PARALLEL HARDWARE AND SOFTWARE IMPLEMENTATIONS FOR ELECTROMAGNETIC COMPUTATIONS

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND

ELECTRONICS ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Ali Rıza Bozbulut

September 2005

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Levent Gürel (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ayhan Altıntaş

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Tuğrul Dayar

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray
Director of the Institute Engineering and Science

To My Family...

ABSTRACT

PARALLEL HARDWARE AND SOFTWARE IMPLEMENTATIONS FOR ELECTROMAGNETIC COMPUTATIONS

Ali Rıza Bozbulut
M.S. in Electrical and Electronics Engineering
Supervisor: Prof. Dr. Levent Gürel
September 2005

Multilevel fast multipole algorithm (MLFMA) is an accurate frequency-domain electromagnetics solver that reduces the computational complexity and memory requirement significantly. Despite the advantages of the MLFMA, the maximum size of an electromagnetic problem that can be solved on a single processor computer is still limited by the hardware resources of the system, i.e., memory and processor speed. In order to go beyond the hardware limitations of single processor systems, parallelization of the MLFMA, which is not a trivial task, is suggested. This process requires the parallel implementations of both hardware and software. For this purpose, we constructed our own parallel computer clusters and parallelized our MLFMA program by using message-passing paradigm to solve electromagnetics problems. In order to balance the work load and memory requirement over the processors of multiprocessors systems, efficient load balancing techniques and algorithms are included in this parallel code. As a result, we can solve large-scale electromagnetics problems accurately and rapidly with parallel MLFMA solver on parallel clusters.

Keywords: Parallelization, Load Balancing, Partitioning, Optimization, Parallel Computer Cluster.

ÖZET

ELEKTROMANYETİK HESAPLAMALARI İÇİN PARALEL DONANIM VE YAZILIM UYGULAMALARI

Ali Rıza Bozbulut
Elektrik ve Elektronik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Levent Gürel
Eylül 2005

Çok seviyeli hızlı cok kutup yötemi (ÇSHÇY) frekans alanında hassas sonuçlar veren bir elektromanyetik çözücüsüdür; bu çözücü hesapsal karmaşıklığı ve bellek gereksinimi oldukça azaltmıştır. ÇSHÇY'nin tüm bu yararlarına karşın, tek işlemcili bilgisayarların donanım kaynakları, örneğin bellek miktarı ve işlemci hızı, çözülebilen elektromanyetik problemlerin büyüklüğünü kısıtlamaktadır. Tek işlemcili sistemlerin bu kısıtlamalarını aşabilmek için, ÇSHÇY'nin paralelleştirilmesi önerilmiştir. ÇSHÇY'nin paralelleştirilmesi kolay bir işlem değildir, bu işlemin yapılabilmesi için paralel donanım ve yazılım alanında yoğun emek ve deneyim gerekmektedir. Elektromanyetik problemleri paralel ortamda çözebilmek icin kendi paralel bilgisayar kümemizi kurduk ve ÇSHÇY kodumuzu paralelleştirdik. Paralel bilgisayarlar üzerindeki iş yükünü ve bellek kullanımını dengeleyen verimli yük dengeleme algoritmalarını ve yöntemlerini paralel kodumuza yerleştirdik. Şu anda, paralel ÇSHÇY çözücümüzle gelişigüzel şekilli çok büyük elektromanyetik problemleri paralel bilgisayar kümeleri üzerinde çok kısa bir zamanda çözebilmekteyiz.

Anahtar sözcükler: Paralelleştirme, Yük Dengelemesi, Yük Dağıtımı, Optimizasyon, Paralel Bilgisayar Kümesi.

ACKNOWLEDGEMENTS

I gratefully thank my supervisor Prof. Dr. Levent Gürel for his supervision and guidance throughout the development of my thesis.

I also thank Prof. Dr. Ayhan Altıntaş and Assoc. Prof. Dr. Tuğrul Dayar for reading and commenting on my thesis.

I also thank all graduate students of Bilkent Computational Electromagnetics Laboratory for their strong support in my studies.

I express my sincere gratitude to TUBITAK for allowing access to their high performance computing cluster, ULAKBIM, on which part of the computations for the work described in this thesis was done.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computational electromagnetics (CEM) deals with the solution of real-life electromagnetic problems in the computer simulation environment. In our studies, we mainly focus on two types of of these real-life problems: Radiation problems and scattering problems.

In radiation problems, we aim to model and simulate the electromagnetic field radiation from a source which is placed on a conducting body e.g., far-field radiation modeling of an antenna. On the other hand; in scattering problems, our goal is focused on the simulation of scattered electromagnetic field, which is transmitted from a conducting object after the illumination of this object with an external electromagnetic radiation source e.g., radar cross section (RCS) profile computations of a helicopter.

In order to solve these kinds of problems accurately, many researchers in the field of computational electromagnetics use the method of moments (MoM). MoM is based on the discretization of electromagnetic integral equation into a matrix equation. The memory requirement and computational complexity of this method are both in the order of O(N^2), where N is number of unknowns;

because of such high computational complexity and memory requirement, the size of the electromagnetic problem that is solvable with this method is very limited. Even today's computers can not satisfy the memory requirement of a MoM based electromagnetic simulation program for the solution of a moderate size electromagnetic problem which might have 50000–60000 unknowns—a MoM based program needs at least 20 GBytes of memory to solve a 50000 unknowns electromagnetic scattering problem.

In 1993, Vladimir Rokhlin proposed the fast multipole method (FMM) for the solution of electromagnetic scattering problems in three dimensions (3-D) [1]. FMM reduces the complexity of matrix-vector computations and memory requirement to O($N^{1.5}$), where N is the number of unknowns. After the proposal of this new method, multilevel versions of FMM were proposed [2]. Weng Cho Chew introduced a new view to multilevel FMM by using translation, interpolation and anterpolation (adjoint interpolation) concepts. This new multilevel FMM approach was renamed as multilevel fast multipole algorithm (MLFMA). Chew and his group reduced the computational complexity and memory requirement of FMM to O($N \log N$) with the new MLFMA [3].

With this new algorithm, MLFMA, people started to solve large scattering problems, which are not solvable with MoM. Despite the advantages of MLFMA in terms of memory and computational time, people have to use very effective supercomputers to solve large scattering problems, which correspond to the number of unknowns in the order of one million. Using the supercomputers for the solution of large electromagnetic scattering problems is very costly; because of that reason and to overcome the hardware limitations of supercomputers, researchers shifted their studies into the parallelization process with message passing paradigm. In 2003, research group of Weng Cho Chew at University of Illinois computed bistatic RCS of an aircraft, whose dimensions are not scaled down, at 8 GHz with the parallel implementation of MLFMA [4]. This computation corresponds to the solution of 10.2 million unknown dense matrix equations and this is a turning point in CEM area. After that achievement, many research groups in CEM community started to work on parallel implementations of MLFMA. Our group has been working on parallel MLFMA since 2001 and we have developed

our own parallel MLFMA code in order to solve very large scattering problems on Beowulf parallel PC clusters. Now, we can solve 1.3 million unknowns helicopter problem with our parallel MLFMA code on a 32 nodes parallel PC cluster. Our studies are still continuing and our aim is to achieve the computation capability to solve much larger electromagnetic scattering problems, which might have number of unknowns greater than 3–3.5 millions.

1.2 Parallel Computer Systems and Parallelization Techniques

In this part, we present an overview of various architectural basics and concepts which lay behind the parallel computer systems. These basics are detailed according to two categories: We give the details of hardware structure choices that most of parallel computers are based, and then, we introduce the programming models in order to use these hardware structures effectively.

From hardware structure point of view, parallel computer platforms can be classified with respect to the communicational model that they use. On the other side, from software point of view, parallel systems can be grouped according to the control structures in order to program them. So, in the following two subsections we mainly focus on the these issues:

1. Communicational Model of Parallel Computer Systems
2. Control Structure of Parallel Computer Systems

1.2.1 Communicational Model of Parallel Computer Systems

Parallel computers need to exchange data during parallel tasks. There are two primary ways to do this data exchange between parallel jobs: By using shared-address-space systems or with the usage of sending messages between tasks. Depending on these mentioned communicational approaches, parallel computer platforms can be grouped in two groups:

Shared-address-space platforms: This type of parallel systems have single memory addressing space, which supports the access to the memory from all the processors of the system. These parallel platforms are also sometimes referred to as multiprocessors [5].

Message-passing platforms: In this approach, every processing unit or processor of these platforms has its own address space for its memory. So, none of the processing units can reach other processing units' data directly. Because of this reason, processing units send and receive messages between themselves in order to do data exchange. Since these systems are mostly based on the connection of separate single processor computers, they are sometimes called as multicomputers [6].

Shared-Address-Space Platforms

Memory in these parallel computers might be common to all the processors or it may be exclusive to each processor. If the time required to access memory address on the system is equal for all processors, this shared-address parallel computer is classified as uniform memory access (UMA) computer. On the other hand, if this access duration changes from one processor to the another processor, this system is categorized as non-uniform access (NUMA) parallel computer system. UMA and NUMA architectures are depicted in the Fig. 1.1:

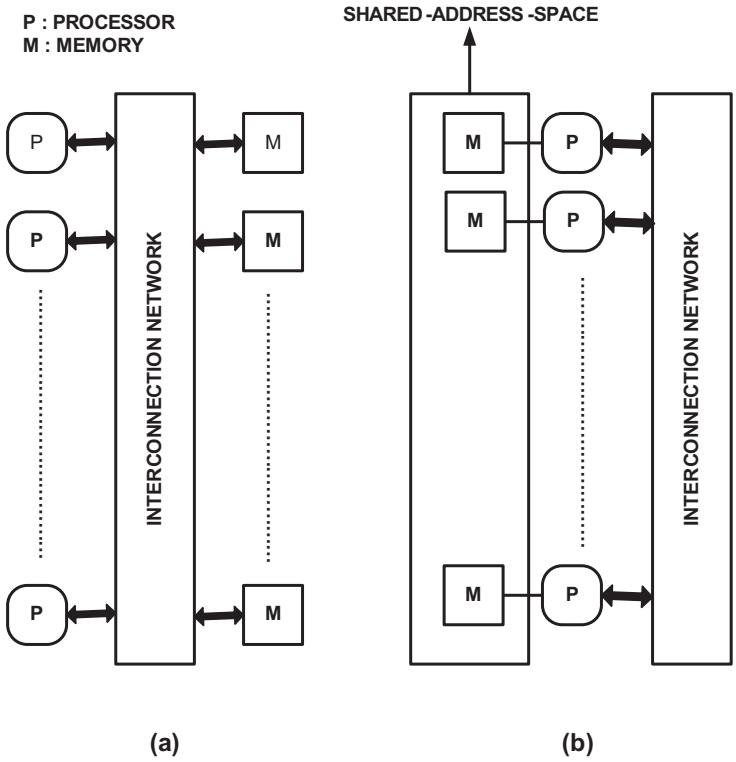

Figure 1.1: Shared-address-space architecture models: (a) uniform-memory-access multiprocessors system (UMA) model and (b) non-uniform-memory-access multiprocessors system (NUMA) model

Programming in these systems is very attractive with respect to message-passing multicomputers; because, the data is shared among all the processors. Therefore, the programmer need not worry about how to update the data if there is a change during the execution of parallel task in these systems. Generally, shared address space computers are programmed using special parallel programming languages, which are extensions of existing sequential languages; e.g. High Performance Fortran 90 (HPF90) is derived from Fortran 90 in order to program the multiprocessors in parallel with Fortran language. In addition to these parallel languages, threads can be used for programming these platforms.

Despite these advantages in programming these systems, generally multicomputers are very expensive with respect to off-the-shelf single processor computers. Moreover, the development stages of these platforms take a lot of time and therefore, these platforms are usually superseded by single processors, which

have been regularly upgraded. For these reasons, message-passing multicomputers have been more popular with respect to these systems in the last 20 years. Nevertheless, Intel and AMD still produce processors for special computational applications: Intel Itanium and Xeon processors, and AMD Opteron processor based systems are recent examples of such systems; but, generally these processors are not used in large scale multiprocessors systems. Researchers working in parallel processing choose to deploy these processors in small scale multiprocessors and utilize them in the parallel message-passing multicomputer systems—these small scale multiprocessors generally support two processors or four processors, very few of them contain eight processors.

Message-Passing Platforms

Another approach to build a parallel processor hardware environment is to connect the complete computers through an interconnection network, which is shown in Fig. 1.2. These computer systems can be chosen from off-the-shelf processors and computer components. Parallel Beowulf Cluster is the most popular example of this type of parallel platforms.

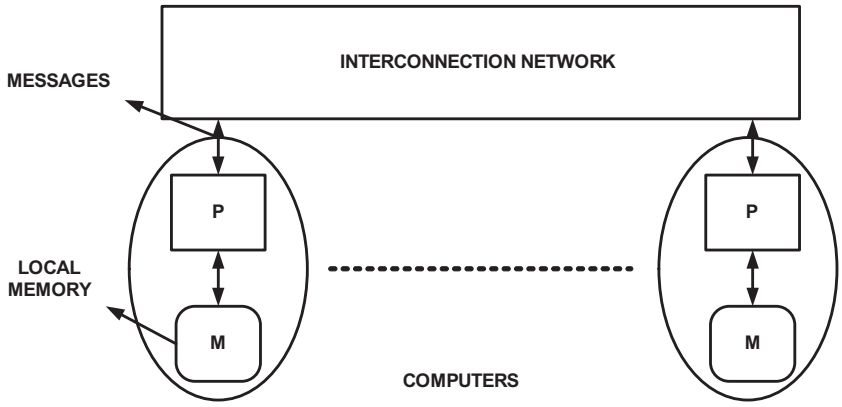

Figure 1.2: Message-passing multicomputer model

Beowulf clusters are generally built from ordinary 'off-the-shelf' computers. These computers are connected to each other using a network switch which supports fast ethernet or gigabit ethernet connections. Linux is the most extensively used operating system on these clusters [7], because parallel programming

libraries, such as message passing interface (MPI) [8] [9] and parallel virtual machine (PVM), are installed as default tools in Linux. Moreover, Linux file system tools and servers make the installation of these clusters very easy and the biggest advantage of Linux is that, especially for research groups, we do not need to pay money in order to use it. On the other hand, Microsoft Windows 2000 based operating systems, such as Microsoft Server 2003, can also be used as the operating system in a Beowulf cluster. However, by experience, Microsoft's operating systems' performance decrease for more than eight computers connected in a cluster connection manner.

1.2.2 Control Structure of Parallel Computer Systems

Processing units in parallel computer systems either work independently or process their tasks under the supervision of a special control unit. Single instruction stream, multiple data stream (SIMD) control architecture corresponds to the previously mentioned special supervision control over the processing units. In this control approach, control unit sends the instructions to each unit of the system and all units execute the same instruction synchronously. Today, Intel Pentium Processors support this control architecture; actually, this control methodology obtains very small scale parallelization inside the single processor architecture.

SIMD works properly on structured data computations; however, this methodology is not feasible for both shared-address space parallel computers and message-passing multicomputer, because, these systems are composed of physically separate processing units. Multiple instruction stream, multiple data stream (MIMD) approach was proposed in order to handle the control mechanism of such parallel computing environments. In Fig. 1.3, these two control methodologies are depicted:

MIMD model supports two programming techniques: Multiple program, multiple data (MPMD) and single program, multiple data (SPMD). In MPMD technique, each processor unit is able to execute a different program on different data; e.g. a hybrid cluster, which is a cluster composed of computers with different

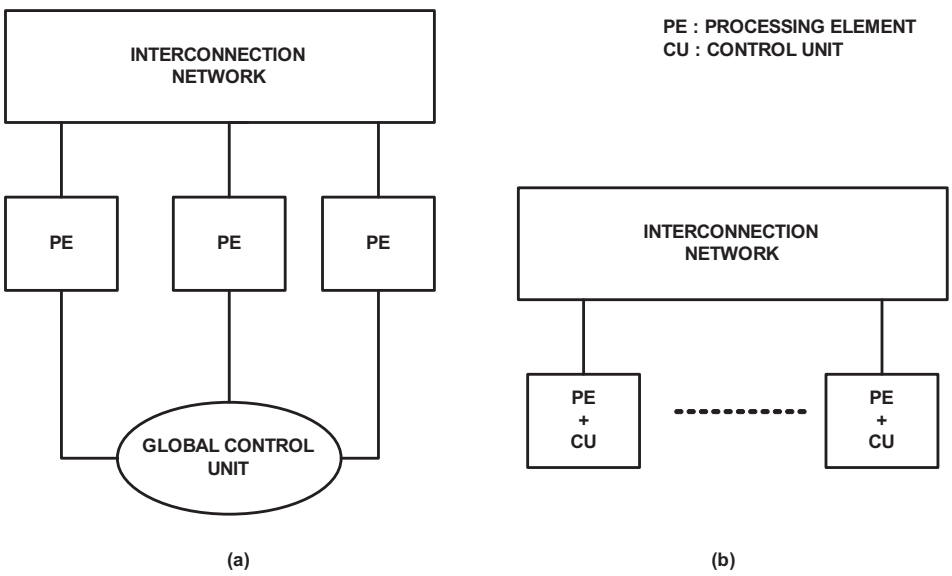

Figure 1.3: Control architectures: (a) single instruction stream, multiple data stream (SIMD) model and (b) multiple instruction stream, multiple data stream (MIMD) model

processor architectures, such as 32 bit and 64 bit machines connected together; different executables and programs, should be compiled for each architecture. On the other hand, SPMD technique, a single code is compiled for all the processors of the parallel system. However, by using if-else blocks with conditional statements, we can assign different tasks to different processors; e.g. master-slave parallel programming approach can be implemented with that programming technique by using conditional statements; we can separate the tasks of the master processing unit from the tasks of the slave units'.

Another instruction model is the multiple instruction stream, single data stream. In this model, different instructions are executed on the same data by different processing units; e.g. pipelining approach.

1.3 Parallelization Basics and Concepts

During the improvement of our parallel MLFMA code, we continuously took measurements to control the effects of our improvements and changes on the performance of our parallel program. In these measurements, we looked at certain metrics, which are used as common performance measurements in parallel processing applications. In this part, brief information is presented on these parallel measurement metrics in order to give some intuition before showing the performance results of our parallel MLFMA code. In addition to these metrics, we give some definitions of some basic parallel concepts for the aim of understanding these metrics:

Processes are simultaneous tasks that can be executed by each parallel processing unit—this processing unit generally refers to a processor in a parallel platform. Parallelization's main methodology for the solution of a problem is actually to divide the problem into smaller pieces, processes, and then solve these pieces over connected processing units, which are placed in either a shared address space parallel platform or a message-passing multicomputer.

Execution time is the duration of the runtime of a program elapsed between the beginning and the end of its execution on a computer system. The execution time of a parallel program is denoted by T_p and the execution time of sequential program which is run on a single processor computer is denoted by T_s.

Overhead, or total parallel overhead of a parallel platform is the excess of total time collectively spent by all processing units in a parallel system with respect to the required time by the fastest known algorithm in order to solve the same problem on a single processor computer. It is also named as overhead function and it is denoted with T_0:

$$T_0 = pT_p - T_s \tag{1.1}$$

where p is the number of processors in the parallel system; T_p and T_s are parallel and sequential execution times respectively.

Overhead in parallel systems is mainly dependent on three factors:

1. Time durations when not all processing units are doing useful work or computations and are waiting for other processors—this is also called as idle time.
2. Extra computations which are done in parallel implementation of computations; for instance recomputation of some constants on the processing units of a parallel platform.
3. Communication time which is spent during the sending and receiving of messages.

Granularity is defined as the size of computations between communication and synchronization operations on message-passing multicomputers. It can be summarized as the ratio of computation time to the communication time during the execution of a parallel program.

$$\text{computation / communication ratio} = \frac{t_{comp}}{t_{comm}}, \tag{1.2}$$

where t_{comp} is the computation time and t_{comm} is the communication time

In parallel systems, most of the time, our aim is to make granularity as big as possible.

Speedup is the ratio of the time duration for the solution of a given problem on a single processor computer to the time needed to solve the same problem with p identical processors on a parallel platform. Speedup is denoted by S and it is formulated as follows:

$$S = \frac{T_s}{T_p}, \tag{1.3}$$

where T_s and T_p are the sequential and the parallel execution times, respectively.

Maximum achievable speedup on parallel processor system with p number of processor is p, if processors do not wait idle or communicate with each other.

If the speedup of a parallel program is p, then this program is **perfectly parallelized**. In real life, this situation does not happen frequently.

Sometimes, a speedup greater than p is observed in the parallelization of a program; this extraordinary situation is named as superlinear speedup. This phenomenon usually occurs if the computations performed by a serial algorithm is greater than its parallel counter part or because of the hardware features that put the serial implementation at a drawback.

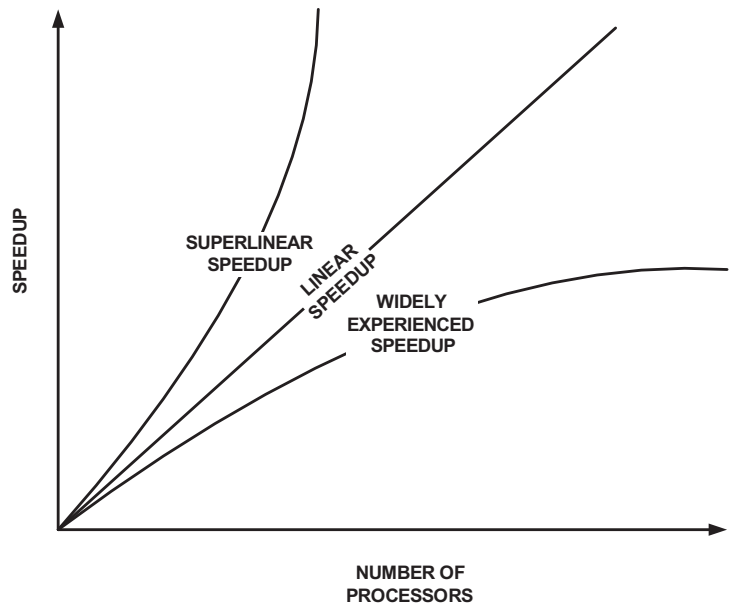

Figure 1.4: Typical speedup curves

Efficiency is the metric of the fraction of time duration for which a processing unit is usefully employed on the system. Actually, it is defined as the ratio of speed up, S, to the number of processors, p, in a parallel platform. In the parallelization of a program, our aim is to achieve efficiencies as close to unity as possible.

