that can be sorted. Since the arrays will be frequently passed between methods and threads, an Integer object array was chosen to store the numbers rather than a primitive integer array as it is easier to pass by reference rather than by value between threads.

This was achieved using the string split operation on the received string, which parses through the array converting each element from a string to an integer. Array Creation code snippet displays this step in the program.

Array Creation 1

```
String[] splitString = fromClient.split(",");
//create a wrapper object array to store
values
Integer[] arrayValues = new
Integer[splitString.length];
//loop through and insert values into Integer
object array to pass by reference
// this is o(n) overhead, maybe find better
way to copy over values
for (int i = 0; i < arrayValues.length; i++) {
    arrayValues[i] =
Integer.parseInt(splitString[i]);
}
```

The next step is to divide the work among the threads into equal parts too. This is first done by creating a vector called workDiv of type <Integer []> that will store each unit of work to be passed to the threads. A 'startIndex' is then defined which holds the size of each unit of work and the starting index of the second subarray. The size of each unit of work can be found by dividing the total length of the input array, n, by the number of threads. If the resulting starting index is not an even factor of n, it is incremented to ensure no extra values are ignored. Because startIndex works as a pointer to the start index of the next portion of values, the size of each increment needs to be stored in a separate variable called 'inc'. At this point sorting is ready to begin and the timer begins tracking. The division of work is shown in the code snippet Dividing the Array.

Dividing the Array 2

```
Vector<Integer[]> workDiv = new Vector<>();
int startIndex = (arrayValues.length /
NUM_OF_THREADS);
if (startIndex % arrayValues.length != 0) {
    startIndex++;
}
System.out.println("after spliting");
//inc below stores the increment to move up at
each itteration of while loop
int inc = startIndex;
int prev = 0;
long start=System.nanoTime();
```

For sorting the array, merge sort was employed for its consistent efficiency and ease of dividing the data among the threads. A MergeSort object was created which implements the Java Runnable interface enabling it to be runnable by a thread. It operates as a typical merge sort, except it works in primitive types so it required us to pass the values as a primitive type and copy them back to an Integer object afterwards. The merge sort constructor and run method is shown in the code snippet MergeSort Instantiation.

MergeSort Instantiation

```
mergeSort(Integer [] array)
{   this.array=array;
    this.intermediate=  new int
[array.length];
    for(int
j=0;j<this.intermediate.length;j++){
        this.intermediate[j] = array[j];
    }
}

@Override
public void run() {
    sort(intermediate, array.length);
    for(int j=0;j<this.array.length;j++){
        this.array[j] = intermediate[j];
    }
}
```

Once the size of the work units has been determined, work can be assigned to the threads. If there are 1 or 2 threads active, a separate case is enacted to minimize the overhead as shown in the Work Divide Case 1 or 2 code snippet. For 1 thread, the singular thread in the thread pool is simply activated and passes through the entire array. For 2 threads, the array is divided in half and each half is passed into a thread. For either case, the threads are initialized and pass these threads a `new mergeSort(arrayValues)` and start each thread. Additionally, there is a check to ensure that there are not more threads than array values in extreme cases. If the number of values is small, for example 10, they are passed to a single thread. This is done because the overhead for separating each value, creating all the threads, 'sorting' and merging would take significantly longer rather than just passing the 10 values to a single thread. Since only one parallel task is worked on at a time, it was determined to not use solutions, such as dynamic blocking where the task size would be a dynamically increased aggregating number of parallel tasks, before enqueuing it as one batch task.[7] In either case, once the threads are started, the server uses a spin lock until the threads join back to ensure that all threads are completed before the merging process can begin.

