1999-10-06

Efficient Implementation of Mesh Generation and FDTD Simulation of Electromagnetic Fields

Jonathan Hill
Worcester Polytechnic Institute

Efficient Implementation of Mesh Generation and FDTD Simulation of Electromagnetic Fields

by

Jonathan Hill

A Thesis
Submitted to the Faculty
of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the
Degree of Master of Science in

Electrical Engineering

by

August 1996

Approved:

Dr. W. R. Michalson, Advisor	Dr. R. Ludwig, Thesis Cmte.
Dr. R. J. Duckworth, Thesis Cmte.	Dr. J. Orr, Head of Department

Abstract

This thesis presents an implementation of the Finite Difference Time Domain (FDTD) method on a massively parallel computer system, for the analysis of electromagnetic phenomenon. In addition, the implementation of an efficient mesh generator is also presented. For this research we selected the MasPar system, as it is a relatively low cost, reliable, high performance computer system. In this thesis we are primarily concerned with the selection of an efficient algorithm for each of the programs written for our selected application, and devising clever ways to make the best use of the MasPar system. This thesis has a large emphasis on examining the application performance.

Revision Notice

Before making this thesis available for republication, a handful of typographical errors were corrected. Corrections were made for extraneous words as well as for spelling errors that surprisingly, were not found by the `ispell` program. Besides these, the most significant changes include the following;

- Page 195, Table 4.3 - In *Entry 2*, ABC was replaced by PMC
- Page 227, Figure 5.12 - The comment was corrected to match caption
- Page 214, Equation (5.4) - Corrected to include extra layer

These corrections were made in good faith by the Author, Jonathan Hill on October 4, 1999.

Acknowledgements

First, I must give my sincerest thanks to my advisor, Professor William Michalson. You have been my supporter, champion, and motivator for this effort. Because of your guidance, I was able to persevere, complete the research and deliver this document. Prof. Michalson, you are an example of professorship that I admire and respect. I appreciate your faith in me.

I must thank Prof. Jin-Fa Lee. You have helped by offerring advice during the research phase of this project. The commercial software that you made available was instrumental in the development of this project. Even long after the research was complete, I still had an account on your computer system.

Thanks go to the rest of my thesis committee, Prof. Reinhold Ludwig who helped review the electromagnetic theory presented here, as well as Prof. James Duckworth who help review the thesis as well.

In no special order, thanks go to a few of my friends. Thanks go to George Gubrell, Hua Hua, Jennifer Stander, Mirko Spasojevic, Witold Jachimczyk, Gary Adamowicz, Lee Evans, Ramesh Sharma, Jacques Beneat, Guanghua Peng, Doris Boronowski, Surrender Mohan, Christine Easton, Imad Nejdawi, and Mustapha Fofana. Your advice and good humor has been a great help.

Thanks go to our system administrator, Babak Najafi. You have been able to keep our computer network up, despite the law of entropy. Thanks are extended to the support staff at MasPar corporation. Their assistance and effort in making the MP 1208 available was beyond the call of duty. Thanks are also extended to Prof. John Conery, for making the MP 1104 owned by the University of Oregon available.

Next, I must thank my family, especially my Mother, Marilyn and my new father, Louis. Thanks also go to my brothers Peter, Timothy, and new brothers John and Paul. Thanks as well go to my new sisters Heidi and Terry. Thank you for the fun times we had. I especially enjoyed our visits and vacations. I am also thankful for your patience and prodding, your effort has in fact helped to make this thesis possible.

At last, my warmest thanks go to my late father, Dr. David Hill who still lives in my memory. I have made a home for you in my heart. Perhaps some of your scholarly genius found its way to me. It is for my father that I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents an application of the Finite Difference Time Domain (FDTD) method on a massively parallel computer system, for the analysis of electromagnetic phenomenon. For this application we have selected the MasPar system, as it is a relatively low cost, reliable, high performance computer system. In general, writing efficient programs for high performance computers represents a real challenge. This difficulty arises in two forms; The need to process large problems, and the presence of computer hardware features. The need to process large problems translates into the need for better algorithms. The use of hardware features calls for better algorithms and better coding. We consider these issues next.