Gain, or memory gain is the ratio of amount of memory needed to solve a problem on a single processor system to the maximum amount of memory needed by a single processing unit of the parallel system which deploys p identical processors in order to solve the same problem. Gain is generally

denoted by G:

$$G = \frac{M_s}{M_p}, \tag{1.4}$$

where M_s and M_p are the amount of sequential and parallel memory requirements, respectively.

Scalability is the ability of a parallel program to be executable in a larger hardware environment with more processors, more dynamic memory, more storage, etc., with a proportional increase in performance.

1.4 An Overview to the Parallelization of MLFMA

As mentioned before, MoM discretizes the electromagnetic integral equation to a matrix equation. This matrix in MoM is called as impedance matrix or $\overline{\mathbf{Z}}$ matrix and this matrix equation corresponds to a full dense matrix equation system. In MLFMA methodology, we rewrite this full impedance matrix as the summation of two matrices:

$$\overline{\mathbf{Z}} = \overline{\mathbf{Z}}_{\mathbf{near}} + \overline{\mathbf{Z}}_{\mathbf{far}}, \tag{1.5}$$

where $\overline{\mathbf{Z}}_{\mathbf{near}}$ corresponds to near-field matrix elements of impedance matrix and $\overline{\mathbf{Z}}_{\mathbf{far}}$ denotes the far-field components of this $\overline{\mathbf{Z}}$ matrix—Near-field elements are electromagnetically inducing currents which have a strong electromagnetic influence (interaction) between them; on the other hand, far-field elements correspond to the electromagnetically induced currents on the same geometry whose electromagnetic interactions can be formulated by far-field approximations [10].

Despite the $\overline{\mathbf{Z}}$ matrix being written as a summation of two matrices, these two new matrices are not stored in the memory as sparse matrices. Memory required for each component of these two matrices are computed carefully and the elements of these matrices is stored in one dimensional arrays, because effective memory usage is one of the most crucial part of our programming approach in MLFMA. In addition to that, in MoM, whole $\overline{\mathbf{Z}}$ matrix is filled with its elements

before the matrix solution starts; however, in the MLFMA approach, just near-field interactions on near-field elements ($\overline{\mathbf{Z}}_{\mathbf{near}}$) are computed before the matrix solution begins. In our MLFMA program, we actually use iterative solvers in order to solve this matrix equation, and far-field interactions in far-field elements ($\overline{\mathbf{Z}}_{\mathbf{far}}$) are computed dynamically in each iteration of the iterative solver.

Far-field interactions are computed very differently with respect to the near-field interactions. In these calculations, far-field related induced currents are grouped in clusters; moreover these clusters are also grouped. This clustering strategy is used recursively and a tree structure is formed by this way. This tree structure is called as MLFMA tree and electromagnetic computations over that tree are done by special operations such as: interpolations, anterpolations and translations [3] [11]. Because of these two different structured matrices ($\overline{\mathbf{Z}}_{\mathbf{near}}$ and $\overline{\mathbf{Z}}_{\mathbf{far}}$) and electromagnetic computational approaches (near-field and far-field computations), parallelization of MLFMA is not trivial. It certainly needs quite a lot of effort and thinking in order to be parallelized.

We use a Beowulf PC cluster, which is relatively cheap and flexible for our purposes, as parallel platform for our parallelization approach. Our sequential MLFMA program was written in a non-standard Fortran 77 format, Digital Fortran 77, and thus even the adaptation of the sequential version of MLFMA to this new hardware backbone caused a lot of problems. After the adaptation of the sequential program into our platform, we started to develop our own parallel MLFMA code. In this parallelization process, we have written our code according to SPMD methodology with the usage of MPI library for Fortran 77. We tried different parallelization techniques. At the first stage we programmed the parallel MLFMA according to master-slave programming approach and since we do not have a parallel iterative solver, one of the nodes on the cluster was used as a master of the other processors. This master node is the main computer of the cluster and it does more work with respect to the other processors. After the adaptation of a parallel iterative solver library into our parallel program, the Portable, Extensible Toolkit for Scientific Computation (PETSc), all nodes share the total work load of parallel MLFMA almost equally. The details of these improvements and developments are given in the following chapters.

Chapter 2

Parallel Environments

As we mentioned earlier, we chose to use a Beowulf cluster [7] [12] as our hardware backbone in order to improve our own parallel MLFMA program. For this aim, we built our own message-passing multicomputer: Building such a system is not a trivial process, but it certainly gives us a lot of advantages in our parallelization efforts:

- By building our own cluster, we freely develop our parallel MLFMA; we do not have a dependence to other sources and this fastens our studies after we get familiar with the parallel system.
- When you have your own cluster, you can have full control over the software and hardware structure of your cluster. Therefore, you have a chance to do and experiments in order to find an optimum hardware and software structure for your own needs.

Depending on these reasons, we did several changes and made upgrades in our cluster system. The clusters that we worked on include:

1. 32-bit homogeneous PC cluster.
2. 32-bit and 64-bit hybrid cluster.
3. 64-bit homogeneous multiprocessor cluster—proposed.

We also did measurements and program development studies on TUBITAK's ULAKBIM high performance cluster.

2.1 32-Bit Intel Pentium IV Cluster

The first cluster that we built is the 32-bit Intel Pentium IV cluster. In the very first stage, it just contained three computers and in time this number went up to nine. As our improvements on parallel MLFMA progressed, the speed and performance of that cluster become the bottleneck. Nevertheless, we still use this cluster architecture and we are solving many problems with this system working at its limits. Today, this cluster consists of four computers since we have taken out some nodes for special purposes, all having identical Intel Prescott Pentium IV 2.8 GHz processors and 2 GBytes of memory installed on each of them. We use a non-blocking fast ethernet based network switch to connect these computers. This cluster is completely based on the Beowulf cluster system in terms of hardware and software structure. Thus, by looking at this cluster, we can understand the Beowulf cluster topology in both hardware and software:

2.1.1 Hardware Structure of 32-Bit Intel Pentium IV Cluster

Beowulf clusters are the most extensively used parallel computers. As a message-passing multicomputer, in the Beowulf clusters, separate computers send messages with the aid of a network environment. Fast ethernet or gigabit ethernet switches are used for interconnection purposes. In our cluster we used 3com's 48

port non-blocking fast ethernet superstack network switch. Hardware structure of our cluster is schematically shown in Fig. 2.1:

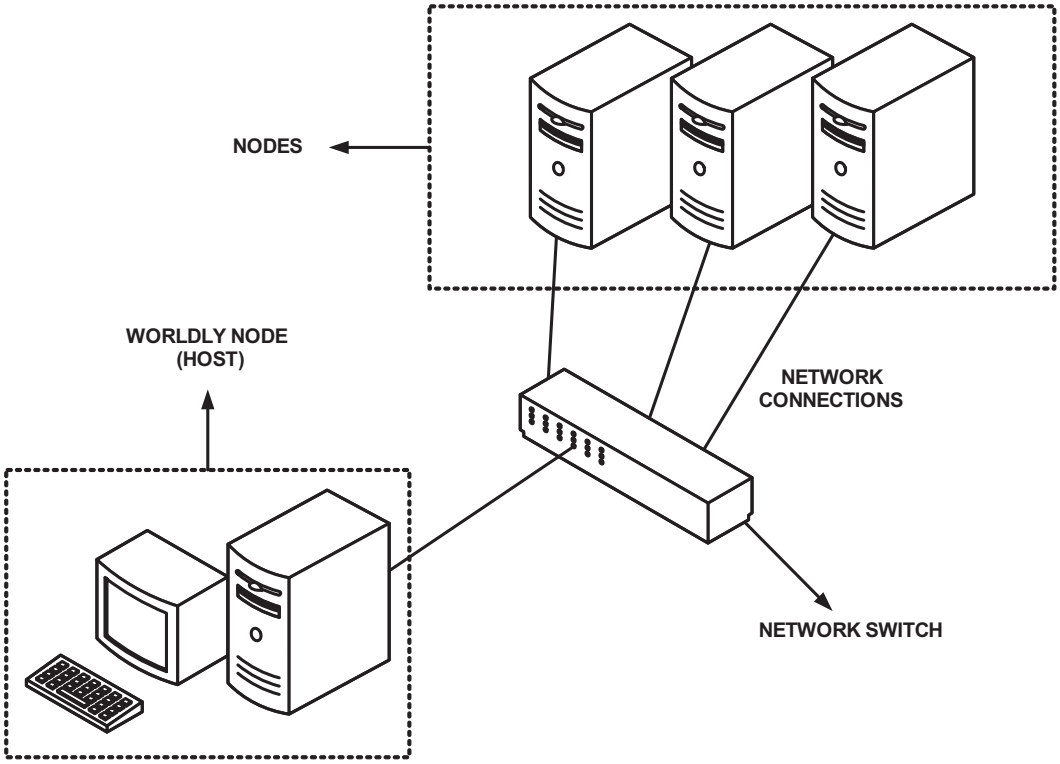

Figure 2.1: Typical Beowulf system

Worldly node shown in Fig. 2.1 is actually the host that is connected to the outside internet connection or local area network. Therefore to reach the Beowulf cluster, first you should connect to the worldly node and then connect to the other nodes of the cluster.

In our cluster, we compiled the software needed to run programs on the system with the best compiler optimizations which are advised by processor manufacturers.

2.1.2 Software Structure of 32-Bit Intel Pentium IV Cluster

In almost all Beowulf clusters, Linux based operating systems are installed. Linux supports many tools about network and cluster technology as default features in itself. Network information service (NIS) and network file system (NFS) [13] are the most important features of Linux operating system in order to install and operate in the parallel environment, the Beowulf parallel multicomputer.

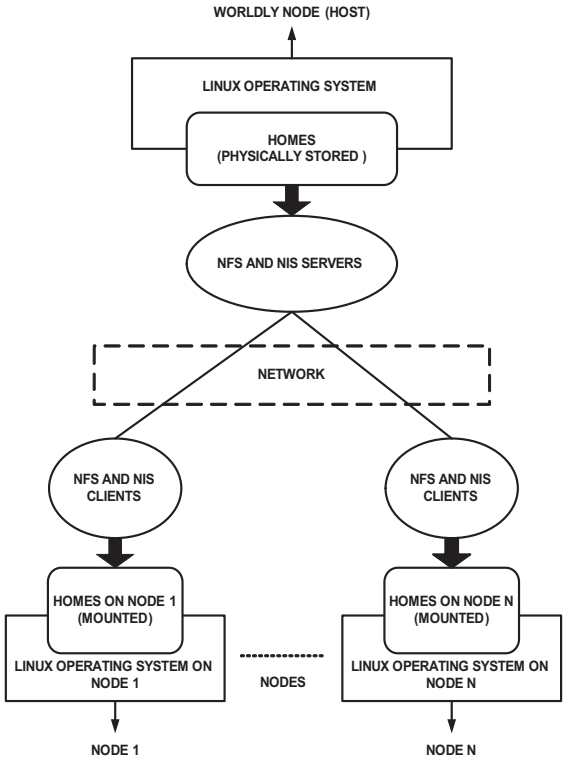

Figure 2.2: File system structure of a Beowulf cluster

As seen in Fig. 2.2, in a Beowulf cluster, users' directories are stored in the host. Despite all home directories being stored in the worldly node, other nodes which are connected to the worldly node can reach users' directories too, with the aid of NFS—this reaching is called as mounting in Linux and Unix operating systems terminology. The other service, NIS, actually supports the connection between the nodes of the cluster without asking for any permission; thus a user is able to directly connect from one node to another if he has an access to the worldly

node. For this reason, security of the connection from the outside world to the worldly node should be well configured; for instance many system administrators install firewalls to secure the connection of host to the external network.

For the execution of a message-passing parallel program, this file system architecture is a must but it is not solely enough to build up the parallel environment. In addition to that file structure, we need to install parallel environment programs to our system: message passing interface (MPI) implementation, scientific sequential and parallel libraries and compilers.

Message passing interface and compilers are a must, in order to code and run basic parallel applications. In our system, we use the local area multicomputer MPI (LAM/MPI) implementation as the MPI interface and Intel Fortran and C/C++ compilers are installed for compilation purposes. However, these software tools are not enough to parallelize and to run our parallel MLFMA program; we use certain parallel and scientific libraries, which are used extensively by the scientific computations community. These libraries make our life easier in order to program and develop applications and modules in our parallel program. Another advantage of scientific libraries are that these libraries are written by very professional groups and these groups give strong support in the usage of these libraries. Also, these libraries are provided for the usage after numerous trials and thus, most of the time, these libraries routines are more reliable for certain tasks than the routines that we write for the same application. In our parallel program, we used the following parallel libraries:

Basic linear algebra subroutines (BLAS) —we use these subroutines in basic matrix-vector and matrix-matrix operations.

Linear algebra package (LAPACK) is actually based on BLAS, and it supports more complicated linear algebra operations such as LU factorization [14] and etc.

AMOS library is a portable package for Bessel functions of a complex argument and nonnegative order.

Portable, extensible toolkit for scientific computation (PETSc) is used as a parallel iterative solver and preconditioner preparation tool in our parallel program.

All these tools are used in order to generate a parallel MLFMA executable:

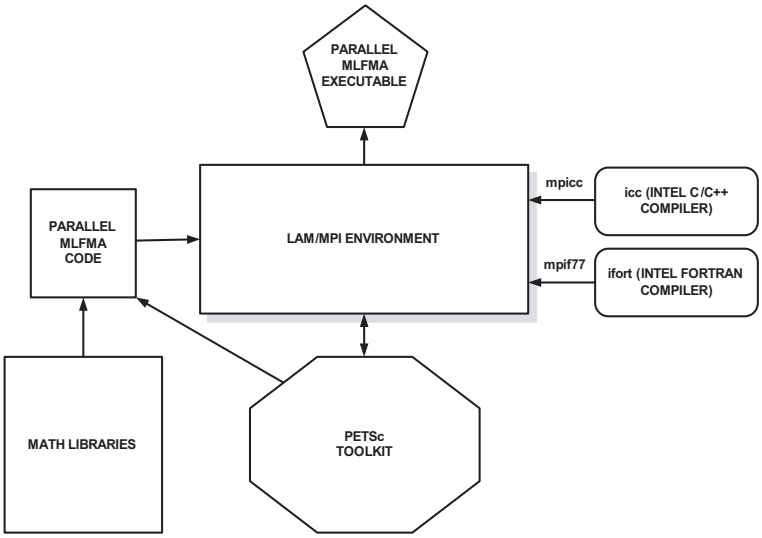

Figure 2.3: Generating a parallel MLFMA executable

After we get a parallel executable, we run our parallel program with the aid of LAM/MPI. LAM/MPI supports SPMD approach and it actually sends the parallel executable, which is compiled on host, to all other nodes. Then, when program execution starts, each processor does the task that is predefined for itself in this single executable code:

During the installation and especially in usage of the parallel libraries and interfaces, we have faced some problems. This situation occurs occasionally; because, we work on the cutting edge in the scientific problem solving. Thus we are the firsts to face these new problems in our country. Especially, the problems that we experienced are mainly based on the compatibility problems between parallel libraries, such as PETSc and Scalable LAPACK (ScaLAPACK)—parallel version of LAPACK—, and parallel implementations, LAM/MPI in our case. In spite of these problems, we worked continuously and have achieved a stable parallel software environment which give us the chance to improve and run our

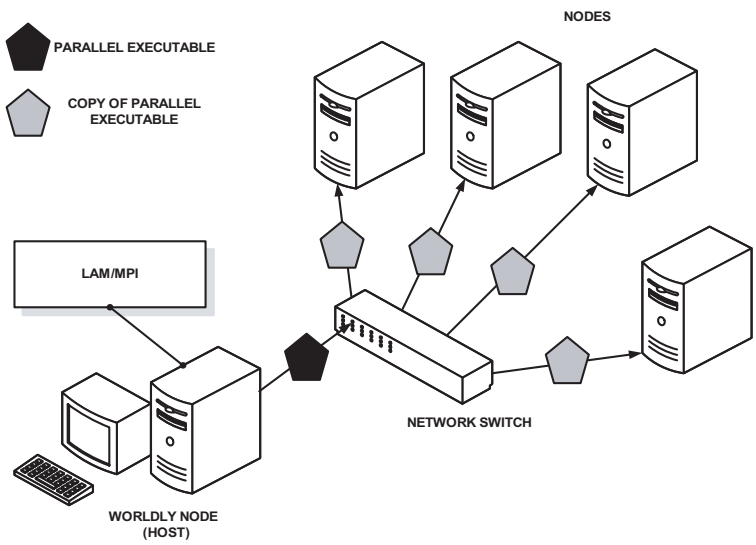

Figure 2.4: Run of a parallel program on homogenous Beowulf cluster

parallel programs.

2.2 Hybrid 32-Bit Intel Pentium IV and 64-Bit Intel Itanium II Cluster

While beginning our parallel programming approach, we assumed that we might assign some of the hard tasks of the parallel program on a special node of the cluster, which has more computational power and memory capacity. For this purpose, we decided to connect our newly bought multiprocessor system, which is based on 64-Bit Intel Itanium II processor architecture, to our homogeneous 32-Bit Intel Pentium IV cluster. This new multiprocessor system has two 64-Bit Intel Itanium II processors, which operates at 1.5 GHz and 24 GBytes of memory. Therefore, as it is mentioned before, this huge memory capacity and computational power might be successfully used in a master-slave approach in our continuous program development effort.

64-bit architecture of Intel Itanium is completely different from the normal 32-bit Intel Pentium based processors. As a result, assembly instruction set of this

new processor also differs from the instruction set of 32-Bit Intel Pentium based processors; so, Itanium multiprocessors need its own executable which is compiled specially for its instruction set. Thankfully, Intel company developed a special version of compilers which we used for generating the executable for our 32-bit machines; Intel Fortran and C/C++ compilers for the Itanium architecture are freely available for Linux operating environment and we installed those mentioned compilers on our multiprocessors system.

Difference between the architectures of processors that we deployed in our cluster system shifts our programming strategy to MPMD programming technique:

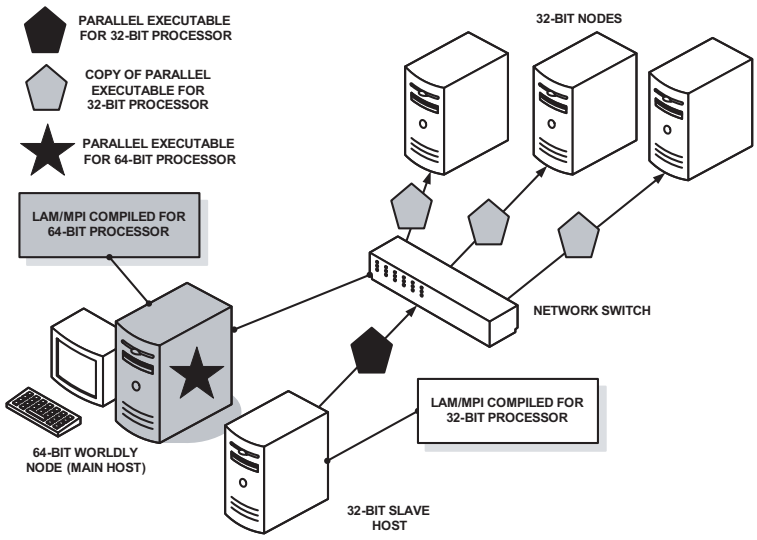

Figure 2.5: Execution of a parallel program on hybrid Beowulf cluster

As it can be seen in Fig. 2.5, we generate two different executables from the same parallel MLFMA code for two different processor architectures. We build the executable for Itanium machine on itself. On the other hand, we specially allocate one 32-bit node of parallel cluster in order to compile programs and to store parallel and sequential system tools for the rest of the 32-bit nodes. Depending on this methodology, parallel executables for 32-Bit nodes are generated on the mentioned special 32-bit node of the cluster, which can be named as a slave host—shown in Fig. 2.5.

In this new hardware architecture home directories are stored on the new

64-bit machine and users' homes are mounted by the nodes from these multi-processors. Nevertheless, as explained before, 32-bit machines need some system programs, which are compiled for their architecture and these system programs are stored on the 32-bit slave host. Other nodes of the cluster mount these system programs, which are compatible for 32-bit systems from this special node. All those mounting operations are done with the aid of NFS.

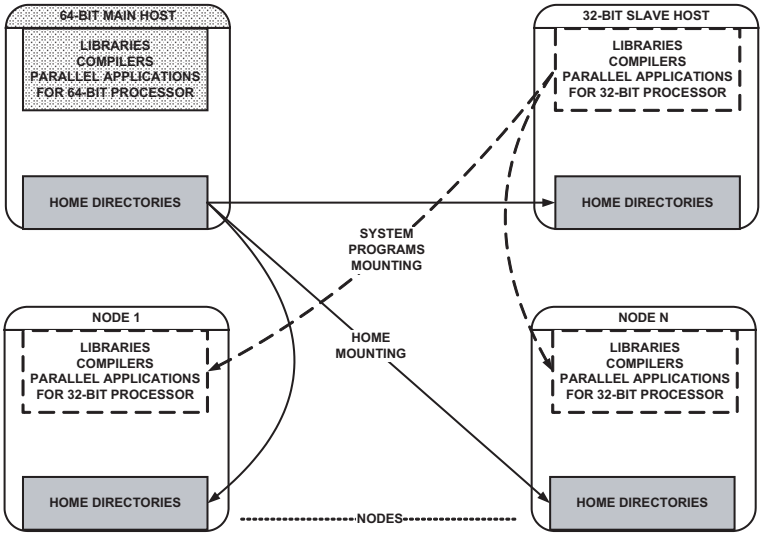

Figure 2.6: Mountings in the hybrid cluster

In the software topology, as depicted in Fig. 2.6, there are a lot of mounting operations done between computers. Since all these mountings use the network of the cluster, the system's overall network performance degrades. In addition, compiling different programs on different architectures and servers working on host computers, decrease the performance of the hosts. Moreover, as we continue to improve our parallel code, we experience that we do not need such a special computer for the tasks of our program. Therefore, we took out our 64-Bit Itanium multiprocessors from the cluster and returned to our homogeneous Beowulf architecture.

2.3 Proposed 64-Bit AMD Opteron Cluster

In the beginning of 2004, Advanced Micro Devices (AMD) released the new 64-bit processors for personal usage and computational purposes. These new processors support both 64-bit and 32-bit assembly instruction sets; therefore, AMD's processors give a very huge flexibility in many applications. In addition to this flexibility, Linux operating system developers always have sympathy for AMD processors; because Intel generally supports and advices to use Microsoft's operating system products on its products—Microsoft has been the natural rival of Linux since the beginning of Linux's first release. As a result, Linux developers released 64-Bit Linux operating systems for these new processors in a very short amount of time; so, AMD's cheap and powerful new processors serve as a good choice for Beowulf cluster computing solutions.

