CERTIFICATION OF APPROVAL

Parallel Processing of RSA Algorithm Using MPI Library

By

Wan Rahaya Bt Wan Dagang

A project dissertation submitted to the
Information Technology Program
Universiti Teknologi PETRONAS
in partial of the requirement for the
BACHELOR OF TECHNOLGY (HONS)
(INFORMATION AND COMMUNICATION TECHNOLOGY)

Approved by,

[illegible]

(Mr. Izzatdin Bin Abdul Aziz)

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK
JUNE 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

[illegible]

WAN RAHAYA BT WAN DAGANG

ACKNOLEDGEMENT

First and foremost, I would like to express my gratefulness to Allah S.W.T for His Bless and Guidance that strengthens me in facing every challenge in completing this project.

I would like to thanks my supervisor, Mr, Izzatdin Bin Abdul Aziz for his understanding, morale support, and guidance that has really motivated me to accomplish this project.

I would also like to thank all my families, and friends for their advices and morale supports in this project.

I would also like to say "thank you" for everybody else that has contributed to this project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS AND NOMENCLATURES

CPU	: Central Processing Unit
MPI	: Message Passing Interface
PVM	: Parallel Virtual Machine
IPC	: Inter Process Communication
SISD	: Single Instruction Single Data
SIMD	: Single Instruction Multiple Data
MISD	: Multiple Instruction Single Data
MIMD	: Multiple Instruction Multiple Data
FLOPS	: Floating Point Operation per Second
PC	: Personal Computer

ABSTRACT

This report explains the project of developing Parallel Processing of RSA Algorithm Using MPI Library. RSA Algorithm is a public-key cryptosystem that offers encryption technique which security is based on the difficulty of factoring large prime integers. The computation of RSA is performed by a series of intensive computational of modular multiplications. The scope of this project is developing a parallel system to generate public and private key, and to encrypt and decrypt files using the algorithm of RSA. The system is needed to be parallel as to overcome the problem of intensive computational by the RSA algorithm. This parallel system is going to be embedded on grid or cluster computing environment. The language and library that are going to be used for the system is C++ and Message Passing Interface (MPI). This project is completed phase by phase and for the system development, the method used is evolutionary development approach. The end result of this project is a parallel algorithm of RSA cryptosystem.

CHAPTER 1

INTRODUCTION

1.1 Background of Study

1.1.1 RSA Cryptosystem

Cryptosystem is used as a tool to protect important secrets and strategies. RSA public-key cryptosystem which is the first usable public-key cryptography developed by Ronald Rivest, Adi Shamir, and Leonard Adleman. It is used for both data encryption and authentication, based on the one-way function of integer factorization, where it is easy to construct a large number which is the product of prime powers, but hard to factorize the resultant number into its constituents.

1.1.2 Parallel Processing

Traditionally, software has been written for sequential computation, to be executed by a single computer having a single Central Processing Unit (CPU); problems are solved by a series of instructions, executed one after another by the CPU. Nowadays we already overcome this problem by having parallel processing and parallel programming for faster computational power.

The massive computational process of RSA Algorithm can be executed more efficiently using parallel processing that is implemented on cluster computing. Parallel Processing refers to the concept of speeding up the execution by dividing the program into multiple tasks that can be executed simultaneously.

The principle of parallel processing involves decomposition as the mechanism of partitioning the work load. As the author's project is on parallelizing the sequential RSA algorithm, the author then needs to discover which part of the sequential algorithm that can be parallelized; which is going to be discussed into more detail on the methodology section.

1.1.3 Message Passing Interface

Message Passing Interface (MPI) is a library specification for message-passing, proposed as standard by a broadly based committee of vendors, developers, and users. Message passing is a programming paradigm where we can directly control the flow operations and data within our own parallel programs. A message-passing library let us explicitly tell each processor what to do and provides a mechanism for us to transfer data between processes.

While debate continues as to what is the best way to pass messages back and forth between nodes of a cluster or grid, with Parallel Virtual Machine (PVM) still having supporters, chances are if we look underneath the hood of a large cluster, supercomputer or grid, we will find MPI. With MPI in place, we can embed our program from one platform to another without undue worry about inter process communication (IPC).

1.1.4 Grid or Cluster Computing

Grid or cluster computing is gaining a lot of attention within the IT industry. Grid computing is an emerging technology that enables large scale resource sharing and coordinated problem solving within distributed, coordinated group that sometimes termed as *virtual organizations*. Grid computing provides scalable high-performance mechanisms for discovering and sharing geographically remote resources.

1.2 Problem Statements

1.2.1 Problem Identification

Even though it has been proven that RSA cryptography is part of many official standards worldwide, but there is still area in the algorithm that can be improved. The security of RSA depends on the difficulty of factoring large generated key. The task of recovering the private key is equivalent to the task of factoring the modulus *n*. That is why it is always suggested to choose 'strong' key primes to generate the modulus *n*.

'Strong' key primes mean larger key primes. The products of key primes create the modulus *n*. But the larger the modulus, the greater the security, but this will lead to slower RSA algorithm operations due to massive modular computation; it takes more time to complete the calculation parts and RSA algorithm is all about mathematical calculation. As generating strong, large key prime is really time consuming (even for a 32bit size of key) therefore it shall be optimum to be processed in a parallel manner as to increase the efficiency of the algorithm operation.

This problem could be overcome by employing super computers to do all the tasks involved in the RSA computation. But then again this is not the best solution as this will lead to high cost of machine requisition. This is where parallel processing and grid computing can be applied to replace the super computers.

A collection of underutilized desktops or processors, which is near to obsolete in terms of processor technology linked by a grid middleware would be comparable to a current high end processor. Cluster can be easily built using open source middleware such as OpenMosix.

1.2.2 Significant of the Project

The expected end product of this project is parallelized RSA algorithm. The idea of this prototype is to speed up the process of RSA computation and ensure the security of the encryption and decryption by generating large, reasonable prime keys. This can be achieved by utilizing the available computer resources in Universiti Teknologi Petronas; the underutilized computing powers in the lab around campus. Idle workstations are put together to form a single cluster running MPI programs.

1.3 Objective of Studies

1.3.1 To conduct a study on RSA algorithm

In order to understand the public-key cryptosystem of RSA, the author needs to be well-versed of all mathematical algorithm used in the cryptosystem. This is an essential part as next the author needs to convert all the formulas involved into C++ language.

Up to this extend, the author has understand all the mathematical functions in the RSA algorithm; the Extended Euclidean Function, Euler Totient Function, appropriate method to generate large prime numbers, and appropriate method to determine a number is a prime.

1.3.2 To conduct a study on Parallel Programming

As the project is all about parallelizing the RSA algorithm, the author needs to know and develop the skill of parallel programming in order to create an appropriate and efficient algorithm to divide the entire subtask for the load balancing of parallel processing to take part.

Flynn's Taxonomy [1] classifies different types of parallel computing models which are Single Instructions Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). And my project makes use of the model of SIMD. And to use that model I have to be well-versed of parallel programming.

1.3.3 To conduct a study on Message Passing Interface (MPI)

MPI is the heart of this project. As in order to make the sequential algorithm turned into parallel, the author need to use and manipulate all the library of subroutines in MPI to pass messages between nodes to execute the algorithm. To achieve this objective, the author first needs to master the art of programming in MPI. As MPI is a library of subroutine specifications that can be called from C and C++ (for MPI 2), thus the author needs to be well-versed with the library functions provided by MPI.

1.3.4 To convert sequential RSA algorithm into parallel RSA algorithm

The study performed in this project is to apply the idea of converting the sequential RSA algorithm a parallel one. The algorithm is in C++ language to be integrated with MPI.

1.3.5 To implement parallel RSA into grid computing environment

The parallel system is going to be implemented in grid computing environment. The objective is to speed up the process as grid computing offers distributed system consists of a set of computers that work together to implement jobs and in this case is to implement the parallel RSA computation process.

1.4 Scope of Studies

The parallel program of RSA algorithm is implemented on LINUX. The part of the sequential program of RSA algorithm that is going to be parallelized is on the prime number generation only. This is because for RSA encryption and decryption to take part, three distinct primes are needed for the value of p, q, and e.

CHAPTER 2

LITERATURE REVIEW

Introduction

The significance of this project is to develop a parallel RSA algorithm and put it in a parallel system using the MPI libraries and implement the system in the grid computing environment. The idea of parallelizing the sequential RSA algorithm is to speed up the process of massive computation involves. The first challenge that the author has to cope up is to understand the underlying mathematic algorithm in the RSA cryptography in order to program it in C++. Studies have to be made on all the algorithms, functions and methods used in the RSA cryptography. The next thing to be considered is on how to parallel the sequential program using the programming model of message passing. In order for the author to implement the parallel program into the grid computing environment, sufficient related work pertaining this area needs to be reviewed. Important points of related works are discussed in this section.

2.1 RSA Cryptosystem

RSA cryptosystem is a public-key cryptosystem that offers both encryption and digital signatures developed by Ronald Rivest, Adi Shamir, and Leonard Adleman.

2.1.1 RSA Algorithm:

1. Generate two large random primes, p and q, of approximately equal size such that their product n = pq is of the required bit length, e.g. 1024 bits.
2. Compute n = pq and (φ) phi = (p-1)(q-1).

3. Choose an integer e, $1 < e < phi$, such that gcd(e, phi) = 1.
4. Compute the secret exponent d, $1 < d < phi$, such that $ed \equiv 1 \pmod{phi}$.
5. The public key is (n, e) and the private key is (n, d). The values of p, q, and phi should also be kept secret.
6. Plaintext is M
7. Cipher text; $C = M^e \bmod n$
8. Plaintext; $M = C^d \bmod n$

- n is known as the *modulus.*
- e is known as the *public exponent* or *encryption exponent.*
- d is known as the *secret exponent* or *decryption exponent.*

"*The RSA system is the most widely used public-key cryptosystem today and has often been called a de facto standard.*" [2]

It has been proven that RSA Cryptosystem is part of many official standards worldwide. The RSA system is currently used in a wide variety of products, platforms, and industries around the world. RSA is also built into current operating systems by Microsoft, Apple, Sun and Novell. It is also used internally in many institutions, including branches of the U.S government, major corporations, national laboratories and universities.

"*In the literature pertaining to the RSA algorithm, it has often been suggested that in choosing a key pair, one should use so-called "strong" primes p and q to generate the modulus n.*" [3]

Strong primes have certain properties that make the product *n* hard to factor by specific factoring methods; this is why choosing strong primes helps to increase security. However, large primes would take more time to be generated compare to an arbitrary prime and this is what the project is mainly about. Segments of sequential code in RSA are done in parallel for example, massive iterations and decisions.

"*RSA currently recommends key sizes of 1024 bits for corporate use and 2048 bits for extremely valuable keys like the root key pair used by a certifying authority.*" [2]

The best size for a modulus depends on one's security needs. The larger the modulus, the greater the security of the system. A modulus *n* is the product of two primes; *p* and *q*. Let say if one chooses to use a 1024-bit modulus, the primes should each have length approximately 512 bits. The primes should be roughly equal length as the modulus is harder to factor than if one prime is much smaller than the other.

One should choose a modulus length upon consideration, firstly, of the value of the protected data and how long it needs to be protected, and secondly, how dreadful potential threats might be.

2.2 Parallel Computing

"*A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem.*"[3]

The main interest of parallel computing is that it offers the potential to concentrate on computational process; whether processors, memory, or I/O bandwidth on important computational process. It is important to note that the performance of a computer depends directly on the time required to perform a basic operation and the number of these basic operations that can be performed concurrently.

The time to perform a basic operation is limited by the *clock cycle* of the processor that is the time required to perform the most primitive operation. One way to overcome the decreasing clock cycle times is by incorporating multiple computers, each with its own processor, memory and associated interconnection mechanism, which is going to be implemented in this project, therefore results in better floating point operation per second (FLOPS).

"*Parallel computing is a divide-and-conquer strategy*". [3]

The idea of partitioning the work in parallel computing is for all processors to keep busy and none remain idle. Parallel computing is a natural extension of the concept of divide and conquer; we first begin with a problem that is need to be solved, then access the available resources that can be used to solve the problem; which is the number of processors that can be used, and attempt to partition the problem into manageable pieces that can be executed concurrently by each processor.

"*A popular taxonomy for parallel computers is the description introduced by Michael Flynn in the mid-1960s of the programming model as single instruction – multiple data stream (SIMD) or multiple instruction-multiple data stream (MIMD).*"[3]

On a SIMD computer, each processor performs the same arithmetic operation or stays idle during each computer clock, as controlled by a central control unit. SIMD applies the concept of master node synchronizes and coordinates the whole process. On the other hand, on a MIMD computer, each processor can execute a separate stream of instructions on its own local data.