High performance computers are generally used to process large problems. To efficiently process large problems, the time complexity of algorithms becomes a very important issue. Consider the following example; For large problems the fast Fourier transform is preferred to the basic definition of the discrete Fourier transform. The reason for this is that while the basic definition of the discrete Fourier transform is simple to program, it has a quadratic relationship between execution time and problem size. While the fast Fourier transform is more complicated, it has an $n \log(n)$ time complexity. For applications that need to transform large data sets, the fast Fourier transform provides much better performance than the basic definition of the transform. The point being made here is that high performance computers demand that programmers devise better, more sophisticated algorithms.

A second issue that makes it difficult to write programs for high performance computers is that these computers generally have hardware features that are simply not found on simpler computers. While such features are supposed to be helpful in enhancing performance, they complicate the task of programming the machine. To write an efficient program for such a machine, a programmer must be able to devise algorithms that can take advantage of these hardware features. In particular, in the MasPar system, processing speed is achieved by use of a multitude of simple processors that we refer to as processing elements (PEs). In such a machine, an efficient program must be able to coordinate all these processing elements.

In this thesis we are concerned with the selection of an efficient algorithm for each of the programs written for our selected application, and devising clever ways to make the best use of the MasPar system. This thesis has a large emphasis on examining the application performance. The issues of characterizing algorithm time complexity and describing hardware use come up again and again throughout this document.

1.1 The Application

Traditionally, RF and microwave design follows an iterative procedure, requiring the construction and evaluation of many prototype models. Unfortunately, the construction and evaluation of prototypes can be labor intensive, time consuming, and expensive. At this juncture, the electronics industry has been under pressure from the marketplace to design better, less expensive products, in less time. Segments of the electronics industry have responded with some success, by developing software tools and techniques that assist designers in their work. It appears that for RF designers to be able to keep pace with the rest of industry, such tools and techniques will need to be developed to assist in the task of RF design.

As an alternative to the construction of real prototype models, this research considers the development of specific tools that will build and evaluate mathematical models of prototypes on a given computer system. Naturally, in performing an evaluation, such a tool should yield data that an RF communications designer would have interest in. Such data might include the estimation of scattering parameters and feed-point impedance. Such topics are discussed in section 4.2.

For this research, a simple, well known technique called the Finite Difference Time Domain (FDTD) Method was selected to perform the required electromagnetic evaluation. Since both the construction of a model for the FDTD method as well as the FDTD method itself are both extremely computationally intensive, it was decided to use a high performance, massively parallel computer architecture as a platform for the software tools. It is hoped that such an implementation will provide a performance level suitable for everyday use by RF design engineers.

1.1.1 Application Outline

Figure 1.1 provides an outline of the steps followed by a designer, to construct and evaluate a single mathematical model of a prototype. For exact details on this process, the reader is referred to the user manual in appendix F. The first step is to enter a description of the structure to be modeled by using commercially available 3D CAD software. For this research we used a program called I-DEAS[1]. In figure 1.1, step 1 shows that the user has entered the solid model of a sphere.