AMD produces many different versions of these new processors, but the most interesting and suitable processor for scientific computing is its Opteron architecture. AMD Opteron is specially designed for server applications and heavy scientific computing. It comes in three different models 1-way, 2-way and 8-way. 1-way model can only be used on single processor mainboards, 2-way can be used on single processor and double processors systems and 8-way can be used in a system which can support 8 AMD Opteron processors. In Turkey, only the 2-way version is sold. We obtained many positive feedbacks about this architecture and decided to build one such Opteron based system and compare its performance with the other computers that we have. Thus, we would be able to plan how to build our future cluster systems and decide which hardware components should be chosen in order to get optimum performance in our parallel program development efforts. For this purpose, we bought a 2-way AMD Opteron multiprocessors system, whose processors operate at 1.8 GHz, and 4 GBytes of memory installed in it. The system has a NUMA communication architecture; 2 GBytes of RAM is directly addressed by one processor and the remaining 2 Gbytes of memory is addressed by the other processor. Thus, when a sequential application needs memory greater than 2 GBytes, its execution speed decreases a little bit because of this NUMA architecture.

After buying this new system, we used the sequential version of MLFMA program as a benchmark tool and we compared performance of the computer systems in different parts of the sequential MLFMA program. The following hardware architectures are compared:

- Intel Prescott Pentium IV, 2.8 GHz single processor computer with 2 GBytes of memory
- AMD Opteron 244, 1.8 GHz dual multiprocessors computer with 4 GBytes of memory
- Intel Itanium II, 1.5 GHz dual multiprocessors computer with 24 GBytes of memory

In order to compare AMD Opteron multiprocessors with the other systems that we have, we especially focused on the performances of these machines in the two important parts of MLFMA: nearfield matrix $\overline{\mathbf{Z}}_{\mathbf{near}}$ filling module and matrix-vector multiplication module in iterative conjugate gradient stabilized (CGS) solver. These performance plots are given in Fig. 2.7 and Fig. 2.8, respectively:

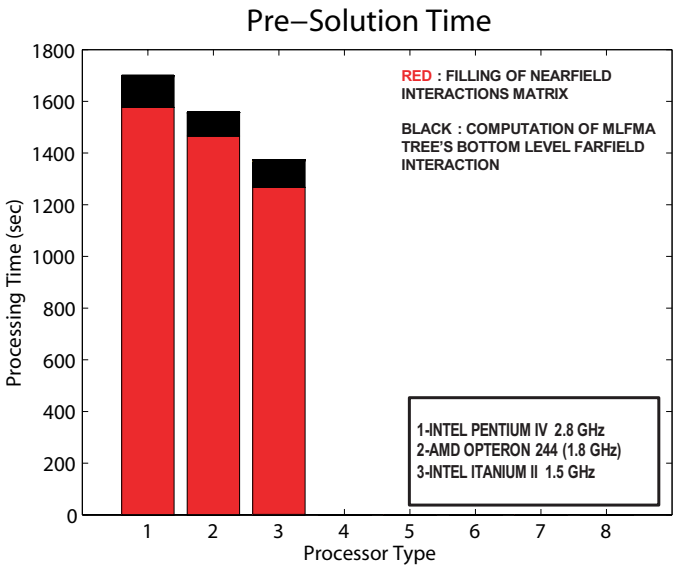

Figure 2.7: Performance comparison of different processors with pre-solution part of sequential MLFMA

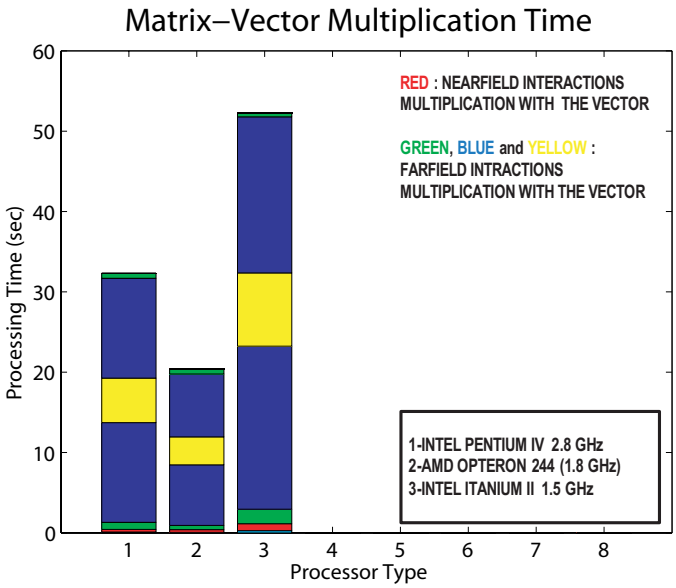

Figure 2.8: Performance comparison of different processors with matrix-vector multiplication in the iterative solver part of sequential MLFMA

These benchmark plots are obtained for the scattering solution of sphere at 6 GHz and this problem corresponds to the solution of a 132003 unknown full dense matrix. As can be seen from the first plot, Fig. 2.7, nearfield matrix filling performance of Itanium machine is the best; despite this, in the matrix-vector multiplications, AMD Opteron machine gives the best timing results. Actually, matrix-vector performance of machines is more important with respect to nearfield matrix filling operation; because, this multiplication is done many times in the iterative solution part of both sequential and parallel MLFMA. Moreover, as the problem size grows—number of unknowns increase, iteration numbers also increase, whereas only one nearfield matrix filling operation is done for the same problem. Therefore, if we want to solve larger problems, we have to choose a processor which performs better with respect to others in matrix-vector multiplications. As a result, we choose AMD Opteron dual multiprocessors as the nodes of our proposed cluster, which is going to be built in shortly.

2.4 ULAKBIM Cluster

Our cluster at Bilkent consists of four nodes. Despite this small cluster, our parallel program has been improved a lot by our group since 2002. In this improvement process, main goal is to build a parallel program which is scalable. For this reason, we wanted to test our program on a bigger cluster which has more processors. Fortunately, TUBITAK built a high performance cluster at ULAKBIM which has 128 nodes and we obtained access to this system and the system administrators of this cluster cooperated with us for our scalability measurements.

This cluster at ULAKBIM has 128 single processor machines which have Intel Celeron Pentium IV processor operating at 2.6 GHz and 1 GBytes of memory installed on all machines. We did certain parallel profiling measurements on those machines and from these measurements, we observed that the network speed of the cluster is more faster than ours. Since, we plan to build a new parallel computer and we had to make decisions on hardware components of this proposed cluster, we used our parallel program as a profiling tool in order to test and compare the speed of network connection at ULAKBIM's cluster, on our cluster and on the two processors of dual AMD Opteron multiprocessors. To achieve this aim, we did measurements on two processors of ULAKBIM's cluster and our cluster. In addition to that, we measured the two processor AMD Opteron by using our parallel MLFMA program. In Figs. 2.9-2.10, these benchmarking results are plotted:

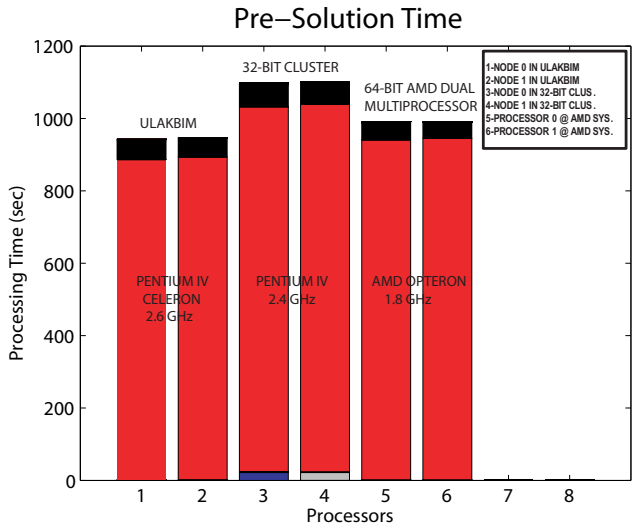

Figure 2.9: Performance comparison of different processors with pre-solution part of parallel MLFMA

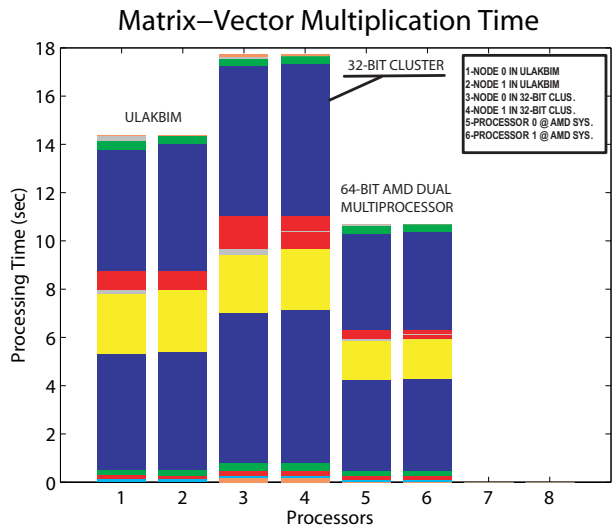

Figure 2.10: Performance comparison of different processors with matrix-vector multiplication in the iterative solver part of parallel MLFMA

From the figures, it can be easily observed that the performance of dual AMD multiprocessors is the best for matrix-vector multiplications. The red and gray parts on Fig. 2.10 actually shows the communications between processors in matrix-vector multiplications—red corresponds to the sending and receiving electromagnetic data, mainly translations; gray denotes idle time. As you can see from the graph, network performance of ULAKBIM is much better than our cluster's network connectivity, but it is still not as good as our AMD Opteron system. We expected this situation before we did our measurements; because AMD Opteron processors do their communication on the same motherboard; no network interface is used during these communications. On the other hand, in our cluster and at ULAKBIM, processors communicate through an ethernet based network. Actually ULAKBIM has a really fast ethernet system since they use gigabit ethernet interfaces and network switches in order to connect their computers. In spite of having a 100 Mbit/sec fast ethernet connection, here in our lab, it is ten times slower than ULAKBIM's gigabit connection. Depending on these measurements, we plan to use a gigabit network or a much faster network in our future cluster.

Chapter 3

Parallelization of MLFMA

3.1 Adaptation of MLFMA into Parallel System

Before starting to parallelize our MLFMA code, firstly, we tried to adapt and run the original sequential version of the program on our message-passing multicomputer system. Our original program is written in DIGITAL FORTRAN 77, which supports Cray pointers for the sake of dynamic memory allocation (DMA). These dynamically allocated memory structures increase the efficiency of memory usage. Nevertheless, the default FORTRAN 77 compiler, g77, in Linux systems does not support this feature; therefore, we could not compile the sequential MLFMA code directly and we could not run it on our Linux based parallel computer system.

For the solution of this problem, we primarily focused on the usage of open source software solutions and compilers, which are installed as default features in Linux based operating systems; we realized that open source Fortran and C/C++ compilers, g77 [15] and gcc, are compatible with each other and Fortran programs are able to call functions or features of C/C++ language by the aid of this compatibility. Then, we wrote small C subroutines which implements malloc (memory allocation) feature of this language and used them in Fortran trial programs. These trials were successful, and then we used these C subroutine in our sequential MLFMA code and it worked well. In spite of all these successful

trials, this new feature of C certainly complicates the array structure of our program. In addition, it also complicates the readability of our program. As a result of these facts, we started to look for another solution for this compilation problem.

At this point, we found freely available Intel Linux FORTRAN compiler, which supports many extensions in Fortran semantics and has a backward compatibility with the old FORTRAN distributions. We installed this free compiler in our system and compiled our parallel applications, such as MPI and parallel-sequential libraries, with that compiler. We followed that way, because we do not want to face any compatibility problems because of different compilations of these system programs. After all these, we compiled the sequential version of MLFMA successfully and executed it on a single node of the parallel cluster. Then, we put this sequential version of MLFMA as a single parallel process in a simple parallel program and executed it as such and we realized that we could start to parallelize our code. Following chapters are going to give the details of this parallelization process.

3.2 Partitioning Techniques

Message-passing parallelization paradigm is actually based on partitioning of data structures among processors. Most of the time, processes do similar tasks on those partitioned data and depending on the type of the problem or computation, they interchange their partitioned or processed data. These partitioned data structures are generally arrays or array-based data structures. One-dimensional arrays and two dimensional arrays are the most commonly used array architectures. For the sake of parallelization, these array structures are firstly partitioned; then, these partitions are mapped onto the processors depending on the criteria of the parallelization approach e.g., minimizing communication, equalizing memory usage, and etc. Arrays can be partitioned as shown in Fig. 3.1:

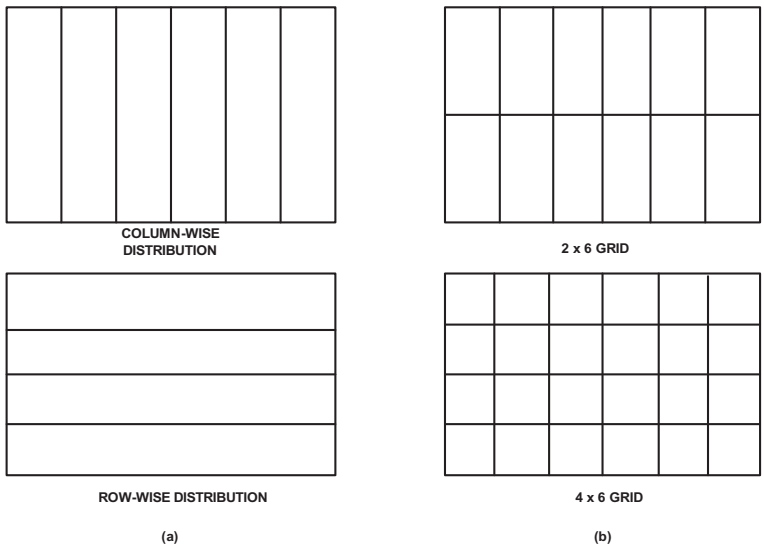

Figure 3.1: (a) One-dimensional partitioning of an array (b) two-dimensional partitioning of an array

In Fig. 3.1, common array partitioning methodologies are shown. These partitioned arrays can be distributed among the processor statically or dynamically, depending on the application or problem. We used statical mapping of these arrays on our processors, because, we know the sizes of the arrays that we need for our electromagnetic data and we try to minimize the communication over processors. For these reasons, in our program, array load of each cluster is determined in the beginning of the code.

In our MLFMA code, all arrays related with the matrix information of our problem, $\overline{\mathbf{Z}}_{\mathbf{near}}$ and $\overline{\mathbf{Z}}_{\mathbf{far}}$ matrices, which need a lot of memory in order to be stored, are one-dimensional arrays. Therefore , partitioning of these arrays among processors is not trivial; because of that reason, we started our parallelization efforts from the relatively easily parallelized parts of our program.

3.3 Partitioning in Parallel MLFMA

We mentioned that our strategy for the parallelization of MLFMA code is to parallelize relatively simple parts at the beginning; then, we decided to move on to more complex parts of the program. According to our point of view, the first parallelized part of MLFMA is the filling of preconditioner matrix [16], which is used in the iterative CGS solver part of the program. After parallelization of this part, we realized that the partitioning applied on the data structure of preconditioner can be used to partition the whole tree structure of MLFMA among the processors.

3.3.1 Partitioning of Block Diagonal Preconditioner

In the iterative solution techniques, preconditioners are used in order to get a better solvable matrix equation system. Preconditioners are actually matrices, which are very similar to the inverse of the original matrix. In our MLFMA code, the preconditioner, $\overline{\mathbf{M}}$, is extracted from the near-field matrix, $\overline{\mathbf{Z}}_{\mathbf{near}}$, of our matrix system and then it is applied to the matrix system:

$$\overline{\mathbf{M}}^{-1}.\overline{\mathbf{Z}}.\mathbf{x} = \overline{\mathbf{M}}^{-1}.\mathbf{y} \tag{3.1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are the unknown and excitation vectors, respectively.

As stated before, $\overline{\mathbf{M}}$ is extracted from $\overline{\mathbf{Z}}_{\mathbf{near}}$. Actually, $\overline{\mathbf{M}}$ contains the most powerful terms of $\overline{\mathbf{Z}}_{\mathbf{near}}$ matrix, which are the self interaction terms of the induced electromagnetic currents grouped in the last level clusters of MLFMA tree structure—self interactions refer to the near-field interactions of clusters in $\overline{\mathbf{Z}}_{\mathbf{near}}$ with themselves. These interaction terms are placed as blocks in the diagonal of near-field matrix; because of this placement, the preconditioner is named as block diagonal preconditioner (BDP). In order to build our preconditioner, we take out these blocks and treat each of them as a normal matrix system: LU factorization is applied on each of these blocks and the inverses of them are taken. LAPACK subroutines, ZGETRF and ZGETRI [17] are used in these operations,

respectively. Finally inverses of these blocks are written in the memory in order to built the preconditioner matrix:

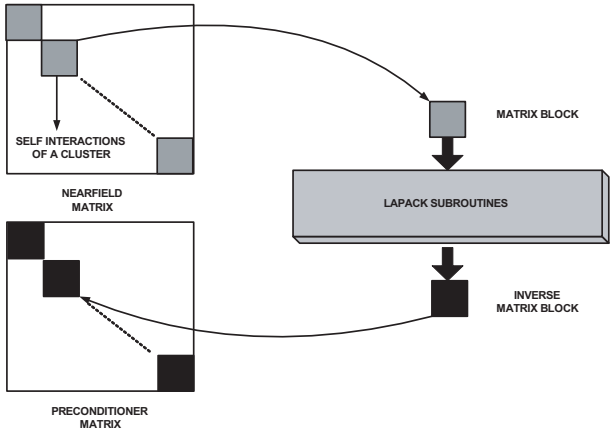

Figure 3.2: Extraction of preconditioner matrix

In parallelization, each of these clusters' self interaction blocks are partitioned among processors, equally; so, each of the processor gets equal number of blocks according to this partitioning. For this purpose near-field matrix's corresponding blocks are row-wise partitioned and then partitioned blocks are assigned to a processor consecutively as it is shown in Fig. 3.3:

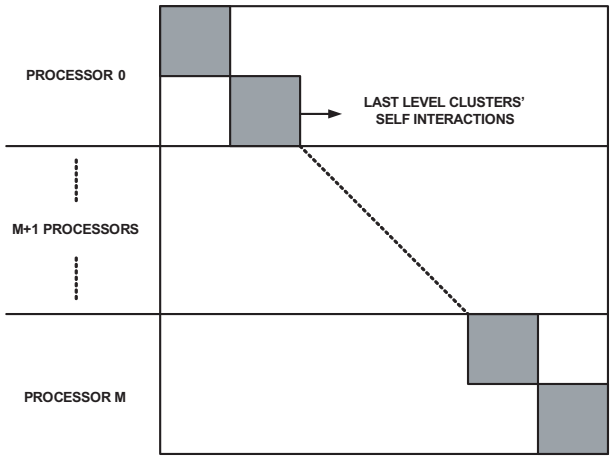

Figure 3.3: Partitioning of preconditioner matrix

3.3.2 Partitioning of MLFMA Tree Structure

It is stated before that MLFMA is actually based on the grouping of electromagnetic interactions in clusters and those clusters are re-grouped in bigger clusters recursively. As a result, we get a tree structure. In that tree structure, bigger clusters have direct electromagnetic relations with the small clusters, which are in its group. For that reason, a cluster, which contains a small cluster, is called as parent of that cluster and the small cluster is called as child of the bigger cluster. Thus, MLFMA's far-field operations—aggregations, disaggregations and translations [11]— and near-field matrix filling operations are done on this hierarchical tree structure. Actually the near-field matrix, $\overline{\mathbf{Z}}_{\mathbf{near}}$, is computed by last level cluster's electromagnetic interactions; because in order to compute the near-field matrix, we should interact each discretized induced electromagnetic current with its near-field neighbor directly according to the MoM. Hence near-field interactions are calculated depending on this approach, we do not need to do any computation for near-field interactions of the upper levels clusters in MLFMA tree.

In the parallelization of BDP, last level clusters of the MLFMA tree are partitioned among the processors in order to assign equal number of last level clusters in each of the processors. Partitioning of MLFMA is based on this basic partitioning: We simply start to put last level clusters' parents on the processor where their children are deployed, and this strategy is applied from the last level, bottom level of clusters, to the top level. This situation is depicted in Fig. 3.4:

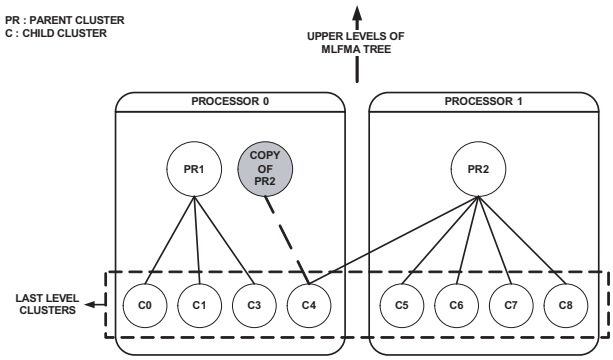

Figure 3.4: Partitioning of MLFMA tree

This partitioning strategy minimizes the communications in far-field matrix-vector product, because, many of the computations about child-parent cluster pairs can be done independently without requiring any information from other processors for aggregation and disaggregation stages of MLFMA. Nevertheless, from experience and profiling studies, near-field computations are not very well distributed over the processors in the partitioning strategy; moreover memory balance for the far-field interaction computations is not very good. As a result of these artifacts, we started to study load balancing techniques and features to balance the load of computations and memory usage over the processors for the mentioned MLFMA parts, near-field and far-field computations.

3.4 Load Balancing in Parallel MLFMA

Load balancing is a very distributed and important issue in parallel computations. By the load balancing techniques and algorithms, memory and workload [18] of a parallel application's processes can be distributed equally over the parallel system and this balance operation increases the overall performance of the parallel code. In addition to these load balancing approaches, communications might be balanced with special methods. Depending on the importance of one of these features, the parallel programmer focuses on a load balancing technique, which is suitable for his application. Nevertheless, it must be noted, most of the time that one load balancing technique applied for one aspect of parallelization may harm the balance of another aspect. For instance, a load balancing technique which is suitable for memory balancing might degrade the balance of computational time over the nodes and it might increase the idle time.

