Distributed Computation of the Pearson Correlation Coefficient

John Rommel B. Octavo

***Abstract*—This study evaluates the performance of a parallel program that distributes the computation of the Pearson Correlation Coefficient across multiple machines using socket programming. The research aimed to develop a parallel program for distributing computation and to analyze its performance in terms of computational and communication costs.**

The experiments used multiple computers with Ubuntu 22.04 LTS, Intel® Core™ i7 processors, and 16.0 GB DDR4 memory. The C program included a master function to initialize, partition, and distribute a matrix to slave nodes. Each slave node computed the Pearson Correlation Coefficient for its submatrix and sent the results back to the master.

The program's performance was measured in terms of time elapsed during computation and communication. The analysis showed that the program's efficiency improved as the number of machines increased to four. Beyond this point, the running time began to increase slightly due to the communication overhead. The computational cost within the slave processes decreased as the number of machines increased, with the most significant improvement at sixteen machines. However, the parallel implementation showed high overhead and lower efficiency compared to the serial implementation, especially when the number of machines exceeded four. Larger matrix dimensions improved the program's speedup and efficiency, suggesting that parallelization benefits larger problem sizes.

While the parallel implementation improved performance up to a certain point, the communication overhead limited further gains. The program was not cost-optimal compared to the serial implementation, indicating the need for balancing the number of machines and the problem size for effective parallelization.

***Index Terms*—Pearson Correlation Coefficient, socket programming, matrix distribution, superlinearity, parallel cost, parallel efficiency, parallel speedup**

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. Socket programming facilitates the communication between processes over a network. It handles the distribution of data over the network for parallel computation.

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 interprocess 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 study will focus on the computation of the Pearson correlation coefficients of a matrix and a vector using socket programming to distribute parts of the matrix for parallel computation. The study aims to analyze the performance in terms of the time cost of both communication and computation.

The Pearson Correlation Coefficient, often denoted as r, is a statistical measure that quantifies the strength and direction of the linear relationship between two variables [6]. It is a dimensionless value that ranges from -1 to 1, where $r = 1$ indicates a perfect positive linear relationship, $r = -1$ indicates a perfect negative linear relationship, and $r = 0$ indicates no linear relationship between the variables.

In this research, we will use the formula for the Pearson Correlation Coefficient shown in Equation 1. It consists of an $m \times n$ matrix X with m rows and n columns, a $m \times 1$ vector y, and a $1 \times n$ vector r that represents the Pearson Correlation Coefficient of the columns X and y.

$$r(j) = \frac{m[X_j y] - [X]_j[y]}{[\sqrt{m[X^2]_j - ([X])^2} \cdot \sqrt{m[y^2] - ([y])^2}} \tag{1}$$

where,

$$[X]_j = \sum_{i=1}^{m} X(i,j) = X(1,j) + X(2,j) + \cdots + X(m,j),$$
$$[X^2]_j = \sum_{i=1}^{m} X(i,j)^2 = X(1,j)^2 + X(2,j)^2 + \cdots + X(m,j)^2,$$
$$[y] = \sum_{i=1}^{m} y(i) = y(1) + y(2) + \cdots + y(m),$$
$$[y^2] = \sum_{i=1}^{m} y(i)^2 = y(1)^2 + y(2)^2 + \cdots + y(m)^2,$$
$$[X_j y] = \sum_{i=1}^{m} X(i,j)y(i) = X(1,j)y(1) + \cdots + X(m,j)y(m).$$

A. Objectives of the Study

This study aims to assess the performance of a program that uses sockets to distribute the computation of the Pearson Correlation Coefficient over the network. It specifically focuses on the use of multiple machines to handle parallel computation.

The specific objectives of this research study are the following:

1) To develop a threaded and core-affine program that will distribute the computation of the Pearson Correlation Coefficient over the network and to multiple machines; and,
2) To analyze the performance in terms of the program in terms of the computational cost of communication with other machines and the computation of the Pearson Correlation Coefficient.

