Efficient Parallel Finite Difference Time Domain Algorithm for Modeling Electromagnetic Wave Interactions with Dispersive Objects

Ahmad S. Salh

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
January 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 Science in Computer Engineering.

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

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 Science in Computer Engineering.

Assoc. Prof. Dr. Muhammed Salamah
Co-Supervisor

Prof. Dr. Omar Ramadan
Supervisor

Examining Committee

1. Prof. Dr. Erden Başar
2. Prof. Dr. Omar Ramadan
3. Assoc. Prof. Dr. Muhammed Salamah
4. Asst. Prof. Dr. Ahmet Ünveren
5. Asst. Prof. Dr. Gürcü Öz

ABSTRACT

The finite difference time domain (FDTD) method is the most wide spread time domain numerical simulation technique for solving Maxwell equations. The advantages of this method is that it is conceptually simple, and it is simple to be implemented for solving complicated electromagnetic problems. This method, however, is computationally expensive in terms of computational time and memory storage requirement. In this thesis, parallel finite difference time domain (FDTD) algorithm is presented for modeling open region dispersive electromagnetic applications. The algorithm is based on spatial partitioning of the problem geometry into adjacent non-overlapping sub-domains using the two-dimensional topology. The communication among the neighboring processors is carried out by using the message-passing-interface (MPI) library. The performance of the proposed algorithm parallel system, which is composed of (1-16) PCs interconnected through 100Mbps Ethernet, was illustrated for a point source radiating in three-dimensional Lorentz dispersive domain. It has been shown that with eight processors, a speedup factor of 5.6348 is obtained. On the other hand when the problem is distributed among many processors, the speedup decreases. This is because the communication time between neighboring processors becomes comparable to the computation time. Also, it has been found that the algorithm not only speed up the computations but also increases the maximum solvable problem size.

Keywords: Finite Difference Time Domain (FDTD), Message Passing Interface (MPI), Maxwell equations, Electromagnetic Applications.

ÖZ

Zamanda Sonlu Farklar Alanı Yöntemi (FDTD), Maxwell denklemlerinin çözümünde kullanılan en popüler sayısal simulasyon zaman alanı yöntemlerden birisidir. Bu yöntemin avantajları kavramsal olarak basit olması ve karışık elektromagnetik problemlerin çözümünde uygulanmasının kolay ve basit olmasıdır. Fakat, aynı zamanda bu yöntem hesaplama için harcanan zaman ve hafıza depolama koşulları bakımından pahalı bir yöntem olarak bilinmektedir. Bu tezde parallel sonlu farklar zaman alanı algoritmasının (FDTD) açık alanlı dağıtımcı elektromagnetik uygulamaların modellenmesinde kullanımı sunulmaktadır. Adı geçen algoritma iki boyutlu topoloji yöntemi kullanılarak geometrik problemlerin birbiriyle örtüşmeyen, bitişik alt alanlara bölünmesine dayanmaktadır. Bitişik işlemcilerin arasındaki iletişim ise mesaj iletme (MPI) kütüphanesi tarafından sağlanmaktadır. 100Mbps Ethernet aracılığıyle bağlanan (1-16) bilgisayarlardan oluşan algoritmik paralel sistemin performansı, üç boyutlu Lorentz dağıtıcı alanı aracılığıyla yayılan nokta kaynak olarak gösterilmiş ve sekiz adet işlemci aracılığıyle (5.6348) değerine sahip hızlanma faktörü elde edilmiştir. Diğer taraftan problem birçok işlemci arasında dağıtıldığı zaman iletişim zamanının hesaplanma zamanı ile kıyaslanabilir olmasından dolayı hızlanma faktörünün düştüğü ortaya çıkmıştır. Bütün bu bulgulara ek olarak adı geçen algoritma hesaplamayı hızlandırmakta ve aynı zamanda çözülebilecek problem büyüklüğünü de artırmaktadır.

Anahtar kelimeler: Zamanda Sonlu Farklar Alanı Yöntemi (FDTD), Message Passing Interface (MPI),, Maxwell denklemleri, Elektromanyetik Uygulamalar

ACKNOWLEDGMENT

I sincerely acknowledge all the help and support that my supervisor Prof. Dr. Omar Ramadan gave me, his knowledge, guidance, and effort make this research go on and see the light. Also, I am grateful to my Co-supervisor Assoc. Prof. Dr. Muhammed Salamah, who helped me with various issues during this thesis.

My deep gratitude also goes to my mother and father for the support, effort, pain, and patience who I own the success of my life to them. Special thanks to my friends Emad muhamad, Ahmad hani, Huthaifa Luay and Abduallah saad and to all my friends for their help and support.

TABLE OF CONTENTS

ABSTRACT iii

ÖZ iv

ACKNOWLEDGMENT v

LIST OF TABLES viii

LIST OF FIGURES ix

1 INTRODUCTION 1

2 INTRODUCTION TO THE FINITE DIFFERNCE TIME DOMAIN METHOD 4

2.1 Basics of Finite Difference Time Domain Algorithm 4

2.2 YEE's FDTD Algorithm 6

2.3 Absorbing Boundary Conditions (ABC) 8

2.4 FDTD discretization of PML equations 10

3 PARALLELIZING THE FDTD ALGORITHM 13

3.1 Introduction 13

3.2 Domain decomposition 17

4 PARALLEL IMPLEMETATION USING MPI SYSTEM 20

4.1 Introduction 20

4.2 Message passing interface (MPI) 21

4.2.1 MPI functions 22

4.2.2 MPI Initialization 23

4.3 MPICH2 installation and configuration 28

4.3.1 Installation MPI's system 29

4.3.2 MPICH2 configuration 31

4.3.3 Visual studio 2010 configuration 34

4.4 Program description 36

4.4.1 Two Dimensional Cartesian domain topology. 38

5 SIMULATION STUDY 43

5.1 Speedup and efficiency 46

5.2 Scalability 49

6 CONCLUSIONS 52

REFERENCES 53

LIST OF TABLES

Table 4.1: MPI library functions. 22

Table 4.2: Parallel system specification 29

Table 5.1: FDTD parallel system characteristics. 45

Table 5.2: Scalability of the parallel algorithm 49

LIST OF FIGURES

Figure 2.1: Typical unit cell in Yee FDTD algorithm 6

Figure 2.2: 2-D domain surrounded by PML at its boundary. 9

Figure 3.1: Shared memory multi-processor computers 14

Figure 3.2: Distributed memory Model 14

Figure 3.3: The distributed shared memory (DSM) 15

Figure 3.4: Two dimensional (2-D) domain decompostion. 17

Figure 3.5: Communications at the boundaries of a sub-domain for the 2-D.....18

Figure 4.1: Master/Worker example. 29

Figure 4.2: MPI register's window. 31

Figure 4.3: MPICH2 main window.....32

Figure 4.4: MPICH2 main window with more option activated. 33

Figure 4.5: Editor configuration. 34

Figure 4.6: Libraries directories of MPI. 35

Figure 4.7: Additional of library dependencies. 36

Figure 4.8: Structure of the program. 39

Figure 4.9: Flowchart of the parallel FDTD program. 41

Figure 5. 1: Cartesian domain with execution pulse. 43

Figure 5.2: Numerical form of the excitation pulse. 44

Figure 5.3: Total simulation time and communication time. 46

Figure 5.4: Speed-up of the parallel simulation. 48

Figure 5.5: Efficiency of the parallel system. 48

Figure 5.6: Total simulation time and the communication time of scailability. 50

Figure 5.7: Ez-field as recorded in 4-PCs at 118 X 60 X 20. 50

Chapter 1

INTRODUCTION

The finite difference time domain (FDTD) [1] method has become the state-of-the-art method for solving Maxwell's equations in complex geometries. It is based on Yee´s algorithm developed in 1966 [2]. Yee chose a geometric relationship for the spatial sampling of the vector component of the electric and magnetic fields that enables representing both the differential and integral forms of Maxwell's equations in a robust manner. This method is still used frequently for solving many problems in electromagnetic area, because it is simple and efficient method to solve Maxwell's equation in discrete time. The FDTD method calculates the electric field and the magnetic field by discretizing the Maxwell equations in time and space. After discretizing the Maxwell equations, it is easy to obtain the electric fields and the magnetic fields in the computation domain. The computation domain is simply the space where the simulation is performed, and it is divided into unit cells. Each unit cell within the computational domain must be associated with electric and magnetic fields. Then, the material of each cell within the computation domain must be specified, and it can be a free-space, metal, dielectric, or boundary cell. When solving open region problem absorbing boundary conditions (ABCs) are needed to limit the computation domain. The perfectly matched layer (PML) [3] has been shown to be one of the most widely used FDTD ABCs.

To model a large problem by using the FDTD method, huge memory and CPU time are required. By using parallel technique, both the CPU time and memory storage requirements can be decreased. Parallel computing algorithms are based on splitting the computational domain into sub-domains among a network of computers, for instance, PCs and workstation. In each subdomain, the computation of the electric and the magnetic fields on the sub-domain boundary cells require information from the neighboring subdomains, hence each PC need to transmit and receive information with neighboring sub-domains. This transmission is done by using the massage passing interface (MPI) system [4].

MPI system is a standard specification for message passing library, which is used on different platforms, ranging from massively parallel structures to networks of computers, MPI provides a rich range of abilities, and support different program languages like Fortran, assembly language, Pascal, ANSI C, C++, Python, and also support new version of MatLab. And there are other software doing same parallel jobs for instance, PVM (Parallel Virtual Machine) is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. MPICH2 [5], which is used throughout this thesis for carrying out the MPI standard, it is a high-performance and widely porTable implementation of the Message Passing Interface (MPI) standard (both MPI-1 and MPI-2), MPICH2 is distributed as source (with an open-source, freely available license). It has been tested on several platforms, including Linux (on IA32 and x86-64), Mac OS/X (PowerPC and Intel), Solaris (32- and 64-bit), and Windows.

MPI standard supports the growth of parallel application in windows platforms. There are many parallel FDTD algorithms available to solve electromagnetic problems [6], but all of these algorithms are suiTable only for lossless, non-dispersive electromagnetic applications. In [6], although MPI-FDTD parallel approach is presented for frequency depended material, it must be noted that this approach is limited to source-free domains. In this thesis, we extend the parallel FDTD formulation for modeling dispersive as well as non-dispersive problems. We have simulated three-dimensional (3-D) cartesian domain entirely composed of Lorentz material [7], and it has been shown that the parallel algorithm can speed up the computation, and it is able to solve bigger problems size. It has been shown that with eight processors, a speedup factor of (5.6348) is obtained. On the other hand, when the program is distributed among many processors, the speedup decreases, because the communication times become comparable to the computation time.

The thesis is organized as follow. A shortcut review of conventional FDTD algorithm is given in Chapter 2. Chapter 3 deals with parallelizing the FDTD algorithm. In Chapter 4, parallel implementations of FDTD using MPI is described. Chapter 5 gives the simulation result, and finally conclusions and future work are given in Chapter 6.

Chapter 2

INTRODUCTION TO THE FINITE DIFFERNCE TIME DOMAIN METHOD

2.1 Basics of Finite Difference Time Domain Algorithm

Considering a three dimensional (3-D) dispersive domain, the frequency domain of Maxwell's curl equations can be written as:

$$j\omega\ \bar{H} = -\,c_0\,\nabla\times\bar{E} \qquad 2.1$$

$$j\omega\ \varepsilon_r(\omega)\bar{E} = c_0\,\nabla\times\bar{H} \qquad 2.2$$

where $j\omega$ is for frequency domain variable, $c_0 = 1/\sqrt{\varepsilon_0\mu_0\varepsilon_\infty}$, $\varepsilon_\infty = \varepsilon_r(\infty)$, is the speed of light, $\bar{E}$ is the electrical field vector, $\bar{H}$ is the magnetic field vector, ε_0 is the electric permittivity, μ_0 is the magnetic permeability, and $\varepsilon_r(\omega)$ is the relative permittivity of the domain, which can be written for Lorentz dispersive material, for example, as:

$$\varepsilon_r(\omega) = 1 + \frac{\Delta\varepsilon\ \omega_0^2}{\omega_0^2 + j2\delta\omega - \omega^2} \qquad 2.3$$

where $\Delta\varepsilon = {}^{\varepsilon_s}/_{\varepsilon_\infty} - 1$, with $\varepsilon_s = \varepsilon_r(0)$, ω_0 is the resonance radial frequency, and δ is the damping constant. In a rectangular coordinator system, equations (2.1) and (2.2) are decomposed into six scalar equations. The FDTD method solve these scalar equations in the time domain by applying the central difference approximation to the

time and space derivatives according to Yee´s algorithm [2]. Based on equations (2.1) and (2.2) the following six scalar field equations can be obtained as [8]:

$$\frac{\partial H_x}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}} \left(\frac{\partial E_y}{\partial z} - \frac{\partial H_z}{\partial y} \right) \quad 2.4$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}} \left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} \right) \quad 2.5$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}} \left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} \right) \quad 2.6$$

and

$$\frac{\partial D_x}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}} \left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} \right) \quad 2.7$$

$$\frac{\partial D_y}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}} \left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} \right) \quad 2.8$$

$$\frac{\partial D_z}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}} \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} \right) \quad 2.9$$

where D_x, D_y and D_z fields are related to E_x, E_y, and E_z respectively as:

$$D_x(\omega) = \varepsilon_r(\omega).E_x \quad 2.10$$

$$D_y(\omega) = \varepsilon_r(\omega).E_y \quad 2.11$$

$$D_z(\omega) = \varepsilon_r(\omega).E_z \quad 2.12$$

Equations (2.4) – (2.12) from the basic algorithm for modeling electromagnetic wave interaction with arbitrary three-dimensional objects.