2.3 Message Passing Interface

"*Almost everything in MPI can be summed up in the single idea of message sent – message received.*" [3]

The basic principle of MPI is that a multiple parallel processes work concurrently toward a common goal using messages as their means of communicating among each other. This is the mechanism of message-passing programming model. Message-passing is probably the most widely used parallel programming model today.

"*Message-passing model does not preclude the dynamic creation of tasks, the execution of multiple tasks per processor, or the execution of different programs by different tasks. However, in practice, most message-passing systems create a fixed number of identical tasks at program startup and do not allow tasks to be created or destroyed during program execution*". [4]

These kinds of systems are said to implement SIMD programming model that is mentioned earlier. This is because each task executes the same program but operates on different data. Based on the reviewed that has be done on journals and book on MPI, it shows that in most MPI implementations, a fixed set of processes is created at program initialization and one process is created per processor.

However, as it is said that MPI does not preclude dynamic creation of tasks; MPI processes may also execute different programs. Thus the MPI programming model is sometimes referred to as MIMD to distinguish it from the SIMD model in which every processor executes the same program.

"*On the other hand, recent development in parallel programming has placed emphasis on more traditional message-passing models, such as PVM, NX, and MPI. MPI especially has been adopted as the de facto standard with support from all vendors*". [5]

The distributed memory paradigm for parallel computing is widely used and standardized interfaces such as MPI have enabled portability between clusters from different manufacturers. An advantage of the MPI standard is that different platforms can have their own optimized implementations. For this project, MPI is chosen to be implemented together with RSA algorithm as there are seamless approach to parallel computing in C++ and MPI available through books and online resources via the internet. The author experiences in writing C++ program for some course project before is also the main reason for choosing the MPI.

2.4 Grid Computing

"Grid computing offers the power to address some of the world's most challenging problems; for example, struggles to prevent cancer and cure smallpox, to reliably predict earthquakes and global warming, and many others". [6]

Two key benefits of grid computing would enable these advances. First, grids tie varied systems into a mega computer, and therefore, can apply greater computational power to a task. Second, a grid virtualizes these varied resources, so that applications for the grid can be written as if for a single, local computer, vastly simplifying the development needed for such powerful applications.

"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." [7]

Like it is said before, grid computing is an emerging technology that enables large scale resource sharing and coordinated problem solving within distributed, coordinated group; this is the computational aspects of grids. Thus for the project, this fundamental element of grid computing is going to help a lot in executing the massive calculation task in a parallel manner by utilizing the available resources; the so-called sleeping PCs in the lab.

CHAPTER 3

METHODOLOGY

3.1 System Architecture – The Evolutionary Approach Development

This project would be completed phase by phase and for the system architecture for this project, *evolutionary approach development* model is chosen. This method is chosen due to its flexible allowances choice of system development methodology.

Traditional approach of system development methodology that needs to get the development model mostly correct in the early stage is impossible as this project involves more than just one area of studies such like RSA algorithm, parallel processing, MPI and grid computing. Various issues need to be considered that is unforeseen at the beginning. Thus different conditions and techniques would be evolved during project development phase from time to time.

Evolutionary development is an iterative and incremental approach for system development. The system will be delivered incrementally over time. Evolutionary development is new to many existing professional developer, and many traditional programmers as well. Figure 1 illustrates the phases involved in evolutionary development approach.

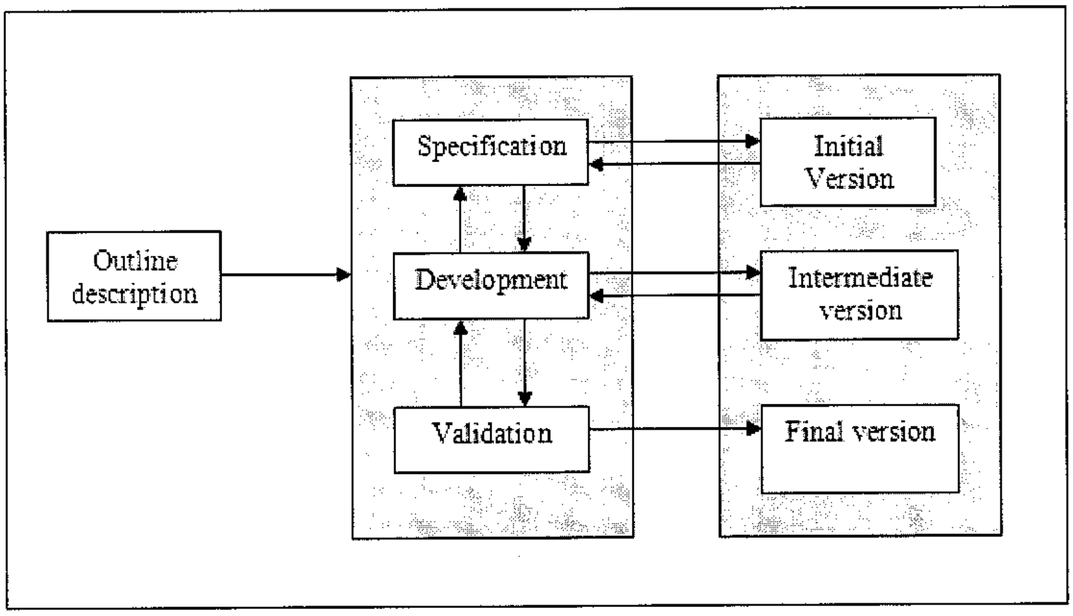


Figure 1: Phases involved in Evolutionary Development Approach

- **Specification Phase**

  The project begins by developing a sequential program of RSA algorithm in C++. Then the phase is to identify which part of the sequential program that could be parallelized. This is the beginning of the specification phase of the project development. Although the main objective is to parallelize the sequential RSA algorithm, but not all part of the program can be parallelized. This is where the partitioning stage of the programming design takes place which is intended to expose the opportunities for parallel execution.

- **Development Phase**

  This is where the execution of the project takes place based on the specification specified. As mentioned earlier, the parallelization of the algorithm is achieved by using MPI libraries. And also as this project is using iterative and incremental approach, the parallel program of RSA algorithm is written incrementally over time which means troubleshooting is done on the program from time to time to avoid error that cannot be debug later on.

- **Validation Phase**

Then the program prototype will go on the validation phase in order to ensure the project requirement is achieved. If there are still areas that need to be modified and altered, the whole phases will be repeated all over again until the final version of the program is released. In this project, most of the evaluation processes are done by the author alone, which means the author as the programmer is testing the program as an evaluator.

3.2 Tools and Devices

3.2.1 Development Tools

Programming language used to write the parallel RSA algorithm is C++. Apart from the author's experiences of writing programs using this language, the major reason of choosing C++ to write the program is because it provides an object-oriented infrastructure that accommodates mechanism of breaking down the problem into a collection of data structures and operations that is similar with the characteristic of parallel processing.

Furthermore, C++ is also compatible with the concept of partitioning, and dynamic memory allocation which are the concept that is going to be involved in the parallel RSA algorithm. As mentioned earlier, MPI is used for the parallel processing of the algorithm; a library of subroutine specifications that can be called from C and C++; this is also another reason why the parallel program is written in C++. The application that is used to edit the program in C++ is Microsoft Visual Studio C++ 6.0 and g++.

3.2.2 Libraries

MPI provides all the subroutines that are needed to break the tasks involved in the massive computational process into subtasks to be distributed to a number of available nodes and processed. The goal of the MPI is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. MPI provides an appropriate environment for general purpose message-passing programs, especially programs with regular communication patterns. Figure 2 shows the general MPI program structure:

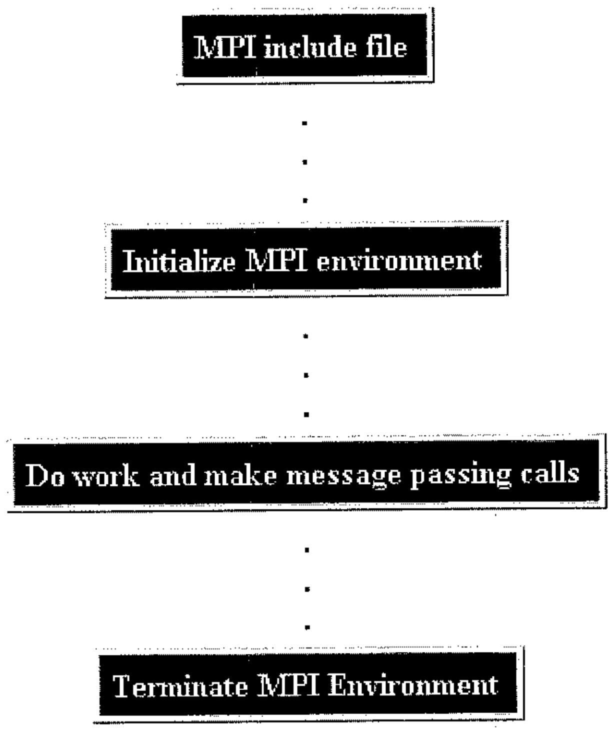

MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

Figure 2: General MPI Program Structure

MPI contains approximately 125 functions avoiding the author from any mishaps when implementing common communication structures, such as send-receive, broadcasts and reductions. However, MPI is reasonably easy to learn, as a complete message-passing program can be written with just six basic functions. Please refer to appendix to have a look at some basic MPI functions.

MPI is such a useful communications library for applications that need to be ported to many platforms. Versions of MPI exist for virtually every major platform: message-passing supercomputers, scalable shared-memory machines, symmetric multiprocessors, loosely-coupled workstation clusters, and even individual PCs. With MPI, the author can write code *once* and merely recompile it for each new platform.

3.2.3 Platform

As mentioned earlier, the parallel program of RSA algorithm will be running on grid computing platform that is developed in the lab. The idle workstations in the lab is put together to form a single cluster running MPI programs.

3.2.4 Hardware/Devices

- Workstation with minimum Pentium III processor, 256 ram memory
- Fast Ethernet Switch
- Network adapter and Unshielded Twisted Pair (UTP) cable

CHAPTER 4

RESULT AND DISCUSSION

4.1 Project Results

4.1.1 Sequential RSA Algorithm

The first development of the project was for the author to write the sequential program of RSA Algorithm. The algorithm of the sequential program is as follow:

Pseudocode

```
1.  Start
2.  Begin while loop until an invalid option is selected
3.  Prompt user to select program option
4.      Switch (method)
5.          Case 1: prompt user to enter a value greater than 10000
6.                If value > 10000, generate key primes
7.               Else
8.                   Exit program
9.          Case 2: open file and encrypt
10.         Case 3: open file and decrypt
11.         Case 4: exit program
12.     End
End
```

Figure 3 shows the flowchart of sequential program of RSA algorithm written in C++:

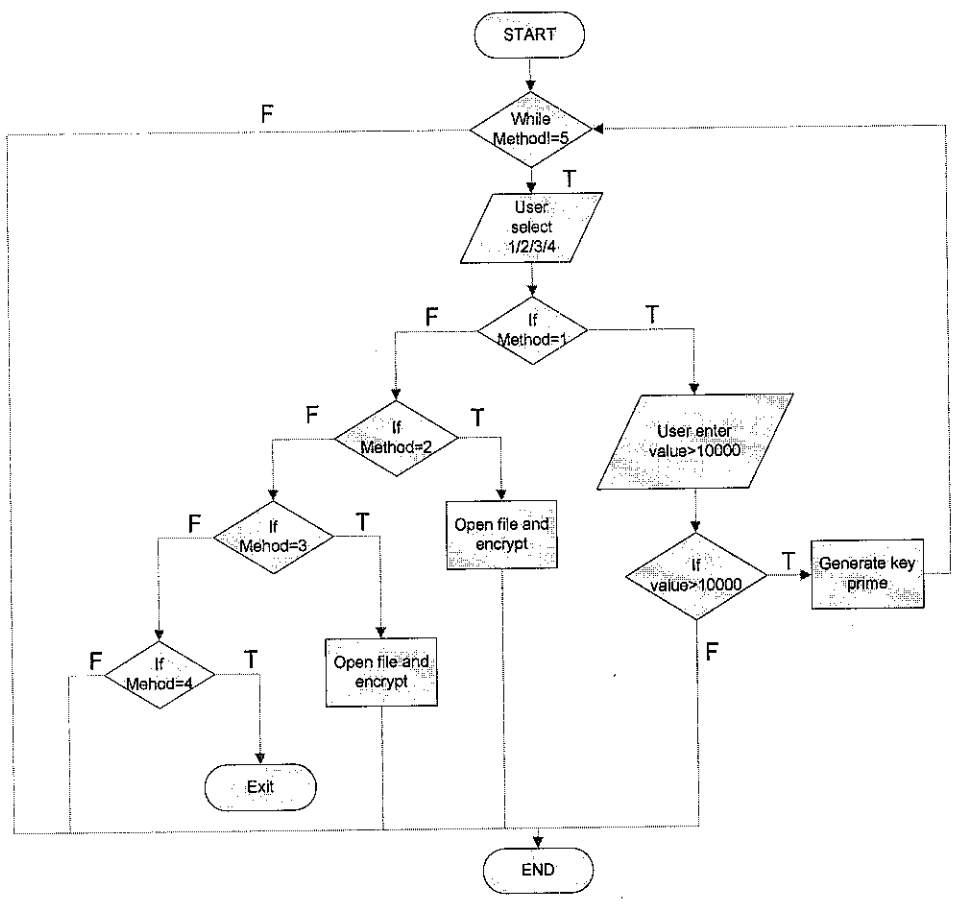


Figure 3: Flowchart of Sequential Program of RSA Algorithm

The sequential program flowchart shown in figure 3, begins by prompting user to select an option; whether to create the key primes (if they are not yet created), or to do the encryption on a particular file (using the generated key primes), or to do decryption on a decrypted file (using the generated key primes) or to exit from the program.