II. Methodology

A. Development Tools

The experiments for the distribution of the parts of the matrix using sockets will be conducted on multiple computers. The specifications of the computers are 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 $n \times n$ matrix M and it is divided into t submatrices of size $n/t \times n$.

In Listing 1, the code for matrix creation and division is shown. The non-zero matrix is created first and the $n/t \times t$ 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 $pthread\_join$. The master rebuilds the full vector r. The execution time is calculated and printed in the terminal for evaluation purposes. $Listing 1$ 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;

    float **matrix = (float **)malloc(
       matrixSize * sizeof(float *));
    for (int i = 0; i < matrixSize; i++)
    {
        matrix[i] = (float *)malloc(
            matrixSize * sizeof(float));
        for (int j = 0; j < matrixSize; j
            ++)
            matrix[i][j] = rand() % 10 +
                1;
    }

    float *y = (float *)malloc(matrixSize
        * sizeof(float));
    for (int i = 0; i < matrixSize; i++)
    {
        y[i] = 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 submatrices = matrixSize / t;
int remainder = matrixSize % t;
int columnSize, endIndex;

columnSize = submatrices;
endIndex = submatrices;

for (int i = 0; i < t; i++)
{
    if (i == t - 1)
    {
        columnSize += remainder;
        endIndex += remainder;
    }
    arrayOfSlaves[i].submatrix = (
        float **)malloc(columnSize *
        sizeof(float *));
    for (int j = 0; j < columnSize; j
        ++)
    {
        arrayOfSlaves[i].submatrix[j]
            = (float *)malloc(
            matrixSize * sizeof(float)
            ); // Transposed
            allocation
        for (int k = 0; k <
            matrixSize; k++)
            arrayOfSlaves[i].
                submatrix[j][k] =
                matrix[j + i *
                submatrices][k]; //
                Transposed indexing
    }
    arrayOfSlaves[i].y = y;
    arrayOfSlaves[i].slaveNo = i;
    arrayOfSlaves[i].rows =
        columnSize;

        // rows and columns swapped
    arrayOfSlaves[i].cols =
        matrixSize;

        // rows and columns swapped
    arrayOfSlaves[i].result = (float
        *)malloc(columnSize * sizeof(
        float)); // Result size
        matches row size
}

printf("\nMatrix and vector y
    prepared.\n\n");

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

int num_cores = sysconf(
    _SC_NPROCESSORS_ONLN);
int core_counter = 0;

gettimeofday(&begin, NULL);

for (int i = 0; i < t; i++)
{
    if(core_counter == 8){
        core_counter = 0;
    }

    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(sizeof(int));
```

```
        *arrayOfSlaves[i].new_sock =
            socket_desc;

        //Assign thread to a specific
            core
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_SET(core_counter, &cpuset);


        pthread_create(&sniffer_thread[i
            ], NULL, connection_handler, (
            void *)&arrayOfSlaves[i]);
        pthread_setaffinity_np(
            sniffer_thread[i], sizeof(
            cpu_set_t), &cpuset);

        core_counter++;
    }

    fclose(configFile);

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

    // Aggregate results
    float *final_result = (float *)malloc
        (matrixSize * sizeof(float));
    for (int i = 0; i < t; i++)
    {
        int startIdx = i * submatrices;
        int rows = arrayOfSlaves[i].rows;
        for (int j = 0; j < rows; j++)
        {
            final_result[startIdx + j] =
                arrayOfSlaves[i].result[j
                ];
        }
    }

    gettimeofday(&end, NULL);
    printf("\n---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);

