

Investigation Performance of Strassen Matrix Multiplication Algorithm on Distributed Systems

Reza Abri Vaighan

Submitted to the
Institute of Graduate Studies and Research
In partial fulfillment of the requirements for the Degree of

Master of Science
In
Computer Engineering

Eastern Mediterranean University
August 2013
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Elvan Yılmaz
Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Computer Engineering Department.

Assoc. Prof. Dr. Muhammed Salamah
Chair, Department of
Computer Engineering Department

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Computer Engineering Department.

Asst. Prof. Dr. Gürcü Öz
Supervisor

Examining Committee

1. Assoc. Prof. Dr. Alexander Chefranov

2. Asst. Prof. Dr. Ahmet Ünveren

3. Asst. Prof. Dr. Gürcü Öz

ABSTRACT

Parallel computation is the concurrent performance of a task with multiple processors in order to obtain rapid results. This method is based on that the process of solving a problem can usually be divided into smaller problem parts and with some coordination, these solution parts perform simultaneously.

Simply put, parallel computing is the concurrent use of different computing resources for solving a computational problem. Parallel computing saves time, solves large problems efficiently and is cost-effective or non-local sources. There are two important models in the architecture of parallel computing:

- I. Shared memory: In this multiprocessor system, all of the allocated processors can access to a common memory.
- II. Message passing: In this multiprocessor system, each processor has its own local memory; processors exchange messages and share data through an internal connection network.

In this thesis Strassen recursive algorithm is implemented for multiplying square matrices in parallel form for a distributed homogeneous system in order to improve its execution time. Strassen multiplying algorithm is a divide and conquer problem, with temporal complexity $O(n^{2.81})$.

Since this algorithm is recursive, total parallelism is impossible thus, matrices must be divided and distributed according to a special distribution topology in which affects on the performance time.

This thesis represents an economical distribution topology with distributing matrices, which minimize the multiplication time of matrices in a parallel environment. Dividing and distributing matrices according to a basic distribution topology (two-fold distribution), led to favorable and unfavorable results. To improve the results, the matrix distribution topology needs to be changed.

Finding a desirable and convenient topology is necessary aiming to achieve suitable results by considering matrices dimensions and the number of nodes. So, this method is expected to reduce the execution time in comparison with Strassen-BMR method.

Keywords: Parallel Computation, Message Passing, Strassen Algorithm, Divide and Conquer, Topology

ÖZ

Paralel hesaplama, hızlı sonuç elde etmek amacıyla, bir görevin birden fazla işlemci tarafından eşzamanlı hesaplanmasıdır. Bu yöntem, genellikle, büyük bir problemi küçük parçalara ayırıp çözüme geçene dayanmaktadır. Ve bu parçaların çözümü, bazı koordinasyonlarla, aynı anda gerçekleştirilir.

Basitçe söylemek gerekirse, paralel hesaplama sistemi bir hesaplama problemi çözmek için farklı işlem kaynaklarının eşzamanlı kullanılmasıdır. Paralel hesaplama sistemi, zaman kazandıran, büyük problemleri verimli bir şekilde çözen, düşük maliyetli, yerel olmayan kaynaklardır. Paralel hesaplama mimarisi için iki önemli model kullanılmaktadır:

- I. Paylaşılan bellek: Bu çok işlemcili sistemde, tüm tahsis edilen işlemciler ortak bir belleğe erişebilir.
- II. Mesaj geçen: Bu çok işlemcili sistemde, her işlemcinin kendi yerel hafızası vardır; işlemciler dahili bir bağlantıyla ağ üzerinden mesaj alış verisi yaparak veri paylaşabilirler.

Bu tezde, Strassen'in özyinelemeli algoritması, kare matrislerin çarpımı için, paralel şekilde dağıtılmış homojen bir sistemde, yürütme süresini iyileştirilmek amacıyla mesaj geçişi modeliyle uygulanmıştır. Strassen çarpım algoritması zamansal karmaşıklığı $O(n^{2.81})$ ile, problemi böl ve yönet (divide and conquer) yöntemidir.

Bu algoritma özyinelemeli olduđu için, tamamen eşzamanlı yapılması imkansızdır. Bu nedenle, yürütme süresini azaltmak için, matrisler özel bir dağıtım topolojisine göre bölünüp dağıtılmalıdır.

Bu tez, paralel bir ortamda, matrislerin çarpma süresini azaltmak maksadıyla, ekonomik bir dağıtım topolojisi önermektedir. Matrisleri temel bir dağıtım topolojisiyle (ikili dağıtım) bölüp ağ üzerinde dağıtmak, olumlu ve olumsuz sonuçlara yol açar. Sonuçları iyileştirmek için, matris dağıtım topolojisinin iyileştirilmesi gerekmektedir.

İstenilen bir sonuç elde etmek için, matris boyutları ve bilgisayar sayısı dikkate alınarak, arzu edilen, uygun bir topoloji bulunması gerekmektedir. Bu tezde, önerilen bir topoloji üzerinde Strassen algoritması uygulanmıştır. Elde edilen sonuçlara göre, önerilen yöntem ve topoloji önceki yöntemlerle karşılaştırıldığında yürütme zamanında azalma olduğu tespit edilmiştir.

Anahtar Kelimeler: Paralel Hesaplama, Mesaj Geçen, Strassen Algoritması, Böl ve Yönet, Topoloji

Dedicated to my family with love

ACKNOWLEDGMENTS

I have taken great deal of efforts in this thesis. Although, its accomplishment could not be possible without effective and helpful support of my dear supervisor Asst. Prof. Dr. Gürcü Öz. In fact she was the tower of strength and knowledge to fulfill this thesis. Furthermore, I would like to extend my honest thanks to all who contributed to finalize this academic mission.

Worth mentioning the extremely respect to Assoc. Prof. Dr. Alexander Chefranov and Asst. Prof. Dr. Ahmet Ünveren who kept track of my progress during my master degree.

In addition, I should send my great respect to my adored parents and lovely siblings who were a strong source of love and concern support in all these years.

TABLE OF CONTENTS

ABSTRACT	iii
ÖZ	v
ACKNOWLEDGMENTS	viii
LIST OF FIGURES	xi
LIST OF TABLES	xiii
LIST OF SYMBOLS/ABBREVIATIONS	xv
1 INTRODUCTION	1
2 PARALLEL AND DISTRIBUTED PROGRAMMING	4
2.1 Parallel Processing	4
2.2 Parallel Computers Architecture	6
2.2.1 Shared Memory Systems	8
2.2.2 Distributed Memory Systems	8
2.3 Internal Communication Network	9
2.4 Parallel Programming Models	9
2.4.1 Shared Memory Model	10
2.4.2 Message Passing Model	11
2.5 Parallel Algorithms	12
2.5.1 Parallel Algorithm Design	13
2.6 Performance Evaluation in Parallel Systems	13
3 MATRIX MULTIPLICATION ALGORITHMS AND RELATED WORKS	16
3.1 Reviews of Matrices Multiplication Using Divide-and-Conquer Method	17
3.2 Considering Matrix Multiplication by Use of Strassen Method	18
3.3 Related Works	23

4 STRASSEN PARALLEL MATRIX MULTIPLICATION ALGORITHM IN DISTRIBUTED SYSTEM.....	27
4.1 Two-fold Distribution Method	28
4.1.1 Reusing Waiting Node.....	29
4.1.2 Performance	30
4.2 Seven-fold Distribution Method.....	31
4.3 Dynamic Distribution Method.....	33
4.3.1 Performance Evaluation of Dynamic Distribution Method.....	39
4.4 Fair Distribution Method.....	40
5 EXPERIMENTAL RESULTS.....	45
5.1 Comparison of Usual and Reuse of Waiting Clients Methods.....	45
5.2 Comparison of Two-fold, Seven-fold and Dynamic Distribution Methods.....	47
5.3 Performance of Dynamic Distribution Method.....	50
5.4 Performance of Two-fold and Seven-fold Distribution Method	53
5.6 Performance of Fair Distribution Method	57
6 CONCLUSION	63
REFERENCES.....	66
APPENDICES	71
APPENDIX A: User Guide	72
APPENDIX B: Programming Part	78

LIST OF FIGURES

Figure 2.1: Sequential Computing	4
Figure 2.2: Parallel Computing	5
Figure 2.3: Types of MIMD Architecture.....	8
Figure 2.4: PRAM Model for Parallel Computing	11
Figure 4.1: Structure of Two-fold Distribution Method	28
Figure 4.2: Structure of Reusing of the Waiting Node in Distribution.....	30
Figure 4.3: Structure of Seven-fold Distribution Method.....	31
Figure 4.4: Some Samples of Dynamic Distribution Method.....	35
Figure 4.5: Flowchart of the Client Program in Dynamic Distribution Method.....	37
Figure 4.6: Flowchart of Server Program in Dynamic Distribution Method.....	38
Figure 4.7: An Example of Fair Distribution Method	41
Figure 4.8: Flowchart of Client Program in Fair Distribution Method.....	43
Figure 4.9: Flowchart of Server Program in Fair Distribution Method.....	44
Figure 5.1: Execution Time versus Number of Computers for Usual and Reuse of Waiting Clients by Two-fold Distribution Method.....	46
Figure 5.2: Execution Time versus Number of Computers for Three Different Distribution Method.....	49
Figure 5.3: Execution Time versus Number of Computers with Different Threshold Values for Dynamic Distribution Method	51
Figure 5.4: Execution Time versus Number of Computers with Different Matrix Size for Dynamic Distribution Method.....	53
Figure 5.5: Execution Time versus Number of Computers with Different Threshold Values for Two-fold Distribution Method	54

Figure 5.6: Execution Time versus Number of Computers with Different Matrix Size for Two-fold Distribution Method	55
Figure 5.7: Execution Time versus Number of Computers with Different Threshold Values for Seven-fold Distribution Method.....	56
Figure 5.8: Execution Time versus Number of Computers with Different Matrix Size for Seven-fold Distribution Method.....	57
Figure 5.9: Execution Time versus Number of Computers with Different Threshold Values for Fair Distribution Method.....	59
Figure 5. 10: Execution time versus Number of Computers with Different Matrix Size for Fair Distribution Method	60

LIST OF TABLES

Table 2.1: Comparison of Standard and Strassen Matrix Multiplication Algorithms	19
Table 5.1: Execution Time for Usual and Reuse of Waiting Clients by Two-fold Distribution Method	46
Table 5.2: Execution Time of Three Different Distribution Method.....	48
Table 5.3: Speed-Up and Efficiency of Three Different Distribution Methods	50
Table 5.4: Execution Time of Dynamic Distribution Method by Different Threshold Values and Using Different Number of Computers.....	51
Table 5.5: Execution Time of Dynamic Distribution Method by Different Matrix Size and Using Different Number of Computers	52
Table 5.6: Execution Time of Two-fold Distribution Method by Different Threshold Values and Using Different Number of Computers.....	54
Table 5.7: Execution Time of Two-fold Distribution Method by Different Matrix Size and Using Different Number of Computers	54
Table 5.8: Execution Time of Seven-fold Distribution Method by Different Threshold Values and Using Different Number of Computers	55
Table 5.9: Execution Time of Seven-fold Distribution Method by Different Matrix Size and Using Different Number of Computers	56
Table 5.10: Execution Time, Speed-Up and Efficiency of Fair Distribution Method by Different Number of Computers	58
Table 5.11: Execution Time of Fair Distribution Method by Different Threshold Values and Using Different Number of Computers.....	59
Table 5.12: Execution Time of Fair Distribution Method by Different Matrix Size Using Different Number of Computers.....	60

Table 5.13: Comparing Execution Time of Strassen-BMR and Fair Distribution

Methods..... 62

LIST OF SYMBOLS/ABBREVIATIONS

VLSI	Very Large-Scale Integration
CPU	Central Processing Unit
FLOPS	Floating-Point Operation Per Second
ENIAC	Electronic Numerical Integrator and Computer
RAM	Random-Access Memory
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
MISD	Multiple Instructions Single Data
MIMD	Multiple Instructions Multiple Data
PRAM	Parallel Random Access Machine
EREW	Exclusive Read, Exclusive Write
ERCW	Exclusive Read, Concurrent Write
CREW	Concurrent Read, Exclusive Write
CRCW	Concurrent Read, Concurrent Write

Chapter 1

INTRODUCTION

Systems with high processing are needed to create applications that require high speed processing. Semiconductor and VLSI [1] technology have made improvements in single processor machine tasks. However, these systems is still not suitable for science and engineering applications that require high speed computations, such as aerodynamic affairs, real-time systems, medical signals processing and aerology. In addition, there are limitations in CPU clock maximum speed. It has led to the development of parallel computers that can process data at speeds of large numbers floating points operation per second (FLOPS).

In 1945, ENIAC [2] the first electronic processor performed 1000 instructions per second. Now a days, the new generation of Risk processors are able to process hundreds of millions per second. These processors are sequential but fast.

About ten years ago, computer manufacturers achieved another economical way to reach the equal power of n witch the use of n processors led to the design of multi-processor systems. They can combine in multi-processor systems, to increase the power as necessary. Improved VLSI processor's design, causes to faster blocks of parallel processors [3].

In recent years, parallel processor systems have developed based on personal computers. These systems offer better efficiency in comparison with supercomputers, and their software and operating systems are readily available.

Parallel computers may have 10 to 50,000 processors that work with each other in parallel form. If a processor can perform more than 10 million instructions in one second, 10 processors can perform 100 million recipes in one second. Parallel computer systems allow for sharing data and creating relationships. There are two important architectures in this field: Shared memory and Message passing [4]. Each of these architectures has its own advantages and disadvantages.

Many software systems are designed for parallel computer programming at the operational system levels and also in programming languages. These systems create a mechanism for dividing the problems into separate tasks.

These mechanisms may be implicitly parallel (system automatically divides the problem and specializes tasks) or explicitly parallel (programmer describes how to divide the problem).

The aim of this thesis is to examine parallel and distributed programming in a homogeneous computer network and optimize performance in this environment. In a homogeneous network, all available computers have the same characteristics. The message passing architecture is used in this parallel environment.

The recursive algorithm is chosen for implementation. The parallelize possibility of recursive algorithms is less than for sequential algorithms. Because the division and

distribution of a task needs maximum overlap, it is important to optimize performance in a parallel environment. The Strassen matrix multiplication algorithm has been chosen for this thesis. In this algorithm problem is divided to seven sub-problems (tasks) and these tasks are divided between computers. Any of these seven multiplication tasks could be divided recursively, to seven more sub-tasks. Computation is done in each state and result is returned to the previous stage recursively.

Distribution of a given problem in a network has significant impact on the running time of the algorithm due to the distribution in different topology, the overlapping rate of computations on different computers varies.

Different problem situations and inputs must define the optimal particular distribution topology. Defining all appropriate distribution topologies for these states is very difficult, so a program should produce a suitable distribution topology according to different situations and inputs.

This thesis includes five chapters. Second chapter reviews basic concepts of parallel and distributed programming. Chapter three presents tasks and algorithms in the field of parallel Strassen matrix multiplication. Chapter four describes the stages of this project and problem solutions. Chapter 5 provides the results, followed by conclusions and appendices at the end.

Chapter 2

2 PARALLEL AND DISTRIBUTED PROGRAMMING

This chapter presents a brief overview of parallel processing, the importance and its usage.

2.1 Parallel Processing

Parallel computing refers to the simultaneous execution of a program on multiple processors in order to achieve faster results. In sequential computing, instructions run orderly in processors; the running speed is proportional to the processor speed (Figure 2.1). In parallel processing, instructions run in several processors, but speed of whole parallel system is not necessarily equal to CPU speed of one processor multiplied by the number of processors (Figure 2.2). Parallel computation can be employed in different parts of the computer, such as software and hardware; therefore, computing generalities should attend to different aspect of software and hardware [5].

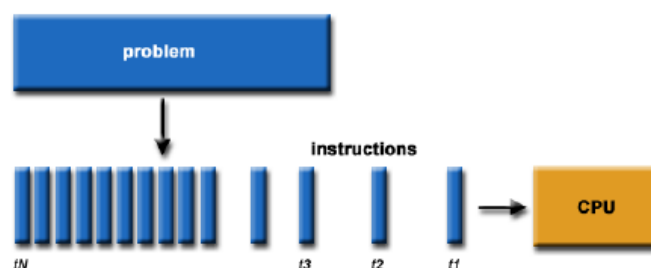


Figure 2.1: Sequential Computing [5]

Parallel processing increases a computer's power. Its main use is solving scientific and engineering problems.

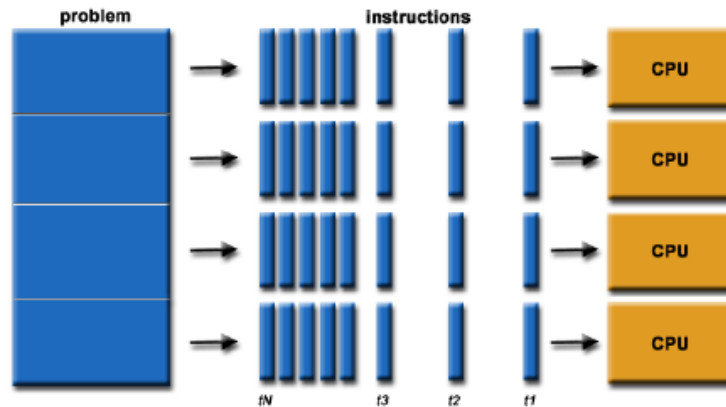


Figure 2.2: Parallel Computing [5]

Commercial software needs to fast computers too. Most programs need to process a large amount of data in a complex form. These programs include:

- ✓ Massive data-base and data mining operation
- ✓ Oil explorations
- ✓ Web searching engine, commercial services under the web
- ✓ Medical imaging and diagnostics
- ✓ Drug design and simulation
- ✓ Management of national and multinational companies.
- ✓ Financial and economic modeling
- ✓ Multimedia technology and video network

The main reasons for using parallel computing are as follows:

1. Economize in rate and time: Using more sources, reduces the time needed for a task. Furthermore, using several cheap sources instead of one expensive source cause reduce costs.
2. Solve larger problems: most large and complex problems that are impractical or impossible to solve with a limited memory computer.
3. Provide concurrency: Multi-computing sources can perform several tasks in the time it takes a single computing source to perform just one task. For example, Access Grid is a global cooperation network in which people all over the world can meet at the same time.
4. Use non-local sources: When local computing sources are limited for solving problems, non-local sources can help to solve such problems through extensive networks and the Internet.

2.2 Parallel Computers Architecture

In 1966, Flynn defined the computer systems architecture classification [2, 6]. Flynn classification design was based on the data stream. Data dealing with processors can be divided into two groups of instructions and data. According to Flynn classification, instructions or data streams can be in one unique form or in multiple forms. As a result, computer systems architecture can be divided into four groups:

1. SISD (Single Instruction Single Data): This architecture is used for sequential computers. In this method, only one instruction stream and one data stream can take action by a processor during each clock [7]. The instructions are independent of other processors-actions. This type of architecture is used in

most computers, including Von Neumann's [8] sequential computers, mainframe systems, and personal computers.

2. SIMD (Single Instruction Multiple Data): This architecture is used for parallel computers. Array processors are one example. SIMD machines have a control unit and execute one instruction, but they have more than one processor element [7, 9]. The control unit signals to all processor's elements which perform similar actions on different data during each clock. This method is suitable for solving special problems that involve data with fixed patterns such as image processing problems.
3. MISD (Multiple Instruction Single Data): In this parallel design, one data stream is sent to several data processing units [7]. Each processing unit acts on the data with independent instruction streams. One example is the Carnegie-Mellon C.mmp experimental computer. This method can be used for several frequencies filters on a signal stream and several cryptography algorithms to decrypt an encrypted message.
4. MIMD (Multiple Instruction Multiple Data): In this architecture, each processor executes separately several instruction streams; instructions apply to several different data streams [2, 7, 9, 10]. Modern super-computers, cluster parallel computers, symmetrical multiprocessors, and modern multi-core computers use this architecture. Most computers with MIMD architecture use SIMD sub-components. One MIMD machine contains processors with control units that can concurrently execute different instructions on different data. This method is the most common design for parallel computers, and modern computers are moving toward this architecture. These kinds of architectures involve several processors and

memory modules that are related by communication networks. They are divided into two main groups: shared memory and distributed memory. Figure 2.3 shows the generic structure of these two groups where P indicates processors and M indicates memory modules.

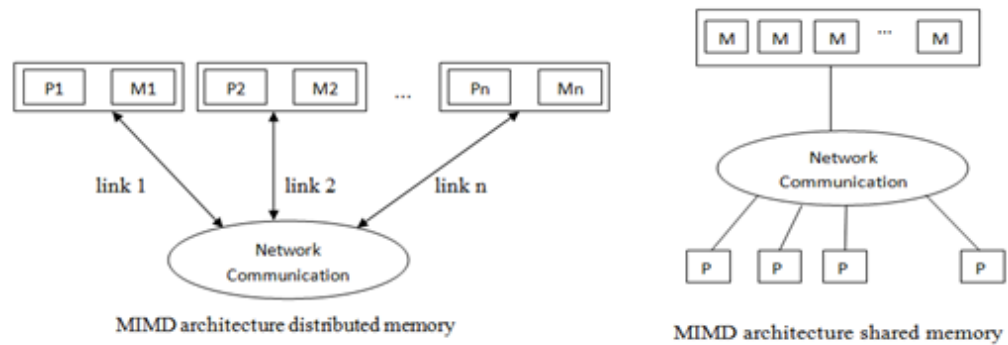


Figure 2.3: Types of MIMD Architecture

2.2.1 Shared Memory Systems

In shared memory systems, all processors have a global shared memory. Communication is established between running operations by reading and writing global memory [2, 11]. Coordination and synchronization of all central processors take place through this shared memory. If all processors have the same availability time to each place of memory, then the shared memory system is called a symmetric multiprocessor system. Design issues for shared memory include access control and data dependence, concurrency, protection, and security.

2.2.2 Distributed Memory Systems

Systems based on distributed memory are groups of processors in which each processor has access to its own local memory. Contrary to shared memory systems, in these systems, connection takes place by sending and receiving message instructions that should be written by the programmer in the application software [2, 12]. A node in such a system contains one processor and its local memory. Each node

usually has the capability of storing a message in the buffer and sending/receiving concurrently with processing. Message processing and calculation is done simultaneously by the operating system. Systems with distributed memory have high extension ability, and their processor units can connect together. Extension capability refers to the ability to increase the number of processors without significant deduction in efficiency.