Work Divide Case 1 or 2

```
if (NUM_OF_THREADS == 1 ||
arrayValues.length<threadpool.length) {
    //create thread and pass it the full array
    threadpool[0] = new Thread(new
mergeSort(arrayValues));
    //start thread
    threadpool[0].start();
    //a timer to ensure thread finishes before
the prints proceed
    while(threadpool[0].isAlive());
} else if (NUM_OF_THREADS == 2) {
    //since there are 2 threads, 2 subarrays
are needed
    //divide the input array (arrayValues)
into 2 subarrays of each half
    Integer[] first =
Arrays.copyOfRange(arrayValues, 0,
arrayValues.length / 2);
    Integer[] second =
Arrays.copyOfRange(arrayValues,
(arrayValues.length / 2), arrayValues.length);
    //create threads for each half
    threadpool[0] = new Thread(new
mergeSort(first));
    threadpool[1] = new Thread(new
mergeSort(second));
    //start threads
    threadpool[0].start();
    threadpool[1].start();
    //method to ensure they finish
    while(threadpool[0].isAlive() ||
threadpool[1].isAlive());
    //finally, merge the two
    merge(arrayValues, first, second);

}
```

The third case is when there are more than 2 threads, the work will need to be dynamically allocated and multiple threads will be created. A simple algorithm was devised to divide the work evenly utilizing the previously mentioned variables 'startIndex' and 'inc' by looping through and adding each 'inc' worth of values starting at each 'startIndex' pointer, then increasing the pointer to assign the next unit of work and saving the previous pointer in a previous variable for later. Then a unit of work is added to the previously defined 'workDiv'. At each iteration of the loop, a check is performed to ensure that an index out of bounds runtime error is not thrown by checking if the next 'startIndex', the 'startIndex' + 'inc', is less than the array length. A boolean flag is used to control the while loop, with flag 'f' initially set to true. If the next 'startIndex' + 'inc is less than the size of the array, it is determined that there is one division of work remaining, and this division of work is smaller than the 'inc' size. The last bit of values is added to a final 'workDiv' unit and fllag f is set to false to terminate the loop. At this point the work is divided out into equal parts with one final unit being slightly smaller. This algorithm is implemented in code snippet Work Divided Case 3.

Work Divided Case 3

```
       while (f) {
                //split work into 1/n parts
                Integer[] work =
Arrays.copyOfRange(arrayValues, prev,
startIndex);
                //insert work into vector
                workDiv.add(work);
                //update prev and start index
to next values
                prev = startIndex;
                startIndex += inc;
                //check if the next iteration
will cause condition to fail
                if (startIndex + inc >=
arrayValues.length) {
                    //if so the below will
create an array of all the remainng values
that will not fill an array
                    Integer[] ww =
Arrays.copyOfRange(arrayValues, prev,
startIndex);
                    prev = startIndex;
                    //add the last full sized
array to the vector
                    workDiv.add(ww);
                    //make the index to stop
equal the last value of the array
                    startIndex =
arrayValues.length;
                    //copy the last values
that will not full an array into their own
smaller arary and add
                    Integer[] w1 =
Arrays.copyOfRange(arrayValues, prev,
startIndex);
                    workDiv.add(w1);
                    //stop the loop
                    f = false;
                }
            }
```

At this point the threads are ready to begin execution by looping through the thread pool. While the threads are running, the server process's spin lock is applied until all threads have completed execution as show in the Run Threads code snippet.

Run Threads

```
//activate all threads
for (int i = 0; i <
Math.min(threadpool.length, workDiv.size()) ;
i++) {
    threadpool[i] = new Thread(new
mergeSort(workDiv.get(i)));
    threadpool[i].start();
}
// small pause to wait for threads to return
to us. (could make these threads synchronous)
boolean [] alive=new
boolean[Math.min(NUM_OF_THREADS,
workDiv.size())];
boolean allAlive=true;
while (allAlive){
    for( int i=0;i< alive.length;i++){
        if(!threadpool[i].isAlive()) {
            alive[i] = true;
        }
    }
    allAlive=false;
    for (boolean I : alive){
        if(!I){
            allAlive=true;
            break;
        }
    }
}
```