In our load balancing approach, our primary goal is to balance the memory allocation for the near-field and far-field parts of the program. In our MLFMA program, far-field and near-field computations are almost calculated independently. Therefore, we decided to apply different cluster partitioning for these parts of the program by using their memory requirement information. Our load balancing algorithm is based on the famous bin-packing algorithm [19] and we

applied this algorithm for the near-field and far-field parts of the parallel MLFMA program separately.

According to this algorithm, we primarily calculate the number of memory packets, whose size is M_i, that are going to delivered to the processors and this memory size information is written in a memory size list L_M. Then, those memory packets are summed and total amount of memory allocated over the whole system is computed, which is abbreviated with M_{total}.

$$M_{total} = \sum_{i=1}^{N} M_i, \tag{3.2}$$

where N is number of memory packets. Then, we compute the average amount of memory, $M_{average}$ that should be delivered to each processor:

$$M_{average} = \frac{M_{total}}{p}, \tag{3.3}$$

where p is number of processors.

After all these memory computations, we start to assign the memory packets to the processors by using the load balancing algorithm. We trace the memory length information from the beginning of the memory packet size list, L_M, and calculate a deviation for this case, $deviation_{total}^{forward}$. Then, we trace the same memory packets size list, L_M, from the end of the list and compute its deviation, which is $deviation_{total}^{backward}$. These deviations are the summation of absolute differences between the processor memory load and average memory that they should take:

$$deviation_{total}^{forward} = \sum_{i=1}^{p} deviation_{proc_i}^{forward} \tag{3.4}$$

$$deviation_{total}^{backward} = \sum_{i=1}^{p} deviation_{proc_i}^{backward}, \tag{3.5}$$

where $proc_i$ is the processor number and p is the number of processors. $deviation_{proc_i}^{forward}$ and $deviation_{proc_i}^{backward}$ are calculated as follows:

$$deviation_{proc_i}^{forward} = |M_{average} - M_{proc_i}^{forward}| \tag{3.6}$$

$$deviation_{proc_i}^{backward} = |M_{average} - M_{proc_i}^{backward}|, \tag{3.7}$$

where $deviation_{proc_i}^{forward}$ is the deviation of the memory assigned on processor $proc_i$, $M_{proc_i}^{forward}$, from $M_{average}$ in the forward load balancing and similarly $deviation_{proc_i}^{backward}$ is deviation of the memory assigned on processor $proc_i$, $M_{proc_i}^{backward}$, from $M_{average}$ in the backward load balancing.

Backward load balancing is exactly similar to this algorithm, except that we start to put the memory packets size information from the back of the L_M, as mentioned before. We apply both of these methods for the load balancing of near-field related arrays partitioning and far-field related arrays partitioning. In the end, by comparing the deviations for forward and backward load balancing, $deviation_{total}^{forward}$ and $deviation_{total}^{backward}$, we chose the most appropriate partitioning for the related parallel MLFMA part. As an example, for the near-field matrix filling part, if $deviation_{total}^{backward}$ is less than $deviation_{total}^{forward}$, we apply the backward load balancing partitioning map to this part and vice versa.

Details of forward memory load balancing and backward memory load balancing algorithms are given in Fig. 3.5 and Fig. 3.6 respectively:

1: **while** There remains a processor to get memory packet **do**
2: Initialize the total forward deviation to 0, $deviation_{total}^{forward} = 0$
3: Put a memory packet onto the available processor from the beginning of memory packet list, L_M
4: Compute total amount of memory, M_{total}^{prev},
before the last memory packet is added
5: Compute total amount of memory, M_{total}^{last},
after the last memory is added
6: **if** $M_{total}^{last} \geq M_{average}$ **then**
7: Calculate deviation of total memory after putting last packet on the *proci*,
$deviation = |M_{total}^{last} - M_{average}|$
8: Calculate deviation of total memory before putting last packet on the *proci*,
$deviation_{prev} = |M_{total}^{prev} - M_{average}|$
9: **if** $deviation \geq deviation_{prev}$ **then**
10: *proci* memory capacity is full,
take the previous memory packet as the last packet of *proci*
11: Calculate total deviation, $deviation_{total}^{forward} = deviation_{total}^{forward} + deviation_{prev}$
12: **else**
13: *proci* memory capacity is full,
take the last memory packet as the last packet of *proci*
14: Calculate total deviation, $deviation_{total}^{forward} = deviation_{total}^{forward} + deviation$
15: Update the available processor
16: Initialize the total memory to informations for the available processor as 0, $M_{total}^{last} = 0$ and $M_{total}^{last} = 0$

Figure 3.5: Structure of forward load balancing algorithm for the memory

1: **while** There remains a processor to get memory packet **do**
2: Initialize the total backward deviation to 0, $deviation_{total}^{backward} = 0$
3: Put a memory packet onto the available processor from the end of memory packet list, L_M
4: Compute total amount of memory, M_{total}^{prev},
before the last memory packet is added
5: Compute total amount of memory, M_{total}^{last},
after the last memory is added
6: **if** $M_{total}^{last} \geq M_{average}$ **then**
7: Calculate deviation of total memory after putting last packet on the *proci*,
$deviation = |M_{total}^{last} - M_{average}|$
8: Calculate deviation of total memory before putting last packet on the *proci*,
$deviation_{prev} = |M_{total}^{prev} - M_{average}|$
9: **if** $deviation \geq deviation_{prev}$ **then**
10: *proci* memory capacity is full,
take the previous memory packet as the last packet of *proci*
11: Calculate total deviation,$deviation_{total}^{backward} = deviation_{total}^{backward} + deviation_{prev}$
12: **else**
13: *proci* memory capacity is full,
take the last memory packet as the last packet of *proci*
14: Calculate total deviation, $deviation_{total}^{backward} = deviation_{total}^{backward} + deviation$
15: Update the available processor
16: Initialize the total memory to informations for the available processor as 0, $M_{total}^{last} = 0$ and $M_{total}^{last} = 0$

Figure 3.6: Structure of backward load balancing algorithm for the memory

3.4.1 Load Balancing of Near-field Interactions Related Operations and Structures

It is mentioned that near-field computations are done on the last level clusters of the MLFMA tree structure. These clusters actually store information about the discretized induced electromagnetic currents, which are called basis and testing functions [20], over the surface of problem geometry. Depending on the near-field interactions criteria, each basis in a last level cluster interact with each testing function of another last level cluster if those clusters are in the near field of each other. Therefore, if two clusters, which are in the near field of each other, have n basis and m testing functions, respectively, first cluster do nm near-field interactions with the second cluster and second cluster also do same amount of near-field interactions with the first cluster.

In our program, we must store all the near-field interaction data in complex arrays and after reading the problem's geometry information we can calculate the size of those arrays by just adding the near-field interaction numbers. Therefore, by using this info, we can fill the memory packet size information list and we are able to apply both forward and backward load balancing for near-field arrays. Hence, we implemented that approach; the results are given in the next chapter.

3.4.2 Load Balancing of Far-field Interactions Related Operations and Structures

In far-field load balancing, we also use the same load balancing algorithms. In this case, the memory information calculations are just changed with respect to the near-field case. In far-field calculations, all MLFMA tree structure is used in order to compute far-field interactions between the clusters, so we must store that tree structure in the arrays. After reading the problem geometry information, the tree structure is formed and the amount of memory needed for that tree structure can be calculated easily by traversing the tree. Thus, in order to apply load balancing in this tree structure, we deal with the memory information of the

clusters in different levels.

We apply load balancing for the far-field tree structure as follows: First, we choose a level in the MLFMA tree, then we open an array which is in the length of the number of clusters in that level; we use that array as our memory size information list, L_M. Then, we calculate the memory need of all the children and the parent itself for each cluster in the chosen level. Finally, the sizes of memory requirements for each children cluster and parent itself are superposed for each parent cluster on the chosen level; thus, we have the memory requirement for each branch in the chosen partitioning load balancing level. By this methodology, we partition the memory needed for far-field computations more equally. Memory information reading is depicted in Fig. 3.7:

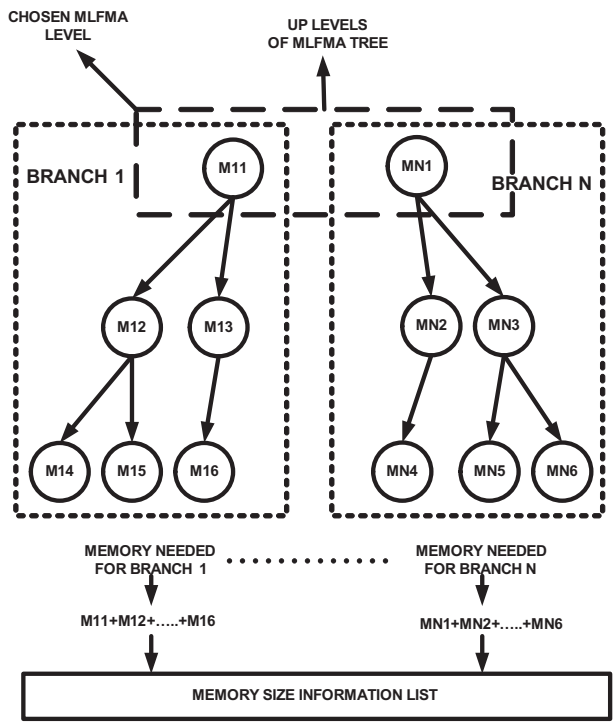

Figure 3.7: Memory calculations for far-field related arrays

As we did in near-field load balancing, we use the far-field array size information list in order to partition the far-field arrays among the processors of clusters by using the algorithm in Fig. 3.5. The results of this approach is also given in detail in Chapter 4.

Chapter 4

Results of Load Balancing Techniques

In this chapter, profiling and scaling measurements of two different scattering problems are presented. A perfectly electric conducting (PEC) sphere and a PEC generic helicopter model are used in the measurements of our parallel MLMFA solver. These measurements on the given geometries are done with two different versions of the parallel MLFMA program. The first version uses the old partitioning algorithm, which does not contain any load balancing approach and it partitions the last level clusters of MLFMA tree among the processors equally. On the other hand, the second version of the program that we used partitions these clusters by using forward and backward load balancing algorithms. These algorithms are separately applied on near-field matrix related parts ($\overline{\mathbf{Z}}_{\mathbf{near}}$) and far-field computations ($\overline{\mathbf{Z}}_{\mathbf{far}}$). As a result, two different partitionings are applied on the geometries according to the memory requirement information related to near-field and far-field computations respectively. In the presented results, first, time and memory profiling measurements for these geometries are given and then, speedup, memory gain, time and memory scaling results are shown. Effects of the load balancing of near field and far field related memory structures on our parallel MLFMA program can be easily seen from these presented plots and graphs.

4.1 Sphere Geometry

Scattering problem from a PEC sphere geometry was used in our primary scaling and profiling studies. In this problem, an external electromagnetic wave is used to illuminate the PEC sphere, then the scattered electromagnetic field after this illumination is simulated with our parallel MLFMA program. Scattering from a PEC sphere geometry is depicted in Fig. 4.1:

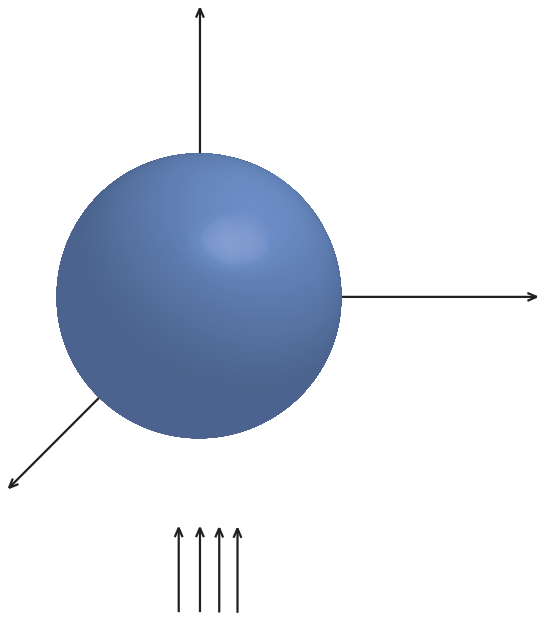

Figure 4.1: Electromagnetic scattering problem from a PEC sphere

Profiling and scaling measurements are done on 7 different meshes of the given sphere geometry, which are tabulated in Table 4.1:

Sphere Geometry Information		
Mesh Size	Number of Unknowns	Solution Frequency
1.50 mm	1,462,854	20.0 GHz
1.75 mm	1,083,282	17.5 GHz
2.00 mm	829,881	15.0 GHz
2.50 mm	528,786	12.0 GHz
3.00 mm	367,821	10.0 GHz
4.00 mm	206,499	7.5 GHz
5.00 mm	132,003	6.0 GHz

Table 4.1: Sphere geometries, which were used in the profiling measurements

During our measurements on TUBITAK's high performance parallel computer system at ULAKBIM, we did parallel runs with these sphere meshes given in Table 4.1. These parallel solutions were computed with the changing number of processors on this parallel system—from 32 processors to 1 processor.

4.1.1 Time and Memory Profiling Results

In this part, the effects of memory load balancing techniques on near-field and far-field related parts of our parallel program are depicted graphically. In these graphs, we show the measurement results for 1.5 mm mesh sphere problems which were solved on 32 processors of ULAKBIM high performance parallel computer system. The solution of this sphere mesh corresponds to 1,462,854 unknowns dense matrix solution. Other graphs for different sphere meshes are not given, because the behavior of our parallel solver in the solution of these meshes are oxymoron.

First, we give the memory distribution of plots for near-field and far-field related arrays, and then, peak memory usage plots of the modules of our parallel MLFMA program during its execution are given. In the near-field related array memory plots, Figs. 4.2-4.3, blue color corresponds to the memory requirement used to store the near-field matrix and black color corresponds to the memory requirement to store the preconditioner. On the other hand, in far-field related memory plots, Figs. 4.4-4.5, orange and cyan colors correspond to the memory requirement of arrays to store far-field radiation patterns of basis functions and green color corresponds to the memory need to store the electromagnetic data of clusters in MLFMA tree structure. In Figs. 4.6-4.7, each color corresponds to the peak memory usage of a processor during the execution of the parallel MLFMA code. Moreover in Figs. 4.6-4.7, vertical axis shows the modules of MLFMA and horizontal axis shows the peak memory usage at the corresponding module of the parallel program.

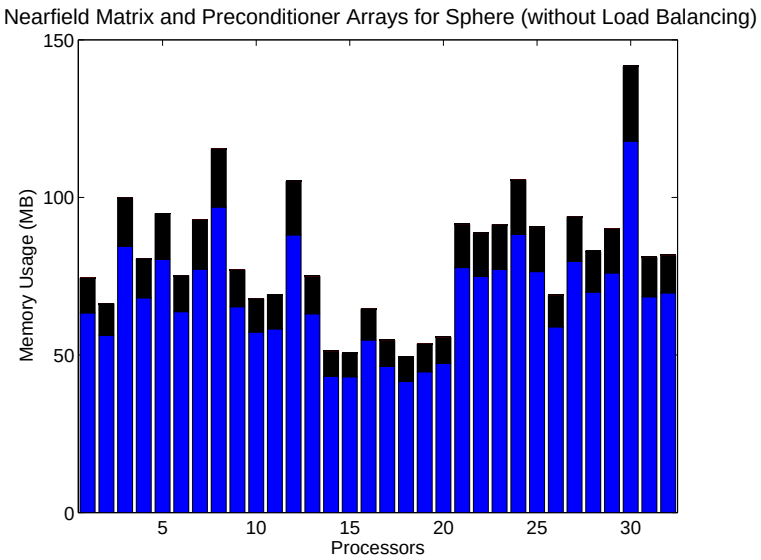

Figure 4.2: Memory usage of near-field interactions related arrays for sphere with 1.5 mm mesh when load balancing algorithms are not applied

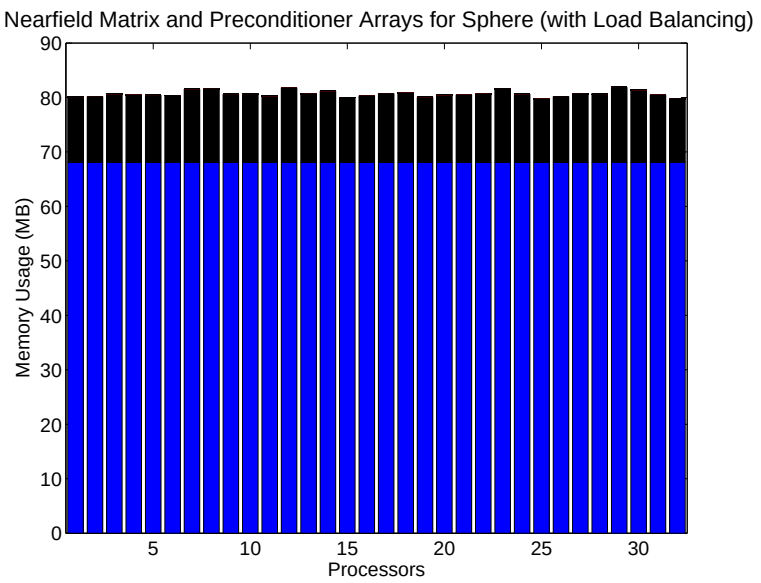

Figure 4.3: Memory usage of near-field interactions related arrays for sphere with 1.5 mm mesh when load balancing algorithms are applied

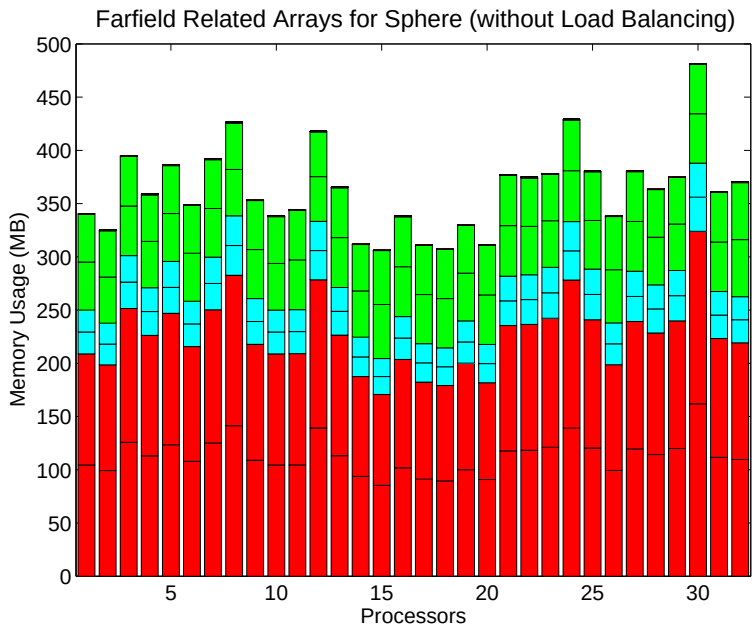

Figure 4.4: Memory usage of far-field interactions related arrays for sphere with 1.5 mm mesh when load balancing algorithms are not applied

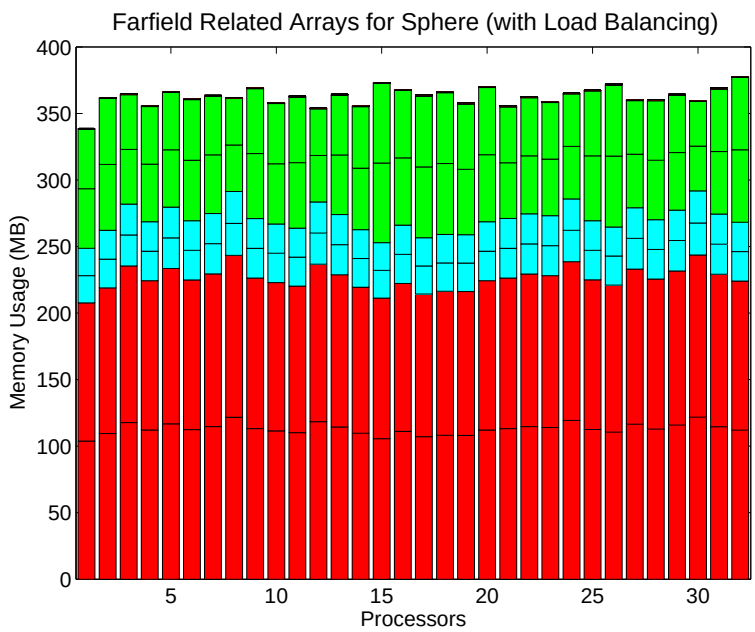

Figure 4.5: Memory usage of far-field interactions related arrays for sphere with 1.5 mm mesh when load balacing algorithms are applied

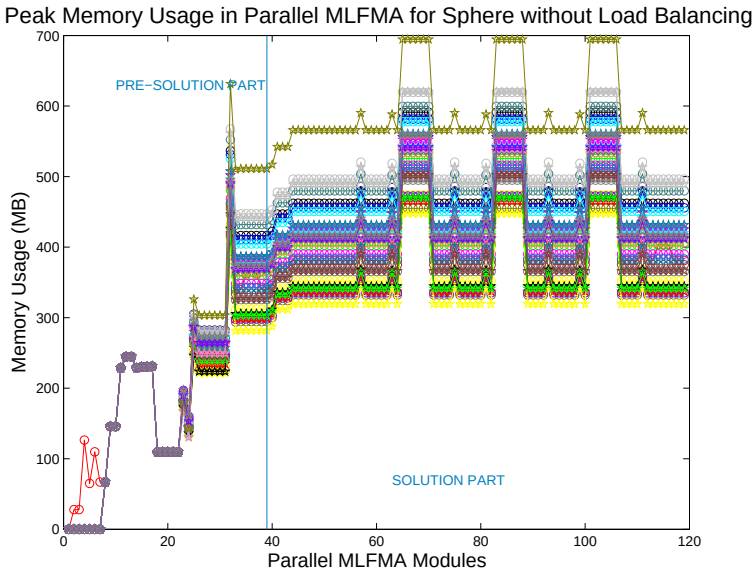

Figure 4.6: Peak memory usage in the modules of MLFMA for sphere with 1.5 mm mesh when load balancing algorithms are not applied

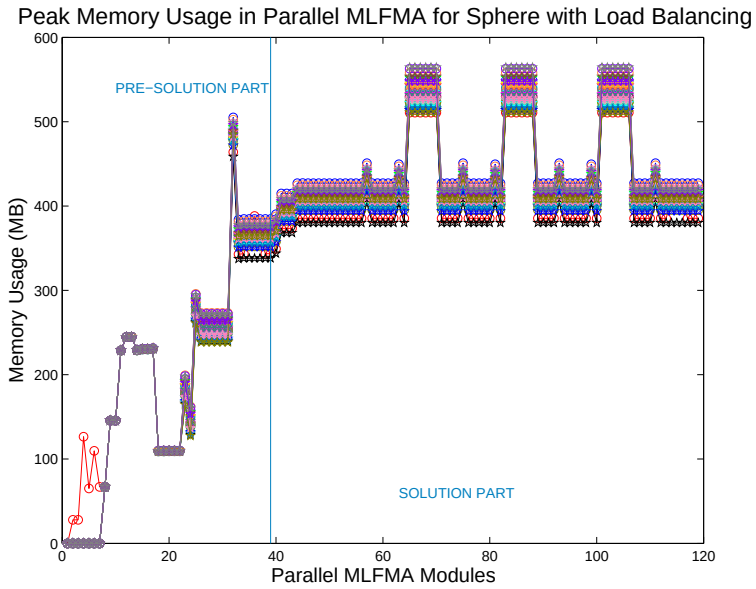

Figure 4.7: Peak memory usage in the modules of MLFMA for sphere with 1.5 mm mesh when load balancing algorithms are applied

As can be seen from Figs. 4.2-4.5, memory load balancing algorithms for near-field and far-field related arrays work properly and supports the balance of memory usage load over 32 processors for this problem. This situation is actually observable from Figs. 4.6-4.7 too; when load balancing for the memory is properly applied, the gap between the peak memory usages of processors decreases.