2.3 Internal Communication Network

Multi-processor system communications networks can be classified according to various criteria, including networks topology. Topology refers to how processors and memories connect to other processors and memories [13]. For example, in complete contact topology, each processor connects to all other available processors in the system. Generally, communication network topology can be divided into static and dynamic groups. In static networks, messages must pass certain links, regardless it is necessary or not. Dynamic networks make connections between two or more nodes if needed for passing messages.

2.4 Parallel Programming Models

Because of their idealism nature, abstract models may not seem appropriate in the real world. However, abstract machines in distributed parallel algorithms are so suitable for parallel machines.

If one algorithm's execution in an abstract system is not satisfactory, then its implementation in a real system is meaningless. Abstract models do not consider some artificial notices in real parallel and distributed systems. This reduces the difficulty of finding executing limitations and complexity estimates. Parallel algorithms designed according to a selection model, and then the model was changed

to run the program [14]. For model implementation in the real world, a set of languages, compilers, libraries, contact systems, and parallel input-output is needed. In following section describe two common parallel models.

2.4.1 Shared Memory Model

In shared memory models, one parallel program is divided into different tasks. Each task execution is assigned to a processor, and all processors act on stored data in the shared memory. For processors, concurrent availability control is used for different concurrent mechanisms like locks and semaphore. For parallel algorithms in this model, execution time, the number of processors and the parallel algorithm rate are considered as criterion.

One model used in shared memory systems is the Parallel Random Access Machine (PRAM). Presented in 1987 by Fortune and Wylie [15] for modeling ideal parallel computers, a PRAM consists of one control unit and one global memory that are shared by a processor. For reduction references to the shared memory by processors, each processor has its own special memory. Figure 2.4 shows a diagram of PRAM. In this model, each processor is not connected to each other, and connections take place only by reading and writing in the shared memory. There are different states for reading and writing [15] operations which divide PRAM into the following classes:

- ✓ EREW (Exclusive Read, Exclusive Write): Reading and writing availabilities in a memory location are exclusive.
- ✓ ERCW (Exclusive Read, Concurrent Write): Some processors have concurrent writing permission in a memory location but reading availability is exclusive.

- ✓ CREW (Concurrent Read, Exclusive Write): Concurrent reading is allowed but writing availabilities are exclusive.
- ✓ CRCW (Concurrent Read, Concurrent Write): Concurrent reading and writing availabilities are allowed.

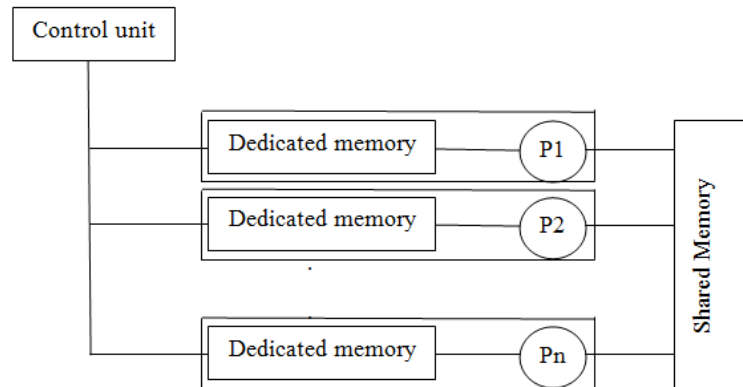


Figure 2.4: PRAM Model for Parallel Computing

2.4.2 Message Passing Model

The message passing model contains a set of processors with their own specific local memory; processors communicate by sending and receiving messages. Data transfer among processors requires mutual operations between processors. This model is widely used in parallel computation due to the many advantages. It offers the following advantages:

- ✓ Compatibility with hardware: This model is appropriate for use in supercomputers and clusters that include separate processors connected through networks.
- ✓ Functionality structure: The message passing model presents essential virtual topology, synchronization, and communication functionality between a set of processes.

- ✓ Efficiency: The effective use of modern processors requires strong management of the memory hierarchy. This model provides location management of data through explicit control tools.

The main disadvantage of this model is that programmer must explicitly recall available functions, distribute data among processors, and manage data.

2.5 Parallel Algorithms

Most algorithms for parallel hardware must be redesigned. Programs that work in a single processor system may not work in a parallel environment. This is because some copies of a program may interfere with each other (for example, interaction in concurrent availability to a location of memory). Therefore, the basic necessity of a parallel system is its own programming. Parallel program design and expansion is often considered a manual process. The programmer is responsible for the determination and actual implementation of parallelism. Manual development of parallel codes is often time-consuming, complex, repetitive, and error-prone. In recent years, most software systems designed for parallel computers programming aim to help the programmer change a sequential program into a parallel program. These systems are at the operation level and at the programming language level. They must have a mechanism to divide a problem into several functions and allocate these functions to processors. This kind of mechanism can include implicit or explicit parallelism.

In implicit parallelism, the system automatically divides the problem into several functions each to a processor; in explicit parallelism, the programmer separates problems into tasks and refers to a processor [16]. Implicit parallelism is limited to a subset of codes and has less flexibility than explicit parallelism. It may also produce

incorrect results and reduce efficiency. Thus most parallel programming is made explicit.

2.5.1 Parallel Algorithm Design

The first step in designing parallel algorithms is learning how to think parallel. The programmer must determine the parts of problem that have parallelism capability; after model selection, he or she must focus on presented the best parallel algorithm.

Several points should be considered when solving a problem in parallel form. First it must be determined whether the problem has parallelism capability [17]. For example, the problem of constructing a Fibonacci sequence is a sequential problem due to its data dependence. Next, the programmer must recognize the basic points of computations and the main areas of the problem. Also, the problem's bottleneck should be recognized; this means that parallel operation is stopped due to attachment or need to perform data input and output.

Next the problem is divided into different sections that can be assigned as a task to a processor. There are two basic methods for dividing computational tasks between processors: domain analysis and functional analysis. In domain analysis, problem data are divided, and each processor executes the same instructions on related data. In functionality analysis, computing instructions are divided among processors. After dividing problems into different functions, if connection between functions is required, concurrent methods and communication among processors are used.

2.6 Performance Evaluation in Parallel Systems

Coefficient speed up or $S(p)$ is one of parallel system evaluation criterion that is defined as follow.

Speed up [18, 19] is ratio of the required time for solving a problem by a processor that showed by $T_{Sequential}$, to required time for solving the same problem by a parallel system that formed by P processors. Parallel system time is shown with $T_{Parallel}$.

$$S(p) = \frac{T_{Sequential}}{T_{Parallel}} \quad (2.1)$$

If: $T_{Parallel} = O\left(\frac{T_{Sequential}}{P}\right)$ (2.2)

If coefficient speed up is equal with P , then the parallel system is optimum. In practice, increasing liner speed (speed proportional with processor number) is difficult. This is due to the sequential nature of many algorithms; thus, some parts of an algorithm are capable of parallelism, while others are not. According to Amdahl's law [20], accelerating the rate of a parallel algorithm rather than a sequential algorithm does not depend on the number of processors used but rather on the part of the algorithm that is not capable of parallelism. If F is the fraction of the algorithm that is incapable of parallelism and should execute in sequential form, then the accelerating rate is defined according to Amdahl's law:

$$S(p) = \frac{1}{F + \frac{1-F}{p}} \quad (2.3)$$

Suppose that 10% of an algorithm is incapable of parallelism. This means that $F=10\%$.

However the rest of the algorithms are run by 20 processors in parallel form. In this state, the execution speed of a program (when run on only one processor) almost be seven times according to Amdahl's law:

$$S(20) = \frac{1}{0.1 + \frac{1-0.1}{20}} \cong 7 \quad (2.4)$$

Another criterion used to evaluate system performance is efficiency, $E(p)$ [21] which is equal to the ratio of cost of an algorithm in sequential system to cost of the same algorithm in a parallel system that is formed by p processors. The cost of implementation is equal to the multiplied execution time in the processor's number:

$$E(p) = \frac{1 * T_{sequential}}{p * T_{parallel}} \quad (2.5)$$

$$= \frac{S(p)}{p}$$

Chapter 3

MATRIX MULTIPLICATION ALGORITHMS AND RELATED WORKS

The evaluation of the product of two matrices can be very computationally expensive. The multiplication of two $n \times n$ matrices, using the standard algorithm can take $O(n^3)$ operations. Consider matrix multiplication with standard algorithm as follows:

```
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++){
        C[i][j]=0;
        for (k=1;k<=n;k++)
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
    }
```

This program multiplies two matrices A and B to obtain matrix C . In each matrices, n (dimension of matrices) is greater than 0.

In the standard algorithm, the number of multiplication equals to $T(n) = O(n^3)$.

The number of additions also equals with $T(n) = O(n^3 - n^2)$ which is explained below.

```
for (i=1;i<=n; i++)
```

```

for(j=1;j<=n;j++){
    C[i][j]=A[i][1]*B[1][j];
    for (k=2; k<=n;k++)
        C[i][j]=C[i][j]+A[i][k]*B[k][j];
    }

```

In the standard state, number of multiplications and additions of a matrix multiplication is in the following form:

Number of multiplications: $n^3 = O(n^3)$

Number of additions: $n^3 - n^2 = O(n^3)$

3.1 Reviews of Matrices Multiplication Using Divide-and-Conquer Method

Now we consider matrix multiplication in the divide-and-conquer method. If n is a power of 2, A and B can be divided into four smaller matrices of $n/2 \times n/2$ each [22].

If the number of multiplies are considered as a main act, each $n \times n$ matrix required eight multiply action in any stage of division to $n/2 \times n/2$:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C_{11}=A_{11}.B_{11}+A_{12}.B_{21}$$

$$C_{21}=A_{21}.B_{11}+A_{22}.B_{21}$$

$$C_{12}=A_{11}.B_{12}+A_{12}.B_{22}$$

$$C_{22}=A_{21}.B_{12}+A_{22}.B_{22}$$

Multiplication of two 1×1 matrices need a scalar multiply action. So, in the divided-and-conquer algorithm for the matrix multiplications, we have:

$$\left. \begin{array}{l} T(n) = 8T(n/2) \\ T(1) = 1 \end{array} \right\} \implies \Theta(n^3) \quad (3.1)$$

This method is similar to the standard method of $\Theta(n^3)$ and has no extra preference.

3.2 Considering Matrix Multiplication by Use of Strassen Method

In 1969, Strassen presented an algorithm that multiplies numbers less than $O(n^3)$; it almost was $O(n^{2.81})$ mentioned in down [22, 23]. Strassen proved that multiplying two matrices A and B , leads to C can be obtained by following relation:

If the matrices A and B have 2×2 dimensions, the necessary number of additions and multiplications for matrix computation is as follows:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$

$$m_2 = (a_{21} + a_{22})b_{11}$$

$$m_3 = a_{11}(b_{12} - b_{22})$$

$$m_4 = a_{22}(b_{21} - b_{11})$$

$$m_5 = (a_{11} + a_{12})b_{22}$$

$$\begin{cases} C_{11} = m_1 + m_4 - m_5 + m_7 \\ C_{12} = m_3 + m_5 \\ C_{21} = m_2 + m_4 \\ C_{22} = m_1 + m_3 - m_2 + m_6 \end{cases}$$

$$m_6 = (a_{21}-a_{11})(b_{11}+b_{12})$$


$$m_7 = (a_{12}-a_{22})(b_{21}+b_{22})$$

Table 3.1 provides the number of multiplications and additions needed for two standard and Strassen algorithms for two matrices of 2×2 .

Table 3.1: Comparison of Standard and Strassen Matrix Multiplication Algorithms

Multiplication type	Multiplication number	Addition number
Standard algorithm	8	4
Strassen algorithm	7	18

For larger matrices, supposing that n (dimensions' of matrices) is a power of 2, Strassen's method can be extended as below:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$


$$A_{11} = \begin{bmatrix} a_{11} & \cdots & a_{1^{n/2}} \\ \vdots & \ddots & \vdots \\ a_{n/2 1} & \cdots & a_{n/2^{n/2}} \end{bmatrix}$$

Using Strassen method, M_i is calculated as:

$$M_1 = (A_{11}+A_{22})(B_{11}+B_{22})$$

$$M_2 = (A_{21}+A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12}-B_{22})$$

$$M_4 = A_{22}(B_{21}-B_{11})$$

$$M_5 = (A_{11}+A_{12})B_{22}$$

$$M_6 = (A_{21}-A_{11})(B_{11}+B_{12})$$

$$M_7 = (A_{12}-A_{22})(B_{21}+B_{22})$$

and $C_{i,j}$ is calculated as:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 + M_3 - M_2 + M_6$$

In the M 's calculation for doing multiplication, again Strassen method will be used.

Strassen algorithm is explained by an example in the following. In this example A and B are input matrices and C is the result of multiplication.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

$$B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} * \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$\begin{matrix} \xleftrightarrow{2} \\ \updownarrow 2 \end{matrix}$

$$M_1 = (A_{11}+A_{22})(B_{11}+B_{22}) \Rightarrow \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) * \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} * \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}$$

$$M_2 = (A_{21}+A_{22})B_{11} \Rightarrow \left(\begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) * \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 11 & 4 \\ 10 & 12 \end{bmatrix} * \begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 100 & 115 \\ 116 & 138 \end{bmatrix}$$

$$M_3 = A_{11}(B_{12}-B_{22}) \Rightarrow \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \right) * \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} - \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} * \begin{bmatrix} -8 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} -6 & 3 \\ -34 & 11 \end{bmatrix}$$

$$M_4 = A_{22}(B_{21}-B_{11}) \Rightarrow \left(\begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) * \left(\begin{bmatrix} 7 & 8 \\ 2 & 3 \end{bmatrix} - \begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} * \begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix} = \begin{bmatrix} -5 & -5 \\ -13 & -13 \end{bmatrix}$$

$$M_5 = (A_{11}+A_{12})B_{22} \Rightarrow \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \right) * \left(\begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 4 & 6 \\ 12 & 14 \end{bmatrix} * \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 60 & 34 \\ 164 & 82 \end{bmatrix}$$

$$M_6 = (A_{21}-A_{11})(B_{11}+B_{12}) \Rightarrow \left(\begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \right) * \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 8 & -1 \\ -1 & -1 \end{bmatrix} * \begin{bmatrix} 9 & 11 \\ 8 & 10 \end{bmatrix} = \begin{bmatrix} 64 & 78 \\ -17 & -21 \end{bmatrix}$$

$$M_7 = (A_{12}-A_{22})(B_{21}+B_{22}) \Rightarrow \left(\begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} - \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) * \left(\begin{bmatrix} 7 & 8 \\ 2 & 3 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 16 & 9 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 22 & 17 \\ 22 & 17 \end{bmatrix}$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$= \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix} + \begin{bmatrix} -5 & -5 \\ -13 & -13 \end{bmatrix} - \begin{bmatrix} 60 & 34 \\ 164 & 82 \end{bmatrix} + \begin{bmatrix} 22 & 17 \\ 22 & 17 \end{bmatrix} = \begin{bmatrix} 43 & 32 \\ 123 & 149 \end{bmatrix}$$

$$C_{12} = M_3 + M_5$$

$$= \begin{bmatrix} -6 & 3 \\ -34 & 11 \end{bmatrix} + \begin{bmatrix} 60 & 34 \\ 164 & 82 \end{bmatrix} = \begin{bmatrix} 54 & 37 \\ 130 & 93 \end{bmatrix}$$

$$C_{21} = M_2 + M_4$$

$$= \begin{bmatrix} 100 & 115 \\ 116 & 138 \end{bmatrix} + \begin{bmatrix} -5 & -5 \\ -13 & -13 \end{bmatrix} = \begin{bmatrix} 95 & 110 \\ 103 & 125 \end{bmatrix}$$

$$C_{22} = M_1 + M_3 - M_2 + M_6$$

$$= \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix} + \begin{bmatrix} -6 & 3 \\ -34 & 11 \end{bmatrix} - \begin{bmatrix} 100 & 115 \\ 116 & 138 \end{bmatrix} + \begin{bmatrix} 64 & 78 \\ -17 & -21 \end{bmatrix} = \begin{bmatrix} 44 & 41 \\ 111 & 101 \end{bmatrix}$$

$$\Rightarrow C = \begin{bmatrix} 43 & 32 & 54 & 37 \\ 123 & 149 & 130 & 93 \\ 95 & 110 & 44 & 41 \\ 103 & 125 & 111 & 101 \end{bmatrix}$$

3.3 Related Works

Strassen matrix multiplication algorithm has been implemented in parallel on some different methods and we are going to briefly survey them in this section. The method proposed in [24] discussed sequential and three parallel programs that have been attempted to implement Strassen's algorithm. The sequential program was written by using the well-known Winograd's method [25]. It stops its recursion on a certain level where it invokes the subroutine DGEMM provided by ATLAS [26]. Since the design of the program is straightforward, its performance and instability issues were introduced, as well as how they vary with the recursion level.

The three parallel programs include one workflow program and two MPI programs. The workflow program is implemented in the client-end on the NetSolve system [27]. It has a workflow controller to check and start the tasks in a task graph. All tasks are sent to the NetSolve servers to be computed. When the dependent tasks are finished, the controller launches a new task immediately. The intensive computation is actually performed on the NetSolve servers, thus the client machine is available to run other tasks. Next, two different approaches are adapted for designing the parallel programs running on distributed memory systems. The first program uses a task-parallel approach, and the second one uses a data-parallel approach which uses the ScaLAPACK library to compute the sub matrix multiplications [28].

The approach proposed in [29] uses Strassen algorithm across all processors, instead of using it only on each processor. This approach leads to have a better potential for speed up.

A parallel algorithm that uses Strassen's matrix multiplication both between the processors for global computations and within each processor for local computations was proposed in [30]. With respect to [30], two-fold is the main conclusions of the performance study; firstly, controlling the communication path via ad hoc routing patterns can provide significant performance gains especially for large networks and even larger matrices. This result is especially crucial for applications that require petaflop or exaflop processing rates. Secondly, the proposed algorithm is quite successful in overlapping the communication with computation. It is well-known that Strassen's algorithm ceases to provide any benefits when local matrix sizes become too small. In other words, beyond some point it is better to stop the recursion and to switch to the conventional algorithm to perform sub-matrix multiplications. In the proposed algorithm, the need to switch occurs much deeper in the recursion tree. As an example of the effectiveness of the proposed scheme, also consider the case in which we have a 64×64 torus at our disposal. The proposed algorithm can only use 49×49 processors and after the fourth recursion each processor performs exactly one computation. In this case, the proposed algorithm still up to 1.3 times faster than the other algorithms.

The research in [31] tried to work on parallel Strassen matrix multiplication algorithm on heterogeneous groups. Suitable data allocation in the heterogeneous grouping context is the most necessary item to obtain optimal execution time. Strassen algorithm decreases the number of multiplication operations from eight to seven in any recursion, therefore the level of the recursion has outcome on the sum up execution count. In Strassen algorithm, not only the charge parity, but also the

extent of recursion should be considered as well. The above mentioned program gains both charge parity and decreasing the whole multiplication operations count.

Due to enlargement of groups, more recent nodes are persistently attached to existing group systems. The nodes may have contrastive hardware execution, like network rapidity and CPU execution which construct the group heterogeneous. The similar charge can be allocated to every processor if the hardware performance of each node is homogeneous. Therefore, charge parity is automatically reached and greater swift is also obtained at ease. Although, in heterogeneous contexts, traditional procedures that allocate same duties to each processor turn down to less optimal due to they would not be able to account differences among nodes in computational performance. For that reason and in order to reach the better speed, data should be allocated properly and equivalently to the hardware operation of every node in the group.

It is very critical to reduce the inactive time of processors by considering the effect of charge parity in a heterogeneous clustering context. However, the level of recursion in Strassen algorithm influences on the total multiplication operation count, and there is a possibility that total multiplication operation count is increased by charge parity. So, both charge parity and the level of recursion should be taken into account in Strassen algorithm. In this case, the recursive data decomposition is suggested and it enables charge parity and increasing of the level of the recursion in Strassen algorithm.

A scalable parallel Strassen's matrix multiplication algorithm for distributed memory named by Strassen-BMR was presented in [32]. The motivation for this method

comes from the observation that the Strassen method is most efficient for large matrices. Therefore it should be used among processors instead of one processor. The seven sub-matrix multiplications of the Strassen method at each recursion seem at first to lead to a task parallelism. The difficulty in implementation results from the fact that the matrices must be distributed among the processors. Sub-matrices must be stored in different processors and if tasks are spawned these sub-matrices must be copied or moved to the appropriate processors. For a distributed memory parallel algorithm, the storage map of sub-matrices to processors is a primary concern. If the sub-matrices are stored among processors in the same pattern at each level of recursion, then they can be added or multiplied together just as if they are stored within one processor.

We compare our obtained results with this method in Chapter 5. The implementation results of our method show some improvement rather than this method. Comparing has been done only over four processors. Bear in mind that, Fair method is not applied on 64 processors so the related results are not available in this study.

Chapter 4

STRASSEN PARALLEL MATRIX MULTIPLICATION ALGORITHM IN DISTRIBUTED SYSTEM

In the previous chapter, we surveyed some well-known algorithms of parallel Strassen matrix multiplication. The current research employs the Strassen matrix multiplication algorithm as a recursive algorithm and it decreases the execution time by using distribution factor.

The focus of the thesis is on the method of data distribution for multiplication in order to expand the overlapping operation. First, the proposed method and then its extensions will be discussed in detail in the following.

Considering recursive nature of the algorithm, the client sends its task to the server(s) and then it waits for the calculated response. If the data received by the servers are not small enough or are needed to the division of problems among other servers, those servers which have received data from the client will change their status and appear in the role of client. The upper-layer client (parent) must distribute tasks among servers and wait for the results from lower-layer servers, this process will continue through the lower-layer servers until the problem is minimized (no further division is needed) or there is no idle server. Then, the results will be sent back to upper nodes sequentially. The details of propose method and existing difficulty will be explained in the following sections.