The final step is to merge the results. Typically, the merge sort algorithm will merge two arrays at a time recursively, however for this implementation the subarrays are merged at once with a custom merge algorithm. The merge function takes in two parameters, a pointer to the output array, the initial input integer array object, and a pointer to a vector containing all the subarrays, 'workDiv'. The algorithm begins by creating a bitmap style counter integer array of size n, that stores the current index of each input array in the vector. A nested loop is then utilized where the outer loop iterates through each index of the output array and the inner loop parses through each sub array at the index of each subarray's counter. The inner loop checks to ensure no memory violation will occur by making sure the size of the array is greater than the soon to be index before going to the current index of each sub array and finding the lowest. It does this by comparing each element to the 'smallest' variable and storing the smallest value as well as the index of the counter array that produced the value. When all the values are compared, and the smallest is found, that value is stored into the output array and the respective counter is indexed to ensure that each subarray progresses. This process repeats for every index of the output array until all the sorted subarrays are merged into one final array. Finally, this array is passed back to the server, and the server's work is complete. This custom algorithm is shown in code snippet Merge.

Merge
```
public static void mergeAll(Integer[] output,
Vector<Integer[]> input) {
    //counters array acts as int bit map to
keep track of each subarrays index
    int[] counters = new int[input.size()];
    //2 vars to keep track of current smallest
val and its index in the given array
    //these values are used so it will be
clear if either is causing a bug
    int smallest = 9999;
    int index = -1;
    //outer loop counts index of output array
    for (int i = 0; i < output.length; i++) {
        //inner loop works through each
subarray in the vector
        for (int j = 0; j < input.size(); j++)
{
            //first check to see if the index
will cause an exceptoin (array is out of novel
values)
            if (counters[j] ==
input.get(j).length) {
                continue;
            }
            //else check if given arrays value
is smaller than running smallest
            else if (input.get(j)[counters[j]]
< smallest) {
                //update running smallest and
its index
                smallest =
input.get(j)[counters[j]];
                index = j;
            }
        }
        //finally update the current index of
output array with correct smallest value
        output[i] = smallest;
        counters[index]++;
        //reset index and smallest counters
        index = -1;
        smallest = Integer.MAX_VALUE;
    }
}
```

Finally, the total time the server took to complete the sort is calculate by subtracting the final time by the initial time and storing as shown below:

```
start = System.nanoTime()-start;
```

Simulation Method

To test the theory that a program's runtime can be decreased by dividing the problem into approximately equal smaller parts. These smaller problems are passed to threads used to simulate a multicore processor that will each smaller problem. After the work is performed by the threads the parts are returned to be combined to form a solution to the original problem. To simulate multiple processors threads were created to act as a core in the CPU that receives the smaller portions of the problem. The problem that was used to test this theory was to sort an array of varying sizes that contained distinct integers. The array is divided into parts equal to the number of processors used during that trial. The number of processors tested were: 1, 2, 4, 8 and 16 with arrays varying from 10 to 300000 elements. The time taken by the sort was recorded in milliseconds after each trial. Two experiments were conducted using two different arrays that increased by two different factors. The first experiment focuses on how smaller arrays are affected by parallel processing from a size 10 to 300 elements. The second experiment then increases the first experiment's arrays by a factor of 1,000 to see how larger arrays are affected with parallel processing practices.

A. Experiment I

For the first experiment, four arrays were generated of sizes 10, 100, 200, and 300. Then each array was populated with random and distinct numbers ranging from 1 to 999,999 to ensure the use of a wide range of data. These arrays were each written to separate text files, that were delimited by commas. Subsequently, these text files were transmitted to the server process via a TCP socket. Once the data reached the server side, each array was sorted using a modified Merge Sort algorithm to support multithreading. For the context of this experiment, this sorting process was performed on two separate computers with different hardware configurations and was conducted in five stages for each array size.

In each stage, the specified array size was sorted using a thread count of 2^n, where n represented the stage number indexed by 0. For instance, in the first stage for the array size of 10, the array was sorted by 20 threads, providing us with the serial performance for that dataset. This procedure was repeated for each stage, thereby giving us the performance for thread counts ranging from 1 to 16 threads.