4.1.1.1 Trial Division

This program is written in C++. This program used *dynamic 2D array*; using *calloc* function to create a table that contains a population of odd numbers and then each of them will be determined whether they are a prime number or not. As the odd numbers populated are only consist of not more than 5 digits, the primality test used to determine whether they are prime or not is by using *trial division* (if the number are consist of more than 200 digits, trial division is impossible). Trial division is one if the simplest method to test a primality of a number. Trial division consists of trial-dividing n (integer to be factored) by every prime number less than or equal to square root of n; since all other combinations of factors would include one number larger than the square root and one smaller.

4.1.1.2 Binary Method

Encryption and decryption part of RSA algorithm involve a massive modular multiplication. In this sequential program, the author has used the Binary method for both parts. The binary method scans the bits of exponentiation either from left to right or from right to left, a squaring is performed at each step, and depending on the scanned bit value, a subsequent multiplication is performed. In this program, the binary method scans the bits of exponentiation from left to right.

$$C: = M^e \text{ (mod n)},$$

To execute the equation above, it cannot be computed by first exponentiating M^e and then performing a division to obtain the remainder of (M^e) % n. This is because the storage required to store the temporary result of M^e is enormous. Thus we have no way of storing it. Then we should know in advance how many modular multiplications that are needed to compute M^e *(mod n)* before we actually execute the multiplication to avoid memory wastage.

For example, to compute M^{15} *(mod n)*, if we compute it in a naive way, we will be computing all powers of M until 15;

$$M \rightarrow M^2 \rightarrow M^3 \rightarrow M^4 \rightarrow M^5 \rightarrow M^6 \rightarrow M^7 \rightarrow M^8 \rightarrow M^9 \rightarrow M^{10} \rightarrow \ldots\ldots\ldots M^{15}$$

which requires 14 multiplications. However, not all powers of M need to be computed in order to obtain M^{15}. Using the Binary method, we require only 6 multiplications;

$$M \rightarrow M^2 \rightarrow M^3 \rightarrow M^6 \rightarrow M^7 \rightarrow M^{14} \rightarrow M^{15}$$

The Binary method is also called as *exponentiation by repeated squaring and multiplication*. The algorithm of binary method is shown below:

```
Input: $M, e, N$.
Output: $C = M^e \bmod n$.
1. **if** $e_{k-1}$ **then** $C := M$ **else** $C := 1$
2. **for** $i = k-2$ **downto** 0
      a. $C := C.C \pmod n$
      b. **if** $e_i = 1$ **then** $C := C.M \pmod n$
3. **return** $C$
```

As an example, let e = 250, then first it is converted into binary values = 11111010, which implies $k = 8$. Thus we take $C := M$ as $e_{k-1} = e_7 = 1$. Then the binary method would take place as follows:

Table 1: Modular Multiplication of M^{15} using Binary Method

i	e_i	Step 2a	Step 2b
6	1	$(M)^2 = M^2$	$M^2 . M = M^3$
5	1	$(M^3)^2 = M^6$	$M^6 . M = M^7$
4	1	$(M^7)^2 = M^{14}$	$M^{14} . M = M^{15}$
3	1	$(M^{15})^2 = M^{30}$	$M^{30} . M = M^{31}$
2	0	$(M^{31})^2 = M^{62}$	M^{62}
1	1	$(M^{62})^2 = M^{124}$	$M^{124} . M = M^{125}$
0	0	$(M^{125})^2 = M^{250}$	M^{250}

As we can see, the number of multiplications required by the binary method for computing M^{250} is only 12 (instead of 250 multiplications).

$M \text{->} M^2 \text{->} M^3 \text{->} M^6 \text{->} M^7 \text{->} M^{14} \text{->} M^{15} \text{->} M^{30} \text{->} M^{31} \text{->} M^{62} \text{->} M^{124} \text{->} M^{250}$

For an arbitrary k-bit number e with $e_{k-1} = 1$, the binary method requires:

- Squarings: $k - 1$ where k is the number of bits in the binary expansion of e.
- Multiplications: $H(e) - 1$ where $H(e)$ is the Hamming weight (the number of 1s in the binary expansion) of e.

This binary method has been successfully implemented in the author's sequential and parallel algorithm of RSA.

4.1.2 Parallel RSA Algorithm

The portion of the sequential RSA algorithm that is going to be parallelized is on the generation of prime numbers that is used for *public keys (n,e)* and the *private key (n,d)*. Which means there are three distinct primes needed for encryption and decryption of RSA to take part; the value of p, q, and e. The algorithm of the parallel program is as follows:

Pseudocode

```
1. Start
2. Master creates a table of odd numbers and initialized row[0] only
3. Master broadcasts row[0] to all slaves
4. Master sends a number of rows to each slaves
       Each slave will receive an initialized row from master
       Each slave will populate row prime numbers
       Each slave will return populated row to Master
```

```
5. Master waits for results from slaves
6. Master receives populated rows from each slave
7. Master checks unpopulated rows
       If maxRow > 0
              Master will send another unpopulated row to slave
8. Master picks prime numbers randomly
9. Prompt user to select program option
10.    Switch (method)
11.        Case 1: prompt user to enter a value greater than 10000
12.              If value > 10000, generate key primes
13.             Else, Exit program
14.        Case 2: open file and encrypt
15.        Case 3: open file and decrypt
16.        Case 4: exit program
17.    End
End
```

Figure 4 shows the flowchart of parallel program of RSA algorithm:

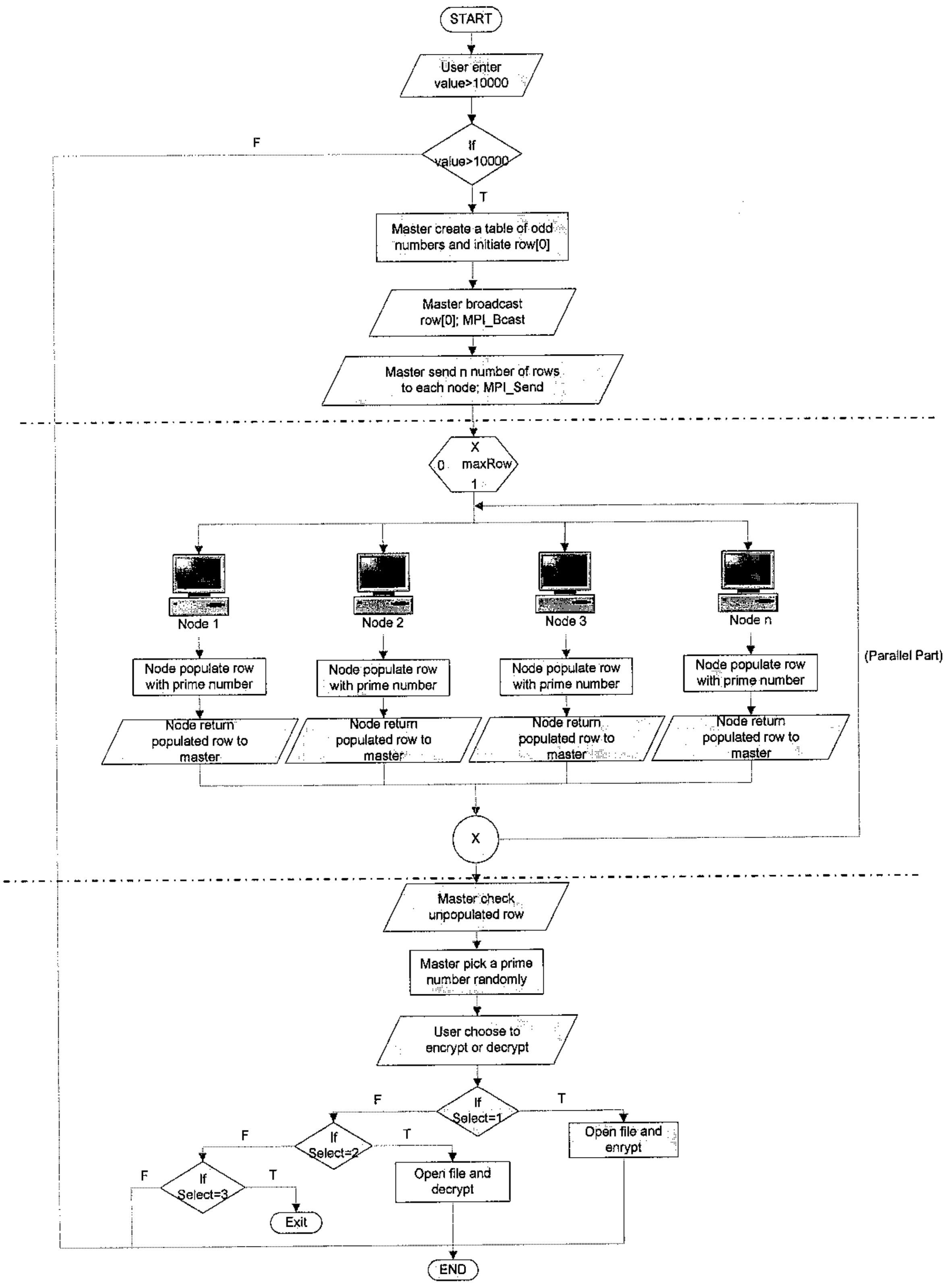

START
User enter value>10000
If value>10000
F
T
Master create a table of odd numbers and initiate row[0]
Master broadcast row[0]; MPI_Bcast
Master send n number of rows to each node; MPI_Send
X
0
maxRow
1
Node 1
Node 2
Node 3
Node n
Node populate row with prime number
Node return populated row to master
(Parallel Part)
X
Master check unpopulated row
Master pick a prime number randomly
User choose to encrypt or decrypt
If Select=1
Open file and enrypt
If Select=2
Open file and decrypt
If Select=3
Exit
END

Figure 4: Flowchart of Parallel Program of RSA Algorithm

When the code runs on the grid cluster, master will create a table of dynamic 2D array that later populated with odd number by slaves. If the degree of the security needed by the user is really high, then they should enter a large number; perhaps 98000, so that large prime numbers can be generated and vice versa. But then again as mentioned earlier, it takes quite a time to generate large prime numbers.

A pointer to pointer variable ***table* in master will point to an array of pointers that subsequently point to a number of rows; this makes up a table of dynamic 2D array. After the table of dynamic 2D array is created, master will then initialize the first row of the table only. This idea is illustrated in figure 5.

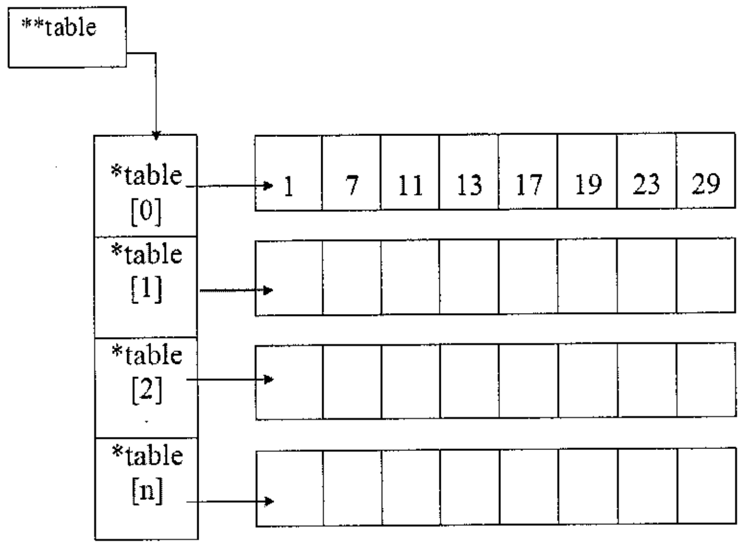


Figure 5: Master creates a dynamic 2D array to populate odd numbers

The parallel segment begins when master broadcast the row[0] to all nodes by using MPI_Bcast. This row[0] will be used by each node to continue populating the rest of the rows of the table with odd numbers. After then master will equally divide *n-1* no of rows left that is yet to be populated by number of nodes available in the grid cluster. Then each node will be given an equal no of rows to be populated with odd numbers. This achieved by using MPI_Send. A visual representation of this idea is depicted in figure 6.

