Distributing Parts of a Matrix over Sockets

John Rommel B. Octavo

***Abstract*—The study aims to assess the performance of a program utilizing socket programming for distributing matrix computations across different configurations, including one machine with multiple terminals and multiple machines.**

The methodology involves developing a threaded and core-affine program in the C programming language. It utilizes sockets for inter-process communication and uses a one-to-many broadcast communication technique. Matrix partitioning and distribution are implemented, and experiments are conducted on various sets of computers to evaluate performance.

Results indicate that on a single machine with multiple terminals, the program's runtime exhibits an increase in runtime on $t > 4$ terminals. This is due to increased computational overhead. Similarly, the core-affine version of the program demonstrates comparable behavior, with additional computational costs incurred by core assignment. In utilizing multiple machines, improved runtime performance is observed. This is attributed to parallel processing across dedicated computing nodes.

Despite variations in runtime across different configurations, the trend of increasing runtime with higher matrix dimensions remains consistent for all the experiments.

***Index Terms*—socket programming, POSIX thread, core-affine, multithreading, matrix distribution, CPU core**

I. Introduction

THE rise in the demand for scalable and efficient computation in various domains such as data analytics and scientific computing pushes the advancement in the field of distributed systems [1]. Distributed computing faces growing challenges in terms of efficient distribution and coordination of tasks among multiple computational units [2]. One of the common ways of distributed computing is by using socket programming. Using socket programming facilitates the communication between processes over a network.

This research paper focuses on the application of socket programming for distributing parts of a matrix across multiple computing nodes. Matrix computation lies at the heart of many scientific and engineering applications, including linear algebra operations, image processing, and numerical simulations. Distributing matrix computation tasks among multiple computing nodes can significantly accelerate the overall computation time, enabling the handling of larger datasets and more complex problems.

The use of sockets, which provide an interface for inter-process communication over a network, allows for seamless communication between distributed computing nodes [3]. By using sockets, it becomes possible to partition a large matrix into smaller submatrices and distribute these submatrices across multiple computing nodes for parallel computation. Each computing node can then independently perform computations on its assigned submatrix and communicate the results back to the main process or other nodes as needed [4].

The distributed nature of socket programming introduces several key advantages. Firstly, it enables parallelism, allowing multiple computations to be performed simultaneously across different computing nodes. Secondly, it enhances scalability by enabling the addition of more computing nodes to the system as the size of the problem or the computational workload increases. Lastly, it enhances fault tolerance by allowing the system to continue operating even in the presence of failures in individual computing nodes.

However, the effective utilization of socket programming for distributed matrix computation requires addressing various challenges, including load balancing, data partitioning, synchronization, and fault tolerance [5]. This research paper aims to analyze the performance of distributing parts of the matrix on a machine with multiple terminals and different machines.

A. Objectives of the Study

This study aims to assess the performance of a program that uses sockets to distribute parts of the matrix. It specifically focuses on the use of one machine with different terminals and multiple machines in the development of this algorithm.

The specific objectives of this research study are the following:

1) To develop a threaded and core-affine program that will distribute parts of the matrix over a network using sockets; and,
2) To analyze the performance of the program when using one machine with different terminals and multiple machines.

II. Methodology

A. Development Tools

The experiments for the distribution of the parts of the matrix using sockets will be conducted on different sets of computers. Firstly, in the experiment on one machine with different terminals, the specification is as follows:

- Operating System: Windows 11 Home Single Language
- Processor: Intel® Core™ i5-1035G1 CPU @ 1.00GHz 1.19 GHz
- Memory: 32 GB DDR3 L

In the experiment on different machines, the specification of the computers is as follows:

- Operating System: Ubuntu 22.04 LTS
- Processor: Intel® Core™ i7
- Memory: 16.0 GB DDR4

The computer program was developed on Visual Studio Code, and it uses the C programming language. The C programming language was used since compiled language has better performance in terms of threaded and core-affined computation.

B. The Matrix Creation and Division

The experiment used different matrix dimensions that were divided and distributed to the slaves. The master function creates a non-zero n x n matrix M and it is divided into t submatrices of size n/t x n. Table 1 shows the matrix dimension n and the number of slaves for each experiment.