4.1 Two-fold Distribution Method

In the first stage, the parts of algorithm that can be parallelized are identified by using the Strassen algorithm and the main calculation of this algorithm should be considered. The main operations are calculation of seven multiplication tasks which should be computed in each stage of problem division. In this method of division and distribution, four multiplication tasks are assigned to one server computer, and three tasks are dedicated to another server. It means that in each stage of division, every client will divide and distribute multiplication tasks (including seven sub-multiplication tasks) between two servers. Considering this method of distribution, each client computer (parent node) for each multiplication task has two server computers as a child, so the distribution topology will resemble a two-fold tree (see Figure 4.1).

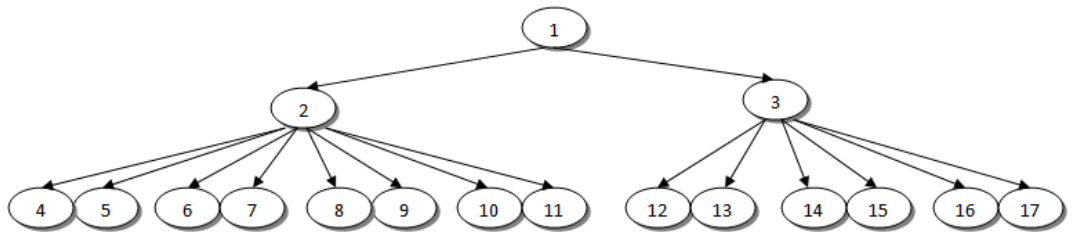


Figure 4.1: Structure of Two-fold Distribution Method

Note that each node represents a computer in Figure 4.1. Node (1) as a client, divides its seven multiplication task between two servers (node (2) and (3)) and it sends four multiplication tasks to node (2) and three remaining multiplication task will be sent to node (3). Now, node (2) has four multiplication tasks and each task divided and distributed between two computers. In other words, node (2) divides and distributes the four multiplication tasks among eight computers. Node (3) receives three multiplication tasks and each of them is divided and distributed between two

nodes. Therefore, its task is distributed among six children. The same procedure is continuously applied in the succeeding layers.

4.1.1 Reusing Waiting Node

After implementing the two-fold distribution method, we faced some difficulties related to having a waiting period for clients (parents) when their child nodes calculate the results.

Servers in each layer which receive tasks, they will change their status from server to client according to existing circumstances. They also divide and distribute the data among free servers and wait for the results. During this period, the efficiency of the computers decreases, because task is dedicated to only some nodes. When this approach deals with large and huge matrices, dividing the problem should be done more times and numbers of sub-problems are increased. Thereby, number of clients and their waiting time will be increased.

In order to increase the processor efficiency, the time spent in waiting status must be minimized. If some of the free servers have finished, then clients which are in waiting status can function as free servers.

By using multi-thread method, all servers are first in idle and listening status. When they receive a task from a client, they change their status to busy and then they process the task. If the division and distribution stage continues, then it changes its status to client and waits for a response from child servers. Simultaneously, waiting clients will be in the listening status (like a server), so that if a task is received for being calculated, it can perform it during the waiting period. In this way, the CPU capacity of the nodes is used efficiently. Figure 4.2 provides an example. Suppose

that there are nine computers (nodes). Computer (1) is in the role of client and distributes the tasks among servers.

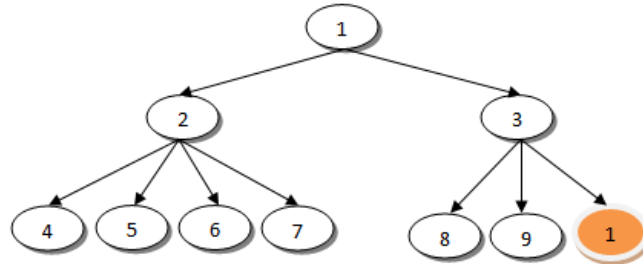


Figure 4.2: Structure of Reusing of the Waiting Node in Distribution

In Figure 4.2, division and distribution of the problem stops in the third layer, because free servers have finished, so there is no possibility to fill all of the leaves. In order to increase the efficiency, upper-layer computers that are waiting for the results, switch to listening mode (server) and execute task after receiving it. In this example, nodes 1, 2 and 3 are in listening (for any task) and waiting state (for the result). Here, node (1) is used again as a server by node (3).

4.1.2 Performance

By implementing the algorithm based on the above-mentioned distribution topology, we improved the execution time by increasing the number of computers which performs tasks. For smaller dimensions of matrices three computers are sufficient for their multiplications and achieved good execution time. But larger matrices which required more computers for multiplication, did not improve the execution time (i.e. the improvement is not proportional with the increase in the number of computers).

For the tree topology in Figure 4.1, completion of the last stage of each tree distribution indicates the percentage of parallelism for that distribution status. For example, with seventeen computers, we achieved better performance time, because

the third layer is completed and fourteen computers are able to do calculations in parallel. With twenty-five computers, eight of them are in the fourth layer thus, the percentage of parallel calculations in the last layer decreases and less improvement is observed. As a result, this distribution topology in a network comprising computers in the interval of 2 to 6 and 15 to 17 has better proportional performance compared to the rest of the computers in the network. To further improve performance, we define other topologies in the following sections.

4.2 Seven-fold Distribution Method

As mentioned in the previous distribution method, computers in the last layer are filled less than 50%; due to this, less parallelism took place. To resolve this problem, we choose a method that in a network including say 8 computers, the number of computers existing in the last layer, could have more computers relatively until we could increase the percentage of parallelism. For this reason, the client in each level of the operating division; divides seven multiplication tasks among seven computers that each multiplication task is dedicated and sent to a computer (see Figure 4.3).

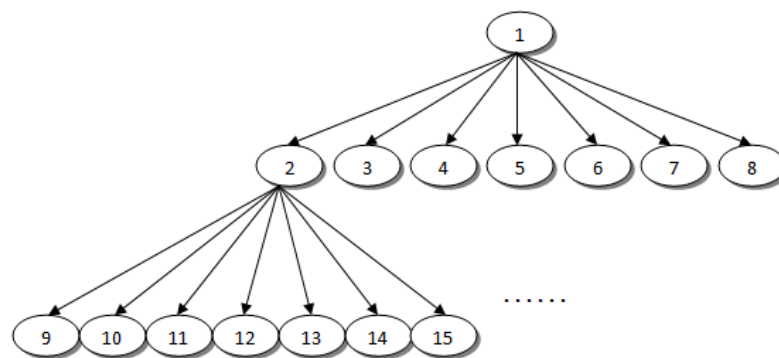


Figure 4.3: Structure of Seven-fold Distribution Method

In the Figure 4.3, client 1 divides seven multiplication tasks among seven servers. According to the algorithm, the servers (computers 2 through 8) receive the matrices

and survey the threshold condition that identifies the limitation of matrix division and distribution. If division and distribution must continue, then the servers will divide and distribute tasks in the network among free and listening computers. In this layer (layer 2), each server has received a multiplication task from the client, each of which has seven sub-tasks. The servers change its role to client and distribute tasks to the sub-layer servers. Similarly, the algorithm continues recursively.

This division and distribution method improved the performance in networks including 7 up to 12 computers. For example, if we had eight computers (two layers) in Figure 4.3, there was maximum parallelism, since in the second layer; seven of the eight computers are performing computation in parallel.

In different circumstances (dimension of matrices, threshold of algorithm, and number of existing computers in the network) increasing the number of computers up to eight could improve the performance, but increasing the number of computers more than 8 we did not experience significant improvement in the performance. With more than eight computers, the third layer is considered the main factor in parallelism. As long as the majority of the leaf nodes in this layer do not fill up, we will not see significant improvement.

To overcome the weakness of the two-fold method, a seven-fold distribution method was introduced. Now if we present a new distribution method in order to improve the weaknesses of the seven-fold method, definitely we experience other weaknesses. However, there is no constant distribution method that will yield the best result in all networks. In the following Section we have presented a method to find optimum distribution topology.

4.3 Dynamic Distribution Method

As previously explained, any of the constant distribution method cannot always respond positively. To achieve an optimum response, we need a special distribution topology for different circumstances. However, it is very difficult to define all optimum topologies for all different circumstances. Therefore, in this section, the program defines optimum distribution topology itself. According to circumstances which are distinguished from user entries, the optimum distribution topology is found, and the division and distribution operation of matrices is performed.

Before explaining how optimum topology is found; first, we clarify the possible levels of task division among computers from the client. The main operation is the seven multiplication tasks of Strassen algorithm. Different methods can be used to divide the seven tasks in a way that maintains the potential for parallelism. We define the following four division methods in the program:

1. The client divides seven multiplication tasks between two computers: four tasks to the first computer and three tasks to the second computer.
2. The client divides seven multiplication tasks among three computers: three tasks to the first and two tasks each to second and third.
3. The client divides seven multiplication tasks among four computers: two tasks each to the first three computers and one task to the fourth.
4. The client divides seven multiplication tasks equally among seven computers.

To find the optimum distribution topology, we need a criterion for optimization. Based on the previous constant distribution methods, if there are more computers in the last (leaf) layer and the layer is complete, then the scope of parallelism increases and thus improves performance. Therefore, the criterion for finding the ideal

topology is the choice of a level that allows for the most number of computers in the leaf layer of the distribution tree according to existing entry circumstances (dimension of matrix, threshold of division, number of existing computers in the network). For this reason, we calculate the number of layers that the distribution tree should have.

In fact, the number of layers in the distribution tree is the number of divisions before the threshold is reached, which means one operation in each layer. The number of layers in the topology tree or the number of divisions equals " $\log_2 matrix\ size - \log_2\ threshold$ ", which are entries of the program.

Next, we design a distribution tree with the desired number of layers and consider the numbers of computers in the network, and also we should have the most possible numbers of computers in the last layer. There will be circumstances when the number of existing computers in network is not enough to build a tree with the number of desired layers. In this case, we choose a distribution tree with the most possible layers and the most number of computers in the last layer. Conversely, there may be too many existing computers in a network for the stated program entries, in which case we also use the required number of computers.

For instance, suppose we have 10 existing computers in a network; the dimension of entry matrix is 1024×1024 , and the threshold for dividing of the matrix is 128. This means that until the dimension of matrices reaches 128, division and distribution continues. Using this entry information, the number of layers for the distribution tree is calculated as follows:

$$(\log_2 1024 - \log_2 128) = 10 - 7 = 3 \quad (4.1)$$

The program finds a tree among those that can be built with 10 computers and three layers, with most computers in the last layer.

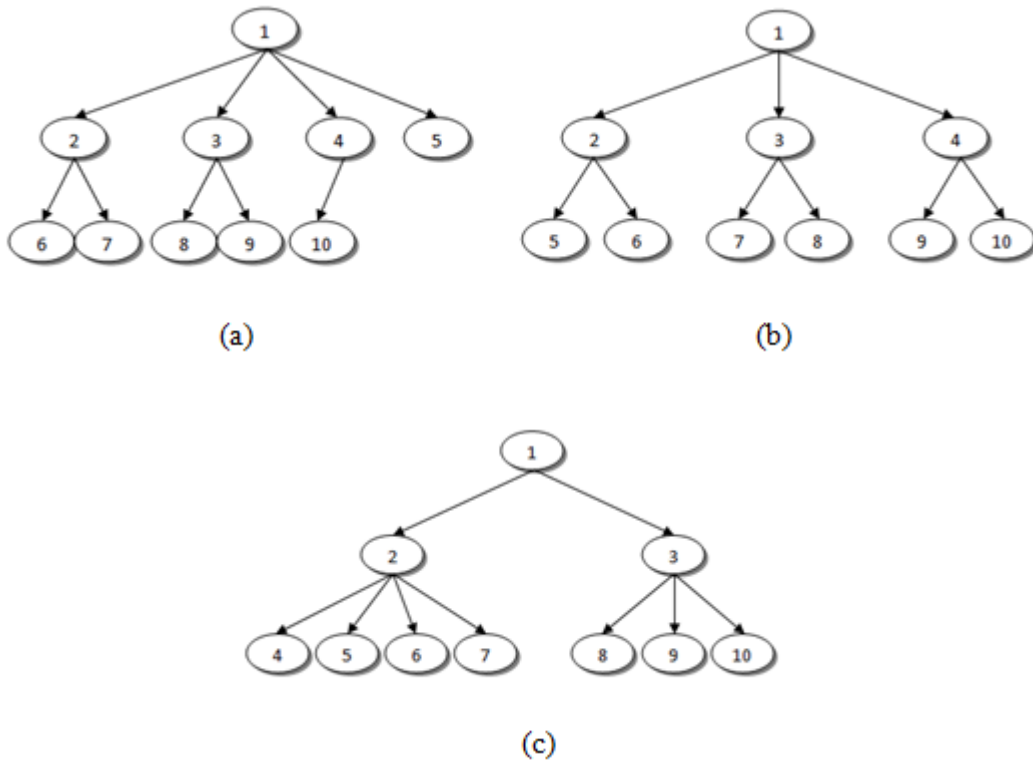


Figure 4.4: Some Samples of Dynamic Distribution Method

As illustrated in Figure 4.4, the three distribution trees can all be defined with 10 computers and three layers but with different numbers of computers in the last layer. In Figure 4.4(a), there are five computers in last layer; in Figure 4.4(b), there are six; and in Figure 4.4(c), there are seven. When the problem is divided equally among seven computers and is processed simultaneously, fewer computers are in the waiting position compared to other trees. This means that the tree in Figure4.4(c) is most efficient.

As previously explained the Dynamic Distribution Method first finds the optimum distribution topology and then attempts to divide and distribute tasks among servers accordingly. As soon as servers receive tasks according to the distribution tree, which is received along with task itself, they will attempt to perform the task. During the execution, we have reused the clients only when a small number of computers in last layer is needed to make the layer completed.

For all propose methods, programs which are executed on computers are comprised of client and server. We have created separate program (modified) for each method in C# to run on server and client nodes. In this program, being either client or server is specified via configuration file. To complete the mentioned explanations about the program, we have provided it flowchart for server and client computers. Figure 4.5 presents flowchart of the program on the client computer located at the root of tree topology. Figure 4.6 presents flowchart of the program which is performed on all server computers.

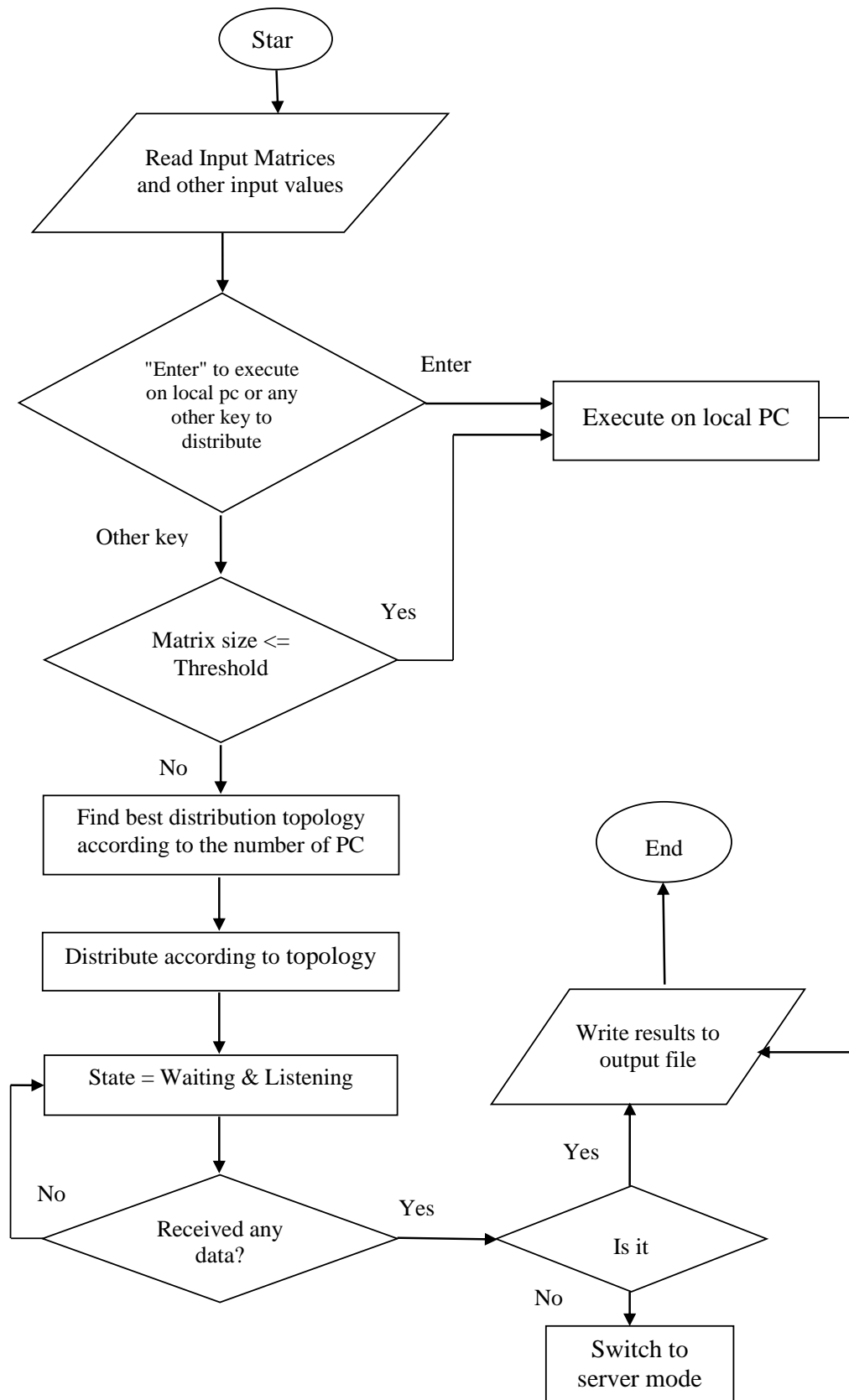


Figure 4.5: Flowchart of the Client Program in Dynamic Distribution

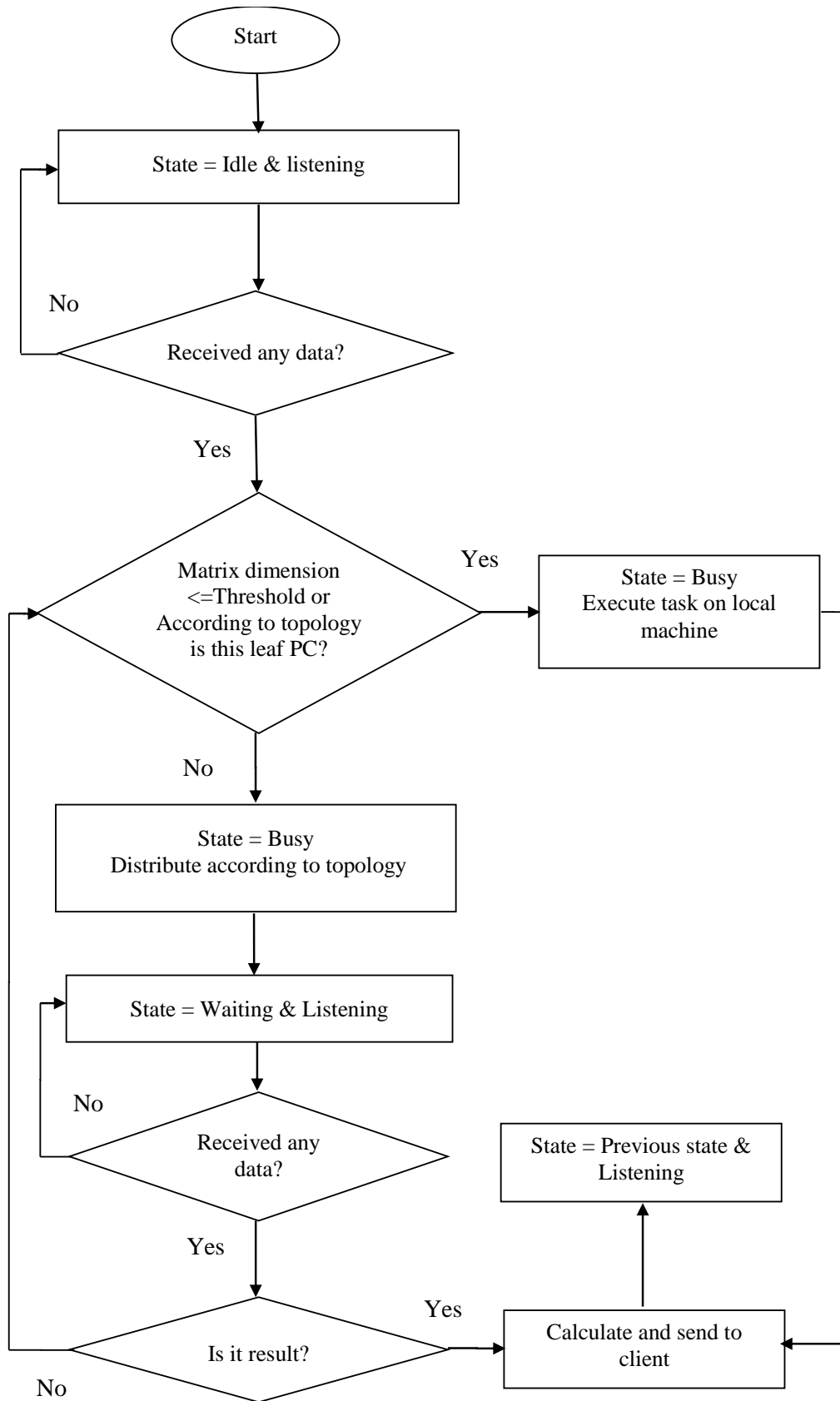


Figure 4.6: Flowchart of Server Program in Dynamic Distribution Method

4.3.1 Performance Evaluation of Dynamic Distribution Method

Section 2.6 described to what extent the function of algorithm in a parallel system is improved compared to a sequential system. Two criteria's were presented for surveying the degree of improvement. Now, using this criteria's, we consider the efficiency of the algorithm in parallel status comparing to sequential status. We calculated the speed-up $S(p)$ according to related formula in Section 2.6, for $p=10$ and $p=20$, which present the number of computers (processors) in parallel system:

$$S(p) = \frac{T_{sequential}}{T_{parallel}} \implies \begin{cases} S(10) = \frac{30.6}{7.7} = 3.94 \\ S(20) = \frac{30.6}{6.52} = 4.65 \end{cases} \quad (4.2)$$

Obtained results show the speed-up values in case of having 10 and 20 nodes over parallel model rather than using sequential model respectively.