2.2 YEE's FDTD Algorithm

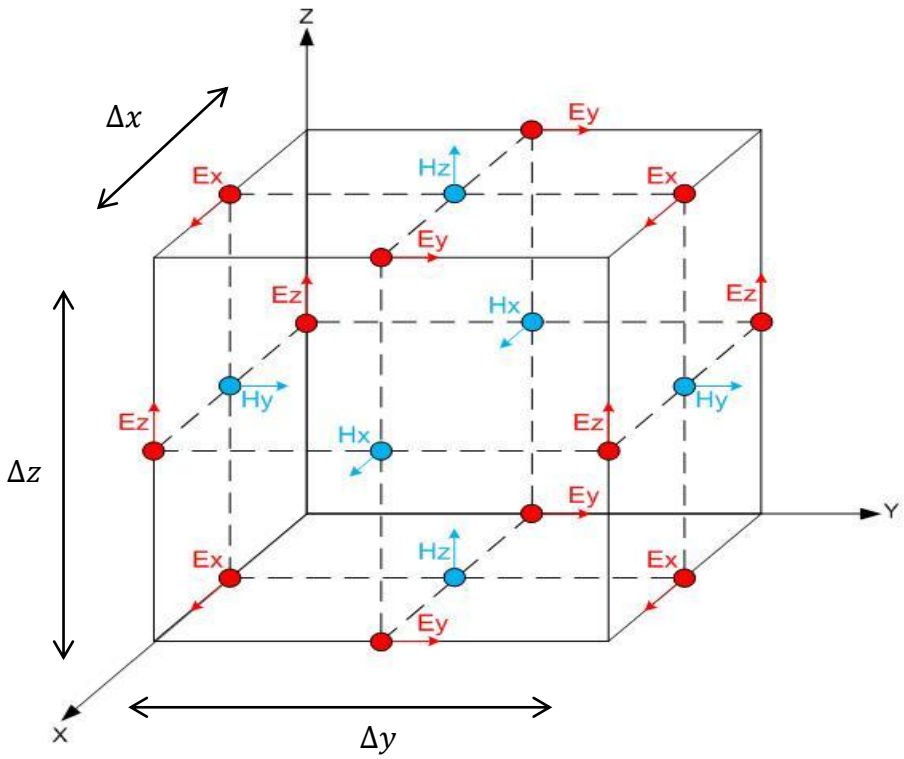

Figure 2.1: Typical unit cell in Yee FDTD algorithm [9].

The basics FDTD algorithm was introduced by Yee in 1966 [2]. The first step of this algorithm is based on dividing the domain into rectangular cells with dimension of $(\Delta x, \Delta y, \Delta z)$, where $\Delta x, \Delta y$ and Δz, are respectly the space cell size in the x, y and z directions. Figure 2.1 shows a typical Yee´s FDTD unit cell. Every unit cell associated with six field's component: $Ex, Ey\ and\ Ez$ for electric fields and $Hx, Hy\ and\ Hz$ for magnetic fields. For each cell, all of H field's components are located at faces of the cell and all of E field's components are located at the edges of the cell [2].

The steps of Yee´s algorithms are summarized as bellow [10]:

1- Replace all space and time derivatives with their finite differences so that the electric and magnetic fields are staggered in both space and time.

2- Solve the resulting difference equations to obtain "the update equations" that express the $un-known$ future fields in terms of the $known$ past fields.

3- Evaluate the magnetic fields one time-step into the future so they are known (effectively they become past fields).

4- Evaluate the electric fields one time-step into the future so they are now known (effectively they become past fields).

5- Repeat the previous steps until the fields have been obtained over the desired duration.

Based on the above algorithm, $E_z \; and \; H_y$ fields, for example can be written in FDTD by using (2.5), (2.9) and (2.12), as follow:

$$H_{y_{i+1/2,j,k+1/2}}^{n+1/2} = H_{y_{i+1/2,j,k+1/2}}^{n-1/2} - \frac{c\Delta t}{\Delta x}\begin{bmatrix} E_{x_{i+1/2,j,k+1}}^{n} - E_{x_{i+1/2,j,k}}^{n} \\ - E_{z_{i+1,j,k+1/2}}^{n} + E_{z_{i,j,k+1/2}}^{n} \end{bmatrix} \quad 2.13$$

$$D_{z_{i,j,k+1/2}}^{n+1} = D_{z_{i,j,k+1/2}}^{n} + \frac{c\Delta t}{\Delta x}\begin{bmatrix} H_{y_{i+1/2,j,k+1/2}}^{n+1/2} - H_{y_{i-1/2,j,k+1/2}}^{n+1/2} \\ - H_{x_{i,j+1/2,k+1/2}}^{n+1/2} + H_{x_{i,j-1/2,k+1/2}}^{n+1/2} \end{bmatrix} \quad 2.14$$

$$E_{z_{i,j,k+1/2}}^{n+1} = \frac{g0}{d0} E_{z_{i,j,k+1/2}}^{n+1} + \frac{1}{d0}\begin{bmatrix} g_1 D_{z_{i,j,k+1/2}}^{n} + g_2 D_{z_{i,j,k+1/2}}^{n-1} \\ -d_1 E_{z_{i,j,k+1/2}}^{n} - d_2 E_{z_{i,j,k+1/2}}^{n-1} \end{bmatrix} \quad 2.15$$

where (i, j, k) is cell´s position, (n) is time step, g_0, g_1, g_2, d_0, d_1 and d_2 are given by.

$$g_0 = 1 + \Gamma\frac{\Delta t}{2} + \omega_0^2\frac{\Delta t^2}{4} \quad \text{2.15a}$$

$$g_1 = -2 + 2\,\omega_0^2\frac{\Delta t^2}{4} \quad \text{2.15b}$$

$$g_2 = 1 - \Gamma\,\frac{\Delta t}{2} + \omega_0^2\frac{\Delta t^2}{4} \quad \text{2.15c}$$

and

$$d_0 = 1 + \Gamma\,\frac{\Delta t}{2} + \omega_0^2 + \Delta_\varepsilon\omega_0^2\,\frac{\Delta t^2}{4} \quad \text{2.15d}$$

$$d_1 = -2 + 2\,\omega_0^2 + \Delta_\varepsilon\omega_0^2\,\frac{\Delta t^2}{4} \quad \text{2.15e}$$

$$d_2 = 1 - \Gamma\,\frac{\Delta t}{2} + \omega_0^2 + \Delta_\varepsilon\omega_0^2\,\frac{\Delta t^2}{4} \quad \text{2.15f}$$

2.3 Absorbing Boundary Conditions (ABC)

Recently many of the FDTD applications are considered to be unbounded ones. Since the limitation in computer storage abilities, the computation domain must be finite. For these application of FDTD, is required to truncate the domain by introducing artificial outer boundaries. The boundaries need to be designed to absorb outgoing waves without reflection into the simulated domain. In recent years, an efficient absorbing boundary condition (ABC) referred as the perfectly matched layer (PML) [3] [6], introduced by the Berenger [3] has been widely used. This kind of ABC is used to surround the computational boundary with appropriate values of electrical and magnetical conductivities. The PML region at the domain boundaries include electric and magnetic

conductivity that gradually increase in the PML region the absorb outgoing electromagnetic waves.

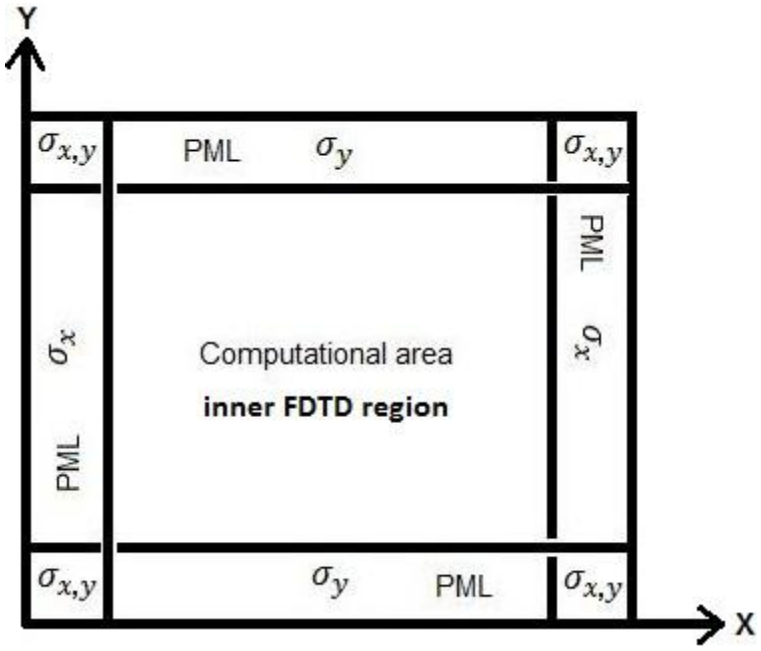

Figure 2.2: 2-D domain surrounded by PML at its boundary.

Anisotropic PML (APML) [11] is the collective formulation of Berenger PML. In this context, Maxwell´s equation (2.1) and (2.2) can be applied in PML region with addition of some mathematical expressions that express the specifications of medium as follow:

$$-j\omega\,\bar{\mu}_r(\omega)_{PML}\,\bar{H} = c_0\,\nabla\times\bar{E} \qquad 2.16$$

$$j\omega\,\varepsilon_r(\omega)\,\bar{\varepsilon}_r(\omega)_{PML}\,\bar{E} = c_0\,\nabla\times\bar{H} \qquad 2.17$$

where $j\omega$ is for frequency domain variable, $\bar{\varepsilon}_r(\omega)_{PML}$ is the APML permittivity, $\bar{\mu}_r(\omega)_{PML}$ is the APML permeability defined as [11] [12]:

$$\bar{\bar{\varepsilon}}_r(\omega) = \bar{\mu}_r(\omega) = \begin{bmatrix} \frac{s_y s_z}{s_x} & & \\ & \frac{s_x s_z}{s_y} & \\ & & \frac{s_x s_y}{s_z} \end{bmatrix} \quad 2.18$$

with s_η $(\eta = x, y, or\ z)$ are given by

$$s_\eta = 1 + \frac{\sigma_\eta}{j\omega\varepsilon_0} \quad 2.19$$

where σ_η is the APML conductivity profile along the η – coordinate designed to absorb the outgoing waves with minimal reflections defined as:

$$\sigma_\eta = \sigma_m \left(\frac{\eta - \eta_0}{\delta}\right)^m \quad 2.20$$

where σ_m is the maximum conductivity, δ is the PML conductivities , η_0 is the PML / Computational domain interface, and m is the order of the polynomial. The benefit of PML is attenuating the electromagnetic wave without reflection. It's important to note that (2.16) and (2.17) can also be applied in the inner domain by setting the APML conductivity to zero, i.e., $\sigma_\eta = 0$. Figure 2.2 shows inner FDTD region and the PML region.