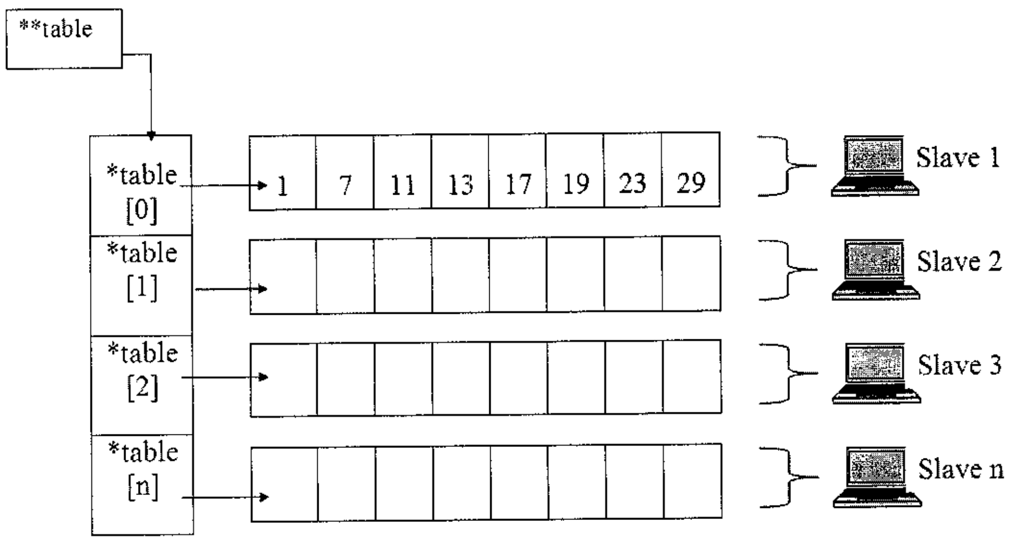


Figure 6: Master sends an equal-sized of row of 2D array to each slave

Then each node will be receiving *n* numbers of rows to be populated with odd numbers. This is where the parallel process will take place. Each node will process each row given concurrently. Each node will first populate the rows with odd numbers then filter out for prime numbers using the primality test chosen; *trial division*. Odd numbers that are prime will remain in the rows but those that are not will be assigned to NULL. Then each populated row is return to master and master will then randomly pick for three distinct primes for the value of $p, q,$ and e.

For an example, if there are 4 processors available to execute above tasks, and there are 1200 rows to needs to be populated with prime numbers, each row will be given 300 rows each to be processed. The overall process is depicted in figure 7.

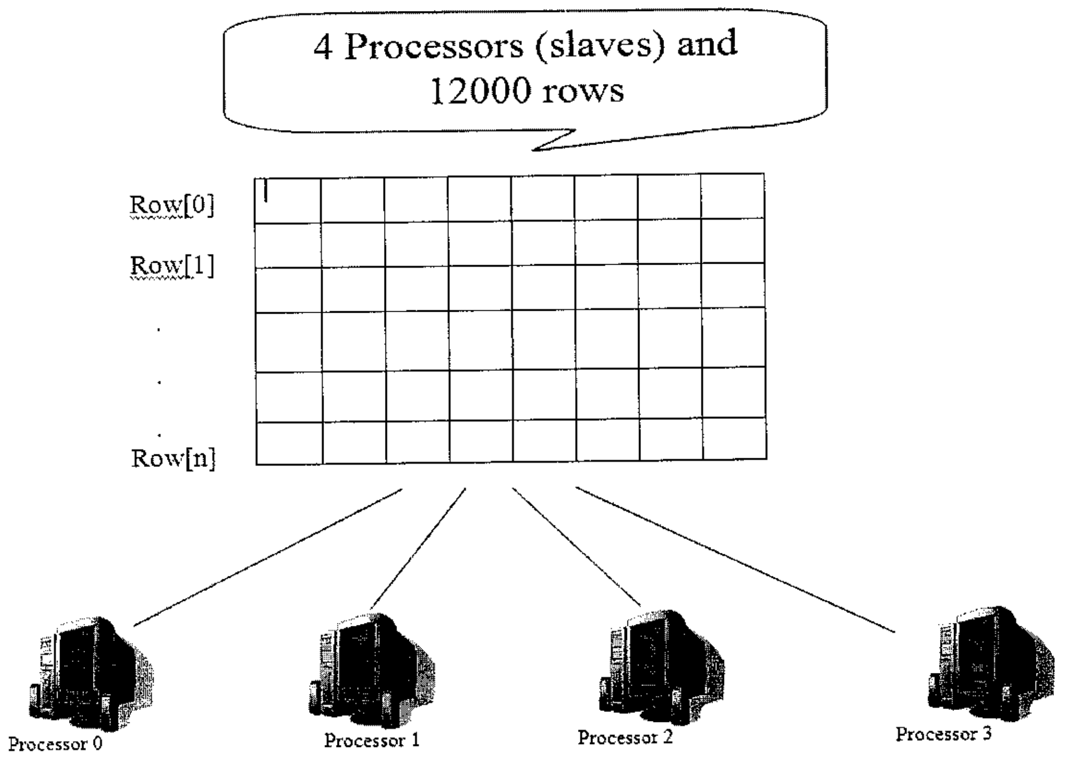


Figure 7: Example of assigning 1200 rows to 4 processors (slaves)

Processor 0 will be processing row[1] up to row[299], processor 1 will be processing row[300] up to row[599], processor 2 will be processing row[600] up to row[899] and lastly processor 3 will be processing row[900] up to the last row, row[1199].

After each node return the populated rows to master, master will then pick randomly prime numbers to be assigned as the value of *p*, *q*, and *e*. the program will then continue with encryption and decryption part of the algorithm. It is clear here that the parallel process that takes place in the whole program is only on the prime number generation.

Below is the algorithm of the parallel part of the whole program:

```
Begin algorithm

    Master part
    1. Generates 2D table, all table elements are assigned to NULL except
       for row[0]
    2. Broadcasts row[0] to all slaves
    3. Sends a number of rows to each slaves
    4. Waits for results from slaves
    5. Repeat from (3) until no more rows to be sent
    6. Proceeds with sequential part

    Slaves part
    7. Receive row[0] from Master
    8. Receive an uninitialized row from Master
    9. Generate odd numbers, fill up all the rows
    10. Filter out for prime numbers, non-prime assigned to NULL
    11. Send populated rows to Master
    12. Repeat from (8) from Master until no more rows obtained from Master

End algorithm
```

4.1.3 Performance Measurement

Number of nodes	Execution Time (ms)
1	7.850
3	0.039
5	0.043
10	0.053
30	0.093

Table 2: Comparison of Execution Time for Different Number of Nodes

4.2 Discussion

The grid computers are running on grid middleware, OpenMosix. Each node is having two processors, which makes up:

20 nodes * 2 processors * 733MHz * 0.28 FLOPS/cycle = 8.206 GFLOPS

Based on the bar chart above, we could see a distinction comparison between running the program only on a single node and more. However, it seems like there is a decrement in performance when the program is running more than 3 nodes. That is caused by the network latency during the distribution of the task that leads to increment in the time taken for the execution to complete.

Thus it is approved that algorithm of parallel RSA Cryptosystem using MPI Library has successfully reduced time taken in generating RSA key primes, hence resulting in reducing time taken for the whole process that takes place in execution of RSA Cryptosystem program.

CHAPTER 5

CONCLUSION AND RECOMMENDATION

There are five objectives of this project. All of them have been achieved successfully. The programming paradigm that is involved in this parallel algorithm of RSA cryptosystem is it used the *master-slave paradigm* in finding the solution. The master process coordinates the work of the slave processes. However, in this solution, it performs self-scheduling where a slave process will be given another portion of the uninitialized table to be calculated when it has finished the current processing. This is suitable in an environment where the job loads in the slave processes are different from each other.

There is some recommendation that should be highlighted here for the furtherance of the project development of this project. The scope of parallel part of the program should be enlarged into the area of calculation of encryption and decryption part in order to achieve an optimum parallelized RSA algorithm. For further refinement, the tedious programming work on parallelizing encryption and decryption part can be completed efficiently.

REFERENCES

[1] Foster, Ian. *Designing and Programming Parallel Programs.* An Online Publishing Project of *Addison-Wesley Inc.*, *Argonne National Laboratory*, and the NSF *Center for Research on Parallel Computation.* <http://www-unix.mcs.anl.gov/dbpp/> Recent access: Feb 2006.

[2] R.L. Rivest, A. Shamir, and L. Adleman. (1978). A journal on *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.*

[3] A. Mollin, Richard, Chapman (2000). A book on *RSA and Public-key Cryptography. Hall/CRC.*

[4] Foster, Ian. (1995). A journal on *Parallel Computers and Computation.*

[5] Bourbonnais.S, Goate V.M, Haas M, Homan R.W, Malaika S, Narang I, Raman V. (Volume 43, Number 4, 2004). IBM Systems Journal; *Towards an Information Infrastructure.*

[7] J. Joseph, M. Ernest, and C. Fellenstein. (Volume 43, Number 4, 2004). IBM Systems Journal; *Evolution of Grid Computing Architecture and Grid Adoption Models*

[8] Mao, Wenbo. (2004). *Modern Cyrptography.* Prentice Hall, PTR.

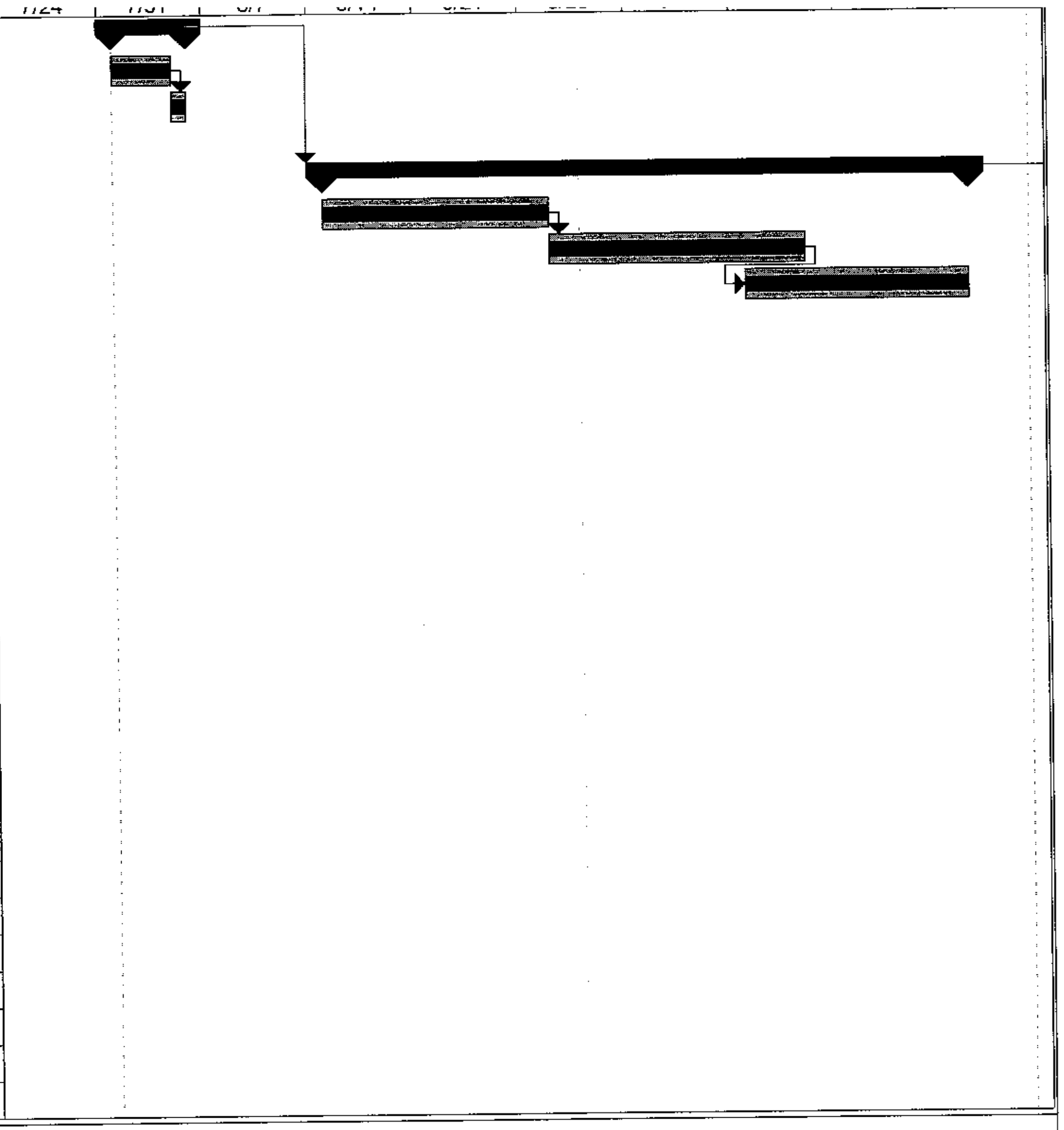
[9] Mohamed Ferdaus Abdul Wahab. (2005). *Implementation of Grid Computing for Cyrptosystem (RSA).*