Using the efficiency related formula in Section 2.6, we have calculate efficiency $E(p)$ for the parallel algorithm with 10 and 20 computers ($p=10, 20$).

$$E(p) = \frac{1 * T_{sequential}}{p * T_{parallel}} = \frac{S(p)}{p} \implies \begin{cases} E(10) = \frac{S(10)}{10} = \frac{3.94}{10} = 0.39 \\ E(20) = \frac{S(20)}{20} = \frac{4.65}{20} = 0.23 \end{cases} \quad (4.3)$$

These calculated results show the efficiency of the algorithm in case of having 10 and 20 nodes over parallel model instead of applying sequential model. It is obviously seen that using 10 nodes instead of 20 nodes leads to more efficiency value and better performance.

4.4 Fair Distribution Method

Dynamic Distribution method was implemented for the improvement of the fixed distribution methods. But, these distribution methods were unfair in task division. For the mentioned distribution methods, in the beginning of the program no task was indicated to the client itself and in the continuation of task distribution procedure, the task was indicated for the client if free servers were finished in the network.

In the new distribution method, this problem has been revised. In this method, according to the number of existing server in the network, at least one of the seven multiplication tasks is considered for the client itself and the rest are distributed among servers. The procedure of task division among computers takes place in the following manner. For one client and one server case, three multiplication tasks for the client and the other four are allocated to the server. In one client and two server status, two multiplication tasks belong to the client and two to three multiplication tasks are allocated to servers respectively. In the status of more than three computers, always one task is considered for the client and the rest of them are distributed among servers. For four to seven computers in network, the numbers of allocated tasks for computers are as follows:

4 PCs: 1 task for client and 2 tasks for any other 3 servers.

5 PCs: 1 task for client and 1, 1, 2, 2 for servers respectively.

6 PCs: 1 task for client and 1, 1, 1, 1, 2 for servers respectively.

7 PCs: 1 task for client and 1, 1, 1, 1, 1, 1 for servers respectively.

Now when the numbers of existing computers are more than seven, with respect to the any PC more than seven, the same number of computers from one to seven would

share and send their tasks to them. For instance, for the status of eight PCs (one PC is more than seven), only computer number one (client) will share its task with that and it will send for PC number 8. For the status of nine PCs (two PCs are more than seven), computers number 1 and 2 (client and a server), will share their tasks with eight and nine PCs. Here the ratio of task division is the same with status of less than seven PCs. It means that in task division between client and a server, three tasks for the client and four others belong to server.

Note that, where the numbers of existing PCs are fourteen, all PCs share their tasks with another server. If the numbers of PCs in the network are more than fourteen for each PC more than fourteen, PCs one to seven would divide their tasks with another two servers instead of one. For example, in the status of eighteen PCs (four PCs are more than fourteen), PCs one to four will share their tasks to another two servers instead of one and PCs five to seven will also share their tasks to one server. Figure 4.7 shows more details in the continuation of this example. This procedure of algorithm carries out the mentioned approach on the expansion of computers in network.

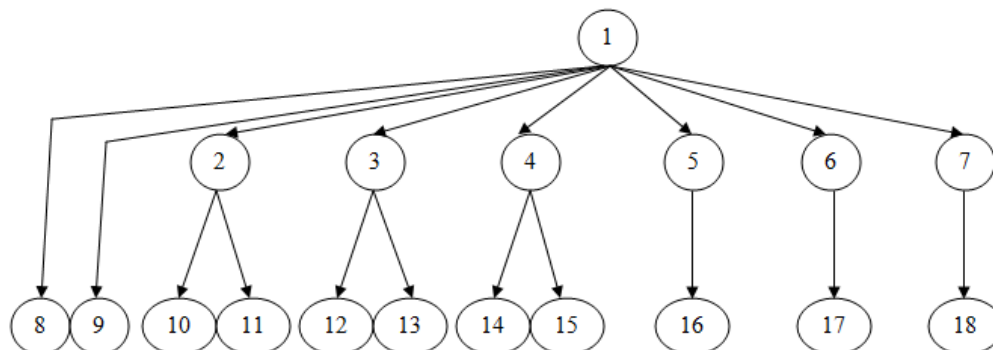


Figure 4.7: An Example of Fair Distribution Method

In this example, there are eighteen PCs (4 PCs more than 14) which PCs one to four distribute its tasks to another two servers while PCs five to seven distribute its tasks to only one server.

Flowchart of the Fair distribution method is proposed in two figures (for client program and server program) in the continuation. Figure 4.8 presents flowchart of the program on the client computer located at the root of tree topology. Figure 4.9 presents flowchart of the program which is performed on all server computers.

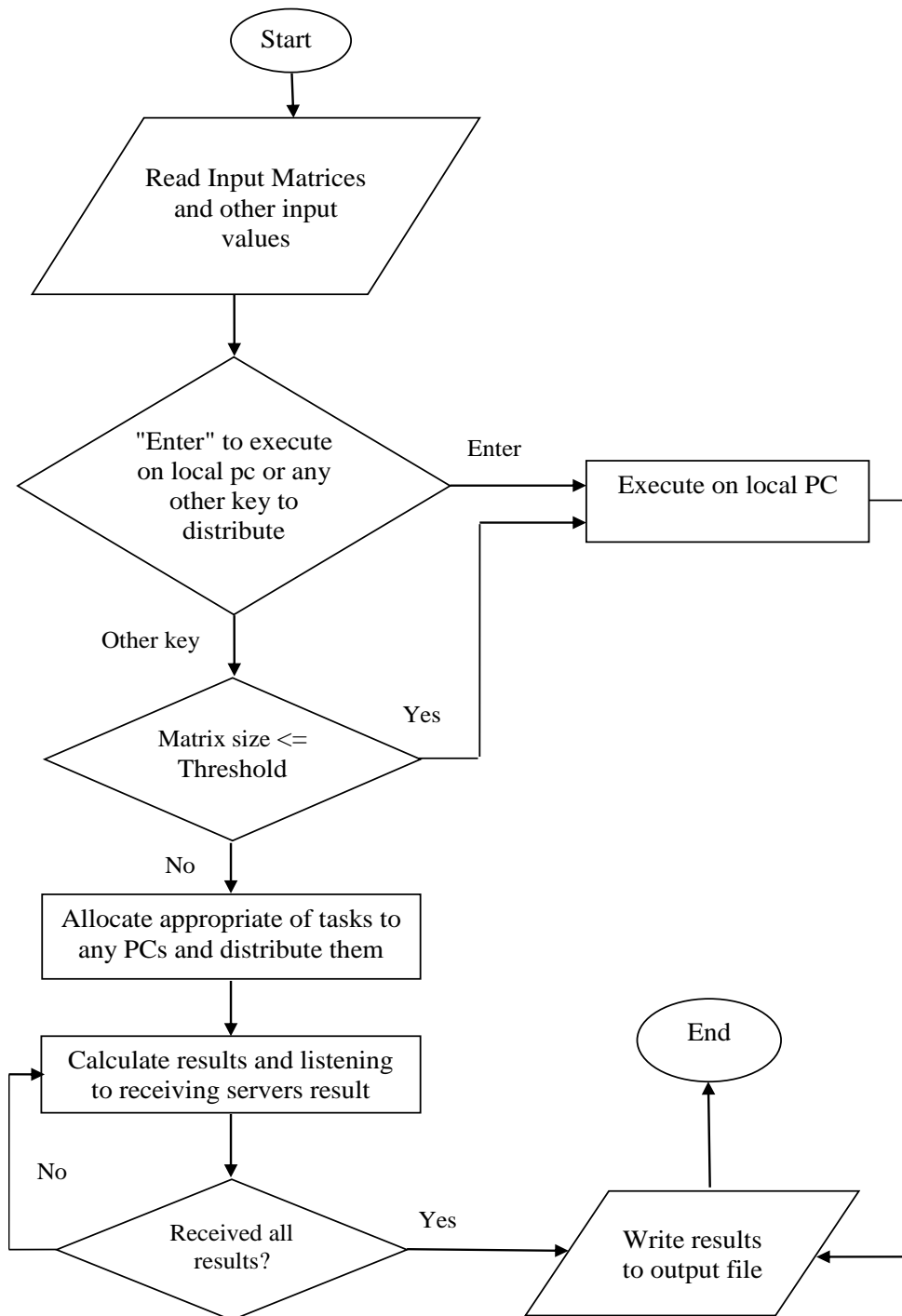


Figure 4.8: Flowchart of Client Program in Fair Distribution Method

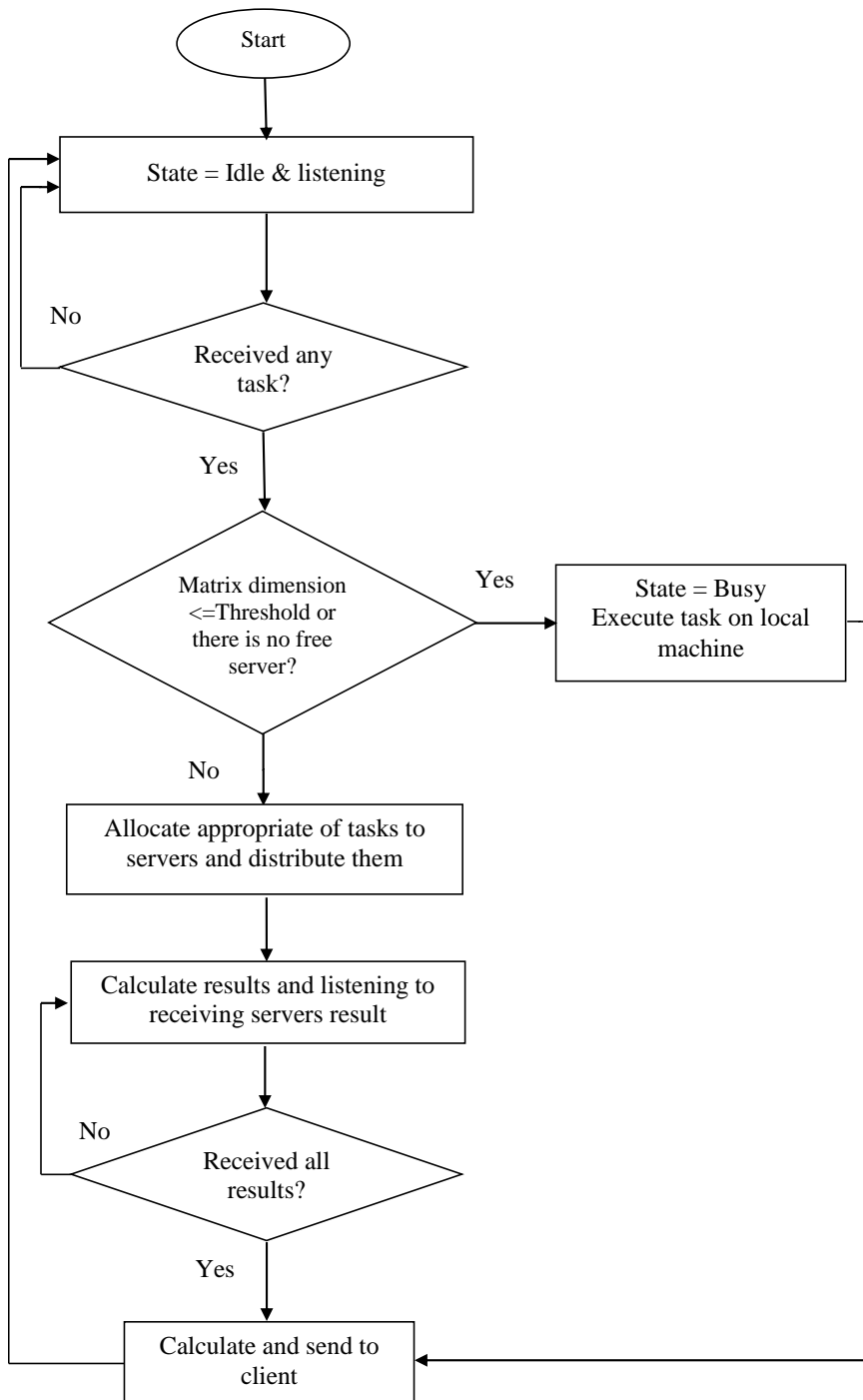


Figure 4.10: Flowchart of Server Program in Fair Distribution Method

Chapter 5

EXPERIMENTAL RESULTS

This chapter presents the experimental results of proposed methods. Results are presented by using different matrix dimensions, thresholds and number of computers.

The network properties and parameter values which we have used in our test system are set as following. The network includes 20 nodes which have been connected through Ethernet switch with 100 Mb/s data rate. The network has employed with 32-bit computers includes Windows 7 Professional-operating system, Intel Core 2 Duo CPU, 4 GB RAM, and 150 GB hard disk. The model of network adapters is Realtek RTL8168D/8111D family PCI-E Gigabit Ethernet NIC.

The related program has been written in C# environment by applying socket programming techniques. The input of our program is two squared matrices of integer numbers. The integer numbers applied in the input matrices have been generated randomly in range $[0, 100]$. Matrix dimensions have been varied in between 128 and 2048 in the form of 2^n .

5.1 Comparison of Usual and Reuse of Waiting Clients Methods

In Section 4.1, we explained the Two-fold distribution method. It is known as a usual distribution topology. In section 4.1.1, we introduced the proposed reusing method.

Table 5.1 compares experimental results of these two methods with five different numbers of computers (PCs). It should be mentioned that, all matrix dimensions are

2048×2048 and the threshold value is 128 for each number of PCs (4, 8, 12, 16 and 20).

Table 5.1: Execution Time for Usual and Reuse of Waiting Clients by Two-fold Distribution Method

PC Numbers	Execution time, minutes	
	Usual	Reuse of waiting clients
4	14.1	16.15
8	11.08	12.01
12	9.33	8.32
16	9.26	8.1
20	8.57	7.32

The table results are also presented in a form of graph in Figure 5.1.

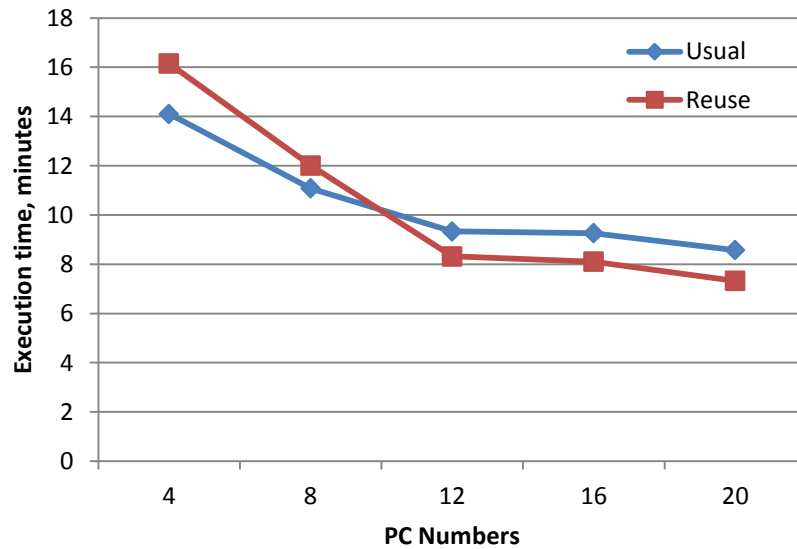


Figure 5.1: Execution Time versus Number of Computers for Usual and Reuse of Waiting Clients by Two-fold Distribution Method

In usual method, a client which is in waiting state after distribution tasks, it does not perform any execution like a server. In reuse method after distribution tasks, clients will also be in listening state like server for any execution. It is observed that for less number of PCs (up to eight) usual method execution time is less than reuse method, due to there are fewer levels, so the number of waiting clients after distribution will

be low. In networks with more than eight computers, there are more clients in the waiting state. Thus, the algorithm efficiency is improved by reusing waiting clients in the network.

As a result execution time of reusing method is less than usual method when more than eight numbers of PCs exist.

5.2 Comparison of Two-fold, Seven-fold and Dynamic Distribution Methods

In Sections 4.1 and 4.2, we explained two-fold and seven-fold distribution topologies that are fixed for all entries.

Dynamic distribution topology has been presented in order to improve two previous topologies. Now, we compare the results of these three distribution methods in Table 5.2. All experiment results have been performed by using of three distribution topologies with 2048×2048 matrix dimension and threshold value equal to 128 from one to twenty computers.

Table 5.2: Execution Time of Three Different Distribution Methods

PC Numbers	Execution time, minutes		
	Two-fold Distribution Topology	Seven-fold Distribution Topology	Dynamic Distribution Topology
1	30.2	38.41	30.6
2	27	31.70	16.42
3	16.5	27.50	18.4
4	15.9	25.10	13
5	17	18.45	13.4
6	14.3	13.90	13.35
7	13.9	10	10
8	15	6.28	9.8
9	13.5	6.29	7.7
10	10.8	6.32	7.7
11	9.9	6.25	7.6
12	11.7	6.2	7.55
13	11.6	7.06	6.5
14	10.4	6.50	6.2
15	9	6	6.5
16	9.2	6.60	6.5
17	8.95	6.30	6.52
18	10	6.50	6.53
19	11	6.60	6.51
20	8.9	6.20	6.52

According to Table 5.2, the execution time in all three type of topologies decreases with increasing number of computers. From the experimental results it is observed that seven-fold distribution topology showed better performance than two-fold distribution for large number of PCs (seven and more).The results are presented in a form of graph in Figure 5.2.

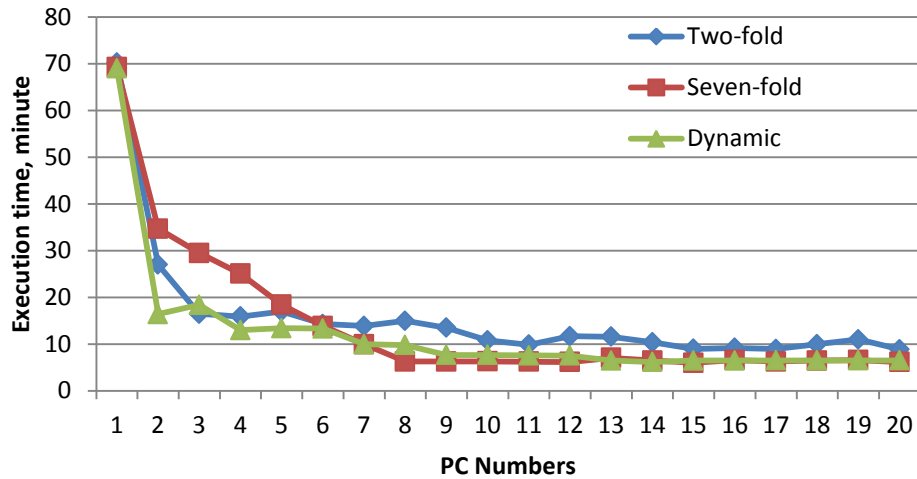


Figure 5.2: Execution Time versus Number of Computers for Three Different Distribution Method

Figure 5.2 shows execution time for Two-fold, Seven-fold and Dynamic distribution methods. In a network with less than seven computers the Two-fold Distribution is better than the Seven-fold Distribution. On the other hand the program execution time for Dynamic Distribution Topology is better than Two-fold and Seven-fold Distribution almost in all number of PCs.

In Table 5.3, values of speed up and efficiency for all the statuses of 2 to 20 PCs, printed in Table 5.2 have been calculated. Process of alteration of speed up and efficiency for all three methods of two-fold, seven-fold and dynamic has been shown in table 5.3.

Table 5.3: Speed-Up and Efficiency of Three Different Distribution Methods

PC Numbers	Two-fold Distribution Topology		Seven-fold Distribution Topology		Dynamic Distribution Topology	
	Speed-up	Efficiency	Speed-up	Efficiency	Speed-up	Efficiency
2	1.11	0.55	1.21	0.60	1.84	0.92
3	1.83	0.61	1.38	0.46	1.65	0.55
4	1.89	0.47	1.53	0.38	2.33	0.58
5	1.77	0.35	2.08	0.41	2.26	0.45
6	2.11	0.35	2.76	0.46	2.27	0.37
7	2.17	0.31	3.84	0.54	3.03	0.43
8	2.01	0.25	6.11	0.76	3.09	0.38
9	2.23	0.24	6.10	0.67	3.94	0.43
10	2.79	0.27	6.07	0.60	3.94	0.39
11	3.05	0.27	6.14	0.55	3.99	0.36
12	2.58	0.21	6.19	0.51	4.02	0.33
13	2.60	0.20	5.44	0.41	4.67	0.35
14	2.90	0.20	5.90	0.42	4.89	0.34
15	3.35	0.22	6.40	0.42	4.67	0.31
16	3.28	0.20	5.81	0.36	4.67	0.29
17	3.37	0.19	6.09	0.35	4.65	0.27
18	3.02	0.16	5.90	0.32	4.65	0.25
19	2.74	0.14	5.81	0.30	4.66	0.24
20	3.39	0.16	6.19	0.30	4.65	0.23

5.3 Performance of Dynamic Distribution Method

Next, we consider our program performance with dynamic distribution topology in different situations. First, we execute the program with fixed matrix dimensions 2048×2048 by changing threshold as 64, 128, 256, and 512; and number of computers as 1, 5, 10, 15 and 20. Here our aim is to see the effect of threshold values to the execution time. The experimental results are presented in Table 5.3 and in the form of graph in Figure 5.3. The execution time is compared in Table 5.3.

Table 5.4: Execution Time of Dynamic Distribution Method by Different Threshold Values and Using Different Number of Computers

PC Numbers	Execution time, minutes			
	64	128	256	512
1	32.25	30.61	31.63	31.88
5	13.1	13.32	13.3	9.9
10	7.4	7.42	5.52	4.55
15	6.17	6.36	5.26	4.55
20	5.44	6.31	4.07	4.55

As shown in Table 5.3, the execution time decreases as the number of available computers in the network increases. Also increasing the threshold value presents little improvement in the execution time, which indicates that the smaller matrix dimension execution is not optimum in parallel form. This signifies that when the matrix dimensions are small enough, it is better to solve the problem on a single machine.