    // Free Alocated Memory
    for (int i = 0; i < t; i++)
    {
        for (int j = 0; j < arrayOfSlaves
            [i].rows; j++)
            free(arrayOfSlaves[i].
                submatrix[j]);
        free(arrayOfSlaves[i].submatrix);
        free(arrayOfSlaves[i].result);
    }

    free(y);
    free(final_result);
}
```

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 the resulting matrix and an acknowledgment message from the slave node using the $recv$ function. 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];

    float **matrixToSlave = slave->
        submatrix;
    float *y = slave->y;
    int rows = slave->rows;
    int cols = slave->cols;
    int slaveNo = slave->slaveNo;
    printf("Slave %d entered handler.\n\n
        ", slave->slaveNo);

    memset(slaveMessage, 0, 256);

    write(sock, &rows, sizeof(rows));
    write(sock, &cols, sizeof(cols));
```

```
    write(sock, &slaveNo, sizeof(slaveNo)
        );

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

    for (i = 0; i < cols; i++)
        write(sock, &y[i], sizeof(y[i]));

    // Receive result from slave
    for (i = 0; i < rows; i++) {
        if (recv(sock, &slave->result[i],
             sizeof(slave->result[i]), 0)
            < 0) {
            perror("recv result failed");
            close(sock);
            free(slave->result);
            free(slave->new_sock);
            pthread_exit(NULL);
        }
    }

    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);
    }

    free(slave->new_sock);

    // Close socket
    close(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, computes for the Pearson correlation coefficient of the submatrix, and sends acknowledgment messages and results back to the master upon successful transmission and computation of the submatrix.

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. The slave node also receives the vector y which is essential in the computation of the Pearson correlation coefficient.

After receiving the submatrix and the vector y, the slave node computes the Pearson correlation coefficient of the submatrix and the vector y. It then sends the result and an acknowledgment message to the master node after successful computation. 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";
    float **receivedMatrix, *y;
    int rows, cols, slaveNo;

    struct timeval begin, end;
    double timeTaken;

    // 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, &cols, sizeof(
    cols), 0) < 0)
{
    perror("recv cols failed");
    return;
}
if (recv(client_sock, &slaveNo,
    sizeof(slaveNo), 0) < 0)
{
    perror("recv slaveNo failed");
    return;
}

// Receive matrix
receivedMatrix = (float **)malloc(
    rows * sizeof(float *));
for (i = 0; i < rows; i++)
{
    receivedMatrix[i] = (float *)
        malloc(cols * sizeof(float));
    for (j = 0; j < cols; 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;
        }
    }
}

// Receive vector y
y = (float *)malloc(cols * sizeof(
    float));
for (i = 0; i < cols; i++)
{
    if (recv(client_sock, &y[i],
        sizeof(y[i]), 0) < 0)
    {
        perror("recv y failed");
        return;
    }
}


printf("Matrix and vector y received
    successfully.\n\n");

float **transposedMatrix =
    transposeMatrix(receivedMatrix,
    rows, cols);

float *result = (float *)malloc(rows
    * sizeof(float));
ThreadData threadData = {
    transposedMatrix, y, cols, rows,
    slaveNo, result};

cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(0, &cpuset);

gettimeofday(&begin, NULL);

pthread_t pearson_thread;
pthread_create(&pearson_thread, NULL,
    pearson_cor, (void *)&threadData)
    ;
pthread_setaffinity_np(pearson_thread
    , sizeof(cpu_set_t), &cpuset);
pthread_join(pearson_thread, NULL);

gettimeofday(&end, NULL);

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;
    mill = 1000 * (end.tv_sec - begin.
        tv_sec) + (end.tv_usec - begin.
        tv_usec) / 1000;
    printf("\nComputation time taken: %f
        seconds\n", timeTaken);
    printf("Computation time taken: %llu
        milliseconds\n", mill);

    // Send result back to master
    for (i = 0; i < rows; i++)
    {