In the following figures, we will show the effects of memory load balancing on computations of near-field interactions and matrix-vector multiplication operations: In Figs. 4.8-4.9, the effect of near-field related memory load balancing on the computations of near-field matrix filling operations is shown and in Figs. 4.10-4.11, far-field related memory load balancing effect on the matrix-vector calculations are shown. Matrix-vector multiplication time profilings are especially presented; because, far-field related computations, aggregations, disaggregations and translations, are done in this stage of our parallel MLFMA solver.

Color map for near-field related time profilings are as follows: Red corresponds to the time required to fill the near-field matrix, $\overline{\mathbf{Z}}_{\mathbf{near}}$, black corresponds to the time required to calculate the radiation patterns of basis functions and colors below red region correspond to the reading of geometry information, distribution of this information and clustering parts of the parallel MLFMA program. On the other hand, main colors seen in the matrix-vector operations are orange, blue, yellow, red and gray, respectively: Orange corresponds to the initialization of matrix-vector operation, blue color corresponds to aggregation and disaggregation stages consecutively, yellow corresponds to the translation operations which can be done by the processor on itself without communication, red corresponds to interprocessor translations and gray shows the idle time.

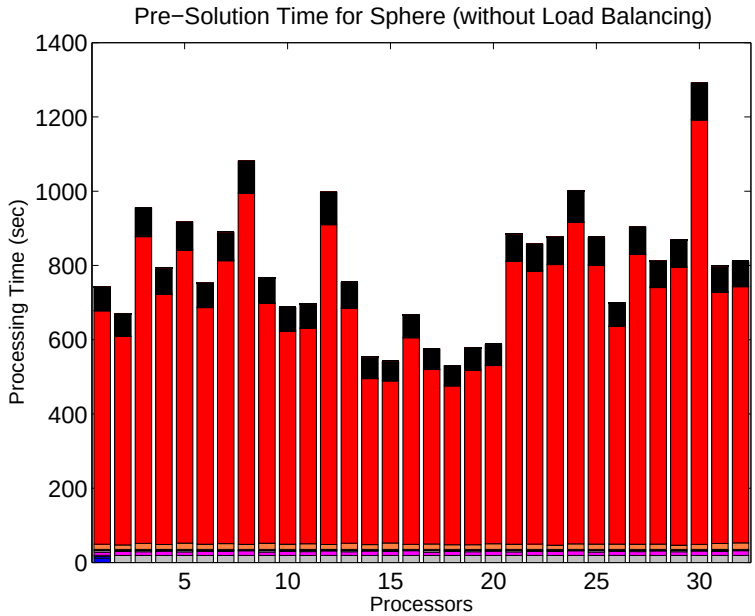

Figure 4.8: Pre-solution time for sphere with 1.5 mm mesh when load balancing algorithms are not applied

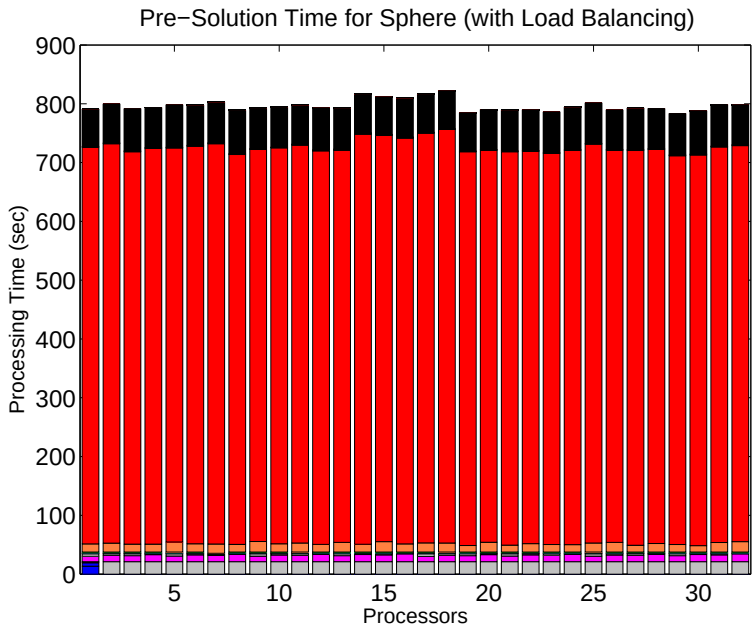

Figure 4.9: Pre-solution time for sphere with 1.5 mm mesh when load balancing algorithms are applied

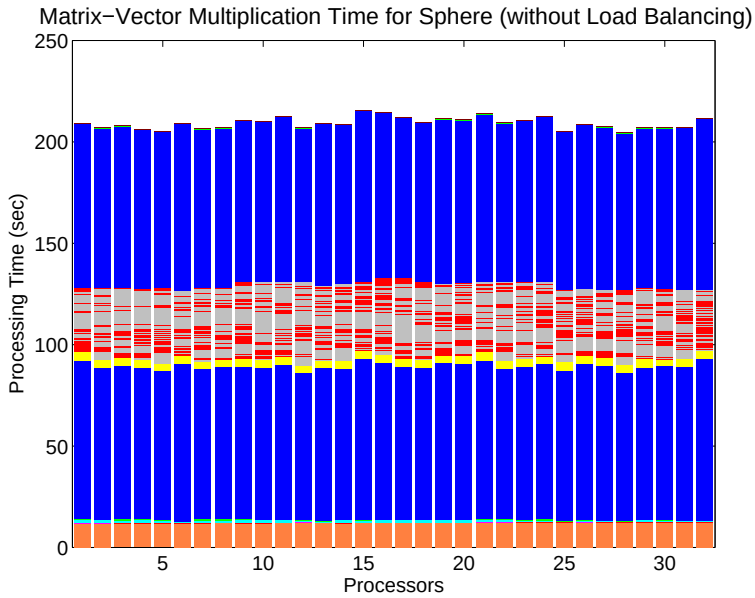

Figure 4.10: Matrix-vector multiplication time profiling for sphere with 1.5 mm mesh when load balancing algorithms are not applied

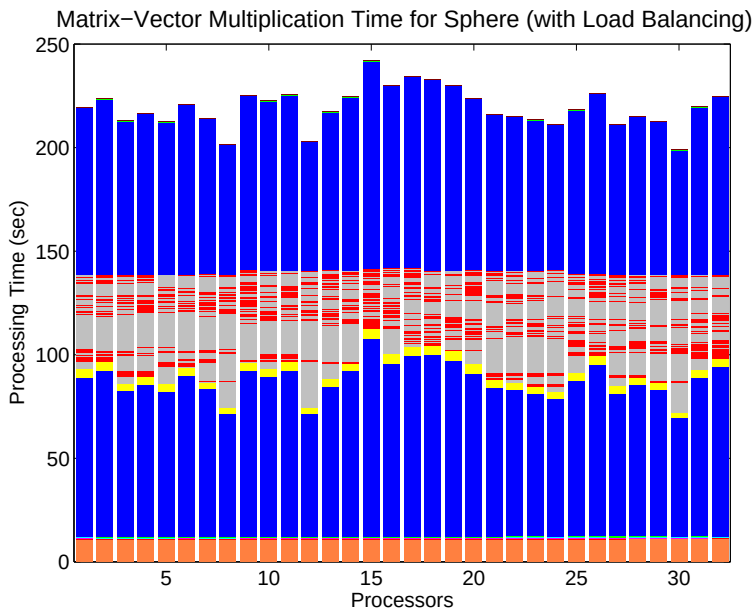

Figure 4.11: Matrix-vector multiplication time profiling for sphere with 1.5 mm mesh when load balancing algorithms are applied

As seen from Figs. 4.8-4.9, near-field related memory load balancing also balances the computational time requirement in order to fill near-field matrix, $\overline{\mathbf{Z}}_{\mathbf{near}}$. This situation actually depends on the linear proportion between storage requirements and filling operations of this matrix. Therefore, by balancing the near-field matrix memory, we also balance the near-field interactions' computations. Despite this positive effect of near-field memory load balancing on near-field related computations, in far-field related calculations, memory load balancing algorithm has a negative impact on the far-field related operations, which are done in matrix-vector multiplications. This situation is depicted in Figs. 4.10-4.11. In order to balance the computational time of matrix-vector multiplications and memory storage need for this part, we shift our focus on different balancing approaches.

4.1.2 Speedup, Memory Gain, and Scaling Results

In this part, with load balancing and without load balancing approaches are compared by using the mentioned parallel measurements metrics, which are speedup, memory gain, time and memory scale plots. In these metrics, we use three sphere solutions: 5 mm mesh sphere, 4 mm mesh sphere and 2.5 mm mesh sphere geometries. 5 mm mesh problem is the biggest problem that can be solvable on a single processor computer with our MLFMA code; as a result, the solution of this problem on a single computer system is used as the reference for speedup and memory calculations. In order to test the scalability of our parallel code, we use 4 mm mesh and 2.5 mm mesh sphere problems, which are the biggest problems that are solvable on 4 processors and 8 processors respectively. In our speedup and memory gain plots, we give the speedup and gain curves, which use 4 processors' and 8 processors' solutions as reference. Moreover, in these gain and speedup plots, gain and speedup curves which depend on 4 processors' solution and 8 processors' solution are normalized with respect to curves, which uses 1 processor solutions as reference. To normalize these curves with respect 1 processor's solution curve, we multiply the data points of 4 processors' and 8 processors' curves with 4 and 8 respectively.

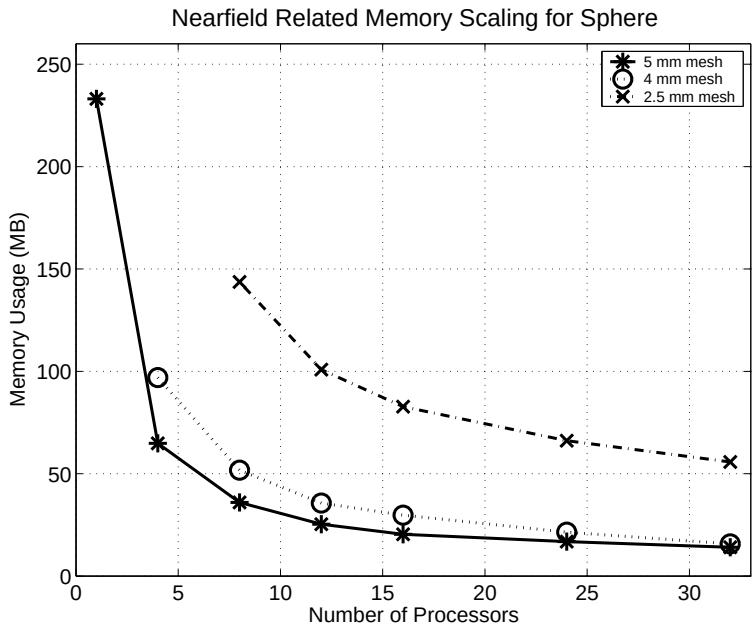

Figure 4.12: Memory scale plots of three different sphere meshes for array structures, which are related with near-field computations (load balancing algorithms are not applied)

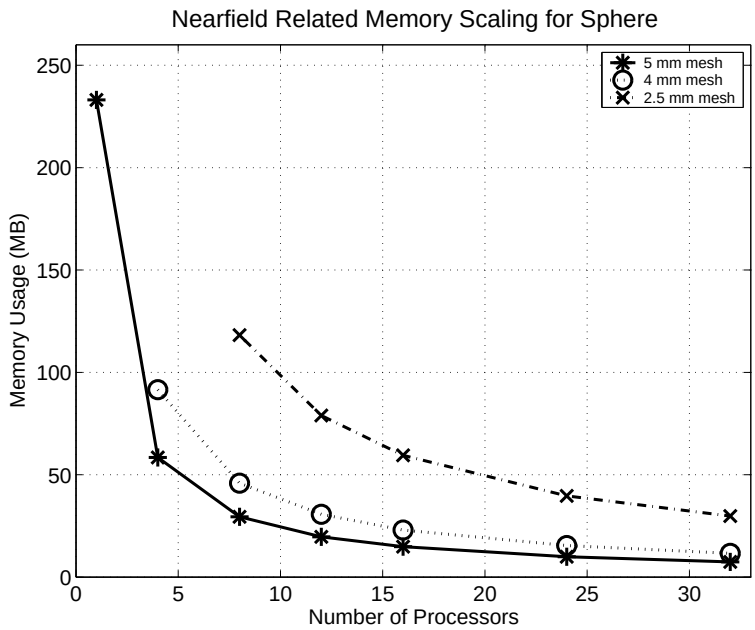

Figure 4.13: Memory scale plots of three different sphere meshes for array structures, which are related with near-field computations (load balancing algorithms are applied)

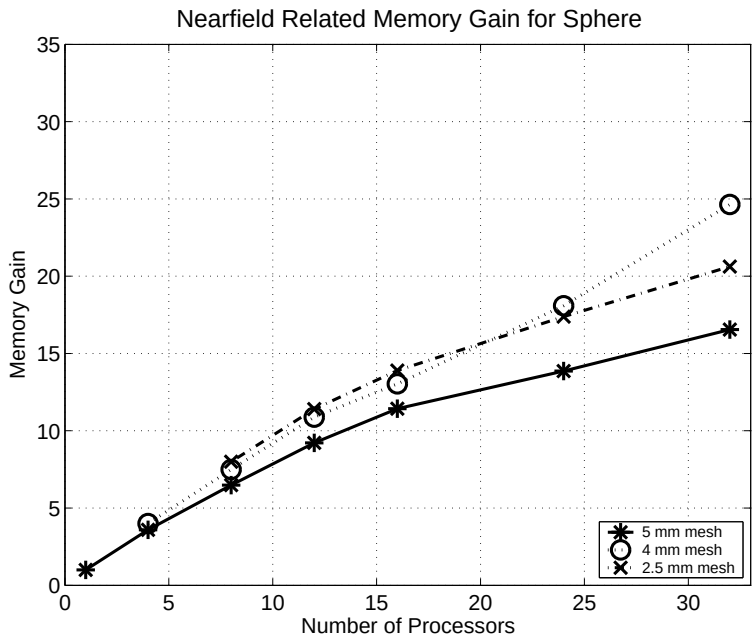

Figure 4.14: Memory gain plots of three different sphere meshes for array structures, which are related with near-field computations (load balancing algorithms are not applied)

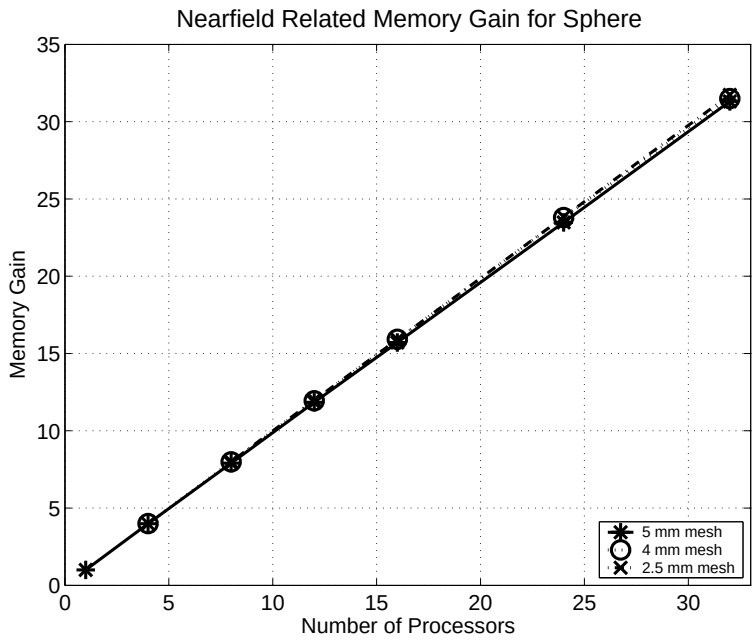

Figure 4.15: Memory scale plots of three different sphere meshes for array structures, which are related with near-field computations (load balancing algorithms are applied)

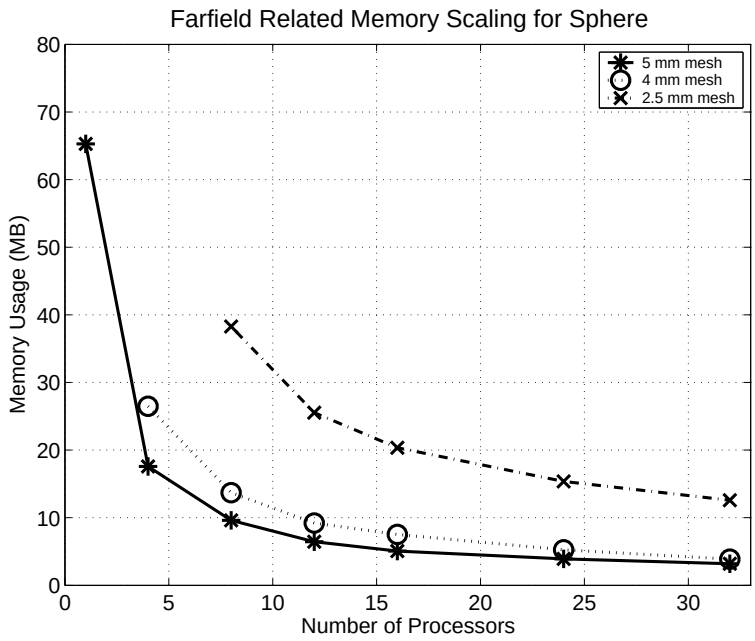

Figure 4.16: Memory scale plots of three different sphere meshes for array structures, which are related with far-field computations (load balancing algorithms are not applied)

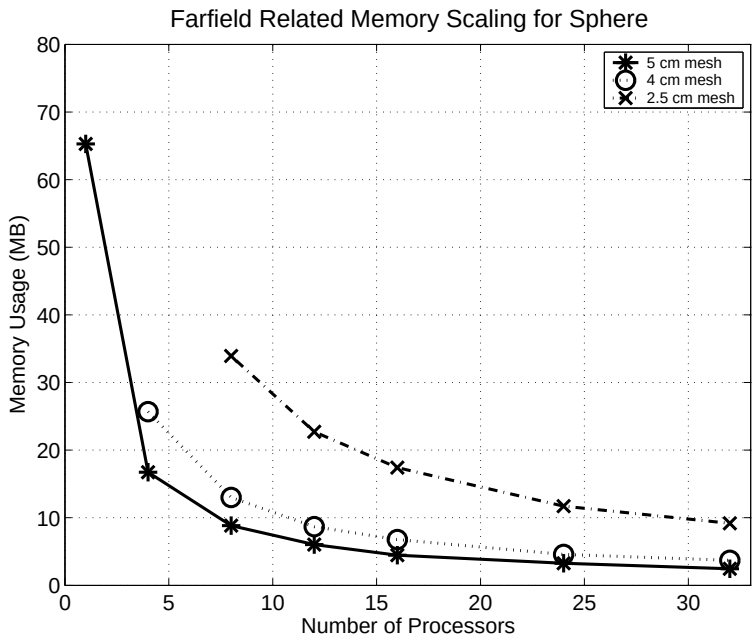

Figure 4.17: Memory scale plots of three different sphere meshes for array structures, which are related with far-field computations (load balancing algorithms are applied)

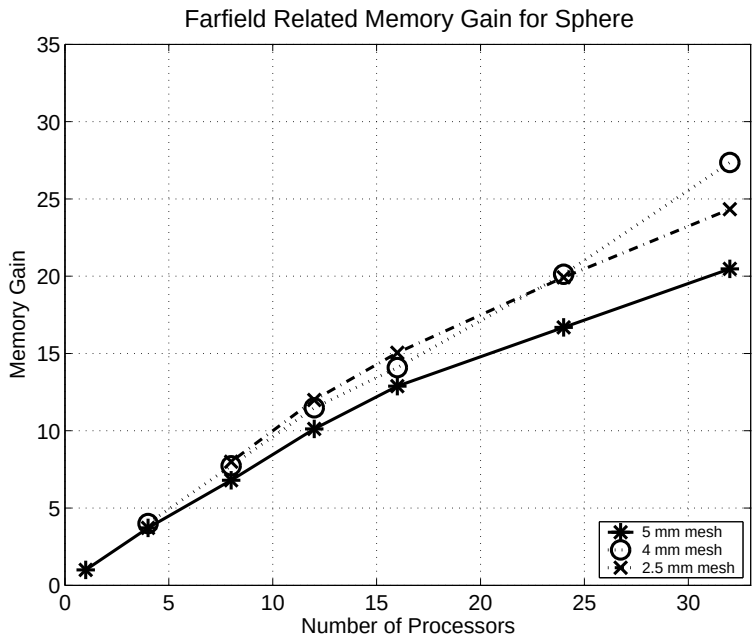

Figure 4.18: Memory gain plots of three different sphere meshes for array structures, which are related with far-field computations (load balancing algorithms are not applied)