TABLE 1: Matrix Dimension and the Corresponding Number of Slaves

n	t			
20000	2	4	8	16
25000	2	4	8	16
30000	2	4	8	16

In Listing 1, the code for matrix creation and division is shown. The non-zero matrix is created first and the n/txt submatrices are assigned to the slave structure. This handles the matrix preparation before distributing its parts.

C. The Master Function

In distributed matrix computations using socket programming, the master function takes an essential role. It is responsible for initializing the matrix, distributing computation tasks to slave nodes, and managing communication and synchronization between these nodes.

The program begins by initializing a square matrix of size $matrixSize$, filling it with random integers between 1 and 10. Memory for the matrix is dynamically allocated using malloc. Then the function reads the configuration file $slave\_address.txt$, which contains the IP addresses and port numbers for each slave node. The file is expected to begin with an integer t representing the number of slave nodes. The matrix is then partitioned into t submatrices. If the matrix size is not evenly divided by t, the remainder is added to the last submatrix. Each submatrix is associated with a slave node.

Communication sockets are then created for each slave, and connections are established using the configuration details read from the file. Each successful connection initiates a thread for handling communication with the slave node using a handler function $connection\_handler$.

After all threads have been initiated, the master function waits for all to complete using $pthread\_join$, and then the execution time is calculated and output for evaluation purposes. $Listing1$ shows the code implementation of the master function in the program.

Listing 1: The Master Function

```
void master(int matrixSize) {
    struct timeval begin, end;
    double timeTaken;
int **matrix = (int **)malloc(
   matrixSize * sizeof(int *));
for (int i = 0; i < matrixSize; i++)
   {
    matrix[i] = (int *)malloc(
       matrixSize * sizeof(int));
    for (int j = 0; j < matrixSize; j
       ++)
        matrix[i][j] = rand() % 10 +
            1;
}

FILE *configFile = fopen("
   slave_address.txt", "r");
if (!configFile) {
    printf("Error opening config file
       .\n");
    return;
}

int t;
fscanf(configFile, "%d", &t);
slaves arrayOfSlaves[t];
pthread_t sniffer_thread[t];

int i, j, k;
int submatrices = matrixSize / t;
int remainder = matrixSize % t;
int columnSize, endIndex;

srand(time(NULL));

columnSize = submatrices;
endIndex = submatrices;
for (i = 0; i < t; i++) {
    if (i == t - 1) {
        columnSize += remainder;
        endIndex += remainder;
    }
    arrayOfSlaves[i].ownMatrix = (int
        **) malloc(matrixSize *
       sizeof(int *));
    for (j = 0; j < matrixSize; j++)
       {
        arrayOfSlaves[i].ownMatrix[j]
            = (int *) malloc(
           columnSize * sizeof(int));
        for (k = 0; k < columnSize; k
           ++)
            arrayOfSlaves[i].
               ownMatrix[j][k] =
               matrix[j][k + i *
               submatrices];
    }
    endIndex += submatrices;
    arrayOfSlaves[i].slaveNo = i;
    arrayOfSlaves[i].rows =
       matrixSize;
```

```
        arrayOfSlaves[i].columns =
            columnSize;
    }

    printf("\nMatrix Prepared.\n\n");
    // printMatrix(matrix, matrixSize,
        matrixSize);


    int socket_desc, sock;
    struct sockaddr_in server;
    char ip[100];
    int port;

    gettimeofday(&begin, NULL);

    for (i = 0; i < t; i++) {
        if (fscanf(configFile, "%s %d",
            ip, &port) != 2) {
            printf("Error reading config
                file for slave %d.\n", i);
            break;
        }

        socket_desc = socket(AF_INET,
            SOCK_STREAM, 0);
        if (socket_desc == -1) {
            printf("Could not create
                socket for slave %d\n", i)
                ;
            continue;
        }

        server.sin_addr.s_addr =
            inet_addr(ip);
        server.sin_family = AF_INET;
        server.sin_port = htons(port);

        if (connect(socket_desc, (struct
            sockaddr *)&server, sizeof(
            server)) < 0) {
            printf("Connection to slave %
                d failed\n", i);
            continue;
        }

        printf("Connected to slave %d\n",
            i);

        arrayOfSlaves[i].new_sock =
            malloc(1);
        *arrayOfSlaves[i].new_sock =
            socket_desc;

        pthread_create(&sniffer_thread[i
            ], NULL, connection_handler, (
            void *)&arrayOfSlaves[i]);
    }

    fclose(configFile);

    for (i = 0; i < t; i++) {
        pthread_join(sniffer_thread[i],
            NULL);
    }

    gettimeofday(&end, NULL);
    printf("---End---\n");
    timeTaken =  (end.tv_sec - begin.
        tv_sec) + ((end.tv_usec - begin.
        tv_usec)/1000000.0);
    unsigned long long mill = 1000 * (end
        .tv_sec - begin.tv_sec) + (end.
        tv_usec - begin.tv_usec) / 1000;
    printf("Time taken: %f seconds\n",
        timeTaken);
    printf("Time taken: %llu milliseconds
        \n", mill);

    for (i = 0; i < t; i++) {
        for (j = 0; j < matrixSize; j++)
            free(arrayOfSlaves[i].
                ownMatrix[j]);
        free(arrayOfSlaves[i].ownMatrix);
    }
}
```