2.4 FDTD discretization of PML equations

Using equations (2.16) and (2.17), the E_z-field component, as an example, can be written as:

$$j\omega\ \varepsilon_r(\omega)\ \frac{\left(1+\frac{\sigma_y}{j\omega\ \varepsilon_0}\right)\left(1+\frac{\sigma_x}{j\omega\ \varepsilon_0}\right)}{\left(1+\frac{\sigma_z}{j\omega\ \varepsilon_0}\right)}\ E_z = c_0\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) \quad 2.21$$

Equation (2.21) can be rearranged as:

$$j\omega \left(1 + \frac{\sigma_x}{j\omega\, \varepsilon_0}\right) G_z = c_0 \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) \quad 2.22$$

where G_z is given by

$$G_z = \frac{\left(1+\frac{\sigma y}{j\omega\, \varepsilon_0}\right)}{\left(1+\frac{\sigma_z}{j\omega\, \varepsilon_0}\right)} D_z \quad 2.23$$

and D_z is related to E_z through

$$D_z = \varepsilon_r(\omega)\, E_z \quad 2.24$$

Using the invers Fourier transforms relation, jω ⇒ $\partial/\partial t$, (2.22), and (2.23) can be written in the time domain as

$$\frac{\partial G_z}{\partial t} + \frac{\sigma_x}{\varepsilon_0} G_z = c_0 \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) \quad 2.25$$

and

$$\frac{\partial G_z}{\partial t} + \frac{\sigma_z}{\varepsilon_0} G_z = \frac{\partial D_z}{\partial t} + \frac{\sigma_y}{\varepsilon_0} D_z \quad 2.26$$

Using the FDTD algorithm, (2.25) and (2.26) can be written in the discrete time domain as

$$G^{n+1}_{z_i,j,k+1/2} = \frac{\alpha^-_{x_i}}{\alpha^+_{x_i}} G^n_{z_i,j,k+1/2} + \frac{c_0\Delta_t}{\Delta} \frac{1}{\alpha^+_{x_i}} \begin{bmatrix} H^{n+1/2}_{y_i+1/2,j,k+1/2} - H^{n+1/2}_{y_i+1/2,j,k+1/2} \\ - H^{n+1/2}_{x_i,j+1/2,k+1/2} + H^{n+1/2}_{x_i,j+1/2,k+1/2} \end{bmatrix} \quad 2.27$$

$$D_{z_i,j,k+1/2}^{n+1} = \frac{\alpha_{y_j}^-}{\alpha_{y_j}^+} D_{z_i,j,k+1/2}^{n} + \frac{\alpha_{z_{k+1/2}}^-}{\alpha_{y_j}^+} \left[G_{z_i,j,k+1/2}^{n+1} - \frac{\alpha_{z_{k+1/2}}^-}{\alpha_{z_{k+1/2}}^+} G_{z_i,j,k+1/2}^{n} \right] \quad 2.28$$

where $\alpha_{\eta_m}^{\pm}$, (for, $\eta = x, y, or\ z$), is given by

$$\alpha_{\eta_m}^{\pm} = 1 \pm \Delta_t \, \sigma_{\eta_m} / 2\varepsilon_0 \quad 2.29$$

Finally, to compute the E_z from D_z, it is required to discretize (2.24) by the methodology used in (2.15). To apply the above equation in the inner region, it is required to eliminate the APML conductivity, i.e.

$$\sigma_x = \sigma_y = \sigma_z = 0 \quad 2.30$$

Chapter 3

PARALLELIZING THE FDTD ALGORITHM

3.1 Introduction

The idea behind of parallel processing is to divide the whole problem into sub-problems that can be computed concurrently. Nowadays, some different architectures, that can provide parallelism have been introduced., A multi-processor computer is an example that has a number of processors. The classification of multi-processor computers are categorized as [13] :

- Shared memory
- Distributed memory
- Distributed shared memory

In shared memory, the memory module and the processor are connected by mean of an interconnection network as shown in Figure 3.1. This means that all processors shares the primary memory, but each processor has its own cash memory [14].

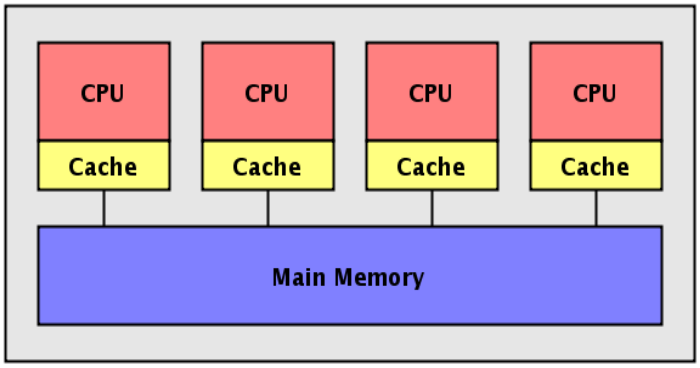

Figure 3.1: Shared memory multi-processor computers [15].

In distributed memory, there is an interconnection network but the difference is that each processor has its own private (main) memory interconnection and support message passing rather than memory reading and writing as shown in Figure 3.2 [14].

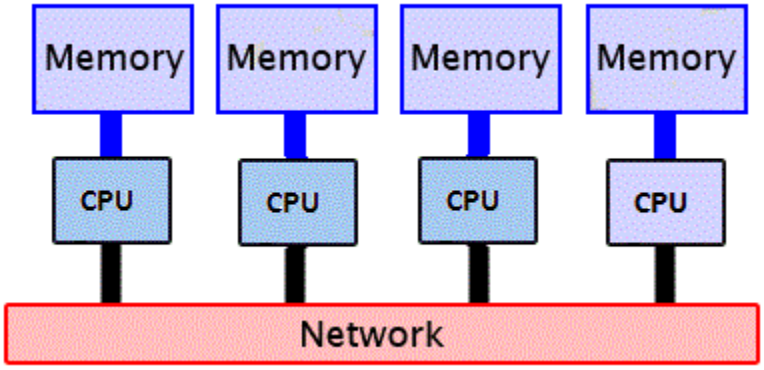

Figure 3.2: Distributed memory Model [16].

This class can be categorized into two categories

- Single instruction multiple data (SIMD).
- Multiple instruction multiple data (MIMD).

SIMD computers are typically an array of processing elements, all connected to a common control host processor by way of one or more processors in the array. MIMD

computers are a group of processors executing one or more operating systems, coordinating or synchronizing their operation, and exchanging data and controlled by mean of message passing [17]. Finally, the distributed shared memory (DSM) implements the shared memory model in distributed systems, which have no physical shared memory. The shared memory model provides a virtual address space shared between all Processors as shown in Figure 3.3.

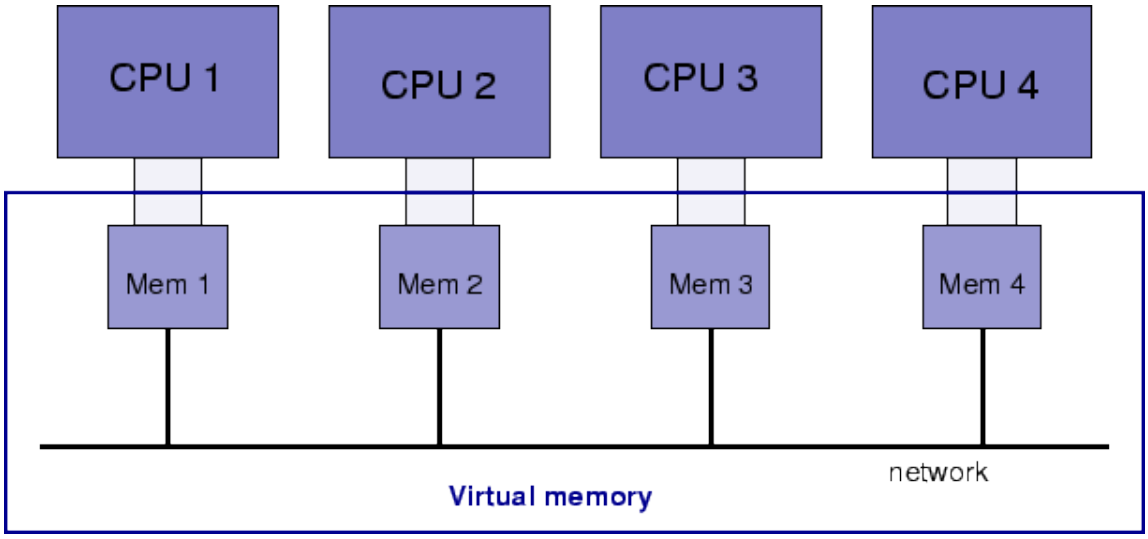

Figure 3.3: The distributed shared memory (DSM) [18].

Parallel processing gives a good solution to reduce computation time for problems that require huge processing time to run on high-performance workstation. Although the parallel processing reduces computation time, which is required to solve the problem in sequential mode, the parallel processing required extra task in terms of parallel computation, which is not required in serial processing mode. These additional tasks which will increase the operation times of parallelism are.

- Processing Idle time
- Synchronization
- Inter-processor communication overhead

Process idle time depending on how much service and application are running in the PC and CPU time reserved for each service or application. Synchronization, it is pointing to synchronizing for one process, which is running on a multi-processor. And finally, inter-processor communication overhead takes additional CPU time, when a remote processor needs to send and receive data.

Assume that the time require to solve a given problem using serial algorithm on a single processor is $T_{(1)}$, and denoting $T_{(p)}$ as the time needed to solve the same problem on P processors using a parallel algorithm, we define the speedup as

$$Speedup = \frac{T_{(1)}}{T_{(p)}} \qquad 3.1$$

It is important to note that the computation time of the parallel part will be decreased, when the number of processors is increased, but the communication time will be increased, and the synchronization's time will be raised up, which limiting speedup's factor [13].

3.2 Domain decomposition

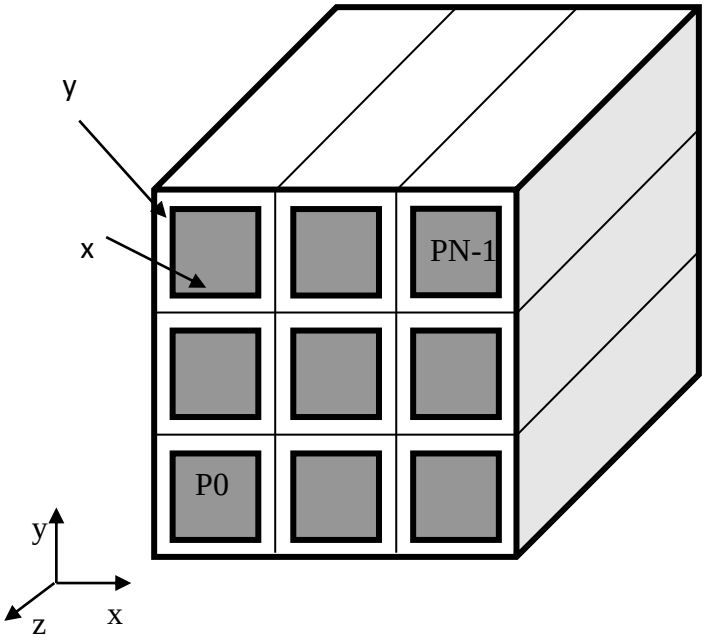

Figure 3.4: Two dimensional (2-D) domain decompostion.

A regularly explored option to exploit parallelism is to split up the domain into subdomains that can be worked in parallel by multiple processors. The advantage of this approach is that if the subdomains need to share data, they can do so with short messages. The domain can be decomposed using one-dimensional (1-D) or two-dimensional (2-D) topology. In this thesis, 2-D topology, as shown in Figure 3.4 used as it is found to be more efficient than the 1-D topology, especially for large number of processors. As shown in Figure 3.4.

3.3 FDTD method in parallel

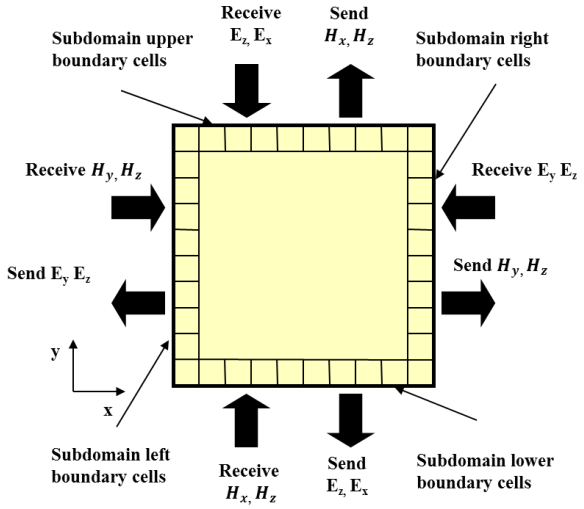

Figure 3.5: Communications at the boundaries of a sub-domain for the 2-D topology.

Using the 2-D topology, the computational domain is divided into subdomains along the x and y directions, where each subdomain is assigned to one processor, as shown in Figure 3.5. To update the field components at the sub-domain boundaries, data from the neighboring sub-domains are needed. The inter-processor communication among the neighboring processors is carried out by using the MPI library. Figure 3.5 shows the data need to be exchanged between neighboring sub-domains. For the communication purpose, ghost layers located at the edges of the sub-domains are used. The MPI system is used to exchange data between processors. To calculate E_z at the cells located at the left boundary of a subdomain, the values of H_y from the subdomain on its left are needed. Also, this subdomain must send the values of H_y at cells located at the right boundary to the subdomain on its right. Similarly, to calculate E_z at the cells located at the lower boundary of a subdomain, the values of H_x from the lower subdomain are are

needed. Also, this subdomain must send the values of H_x at cells located at its upper boundary to the upper subdomain. To calculate H_y at the cells located at the right boundary of the subdomain, the values of E_z from the right subdomain are needed. This subdomain should also send the values of E_z at the cells located at the left boundary to the left subdomain. Similarly, to calculate H_y at the cells located at the upper boundary of the subdomain, the values of E_x from the upper subdomain are needed. Also, this subdomain should send the values of E_x at the cells located at its lower boundary to the lower subdomain. Finally, the steps for the proposed parallel algorithm can be summarized as:

1. MPI initialization: Initialize the MPI execution environment.
2. Reading of simulation parameters.
3. Creation of the 2-D topology
4. At each time step, perform the following:

   4.1 Update the E-fields (electric fields) and other auxiliary variables in each sub-domain.

   4.2 Exchange E-fields (electric fields) with the neighbor subdomains by using the MPI library functions.

   4.3 Update the H-fields (magnetic fields) in each sub-domain.

   4.4 Exchange H-fields (magnetic fields) with the neighbor sub-domains by using the MPI library functions.

5. MPI finalization.

Chapter 4

PARALLEL IMPLEMETATION USING MPI SYSTEM

4.1 Introduction

In this thesis, the parallel implementation of the FDTD algorithm on a network of PCs has been examined. The regularity of the computational grid makes the decomposition into a networked PCs relatively straightforward. Each PC is allocated a block of the computational grid and handles the calculations of the E and H fields of the cells in that block. When cells on a block boundary need data from their neighbors in adjacent blokes in order to update the E and H fields in each iteration, these data are transferred between the PCs over the local area network (LAN) connection.