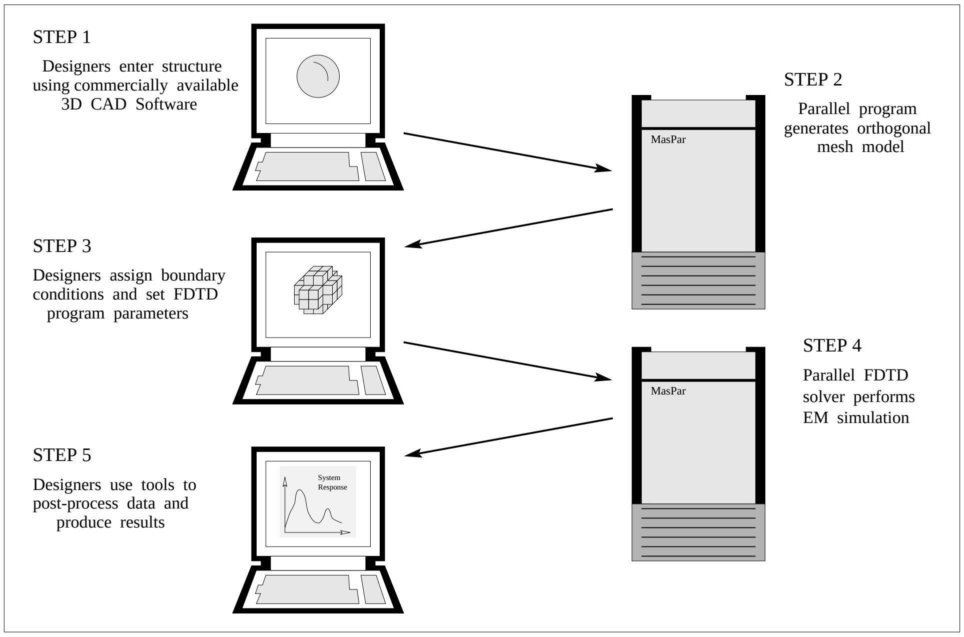

STEP 1
Designers enter structure using commercially available 3D CAD Software
STEP 2
Parallel program generates orthogonal mesh model
MasPar
STEP 3
Designers assign boundary conditions and set FDTD program parameters
STEP 4
Parallel FDTD solver performs EM simulation
MasPar
STEP 5
Designers use tools to post-process data and produce results
System Response

Figure 1.1: Steps in Using Software Tools

We used I-DEAS to produce what is called a Universal File. The Universal File Format was selected for our use, because it is an easy to manipulate ASCII type file format. Despite the name, it turns out that the Universal File Format is proprietary and is primarily a convenience for developers who wish to support the I-DEAS product. We use a conversion program to convert Universal Files to our own *Solid File Format*. Note that since Universal Files are not widely supported, if a user wished to use some other solid modeling program, it would be necessary to write a new conversion program.

[1]I-DEAS is the name of a software product from Structural Dynamics Research Corporation

The second step is to generate a mesh model from the structural description. The mesh generator reads the *solid file* and produces two files, the first is referred to as the *mesh file* and is used to store the entire FDTD mesh. The second file is referred to as the *hollow file* which is used to assign boundary conditions.

In step 3 the user assigns boundary conditions. In figure 1.1, the third step shows the boxes associated with the surface of the FDTD mesh of the sphere that was just modeled. Boundary conditions are assigned by giving properties to surfaces like those illustrated in figure 1.1. Currently, PATRAN[2] is used to assign boundary conditions. The hollow file follows PATRAN's Neutral File Format, which is another proprietary file format. After boundary conditions have been assigned, PATRAN is used to write a neutral file that contains the assigned surface properties. The assigned boundary conditions are written to the mesh file by a program called `fconv2` that read the neutral file.

To prepare the mesh file for FDTD simulation in the MasPar, a program called `ComEdit` is called to insert simulation parameters into the mesh file. After this is done, the mesh file contains all the data that the FDTD solver will need. In step 4, the completed mesh file is submitted to the FDTD solver which performs an electromagnetic simulation.

In step 5, tools are used to *post process* data from the FDTD simulation. While the FDTD solver produces some data that can immediately be examined, in general a designer will use software tools like PATRAN and fast Fourier transform related tools. These tools process the raw data that the FDTD solver produces. In figure 1.1, the computer display shows results from some unnamed software tool.