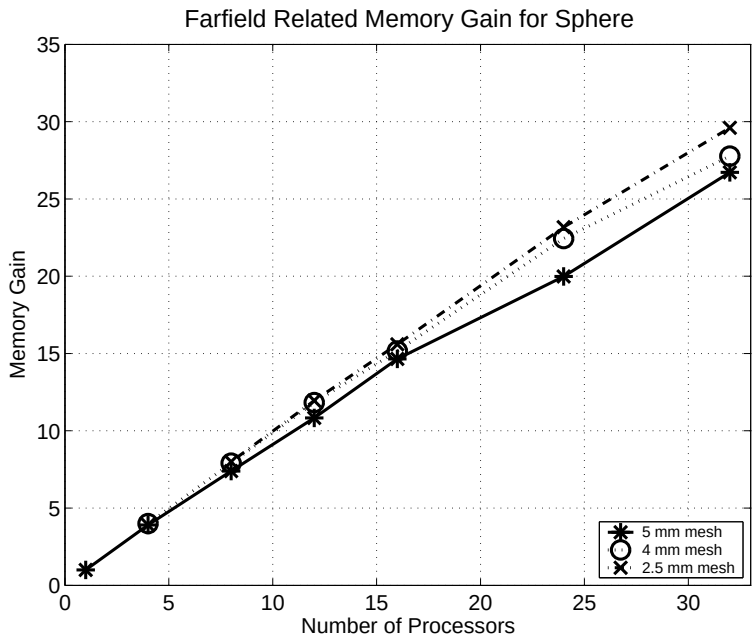

Figure 4.19: Memory gain plots of three different sphere meshes for array structures, which are related with far-field computations (load balancing algorithms are applied)

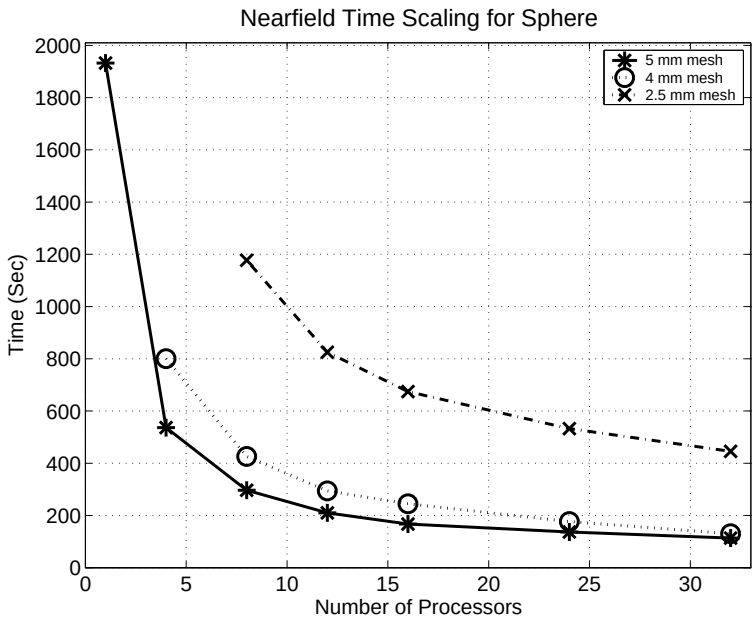

Figure 4.20: Time scale plots of three different sphere meshes for the computations of near-field interactions (load balancing algorithms are not applied)

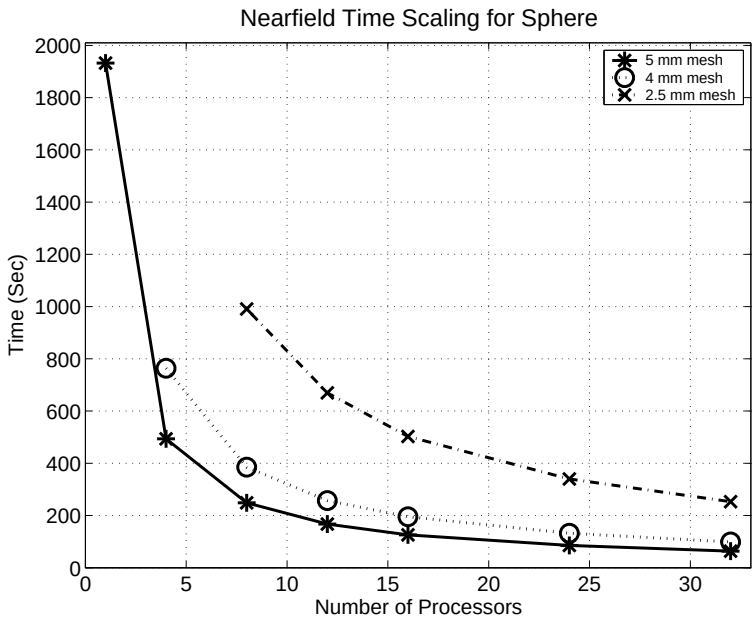

Figure 4.21: Time scale plots of three different sphere meshes for the computations of near-field interactions (load balancing algorithms are applied)

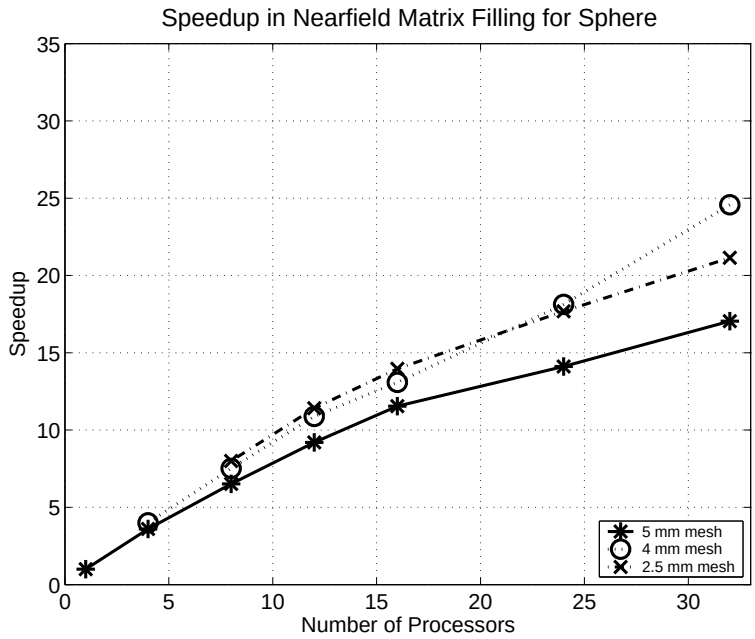

Figure 4.22: Speedup plots of three different sphere meshes for the computations of near-field interactions (load balancing algorithms are not applied)

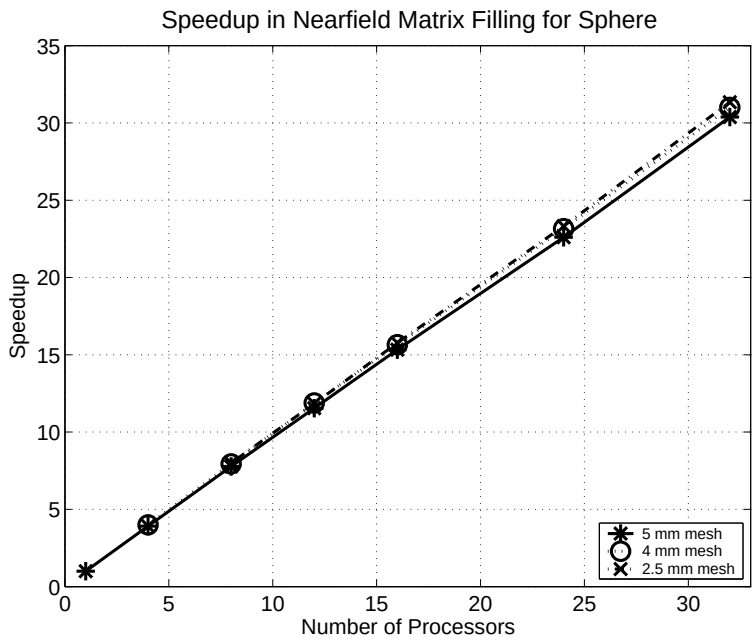

Figure 4.23: Speedup plots of three different sphere meshes for the computations of near-field interactions (load balancing algorithms are applied)

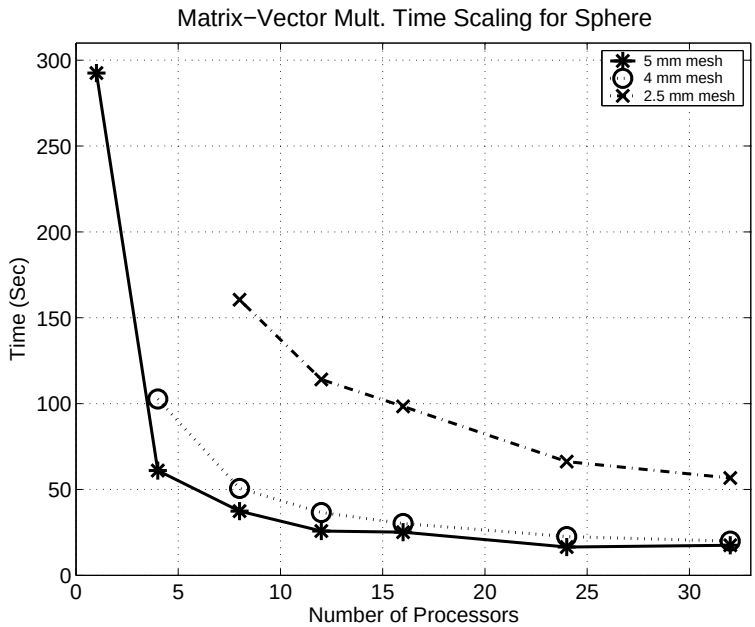

Figure 4.24: Time scale plots of three different sphere meshes for the matrix-vector multiplication (load balancing algorithms are not applied)

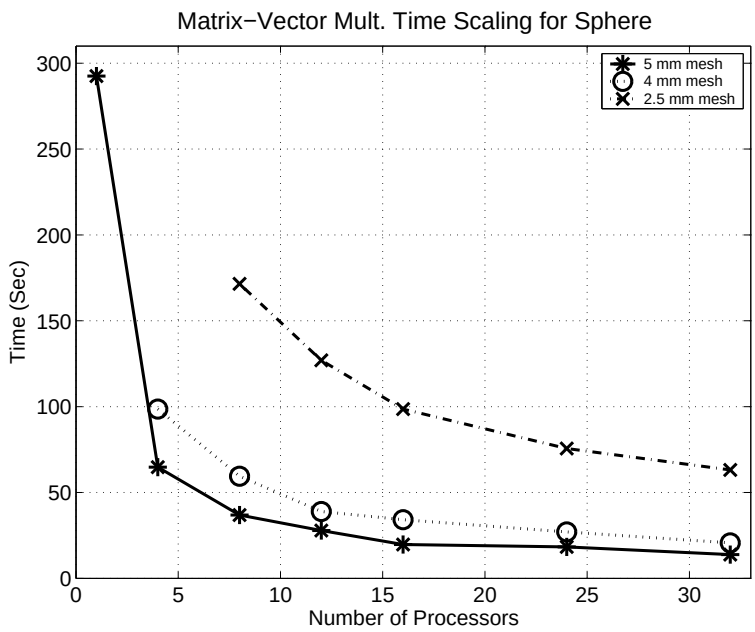

Figure 4.25: Time scale plots of three different sphere meshes for the matrix-vector multiplication (load balancing algorithms are applied)

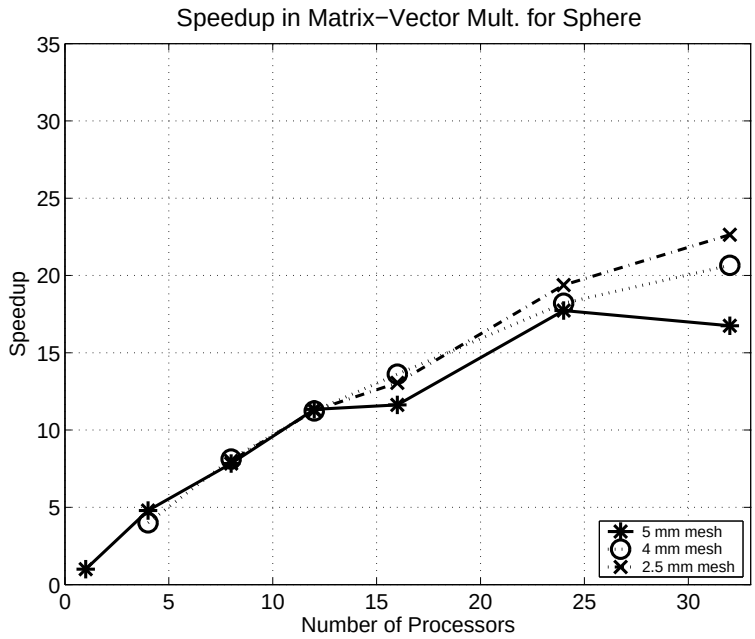

Figure 4.26: Speedup plots of three different sphere meshes for the matrix-vector multiplication (load balancing algorithms are not applied)

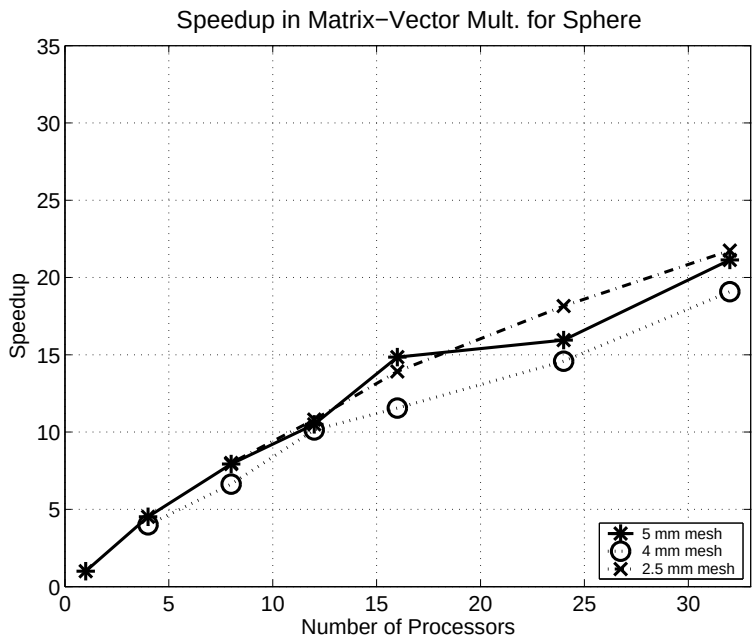

Figure 4.27: Speedup plots of three different sphere meshes for the matrix-vector multiplication (load balancing algorithms are applied)

As we expected before, memory load balancing technique applied on near-field partitioning makes near-field related memory structures perfectly scalable: In Fig. 4.15, all memory gain curves are linear and they coincide. In addition, this memory load balancing technique for near-field part makes the computations of near-field matrix filling perfectly scalable too; this can be seen from Fig. 4.23. On the other hand, speedup and memory gain plots for without load balancing case in the near-field part show us that the old partitioning that we used decreases the efficiency of near-field computations. From Fig. 4.14 and Fig. 4.22, it is easily observable that the speedup and memory gain values are less than 30 when 32 processors used. Therefore, we can state that the memory load balancing scheme affects the computational time and memory storage of this part very positively.

In far-field related figures, we observe that memory load balancing of far-field related array structures does not have a positive effect on speedup plots. Actually, memory load balancing for far-field structures are not deeply related with far-field related computations. Moreover, as can be seen from Figs. 4.26-4.27, speedup for the matrix-vector multiplication approximately goes to 20 when 32 processors are used. Therefore, we decided to work on new approaches for the load balancing of this part and this new strategy is going to include the effects of interprocessor communications and computational effects of far-field interactions' operations.

4.2 Helicopter Geometry

We also did profiling and scaling measurements with a generic helicopter model. This geometry is more complex with respect to PEC sphere problem and computational time for the solution of this problem, is much larger because of the complex geometrical structure. Picture of this helicopter is given in Fig. 4.28:

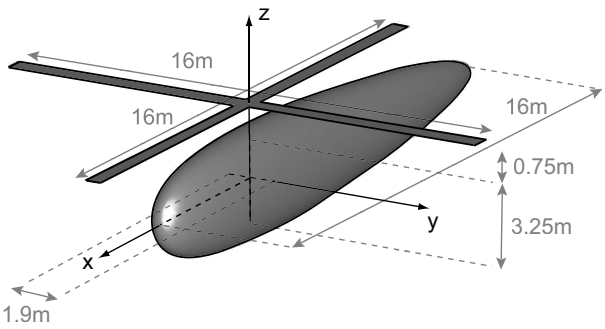

Figure 4.28: PEC helicopter model

Different mesh sizes data for this helicopter geometry were used, they are given in Table 4.2:

Helicopter Geometry Information		
Mesh Size	Number of Unknowns	Solution Frequency
1.5 cm	1,316,235	1.80 GHz
2.0 cm	739,404	1.30 GHz
2.5 cm	469,017	1.00 GHz
3.0 cm	325,665	0.85 GHz
4.0 cm	183,546	0.65 GHz
5.0 cm	117,366	0.50 GHz

Table 4.2: Helicopter geometries, which were used in the profiling measurements

In the solution of these meshes of helicopter geometry, we again used memory load balanced version of the parallel MLFMA and its counterpart, which does not have load balancing feature. Load balancing techniques are again applied to near-field related and far-field related parts, separately; thus, two different partitionings are used in the solution of these helicopter meshes as in the case of sphere meshes.

4.2.1 Time and Memory Profiling Results

As in sphere problem, time and memory profiling plots of the solution of 1.5 cm mesh helicopter geometry, which was solved on 32 processors of ULAKBIM's parallel system, are shown in this part. Color map for time and memory profiling plots are exactly the as the one, which is used in the 1.5 mm sphere mesh profiling results. The solution of this mesh of helicopter corresponds to the solution of a 1,316,235 unknowns full matrix equation.

From the figures, memory load balancing algorithms work properly for this complex geometry too. In Figs. 4.33 and 4.34, it is noticed that the parallel program achieves its peak memory usage before matrix-vector computations. Actually, this depends on a bug in our MLFMA program and it will certainly be fixed in short time.

As in the sphere problems, near-field memory load balancing implementation also balances the computational load in the near-field matrix filling operations and this is shown in Fig. 4.36. Nevertheless, load balancing applied on far-field related memory structures increases the idle time of processors. Thus, as mentioned before, we will move on a different strategy for this part of the parallel MLFMA. In the following part of this chapter, speedup and memory gain plots are presented and this situation about the far-field related calculations and array structures can be seen from these speedup and gain plots.

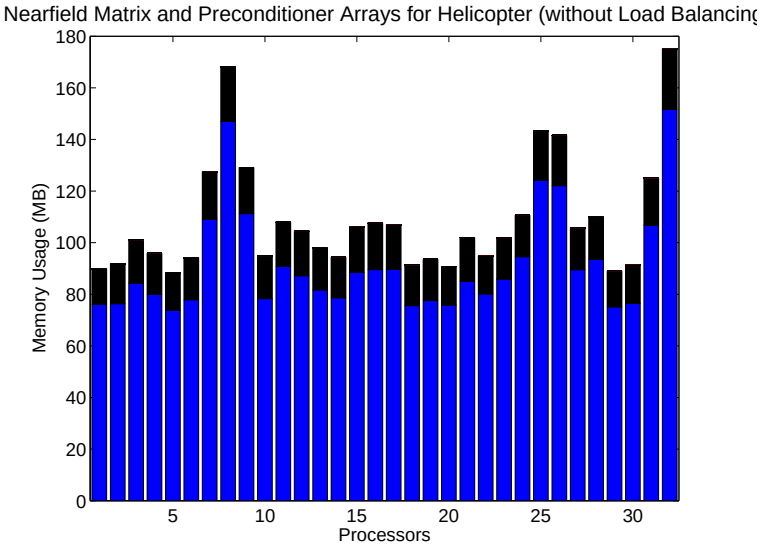

Figure 4.29: Memory usage of arrays structures which are related with near-field interactions for helicopter with 1.5 cm mesh when load balancing algorithms are not applied

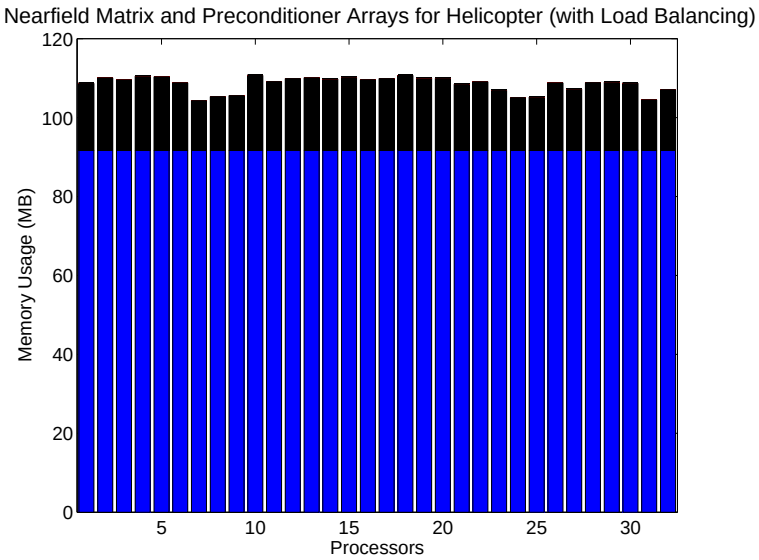

Figure 4.30: Memory usage of arrays structures which are related with near-field interactions for helicopter with 1.5 cm mesh when load balancing algorithms are applied

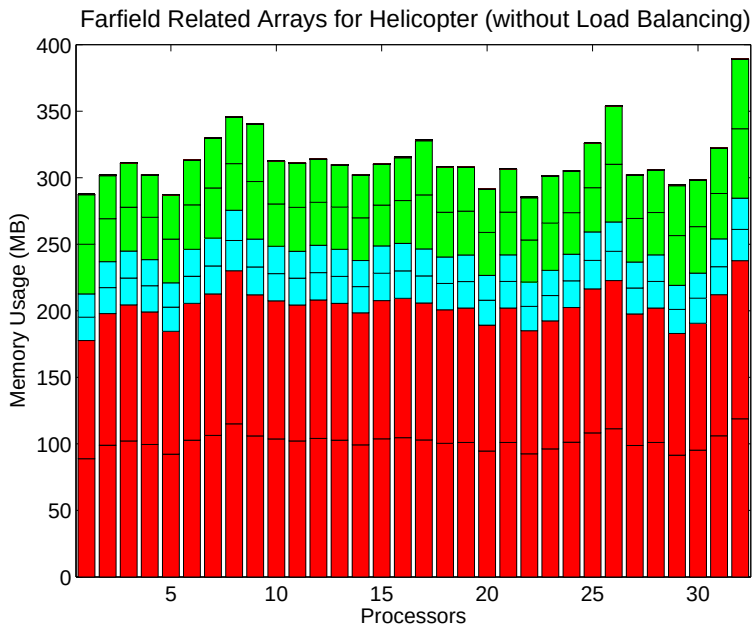

Figure 4.31: Memory usage of arrays structures which are related with far-field interactions for helicopter with 1.5 cm mesh when load balancing algorithms are not applied

