Distributing Parts of a Matrix over Sockets

Jefferson Louise Basilio[1]*

Abstract
An entirely new but related exercise from previous ones. 'Processors' or computers that run the program are classified into two: *master* and *slave*. User is made to input *t* slaves. The *master* sends *t* sub-matrices to *slaves*, then each slave sends back an acknowledgement message. In order to handle multiple slaves, threading is implemented. POSIX Thread Library is used for the threads. Libraries allowing socket programming are also imported. The program is ran into two different ways, one is on one PC running *t* instances of the program, and the other is ran on different *t* computers instead of terminals. Results show that the latter ways is relatively faster.

[1] *Institute of Computer Science, University of the Philippines Los Baños*
*jcbasilio1@up.edu.ph

Contents

Introduction 1
Objectives 1
1 Methodology 1
1.1 Slave model 2
1.2 Starting the program 2
1.3 Dividing the matrix 2
1.4 Creating sockets 2
1.5 Thread function 3
1.6 Running as a slave 3
2 Results and Discussion 3
2.1 One machine with *t* terminals 3
2.2 On *t* machines 4
3 Conclusion 4
4 List of Collaborators 4
5 References 4
6 Appendices 5
6.1 *master()* function 5
6.2 Thread function 5
6.3 *slave()* function 5

Introduction

Socket programming is relatively new to the students, especially those who have not taken *CMSC 137: Data Communications and Networking* and *CMCS 100: Web Programming* in the previous curriculum of ICS-UPLB. Sending data over open sockets is more of a self-learning material as the topic is not heavily taught, thus softly requiring the students to allocate time to learn what the code snippets they find on the internet do. Upon learning, the codes found will serve as a baseline, modifying them so that the output would be fit to the required results of the problem given. Given that there are other requirements that the students must submit, it took two weeks to finally make a valid, working program. This program sends an *n x n/t*-sized sub-matrices over *t* processors. The said processors can be either different computers or one computer with *t* terminals running. In this case, both ways are done. Note that *slaves* and *threads* in this paper may be used interchangeably.

Objectives

The objective of this exercise is to determine the run-time of the program by distributing the parts of a matrix into different processors. Additionally, this also serves as a material or challenge to learn how sockets work, and be able to implement a working program infused with sockets.

1. Methodology

For this exercise, there will be two main observations, namely:

1. Running the program on different machines with same specifications, shown below:
   - Processor - Intel Core i3-3570
   - Memory - 8GB DDR3 1333MHz
2. Running the program on one machine with different specifications from *1*, shown below:
   - Processor - Intel Core i7-7700HQ
   - Memory - 8GB DDR4 2400MHz

Different dimensions of matrix (n) will be tested, as well as different numbers of slaves (t). Additionally, processes are, if not terminated, minimized. This includes browsers, CPU monitors, text editors, and other background services. Only the terminal is left open. In the case of number two, only two terminals are open. One for the one master, and one with *t-1* tabs acting for the slaves.

1.1 Slave model

The code shown below is the struct model for slave. The *slaveNo* determines the slave identification number, which has a range of [*0*, *t-1*]. Next, *rows* and *columns* determine the dimensions of the matrix that will be sent, which is then stored in ***ownMatrix*. Finally, **new_sock* is the identifier given to the slave, in which the slave can send and receive data or messages to or from, respectively. Do note that the master or the one that will send the matrix is a slave itself too.

```
typedef struct node {
  int **ownMatrix, *new_sock;
  int slaveNo, rows, columns;
} slaves;
```

The reason a *struct* is used is because the POSIX Thread Library function *pthread_create()* - which is used to execute a function in a threaded manner, allows only one argument for the thread function. The solution for this is to store those arguments in a *struct*.

1.2 Starting the program

The program is compiled with an additional flag, *-lpthread*. The flag stands for *Linux POSIX Thread*. It imports the necessary libraries to be used in the program. Then the program will be ran with one command line argument. The argument can only either be *1* or *0*. If it is *0*, program will proceed to the function *master()*. Otherwise, the function *slave()* will be called.