D. The Connection Handler on the Master

The connection handler function facilitates the transmission of submatrices from the master to each slave node in a program. By encapsulating the communication logic in a separate thread for each slave node, it enables parallel processing. This function is a crucial component of the master function as it enables communication and coordination between the master and slave nodes.

The connection handler function begins by casting the argument $client$ to the appropriate data type, slaves, which encapsulates information about the slave node. This includes the socket descriptor, assigned submatrix, and other identifying information. It then sends information to the slave node using the write function to send data through the socket connection.

After sending the submatrix, the handler waits to receive an acknowledgment message from the slave node using the $recv$ function. If the acknowledgment is received successfully, it is printed along with the identifier of the slave node. If an error occurs during the reception of the acknowledgment, an error message is printed.

Finally, the socket connection is closed, memory allocated for the socket descriptor is freed, and the thread exits. $Listing 2$ shows the code implementation of the $connection\_handler$ function.

Listing 2: The $connection\_handler$ Function

```
void *connection_handler(void *client) {
```

```
    slaves *slave = (slaves *)client;
    int sock = *(int *)slave->new_sock;
    int read_size;
    int i, j;
    char slaveMessage[256];

    int **matrixToSlave = slave->
       ownMatrix;
    int rows = slave->rows;
    int columns = slave->columns;
    int slaveNo = slave->slaveNo;
    printf("Slave %d entered handler.\n\n
       ", slave->slaveNo);

    memset(slaveMessage, 0, 256);

    write(sock, &rows, sizeof(rows));
    write(sock, &columns, sizeof(columns)
       );
    write(sock, &slaveNo, sizeof(slaveNo)
       );

    for (i = 0; i < rows; i++)
        for (j = 0; j < columns; j++)
            write(sock, &matrixToSlave[i
               ][j], sizeof(matrixToSlave
               [i][j]));

    read_size = recv(sock, slaveMessage,
       256, 0);

    if (read_size < 0) {
        perror("recv ack failed");
    } else {
        printf("Message from Slave %d: %s
           \n", slave->slaveNo,
           slaveMessage);
        printf("Slave %d disconnected.\n\
           n", slave->slaveNo);
        fflush(stdout);
    }

    close(sock);
    free(slave->new_sock);
    pthread_exit(NULL);
}
```

E. The Slave Function

The slave function is responsible for setting up a slave node in a program. It listens for incoming connections from the master node, receives submatrices and other relevant information from the master, and sends acknowledgment messages back to the master upon successful transmission of the submatrices.

The slave function begins by creating a socket for communication with the master node using the socket function. If the socket creation fails, an error message is printed, and the function returns. The function then prepares the *sockaddr_in* structure with information about the server, including the IP address and the port number specified as the function argument.