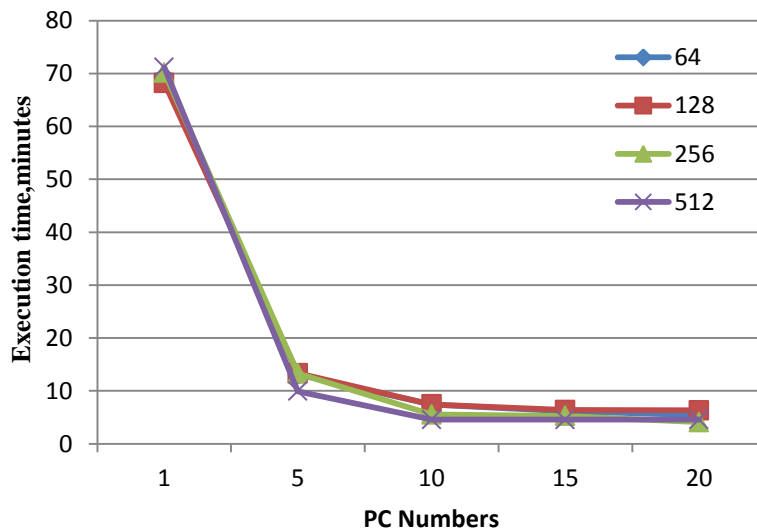


Figure 5.3: Execution Time versus Number of Computers with Different Threshold Values for Dynamic Distribution Method

Figure 5.3 shows the program execution time with different thresholds for two 2048×2048 matrices. According to the results for large matrix dimensions having large threshold values improves the execution time.

To determine the effect of matrix size, we executed the program with various dimensions (128×128 , 256×256 , 512×512 , 1024×1024 and 2048×2048) using fixed threshold 128 in a network with 1, 5, 10, 15 and 20 computers. The results in minutes are compared in Table 5.4.

Table 5.5: Execution Time of Dynamic Distribution Method by Different Matrix Size and Using Different Number of Computers

PC Numbers	Execution Time, minutes				
	128	256	512	1024	2048
1	0.029	0.2	1.02	6.25	30.61
5	0.035	0.09	0.215	1.56	13.32
10	0.032	0.05	0.125	1.23	7.42
15	0.035	0.031	0.124	0.916	6.36
20	0.03	0.031	0.12	0.666	6.31

The execution time decreases as the number of available computers increases. However, the reduction procedure of execution time in columns with bigger matrix dimensions is more rather than columns with smaller matrix dimensions. We clarified these results in Figure 5.4.

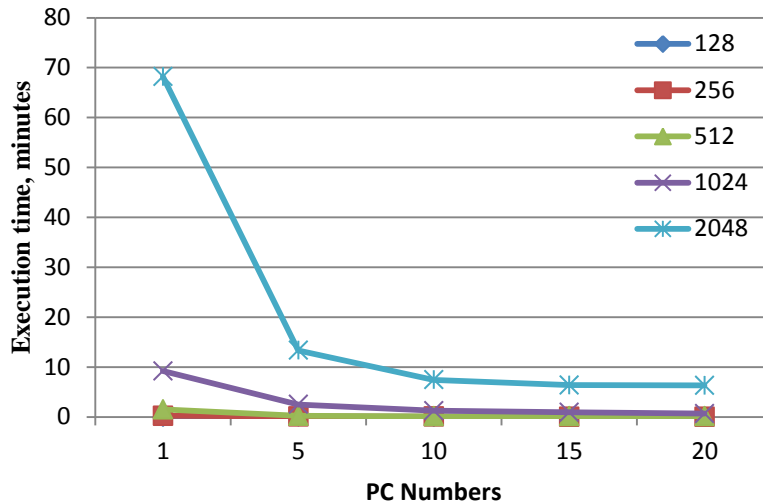


Figure 5.4: Execution Time versus Number of Computers with Different Matrix Size for Dynamic Distribution Method

Figure 5.4 shows program execution time for different matrix dimensions using of fixed threshold. The percentage of execution time improvement in parallel form for larger matrix dimensions (1024, 2048) is more than matrices with smaller dimensions (128, 256, 512). This indicates necessity and importance of parallelism for large matrix dimensions. Increasing the number of available computers (15, 20) also has significant impact on the program execution time.

5.4 Performance of Two-fold and Seven-fold Distribution Method

In continuing, the operations of Two-fold and Seven-fold distribution methods have been indicated in the following tables and their related figures. Note that in Table 5.6 and Table 5.8 input matrices dimensions have been considered as constant while the threshold values have been assumed as variable for figuring out the effect of them whereas, in Table 5.7 and Table 5.9 the input matrices dimensions have been considered as variable for figuring out the effect of them. But the threshold values have been assumed as constant.

Table 5.6: Execution Time of Two-fold Distribution Method by Different Threshold Values and Using Different Number of Computers

PC Numbers	Execution time, minutes			
	64	128	256	512
1	31.28	31.31	30.35	29.23
5	16.98	15.38	14.45	12.31
10	19.21	12.88	7.98	5.63
15	19.00	10.8	7.46	5.63
20	18.45	8.95	6.28	5.63

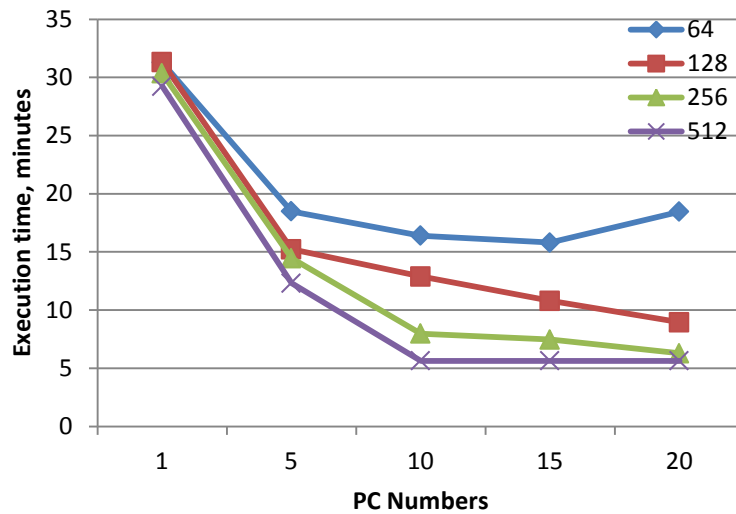


Figure 5.5: Execution Time versus Number of Computers with Different Threshold Values for Two-fold Distribution Method

Table 5.7: Execution Time of Two-fold Distribution Method by Different Matrix Size and Using Different Number of Computers

PC Numbers	Execution Time, minutes				
	128	256	512	1024	2048
1	0.02	0.10	0.58	4.11	31.31
5	0.03	0.06	0.30	2.36	15.38
10	0.03	0.06	0.20	2.05	12.88
15	0.03	0.05	0.15	2.00	10.8
20	0.03	0.05	0.18	1.7	8.50

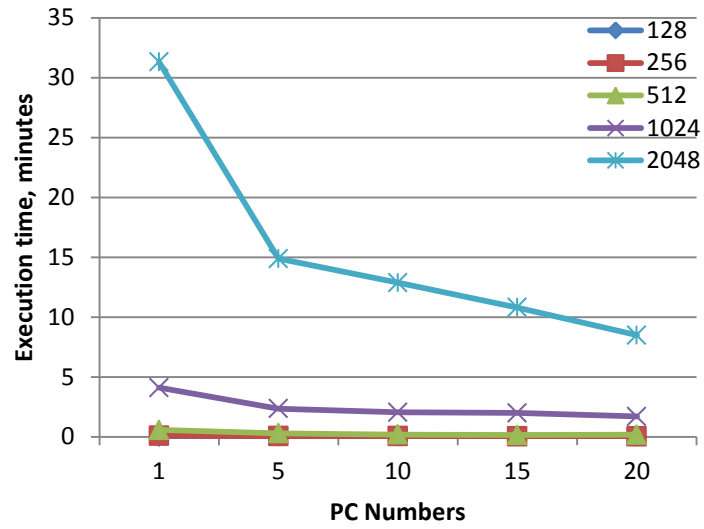


Figure 5.6: Execution Time versus Number of Computers with Different Matrix Size for Two-fold Distribution Method

Table 5.8: Execution Time of Seven-fold Distribution Method by Different Threshold Values and Using Different Number of Computers

PC Numbers	Execution time, minutes			
	64	128	256	512
1	39.31	38.68	39.61	39.28
5	19.23	18.23	18.85	17.53
10	6.50	6.81	6.55	5.03
15	6.51	6.26	6.25	4.95
20	6.53	6.68	6.50	4.88

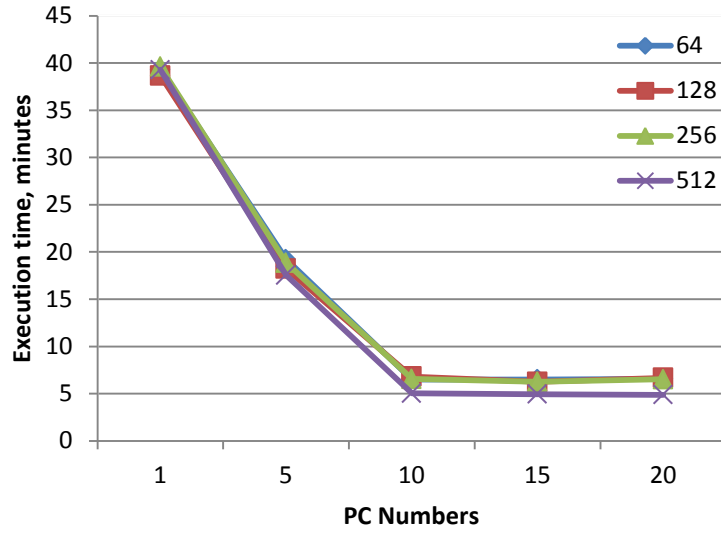


Figure 5.7: Execution Time versus Number of Computers with Different Threshold Values for Seven-fold Distribution Method

Table 5.9: Execution Time of Seven-fold Distribution Method by Different Matrix Size and Using Different Number of Computers

PC Numbers	Execution Time, minutes				
	128	256	512	1024	2048
1	0.02	0.15	0.75	5.33	39.31
5	0.03	0.06	0.38	3.01	18.23
10	0.03	0.01	0.11	1.46	6.81
15	0.03	0.01	0.13	1.38	6.26
20	0.03	0.03	0.21	0.98	6.68

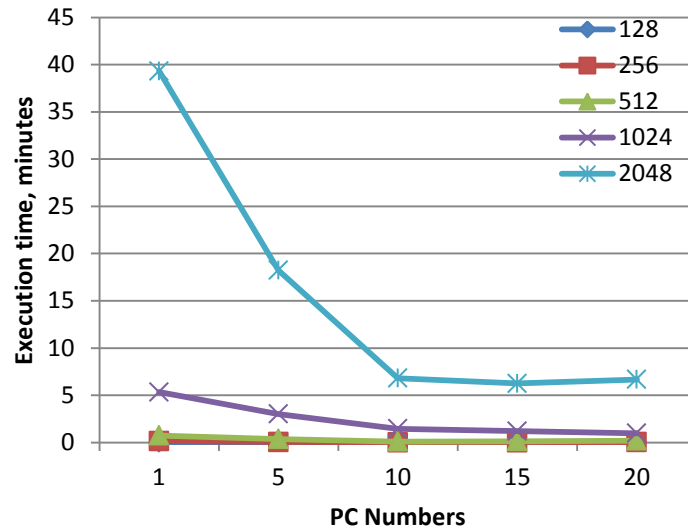


Figure 5.8: Execution Time versus Number of Computers with Different Matrix Size for Seven-fold Distribution Method

5.6 Performance of Fair Distribution Method

Execution time achieved from the experimental results of Fair distribution method has been included in this section. First, in Table 5.10, execution time of program on a network including one to twenty computers has been entered. To do these tests, input matrices with dimension of 2048×2048 has been used. Also, the threshold value used for these tests is 128. In Table 5.10, execution time, speed up and efficiency of program in different situation have been calculated.

Table 5.10: Execution Time, Speed-Up and Efficiency of Fair Distribution Method by Different Number of Computers

PC Numbers	Execution time of Fair distribution method	Performance	
		Speed-Up	Efficiency
1	24.5	- - -	- - -
2	21.11	1.16	0.58
3	18.43	1.32	0.44
4	11.15	2.19	0.73
5	10.5	2.33	0.46
6	10.01	2.44	0.40
7	8.18	2.99	0.42
8	7.25	3.37	0.42
9	6.46	3.79	0.42
10	6.18	3.96	0.39
11	6.18	3.96	0.36
12	6.23	3.93	0.32
13	6.26	3.91	0.30
14	6.2	3.95	0.28
15	5.9	4.15	0.27
16	5.68	4.31	0.26
17	5.66	4.32	0.25
18	5.46	4.48	0.24
19	5.53	4.43	0.23
20	5.55	4.41	0.22

As it is seen in Table 5.10 by the increase of the number of computers we see the decrease in the execution time. As a result of this reduction execution time, we always had improvement in speed up. But, the altering process of efficiency in some points is rising and the rest descending. In applications which we intend to use the parallel program, noticing to the importance of speed up or efficiency, we can choose the ideal situation from the modes of table.

For this version of distribution, too, the results of effects of threshold value changes and different sizes of input matrices have been gathered. To see the effects of threshold changes, size of entering matrices are 2048×2048 . Also, these tests on the

networks including 1, 5, 10, 15 and 20 PCs have been done. Results of these tests are in the table 5.11.

Table 5.11: Execution Time of Fair Distribution Method by Different Threshold Values and Using Different Number of Computers

PC Numbers	Execution time, minutes			
	64	128	256	512
1	23.83	24.3	23.95	24.01
5	11.06	11.50	11.20	10.83
10	6.18	6.18	6.16	6.23
15	5.63	5.90	5.63	5.63
20	7.10	7.08	6.90	6.21

In following Figure 5.9 shows the program execution time with different threshold values for two 2048×2048 input matrices.

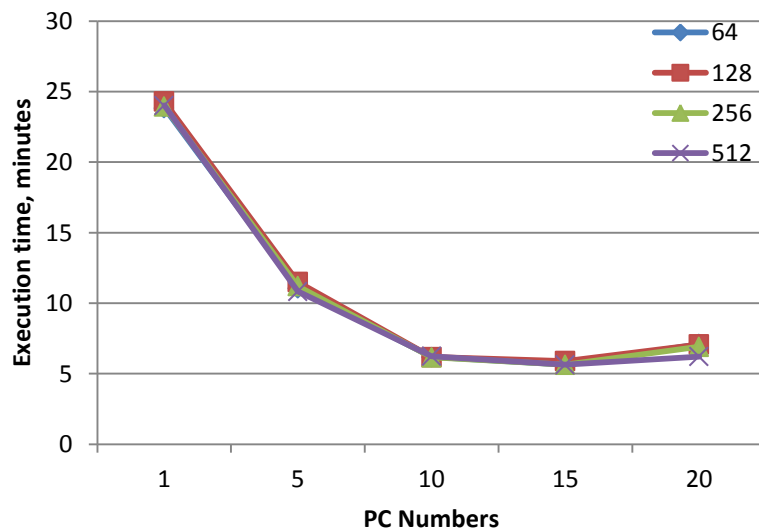


Figure 5.9: Execution Time versus Number of Computers with Different Threshold Values for Fair Distribution Method

For figuring out the effect of changes in size of input matrices, threshold value has been considered as constant while input matrices dimensions have been assumed as

variable. These tests have been done on the networks including 1, 5, 10, 15 and 20 computers. Results of these tests are shown in Table 5.12.

Table 5.12: Execution Time of Fair Distribution Method by Different Matrix Size Using Different Number of Computers

PC Numbers	Execution Time, minutes				
	128	256	512	1024	2048
1	0.02	0.06	0.50	3.45	24.30
5	0.03	0.03	0.16	1.58	11.50
10	0.03	0.03	0.13	0.80	6.18
15	0.03	0.03	0.19	0.61	5.90
20	0.03	0.03	0.18	0.71	6.50

Figure 5.10 shows program execution time for different matrix dimensions using fixed threshold value by Fair distribution method.

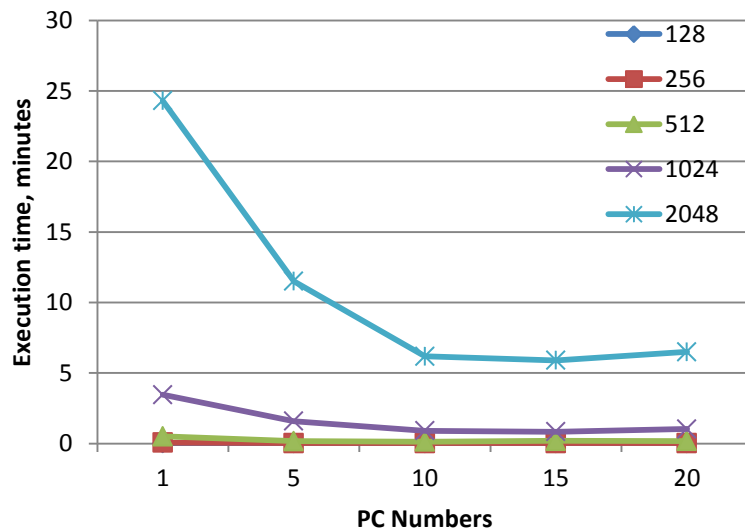


Figure 5.10: Execution time versus Number of Computers with Different Matrix Size for Fair Distribution Method

The method we have used for comparing to our Fair distribution method is Strassen-BMR. The results for Strassen-BMR method in [32] have been already reported over the system defined by following properties. All the applied processors are Intel

iPSC/860. This processor is a high performance parallel computer system. The processing power of the iPSC/860 comes from its processing nodes. Each node in the iPSC/860 is either a CX or an RX processor. Every iPSC/860 system contains at least one RX node. The CX node is based on the Intel386 microprocessor. An RX node consists of an Intel i860 microprocessor capable of a peak performance of 80 MFLOPS. The i860 has multiple arithmetic units: an integer unit, a floating point adder and a floating point multiplier. The processing nodes of the iPSC/860 are interconnected in hypercube architecture having 2GB memory over each node. Peak data transfer rate for inter-processor communication is 176 Mb/s. In a hypercube of dimension n , each node has n neighbors and the total number of nodes in the hypercube is 2^n .

Regarding to what we have just mentioned, the connection type between processors in Strassen-BMR is internal, whereas in our test system, connections is established via cables. Hence, the transfer rate of the Strassen-BMR system is faster. Therefore, the comparison can be done as follows. When the number of applied processors is pretty less and inter-processor communications are not so significant, our method will be working better than Strassen-BMR. In contrast, when the number of used processors is more, inter-processor communications will be considerable. Hence, Strassen-BMR will work better due to fast communication between processors.

Results of STRASSEN.BMR method in the related reference are on four computers by input matrices with the size of 500×500 . The results of our implementation on four computers and entering matrices with the size of 512×512 are achieved, too. In

this comparison, the execution time of program is calculated by second. Results of these comparing are showed in Table 5.13.

Table 5.13: Comparing Execution Time of Strassen-BMR and Fair Distribution Methods

Method	Number of Computers	Matrix Size	Execution time, sec
Strassen-BMR	4	500×500	10
	16	1000×1000	20
Fair distribution method	4	512×512	8.58
	16	1024×1024	34.6

Fair distribution method for parallel Strassen matrix multiplication algorithm has been presented and compared with Strassen-BMR method. When the communication is not very costly compared to computation, Fair distribution method may offer a fast approach for large matrix dimensions.

Chapter 6

CONCLUSION

The aim of this thesis is to study necessities in parallel programming. Due to the development of applying computers in all scientific aspects and need faster processes, it is considerably important to study parallel programming methods of applying hardware and software to solve scientific problems. In order to improve the functionality of parallel algorithms, this thesis proposed data division and distribution methods for implementing parallel calculations in homogeneous networks with different number of nodes. This thesis uses Strassen matrix multiplication algorithm. Strassen method is a recursive algorithm and has been designed based on divide and conquer technique.

During the thesis initially, two types of distribution methods of tasks among computers are designed. Then, by comparing two-fold and seven-fold methods, we figured out constant distribution methods which are not satisfied all circumstances of the program. Thereby, we select dynamic distribution method for division and distribution of tasks belongs to Strassen algorithm. We apply P processors and distribute input matrices of size $n \times n$ over an optimum network topology to perform parallel computation.

The dynamic distribution method is applied as a tree distribution in such a way that most computers are taken into account for the last layer. Hence, the maximum possibility of parallel processing in recursive algorithms is provided.

Speed-up and efficiency as two measurement criterion are calculated using related formulas. When the numbers of computers are ten or twenty, the speed-up values have been calculated as 3.94 and 4.65 respectively. As well as, values of efficiency for the same numbers of computers using related formula are 0.39 and 0.23 respectively.

We compared the execution time in usual and reusing of client methods in two-fold distribution method. In the further, results of three distribution methods named by two-fold, seven-fold and dynamic achieved by experimental results, were compared to each other. It was observed that fixed distribution methods are not optimal, but dynamic distribution method covers each optimum point in the fixed distribution methods.

We are trying to occupy the clients at the begging of execution for improving the performance of the previous methods. This method has been named by Fair distribution.

After that, comparing of the achieved results in Fair distribution (10, 8.58 Sec for Strassen-BMR and Fair distribution respectively) shows better improvement in execution time rather than Strassen-BMR method (where the numbers of computers are four, 500*500 and 512*512 are matrices dimension for Strassen-BMR and Fair

distribution method respectively). Finally, some efficiency improvements are observed in case of having larger matrices in parallel environment.