1.3 Dividing the matrix

Upon calling the function *master()*, the program will initiate necessary variables. Worth mentioning is the *pthread_t arrayOfSlaves[t]*. This would serve as the array in which the threads will be stored. The first thread, *arrayOfSlaves[0]* will be the master. The master's ***ownMatrix* is instantiated with *n x n/t* size then filled with elements. This goes on for the remaining *t* slaves. On the *t*th slave, the remainder upon dividing *n/t* is added to its ***ownMatrix*. By this method, the matrix is not really divided. The *n x n/t* sub-matrices are created first, effectively making a whole *n x n* matrix when combined.

```
void master() {
 . . .
 columnSize = submatrices;    // n / t
 endIndex = submatrices;      // the initial column end is =
      submatrices.
 for (i = 0; i < t; i++) {

  // last slave gets the remainder
  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] = rand() % 10 + 1;
  }

  endIndex += submatrices; // increment to where the next would
      start
  arrayOfSlaves[i].slaveNo = i;
  arrayOfSlaves[i].rows = matrixSize;
  arrayOfSlaves[i].columns = columnSize;
 }
}
```

1.4 Creating sockets

Below is the code for opening a socket for incoming clients. Most parts of the code are retrieved from the internet, but made sure that everything is clear and understood.

```
#include<sys/socket.h>
#include<arpa/inet.h>

 struct sockaddr_in server, client;
 int socket_desc, slave_sock, sockSize;
 socket_desc = socket(AF_INET, SOCK_STREAM, 0);
 if (socket_desc == -1){
   printf("Could not create socket\n");
   return;
 }

 //Prepare the sockaddr_in structure
 server.sin_family = AF_INET;
 server.sin_addr.s_addr = INADDR_ANY;
 server.sin_port = htons(8888);

 //Bind
 if(bind(socket_desc, (struct sockaddr *)&server, sizeof(server)) <
      0) {
   perror("Bind failed");
   return;
 }
 //Listen and accept and incoming connections
 listen(socket_desc, 3);
 sockSize = sizeof(struct sockaddr_in);
 printf("Waiting for slaves...\n");
```

The program now wait for incoming connections. To be more specific, the program will be wait for program instances that will run as slaves.

```
for(i = 1; i < t; i++) {

  // Blocking function
  slave_sock = accept(socket_desc, (struct sockaddr *) &client, (
      socklen_t *) &sockSize);

  printf("New slave entered. Total slaves: %d\n", i + 1);
  arrayOfSlaves[i].new_sock = malloc(1);
  *arrayOfSlaves[i].new_sock = slave_sock;

  // Handle the new slave through this thread function
  pthreadCreate = pthread_create(&sniffer_thread[i], NULL,
      connection_handler, (void *) &arrayOfSlaves[i]);
  if(pthreadCreate < 0) {
    perror("Unable to create thread");
    return;
  }
  connectedSlaves++;
}
```

As seen in the above code snippet, there is a blocking function[1] *accept()*. An iteration will not proceed until a new slave connects. The whole loop only ends once the required number of slaves (*t*), is reached. Then, the *pthread_create()* function will be executed to handle the new slave.

1.5 Thread function

The *t* slaves are handled using *pthread_create* which takes a minimum of four(4) arguments. First is the address of the thread, second are thread attributes which when set to *NULL* becomes default, third is the thread function to be executed, and fourth is the thread function argument. The thread function argument should be enclosed in a *struct* if it is more than one, hence the creation of *struct slaves*. In this case, the function name is **connection_handler()*.

```
void *connection_handler(void *client) {
  . . .
  printf("Slave %d entered handler.\n\n", slave->slaveNo);
  //Send rows and columns to slave
  write(sock, &rows, sizeof(rows));
  write(sock, &columns, sizeof(columns));
  write(sock, &slaveNo, sizeof(slaveNo));

  // Rows and columns are now sent
  // Loop until all elements of submatrix are sent
  for (i = 0; i < rows; i++)
   for (j = 0; j < columns; j++)
     write(sock, &matrixToSlave[i][j], sizeof(matrixToSlave[i][j]))
     ;

  //Receive a message from slave
  read_size = recv(sock, slaveMessage, 256, 0);

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

It is assumed that the slave is already connected when *connection_handler()* is called. The necessary data for the slave to process are then sent via *write()*. After sending, the function now waits for the slave's response via recv(). After receiving response, the thread function will now exit.

1.6 Running as a slave

Going back, if the program is ran as a slave by setting the command argument to *1*, the function will look into a text file containing the IP address of the master to connect. This is happening while the *master()* is blocked by the function *accept()*, as stated earlier.