After preparing the structure, the function binds the socket to the specified port using the bind function. If the bind operation fails, an error message is printed, and the function returns. The function then listens for incoming connections from the master node using the listen function. It waits for the master to initiate a connection and accepts the incoming connection using the accept function.

Once a connection is established, the slave node receives information about the submatrix it is assigned to compute. This includes the number of rows, columns, and the slave node's identifier sent by the master. The function then receives the elements of the submatrix one by one and stores them in a dynamically allocated two-dimensional array.

After receiving the submatrix, the slave node sends an acknowledgment message to the master node using the write function. Finally, the dynamically allocated memory for the received submatrix is freed, and the socket connections are closed. Listing 3 shows the code implementation of the slave function.

Listing 3: The Slave Function

```
void slave(int port) {
    int sock, client_sock, c, i, j;
    struct sockaddr_in server, client;
    char ackMessage[256] = "ack";
    int **receivedMatrix, rows, columns,
       slaveNo;

    // Create socket
    sock = socket(AF_INET , SOCK_STREAM ,
        0);
    if (sock == -1) {
        perror("Could not create socket")
           ;
        return;
    }
    printf("Socket creation successful.\n
       ");


    // Prepare the sockaddr_in structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(port);

    // Bind
    if (bind(sock, (struct sockaddr *)&
       server, sizeof(server)) < 0) {
        perror("Bind failed");
        return;
    }
    printf("Bind successful.\n");

    // Listen
```

```
listen(sock, 3);
printf("Waiting for master to
    initiate connection...\n");

// Accept incoming connection
c = sizeof(struct sockaddr_in);
client_sock = accept(sock, (struct
    sockaddr *)&client, (socklen_t *)&
    c);
if (client_sock < 0) {
    perror("Accept failed");
    return;
}
printf("Connection accepted.
    Receiving data from master...\n");

// Receive rows and columns, and
    slaveNo from master
if (recv(client_sock, &rows, sizeof(
    rows), 0) < 0) {
    perror("recv rows failed");
    return;
}
if (recv(client_sock, &columns,
    sizeof(columns), 0) < 0) {
    perror("recv columns failed");
    return;
}
if (recv(client_sock, &slaveNo,
    sizeof(slaveNo), 0) < 0) {
    perror("recv slaveNo failed");
    return;
}

// Receive elements one by one (((
    inefficiently)))
receivedMatrix = (int **)malloc(rows
    * sizeof(int *));
for (i = 0; i < rows; i++) {
    receivedMatrix[i] = (int *)malloc
        (columns * sizeof(int));
    for (j = 0; j < columns; j++) {
        if (recv(client_sock, &
            receivedMatrix[i][j],
            sizeof(receivedMatrix[i][j
            ]), 0) < 0) {
            perror("recv Matrix
                failed");
            printf("Error @ [%d][%d]\
                n", i, j);
            return;
        }
    }
}

printf("Matrix received successfully
    .\n");

    // Send acknowledgment to master
    write(client_sock, ackMessage, strlen
        (ackMessage) + 1);
    printf("Acknowledgment sent to master
        .\n");

    // Clean up
    for (i = 0; i < rows; i++) {
        free(receivedMatrix[i]);
    }
    free(receivedMatrix);

    close(client_sock);
    close(sock);
}

}
```

F. Running the Program

The program takes three arguments to execute it. The first argument n specifies the $n \times n$ dimension of the matrix. The second argument is the port number on which the slave nodes listen for the master node to initiate connection. The last argument s specifies the mode of operation, where 0 indicates the master node and 1 indicates the slave node.

The main program handles the different executions based on the mode specified in the argument. Moreover, in case of an error in specifying the arguments, the program will print a prompt and exit. Listing 4 shows the code for the main function.

Listing 4: The Main Function

```
int main(int argc , char *argv[]) {
    if (argc != 4) {
        printf("Usage: %s [n] [port] [s]\
            n", argv[0]);
        return EXIT_FAILURE;
    }

  int n = atoi(argv[1]);
  int port = atoi(argv[2]);
  int s = atoi(argv[3]);

  if (s == 0)
    master(n);
  else if (s == 1)
    slave(port);
  return 0;
}
```

G. Core-Affine Program

The experimentation includes the analysis when the program is core affined. In the program, the master function assigns the POSIX threads to the specific core using a function that sets its core affinity. Listing 5 shows this function.