REFERENCES

- [1] J. D. Plummer, "Material and Process Limits in Silicon VLSI Technology", Stanford University, Stanford, IEEE, Volume. 89, NO. 3, PP. 240-258, March 2001.
- [2] C. D. Martin, "ENIAC: The Press Conference That Shook the World", IEEE Technology and Society Magazine, Volume 14, PP. 3-10, December, 1995.
- [3] W. Gehrke, K. Gaedke, "Associative Controlling of MonolithicParallel Processor Architectures", IEEE, Transactions on Circuits and Systems for Video Technology, Volume 5, NO. 5, PP. 453-464, October 1995.
- [4] H. El-Rewini, M. Abd-El-Barr, "Advanced Computer Architecture and Parallel Processing", John Wiley & Sons, Canada, 2005.
- [5] B. Barney, L. Livermore, "Introduction to Parallel Computing", Mar 2010, https://computing.llnl.gov/tutorials/parallel_comp/, Ebook.
- [6] M. J. Flynn, "Very high-speed computing systems", IEEE, volume. 54, no. 12, pp. 1901–1909, 1966.
- [7] M. J. Flynn, K. W. Rudd, "Parallel Architectures", ACM Computing Surveys, Vol. 28, No. 1, PP. 1479-1496, March 1996.

- [8] R. Eigenmann, D. J. Lilja, "Von Neumann Computers", Wiley Encyclopedia of Electrical and Electronics Engineering, Volume 23, PP. 387-400, January 30, 1998.
- [9] R. Duncan, "A Survey of Parallel Computer Architectures", IEEE, PP. 5-16, February 1990.
- [10] P. S. Pacheco, "An Introduction to Parallel Programming", Burlington, USA, Elsevier Inc, 2011.
- [11] J. Protic, M. Tomagevic, V. Milutinovic, "A Survey of Distributed Shared Memory Systems", Proceedings of the 28th Annual Hawaii International Conference on System Sciences, PP. 61-66, IEEE, 1995.
- [12] W. Gropp, E. Lusk, R. Thakur, "Using MPI-2 Advanced Features of the Message-Passing Interface", Massachusetts Institute of Technology, Wilson, 1999.
- [13] B. Wah, "Interconnection Networks for Parallel Computers", John Wiley & Sons, Inc, PP. 1613- 1623, 2008.
- [14] M. Wimmer, "Programming Models for Parallel Computing", MSC Thesis, Wien university, Wien, 2010.
- [15] F.E. Fich, P. Ragde, A. Wigderson, "Relations Between Concurrent-Write Models of Parallel Computation", SIAM J. Comput, Volume 17, No 3, PP. 606-626, June 1988.

- [16] U. Vishkin, "Thinking in Parallel:Some Basic Data-Parallel Algorithms and Techniques", University of Meryland, October 12, 2010.
- [17] A. Grama, A. Gupta, G. Karypis, V. Kumar, "Introduction to Parallel Computing" Second Edition, Addison Wesley, ISBN: 0-201-64865-2, January 16, 2003.
- [18] L. Hu, I. Gorton, "Performance Evaluation for Parallel Systems: A Survey",MSC Thesis, University of NSW, Australia, October 1997.
- [19] M. A. Oliveira, "Parallel Computing and Parallel Programming", LNEC, April 2010.
- [20] D.L. Eager, J. Zahorjan, E.D. Lazowska, "Speedup Versus Efficiency in Parallel Systems", IEEE, Transactions on Computers, Volume 38, NO. 3, PP. 408-423, MARCH 1989.
- [21] A. Gupta, V. Kumar, "Performance Properties of Large ScaleParallel Systems", Journal of Parallel and Distributed Computing, Volume 19, No.3, pp. 234-244, 1993.
- [22] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani, "Algorithms", MC Graw-Hill, July 18, 2006.
- [23] N. Hyodo, H. Murao, T. Saito, "Matrix Multiplication Made Fast-Practical View of Fast Matrix Operation for Computer Algebra System", Japan Society for Symbolic and Algebraic Computation, Volume 11, No 3,4, PP. 3-19, 2005.

- [24] F. Song, J. Dongarra, S. Moore, "Experiments With Strassen's Algorithm: From Sequential to Parallel", Parallel and Distributed Computing and Systems, IASTED 18th International Conference Parallel and Distributed Computing and Systems, PP 415-421, 2006.
- [25] S. Huss-Lederman, E. Jacobson, " Implementation of Strassen's Algorithm for Matrix Multiplication", Supercomputing '96 proceeding of the 1996 ACM/IEEE Conference on Supercomputing (CDROM), Pittsburgh, PA, USA — November 17 - 22, 1996.
- [26] Automatically Tuned Linear Algebra Software (ATLAS), <http://www.netlib.org/atlas>.
- [27] NetSolve/GridSolve, <http://icl.cs.utk.edu/netsolve>.
- [28] F. Desprez, F. Suter, " Mixed Parallel Implementation of the Top Level Step of Strassen and Winograd Matrix Multiplication Algorithms", Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01), San Francisco, 14 March 2001.
- [29] B. Grayson, A. P. Shah, R. A. Geijn, " A High Performance Parallel Strassen Implementation" Parallel Process, University of Texas at Austin, TX, USA, 1995.
- [30] C. Baransel, K. M. Imre, "A parallel implementation of Strassen's matrix multiplication algorithm for wormhole-routed all-port 2D torus networks", The Journal of Supercomputing, Volume 62, Issue 1, PP 486-509, October 2012.

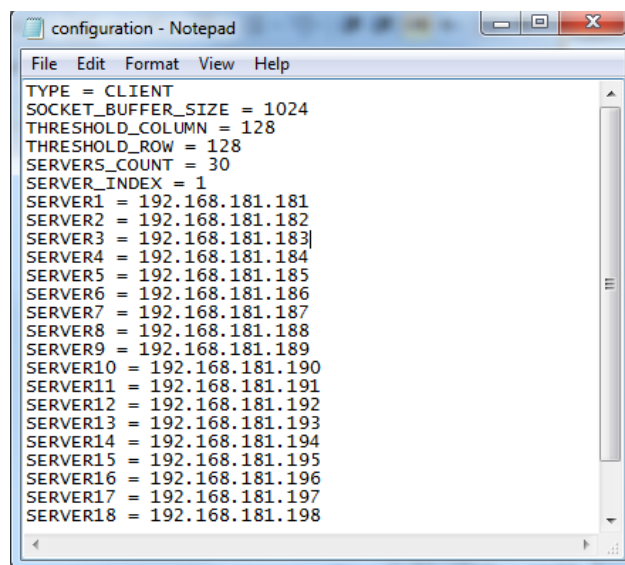
[31] Y. Ohtaki, " Parallel Implementation of Strassen's Matrix Multiplication Algorithm for Heterogeneous Clusters", Proceedings of the 18th International Parallel and Distributed Processing Symposium, PP. 220-228, IEEE, 2004

[32] Q. Luo, J. B. Drake, "A Scalable Parallel Strassen's Matrix Multiply Algorithm for Distributed Memory Computers", in proceedings of the Symposium on Applied Computing, SAC'95, Nashville, TN. Feb 26-28, ACM Press, 1995.

APPENDICES

APPENDIX A: User Guide

At the starting point of perform, program can have 2 different positions, whether to perform in the role of client or to start work in the role of server. If it starts to work as client, directly finds the topology of distribution and will process the division and distribution of the work, but if it is in the role of server, it will be waiting in to receive work from client. In the program there is a configuration file which includes the settings of program implementation which are actually entries of program too. The figure of this file is shown below. We explain the existing information in the file and the information which will be entered sequentially.



```
configuration - Notepad
File Edit Format View Help
TYPE = CLIENT
SOCKET_BUFFER_SIZE = 1024
THRESHOLD_COLUMN = 128
THRESHOLD_ROW = 128
SERVERS_COUNT = 30
SERVER_INDEX = 1
SERVER1 = 192.168.181.181
SERVER2 = 192.168.181.182
SERVER3 = 192.168.181.183
SERVER4 = 192.168.181.184
SERVER5 = 192.168.181.185
SERVER6 = 192.168.181.186
SERVER7 = 192.168.181.187
SERVER8 = 192.168.181.188
SERVER9 = 192.168.181.189
SERVER10 = 192.168.181.190
SERVER11 = 192.168.181.191
SERVER12 = 192.168.181.192
SERVER13 = 192.168.181.193
SERVER14 = 192.168.181.194
SERVER15 = 192.168.181.195
SERVER16 = 192.168.181.196
SERVER17 = 192.168.181.197
SERVER18 = 192.168.181.198
```

Configuration File

Type: It implies the kind of computer at which program performs on and has 2 kinds of client or server.

SOCKET_BUFFER_SIZE: is the buffer size of receiving and sending data. For instance when SOCKET_BUFFER_SIZE is 1024, the data that should be sent is divided to 1024 byte packages. And, these packages are sent sequentially and one

after another. It should be mentioned that amount of `SOCKET_BUFFER_SIZE` must be equal in both sender and receiver.

`THRESHOLD_ROW`, `THRESHOLD_COLUMN`: It is threshold for rows and columns of matrices, here a user defines that to what optimum limit matrices should be divided and distributed. Whatever is the threshold, division and distribution of the matrices are stopped and continuation of the calculation is performed by local computer.

`SERVER_COUNT`: It displays number of all computers in network. For instance if we have 30 computers in network, it equals to 30.

`SERVER_INDEX`: It equals with an index which IP of local computer has come to this index in the configuration file.

`SERVER1`, `SERVER2`, etc: IP of existing computers in network are entered sequentially these variables. Computers which should work on division and distribution in the role of client will identify existing computers in authorized network and with having their IP attempt to distribute the work.

Configuration file is in the follow path in program file:

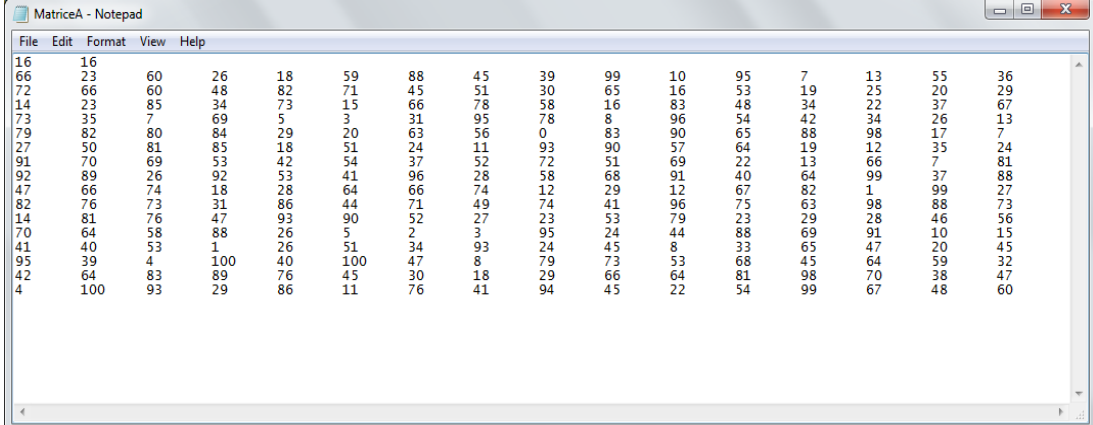
StrassenMatrixMultiplication/ bin / Debug / Configuration

Two files of entering matrix which multiplication task should be performed on, too, are copied in the above path. It should be mentioned here that after copying the

program file to the computers, settings of configuration file should be conducted on every single computer. After doing the settings, program will be performed on all server computers and while server computers are in listening status, programs of client computer are performed.

Performing the client program, after finding the distribution topology via client, steps of sending process to server computers according to mentioned procedure are started and eventually after receiving results of distributed calculations, these results are stored in an output file which has been created in the path of entering files via client.

To make sure of correctness, obtained results of multiplying 16×16 matrices are illustrated in the following. Figures 6.2 and 6.3 show the input matrices and Figure 6.4 presents the output of multiplication calculated by the program.



```
MatriceA - Notepad
File Edit Format View Help
16 16
66 23 60 26 18 59 88 45 39 99 10 95 7 13 55 36
72 66 60 48 82 71 45 51 30 65 16 53 19 25 20 29
14 23 85 34 73 15 66 78 58 16 83 48 34 22 37 67
73 35 7 69 5 3 31 95 78 8 96 54 42 34 26 13
79 82 80 84 29 20 63 56 0 83 90 65 88 98 17 7
27 50 81 85 18 51 24 11 93 90 57 64 19 12 35 24
91 70 69 53 42 54 37 52 72 51 69 22 13 66 7 81
92 89 26 92 53 41 96 28 58 68 91 40 64 99 37 88
47 66 74 18 28 64 66 74 12 29 12 67 82 1 99 27
82 76 73 31 86 44 71 49 74 41 96 75 63 98 88 73
14 81 76 47 93 90 52 27 23 53 79 23 29 28 46 56
70 64 58 88 26 5 2 3 95 24 44 88 69 91 10 15
41 40 53 1 26 51 34 93 24 45 8 33 65 47 20 45
95 39 4 100 40 100 47 8 79 73 53 68 45 64 59 32
42 64 83 89 76 45 30 18 29 66 64 81 98 70 38 47
4 100 93 29 86 11 76 41 94 45 22 54 99 67 48 60
```

Input Matrix A

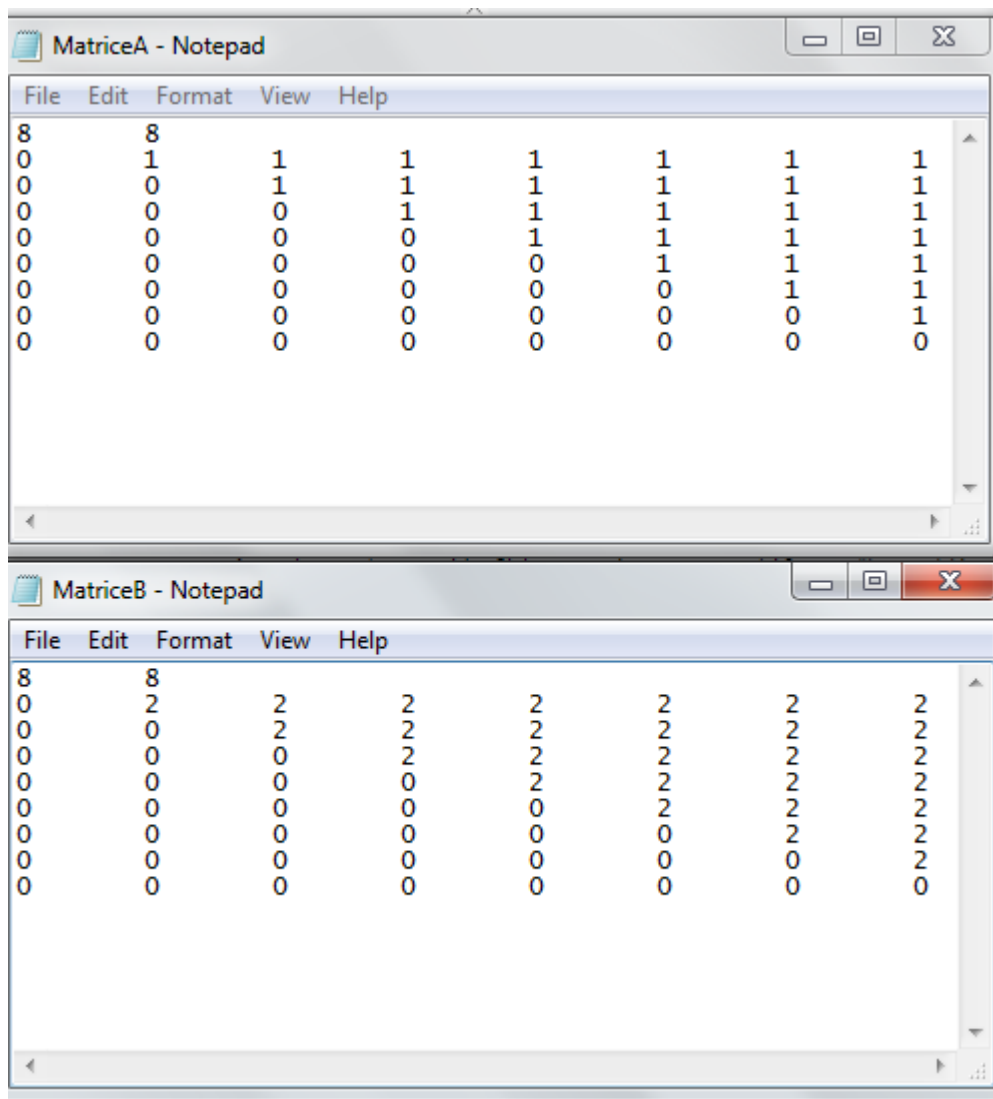
File	Edit	Format	View	Help												
16	16															
4	97	2	8	4	42	45	7	66	91	64	94	18	71	34	94	
93	54	73	29	51	46	66	17	96	20	72	27	49	46	27	81	
54	61	43	55	53	3	71	13	70	7	82	80	53	58	62	52	
26	78	52	25	86	22	19	58	31	86	13	85	71	18	58	20	
63	84	39	57	27	79	40	90	84	59	60	43	65	86	64	37	
58	19	0	72	94	2	95	32	42	86	24	68	90	87	43	52	
49	61	17	69	19	5	25	47	38	63	42	26	63	76	94	45	
21	76	26	51	17	87	27	61	90	7	14	42	54	72	100	97	
16	7	95	3	69	95	18	53	95	53	33	30	60	68	69	66	
92	80	47	14	60	92	34	62	34	56	80	12	91	84	94	66	
63	45	10	55	82	90	77	26	58	73	57	64	53	97	0	21	
38	49	50	75	25	86	36	24	73	32	70	12	14	6	63	11	
60	68	68	15	58	13	85	2	3	20	56	88	35	71	8	38	
55	59	74	42	33	18	40	78	8	86	83	49	20	64	12	51	
10	46	4	86	28	86	64	25	64	47	72	5	29	49	90	2	
24	31	57	56	33	21	78	42	27	91	6	72	31	36	83	39	

Input Matrix B

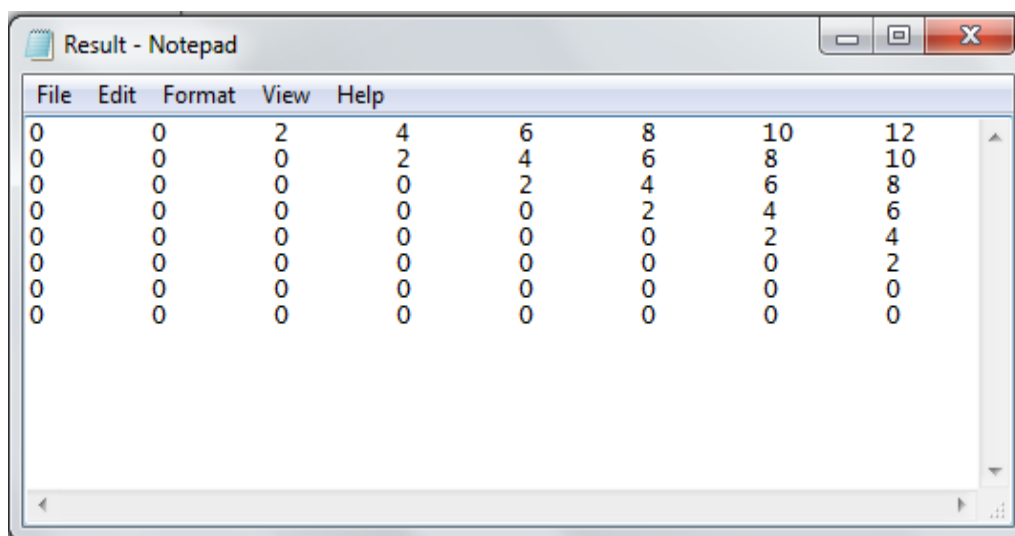
File	Edit	Format	View	Help												
32653	42940	26029	35361	30711	38171	34683	28190	42157	39617	40034	31057	38220	44129	50686	36572	
36367	46123	28917	33016	33623	36432	37342	31082	45123	40759	39697	37521	40376	47156	44543	40249	
32691	41970	30526	37008	33532	39569	38001	31375	45253	37527	36649	37677	37550	47483	45617	34552	
24768	38981	27445	25892	30802	40133	29848	25351	40686	35984	31257	35103	30536	40840	34425	34567	
48480	61078	39364	38083	43804	42539	47516	35071	47112	49327	54910	50448	45305	58284	44858	47020	
35358	39577	33526	29778	41545	39874	35977	28457	43357	39790	39263	35370	41481	43539	43268	34284	
38112	48091	36074	33845	39983	39844	43406	34268	48465	50588	42434	46790	42632	53914	45421	46785	
49654	62231	45468	43431	49245	48285	53214	43220	53378	65489	54546	55554	51694	65811	54706	51832	
33829	44512	26726	38767	31987	35551	42905	23846	44416	33611	41141	36014	36545	45836	46754	36597	
50278	63214	46014	51536	48453	56917	58615	43883	63002	61926	61873	54627	51228	70181	59478	52544	
43288	45339	30902	40146	42237	37874	46588	33385	46490	45096	42581	41027	45822	53636	44885	37583	
33190	43140	40322	26137	37030	36800	35207	28108	41089	41183	42744	40735	32584	40926	33319	35757	
29034	36996	26011	27477	26025	28494	33474	24205	34024	29510	31786	32166	30633	40405	36755	35782	
39095	50752	35698	37245	45997	44609	44614	36301	46772	58439	45902	46030	46628	55246	48100	42101	
47046	56303	42117	40458	46486	43067	50087	36228	47698	50247	52282	50103	46123	55571	47136	41184	
46360	51555	47794	39760	41653	44023	47133	37568	52740	42898	51306	43096	45256	56340	52519	44888	

Result of Multiplying A and B

Next example is presented over 8×8 input matrices. Matrices are in the form of upper triangular contains simple elements. Figure 6.5 and Figure 6.6 illustrate the input matrices and output matrix respectively.