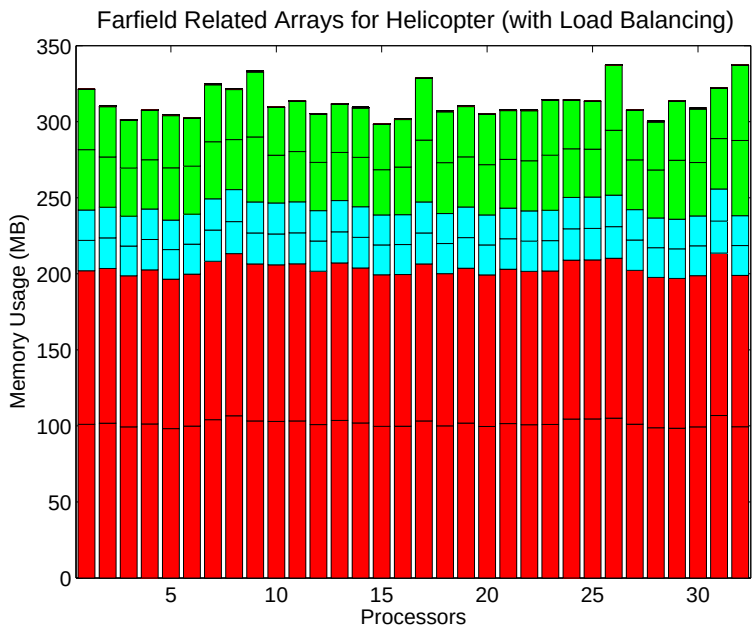

Figure 4.32: Memory usage of arrays structures which are related with far-field interactions for helicopter with 1.5 cm mesh when load balancing algorithms are applied

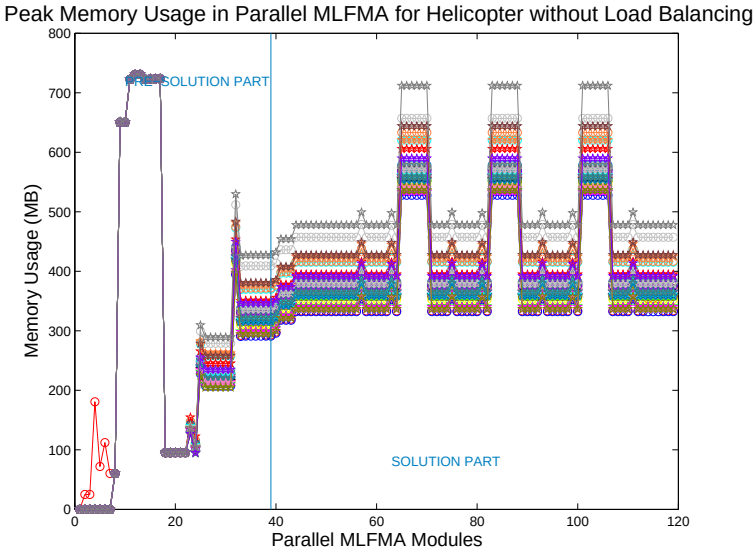

Figure 4.33: Peak memory usage in the modules of MLFMA for helicopter with 1.5 cm mesh when load balancing algorithms are not applied

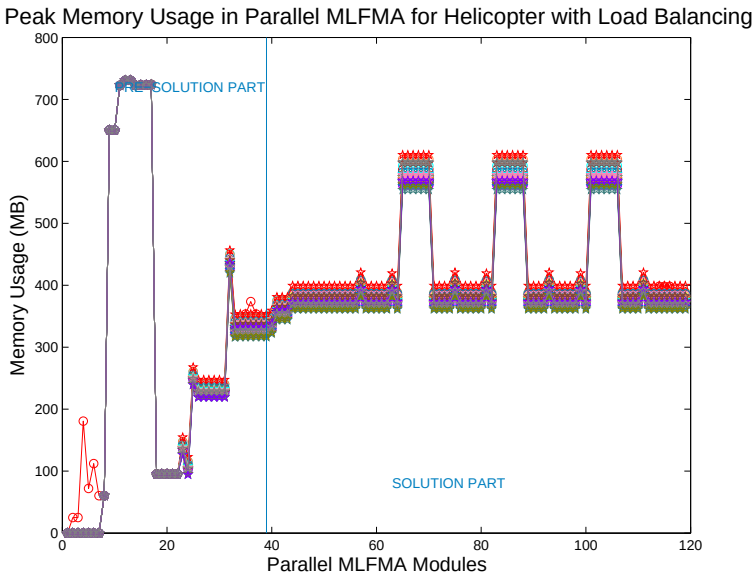

Figure 4.34: Peak memory usage in the modules of MLFMA for helicopter with 1.5 cm mesh when load balancing algorithms are applied

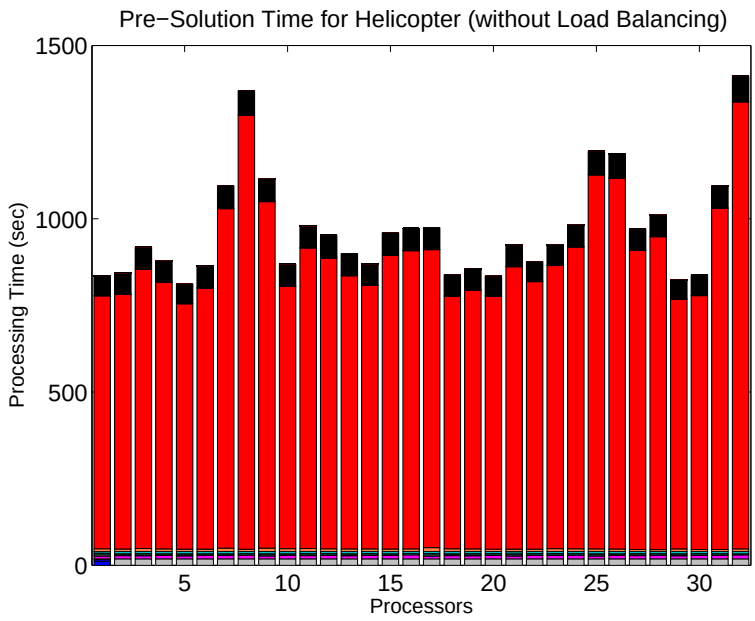

Figure 4.35: Pre-solution time for helicopter with 1.5 cm mesh when load balancing algorithms are not applied

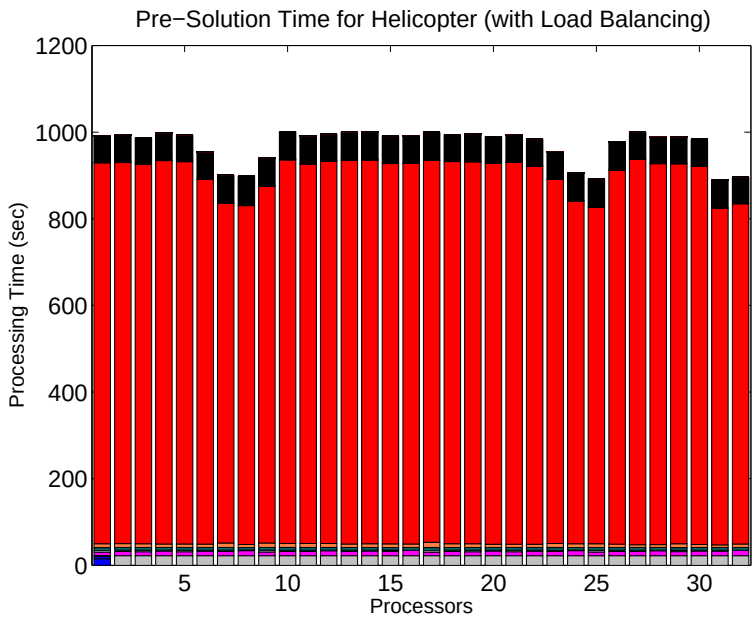

Figure 4.36: Pre-solution time for helicopter with 1.5 cm mesh when load balancing algorithms are applied

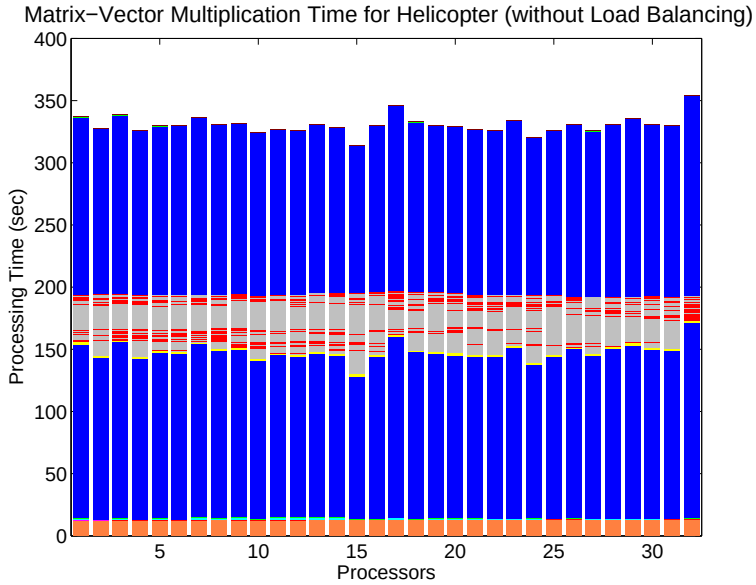

Figure 4.37: Matrix-vector multiplication-time profiling for helicopter with 1.5 cm mesh when load balancing algorithms are not applied

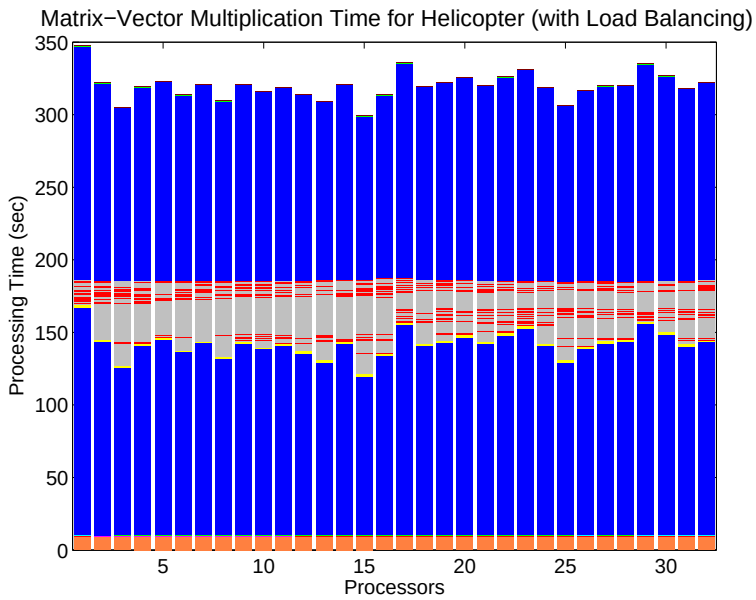

Figure 4.38: Matrix-vector multiplication-time profiling for helicopter 1.5 cm mesh when load balancing algorithms are applied

4.2.2 Speedup, Memory Gain, and Scaling Results

In this part of this chapter, we provide the parallel performance metrics' results for the different meshes of helicopter geometry with the different number of processors. These performance metrics plots show the effects of memory load balancing techniques on the scalability of the different parts of our parallel MLFMA program. This kind of work is also presented in previous parts of this chapter for the sphere problem; so, we follow the same analysis methodology for different helicopter mesh solutions. The meshes of helicopter that we use in this part are of sizes 5 cm, 4 cm and 2.5 cm. Solution frequencies and corresponding number of unknowns for these helicopter meshes can seen in Table 4.2.

In these parallelization metrics, plots of different helicopter meshes, scalability of near-field related computations and memory structures with the presentation of our load balancing approach can be observed again as in the cases of sphere meshes' near-field related scalability analysis. On the other hand, far-field related memory balance approach affects the computations about far-field interaction negatively. Since this situation is also observed in the scattering simulations of different meshes, we again decided to move our focus to different load balancing algorithms which might parallelize far-field related computations and memory storages by using optimization techniques. This would be our primary future work in load balancing studies of our parallel MLFMA program.

Memory scale and memory gain plots for the array structures, which are related with near-field computations are given in Figs. 4.39-4.42 and same plots for the array structures of far-field related computations are presented in Figs. 4.43-4.46. Following these figures, time scale and speedup plots for near-field computations are given in Figs. 4.47-4.4.50 and same plots for far-field computations can be shown in Figs. 4.51-4.54:

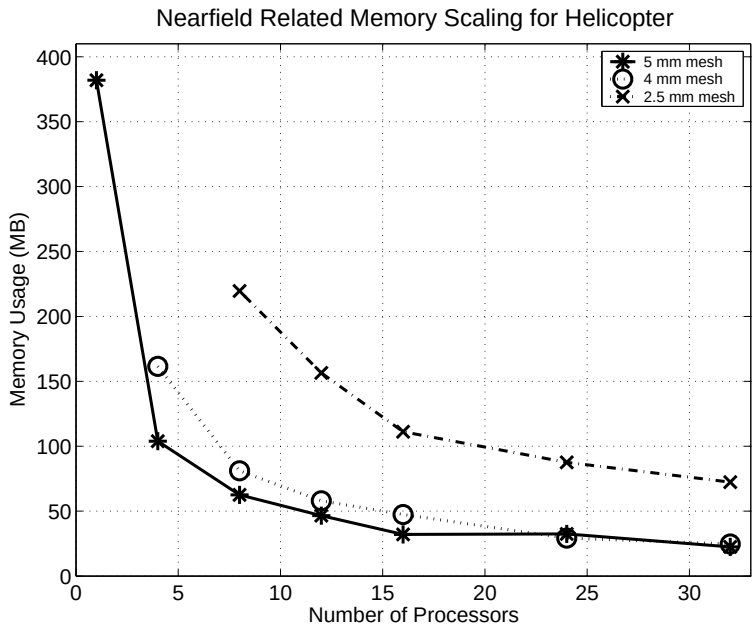

Figure 4.39: Memory scale plots of three different helicopter meshes for array structures, which are related with near-field interactions (load balancing algorithms are not applied)

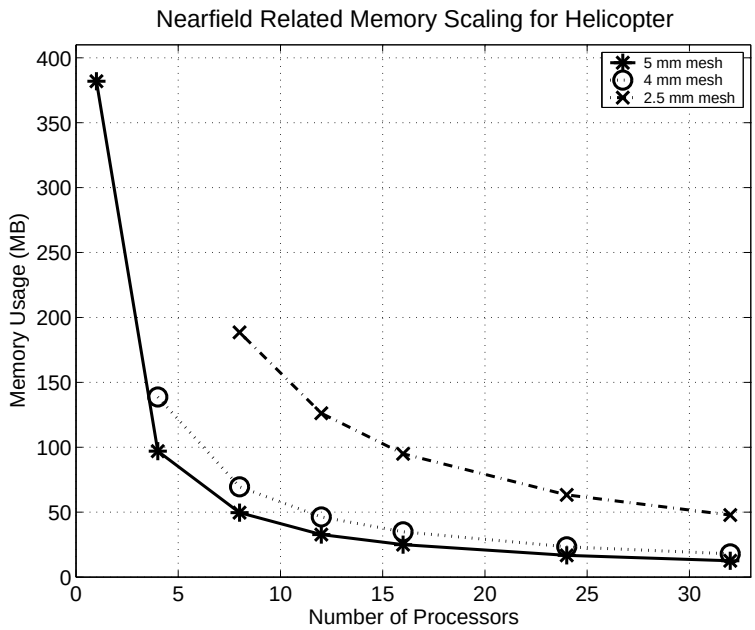

Figure 4.40: Memory scale plots of three different helicopter meshes for array structures, which are related with near-field interactions (load balancing algorithms are applied)

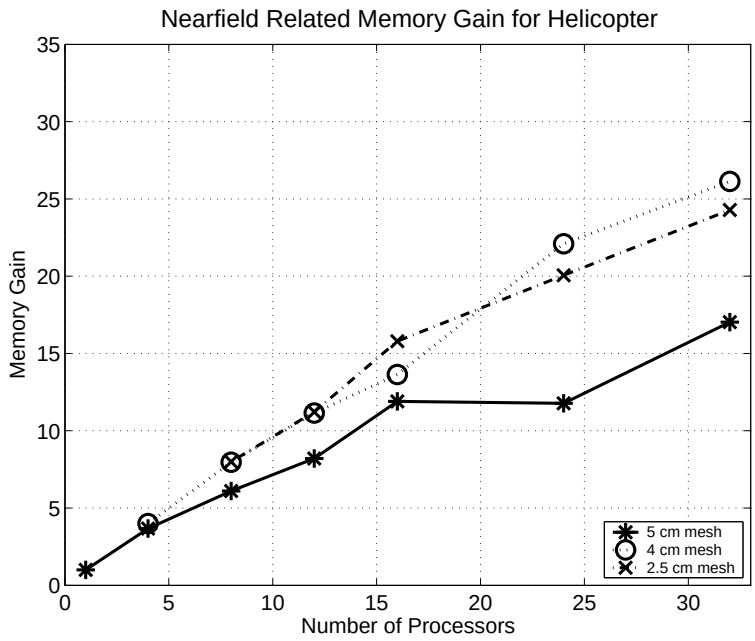

Figure 4.41: Memory gain plots of three different helicopter meshes for array structures, which are related with near-field interactions (load balancing algorithms are not applied)

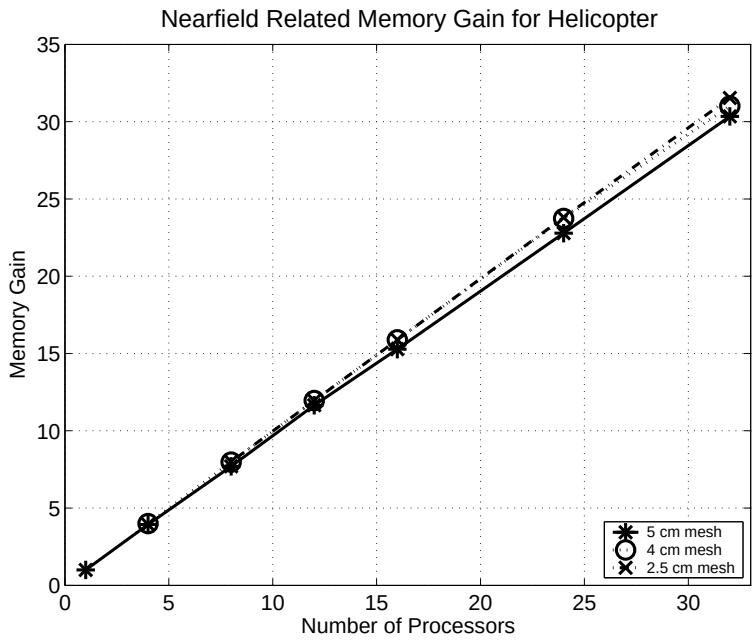

Figure 4.42: Memory gain plots of three different helicopter meshes for array structures, which are related with near-field interactions (load balancing algorithms are applied)

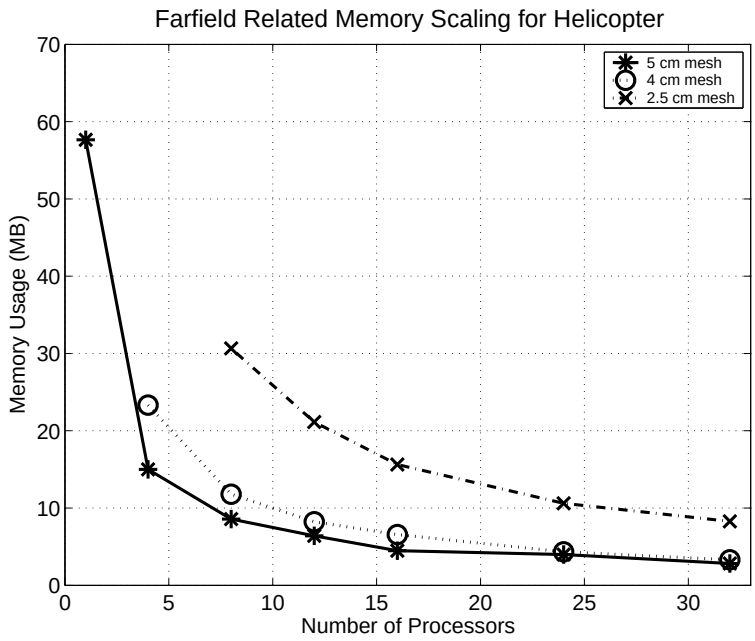

Figure 4.43: Memory scale plots of three different helicopter meshes for array structures, which are related with far-field interactions (load balancing algorithms are not applied)

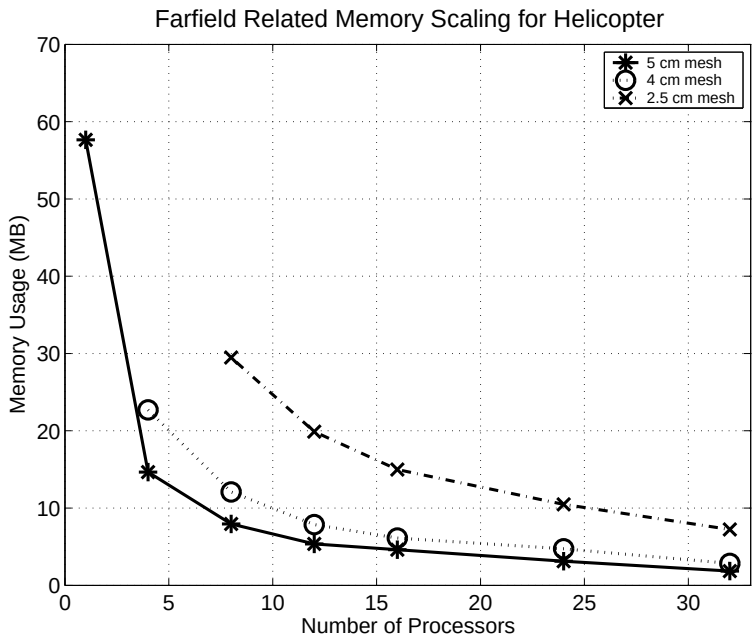

Figure 4.44: Memory scale plots of three different helicopter meshes for array structures, which are related with far-field interactions (load balancing algorithms are applied)