[10] Stalling, William. (2004). *Cryptography and Network Security, Principle and Practive. Prentice Hall.*

APPENDIX

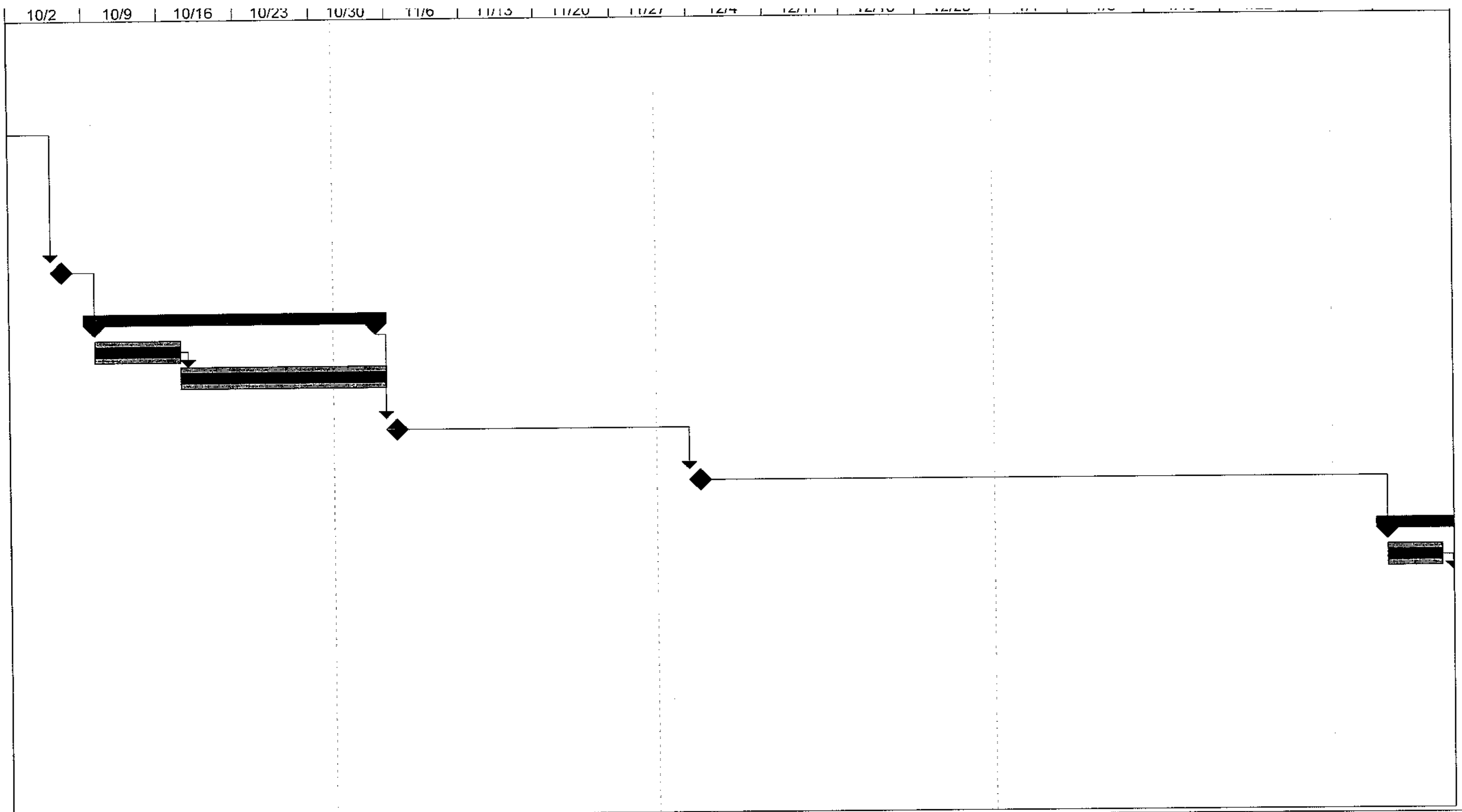
APPENDIX A: Project Gantt Chart

1	**Selection of Project Topic**	**Mon 8/1/05**	**Fri 8/5/05**
2	Topic Proposal	Mon 8/1/05	Thu 8/4/05
3	Topic Assigment	Fri 8/5/05	Fri 8/5/05
4			
5	**Preliminary Research/Design Work**	**Mon 8/15/05**	**Mon 9/26/05**
6	Determine project concept, objectives and scope	Mon 8/15/05	Mon 8/29/05
7	Literature Studies	Tue 8/30/05	Thu 9/15/05
8	Project Planning	Mon 9/12/05	Mon 9/26/05
9			
10	Submission of Preliminary Report	Fri 10/7/05	Fri 10/7/05
11			
12	**Project Work (Part I)**	**Mon 10/10/05**	**Fri 11/4/05**
13	Defining Requirement	Mon 10/10/05	Mon 10/17/05
14	Sequential Program Design	Tue 10/18/05	Sat 11/5/05
15			
16	Submission of Interim Report	Mon 11/7/05	Mon 11/7/05
17			
18	Oral Presentation with Internal Examiner	Mon 12/5/05	Mon 12/5/05
19			
20	**Project Work (Part II)**	**Mon 2/6/06**	**Mon 4/3/06**
21	Tools Setting	Mon 2/6/06	Fri 2/10/06
22	Parallel Program Design	Sun 2/12/06	Sun 2/12/06
23	Project Development	Thu 3/2/06	Wed 3/29/06
24	Project Testing	Thu 3/30/06	Mon 4/3/06
25			
26	Pre-EDX	Tue 4/4/06	Tue 4/4/06
27			
28	Submission of Dissertation Final Draft	Fri 5/19/06	Fri 5/19/06
29			
30	Oral Presentation	Fri 6/16/06	Fri 6/16/06



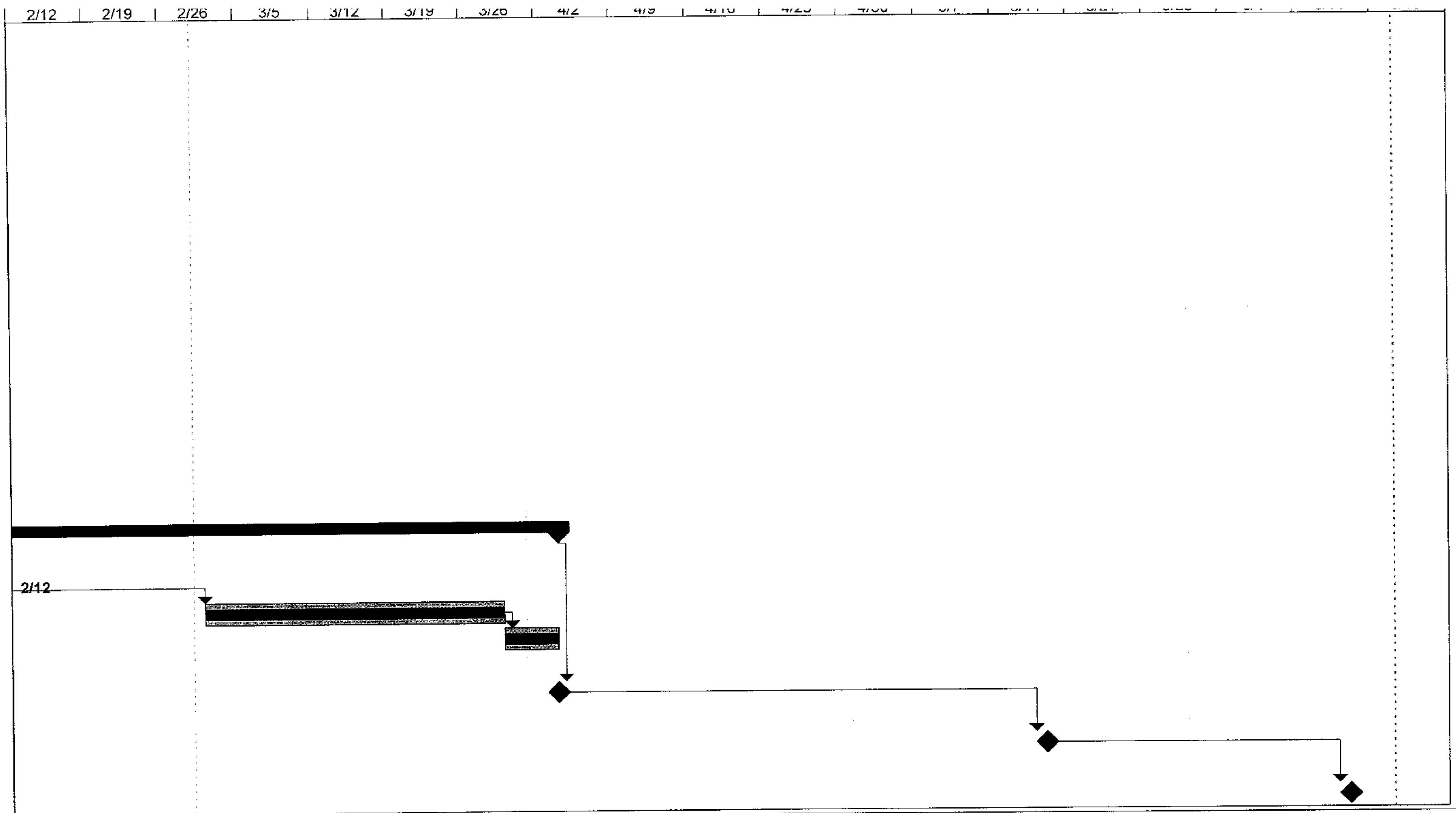
Project: Final Year Project Schedule
Date: 1st August 2005

Task	▆	Milestone	◆	External Tasks	▨
Split	⋯	Summary	▬	External Milestone	◆
Progress	▬	Project Summary	▬	Deadline	⇩



Project: Final Year Project Schedule
Date: 1st August 2005

Task	▨	Milestone	◆	External Tasks	▨
Split	' ' ' ' ' ' '	Summary	▬	External Milestone	◆
Progress	▬	Project Summary	▬	Deadline	⇩



Project: Final Year Project Schedule
Date: 1st August 2005

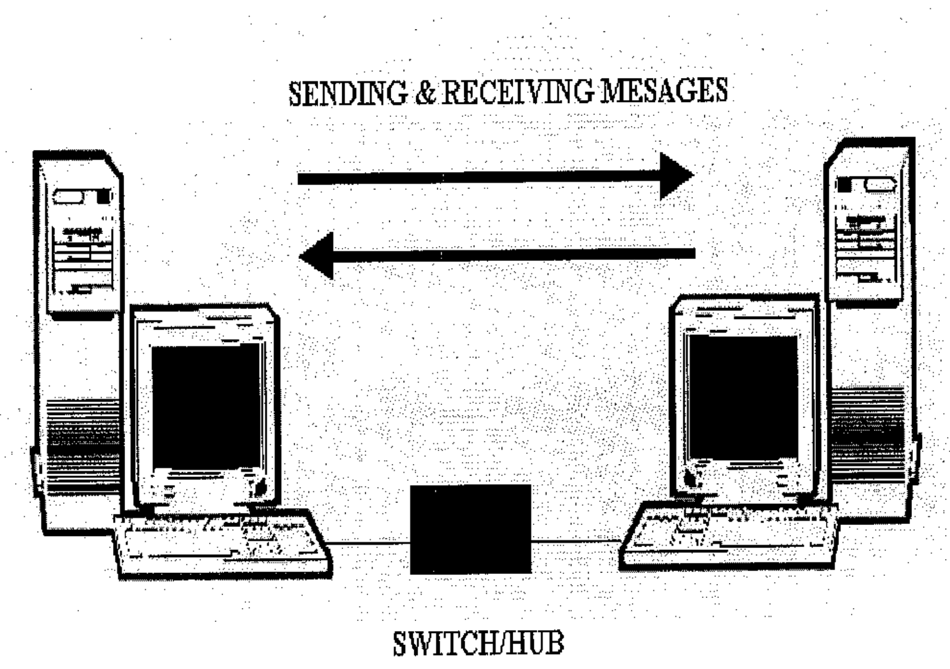
Task	▨	Milestone	◆	External Tasks	▨
Split	· · · · · · ·	Summary	▬	External Milestone	◆
Progress	▬	Project Summary	▬	Deadline	⇩