After each stage, the results for each array size were compiled into a table, which will be discussed later.

B. Experiment II

The second and final experiment, four arrays were generated again, following a structure similar to that of experiment one. However, each array size was scaled up by a factor of 1,000, resulting in sizes of 10,000, 100,000, 200,000, and 300,000 elements, respectively. Once again, each array was populated with random and distinct integers ranging from 1 to 999,999. Subsequently, the arrays were transmitted via a TCP socket to the server process.

Similar to experiment one, the server process executed a modified Merge Sort on each text file corresponding to a different array size and calculated the overall time required to sort the elements. The sorting process was divided into five distinct stages. In these stages, the thread counts varied, starting with 2^0 threads in the initial stage and concluding with 2^4 threads in the final stage. The results obtained for each scaled array size were compiled into a table, which will be discussed in the following section.

Data Analysis

From each experiment the time was collected, in nanoseconds and recorded in milliseconds, for the time it takes the server to process and sort a received array. The timing begins at the creation of the threads and finishes when the data from each thread finishes merging creating a complete sorted array. For the experiments the improvement in performance of increasing the number of parallel processors versus the serial one, is measured using the speedup and efficiency metrics. The speedup, the ratio of the program runtime in serial over the runtime in parallel:

$$Speedup(n,p) = \frac{T_{Serial(n)}}{T_{Parallel(n,p)}}$$

[8]

Where n is the size of the input and p is the number of processors. A perfect speedup score is where the speedup equals the number of processors, Speedup(n,p) = p, also known as linear speedup. To determine how each processor contributed to the speedup parallel efficiency is used. Parallel efficiency is calculated using the following formula:

$$Efficiency(n,p) = \frac{Speedup(n,p)}{p} = \frac{T_{Serial(n)}}{p * T_{Parallel(n,p)}}$$

[8]

Parallel efficiency is given by the speedup over the number of processors. A perfect score, which corresponds to linear speedup, is the # of processors / # of processors = 1.0, meaning that each processor is being used to its maximum potential.

These three metrics; runtime, speedup and efficiency; are used to evaluate and graph the output of the parallel server on varying types of input sizes. The first experiment captured the performance of the server for a smaller set of array sizes in order to test the lower boundary of performance. In contrast, the second experiment scaled these original input sizes to mimic real world performance with much larger input sizes.

C. Experiment I Results

In the initial experiment, four separate files were transmitted with input sizes ranging from 10 to 300 elements to be sorted. As detailed earlier in the Simulation Method, each of these input files underwent five distinct stages, with each stage spawning a different number of threads to aid in computation. In the first stage, illustrated in Table 1, the server sorted input sizes serially, utilizing only one core.

Examining the first column of Table 1, where the array size of 10 was sorted using 1 through 8 cores, it is observed that the increase in cores has no effect on the time required to sort the data. This lack of impact stems from the negligible input size, causing any overhead incurred by using additional threads to outweigh the benefits of parallelism. Additionally, it is important to note for the program's implementation the program executed in serial whenever the thread count would exceed the elements in the arrays causing the last value in the column to be the same as the cell with serial time. This phenomenon is further underscored by the second column of Table 1, which depicts the runtime for an array size of 100 elements. Here, an increase in runtime corresponding to the increase in the number of threads used is seen, exemplifying the scenario where the overhead from parallel sorting surpasses the benefits.

However, upon examining the latter two input sizes in the table, it is discerned that there is an increase in performance with the escalation of cores used, until four cores is surpassed. This improvement arises from the problem size being sufficiently large to yield faster performance despite the overhead. Nevertheless, as the number of cores increases, the performance gain diminishes, albeit still outperforming the serial time.

Runtime

Number of Threads	Array Sizes (# of elements)			
	10	100	200	300
p=1	2	2	6	7
p=2	2	2	4	5
p=4	2	2	2	3
p=8	2	3	4	4
p=16	2	4	5	7

in experiment 1.