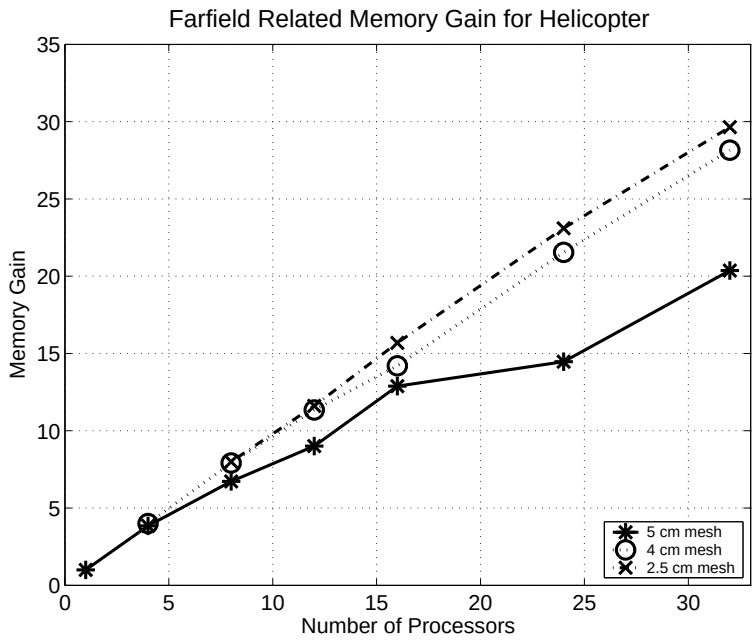

Figure 4.45: Memory gain plots of three different helicopter meshes for array structures, which are related with far-field interactions (load balancing algorithms are not applied)

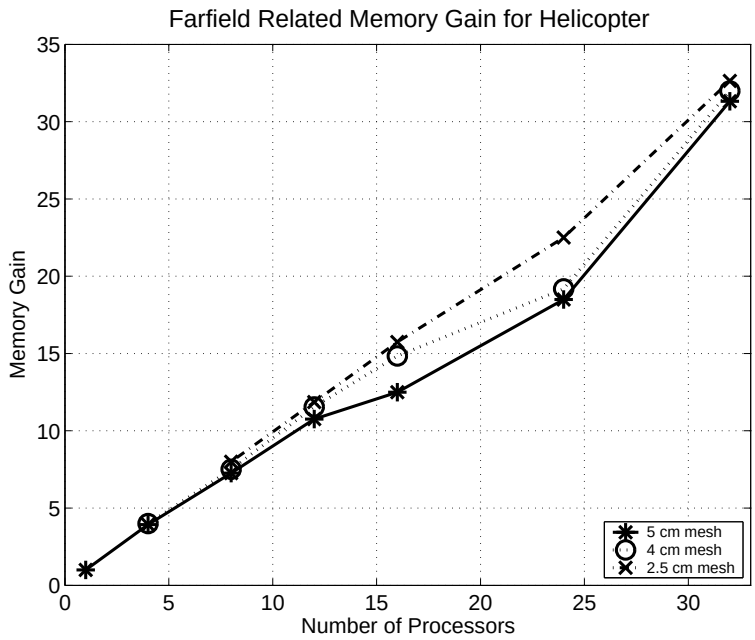

Figure 4.46: Memory gain plots of three different helicopter meshes for array structures, which are related with far-field interactions (load balancing algorithms are applied)

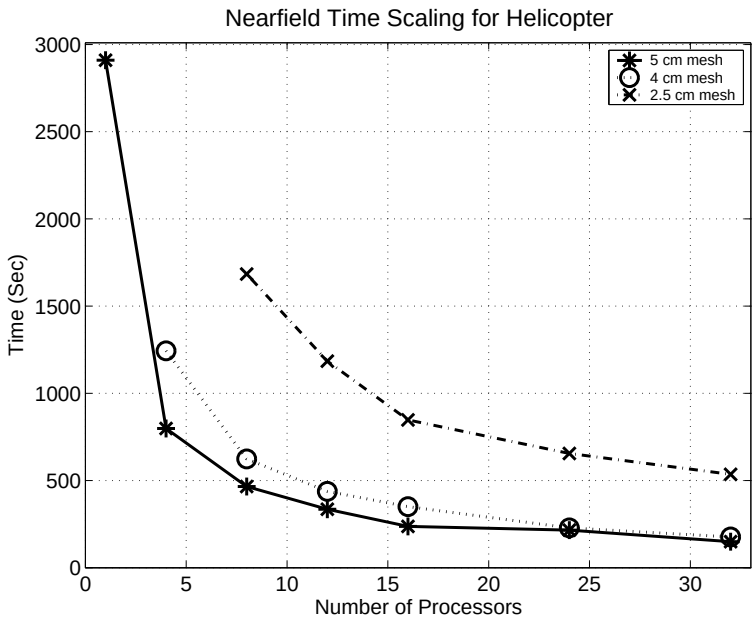

Figure 4.47: Time scale plots of three different helicopter meshes for the computations of near-field interactions (load balancing algorithms are not applied)

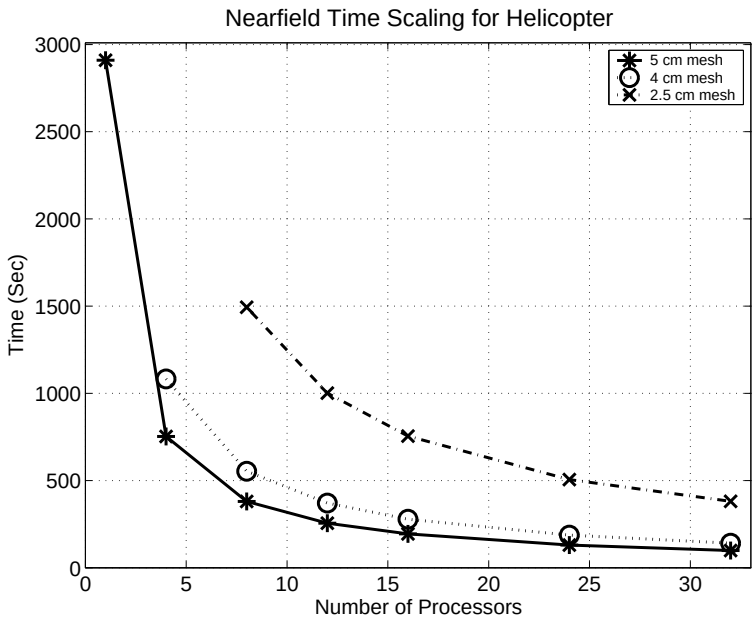

Figure 4.48: Time scale plots of three different helicopter meshes for the computations of near-field interactions (load balancing algorithms are applied)

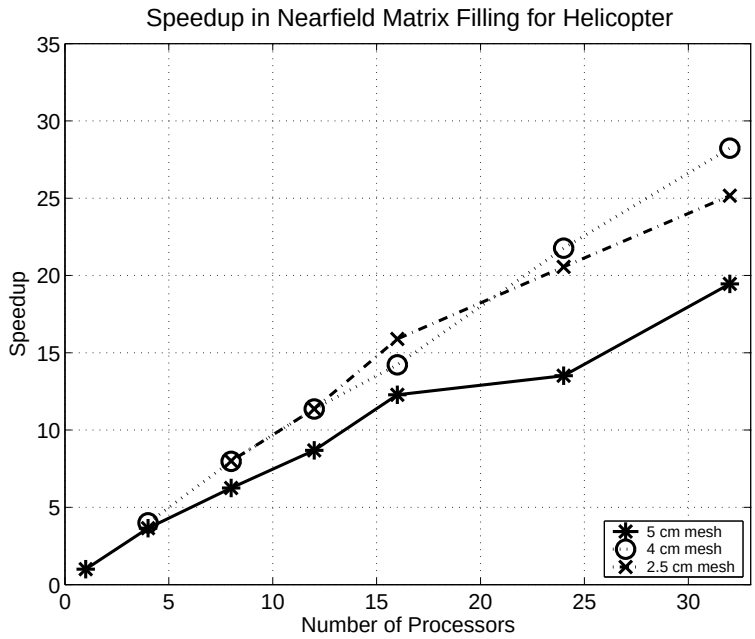

Figure 4.49: Speedup plots of three different helicopter meshes for the computations of near-field interactions (load balancing algorithms are not applied)

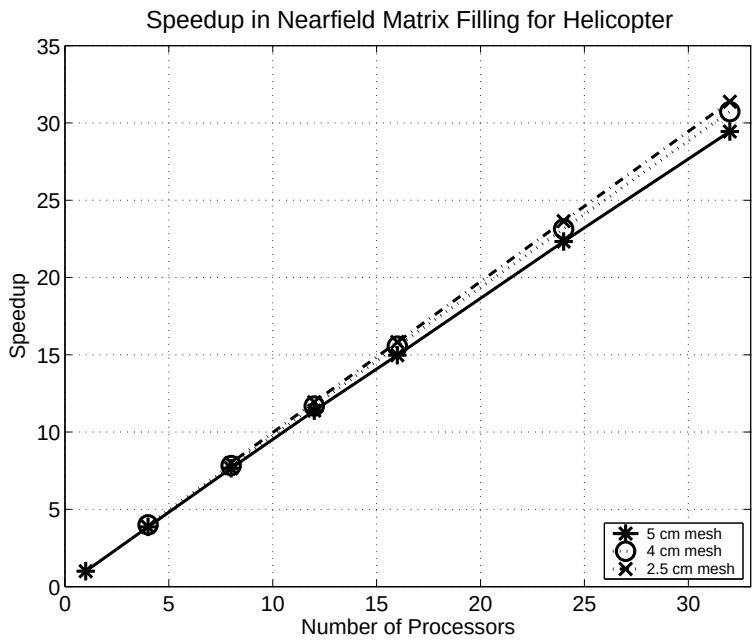

Figure 4.50: Speedup plots of three different helicopter meshes for the computations of near-field interactions (load balancing algorithms are applied)

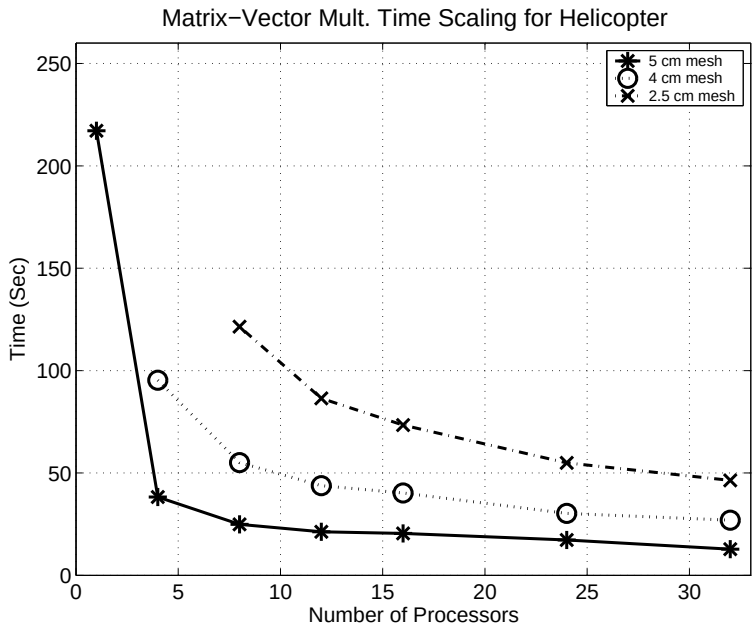

Figure 4.51: Time scale plots of three different helicopter meshes for the matrix-vector multiplication (load balancing algorithms are not applied)

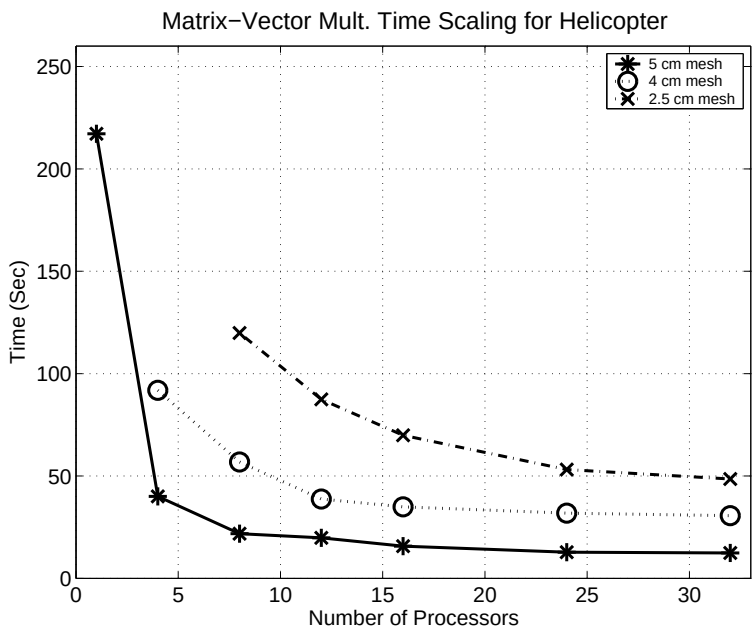

Figure 4.52: Time scale plots of three different helicopter meshes for the matrix-vector multiplication (load balancing algorithms are applied)

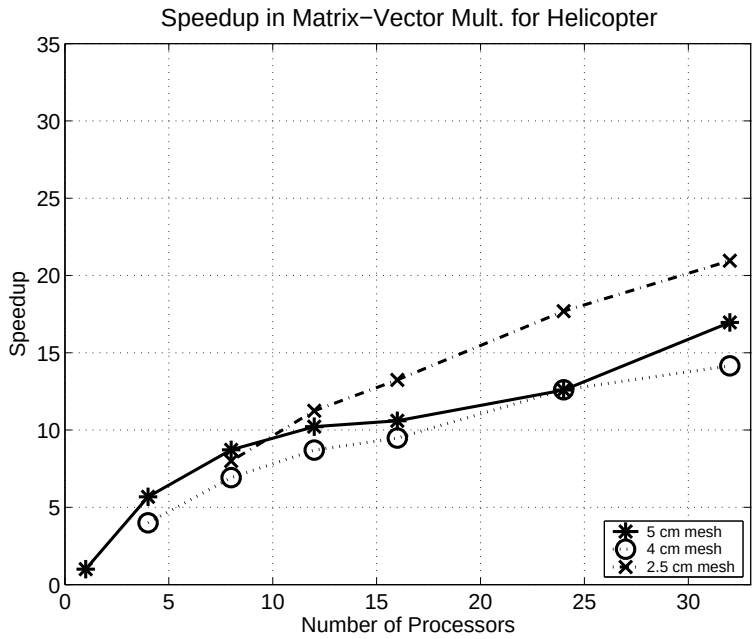

Figure 4.53: Speedup plots of three different helicopter meshes for the matrix-vector multiplication (load balancing algorithms are not applied)

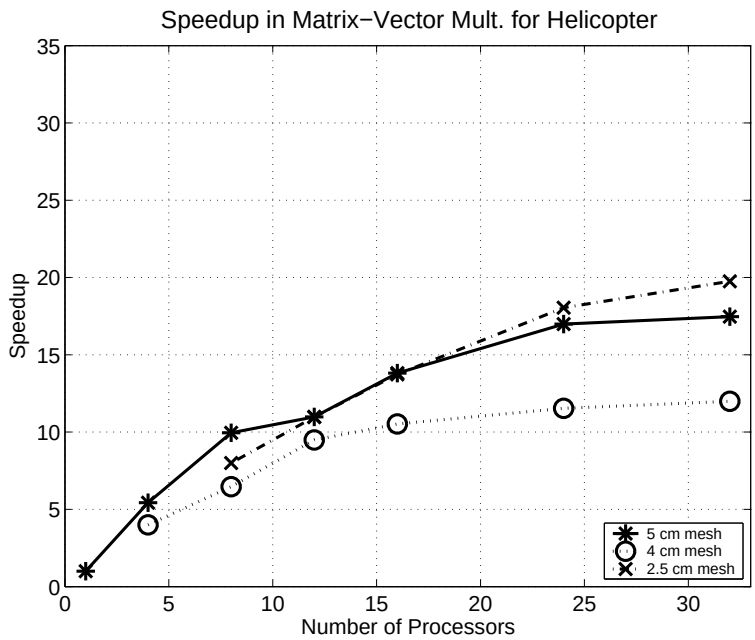

Figure 4.54: Speedup plots of three different helicopter meshes for the matrix-vector multiplication (load balancing algorithms are applied)

Chapter 5

Conclusion

In this thesis, different parallel hardware and software approaches are presented for the parallelization of the efficient electromagnetic simulation method MLFMA. Parallel hardware and software structures, parallelization performance metrics and the basics of the parallelization of MLFMA are presented in the introduction chapter.

Following this introductory information, in the second chapter, details of parallel computation environments, which we used in our studies, are described. In this chapter, mostly, hardware backbones for parallel environments are examined in detail. These parallel hardware backbones are the parallel cluster systems that we used, built or plan to build. Software components of parallel environments that are needed to execute parallel applications on these systems are also explained in Chapter 2. Moreover, performance comparisons of different parallel cluster systems are presented in this chapter by using our sequential and parallel versions of MLFMA program as benchmark tools. Depending on these performance comparisons, we chose the hardware components of our proposed parallel cluster, which is expected to give the optimum parallel performance during the execution of our parallel code.

In Chapter 3, we mainly focused on the parallelization of MLFMA. First, adaptation problems about sequential MLFMA is described in this chapter; and

then, the primary parallelization studies done on this sequential version of the program are summarized. After these, partitioning techniques and approaches that are used in our parallel program are explained in detail. Following these partitioning concepts, importance of load balancing and our load balancing strategies for the memory storage of parallel MLFMA are presented. In this presentation, we underline that different load balancing strategies are used for the near-field and far-field memory storages of our parallel program. Moreover, the details of the memory load balancing algorithms are given in this chapter.

The effects of memory load balancing for the near-field interactions and far-field interactions parts of the parallel MLFMA program are shown by computational electromagnetic simulations in Chapter 4. Different meshes of a PEC sphere and an antitank helicopter were used in those simulations. By using the simulation results. The time and memory usage profiling of our parallel MLFMA program are depicted. Furthermore, parallelization measurement metrics are applied these simulations results: Time and memory scaling plots, speedup and memory gain curves for different meshes of these two problems are presented in this chapter and scalability analysis were done using these plots. It is observed that memory load balancing approach for near-field related parts of the program affects the scalability of these parts very positively; we get more scalable near-field matrix operations and memory storage for near-field related arrays with this balancing algorithm. On the other hand, far-field related memory load balancing approach does not have this kind of positive effect on far-field computations, which are mostly done in matrix-vector multiplications of our parallel solver. Therefore, we decided to consider different optimization and scalability improvement approaches for the far-field related parts of the parallel MLFMA program.

To conclude this study, we presented the effects of memory load balancing on different features of parallel MLFMA program. By using computational data, we observed that load balancing approaches certainly have a significant effect on the overall parallel performance of the code. Proper load balancing approaches would certainly result with a real scalable version of parallel MLFMA and this kind of study has not been done in the CEM community. We now plan to focus

our efforts to build such a completely scalable version of the parallel MLFMA.

Bibliography

[1] R. Coifman, V. Rokhlin, and S. Wandzura, "The fast multipole method for the wave equation: A pedestrian prescription" *IEEE Antennas and Propagat. Mag.*, vol. 35, no. 3, pp. 7–12, June. 1993.

[2] B. Dembart and E. Yip, "A 3-D fast multipole method for electromagnetics with multiple levels" *in 11th Annu. Rev. Progress Appl. Computat. Electromagn.*, Monterey, CA, Mar. 1995, pp. 621–628.

[3] J. Song, C.C. Lu, and W.C. Chew, "Multilevel fast multipole algorithm for electromagnetic scattering by large objects" *IEEE Trans. Antennas Propagat.*, vol. 45, no. 10, pp. 1488–1493, October. 1997.

[4] S. Velamparambil, W.C. Chew, and J. Song, "10 million unknowns: Is it that big?" *IEEE Antennas and Propagat. Mag.*, vol. 45, no. 2, pp. 43–58, April. 2003.

[5] Grama A., Gupta A., Karypis G., Kumar V., *Introduction to Parallel Computing*, Addison-Wesley, Edinburgh, 2003.

[6] Wilkinson B., Allen M., *Parallel Programming*, Upper Saddle River, NJ: Prentice Hall, Inc., 1999.

[7] Sterling T., *Beowulf Cluster Computing with Linux*, Cambridge, MA: The MIT Press, 2002.

[8] Gropp W., Lusk E., Skjellum A., *Using MPI*, Cambridge, MA: The MIT Press, 1994.

[9] Gropp W., Lusk E., Thakur R., *Using MPI-2*, Cambridge, MA: The MIT Press, 1999.

[10] Balanis C. A., *Advanced Engineering Electromagnetics*, John Wiley & Sons, New York, 1989.

[11] O. S. Ergul, *Fast multipole method for the solution of electromagnetic scattering problems*, M. Eng. Thesis, Bilkent University, Ankara, Turkey, June 2003.

[12] Sterling T., *Beowulf Cluster Computing with Windows*, Cambridge, MA: The MIT Press, 2002.

[13] Danesh Arman., *Making Linux Work: Essential Tips and Techniques*, Albany, NY: OnWord Press, 2002.

[14] Strang G., *Linear Algebra and its Aplications*, third edition, Harcourt Brace Jovanovich, New York, 1988.

[15] Kupferschmid M., *Classical Fortan: Programming for Engineering and Scientific Applications*, Marcel Dekker Inc., New York, 2002.

[16] Saad Y., *Iterative Methods for Sparse Linear Systems*, second edition, SIAM, Philadelphia, 2003.

[17] Anderson E., Bai Z., Bischof C., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Ostrouchov S., and Sorensen D., *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[18] Guller D. E., Singh J. S., and Gupta A., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kauffman Publishers Inc., San Francisco, CA: 1998.

[19] Cormen T.H, Leiserson C. E., and Rives R. L., *Introduction to Algorithms*, MIT Press, Cambridge, MA: 2000.

[20] Harrington R. F., *Field Computation by Method of Moments*, IEEE Press, 1993.