APPENDIX B: Basic Functions in MPI

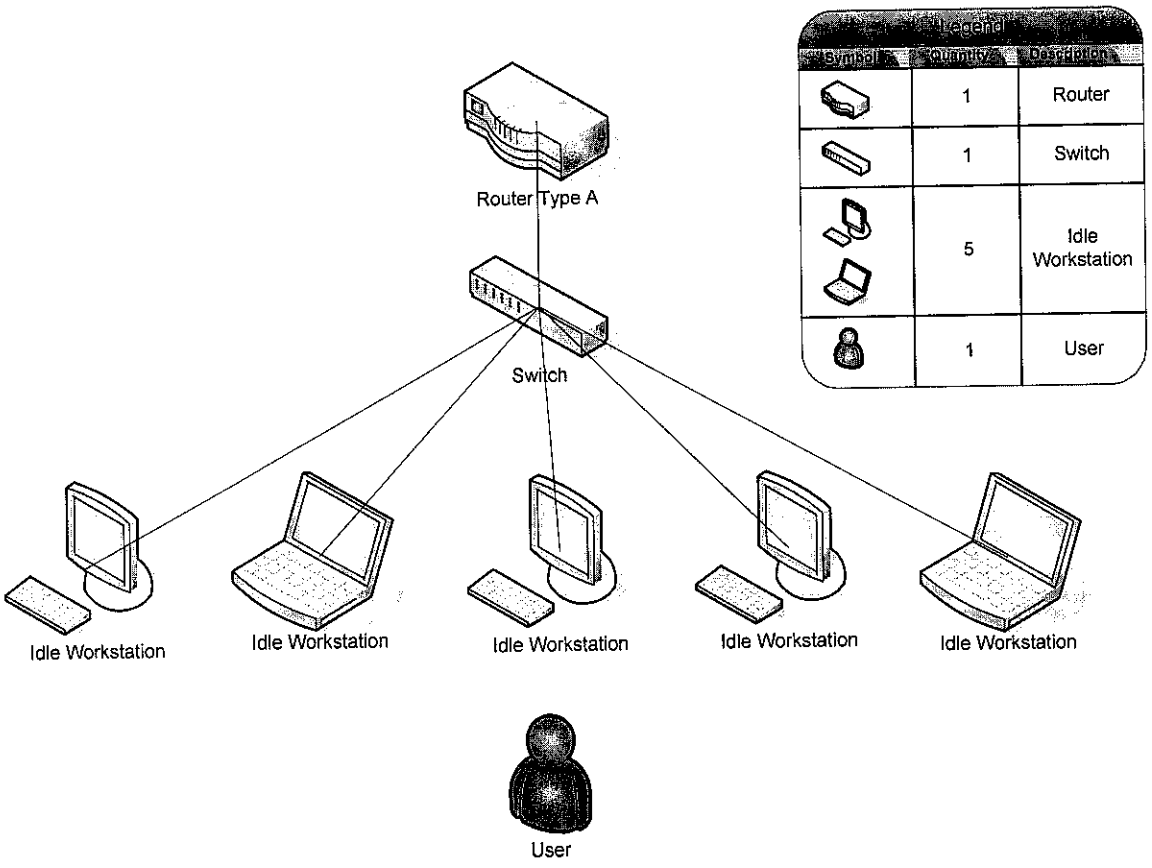
MPI Routines	Function
MPI_Send	Basic blocking send operation. Routine returns only after th application buffer in the sending task is free for reuse. The MP standard permits the use of a system buffer but does not require it. `MPI_Send(&buf,count,datatype,dest,tag,comm)`
MPI_Recv	Receive a message and block until the requested data is available i the application buffer in the receiving task. `MPI_Recv (&buf,count,datatype,source,tag,comm,&status)`
MPI_Bcast	Broadcasts (sends) a message from the process with rank "root" t all other processes in the group. `MPI_Bcast(&buffer,count,datatype,source,comm)`
MPI_Scatter	Distributes distinct messages from a single source task to each tas in the group. `MPI_Scatter(&sendbuf,sendcnt,sendtype,&recvbuf, recvcnt,recvtype,source,comm)`
MPI_Gather	Gathers distinct messages from each task in the group to a sing

	destination task. This routine is the reverse operation o MPI_Scatter. `MPI_Gather(&sendbuf,sendcnt,sendtype,&recvbuf, recvcount,recvtype,source,comm)`