A SIMD type problem, like FDTD algorithm, on a system of networked PCs can be implemented by using a Single Program Multiple Data (SPMD) computing model. The SPMD model gives each PC a copy of the same program and requires each PC to communicate with its neighboring PCs. In the parallel implementation of the FDTD algorithm, each PC runs the same parallel code and requires closest neighbor communication between cells [13].

To provide communication between PCs over the LAN, MPI is used as a message passing system. It provides a flexibility to design and implement a parallel application based on a distributed memory model. MPICH2 [5] version 1.4.1p1 is a sTable

release used for the implementation of the parallel code. The code is written in C language using the Microsoft Visual C++ 2010 as the application package.

The parallel code is organized in a master and worker's PCs (server-client). There is just one master PC and one or more worker's PCs. The master is responsible for both the data Input/Output, and the update function, where the workers are responsible for the calculations of the E and H fields only. Both master PC and workers PCs calculate their private data then master receives all the results from the worker's PCs.

4.2 Message passing interface (MPI)

The Message Passing Interface (MPI) is a standardized and porTable message passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers [4]. The standard defines the syntax and semantics of a core of library routines useful for a wide range of users who write porTable message-passing programs in FORTRAN or C [4]. MPI has gained wide acceptance in the parallel computing community and it is available on a wide variety of platforms, ranging from massively parallel systems to network of computers, or workstations.

In the MPI programming model, a computation comprises one or more processes, which are grouped inside a communicator. The communicator defines the communication context. A process has a local memory and an execution unit. Thus, one process cannot directly access variables in another process's memory. Because of this reason, processes communicate by calling MPI library functions in order to send and receive messages between each other [12].

4.2.1 MPI functions

The required communication between processors is very simply to be handled using MPI. In this thesis, communication was implemented using the MPICH2 library integrated into C source code. MPI provides around 200 functions, and it covers a wide range of parallel programs solutions in different fields. In this thesis, eleven functions were were used. These eleven functions are given in Table 4.1 [18].

Table 4.1: MPI library functions.

1	MPI_Init	Initialize the MPI execution environment
2	MPI_Comm_rank	Determines the rank of the calling process in the communicator
3	MPI_Comm_size	Determines the size of the group associated with a communicator
4	MPI_Cart_create	Makes a new communicator to which topology information has been attached
5	MPI_Cart_coords	Determines process coords in Cartesian topology given rank in group
6	MPI_Cart_shift	Returns the shifted source and destination ranks, given a shift direction and amount
7	MPI_Comm_free	Marks the communicator object for de-allocation
8	MPI_Wtime	Returns an elapsed time on the calling processor
9	MPI_Send	Performs a blocking send
10	MPI_Recv	Blocking receive for a message
11	MPI_Finalize	Terminates MPI execution environment

4.2.2 MPI Initialization

Before the execution of the parallel programs, it is needed to indicate the number of processes to be used from the operating-system command line. Each parallel program is required to determine the number of processes used and the identifier, or rank, of each process. At the beginning of each parallel program, the first MPI instructions concern providing of those data which are obtained by the following functions [12].

- MPI_Init(int *argc, char ***argv)

Initialize the MPI execution environment, with the parameter of

argc: [input] Pointer to the number of arguments

argv: [input] Pointer to the argument vector

MPI_Init, is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment. Calling MPI_Init more than once during the execution of a program will lead to an error. This routine must be called before any other MPI routine. The MPI standard does not say what a program can do before an MPI_Init or after an MPI_Finalize [19].

- MPI_Comm_rank (MPI_Comm comm,int *rank)

Determines the rank of the calling process in the communicator, with the parameter of

comm: [input] communicator (handle)

rank: [output] rank of the calling process in the group of communicator (integer)

MPI_COMM_RANK indicates the rank of the process that calls it in the range from (zero to (size-1)), where size is the return value of MPI_Comm_size [19].

- MPI_Comm_size(MPI_Comm comm, int *size)

Determines the size of the group associated with a communicator, with the parameter of

comm: [input] communicator (handle)

size: [output] number of processes in the group of communicator (integer)

This function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available [19].

- MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

Makes a new communicator to which topology information has been attached, with the parameter of

comm_old: [input] input communicator (handle)

ndims: [input] number of dimensions of cartesian grid (integer)

dims: [input] integer array of size ndims specifying the number of processes in each dimension

periods: [input] logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension

reorder: [input] ranking may be reordered (true) or not (false) (logical)

comm_cart: [output] communicator with new Cartesian topology (handle)

MPI_CART_CREATE, returns a handle to a new communicator to which the Cartesian topology information is attached If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes If the total size of the Cartesian grid is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL [19].

- MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *cords)

Determines process coordinators in cartesian topology given rank in group, with the parameter of

comm: [input] communicator with cartesian structure (handle)

rank: [input] rank of a process within group of comm (integer)

maxdims: [input] length of vector coords in the calling program (integer)

coords:[output] integer array (of size ndims) containing the Cartesian coordinates of specified process (integer)

The inverse mapping, rank-to-coordinates translation is provided by MPI_Cart_coords [19].

- MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source, int *dest)

Returns the shifted source and destination ranks, given a shift direction and amount, with the parameter of

comm: [input] communicator with cartesian structure (handle)

direction: [input] coordinate dimension of shift (integer)

displ: [input] displacement (> 0: upwards shift, < 0: downwards shift) (integer)

source: [output] rank of source process (integer)

dest: [output] rank of destination process (integer) [19].

- MPI_Comm_free(MPI_Comm *comm)

Marks the communicator object for de-allocation, with the parameter of

comm: [input] Communicator to be destroyed (handle)

This routine frees a communicator, because the communicator may still be in use by other MPI routines [19].

- MPI_Wtime(void)

Returns an elapsed time on the calling processor, returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past [19].

- MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Performs a blocking send, with the parameter of

buf: [input] initial address of send buffer (choice)

count: [input] number of elements in send buffer (nonnegative integer)

datatype: [input] datatype of each send buffer element (handle)

dest: [input] rank of destination (integer)

tag: [input] message tag (integer)

comm: [input] communicator (handle)

This routine may block until the message is received by the destination process [19].

- MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,

MPI_Comm comm, MPI_Status *status)

Blocking receive for a message, with the parameter of

buf: [output] initial address of receive buffer (choice)

count: [input] maximum number of elements in receive buffer (integer)

datatype: [input] datatype of each receive buffer element (handle)

source: [input] rank of source (integer)

tag: [input] message tag (integer)

comm: [input] communicator (handle)

status: [output] status object (Status)

The receive buffer consists of the storage containing count consecutive elements of the type specified by data-type, starting at address buf [19].

- MPI_Finalize(void)

Terminates MPI execution environment

This routine cleans up all MPI state. Once this routine is called, no MPI routine (even MPI_INIT) may be called [19].

4.3 MPICH2 installation and configuration

MPICH2 is a high-performance and widely porTable implementation of the Message Passing Interface (MPI) standard (both MPI-1 and MPI-2) [5], as the goals of MPICH team are:

1: to provide an MPI implementation that efficiently supports different computation and communication platforms including commodity clusters (desktop systems, shared-memory systems, multicore architectures), high-speed networks (10 Gigabit Ethernet, InfiniBand, Myrinet, Quadrics) and proprietary high-end computing systems (Blue Gene, Cray).

2: to enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations.

MPICH is distributed as source (with an open-source, freely available license). It has been tested on several platforms, including Linux (on IA32 and x86-64), Mac OS/X (PowerPC and Intel), Solaris (32- and 64-bit), and Windows.

The MPICH2 library aimed to implement all the functionality specified by the MPI standard in an efficient and porTable fashion. Due to its characteristics, MPICH served as a development base for many other implementations, which addressed different operating systems and architectures [13].

4.3.1 Installation MPI's system

In this thesis we had download of MPICH2 version 1.4.1p1. It is sTable release version of MPICH2 for windows platforms X86 systems. For installing and preparing, computers used in this thesis have the following specification shown in Table 4.2, and the example of the system´s network with three workers and one master, is shown in Figure 4.1.

Table 4.2: Parallel system specification

CPU: Core 2 due @ 3.0 GHz
RAM: 2 Gigabytes
OS: Windows XP Professional X86 service pack 2

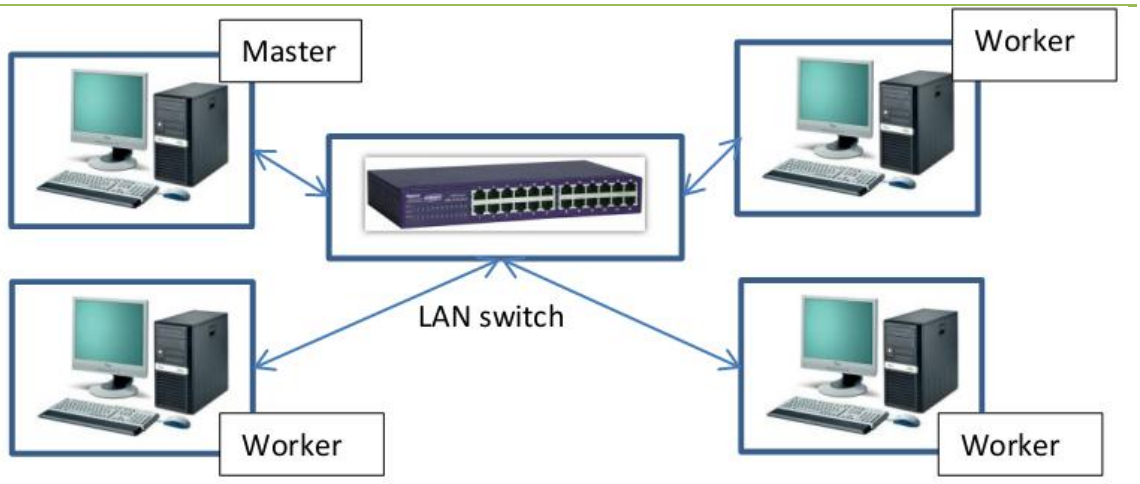

Figure 4.1: Master/Worker example.

To guarantee install without problems, the following steps are came out. We need first, to make new user account with administrator privileges on all PCs with secure password. All PCs should have the same user account name and password.

Second, login in the new account, double click on the execution file (mpich2-1.4.1p1-win-ia32.exe), the wizard will begin, follow the wizard and accept license agreement, and don’t change the installation directory, then click "Close" bottom to finish wizard. Before run the program, the following steps should be done.

- Press Start bottom; click on run type "cmd" and click Enter. Command window will open
- Type in the command window "cd C:\Program Files\MPICH2\bin" then hit Enter,
- Type " smpd.exe -install" and hit Enter.
- Type "smpd.exe –sethosts <host name>", you can take computer name from these steps, after doing it click Enter:
  - ❖ Right click on My computer
  - ❖ Select Properties
  - ❖ Select Computer Name tab, copy full computer name and exchange it with <host name>
- Finally, type "smpd.exe -restart" and click Enter and close command window.

Now, we need to enter account information such as user name and password. Press start bottom, click programs, MPICH2, and then click on "wmpiregister" then Figure 4.2 will appear

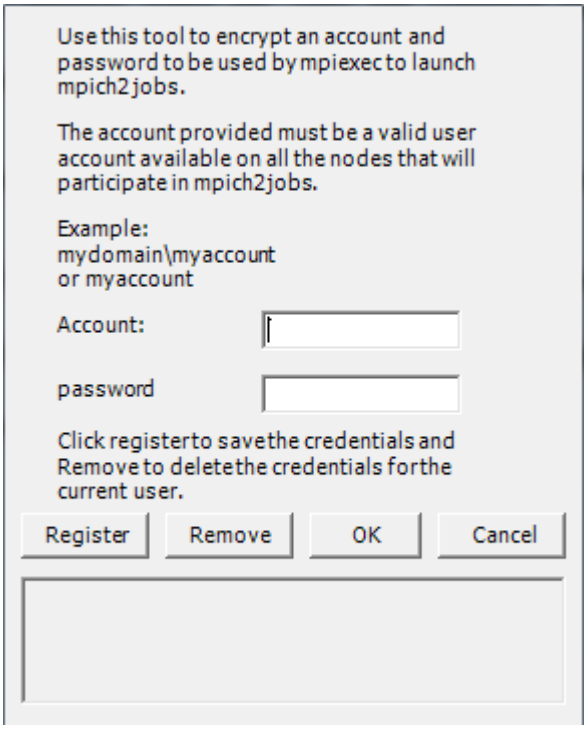

Figure 4.2: MPI register's window.