The results presented in Table 1 are further elucidated by the 3-Dimensional plot depicted in Figure 2. This graphical representation showcases four curves, each illustrating the runtime plotted against the input size and the number of cores utilized. It is evident that as both the input size and core count increase, the trough of the curve deepens, indicating an enhancement in performance resulting from the utilization of more cores to process the data.

However, this performance gain begins to diminish once a threshold of 4 cores is reached. This decline can be attributed to the overhead incurred from spawning additional threads outweighing the benefits yielded by those threads.

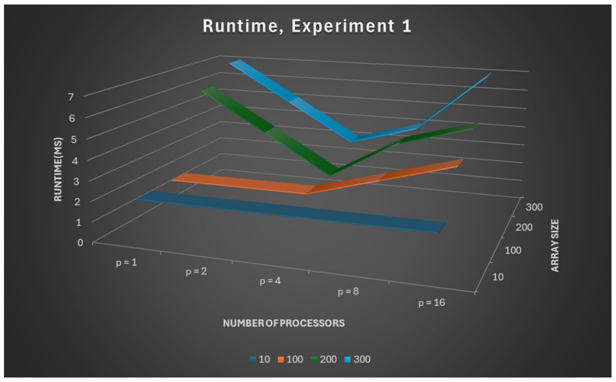

Figure 2: A line graph displaying the recorded runtimes for each array size for a given number of processors for experiment 1.

After collecting the runtimes, the speedup was computed using the respective formula for each set of threads across all array sizes. Table 2 showcases these calculated values. As anticipated, the first column in Table 2 exhibits a speedup of 1 since each set of cores operated within serial time. The speedup begins to noticeably decline when the second column in Table 2 is examined, where the overhead from employing 8 and 16 threads resulted in an overall slowdown.

Speedup				
Number of Threads	Array Sizes (# of elements)			
	10	100	200	300
p = 1	1	1	1	1
p = 2	1	1	1.5	1.4
p = 4	1	1	3	2.33
p = 8	1	0.67	1.5	1.75
p = 16	1	0.5	1.2	1

Table 2: Calculated speedup for the randomly generated arrays in experiment 1.

However, notably improved results are observed in the latter two columns, particularly with the array size of 200, where a speedup of 3 is achieved when using 4 threads to sort the elements. Remarkably, the server attained the most favorable speedup results with an array size of 200, slightly surpassing the adjacent column. This discrepancy is further accentuated by the 3-D graph depicting the speedups in Figure 3. It is evident that the curve for the array size of 200 exhibits a substantially higher peak but a swifter decline in speedup. This variation can be attributed to the divisibility of the data in the implementation. The array size of 200 is considerably more divisible by powers of two, leading to a more even distribution of the data. This equitable distribution of the workload accentuates the speedup, particularly for smaller datasets.

Although, upon increasing the cores beyond 4, a sharper decrease is observed in speedup compared to the last column. This decline can be attributed to the smaller overhead-to-performance ratio for the smaller datasets, unlike the array size of 300 elements.

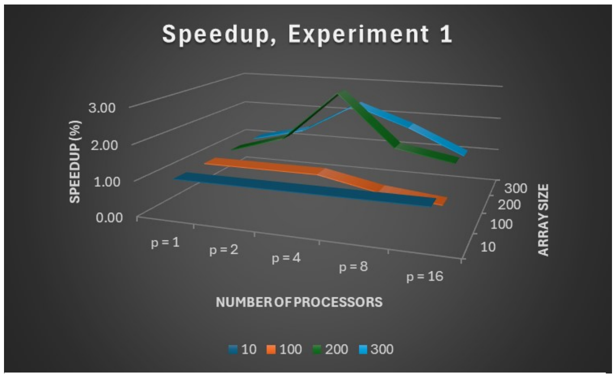

Figure 3: A line graph displaying the calculated speedup for each array size for a given number of processors for experiment 1.