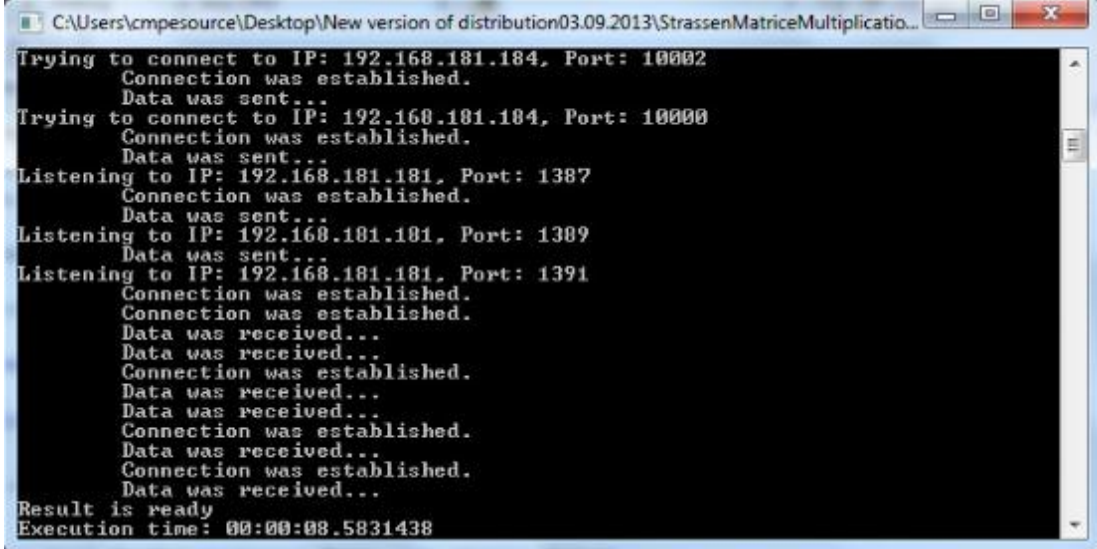


Input Matrices



Result Matrix

Meanwhile, the execution time of program performed over 4 processors, matrices by size 512 and 128 as threshold value is shown in figure 6.7.



```
CA\Users\cmpeource\Desktop\New version of distribution03.09.2013\StrassenMatriceMultiplicatio...
Trying to connect to IP: 192.168.181.184, Port: 10002
  Connection was established.
  Data was sent...
Trying to connect to IP: 192.168.181.184, Port: 10000
  Connection was established.
  Data was sent...
Listening to IP: 192.168.181.181, Port: 1387
  Connection was established.
  Data was sent...
Listening to IP: 192.168.181.101, Port: 1309
  Data was sent...
Listening to IP: 192.168.181.181, Port: 1391
  Connection was established.
  Connection was established.
  Data was received...
  Data was received...
  Connection was established.
  Data was received...
  Data was received...
  Connection was established.
  Data was received...
  Connection was established.
  Data was received...
Result is ready
Execution time: 00:00:08.5831438
```

An Example of a Test Execution Time

APPENDIX B: Programming Part

The program below has been implemented to divide the matrices and distribute them over the network nodes. The following information illustrates the mentioned program.

Property	Explanation
Author	Reza Abri Vaighan
Name of the program	Matrix Multiplication Distributer
Language	C#
Program type	Object-Oriented
Release date	05.06.2013
Purpose	Investigating Strassen matrices multiplication algorithm to be executed over distributed systems
Usage	Finding a better solution for matrix multiplication by distributed execution
Headers of the program	We have applied only C# system files and no extra library have been used.

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;

namespace StrassenMatriceMultiplication
{
    class Program
    {
        static ConfigurationHandler Config =
            new ConfigurationHandler("configuration.cfg");
        static bool IsClient;
        static string Status = "Idle";
        static DateTime ReservedTime = new DateTime();
    }
}
```

```

staticstring Command = "";
staticstring Tree = "";
staticstring LocalIP = "";
static System.IO.StreamWriter file = new System.IO.StreamWriter("log.txt");
static System.Threading.Mutex Mtx = new System.Threading.Mutex();

staticvoid Main(string[] args)
{
string TypeValue = "";
if (Config.GetValue("TYPE", out TypeValue))
    IsClient = (TypeValue.ToUpper() == "CLIENT") ? true :false;
else
{
Console.WriteLine("Error: Could not read TYPE value from configuration file");
return;
}
}

```

In the beginning of the program running, a question is asked from user, then user, for running of the program on local computer, should press "Enter" key and for distributing of the program in order to parallel execution should press any other key.

```

if (IsClient)
{
Console.WriteLine("Press \"Enter\" key to execute in local machine,\r\nnor
other key to send to servers...");
ConsoleKeyInfo key = Console.ReadKey(true);
Console.Clear();
if (key.Key == ConsoleKey.Enter)
{
ExecuteClient(false);
return;
}
}

string ServerIndex = "";
Config.GetValue("SERVER_INDEX", out ServerIndex);
Config.GetValue("SERVER" + ServerIndex, out LocalIP);

```

```

        System.Threading.Thread StatusResponder =
newSystem.Threading.Thread(() => ResponseToStatus());
StatusResponder.Start();

```

```

        System.Threading.Thread CommandResponder =
newSystem.Threading.Thread(() => ResponseToCommand());
CommandResponder.Start();

```

This section is related to starting point of client activity in the role of server which is again related to renewed usage.

```

if (IsClient)
{
        System.Threading.Thread Server =
newSystem.Threading.Thread(() => ExecuteServer());
Server.Start();
}

```

```

ExecuteClient(true);

if (Server.IsAlive)
    Server.Join();
    }
else
    ExecuteServer();

if (StatusResponder.IsAlive)
    StatusResponder.Join();
if (CommandResponder.IsAlive)
    CommandResponder.Join();

    }

```

This function finds optimum distribution topology regarding to inputs automatically.

```

staticstring FindBestTopology(int Count, int Level)
    {
    string Topology = GetBestTopologyFromFile(Count, Level);
    if (Topology != "")
    return Topology;

    ArrayList Tree = newArrayList();
    ArrayList LevelArray = newArrayList();

        if (Count == 2)
        {
        Tree.Add(-1);
        LevelArray.Add(1);

        Tree.Add(0);
        LevelArray.Add(2);
        }
    elseif (Count == 8)
    {
        Tree.Add(-1);
        LevelArray.Add(1);

        Tree.Add(0);
        LevelArray.Add(2);
        Tree.Add(0);
        LevelArray.Add(2);
        Tree.Add(0);
        LevelArray.Add(2);
        Tree.Add(0);
        LevelArray.Add(2);
        Tree.Add(0);
        LevelArray.Add(2);
        Tree.Add(0);
        LevelArray.Add(2);
        Tree.Add(0);
        LevelArray.Add(2);
        Tree.Add(0);
        LevelArray.Add(2);
        }
    else
        {

        Count, 1, Level);
        GetBestTopology(ref Tree, ref LevelArray, -1,
    }
}

```

```

        string Printable = "";
for (int i = Tree.Count - 1; i >= 0; i--)
    {
        Printable = "\n" + Printable;
        Printable = "Server" + i + Printable;
for (int j = 0; j < (int)LevelArray[i] - 1; j++)
        Printable = "\t" + Printable;
    }

Console.WriteLine("Topology : ");
Console.WriteLine("=====");
Console.WriteLine(Printable);
Console.WriteLine("=====");
Console.Out.Flush();

string Output = "";
for (int i = 0; i < Tree.Count - 1; i++)
    Output += (i + "," + (Tree[i].ToString() + ";"));
    Output += ((Tree.Count - 1) + "," + Tree[Tree.Count -
1].ToString());

return Output;
    }

static int GetLeavesCount(ArrayList Level, int MaximumLevel)
    {

int Count = 0;
int LastNumber = -2;
for (int i = 0; i < Level.Count; i++)
    {
if ((int)Level[i] == MaximumLevel)
        {
            Count++;
if (LastNumber != (int)Level[i])
                {
                    Count += 2;
                    LastNumber = (int)Level[i];
                }
        }
    }
return Count;
    }

static string GetBestTopologyFromFile(int Count, int Level)
    {
string[] Topologies = System.IO.File.ReadAllLines("topology.txt");
for (int i = 0; i < Topologies.Length; i++)
    {
string[] Parts = Topologies[i].Split('|');
int tmpCount = -1;
int tmpLevel = -1;
char CountOperand = Parts[0][0];
Parts[0] = Parts[0].Substring(1);

if (Parts[0].IndexOf('=') >= 0)
        {
            tmpCount = Convert.ToInt32(Parts[0].Substring(0, Parts[0].IndexOf('=')));
            tmpLevel = Convert.ToInt32(Parts[0].Substring(Parts[0].IndexOf('=') + 1));

if (CountOperand == '=')

```



```

        {
    if (Count == tmpCount && Level == tmpLevel)
    return Parts[1];
        }
    elseif (CountOperand == '>')
        {
    if (Count >= tmpCount && Level == tmpLevel)
    return Parts[1];
        }
    elseif (CountOperand == '<')
        {
    if (Count <= tmpCount && Level == tmpLevel)
    return Parts[1];
        }
    }
    elseif (Parts[0].IndexOf('<') >= 0)
        {
    tmpCount = Convert.ToInt32(Parts[0].Substring(0, Parts[0].IndexOf('<')));
    tmpLevel = Convert.ToInt32(Parts[0].Substring(Parts[0].IndexOf('<') + 1));

    if (CountOperand == '=')
        {
    if (Count == tmpCount && Level <= tmpLevel)
    return Parts[1];
        }
    elseif (CountOperand == '>')
        {
    if (Count >= tmpCount && Level <= tmpLevel)
    return Parts[1];
        }
    elseif (CountOperand == '<')
        {
    if (Count <= tmpCount && Level <= tmpLevel)
    return Parts[1];
        }
    }
    elseif (Parts[0].IndexOf('>') >= 0)
        {
    tmpCount = Convert.ToInt32(Parts[0].Substring(0, Parts[0].IndexOf('>')));
    tmpLevel = Convert.ToInt32(Parts[0].Substring(Parts[0].IndexOf('>') + 1));

    if (CountOperand == '=')
        {
    if (Count == tmpCount && Level >= tmpLevel)
    return Parts[1];
        }
    elseif (CountOperand == '>')
        {
    if (Count >= tmpCount && Level >= tmpLevel)
    return Parts[1];
        }
    elseif (CountOperand == '<')
        {
    if (Count <= tmpCount && Level >= tmpLevel)
    return Parts[1];
        }
    }
    }
}

return "";
}

```

```

static int GetBestTopology(ref ArrayList Tree, ref ArrayList Level, int index,
int Count, int ChildrenCount, int MaximumLevel)
{
if (ChildrenCount > Count - Tree.Count)

return -1;

for (int i = 0; i < ChildrenCount; i++)
{
Tree.Add(index); //index of parent
if (index == -1)
Level.Add(1);
else
Level.Add((int)Level[index] + 1);

if (Tree.Count == Count)
return GetLeavesCount(Level, MaximumLevel);
}

ArrayList T2 = (ArrayList)Tree.Clone();
ArrayList T3 = (ArrayList)Tree.Clone();
ArrayList T4 = (ArrayList)Tree.Clone();
ArrayList T7 = (ArrayList)Tree.Clone();

ArrayList L2 = (ArrayList)Level.Clone();
ArrayList L3 = (ArrayList)Level.Clone();
ArrayList L4 = (ArrayList)Level.Clone();
ArrayList L7 = (ArrayList)Level.Clone();

int[] Res = new int[4];
Res[0] = GetBestTopology(ref T2, ref L2, index + 1, Count, 2, MaximumLevel);
Res[1] = GetBestTopology(ref T3, ref L3, index + 1, Count, 3, MaximumLevel);
Res[2] = GetBestTopology(ref T4, ref L4, index + 1, Count, 4, MaximumLevel);
Res[3] = GetBestTopology(ref T7, ref L7, index + 1, Count, 2, MaximumLevel);

int max = Res[0];
int ind = 0;
for (int i = 1; i < Res.Length; i++)
if (Res[i] > max)
{
max = Res[i];
ind = i;
}

if (ind == 0)
{
Tree = (ArrayList)T2.Clone();
Level = (ArrayList)L2.Clone();
}
elseif (ind == 1)
{
Tree = (ArrayList)T3.Clone();
Level = (ArrayList)L3.Clone();
}
elseif (ind == 2)
{
Tree = (ArrayList)T4.Clone();
Level = (ArrayList)L4.Clone();
}
elseif (ind == 3)
{
Tree = (ArrayList)T7.Clone();
}
}

```

```

        Level = (ArrayList)L7.Clone();
    }

return max;
}

static void ResponseToStatus()
{
int ReceivePort = 10001;

while (true)
{
Mtx.WaitOne();

TCPIPsocket socket = new TCPIPsocket("", LocalIP, ReceivePort, 0);
byte[] Message = socket.ReceiveMessage();
if (Encoding.ASCII.GetString(Message) == "Status")
{
if (Status == "Idle")
{
Status = "IdleReserved";
socket.SendMessage(Encoding.ASCII.GetBytes("Idle"));
file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + "Idle" + " was sent
to" + socket.RemoteIP);
file.Flush();
Console.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + "Idle" + "\t" +
socket.RemoteIP);
}
elseif (Status == "Waiting")
{
Status = "WaitingReserved";
socket.SendMessage(Encoding.ASCII.GetBytes("Waiting"));
file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + "Waiting" + " was sent
to" + socket.RemoteIP);
file.Flush();
Console.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + "Waiting" + "\t" +
socket.RemoteIP + "\t");
ReservedTime = DateTime.Now;
}
elseif (Status == "WaitingReserved")
{
Console.WriteLine("==>" + DateTime.Now + "\t\t" + ReservedTime);
if (ReservedTime.AddSeconds(60) > DateTime.Now)
{
socket.SendMessage(Encoding.ASCII.GetBytes(Status));
}
else
{
Status = "Waiting";
socket.SendMessage(Encoding.ASCII.GetBytes(Status));
}
file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + Status + " was sent
to" + socket.RemoteIP);
file.Flush();
Console.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + Status + "\t" +
socket.RemoteIP);
}
else
{
socket.SendMessage(Encoding.ASCII.GetBytes(Status));
}
}
}
}

```

```

file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + Status + " was sent
to" + socket.RemoteIP);
file.Flush();
Console.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + Status + "\t" +
socket.RemoteIP);
        }
    }
else
    {

socket.SendMessage(Encoding.ASCII.GetBytes("UNKNOWN"));
file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + "UNKNOWN" + " was sent
to" + socket.RemoteIP);
file.Flush();
    }

Mtx.ReleaseMutex();
    }
}

static void ResponseToCommand()
{

int ReceivePort = 10002;

while (true)
    {
        TCPIPsocket socket = new TCPIPsocket("", LocalIP, ReceivePort, 0);
        byte[] Message = socket.ReceiveMessage();

                Status = "WaitingWaiting";
                Tree = Encoding.ASCII.GetString(Message);
file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + Command + " was
received from" + socket.RemoteIP);
file.Flush();
    }
}

static void ExecuteClient(bool IsStrassen)
{
Console.WriteLine("Start : " + DateTime.Now);
    System.Diagnostics.Stopwatch stopWatch =
new System.Diagnostics.Stopwatch();
stopWatch.Start();
    TimeSpan ExecutionTime = new TimeSpan();

int Port = 10000;

string ThresholdColumn;
string ThresholdRow;
ConfigurationHandler config = new ConfigurationHandler("configuration.cfg");
config.GetValue("THRESHOLD_COLUMN", out ThresholdColumn);
config.GetValue("THRESHOLD_ROW", out ThresholdRow);
if (ThresholdRow != ThresholdColumn)
    {
        Console.WriteLine("Error: THRESHOLD_ROW and THRESHOLD_COLUMN must be equal.");
        Console.Read();
        return;
    }
}

```

```

int[,] Matrices = newint[2][,];
Matrices[0] = MatriceUtilities.ReadMatriceFromFile("MatriceA.txt");
Matrices[1] = MatriceUtilities.ReadMatriceFromFile("MatriceB.txt");

if (Matrices[0].GetLength(0) != Matrices[0].GetLength(1) &&
    Matrices[0].GetLength(1) != Matrices[1].GetLength(0) &&
    Matrices[1].GetLength(0) != Matrices[1].GetLength(1))
{
    Console.WriteLine("Error: Input matrices are invalid. Check rows and columns
count.");
    Console.Read();
    return;
}

for (int i = 0; i < Matrices.GetLength(0); i++)
    Matrices[i] = MatriceUtilities.AddZero(Matrices[i]);

int[,] ResultMatrice = null;

```

This section is for a status which running of the program via user on local machine has been chosen or division of matrices have reached to their maximum level and again local computer is responsible for the continuation of performing program.

```

if (IsStrassen == false ||
    (Matrices[0].GetLength(0) <= Convert.ToInt32(ThresholdRow) &&
    Matrices[0].GetLength(1) <= Convert.ToInt32(ThresholdColumn) &&
    Matrices[1].GetLength(0) <= Convert.ToInt32(ThresholdRow) &&
    Matrices[1].GetLength(1) <= Convert.ToInt32(ThresholdColumn)))
{
    Status = "Busy";
    Console.WriteLine("Multiplication is executing on local machine.");
    MatriceUtilities.Multiplication(Matrices[0], Matrices[1], ref ResultMatrice);
    MatriceUtilities.WriteToFile("Result.txt", ResultMatrice);
    Console.WriteLine("Result is ready");
    Status = "Idle";
    stopWatch.Stop();
    ExecutionTime = stopWatch.Elapsed;
    Console.WriteLine("Execution time: " + ExecutionTime.ToString());
    Console.Read();
    return;
}

Status = "Waiting";

int[,] A11 = null, A12 = null, A21 = null, A22 = null;
int[,] B11 = null, B12 = null, B21 = null, B22 = null;

MatriceUtilities.StrassenDivide(Matrices[0], out A11, out A12, out A21, out
A22);
MatriceUtilities.StrassenDivide(Matrices[1], out B11, out B12, out B21, out
B22);

```

```

int[,] A11A12 = null, A21A11 = null, B11B12 = null, A12A22 = null, B21B22 =
null;
int[,] A11A22 = null, B11B22 = null, A21A22 = null, B12B22 = null, B21B11 =
null;
MatriceUtilities.Add(A11, A22, ref A11A22);
MatriceUtilities.Add(B11, B22, ref B11B22);
MatriceUtilities.Add(A21, A22, ref A21A22);
MatriceUtilities.Subtract(B12, B22, ref B12B22);
MatriceUtilities.Subtract(B21, B11, ref B21B11);
MatriceUtilities.Add(A11, A12, ref A11A12);
MatriceUtilities.Subtract(A21, A11, ref A21A11);
MatriceUtilities.Add(B11, B12, ref B11B12);
MatriceUtilities.Subtract(A12, A22, ref A12A22);
MatriceUtilities.Add(B21, B22, ref B21B22);

```

```

int[,] P1 = null, P2 = null, P3 = null, P4 = null, P5 = null, P6 = null, P7 =
null;

```

```

int[,] P1_Parts = null, P2_Parts = null, P3_Parts = null, P4_Parts = null,
P5_Parts = null, P6_Parts = null, P7_Parts = null;

```

```

string val = "";
config.GetValue("SERVERS_COUNT", out val);
int ServerCount = Convert.ToInt32(val);
int Level = (int)(Math.Log(Matrices[0].GetLength(0), 2) -
Math.Log(Convert.ToInt32(ThresholdRow), 2));
Tree = FindBestTopology(ServerCount, Level);
string[] Topology = Tree.Split(';');

```

```

config.GetValue("SERVER_INDEX", out val);
string ServerIndex = (Convert.ToInt32(val) - 1).ToString();
ArrayList Servers = new ArrayList();
int ChildrensCount = 0;
for (int i = 0; i < Topology.Length; i++)
{
string[] TopologyParts = Topology[i].Split(';');
if (TopologyParts[1] == ServerIndex)
{
ChildrensCount++;
config.GetValue("SERVER" + (Convert.ToInt32(TopologyParts[0]) + 1), out val);
Servers.Add(val);
}
}

```

```

System.Threading.Thread MulThread1 = null;
System.Threading.Thread MulThread2 = null;
System.Threading.Thread MulThread3 = null;
System.Threading.Thread MulThread4 = null;
System.Threading.Thread MulThread5 = null;
System.Threading.Thread MulThread6 = null;
System.Threading.Thread MulThread7 = null;

```

In this section of the program, division modes of 7 multiplication operations of Strassen algorithm among servers have been defined.