Enter user name in account field and enter password in password field. Then click Register then click ok. After these steps, the installation of the system finished.

4.3.2 MPICH2 configuration

MPICH2 provide GUI utility, which makes the process work easier than using command window. After finish of install MPICH2 on all PCs, a program called "wmpiexec.exe" appears under Programs in the Start menu list, click on MPICH2, finally, click on wmpiexec.exe, Figure 4.3 showing main window of wmpiexec to start a parallel job under Windows

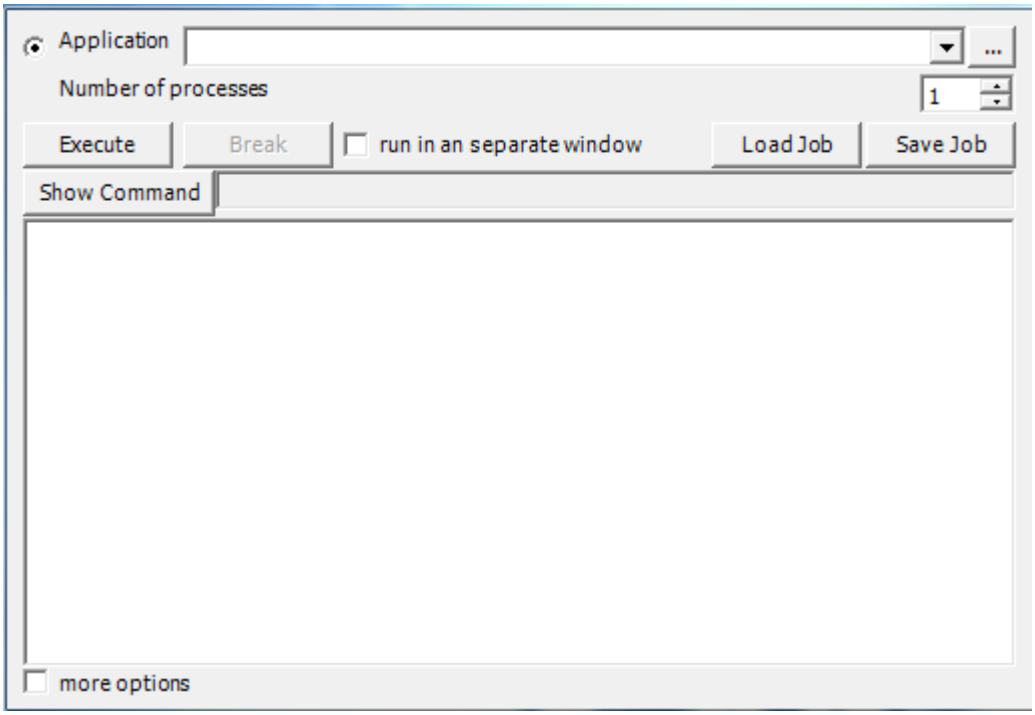

Figure 4.3: MPICH2 main window.

After building the project with Visual C++ 2010, the execution file should be distributed among all PCs in the system with same path of execution file in master PC. Then at the application field click on brows bottom and specify the execution file project; "Number of processes" is for specifying the number of PCs that will do the parallel project, because each PC will take one process, and we should specify the address of all PCs in parallel system to make communication work, checkbox on "more option" as shown in Figure 4.4.

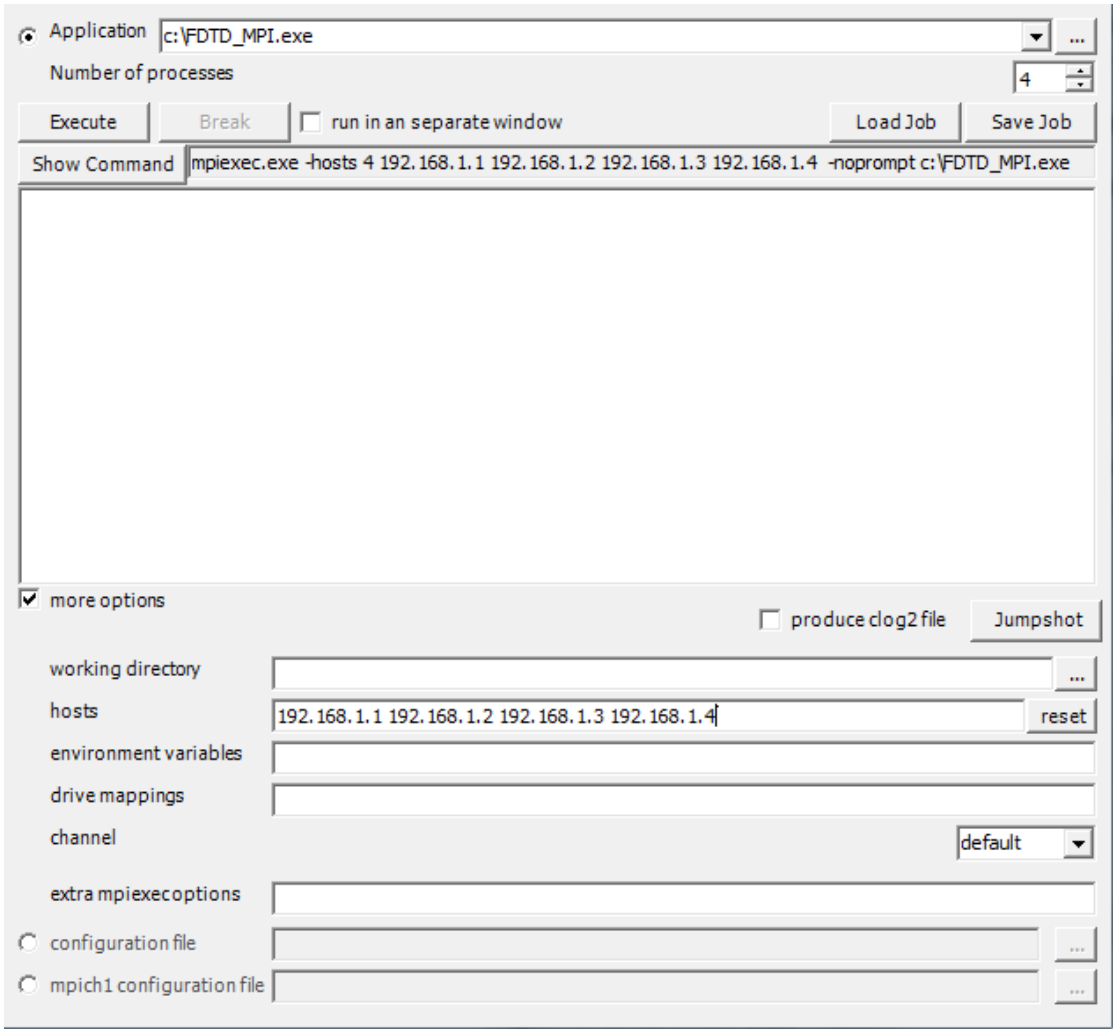

Figure 4.4: MPICH2 main window with more option activated.

Now, after active more option in main window, all PCs addresses specified in the host field, as shown in Figure 4.4, for instance, parallel system of 4 PCs. Finally, to execute the system, press Execute bottom, to make parallel system begin execute parallel project that specified in main window of program. Then system monitor and program message appear in big field down of "Show Command" bottom and field.

4.3.3 Visual studio 2010 configuration

For building code program, the editor of visual C, need some additional configuration to work with MPI's libraries. After open visual studio 2010 and creating a new project, right click at the project's name, click properties, go to configuration properties, then C/C++, then select general, then click additional include directories then add new, then add the following path: "C:\program files\MPICH2\include", as shown in Figure 4.5.

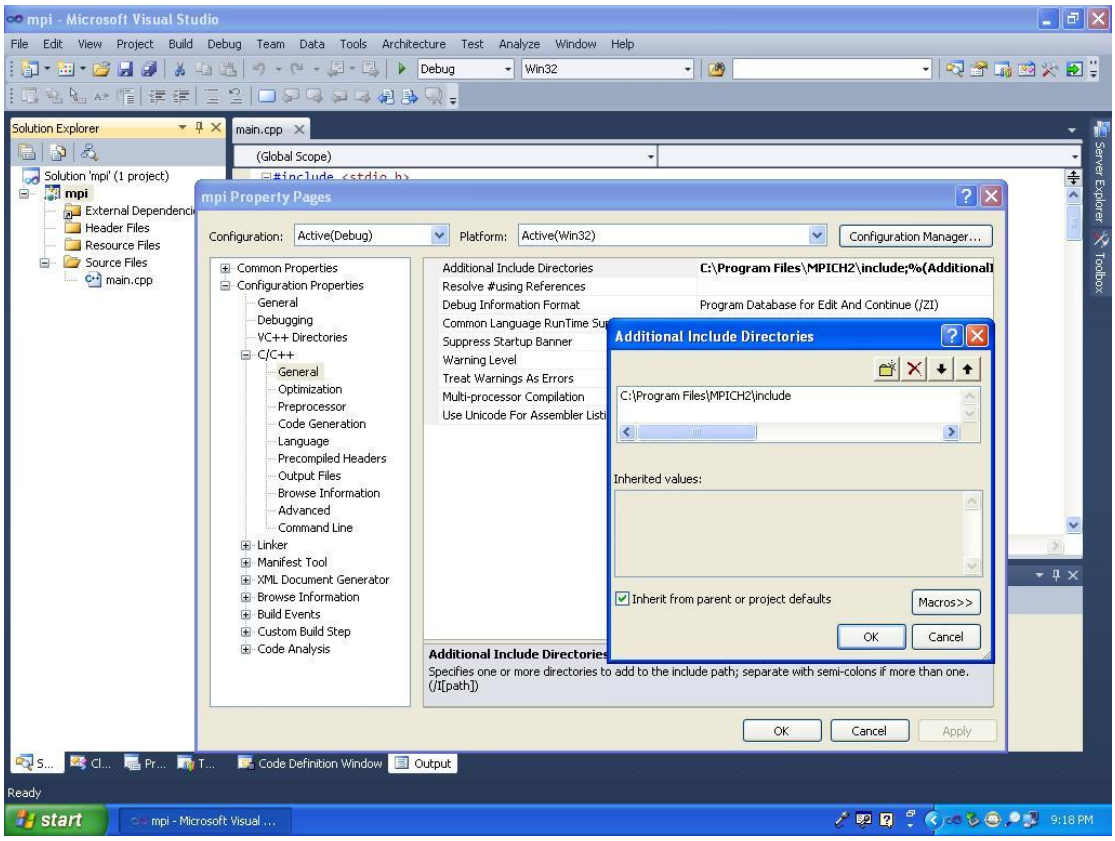

Figure 4.5: Editor configuration.

Next step, click linker, then go to general, and then select additional library directories, double click, add new, and add this path: "c:\program files\MPICH2\lib" as shown in Figure 4.6.

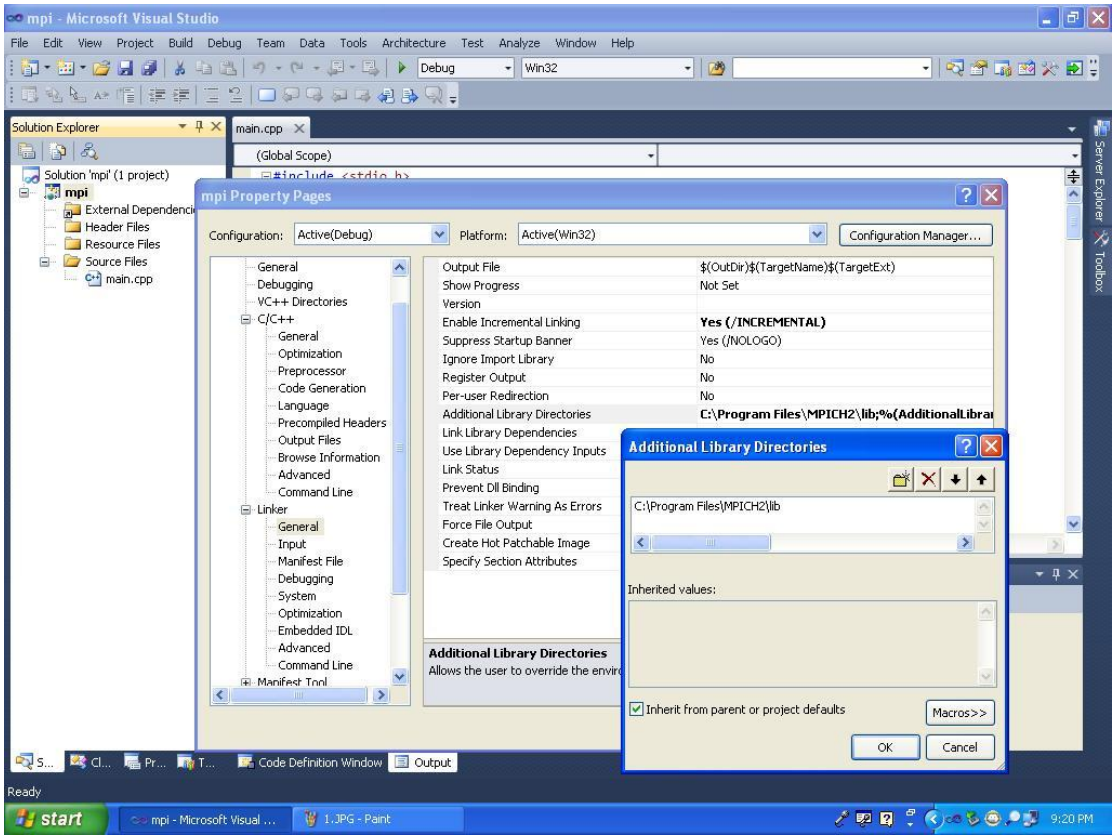

Figure 4.6: Libraries directories of MPI.