Following the computation of speedup for the parallel server, the efficiency is analyzed to see how effectively the server distributed the work among the cores. Table 3 presents all the calculated efficiencies derived using the efficiency formula mentioned earlier. As anticipated, the efficiency for the array size of 10 elements was notably subpar. Upon examination, it becomes apparent that one core predominantly handles the workload. This outcome is expected, considering the small input size, where most of the data is assigned to the initial core, with any remaining portion distributed among others.

This declining trend persists across the remaining dataset, with efficiency diminishing as the number of cores increases. This suggests that the server struggles to distribute the workload efficiently for larger core counts, particularly evident in smaller input sizes, indicating that a single core bears the brunt of the computation.

Efficiency				
Number of Threads	Array Sizes (# of elements)			
	10	100	200	300
p = 1	1	1	1	1
p = 2	0.5	0.5	0.75	0.7
p = 4	0.25	0.25	0.75	0.58
p = 8	0.13	0.08	0.19	0.22
p = 16	0.06	0.03	0.08	0.06

Table 3: Calculated efficiency for the randomly generated arrays in experiment 2.

Furthermore, it's noteworthy that the efficiency peaks for the test file with an input size of 200. This is clearly illustrated in the 3-D plot presented in Figure 4, depicting all efficiencies plotted against the number of threads and the input size. Figure 4 highlights the highest peak for efficiency, observed with more than one core, specifically at thread counts of 2 and 4 for an input size of 200 elements. As mentioned earlier, the server exhibited the best speedup with an input size of 200 elements due to the improved divisibility of the data. This observation is further supported by the efficiency, indicating that the optimal efficiency was achieved when the input size was 200 elements in experiment 1.

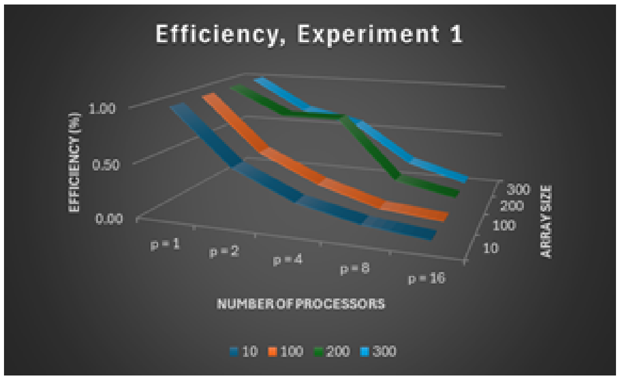

Figure 4: A line graph displaying the calculated efficiency for each array size for a given number of processors for experiment 1.

D. Experiment II Results

After it was observed in the first experiment that there was no real increase in efficiency or speedup as the array was divided into an increasing number of processors. The steps of the first experiment were repeated with the size of the arrays increased by a factor of 1,000. With the array sizes increased 1,000 times the benefits of parallel processing can be seen.

As the array increased the benefits of parallel processing can be seen in Table 4. When the array is only 1000 the time it takes to process the array is by approximately half from one to two processers. The decrease in time is again reduced by half from using two processors. When eight and sixteen processors are used on an array of 10,000 elements the decrease in time to process the arrays is hampered by the work to divide the array. When the array size is increased from 100,000 to 300,000, the decrease in time to process for number of splits equal to the number of cores is noticeable. The runtime decreases between about a quarter a third of the time when the number of processors is increased between two and four processors. When the processors are increased to eight or sixteen the runtime is only decreased by about half.

Runtime				
Number of Threads	Array Sizes (# of elements)			
	10000	100000	200000	300000
p = 1	591	48221	183908	403233
p = 2	258	14191	57020	109616
p = 4	123	4532	15974	34549
p = 8	123	2278	8335	18565
p = 16	104	1257	4668	10450

Table 4: Runtimes for the randomly generated arrays in experiment 2.

When the runtimes are plotted onto a 3D graph the difference between runtimes depending on the number of processors and array size can be seen. It is shown in figure 5 that the greatest decrease in time from one processor is seen in an array of 300,000 elements. While an array of 10,000 elements might look like it does not decrease but that is due to the fact that the 300,000 elements decrease so much compared to the 10,000 elements array.