Listing 5: The $set\_affinity$ Functionn

```
void set_affinity(int cpu_core) {
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu_core, &cpuset);

    if (pthread_setaffinity_np(
        pthread_self(), sizeof(cpu_set_t),
         &cpuset) != 0) {
        perror("pthread_setaffinity_np");
    }
}
```

H. Communication Technique

The program utilizes a one-to-many broadcast pattern, where the master process communicates individually with each slave process. The master process initializes a socket connection with each slave process individually. It reads the addresses of the slave processes from a configuration file and establishes connections with them one by one. Once connected, the master sends the submatrices to each slave process over their respective connections.

In the slave process, each slave waits for the master to initiate a connection and then accepts the incoming connection request. Upon connection establishment, the slave process receives the required data from the master. After receiving the matrix data, each slave process sends an acknowledgment message back to the master to confirm successful reception. Upon receiving acknowledgment from all slaves, the master concludes the communication process.

One-to-many broadcast pattern may not be the most efficient for large-scale matrix distribution tasks. This approach can lead to increased overhead due to the management of multiple individual connections between the master and each slave.

III. Results and Discussion

A. One Machine with t Terminals

The result of the experiment on running the program on one machine with t terminals is shown in Table 2. The graph of the average runtime of the program is shown in Figure 1.

Analyzing the results, the program stops improving at $t = 4$ for all n. The average runtime decreases from $t = 2$ to $t = 4$. However, on $t > 4$, the average runtime increases. It suggests that as the terminal increases when $t > 4$, the average runtime also increases. This increase in the average runtime of the program can be explained by the computational cost of having more slaves. More slaves means that there is more work for the master in terms of thread creation and memory allocation.

Moreover, it is observed that as matrix dimension n increases the average runtime also increases. The trend of the data in the average runtime is similar for all n dimensions as shown in Figure 1.

TABLE 2: Runtime in One Machine with t Terminals

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
20000	2	114.2395	114.3873	115.8518	114.8262
20000	4	109.3701	102.6827	113.0622	108.3717
20000	8	148.3733	148.1747	144.9551	147.1677
20000	16	132.2522	147.6856	161.1572	147.0317
25000	2	194.2551	185.0295	190.6248	189.9698
25000	4	168.6061	174.7412	167.9323	170.4265
25000	8	267.1237	253.4560	247.0042	255.8613
25000	16	240.6956	251.0457	250.2108	247.3174
30000	2	284.0401	297.8693	292.4959	291.4684
30000	4	256.2917	240.1175	260.9648	252.4580
30000	8	326.7611	343.6232	347.1583	339.1809
30000	16	325.9949	355.2153	328.6096	336.6066

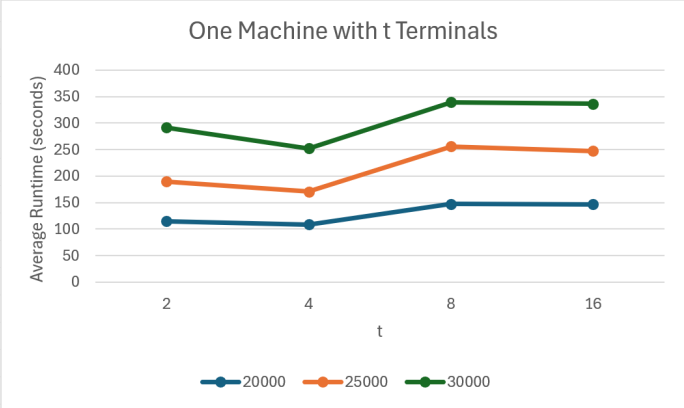

Fig. 1: Average Runtime on One Machine with t Terminals

B. One Machine with t Terminals (core-affine)

The result of the experiment on running the core-affine program on one machine with t terminals has similarity based on the behavior of the graph. Table 3 shows the result of the experiment. The graph of the average runtime of the program is also shown in Figure 2.