Finally, click input then, go to dependencies then, add these libraries cxx.lib, fmpich2.lib, fmpich2g.lib, fmpich2s.lib, mpe.lib and mpi.lib. as shown in Figure 4.7. These are a functions library that you needed under editor for building parallel applications.

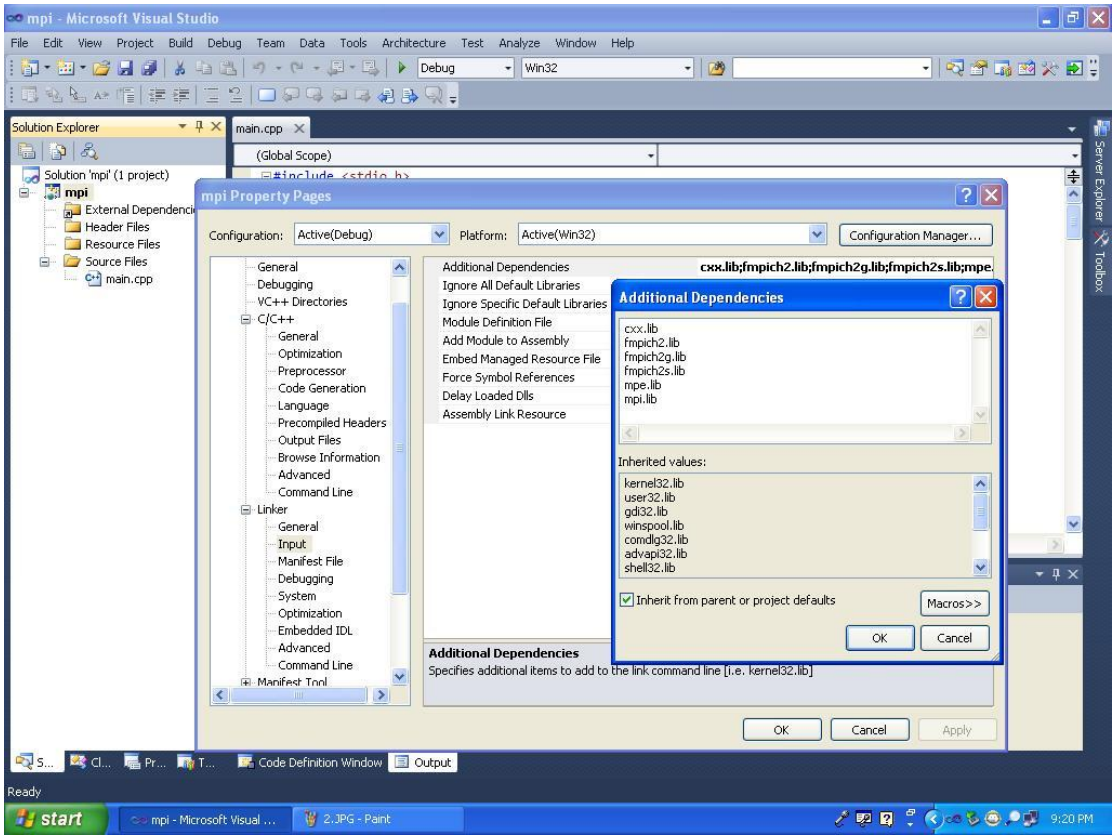

Figure 4.7: Additional of library dependencies.

4.4 Program description

The program calculates the E and H fields at each cell in a (2-D) topology. In the program, the computational domain, with dimensional $X \times Y$ is decomposed equally for computation on P PCs; where X is the long of the x-dimension of the computational grid, Y is the long of the y-dimension of the computational grid, and P is the number of PCs used in the simulation [13].

In this thesis, the parallel implementation of the 3-D FDTD algorithm in dispersive medium has been examined. The parallelization of the given FDTD algorithms is performed using the 2-D topologies. For the communications between the neighboring

processors, *MPI_Send* and *MPI_Recv* of communications functions have been implemented with MPI library.

The parallel code is written in ANSI C language using the MS Visual C++ 2010 as the application package. The parallel codes are organized in a master-worker fashion. There is only one master processor and one or more worker processors (in this thesis up to 16 worker processors were used). The master processor is responsible for both data I/O and calculations of the E-field and the H-field components at the cells in its subdomain, where each worker processor is responsible only for the calculations of the E-field and the H-field components at the cells in its subdomain. During the execution, both master and worker processors calculate the field components in their own subdomain concurrently. After the execution of the field´s components, each worker processor sends its part of the result to the master processor and terminates its process, where the master processor receives this result from the worker processors, prints them together with its own result and then terminates its process.

At the beginning of the execution, each processor defines the program parameters such as the problem size, the number of time steps, etc. Then, each processor initialize the MPI communication system, determines the number of processors to be used in the execution and its own processor identifier with the following MPI instructions:

- MPI_Init (&argc, &argv)
- MPI_Comm_size (MPI_COMM_WORLD, &nproc);
- MPI_Comm_rank (MPl_COMM_WORLD, &procid);

where MPI_COMM_WORLD is the default communicator, which indicates all processors involved in the execution. The function MPI_Comm_size gets the number of processors and stores it in *nproc*, and function MPI_Comm_rank gets the identifier of the current processor and stores it in *procid*. Then, each processor defines the Cartesian topology on the 3-D computational domain [4].

4.4.1 Two Dimensional Cartesian domain topology.

The 2-D Cartesian topology of processors is created by the following function:

MPI_Cart_create (MPI_COMM_WORLD, ndims, &dims, &periods, reorder,

&comm cart)

where: ndims = 2, refer to 2D topology, communication with x and y,

dims[0] = number of processors in the x-direction,

dims[1] = number of processors in the y-direction,

periods[0], periods[1], and reorder are defined as zero in each program, which mean we are using default communicator ranking order [4]. To determine the new processor identifier, the Cartesian coordinates and the neighboring processors in the Cartesian topology, the following functions are used for each processor:

MPI_Comm_rank (comm _cart, &procidcart);

MPI_Cart_coords (comm_cart, procidcart, ndims, &proccoord);

MPI_Cart_shift (comm_cart, rightleft, right, &left_n, & right_n);

MPI_Cart_shift (comm_cart, updown, up, &down_n, &up_n);

where: ndim: is equal 2 for 2-dimentional, left_n: is the left buffer node, right_n: is the right buffer node, down_n: is the down buffer node, up_n: is the up buffer node , rightleft = 0: is x-directional exchanging data, right = 1: is the number of moving steps, updown = 1: is y-directional exchanging data, and up = 1: is the number of moving steps. MPI_Comm_rank, gets the identifier of the current processor in the Cartesian topology and stores it in procidcart (process id cartesian), MPI_Cart_coords, gets the coordinates of the current processor in the Cartesian topology and stores in proccoord, the first function of MPI_Cart_shift, gets the identifiers of the left and the right neighbors along the x-direction and stores in left_n and right_n, respectively, and the second function of MPI_Cart_shift, gets the identifiers of the down and the up neighbors along the y-direction and store in down_n and up_n, respectively.

When the number of processors in the x-direction and the y-direction and the processor coordinates are determined, each processor calculates the values of n_x and n_y to define its subdomain size. $n_x \times n_y \times N_z$, where $n_x, n_y,$ are volume size of sub domain at each PC, and N_z, is the number of cells in Z direction, and determines the starting point of the subdomain along the x-direction and the y-direction. Then, each processor initializes the values of the field components and the additional auxiliary variables at each cell in its subdomain. At each time step, each processor calculates the required field components and the additional auxiliary variables and then, communicates the calculated field components between the related neighboring processors. The general form of the program structure and the program flowchart are given in Figure 4.8 and Figure 4.9, respectively.

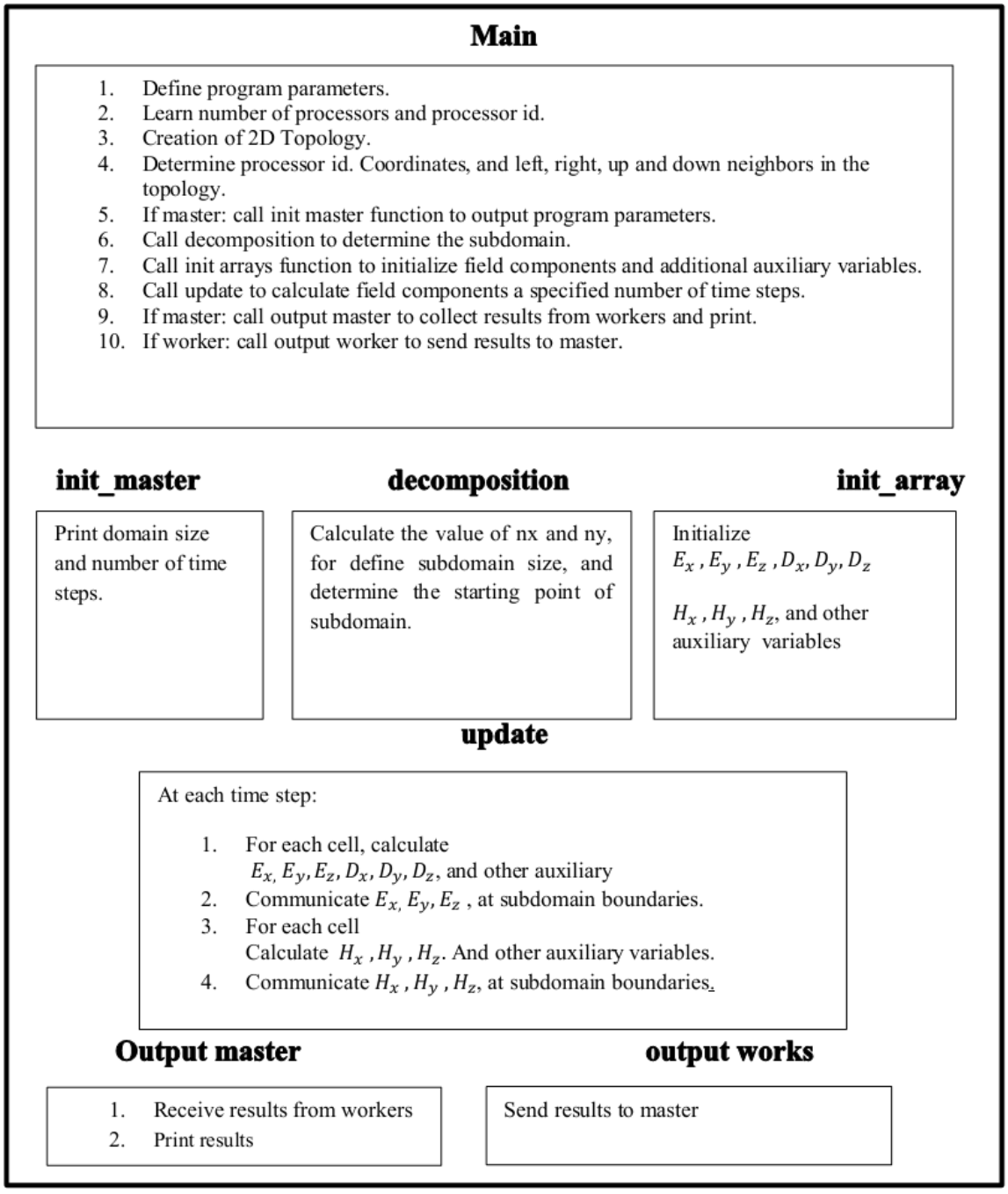

Figure 4.8: Structure of the program.