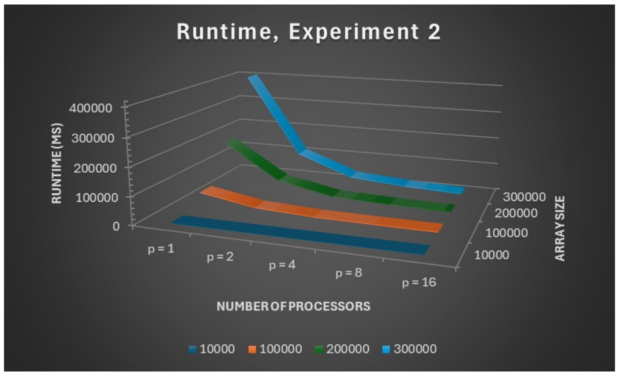

Figure 5: A line graph displaying the recorded runtimes for each array size for a given number of processors for experiment 2.

The speedup gained by using parallel processing is almost equal for each processor regardless of the size of the array. The speed up almost increase by the same factor has the number of processors doubles. The speed up is more than tripled from one processor to two processors and from two to four processors. The speedup is slightly slowed to an increase of a factor of two when the number of processors increases from four to eight. When the number of processors is doubled from eight to sixteen the speedup is only gained by a factor of 1.75.

Speedup				
Number of Threads	Array Sizes (# of elements)			
	10000	100000	200000	300000
p = 1	1	1	1	1
p = 2	3.68	3.4	3.23	3.68
p = 4	11.67	10.64	11.51	11.67
p = 8	21.72	21.17	22.06	21.72
p = 16	38.59	38.36	39.4	38.59

Table 5: Calculated speedup for the randomly generated arrays in experiment 2.

The similarities between each sized array are best visualized by graphing table 5 to a three-dimensional line graph where the depth axis is the size of the arrays used, the horizontal axis is the number of processors used and the vertical axis is the speedup represented as a percentage. The graph in figure 6 shows that the speedup follows the same shape as the theoretical big O trajectory as a merge sort, which is O(nlog$_2$n).

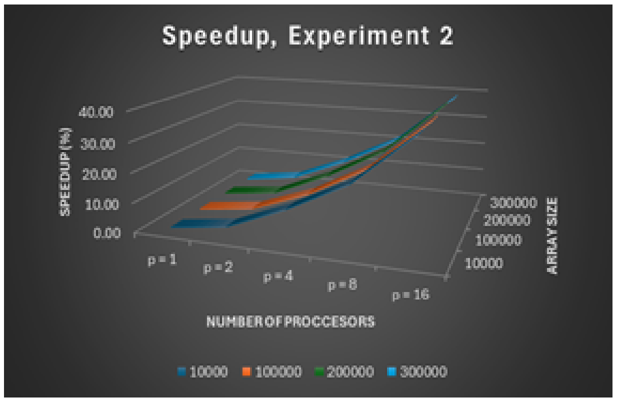

Figure 6: A line graph displaying the calculated speedup for each array size for a given number of processors for experiment 2.

Unlike the speedup the efficiency did increase the same amount for all the array sizes. The array with 10,000 elements was the least efficient while the arrays with a hundred thousand elements or more had approximately the same efficiency. The ten thousand element array followed the same pattern as the larger arrays but had a lower increase in efficiency for each increase in the number of processors used. All the arrays had their greatest efficiency when four threads were used to sort an array. The efficiency for each processor begins to decrease after four threads are used.

Efficiency				
Number of Threads	Array Sizes (# of elements)			
	10000	100000	200000	300000
p = 1	1	1	1	1
p = 2	1.15	1.7	1.61	1.84
p = 4	1.2	2.66	2.88	2.92
p = 8	0.6	2.65	2.76	2.72
p = 16	0.36	2.4	2.46	2.41

Table 6: Calculated efficiency for the randomly generated arrays in experiment 2.