Similar to the first experiment, the core-affine program stops improving at $t = 4$ for all n. The average runtime decreases from $t = 2$ to $t = 4$ but is observed to increase on $t > 4$. This increase in the average runtime of the program can also be explained by the computational cost of having more slaves. More slaves means that there is more work for the master in terms of thread creation and memory allocation.

On $n = 3000$, a decrease in runtime is observed on using 16 terminals. However, both $n = 20000$ and $n = 25000$ increase the runtime when using 16 terminals. The variation in the result may be caused by the unpredictable result when $t > 4$.

Moreover, it is observed that the core-affine program increases in the average runtime compared to the first experiments. This increase in the run time can be explained by the additional computational cost brought by the core assignment of threads. For both experiments, as the matrix dimension n increases the average runtime also increases.

C. On t Machines

The result of the experiment on the program on t machines with t terminals has different behavior compared to the first

TABLE 3: Runtime of Core-affined Program in One Machine with t Terminals

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
20000	2	124.6188	128.5223	125.9825	126.3745
20000	4	116.0175	117.2423	118.0324	117.09747
20000	8	138.8314	137.3246	138.2341	138.1300
20000	16	146.5563	168.6782	155.1548	156.7964
25000	2	212.1149	210.8594	211.3539	211.4427
25000	4	192.6260	193.8993	192.0485	192.8580
25000	8	240.5214	232.2963	237.2942	236.7040
25000	16	244.4268	249.8720	247.6584	247.3191
30000	2	329.3263	330.7483	328.4825	329.5190
30000	4	281.8900	280.4729	282.9764	281.7798
30000	8	385.6977	372.6138	376.2941	378.2019
30000	16	359.4947	354.9379	356.4893	356.97406

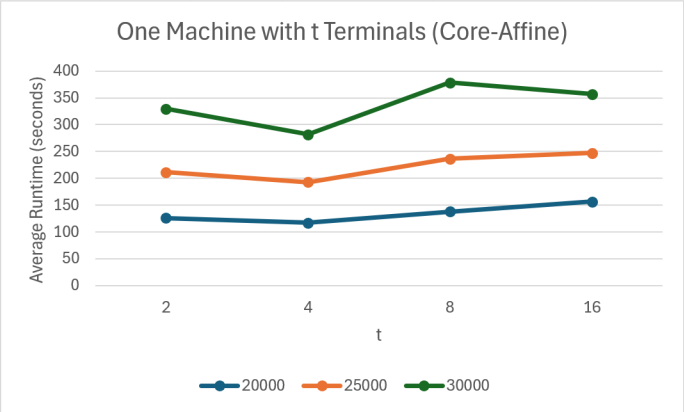

Fig. 2: Average Runtime on One Machine with t Terminals (Core-Affine)

TABLE 4: Runtime in t Machines

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
20000	2	181.7976	182.3565	186.0487	183.4010
20000	4	101.9719	101.9738	101.9695	101.9717
20000	8	118.9677	118.9663	118.9562	118.9634
20000	16	118.9760	119.2365	118.9736	119.0621
25000	2	288.7171	280.6176	287.2586	285.5311
25000	4	163.7737	159.3314	159.3303	160.8118
25000	8	185.8812	185.8822	185.8823	185.8819
25000	16	185. 9584	186.2674	186.1204	186.1939
30000	2	416.9716	419.6505	410.3890	415.6704
30000	4	229.4351	229.4300	229.4285	229.4312
30000	8	267.6673	267.5264	267.6532	267.6156
30000	16	267.6715	267.8385	268.0912	267.8670