        write(client_sock, &result[i],
            sizeof(result[i]));
    }

    write(client_sock, ackMessage, strlen
        (ackMessage) + 1);
    printf("\nAcknowledgment sent to
        master.\n");

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

    for (int i = 0; i < rows; i++) {
        free(transposedMatrix[i]);
    }
    free(transposedMatrix);

    free(y);
    free(result);

    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 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, the slave computes the Pearson correlation coefficient. Then, each slave process sends the resulting vector and an acknowledgment message back to the master to confirm

successful computation. Upon receiving acknowledgment from all slaves, the master concludes the communication process and rebuilds the resulting matrix from the slave processes.

Since the master communicates with individual slaves, the program does not implement the basic communication operations. Specifically, it does not implement the one-to-many broadcast (1MB), one-to-many personalized broadcast (1MPB), and many-to-many personalized broadcast (MMPB).

If the program implements the basic communication operation, the master need not communicate with all slaves since slaves can send data to other slaves. Moreover, the communication time may be improved if the communication operation mentioned above is implemented in the program.

III. Results and Discussion

A. Communication and Computational Cost of the Program

The communication and computational cost in terms of time of the program is calculated within the master process. The result of the experiment in distributing the computation of the Pearson Correlation Coefficient of a matrix and a vector is shown in Table 1. The three-dimensional plot of the result is shown in Figure 1.

Analyzing the results, the program improves as the number of machines t is increased. It is observed that the fastest the program runs is at $t = 4$. The running time in $t > 4$ is relatively lower than at $t = 2$. However, as the t increases at $t > 4$, a slight increase in the average running time is observed.

The increasing running time as the t machines increase can be explained by the increasing communication time. More machines mean more effort for the master process to handle more communication with other machines.

Moreover, the n dimension of the matrix also affects the running time of the program. The average running time increases as the n dimension of the matrix increases. This can be observed in Figure 1 where each color represents n dimensions of the matrix.

TABLE 1: Time Elapsed as Reported by the Master Process Only

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
20000	2	200.0479	200.6473	200.8452	200.5135
20000	4	110.1352	110.2494	110.5739	110.3195
20000	8	127.6167	127.7935	127.5420	127.6508
20000	16	131.8125	131.7957	131.8035	131.8039
25000	2	312.5862	312.5472	312.8462	312.6599
25000	4	191.4337	191.8379	191.2650	191.5122
25000	8	199.8973	199.8736	199.7482	199.8397
25000	16	206.1124	206.1546	206.1337	206.1336
30000	2	451.2836	451.9463	451.7285	451.6528
30000	4	247.5710	247.5827	247.9274	247.6937
30000	8	289.3262	289.4924	289.2482	289.3556
30000	16	297.4054	297.2850	297.3325	297.3410

B. Computation Cost of the Program in the Slave Process

The computational cost in the individual computation of the Pearson correlation coefficient is calculated within each of the slave processes. The result of the experiment in the computation of the Pearson Correlation Coefficient of the submatrix and a vector in the slave processes is shown in Table 2. The three-dimensional plot of the result is shown in Figure 2.

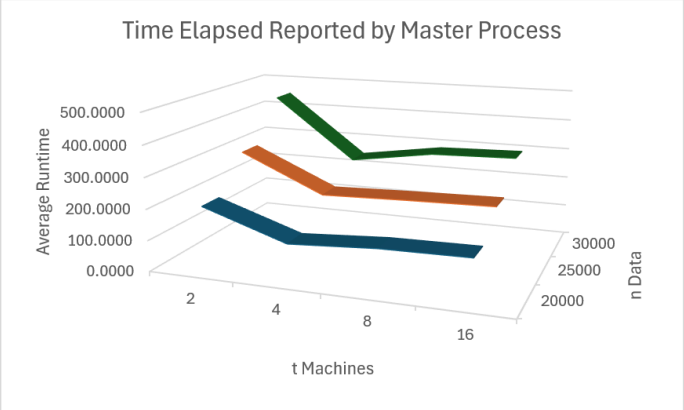

Fig. 1: Time Elapsed Reported by the Master Process

In each run, the highest running time among the slave processes is noted. The highest running time represents the actual computational cost of the program. Note here that the master process cannot continue to progress in rebuilding the result vector unless all the machines finish the computation of the individual submatrix assigned to them. Noting the highest running time among the slave processes accurately measures the computational time.