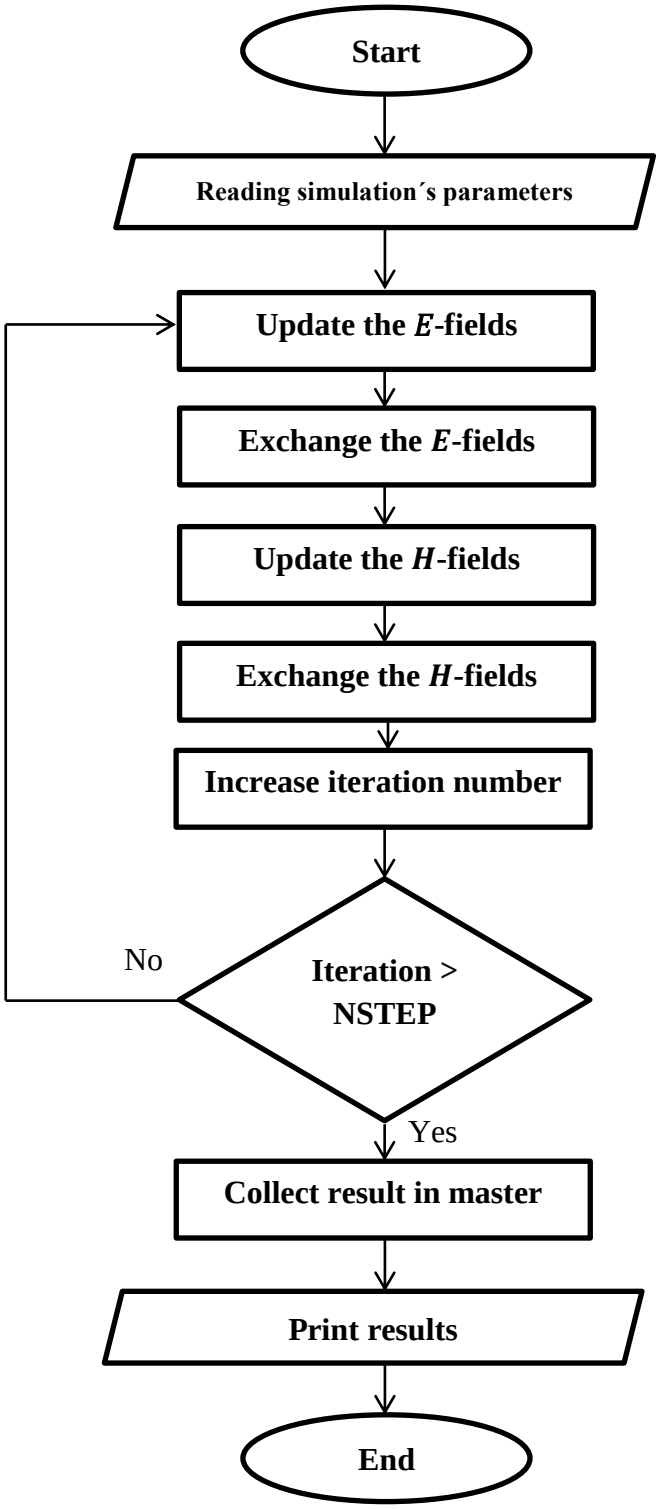

Figure 4.9: Flowchart of the parallel FDTD program.

First step in this algorithm is Initialize the MPI execution environment. Then reading simulation´s parameters. After this step, the algorithm deals with parallel topology, it is required to create of the 2-D topology, now at each time step, the algorithm perform the following

- ☒ Update the E-fields (electric fields) and other auxiliary variables in each sub-domain.
- ☒ Exchange E-fields (electric fields) with the neighbor subdomains by using the MPI library functions.
- ☒ Update the H-fields (magnetic fields) in each sub-domain.
- ☒ Exchange H-fields (magnetic fields) with the neighbor sub-domains by using the MPI library functions.

And, then increase the iteration number, then checking if the iteration number is it greater than number of steps, if it is yes, go to terminate the algorithm by MPI finalization, if it is not, repeat previous steps until iteration number be greater than number of steps, finally algorithm collect results in master PC and print the collected results, then the algorithm, goes to terminating.

Chapter 5

SIMULATION STUDY

In this chapter, the simulation results of the proposed parallel FDTD algorithms, which is used for solving Maxwell's equations in 3-D dispersive medium, are presented. The simulation cartesian domain is shown in Figure 5.1.

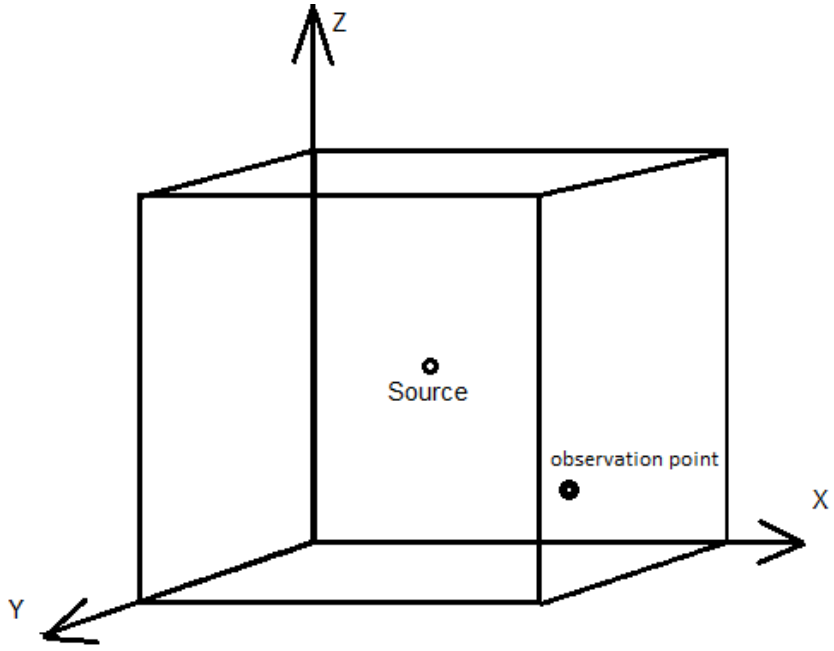

Figure 5.1: Cartesian domain with execution pulse.

The excitation pulse applied at the center of domains as shown in Figure 5.1. The form of the excitation pulse as shown in Figure 5.2

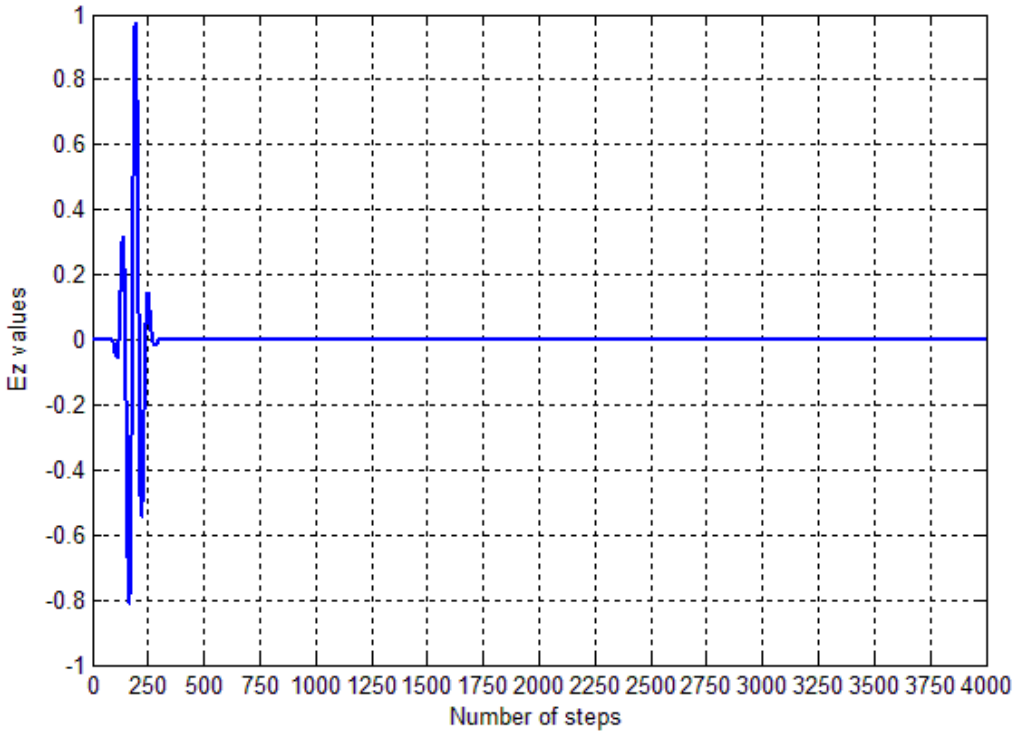

Figure 5.2: Numerical form of the excitation pulse.

The experiments was carried out using different number of PCs for comparing the simulation time by increasing the number of PCs with fixed problem size. In addition, the grid size (number of cells) is increased as much as possible to find a relationship between the number of cells in the grid and the simulation time.

The computational domain size was, $240\Delta x \times 240\Delta y \times 40\Delta z$,where the space cell size was chosen as:

$$\Delta = \Delta x = \Delta y = \Delta z = 1 \times 10^{-10} m \qquad 5.1$$

The computational domain was entirely composed of linear Lorentz material with a dielectric permittivity given by:

$$\varepsilon_r(\omega) = 1 + \frac{\Delta\varepsilon\omega_0^2}{\omega_0^2 + j2\delta\omega - \omega^2} \quad 5.2$$

where $\varepsilon_\infty = \varepsilon_r(\infty) = 1.0$, $\Delta\varepsilon = {\varepsilon_s}/{\varepsilon_\infty} - 1$, with $\varepsilon_s = 2.25$, $\omega_0 = 4 \times 10^{16} rad/s$, and $\delta = 0.28 \times 10^{16} s^{-1}$ [6]. The computational domain was truncated by eight additional PML layers with a quadratic conductivity profile and with a theoretical reflection coefficient of 10^{-5}, as defined in [3].

The simulation time was carried out for the first 4000 time steps and the time step was taken as $\Delta t = \frac{1}{2} \times \frac{10^{-10}}{c_0}$, where c_0, is the speed of light in vacuum (3×10^8). The characteristic of the parallel system used in this thesis is shown in Table 5.1.The parallel system used in this study was composed of 1-16 PCs interconnected through 100Mbps Ethernet. Figure 5.3 shows the total simulation time and the communication time of the proposed parallel algorithm.

Table 5.1: FDTD parallel system characteristics.

CPU	Core 2 due @ 3.0 GHz
RAM	2 Gigabytes
OS	Windows XP Professional X86 service pack 2
Compiler	C++
Communication software	Message Passing Interface
No. of PCs	1-16 PCs

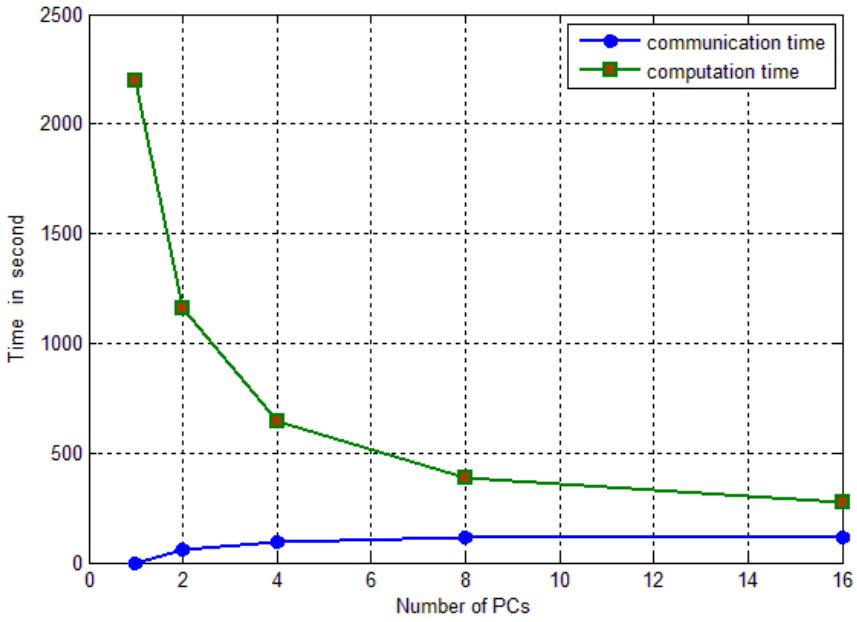

Figure 5.3: Total simulation time and communication time.

The performance of the proposed parallel algorithm was studied according to the following three factors:

1. Speedup
2. Efficiency
3. Scalability

5.1 Speedup and efficiency

The speedup was calculated as:

$$S(P) = T(1)/T(P) \quad 5.3$$

where $T(1)$ the time is needed to solve the problem using one processor and $T(P)$ is the time needed to solve the same problem using P processors [20].

The efficiency was calculated as

$$E(P) = S(P)/P \qquad 5.4$$

Figures 5.4 and 5.5 show, respectively, the speedup and the efficiency of the proposed parallel algorithm. For the purpose of comparison, the ideal speedup and efficiency were also shown in Figures 5.4 and 5.5. As can be seen from Figure 5.4, almost linear speedup was obtained when the parallel code was run on less than four processors. It also clear from Figure 5.4 that with eight PCs, speed-up factor of 5.6348 has been achieved. Beyond this, the efficiency of the parallel system decreases. This is due to the fact, that as the number of processors increases, the size of each sub-domain will be too small and hence the communication time becomes comparable to the computational time in the sub-domain. It is important to note that the performance of the parallel system can be improved further by using 3-D topology, which involves dividing the computational domain in the $x, y,$ and $z - directions$ [6] [21].