```

if (Servers.Count == 0)
{
    MulThread1 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11A22, B11B22, ref P1));
MulThread1.Start();
    MulThread2 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A21A22, B11, ref P2));
MulThread2.Start();
    MulThread3 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11, B12B22, ref P3));
MulThread3.Start();
    MulThread4 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A22, B21B11, ref P4));
MulThread4.Start();
    MulThread5 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11A12, B22, ref P5));
MulThread5.Start();
    MulThread6 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A21A11, B11B12, ref P6));
MulThread6.Start();
    MulThread7 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A12A22, B21B22, ref P7));
MulThread7.Start();
}
elseif (Servers.Count == 1)
{
    System.Threading.Mutex mtx0 = newSystem.Threading.Mutex();
    MulThread1 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11A22, B11B22, ref P1));
MulThread1.Start();
    MulThread2 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A21A22, B11, ref P2));
MulThread2.Start();
    MulThread3 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11, B12B22, ref P3));
MulThread3.Start();
    MulThread4 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A22, B21B11, ref P4));
MulThread4.Start();
    MulThread5 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A12, B22, out P5_Parts, mtx0,
Tree));
MulThread5.Start();
    MulThread6 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx0,
Tree));
MulThread6.Start();
    MulThread7 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx0,
Tree));
MulThread7.Start();
}

```

```

    }
elseif (Servers.Count == 2)
{
    System.Threading.Mutex mtx0 = new System.Threading.Mutex();
    System.Threading.Mutex mtx1 = new System.Threading.Mutex();
    MulThread1 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A22, B11B22, out P1_Parts, mtx0,
Tree));
MulThread1.Start();
    MulThread2 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A22, B11, out P2_Parts, mtx0,
Tree));
MulThread2.Start();
    MulThread3 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11, B12B22, out P3_Parts, mtx0,
Tree));
MulThread3.Start();
    MulThread4 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A22, B21B11, out P4_Parts, mtx0,
Tree));
MulThread4.Start();
    MulThread5 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A11A12, B22, out P5_Parts, mtx1,
Tree));
MulThread5.Start();
    MulThread6 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx1,
Tree));
MulThread6.Start();
    MulThread7 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx1,
Tree));
MulThread7.Start();
}
elseif (Servers.Count == 3)
{
    System.Threading.Mutex mtx0 = new System.Threading.Mutex();
    System.Threading.Mutex mtx1 = new System.Threading.Mutex();
    System.Threading.Mutex mtx2 = new System.Threading.Mutex();
    MulThread1 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A22, B11B22, out P1_Parts, mtx0,
Tree));
MulThread1.Start();
    MulThread2 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A22, B11, out P2_Parts, mtx0,
Tree));
MulThread2.Start();
    MulThread3 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11, B12B22, out P3_Parts, mtx0,
Tree));
MulThread3.Start();
    MulThread4 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A22, B21B11, out P4_Parts, mtx1,
Tree));
MulThread4.Start();
    MulThread5 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A11A12, B22, out P5_Parts, mtx1,
Tree));
MulThread5.Start();
}

```



```

        MulThread6 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx2,
Tree));
MulThread6.Start();
        MulThread7 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx2,
Tree));
MulThread7.Start();
    }
elseif (Servers.Count == 4)
    {
        System.Threading.Mutex mtx0 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx1 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx2 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx3 = newSystem.Threading.Mutex();
        MulThread1 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A22, B11B22, out P1_Parts, mtx0,
Tree));
MulThread1.Start();
        MulThread2 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A22, B11, out P2_Parts, mtx0,
Tree));
MulThread2.Start();
        MulThread3 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A11, B12B22, out P3_Parts, mtx1,
Tree));
MulThread3.Start();
        MulThread4 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A22, B21B11, out P4_Parts, mtx1,
Tree));
MulThread4.Start();
        MulThread5 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A11A12, B22, out P5_Parts, mtx2,
Tree));
MulThread5.Start();
        MulThread6 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx2,
Tree));
MulThread6.Start();
        MulThread7 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[3].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx3,
Tree));
MulThread7.Start();
    }
    elseif (Servers.Count == 7)
    {
        System.Threading.Mutex mtx0 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx1 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx2 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx3 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx4 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx5 = newSystem.Threading.Mutex();
        System.Threading.Mutex mtx6 = newSystem.Threading.Mutex();
        MulThread1 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A22, B11B22, out P1_Parts, mtx0,
Tree));
MulThread1.Start();
        MulThread2 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A21A22, B11, out P2_Parts, mtx1,
Tree));
MulThread2.Start();

```

```

        MulThread3 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A11, B12B22, out P3_Parts, mtx2,
Tree));
MulThread3.Start();
        MulThread4 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[3].ToString(), Port, A22, B21B11, out P4_Parts, mtx3,
Tree));
MulThread4.Start();
        MulThread5 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[4].ToString(), Port, A11A12, B22, out P5_Parts, mtx4,
Tree));
MulThread5.Start();
        MulThread6 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[5].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx5,
Tree));
MulThread6.Start();
        MulThread7 = newSystem.Threading.Thread(() =>
Server_Thread(Servers[6].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx6,
Tree));
MulThread7.Start();
    }

    if (MulThread1.IsAlive)
MulThread1.Join();
    if (MulThread2.IsAlive)
MulThread2.Join();
    if (MulThread3.IsAlive)
MulThread3.Join();
    if (MulThread4.IsAlive)
MulThread4.Join();
    if (MulThread5.IsAlive)
MulThread5.Join();
    if (MulThread6.IsAlive)
MulThread6.Join();
    if (MulThread7.IsAlive)
MulThread7.Join();

    if (Servers.Count != 0)
    {
        if (Servers.Count != 1)
        {
            P1 = MatriceUtilities.StrassenConquer(P1_Parts[0],
P1_Parts[1], P1_Parts[2], P1_Parts[3], P1_Parts[4], P1_Parts[5], P1_Parts[6]);
            P2 = MatriceUtilities.StrassenConquer(P2_Parts[0],
P2_Parts[1], P2_Parts[2], P2_Parts[3], P2_Parts[4], P2_Parts[5], P2_Parts[6]);
            P3 = MatriceUtilities.StrassenConquer(P3_Parts[0],
P3_Parts[1], P3_Parts[2], P3_Parts[3], P3_Parts[4], P3_Parts[5], P3_Parts[6]);
            P4 = MatriceUtilities.StrassenConquer(P4_Parts[0],
P4_Parts[1], P4_Parts[2], P4_Parts[3], P4_Parts[4], P4_Parts[5], P4_Parts[6]);
        }
        P5 = MatriceUtilities.StrassenConquer(P5_Parts[0],
P5_Parts[1], P5_Parts[2], P5_Parts[3], P5_Parts[4], P5_Parts[5], P5_Parts[6]);
        P6 = MatriceUtilities.StrassenConquer(P6_Parts[0],
P6_Parts[1], P6_Parts[2], P6_Parts[3], P6_Parts[4], P6_Parts[5], P6_Parts[6]);
        P7 = MatriceUtilities.StrassenConquer(P7_Parts[0],
P7_Parts[1], P7_Parts[2], P7_Parts[3], P7_Parts[4], P7_Parts[5], P7_Parts[6]);
    }

    Status = "Busy";

```

```

        ResultMatrice = MatriceUtilities.StrassenConquer(P1, P2, P3, P4,
P5, P6, P7);
MatriceUtilities.WriteToFile("Result.txt", ResultMatrice);
Console.WriteLine("Result is ready");
        Status = "Idle";
stopWatch.Stop();
        ExecutionTime = stopWatch.Elapsed;
Console.WriteLine("Execution time: " + ExecutionTime.ToString());
    }

staticvoid ExecuteServer()
    {
while (true)
    {
int Port = 10000;
TCPIPsocket socket = newTCPIPsocket("", LocalIP, Port, 0);
byte[] Message = socket.ReceiveMessage();
        System.Threading.Thread Responder =
newSystem.Threading.Thread(() => ResponseToMessage(Message, socket, Command));
Responder.Start();
    }
}

```

This function is performed for server computers. Servers from port 10000 are listening until they receive sent matrices of clients. Calculating the multiplication of these matrices, it presents the results.

```

staticvoid ResponseToMessage(byte[] Message, TCPIPsocket socket, string com)
    {

int Port = 10000;
string IP = socket.RemoteIP;

string ThresholdColumn;
string ThresholdRow;
ConfigurationHandler config = newConfigurationHandler("configuration.cfg");
config.GetValue("THRESHOLD_COLUMN", out ThresholdColumn);
config.GetValue("THRESHOLD_ROW", out ThresholdRow);
int[,] Matrices = MatriceUtilities.FromByteArray(Message);

        file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" +
Matrices[0].GetLength(0) + "," + Matrices[0].GetLength(1) + "\t" +
Matrices[1].GetLength(0) + "," + Matrices[1].GetLength(1) + " were received");
file.Flush();

for (int i = 0; i < Matrices.GetLength(0); i++)
    Matrices[i] = MatriceUtilities.AddZero(Matrices[i]);

int[,] A11 = null, A12 = null, A21 = null, A22 = null;
int[,] B11 = null, B12 = null, B21 = null, B22 = null;

        Status = "Busy";

```

```

MatriceUtilities.StrassenDivide(Matrices[0], out A11, out A12, out A21, out
A22);
MatriceUtilities.StrassenDivide(Matrices[1], out B11, out B12, out B21, out
B22);

//Calculate P1, P2, P3, P4, P5, P6 and P7
int[,] A11A12 = null, A21A11 = null, B11B12 = null, A12A22 = null, B21B22 =
null;
int[,] A11A22 = null, B11B22 = null, A21A22 = null, B12B22 = null, B21B11 =
null;
MatriceUtilities.Add(A11, A22, ref A11A22);
MatriceUtilities.Add(B11, B22, ref B11B22);
MatriceUtilities.Add(A21, A22, ref A21A22);
MatriceUtilities.Subtract(B12, B22, ref B12B22);
MatriceUtilities.Subtract(B21, B11, ref B21B11);
MatriceUtilities.Add(A11, A12, ref A11A12);
MatriceUtilities.Subtract(A21, A11, ref A21A11);
MatriceUtilities.Add(B11, B12, ref B11B12);
MatriceUtilities.Subtract(A12, A22, ref A12A22);
MatriceUtilities.Add(B21, B22, ref B21B22);

int[,] P1 = null, P2 = null, P3 = null, P4 = null, P5 = null, P6 = null, P7 =
null;
if (Matrices[0].GetLength(0) <= Convert.ToInt32(ThresholdRow) &&
Matrices[0].GetLength(1) <= Convert.ToInt32(ThresholdColumn))
{
file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + "on local machine");
file.Flush();
        Status = "Busy";
        System.Threading.Thread MulThread1 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A11A22,
B11B22, ref P1));
MulThread1.Start();
        System.Threading.Thread MulThread2 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A21A22, B11,
ref P2));
MulThread2.Start();
        System.Threading.Thread MulThread3 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A11, B12B22,
ref P3));
MulThread3.Start();
        System.Threading.Thread MulThread4 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A22, B21B11,
ref P4));
MulThread4.Start();
        System.Threading.Thread MulThread5 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A11A12, B22,
ref P5));
MulThread5.Start();
        System.Threading.Thread MulThread6 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A21A11,
B11B12, ref P6));
MulThread6.Start();
        System.Threading.Thread MulThread7 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A12A22,
B21B22, ref P7));
MulThread7.Start();

if (MulThread1.IsAlive)
MulThread1.Join();
if (MulThread2.IsAlive)
MulThread2.Join();

```

```

if (MulThread3.IsAlive)
MulThread3.Join();
if (MulThread4.IsAlive)
MulThread4.Join();
if (MulThread5.IsAlive)
MulThread5.Join();
if (MulThread6.IsAlive)
MulThread6.Join();
if (MulThread7.IsAlive)
MulThread7.Join();
    }
else
    {
int[,] P1_Parts = null, P2_Parts = null, P3_Parts = null, P4_Parts = null,
P5_Parts = null, P6_Parts = null, P7_Parts = null;

string val = "";
config.GetValue("SERVERS_COUNT", out val);
int ServerCount = Convert.ToInt32(val);
int Level = (int)(Math.Log(Matrices[0].GetLength(0), 2) -
Math.Log(Convert.ToInt32(ThresholdRow), 2));
string[] Topology = Tree.Split(';');

config.GetValue("SERVER_INDEX", out val);
string ServerIndex = (Convert.ToInt32(val) - 1).ToString();
ArrayList Servers = newArrayList();
int ChildrensCount = 0;
for (int i = 0; i < Topology.Length; i++)
    {
string[] TopologyParts = Topology[i].Split(';');
if (TopologyParts[1] == ServerIndex)
    {
        ChildrensCount++;
config.GetValue("SERVER" + (Convert.ToInt32(TopologyParts[0]) + 1), out val);
Servers.Add(val);
    }
    }

System.Threading.Thread MulThread1 = null;
System.Threading.Thread MulThread2 = null;
System.Threading.Thread MulThread3 = null;
System.Threading.Thread MulThread4 = null;
System.Threading.Thread MulThread5 = null;
System.Threading.Thread MulThread6 = null;
System.Threading.Thread MulThread7 = null;

if (Servers.Count == 0)
    {
        MulThread1 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11A22, B11B22, ref P1));
MulThread1.Start();
        MulThread2 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A21A22, B11, ref P2));
MulThread2.Start();
        MulThread3 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11, B12B22, ref P3));
MulThread3.Start();
        MulThread4 = newSystem.Threading.Thread(()
=>MatriceUtilities.Multiplication(A22, B21B11, ref P4));
MulThread4.Start();
    }

```

```

        MulThread5 = new System.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11A12, B22, ref P5));
MulThread5.Start();
        MulThread6 = new System.Threading.Thread(()
=>MatriceUtilities.Multiplication(A21A11, B11B12, ref P6));
MulThread6.Start();
        MulThread7 = new System.Threading.Thread(()
=>MatriceUtilities.Multiplication(A12A22, B21B22, ref P7));
MulThread7.Start();
    }
elseif (Servers.Count == 1)
    {
        System.Threading.Mutex mtx0 = new System.Threading.Mutex();
        MulThread1 = new System.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11A22, B11B22, ref P1));
MulThread1.Start();
        MulThread2 = new System.Threading.Thread(()
=>MatriceUtilities.Multiplication(A21A22, B11, ref P2));
MulThread2.Start();
        MulThread3 = new System.Threading.Thread(()
=>MatriceUtilities.Multiplication(A11, B12B22, ref P3));
MulThread3.Start();
        MulThread4 = new System.Threading.Thread(()
=>MatriceUtilities.Multiplication(A22, B21B11, ref P4));
MulThread4.Start();
        MulThread5 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A12, B22, out P5_Parts, mtx0,
Tree));
MulThread5.Start();
        MulThread6 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx0,
Tree));
MulThread6.Start();
        MulThread7 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx0,
Tree));
MulThread7.Start();
    }
elseif (Servers.Count == 2)
    {
        System.Threading.Mutex mtx0 = new System.Threading.Mutex();
        System.Threading.Mutex mtx1 = new System.Threading.Mutex();
        MulThread1 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A22, B11B22, out P1_Parts, mtx0,
Tree));
MulThread1.Start();
        MulThread2 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A22, B11, out P2_Parts, mtx0,
Tree));
MulThread2.Start();
        MulThread3 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11, B12B22, out P3_Parts, mtx0,
Tree));
MulThread3.Start();
        MulThread4 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A22, B21B11, out P4_Parts, mtx0,
Tree));
MulThread4.Start();
        MulThread5 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A11A12, B22, out P5_Parts, mtx1,
Tree));

```

```

MulThread5.Start();
        MulThread6 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx1,
Tree));
MulThread6.Start();
        MulThread7 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx1,
Tree));
MulThread7.Start();
    }
elseif (Servers.Count == 3)
    {
        System.Threading.Mutex mtx0 = new System.Threading.Mutex();
        System.Threading.Mutex mtx1 = new System.Threading.Mutex();
        System.Threading.Mutex mtx2 = new System.Threading.Mutex();
        MulThread1 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A22, B11B22, out P1_Parts, mtx0,
Tree));
MulThread1.Start();
        MulThread2 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A22, B11, out P2_Parts, mtx0,
Tree));
MulThread2.Start();
        MulThread3 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11, B12B22, out P3_Parts, mtx0,
Tree));
MulThread3.Start();
        MulThread4 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A22, B21B11, out P4_Parts, mtx1,
Tree));
MulThread4.Start();
        MulThread5 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A11A12, B22, out P5_Parts, mtx1,
Tree));
MulThread5.Start();
        MulThread6 = new System.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx2,
Tree));
MulThread6.Start();
        MulThread7 = new System.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx2,
Tree));
MulThread7.Start();
    }
elseif (Servers.Count == 4)
    {
        System.Threading.Mutex mtx0 = new System.Threading.Mutex();
        System.Threading.Mutex mtx1 = new System.Threading.Mutex();
        System.Threading.Mutex mtx2 = new System.Threading.Mutex();
        System.Threading.Mutex mtx3 = new System.Threading.Mutex();
        MulThread1 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A11A22, B11B22, out P1_Parts, mtx0,
Tree));
MulThread1.Start();
        MulThread2 = new System.Threading.Thread(() =>
Server_Thread(Servers[0].ToString(), Port, A21A22, B11, out P2_Parts, mtx0,
Tree));
MulThread2.Start();
        MulThread3 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A11, B12B22, out P3_Parts, mtx1,
Tree));
MulThread3.Start();

```

```

        MulThread4 = new System.Threading.Thread(() =>
Server_Thread(Servers[1].ToString(), Port, A22, B21B11, out P4_Parts, mtx1,
Tree));
MulThread4.Start();
        MulThread5 = new System.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A11A12, B22, out P5_Parts, mtx2,
Tree));
MulThread5.Start();
        MulThread6 = new System.Threading.Thread(() =>
Server_Thread(Servers[2].ToString(), Port, A21A11, B11B12, out P6_Parts, mtx2,
Tree));
MulThread6.Start();
        MulThread7 = new System.Threading.Thread(() =>
Server_Thread(Servers[3].ToString(), Port, A12A22, B21B22, out P7_Parts, mtx3,
Tree));
MulThread7.Start();
    }

    Status = "Waiting";

    if (MulThread1.IsAlive)
MulThread1.Join();
    if (MulThread2.IsAlive)
MulThread2.Join();
    if (MulThread3.IsAlive)
MulThread3.Join();
    if (MulThread4.IsAlive)
MulThread4.Join();
    if (MulThread5.IsAlive)
MulThread5.Join();
    if (MulThread6.IsAlive)
MulThread6.Join();
    if (MulThread7.IsAlive)
MulThread7.Join();

        Status = "Busy";
    if (Servers.Count != 0)
    {
        if (Servers.Count != 1)
        {
            P1 = MatriceUtilities.StrassenConquer(P1_Parts[0],
P1_Parts[1], P1_Parts[2], P1_Parts[3], P1_Parts[4], P1_Parts[5], P1_Parts[6]);
            P2 = MatriceUtilities.StrassenConquer(P2_Parts[0],
P2_Parts[1], P2_Parts[2], P2_Parts[3], P2_Parts[4], P2_Parts[5], P2_Parts[6]);
            P3 = MatriceUtilities.StrassenConquer(P3_Parts[0],
P3_Parts[1], P3_Parts[2], P3_Parts[3], P3_Parts[4], P3_Parts[5], P3_Parts[6]);
            P4 = MatriceUtilities.StrassenConquer(P4_Parts[0],
P4_Parts[1], P4_Parts[2], P4_Parts[3], P4_Parts[4], P4_Parts[5], P4_Parts[6]);
        }
        P5 = MatriceUtilities.StrassenConquer(P5_Parts[0],
P5_Parts[1], P5_Parts[2], P5_Parts[3], P5_Parts[4], P5_Parts[5], P5_Parts[6]);
        P6 = MatriceUtilities.StrassenConquer(P6_Parts[0],
P6_Parts[1], P6_Parts[2], P6_Parts[3], P6_Parts[4], P6_Parts[5], P6_Parts[6]);
        P7 = MatriceUtilities.StrassenConquer(P7_Parts[0],
P7_Parts[1], P7_Parts[2], P7_Parts[3], P7_Parts[4], P7_Parts[5], P7_Parts[6]);
    }
}

byte[] Result = MatriceUtilities.ToByteArray(P1, P2, P3, P4, P5, P6, P7);
socket.SendMessage(Result);
    Status = "Idle";
}

```



```

static void Server_Thread(string IP, int Port, int[,] Input1, int[,] Input2,
    out int[,] Ouptuts, System.Threading.Mutex mtx, string Topology)
    {
    if (IP != null)
        {

TCPIPsocket TopologySocket = new TCPIPsocket(IP, LocalIP, 0, 10002);
byte[] Message = Encoding.ASCII.GetBytes(Topology);
mtx.WaitOne();
TopologySocket.SendMessage(Message);
mtx.ReleaseMutex();

TCPIPsocket socket = new TCPIPsocket(IP, LocalIP, 0, Port);
byte[] BytesSend = MatriceUtilities.ToByteArray(Input1, Input2);
mtx.WaitOne();
socket.SendMessage(BytesSend);
mtx.ReleaseMutex();

byte[] BytesReceived = socket.ReceiveMessage();
        Ouptuts = MatriceUtilities.FromByteArray(BytesReceived);
        }
    else
        {

int[,] A11 = null, A12 = null, A21 = null, A22 = null;
int[,] B11 = null, B12 = null, B21 = null, B22 = null;

        Status = "Busy";

MatriceUtilities.StrassenDivide(Input1, out A11, out A12, out A21, out A22);
MatriceUtilities.StrassenDivide(Input2, out B11, out B12, out B21, out B22);

int[,] A11A12 = null, A21A11 = null, B11B12 = null, A12A22 = null, B21B22 =
null;
int[,] A11A22 = null, B11B22 = null, A21A22 = null, B12B22 = null, B21B11 =
null;
MatriceUtilities.Add(A11, A22, ref A11A22);
MatriceUtilities.Add(B11, B22, ref B11B22);
MatriceUtilities.Add(A21, A22, ref A21A22);
MatriceUtilities.Subtract(B12, B22, ref B12B22);
MatriceUtilities.Subtract(B21, B11, ref B21B11);
MatriceUtilities.Add(A11, A12, ref A11A12);
MatriceUtilities.Subtract(A21, A11, ref A21A11);
MatriceUtilities.Add(B11, B12, ref B11B12);
MatriceUtilities.Subtract(A12, A22, ref A12A22);
MatriceUtilities.Add(B21, B22, ref B21B22);

int[,] P1 = null, P2 = null, P3 = null, P4 = null, P5 = null, P6 = null, P7 =
null;
file.WriteLine(DateTime.Now.ToLongTimeString() + "\t" + "on local machine");
file.Flush();

        System.Threading.Thread MulThread1 =
new System.Threading.Thread(() => MatriceUtilities.Multiplication(A11A22,
B11B22, ref P1));
MulThread1.Start();
        System.Threading.Thread MulThread2 =
new System.Threading.Thread(() => MatriceUtilities.Multiplication(A21A22, B11,
ref P2));

```

```
MulThread2.Start();
    System.Threading.Thread MulThread3 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A11, B12B22,
ref P3));
MulThread3.Start();
    System.Threading.Thread MulThread4 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A22, B21B11,
ref P4));
MulThread4.Start();
    System.Threading.Thread MulThread5 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A11A12, B22,
ref P5));
MulThread5.Start();
    System.Threading.Thread MulThread6 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A21A11,
B11B12, ref P6));
MulThread6.Start();
    System.Threading.Thread MulThread7 =
newSystem.Threading.Thread(() =>MatriceUtilities.Multiplication(A12A22,
B21B22, ref P7));
MulThread7.Start();

if (MulThread1.IsAlive)
MulThread1.Join();
if (MulThread2.IsAlive)
MulThread2.Join();
if (MulThread3.IsAlive)
MulThread3.Join();
if (MulThread4.IsAlive)
MulThread4.Join();
if (MulThread5.IsAlive)
MulThread5.Join();
if (MulThread6.IsAlive)
MulThread6.Join();
if (MulThread7.IsAlive)
MulThread7.Join();

    Ouptuts = newint[7][,];
Ouptuts[0] = P1;
Ouptuts[1] = P2;
Ouptuts[2] = P3;
Ouptuts[3] = P4;
Ouptuts[4] = P5;
Ouptuts[5] = P6;
Ouptuts[6] = P7;
    }
}
}
```