Analyzing the results, the pattern in Figure 2 is similar to the pattern in Figure 1. The program improves as the number of machines t increases (up to $t = 16$). It is observed that the fastest the slave process runs is at $t = 16$ which is the highest t in the experiment. The running time of the program in the slave process generally decreases from $t = 2$ to $t = 16$. However, for all cases of n, the average run time increases at $t = 8$. This slight increase in runtime at $t = 8$ can be caused by cache misses, threading, core assignments, and the like.

Moreover, the n dimension of the matrix also affects the running time of the program. The average running time increases as the n dimension of the matrix increases. This can be observed in Figure 1 where each color represents n dimensions of the matrix. At $t = 16$, the rate of increase in the average runtime is lower when n is increased. This could mean more t machines can improve the computational time in each slave process.

C. Serial Versus Parallel Implementation

The performance metrics of the implementation of the parallel distributed Pearson Correlation Coefficient are shown in Table 3. The serial time T_s, parallel overhead T_o, parallel speedup S, parallel efficiency E, and parallel cost pT_p are presented in the table. These data are essential in comparing the performance of serial versus the parallel implementation of the Pearson Correlation Coefficient.

The runtime in the serial program T_s is obtained in the serial computation of the Pearson Correlation Coefficient of a matrix and a vector. This is the result obtained from the first

TABLE 2: Time Elapsed as Reported by the Slave Processes

n	t	Time Elapsed (seconds)			Average Runtime
		Run 1	Run 2	Run 3	
20000	2	10.9202	10.9319	10.4793	10.7772
20000	4	5.3786	5.3957	5.8780	5.5508
20000	8	8.2598	8.9742	8.1645	8.4662
20000	16	4.0760	4.0691	4.0575	4.0675
25000	2	17.5905	17.9372	17.2402	17.5893
25000	4	8.7054	8.7849	8.5473	8.6792
25000	8	13.2698	13.3749	13.1942	13.2796
25000	16	6.6012	6.6742	6.6539	6.6431
30000	2	25.0644	25.8492	25.2467	25.3868
30000	4	13.3534	13.6842	13.1747	13.4041
30000	8	20.2834	21.4823	20.8274	20.8644
30000	16	9.9521	9.8223	9.9238	8.8994

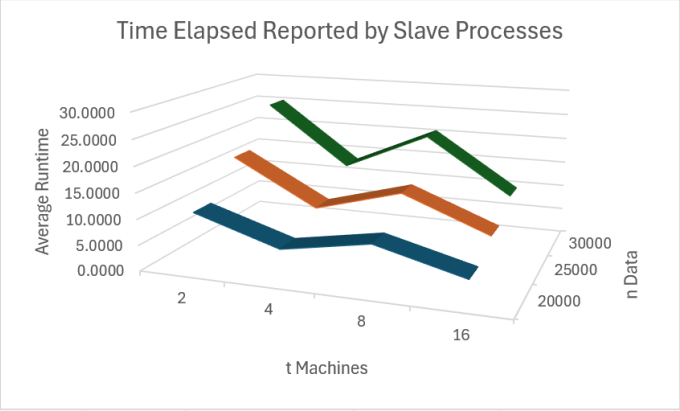

Fig. 2: Time Elapsed Reported by Slave Processes

laboratory activity. The runtime is constant for varying t since the serial implementation on the first laboratory activity does not use treads, core affinity, or other machines. In terms of n-dimensions of the matrix, the serial time T_s is observed to increase as n increases.

The parallel overhead T_o is computed using the formula $p \times T_p$–T_s. It is the difference between the total time collectively spent by all the processing elements p and the serial time T_s. In Table 3, it is observed that the parallel overhead is significantly high. As t machines and n-dimension of the matrix increases, the parallel overhead also increases. This means that the parallel implementation costs a significant amount of overhead time.

On the speedup of the parallel implementation, it is observed that the value is below one. Parallel speed-up is computed using the formula $S = T_s/T_p$. This means that the parallel implementation is not as fast as the serial implementation of the Pearson Correlation Coefficient. It is observed that the speedup S increases as t increases for $t <= 4$. However, at $t > 4$ decrease in speedup is observed as t increases. Moreover, as n increases, the speedup also increases.

Similar to the result of the speed-up, the result for the efficiency is low. Parallel efficiency is computed by the formula $E = S/p$, where S is the speed up and p is the number of processors ($p = t$). The result shows that the parallel implementation is not as efficient as the serial counterpart. It is observed that as t machines increases, the efficiency decreases. However, a direct proportionality is observed in terms of n and E such that as n-dimensions of the matrix increase, efficiency E also increases.