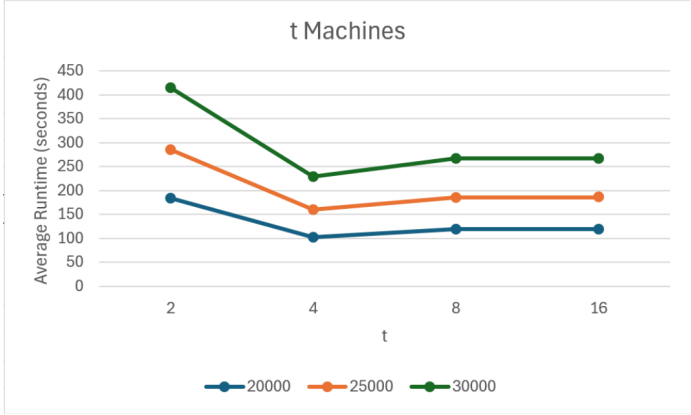

Fig. 3: Average Runtime on t Machines

two experiments. Table 4 shows the result of the experiment. The graph of the average runtime of the program is also shown in Figure 3.

In the result, it is observed that the average runtime decreases as t machines increase for $t < 4$. The highest runtime of the program is observed on $t = 2$ and the lowest runtime is observed on $t = 4$. The average runtime slightly increases at $t > 4$ but is still lower compared to the average runtime at $t = 2$.

In comparison to the two previous experiments, the overall average runtime of the program improves as t increases. This can be explained by the use of several computing nodes each with dedicated computational processing. Unlike the two previous experiments which run on a CPU with four cores, this experiment uses the dedicated CPUs on t machines.

For all the experiments, as the matrix dimension n increases, the average runtime also increases. This can be observed based on the line trend on the graphs for each of the experiments.

IV. Conclusion

In this study, a program was developed in C programming language to distribute parts of the matrix over sockets. The one-to-many broadcast is used as the communication technique. The performance of the program is analyzed on a machine with different terminals, on a machine with different terminals but the program is core-affine, and on different machines.

On running the program on one machine with t terminals, the program exhibits an increase in runtime beyond $t = 4$ terminals This increase is attributed to the computational overhead associated with managing additional slave processes. Across different matrix dimensions n, it is observed that increasing n leads to higher average runtimes.

Similar to the first experiment, the core-affine version of the program also stops improving beyond $t = 4$ terminals, with the average runtime increasing for $t > 4$ due to increased computational workload. Notably, while some fluctuations are observed in runtime for different values of n and t, there is an overall increase in runtime compared to the standard version of the program. This is attributed to the additional computational cost incurred by core assignment.

On using t machines to run the program, the program's runtime improves as the number of machines t increases, particularly for $t < 4$. This improvement is attributed to the utilization of multiple computing nodes, each with dedicated processing capabilities.

Despite variations in runtime across different values of t, the trend of increasing average runtime with higher matrix dimensions n remains consistent across all experiments.

References

[1] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, "Trends in big data analytics," *Journal of Parallel and Distributed Computing*, vol. 74, no.

7, pp. 2561-2573, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514000057. [Accessed: May 11, 2024].

[2] D. Sitaram and G. Manjunath, "Chapter 9 - Related Technologies" in *Moving To The Cloud*, D. Sitaram and G. Manjunath, Eds. Boston: Syngress, 2012, pp. 351-387. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781597497251000093.

[3] Z. Saif, "Let's talk about the process to process communication," Medium, [Online]. Available: https://medium.com/@zakariasaif/lets-talk-about-the-process-to-process-communication-965f3ffbf3a8. [Accessed: May 11, 2024].

[4] Z. Bai, T. Hiraishi, A. Ida, and M. Yasugi, "Parallelization of matrix partitioning in hierarchical matrix construction on distributed memory systems," *Journal of Information Processing*, vol. 30, no. 0, pp. 742–754, 2022. doi:10.2197/ipsjjip.30.742.

[5] L. Belcastro, R. Cantini, F. Marozzo, A. Orsino, D. Talia, and P. Trunfio, "Programming big data analysis: principles and solutions," *Journal of Big Data*, vol. 9, no. 1, p. 4, 2022. doi:10.1186/s40537-021-00555-2. [Online]. Available: https://doi.org/10.1186/s40537-021-00555-2.

John Rommel B. Octavo is currently pursuing a Bachelor of Science degree in Computer Science at the University of the Philippines Los Baños. His hobbies include cycling, playing piano and recorder flute, watching movies, reading religious literature, and programming. He dreams to be a software developer and fulfill his priestly vocation in the future.