```
void slave() {
 ...
 fp = fopen("addresses.txt", "r");
 fscanf(fp, "%s", masterIP);
 fclose(fp);

 printf("Master IP: %s\n", masterIP);

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

 // Change IP to master's everytime you run as a slave
 server.sin_addr.s_addr = inet_addr(masterIP);
 server.sin_family = AF_INET;
 server.sin_port = htons(8888);

}
```

After going through the above, the slave will now receive the *rows*, *columns*, and every element of the sub-matrix from the *master()*. Like wise, the *master()* will proceed to the next iteration if the number of connected slaves is not reached yet. When done receiving, the slave will now send an acknowledgement message *ack*, to signal that the sub-matrix is done being sent. This signals the thread function handling the slave to finish. Then the program will now terminate.

Overall, the flow of the program is:

1. The program is ran as a *master()*.
2. The *master()* waits for a *slave()*.
3. When connected, the thread function sends the data to the *slave()*. *master()* waits for another slave and repeat *Step 2* until *t* is reached.
4. Upon receiving the data, send acknowledgement and terminate the program.
5. Receive acknowledgement then end the thread function.

Steps 3 to *5* are running concurrently.

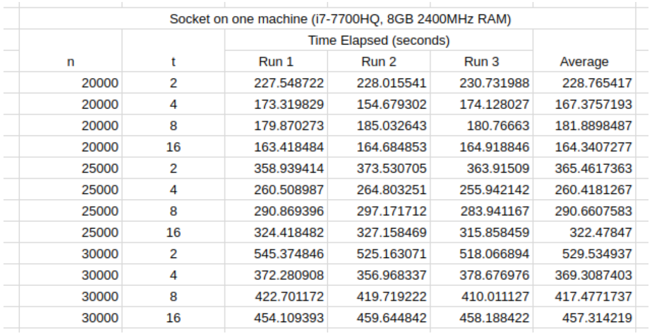
2. Results and Discussion

2.1 One machine with *t* terminals

The results of running the program on one machine is shown in *Figure 1*.

As expected, higher number of dimensions merits higher runtimes. What is unusual is that as *t* increases, the run-time increases as well for $t > 4$, when *n* is = *25000* and *30000*.

This can be explained by the number of slaves. As *t* increases, the number of *n x n/t* to be sent by the master increases too. Unlike when $t = 2$, there are only 2 slaves. The first is the master, the second is a true slave. Because of this, the master and the slave both have the halves. Basically, the less slave, the less work for master. Also, as *t* increases, it would mean more strain to the machine. Specifically, more threads to be created, memories to be allocated and threads be waited to join. Generally, more slaves, more work. Less slaves, less work for the master.



Socket on one machine (i7-7700HQ, 8GB 2400MHz RAM)					
		Time Elapsed (seconds)			
n	t	Run 1	Run 2	Run 3	Average
20000	2	227.548722	228.015541	230.731988	228.765417
20000	4	173.319829	154.679302	174.128027	167.3757193
20000	8	179.870273	185.032643	180.76663	181.8898487
20000	16	163.418484	164.684853	164.918846	164.3407277
25000	2	358.939414	373.530705	363.91509	365.4617363
25000	4	260.508987	264.803251	255.942142	260.4181267
25000	8	290.869396	297.171712	283.941167	290.6607583
25000	16	324.418482	327.158469	315.858459	322.47847
30000	2	545.374846	525.163071	518.066894	529.534937
30000	4	372.280908	356.968337	378.676976	369.3087403
30000	8	422.701172	419.719222	410.011127	417.4771737
30000	16	454.109393	459.644842	458.188422	457.314219

Figure 1. The run-time results of one machine with *t* terminals

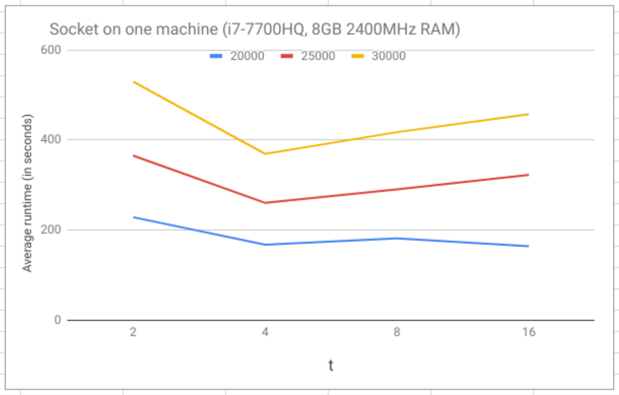
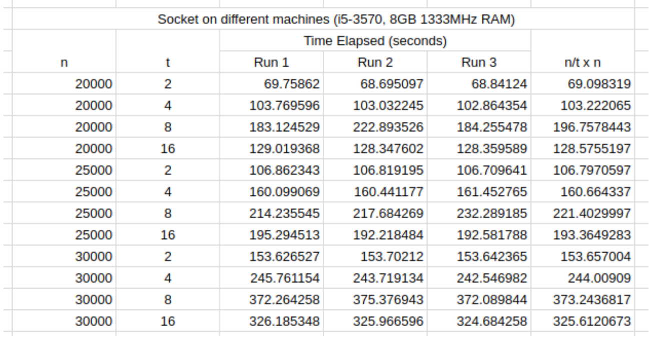