APPENDIX C: MPI Send Receive


SENDING & RECEIVING MESAGES
SWITCH/HUB

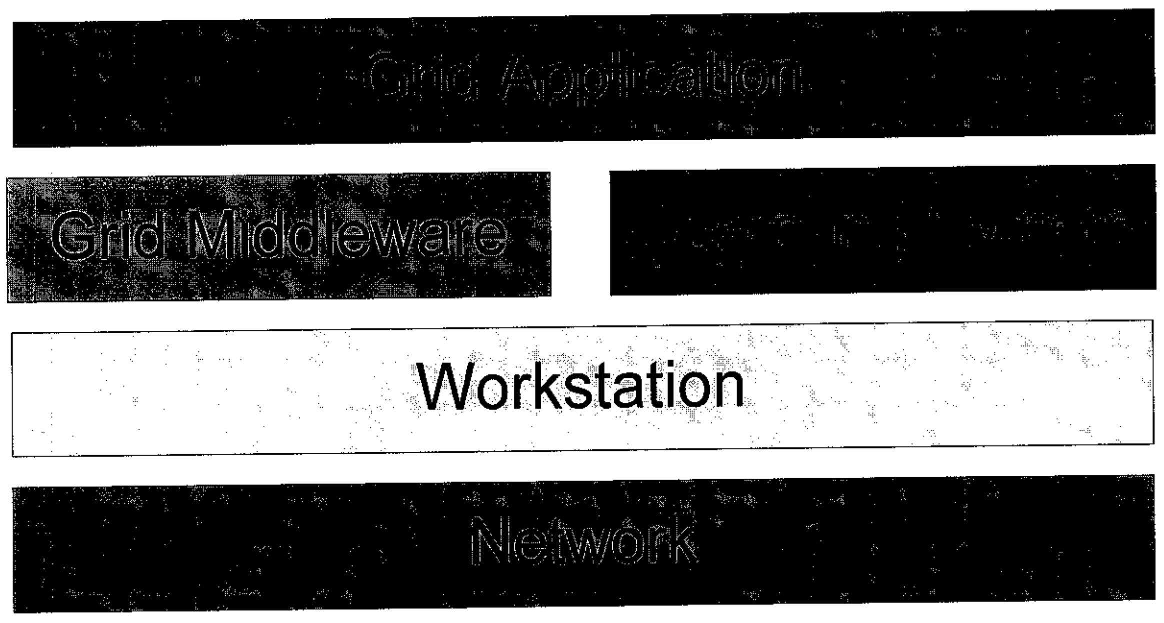
APPENDIX D. Grid Design



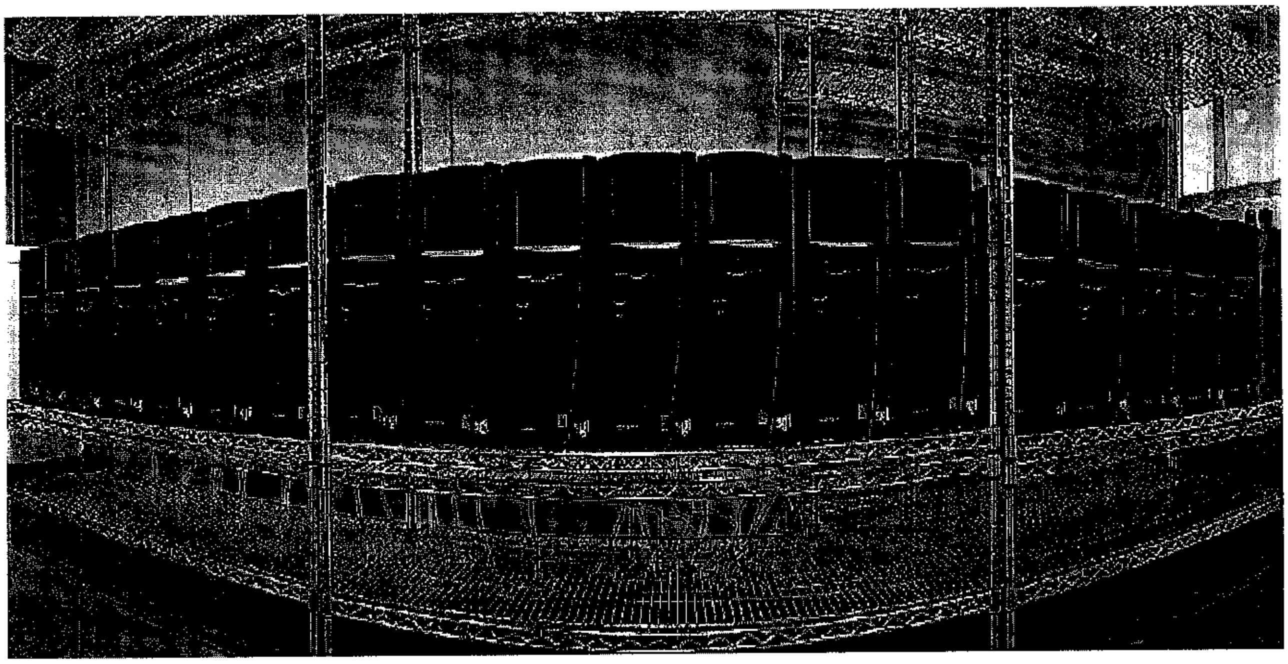
Legend

Symbol	Quantity	Description
	1	Router
	1	Switch
	5	Idle Workstation
	1	User

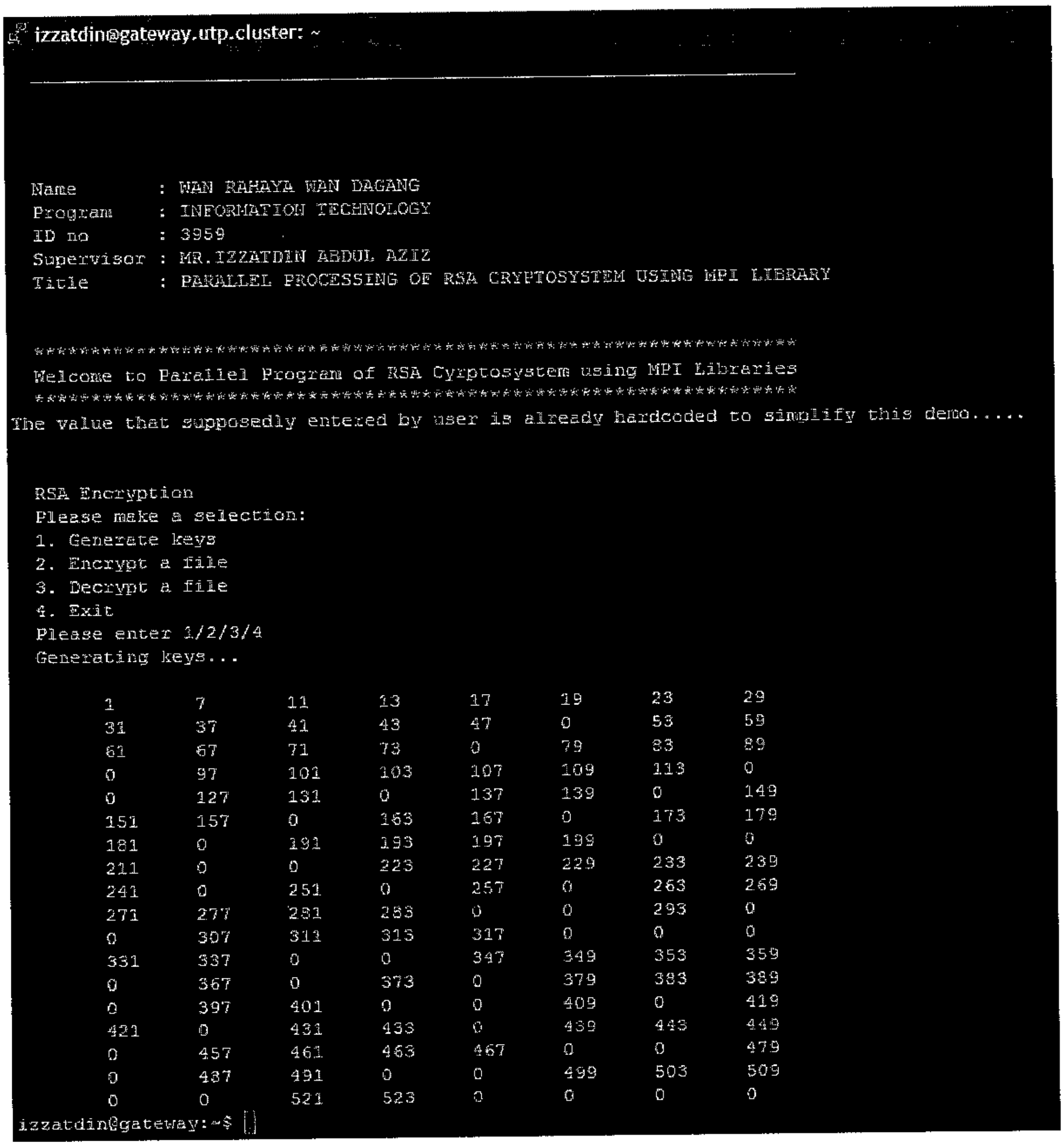
APPENDIX E: Grid Layer


Grid Application
Grid Middleware
Workstation
Network

APPENDIX F: Grid Computers Running OpenMosix

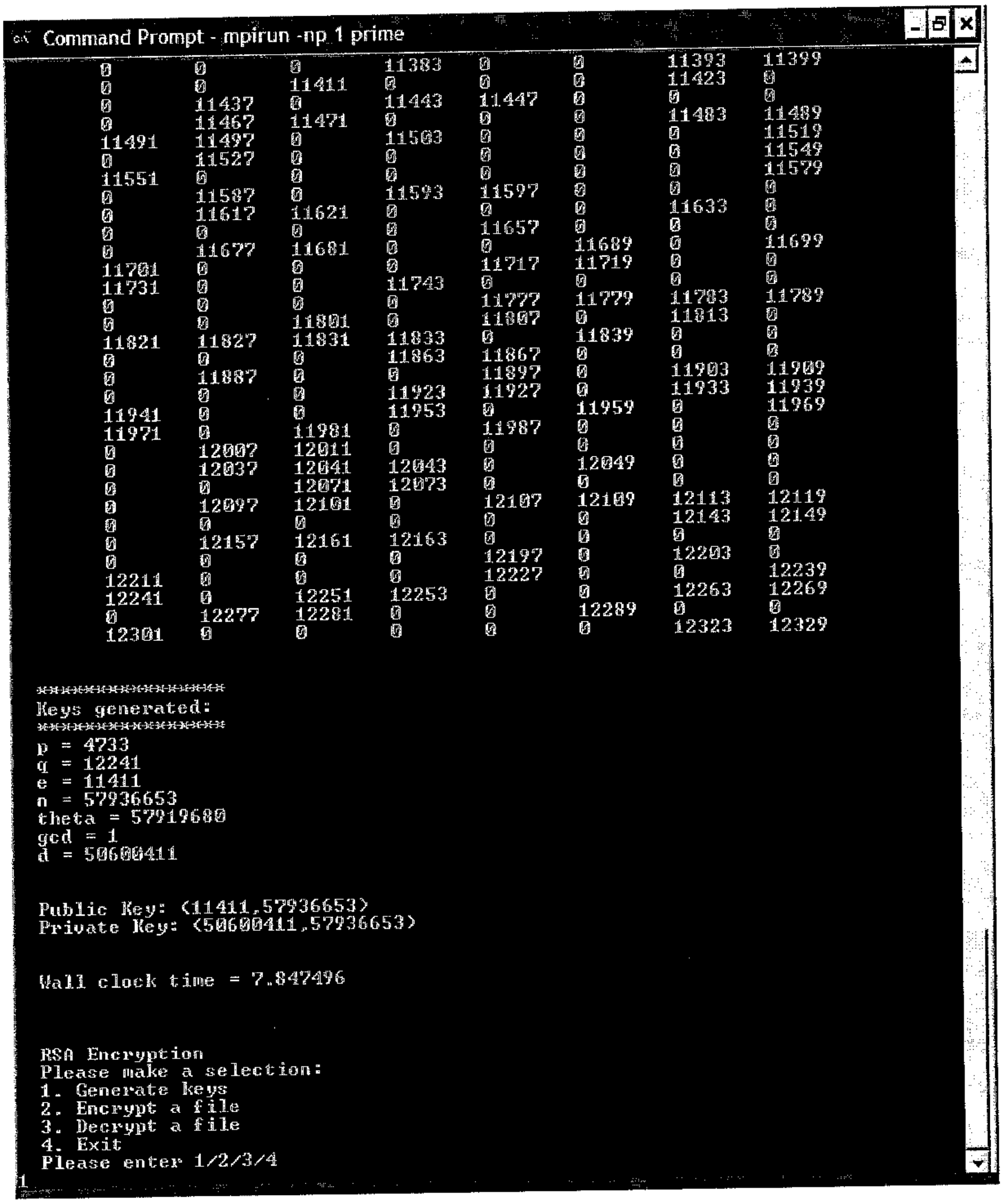


APPENDIX G: Generating Prime Numbers



```
izzatdin@gateway.utp.cluster: ~

  Name       : WAN RAHAYA WAN DAGANG
  Program    : INFORMATION TECHNOLOGY
  ID no      : 3959
  Supervisor : MR.IZZATDIN ABDUL AZIZ
  Title      : PARALLEL PROCESSING OF RSA CRYPTOSYSTEM USING MPI LIBRARY


  *********************************************************************
  Welcome to Parallel Program of RSA Cyrptosystem using MPI Libraries
  *********************************************************************
The value that supposedly entered by user is already hardcoded to simplify this demo.....



  RSA Encryption
  Please make a selection:
  1. Generate keys
  2. Encrypt a file
  3. Decrypt a file
  4. Exit
  Please enter 1/2/3/4
  Generating keys...

        1       7       11      13      17      19      23      29
        31      37      41      43      47      0       53      59
        61      67      71      73      0       79      83      89
        0       97      101     103     107     109     113     0
        0       127     131     0       137     139     0       149
        151     157     0       163     167     0       173     179
        181     0       191     193     197     199     0       0
        211     0       0       223     227     229     233     239
        241     0       251     0       257     0       263     269
        271     277     281     283     0       0       293     0
        0       307     311     313     317     0       0       0
        331     337     0       0       347     349     353     359
        0       367     0       373     0       379     383     389
        0       397     401     0       0       409     0       419
        421     0       431     433     0       439     443     449
        0       457     461     463     467     0       0       479
        0       487     491     0       0       499     503     509
        0       0       521     523     0       0       0       0
izzatdin@gateway:~$ 
```

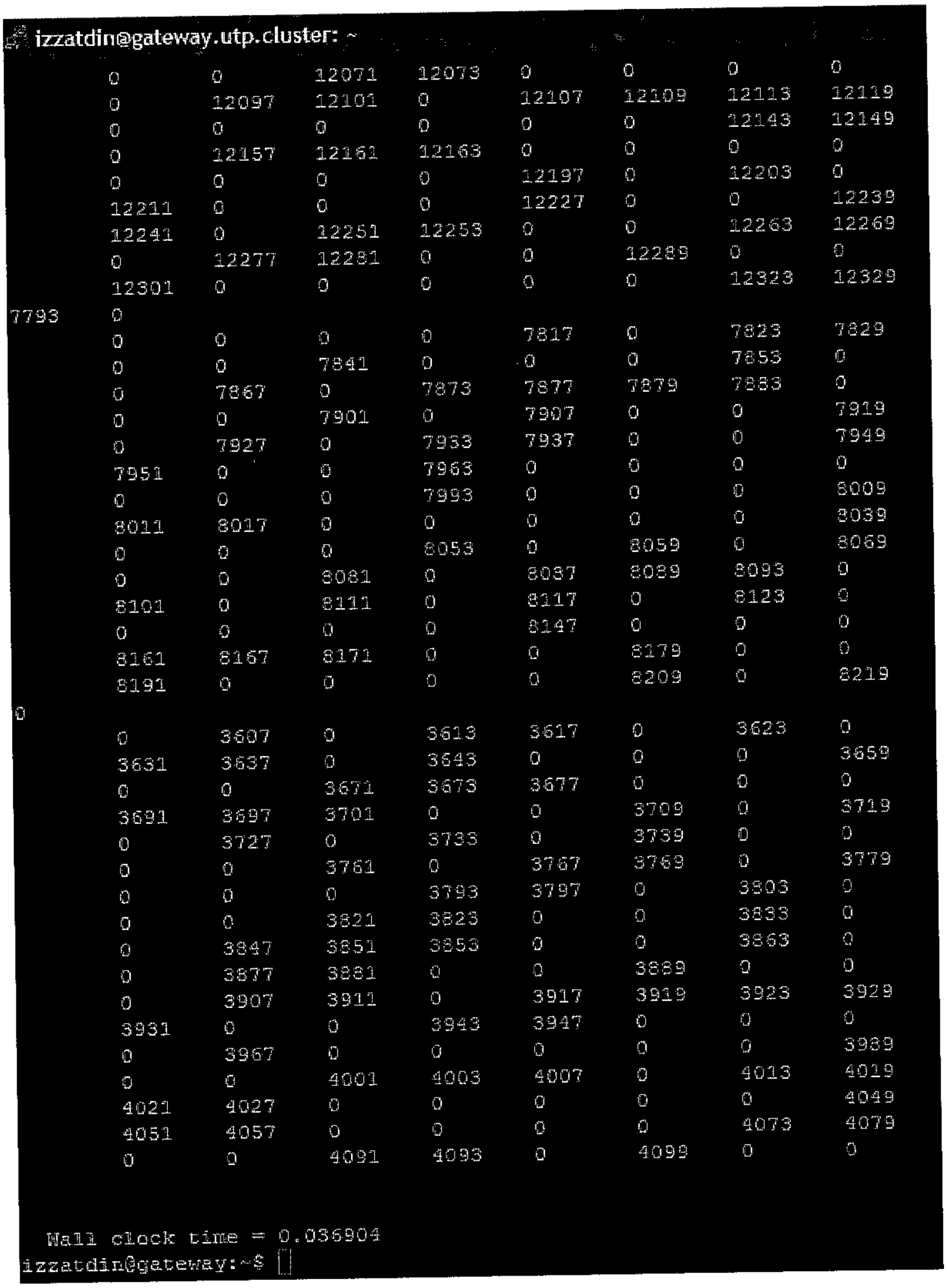
APPENDIX H: Generating Prime Numbers on One Processor



Command Prompt - mpirun -np 1 prime

```
        0       0       0       11383   0       0       11393   11399
        0       0       11411   0       0       0       11423   0
        0       11437   0       11443   11447   0       0       0
        0       11467   11471   0       0       0       11483   11489
        11491   11497   0       11503   0       0       0       11519
        0       11527   0       0       0       0       0       11549
        11551   0       0       0       0       0       0       11579
        0       11587   0       11593   11597   0       0       0
        0       11617   11621   0       0       0       11633   0
        0       0       0       0       11657   0       0       0
        0       11677   11681   0       0       11689   0       11699
        11701   0       0       0       11717   11719   0       0
        11731   0       0       11743   0       0       0       0
        0       0       0       0       11777   11779   11783   11789
        0       0       11801   0       11807   0       11813   0
        11821   11827   11831   11833   0       11839   0       0
        0       0       0       11863   11867   0       0       0
        0       11887   0       0       11897   0       11903   11909
        0       0       0       11923   11927   0       11933   11939
        11941   0       0       11953   0       11959   0       11969
        11971   0       11981   0       11987   0       0       0
        0       12007   12011   0       0       0       0       0
        0       12037   12041   12043   0       12049   0       0
        0       0       12071   12073   0       0       0       0
        0       12097   12101   0       12107   12109   12113   12119
        0       0       0       0       0       0       12143   12149
        0       12157   12161   12163   0       0       0       0
        0       0       0       0       12197   0       12203   0
        12211   0       0       0       12227   0       0       12239
        12241   0       12251   12253   0       0       12263   12269
        0       12277   12281   0       0       12289   0       0
        12301   0       0       0       0       0       12323   12329


*****************
Keys generated:
*****************
p = 4733
q = 12241
e = 11411
n = 57936653
theta = 57919680
gcd = 1
d = 50600411


Public Key: (11411,57936653)
Private Key: (50600411,57936653)


Wall clock time = 7.847496



RSA Encryption
Please make a selection:
1. Generate keys
2. Encrypt a file
3. Decrypt a file
4. Exit
Please enter 1/2/3/4
1
```