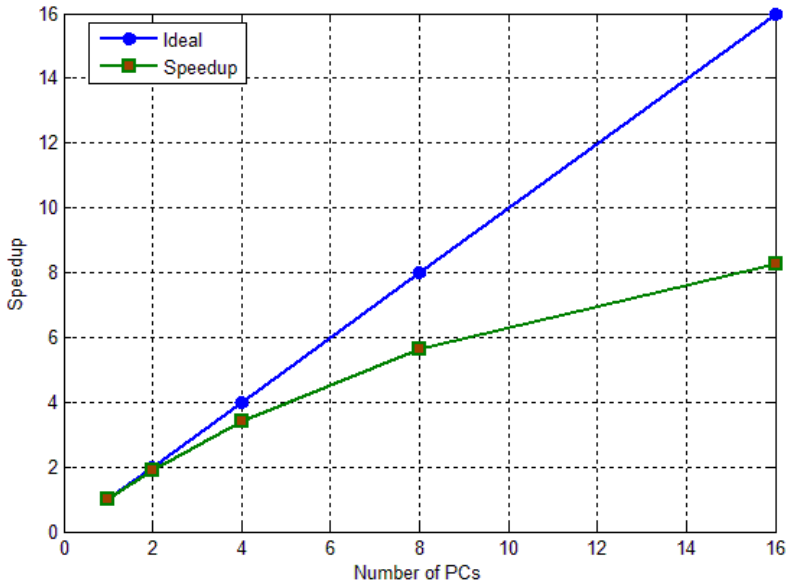

Figure 5.4: Speed-up of the parallel simulation.

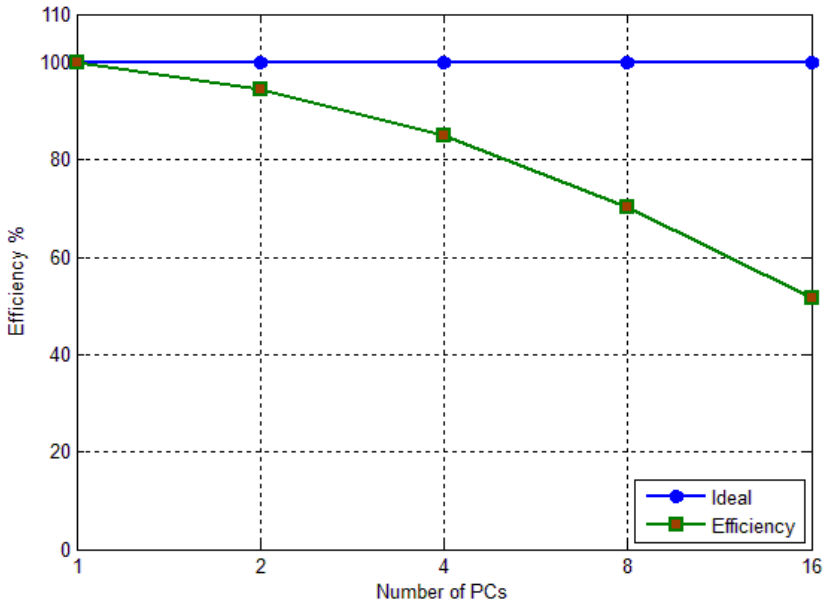

Figure 5.5: Efficiency of the parallel system.

5.2 Scalability

The scalability evaluates the performance of the parallel algorithm as the problem size scales proportionally to the number of processors. In this case, the computational problem size is kept constant per processor, while the number of processors increases. In the present study, the sub-domain size is kept fixed at 240 x 240 x 40 per processor. Table 5.2 shows the scalability of the parallel simulation. Figure 5.6 shows the total time and the communication time

Table 5.2: Scalability of the parallel algorithm.

P	P_x	P_y	N_x	N_y	**Computation time in sec**	**Communication time in sec**
1	1	1	240	240	1695.71307	00.00000
2	2	1	480	240	1703.82650	91.48884
4	2	2	480	480	1705.72423	312.79285
8	4	2	960	480	1701.15015	569.74148
16	4	4	960	960	2009.66457	607.78940

P is total number of PCs, Px is number of PC in x direction, Py is number of PC in y direction, Nz = 40.

As can be seen from these results, although the problem size is increased, there is a slight change in the total simulation time, which is due to the communication time between the processors. Hence, the proposed algorithm allows to solving very large problems easily.

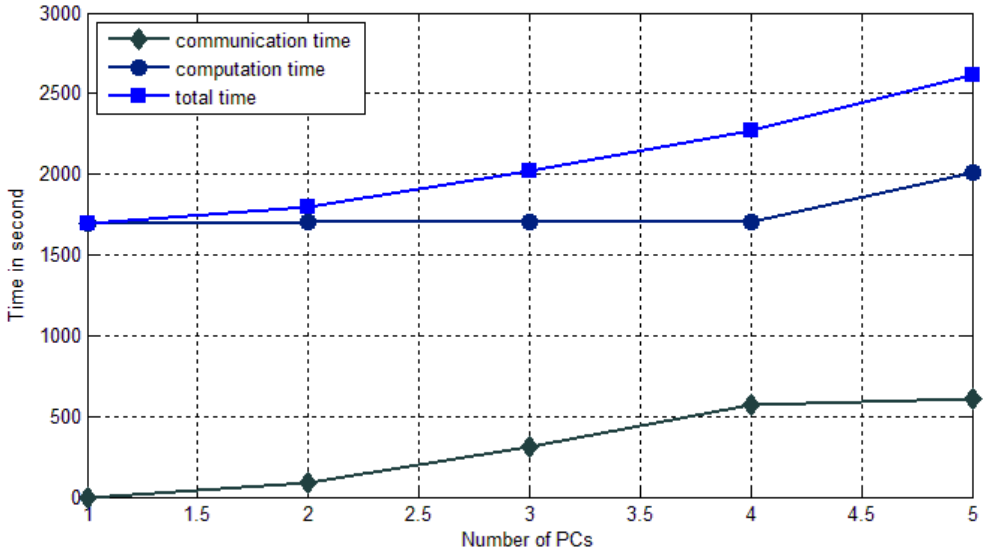

Figure 5.6: Total simulation time and the communication time of scalability.

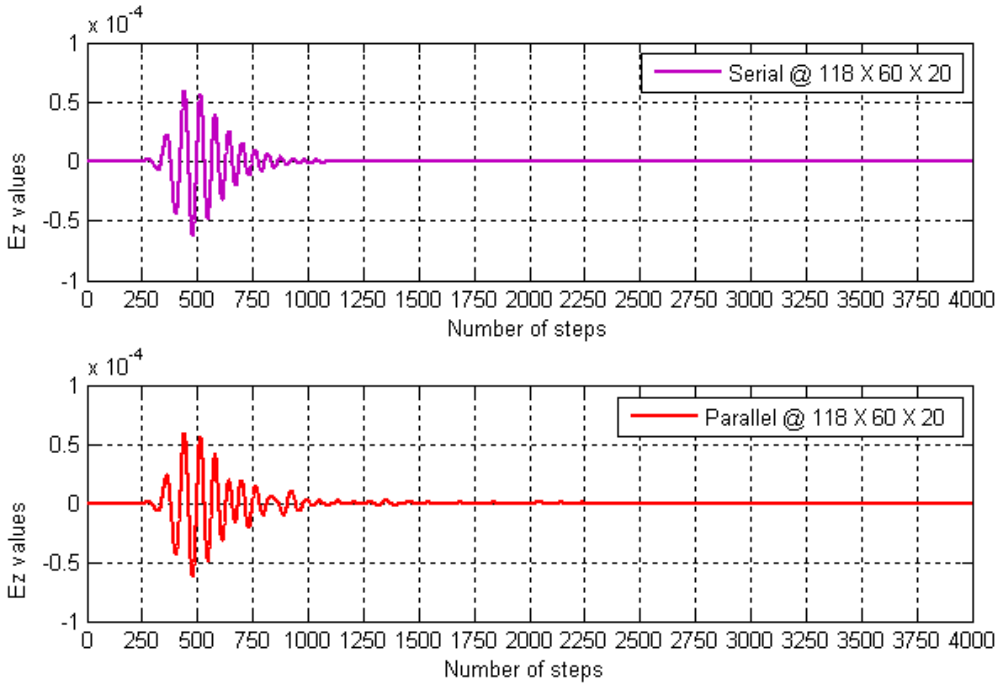

Figure 5.7: Ez-field as recorded in 4-PCs at 118 X 60 X 20.

Finally, the response of E_z at the point 118 X 60 X 20 was also examined. Figure 5.7 shows E_z versus time as obtained by the serial and parallel. From Figure 5.7, we can see, the serial and the parallel approach give same result but in a shorter time in the case of parallel.

Chapter 6

CONCLUSIONS

In this thesis, we had designed and implemented 3-D parallel algorithm, for modeling wave propagation in dispersive medium by using the MPI system incorporate with Anisotropic perfectly matched layer. The performance of the parallel algorithm has been studied for simulating a point source radiating in a 3-D Lorentz dispersive material domain.

The domain geometry is divided into non-overlapping sub-domains using the 2-D topology. It has been shown that with eight processors, a speedup factor of 5.6348 was obtained. In the other hand, when the program is distributed among many processors, the speedup decreases because the communication times become comparable to the computation time. Also it has been found, the algorithm not only speed up computations but also increases the maximum solvable problem size.

As a future study, the presented formulations can be extended for modeling electromagnetic waves interactions with human tissues like mobile phone radiations effect on human head, and this issue is under investigations.

REFERENCES

[1] A. Taflove, "The finite-difference time-domain method," in *Computational Electrodynamics*, Artech House, Boston, Mass., 2000.

[2] K. Yee, "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media," *IEEE Transactions on Antennas and Propagation*, vol. 14, no. 3, p. 302–307, 1966.

[3] J. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, p. 185–200, October 1994.

[4] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, "MPI: the complete reference," in *Scientific and Engineering Computation*, vol.1, 2nd. edtion ,ISBN 978-0262692151, The MIT Press, September 19, 1998.

[5] "MPICH2," [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/.

[6] O. Ramadan, "An efficient MPI-based parallel wave-equation FDTD algorithm for dispersive electromagnetic applications", The 5th International Conference on Information Technology, (ICIT'11), Amman, Jordan, May 11-3, 2011.

[7] R. Ziolkowski, "Time-derivative Lorentz material model-based absorbing boundary condition," *IEEE Transactions on Antennas and Propagation,* vol. 45, no. 10, pp. 1530 - 1535 , Oct 1997.

[8] D. M. Sullivan, "Electromagnetic simulation using the FDTD method," in *electrical engineering and computer science,* IEEE Press, 2000.

[9] Y. cell. [Online]. Available: http://fdtd.wikispaces.com/The+Yee+Cell.

[10] J. B. Schneider, "Understanding the finite-difference time-domain method," in *electrical engineering and computer science,* Lecture notes by John Schneider, September 4, 2012.

[11] S. Gedney, "An anisotropic perfectly matched layer-absorbing medium for the truncation of FDTD lattices," *IEEE Transactions on Antennas and Propagation,* Vol. 44 , no. 12, pp.1630 - 1639, Dec 1996.

[12] A. Oyko, "Efficient parallel algorithm for modelling open region finite differnce time domain grids," P.hD thesis, Eastern Mediterranean University, 2008.

[13] A. Oyko, "Parallel implementation of the FD-TD method using MPI," M.thesis, Eastern Mediterranean University, 2001.

[14] G. R. Andrews, "Foundations of multithreaded, parallel, and distributed programming," in *application of parallel systems*, ISBN- 978-0201357523, 1st. Edition, Addison Wesley, December 10, 1999.

[15] P. A. Kaminsky, "Parallel java a unified API for shared memory and cluster parallel programming in 100% java," [Online]. Available: http://www.cs.rit.edu/~ark/lectures/pj04/fig03.png.

[16] Dauger, "Parallel programming paradigms - processors and memory," [Online]. Available: http://daugerresearch.com/vault/DistributedMemoryModel.gif.

[17] K. Hwang, Z. Xu, "Scalable parallel computing: technology, architecture, programming," in *parallel and distributed computing*, Mishawaka IN U.S.A, ISBN-978-0070317987, 1st. edition, McGraw-Hill, 1998.

[18] P. H. Oser, "Technical design issues," 8 6 2001. [Online]. Available: http://www.oser.org/~hp/ds/img12.gif.

[19] "web pages for MPI routines," [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/www/www3/.

[20] A.D. Tinniswood, P.S.Excell, M. Whittle and Spicer, "Parallel computation of large-scale FDTD problems," *Third International Conference on Computation in Electromagnetics*, no. 420, pp. 7 - 12, 10-12 April 1996.

[21] J. A. Six, "Development of a 1-dimentional parallel FDTD algorithm," May 1999.