Figure 2. Line graph of *Figure 1*

2.2 On *t* machines

Shown in *Figure 3* are the results of the run-time of the program with different number *n* and *t*. The run-time is directly proportional to *t* for $t < 8$. The run-time starts to go down when $t \geq 8$, which may be explained by the number of *t*.
On the previous data, more slaves means more work. This is because all instances of the program is running on the same machine. This time, as the number of machine increases, the number of computational power increases too. Every machine has its own hardware power that is utilized by the program.



Socket on different machines (i5-3570, 8GB 1333MHz RAM)					
		Time Elapsed (seconds)			
n	t	Run 1	Run 2	Run 3	n/t x n
20000	2	69.75862	68.695097	68.84124	69.098319
20000	4	103.769596	103.032245	102.864354	103.222065
20000	8	183.124529	222.893526	184.255478	196.7578443
20000	16	129.019368	128.347602	128.359589	128.5755197
25000	2	106.862343	106.819195	106.709641	106.7970597
25000	4	160.099069	160.441177	161.452765	160.664337
25000	8	214.235545	217.684269	232.289185	221.4029997
25000	16	195.294513	192.218484	192.581788	193.3649283
30000	2	153.626527	153.70212	153.642365	153.657004
30000	4	245.761154	243.719134	242.546982	244.00909
30000	8	372.264258	375.376943	372.089844	373.2436817
30000	16	326.185348	325.966596	324.684258	325.6120673

Figure 3. The run-time results of *t* machines

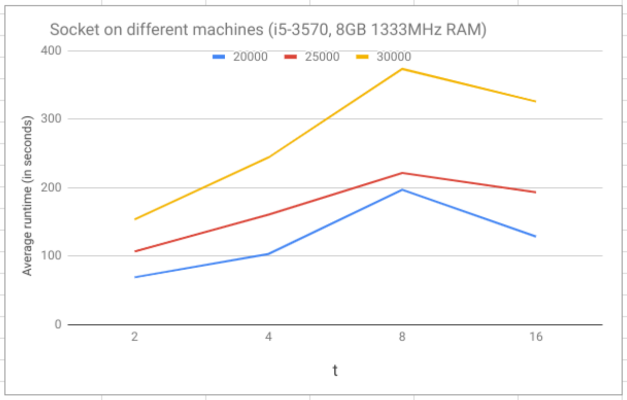


Figure 4. Line graph of *Figure 3*

3. Conclusion

Concerning the first observation, the more slaves, the longer the program will run. Judging from the results, the optimal *t* that should be used will be no larger than *4*. On the second observation, the program ran better when *t* becomes larger for $t \geq 8$. This is because each computer running the program has its own computational power. In general, the program ran on *t* machines has relatively better run-times than the program that is ran on one machine with *t* terminals. This, as stated, is because one machine's computational power *x* is reduced into *x / t* when *t* instances of the program are running on it. Moreover, an uncomplicated program bears large run-times when dealing with large data. The program only sends a piece of matrix to the other program instances, yet large run-times still are still incurred. Especially in this program where the sub-matrices are transferred element by element. Transferring matrix directly also works, but only for global, static arrays. Global because the matrix cannot be instantiated locally given the size of *n*. Global variables get resources from heap, compared to local ones which get from the stack. Static, because dynamic arrays cannot be directly sent into the socket.

4. List of Collaborators

1. Jefferson Louise Basilio
   Nothing follows

5. References

1. accept(2) - Linux man page (n.d.). Retrieved from http://man7.org/linux/man-pages/man2/accept.2.html
2. Moon, S. (2012). Server and client example with C sockets on Linux. Retrieved from https://www.binarytides.com/server-client-example-c-sockets-linux/

6. Appendices

6.1 *master()* function