On the cost of the parallel implementation, it is observed that the value is significantly high. This means that the parallel implementation of the Pearson Correlation Coefficient incurs more cost in terms of time. The parallel cost is computed by $p \times T_p$, where p is the number of processors ($p = t$) and T_p is the running time of the parallel program. In the table, as t machines and n-dimensions of the matrix increase, the parallel cost also increases.

The comparison of the serial and parallel implementation of the Pearson Correlation Coefficient proves that more work is done on parallel implementation compared to the serial counterpart. This means that there is no superlinearity observed in the parallel implementation. Moreover, since the parallel cost pT_p, is not asymptotically identical to the serial cost T_s, the parallel implementation of the Pearson Correlation Coefficient is not cost-optimal.

TABLE 3: Performance metrics of the parallel distributed Pearson Correlation Coefficients

n	t	T_s	Parallel			
			T_o	S	E	pT_p
20000	2	17.2992	383.7277	0.0863	0.0431	401.0269
20000	4	17.2992	423.9788	0.1568	0.0392	441.2780
20000	8	17.2992	1003.9068	0.1355	0.0169	1021.2060
20000	16	17.2992	2091.5627	0.1312	0.0082	2108.8619
25000	2	30.6970	594.6228	0.0982	0.0491	625.3198
25000	4	30.6970	735.3518	0.1603	0.0401	766.0488
25000	8	30.6970	1568.0207	0.1536	0.0192	1598.7177
25000	16	30.6970	3267.4399	0.1489	0.0093	3298.1369
30000	2	47.4731	855.8324	0.1051	0.0526	903.3055
30000	4	47.4731	943.3018	0.1917	0.0479	990.7749
30000	8	47.4731	2267.3717	0.1641	0.0205	2314.8448
30000	16	47.4731	4709.9826	0.1597	0.0100	4757.4557

IV. Conclusion

In this study, a parallel distributed program was developed in C programming language to compute the Pearson Correlation Coefficient. The distribution of submatrices on different machines was handled through socket programming. The performance was analyzed by getting the elapsed time in the master process and slave processes. The performance metrics were also used to compare the performance of the serial and parallel implementation of the Pearson Correlation Coefficient.

The analysis of computation and communication costs, measured as the time elapsed in the master process, indicates that the program's efficiency improves with an increase in the number of machines up to $t = 4$. Beyond this point, the running time begins to increase slightly, which can be attributed to the overhead involved in managing communication between a larger number of machines. This suggests that parallelization is effective up to a certain threshold, beyond which the benefits are offset by the increased communication overhead. Similarly, the computational cost within slave processes shows a decreasing trend as the number of machines increases. The most significant improvement was observed at $t = 16$.

Comparing the serial and parallel implementations reveals several critical points. The parallel overhead To is significantly

high and increases with both the number of machines and the matrix dimensions n. This indicates substantial communication and synchronization costs in the parallel implementation. Furthermore, both speedup S and efficiency E metrics are low, with speedup peaking at $t = 4$ and declining as the number of machines increases. This trend highlights the inefficiencies introduced by parallelization, particularly for $t > 4$, where communication overhead outweighs the benefits of additional parallel computation. However, an increase in matrix dimension n improves both speedup and efficiency, suggesting that larger problem sizes benefit more from parallelization.

The parallel cost pT_p is significantly higher than the serial cost T_s, indicating that the parallel implementation incurs more time-related expenses. This finding reveals that the parallel implementation is not cost-optimal and performs more work compared to the serial counterpart.

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.

[6] D. L. Hahs-Vaughn, "Foundational methods: Descriptive statistics: Bivariate and Multivariate Data (correlations, associations)," *International Encyclopedia of Education(Fourth Edition)*, pp. 734–750, 2023. Available: https://doi.org/10.1016/b978-0-12-818630-5.10084-3.

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.