APPENDIX I: Generating Prime Numbers on Three Processors



```
izzatdin@gateway.utp.cluster: ~
         0       0       12071   12073   0       0       0       0
         0       12097   12101   0       12107   12109   12113   12119
         0       0       0       0       0       0       12143   12149
         0       12157   12161   12163   0       0       0       0
         0       0       0       0       12197   0       12203   0
         12211   0       0       0       12227   0       0       12239
         12241   0       12251   12253   0       0       12263   12269
         0       12277   12281   0       0       12289   0       0
         12301   0       0       0       0       0       12323   12329
7793     0
         0       0       0       0       7817    0       7823    7829
         0       0       7841    0       0       0       7853    0
         0       7867    0       7873    7877    7879    7883    0
         0       0       7901    0       7907    0       0       7919
         0       7927    0       7933    7937    0       0       7949
         7951    0       0       7963    0       0       0       0
         0       0       0       7993    0       0       0       8009
         8011    8017    0       0       0       0       0       8039
         0       0       0       8053    0       8059    0       8069
         0       0       8081    0       8087    8089    8093    0
         8101    0       8111    0       8117    0       8123    0
         0       0       0       0       8147    0       0       0
         8161    8167    8171    0       0       8179    0       0
         8191    0       0       0       0       8209    0       8219
0
         0       3607    0       3613    3617    0       3623    0
         3631    3637    0       3643    0       0       0       3659
         0       0       3671    3673    3677    0       0       0
         3691    3697    3701    0       0       3709    0       3719
         0       3727    0       3733    0       3739    0       0
         0       0       3761    0       3767    3769    0       3779
         0       0       0       3793    3797    0       3803    0
         0       0       3821    3823    0       0       3833    0
         0       3847    3851    3853    0       0       3863    0
         0       3877    3881    0       0       3889    0       0
         0       3907    3911    0       3917    3919    3923    3929
         3931    0       0       3943    3947    0       0       0
         0       3967    0       0       0       0       0       3989
         0       0       4001    4003    4007    0       4013    4019
         4021    4027    0       0       0       0       0       4049
         4051    4057    0       0       0       0       4073    4079
         0       0       4091    4093    0       4099    0       0


  Wall clock time = 0.036904
izzatdin@gateway:~$ 
```

APPENDIX J: Source Code

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <fstream.h>
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
#include <time.h>

int GCD (int, int);
int FastExp(int,int, int);
const int maxsize = 50000;
int r[maxsize]; //hold remainders
int q[maxsize]; //hold quotients
int top = 0;
int bottom = 0;
int ExtEuclid ();
int readFile(char filename []);
int writeFile();
void IntBin(int);
const int MAXLENGTH = 20;
const int bytesize = 100;
char filename[MAXLENGTH] ;
int numericalText;
int bin[bytesize];
int rearrange[10];
int index=0;
int cipherText;


int main(int argc, char *argv[])
{
        int maxCols = 8;
        int maxRows;
        int temp[8];
        int buffer[8];
        int **table;
        int myid, numprocs, numsent, slave, sender, rowrecvd, row, col, sr, modnum, max,row2;
        int n,theta;
        int p=0, q=0, e=0, gcd=0, d=0;


        MPI_Status stat;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  if(myid == 0)
  {
        cout<<endl<<endl;
        cout<<"Name        : WAN RAHAYA WAN DAGANG"<<endl;
        cout<<"Program : INFORMATION TECHNOLOGY"<<endl;
        cout<<"ID no        : 3959"<<endl;
        cout<<"Title        : PARALLEL PROCESSING OF RSA CRYPTOSYSTEM USING MPI
LIBRARY"<<endl<<endl<<endl;
        cout<<"*********************************************************************"<<endl;
        cout<<"Welcome to Parallel Program of RSA Cyrptosystem using MPI Libraries"<<endl;
        cout<<"*********************************************************************"<<endl<<endl;

        //user is prompt to enter a number that determines the strength of the key prime
        //that is going to be generated
        cout<<"Enter the max value between 10000 and 99999"<<endl;
   cin>>max;

        maxRows = max;
```

```
if(max > 99999 || max < 10000){exit(0);}
system("CLS");

        table = (int **)calloc(maxRows, sizeof(int*));

        table[0] = (int*)calloc(maxCols, sizeof(int));

                table[0][0] = 1;
                table[0][1] = 7;
                table[0][2] = 11;
                table[0][3] = 13;
                table[0][4] = 17;
                table[0][5] = 19;
                table[0][6] = 23;
                table[0][7] = 29;

        for(int q=1; q<maxRows; q++)
        {
                table[q] = (int*)calloc(maxCols, sizeof(int));
        }

        for(int a=0; a<maxCols; a++)
        {
                temp[a] = table[0][a];    //a temp array to save row[0] of table[maxRows][maxCols]
        }

        //broadcast row[0] to all nodes
        MPI_Bcast(temp, maxCols, MPI_INT, 0, MPI_COMM_WORLD);

        numsent = 1;
        slave = 1;
        int k=1;

        //master despathces 1 row from table to each node accordingly; tag with row no
        for(int i=0; i<numprocs; i++)
        {

                        for(int j=0; j<maxCols; j++)
                        {
                                buffer[j] = table[k][j];
                        }
                        k++;


                MPI_Send(buffer, maxCols, MPI_INT, slave, numsent, MPI_COMM_WORLD);
                numsent++; slave++;
        }

        //master waits result from slave and send another row of table for another processing
        for(int x=0; x<maxRows; x++)
        {
                MPI_Recv(buffer, maxCols, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&stat);
                sender = stat.MPI_SOURCE;
                rowrecvd = stat.MPI_TAG;

                //retrieve each element from buffer array and save into table
                for(int y=0; y<maxCols; y++)
                {
                        table[rowrecvd][y] = buffer[y];
                }
                if(numsent<maxRows)
                {
                        for(int h=0; h<maxCols; h++)
                        {
                                buffer[h] = table[numsent][h];
                        }
                        MPI_Send(buffer, maxCols, MPI_INT, sender, numsent, MPI_COMM_WORLD);
                        numsent++;
                }
```

```
                    else
                    {
                                MPI_Send(buffer, maxCols, MPI_INT, sender, -1, MPI_COMM_WORLD);
                    }
          }

}else //slave's part
{
          do{
                    MPI_Recv(buffer, maxCols, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
                    if(stat.MPI_TAG != -1)
                    {
                              row = stat.MPI_TAG;

                              //filtering the table array for primes:

                              for(int v=0; v<maxCols; v++)
                              {
                                        //populate buffer with odd numbers
                                        buffer[v] = (30*row) + temp[v];
                                        sr = sqrt(buffer[v]);           //determine primality of odd no

                                        for(int c=2; c<sr+2; c++)
                                        {
                                                  modnum = buffer[v]%c;
                                                  if(modnum ==0)      //if odd no is not prime then NULL
                                                  {
                                                            c=sr+3;
                                                            buffer[v] = NULL;
                                                  }
                                        }
                              }
                              MPI_Send(buffer, maxCols, MPI_INT, 0, row, MPI_COMM_WORLD);
                    }
          }while(stat.MPI_TAG != -1);
}

//master prints result
if(myid ==0)
{
          for(int g=0; g<maxRows; g++)
          {
                    for(int s=0; s<maxCols; s++)
                    {
                              cout<<setw(5)<<table[g][s];
                    }
          }


 srand((unsigned)time(NULL)); //seed random number generator with clock
  while(p == 0) //until suitable primes are picked
  {
          row2 = rand() % max;
    col = rand() % 8;
    p = table[row][col];//assigns prime to value in matrix
  }

          srand((unsigned)time(NULL));
  while (q<= p && q <= max)
  {
                    while(q == 0) //until suitable primes are picked
                    {
                              row2 = rand() % max;
                              col = rand() % 8;
                              q = table[row][col];//assigns prime to value in matrix
                    }
  }

  n = p*q;
  theta = (p-1)*(q-1);
 srand((unsigned)time(NULL)); //srand() is used to pick a number from the array randomly


while(e==q ||e == 0)    //to avoid e == q
{
                        while(e == 0) //until suitable primes are picked
                        {
                                    row2 = rand() % max;
                                    col = rand() % 8;
                                    e = table[row][col];//assigns prime to value in matrix
                        }
}

gcd = GCD(e,theta); //determine gcd
d = ExtEuclid();
if(d<0)
{
  d = theta - abs(d);
}


 cout<<endl;
 cout<<endl;
 cout<<" P value = " <<p<<endl;
 cout<<" Q value = "<<q<<endl;
 cout<<" N value = "<<n<<endl;
 cout<<" E value = "<<e<<endl;
 cout<<" Modulus N is "<<theta<<endl;
 cout<<" GCD is "<<gcd<<endl;
 cout<<" D is "<<d<<endl;

           cout<<endl<<endl;

           //starting to encrypt a file:
           cout<<"Enter file name : "<<endl;
  cin >>filename;
  numericalText = readFile(filename);
  cout<<"File content : "<<numericalText<<endl;
           IntBin(numericalText);
  cipherText = FastExp(numericalText,e,n);
  writeFile();

           //starting to decrypt a file:
           cout<<"Enter file name : "<<endl;
  cin >>filename;
           numericalText = readFile(filename);
  cout<<"File content : "<<numericalText<<endl;
           IntBin(numericalText);
  cipherText = FastExp(numericalText,d,n);
  writeFile();

}


 getch();

 MPI_Finalize();
 free (table);
 return 0;

}
//************************************************************
//this function will take 2 parameters (e and theta) and
//solve the GCD from Euclid's Algorithm
//GCD of two integers which are not both zero is the largest
//integer that divides both numbers.
//This function is using 'Table' method
//************************************************************
```

```
int GCD(int a, int b)  //e,theta
{
 int rem=0, gcd=0, i=0, k=0;
            //initialize array of remainders and quotients
            for(k=0; k<maxsize; k++)
            {
                        r[k]=(0);   //creating an array of a table consist of remainders and quotients
    q[k]=(0);
            }
            r[i] = b; //first remainder value; starting value in the table
            i++;
            while ( b%a != 0)  //doesnt divide; both are prime
            {
            r[i] = a;  //second remainder value
    rem = b%a;
    q[i] = b/a; // q starts at q[1], no value for q[0]
    b = a;    //next 'b'
    a = rem;  //next 'a'
    i++;
            }
   gcd =  rem;
   if( i == 1)
   {
            gcd = a;    //both no are not prime
   }
   r[i] = a;                //store last value of r[i] = a as the compiler is stopped looping
   bottom = i+1;            //store length of array for ExtEuclid loop

   return gcd;
 }

//***********************************************************************
//This function will take 2 parameters sent in and solve a two variable
//equation using the Extended Euclid Algorithm.
//This function will return the value of d
//This function is also used the 'Table' method
//***********************************************************************

 int ExtEuclid()
 {
            int d=0, k=0, length=0, i=0;
            const int max = 5000;
   int x[max];
   int y[max];

   for(k=0; k<max; k++)
   {
            x[k]=(0);   //creating an array of a table consist of x;coefficient of theta
    y[k]=(0);   //and y;coefficient of e
   }
   x[0] = 1;
   x[1] = 0;
   y[0] = 0;
   y[1] = 1;

   length = bottom; //which is the total no of remainders

   for(i=2; i<length; i++)    //i starts at 2 to calculate the 3rd value of both coefficients
   {
            x[i] = x[i-2] - (q[i-1]*x[i-1]);  //the next x value until the last value of remainder is reached
     y[i] = y[i-2] - (q[i-1]*y[i-1]);  //the next y value until the last value of remainder is reached
   }

   d = y[i-1]; //the coefficient of e; satisfy the equation of: de mod theta = 1 OR d=e^-1 mod theta
   return d;
  }

  int readFile(char filename[])
  {
        ifstream inStream;

        inStream.open("infile.txt");
        if (inStream.fail())
        {
              cout<<"Input file opening failed.\n";
          exit(1);
        }

        int numeric;
        inStream>>numeric;

        inStream.close();

        return numeric;
}

int writeFile()
{

        ofstream outStream;

        outStream.open("outfile.txt");
        if (outStream.fail())
        {
              cout<<"Output file opening failed.\n";
          exit(1);
        }


        outStream<<cipherText;

        outStream.close();

        return 0;
}

//binary method is used to reduce the complexity of modular multiplication

void IntBin(int M2)
{
  int tmp=1,remain=1;
  int c=0,i=0,j=0,test=0;
  bool first(true);

  cout<<"Binary conversion: ";

  for (i = 0; i < bytesize; i++)
  {
                          bin[i] = 0; //initialize array to 0
  }
  M2 = abs(M2);
  while (M2 != 0)
  {
                          while (tmp <= M2)
                          {
                                      tmp = tmp * 2;
        c++; //gets power
      }

      test = M2%2; //determine if even or odd
      if (c == 0 && test==0)//special case; ones place
      {
                                      bin[c]=0;//even number, so assign one's place as 0
      }
      else
                          {bin[c]=1;}

      for (j=1; j<c; j++)
```