The changes in efficiency for each array are visualized in figure 7. The arrays 100,000 elements and larger the slope of the graph increases between a factor of 0.6 to 0.8 for one to eight processors used. When the number of processors is increased to eight and higher the efficiency is between 88% to 95% the efficiency of the previous number of processors. The ten thousand element array has the smallest gain in efficiency and has the largest decrease in efficiency as the number of processors increases after four threads. Efficiency only increases by 15% and then 5% as the number of processors is doubled from one to four. When the number of processors is doubled again to eight and sixteen the efficiency decreases by about 50% for each increase.

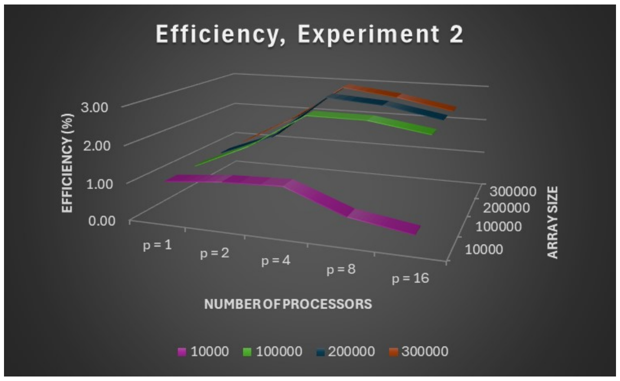

Figure 7: A line graph displaying the calculated efficiency for each array size for a given number of processors for experiment 7.

Conclusion

The primary goal of this research paper was to implement a multi-threaded server capable of sorting large data inputs sent over a network from clients. The SPMD task parallelism method of implementation effectively addressed the issues of multi-threading including thread synchronization and load balancing of the data among the concurrently running threads [8].

Times were documented for various combinations of array sizes ranging from 10 to 300,000 as well as the number of threads used between 1 and 16. These conditions were each run separately on two computers of differing specifications. During the two experiments run, there was almost no speedup and even increased time in some cases when the number of cores increased for the smaller arrays. This can be explained by the overhead required to distribute the work across many threads when there is just not enough work required to justify it. However, for the large arrays (>100,000), it was observed that a significant speedup occurred when the number of threads increased. This can be explained due to the cost of overhead minimal compared to the increased efficiency of the added cores. The efficiency in these large test cases indicated a super linear speedup where the speedup is greater than anticipated for an increased number of cores.

This research showed that there is no optimal number of cores that will suit all cases. To allow more consistent results in a dynamic environment of differing input sizes, a threshold could've been implemented to assign the number of cores on runtime. Overall, it was determined that spending the extra time to implement parallel processing for a sorting algorithm yielded significantly better results when the amount of data is substantial.

References

[1] B. A. Forouzan, Data Communications and Networking, 5th ed. New York, NY: McGraw-Hill Professional, 2012.

[2] "Writing the server side of a socket," Oracle.com. [Online]. Available: https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html. [Accessed: 09-Feb-2024].

[3] "Socket (java platform SE 8)," Oracle.com, 08-Jan-2024. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html. [Accessed: 23-Feb-2024].

[4] M. L. Liu, Distributed computing: Principles and applications: United States edition. Upper Saddle River, NJ: Pearson, 2003.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms. Cambridge (Mass.): MIT Press, 2009.

[6] Baeldung.com. [Online]. Available: https://www.baeldung.com/java-socketexception. [Accessed: 23-Feb-2024].

[7] K. Streit, J. Doerfert, C. Hammacher, A. Zeller, and S. Hack, "Generalized task parallelism," ACM Transactions on Architecture and Code Optimization, Available: https://dl.acm.org/doi/abs/10.1145/2723164 [accessed Apr. 13, 2024].

[8] P. Pacheco, "An Introduction to Parallel Computing", Elsevier Inc., 2011. ISBN-10: 01237426095

Appendix

The demonstration and the execution of this program with the different input sizes can be found in the mp4 file attached in the submission of this report.