```
void master() {
 ...
 for(i = 1; i < t; i++) {

  // Blocking function
  slave_sock = accept(socket_desc, (struct sockaddr *) &client, (
      socklen_t *) &sockSize);

  printf("New slave entered. Total slaves: %d\n", i + 1);
  arrayOfSlaves[i].new_sock = malloc(1);
  *arrayOfSlaves[i].new_sock = slave_sock;

  // Handle the new slave through this thread function
  pthreadCreate = pthread_create(&sniffer_thread[i], NULL,
      connection_handler, (void *) &arrayOfSlaves[i]);
  if(pthreadCreate < 0) {
   perror("Unable to create thread");
   return;
  }
  connectedSlaves++;
 }

 printf("\nAll slaves entered.\n\n");

 // Join the threads to terminate the program
 for(i = 1; i < t; i++){
  printf("Slave %d joined.\n", i);
  pthread_join( sniffer_thread[i], NULL);
 }
}
```

6.2 Thread function

```
void *connection_handler(void *client) {
  //Get the socket descriptor
  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);

  // Reset message so no trash values
  memset(slaveMessage, 0, 256);

  //Send some rows and columns to slave
  write(sock, &rows, sizeof(rows));
  write(sock, &columns, sizeof(columns));
  write(sock, &slaveNo, sizeof(slaveNo));

  // Loop until all elements are sent
  for (i = 0; i < rows; i++)
    for (j = 0; j < columns; j++)
      write(sock, &matrixToSlave[i][j], sizeof(matrixToSlave[i][j])
      );

  //Receive a message from slave
  read_size = recv(sock, slaveMessage, 256, 0);

  if(read_size < 0){
    perror("recv ack failed");
     pthread_exit(NULL);
  }

  else {
    printf("Message from Slave %d: %s\n", slave->slaveNo,
     slaveMessage);
    printf("Slave %d disconnected.\n\n", slave->slaveNo);
    fflush(stdout);
  }

  pthread_exit(NULL);
}
```

6.3 *slave()* function

```
void slave() {
  ...
  /* Create socket */
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock == -1) {
    perror("Could not create socket"); return;
  }

  server.sin_addr.s_addr = inet_addr(masterIP);
  server.sin_family = AF_INET;
  server.sin_port = htons(8888);

  /* Connect to master */
  if (connect(sock, (struct sockaddr *)&server, sizeof(server)) <
      0) {
    perror("connect failed"); return;
  }

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

  /* Receive elements one by one */
  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(sock, &receivedMatrix[i][j], sizeof(receivedMatrix[i
      ][j]), 0) < 0) {
        perror("recv Matrix failed");
        printf("Error @ [%d][%d]\n", i, j);
        return;
      }
  }

  /* Send the acknowledgement message */
  if(send(sock, ackMessage, strlen(ackMessage), 0) < 0) {
    puts("Send ack failed"); return;
  }
  close(sock);
  return;
}
```