1.1.2 Application Summary

These software tools have successfully been used to perform several simulations. Of the simulations performed, this document presents three such examples. In section 5.4.1 a simple model is presented that we use to examine the performance of the FDTD solver. In appendix B an example of a microstrip structure is presented, and the user manual in appendix F the example of a waveguide is given. Again, the user is referred to the user manual for in depth details regarding the use of these software tools.

The development of any high performance implementation is not a trivial task. Sometimes special algorithms or special computer hardware is required to provide

[2]P3/PATRAN is the name of a software product from PDA Engineering's PATRAN Division

a level of performance that is adequate for the given application. Of the programs written, the two that were considered the most significant were the orthogonal mesh generator and the FDTD solver. These two programs were each implemented specially on a massively parallel computer system. This document presents an emphasis on these two parallel implementations.

1.2 FDTD Algorithm and Mesh Generation

In 1966 Kane Yee[67] presented what we now refer to as the finite difference time domain (FDTD) method for modeling electromagnetic phenomenon. The FDTD method is just one tool available in the field of computational electromagnetics (CEM). Many introductory papers to the FDTD technique have been published since Yee presented his seminal paper. Taflove and Brodwin[62] wrote a paper that was one of the first widely recognized overviews of Yee's method. Taflove and Umashankar[63, 64] have also written several overviews, of which two are referenced. Mittra and Lee[40] wrote a paper that discusses the FDTD method in general terms along with the finite element method. Li, Tassoudji, Shin and Kong[29] also present a general overview. In addition to this short list, many more such papers have been published.

The FDTD method directly solves a discrete difference form of Maxwell's equations for closely spaced points on an orthogonal mesh. Since the FDTD method is performed by repeatedly applying the same equations over a mesh in such a way that information is only exchanged between nearest neighbors, the use of an array processor such as the MasPar series is appealing. Unfortunately two potential pitfalls exist between the FDTD method and parallel computation. In chapters 4 and 5, we shall see why boundary conditions as well as effective mapping and memory allocation are critical issues that determine the effectiveness of FDTD implementations on a massively parallel computer architecture.

Chapter 4 introduces many topics related to FDTD analysis and introduces the FDTD method as an algorithm. While the FDTD algorithm could be implemented on any computer system, special emphasis is made to suggest a parallel implementation. Chapter 5 continues where chapter 4 left off by first presenting the implementation of the FDTD algorithm on the MasPar computer system. Chapter 5 also presents results and a characterization of the performance of the implementation.

In reading the derivation of the FDTD equations in appendix E, the reader will note that because of the way that the FDTD method solves Maxwell's equations, the space being modeled must be represented as an orthogonal mesh. Because of this fact, all objects to be included in that space must be represented in terms of an orthogonal mesh as well. Thus in developing an FDTD implementation, a secondary application must also be solved, that is the spatial decomposition of solid objects into an orthogonal mesh. A program written to produce an orthogonal mesh is appropriately called a mesh generation program.

Chapter 2 describes the mesh generation algorithm that was developed specifically for the parallel MasPar computer system. The whole point behind developing the algorithm was to match available resources in the hardware platform to the task at hand. Because of this fact, the wisdom of many of the decisions made in developing the algorithm may not be immediately self evident. In reading chapter 2 the reader should gain a great understanding of the mesh generation algorithm and will be prepared to examine the implementation presented in chapter 3. After reading these two chapters, a reader should understand why the mesh generation algorithm is particularly well suited for an array processor environment such as the MasPar.

In requiring that all objects be represented in an orthogonal mesh, all contours will be modeled as stepped curves. What this means is that the contours associated with objects are not preserved. This situation introduces a problem that is not immediately obvious. The problem is how to provide the user a means of assigning boundary conditions. Since an orthogonal mesh generator does not preserve contours, it is pointless to assign boundary conditions to surfaces of the original solid object descriptions. To assign boundary conditions, it was found to be possible to treat a resultant orthogonal mesh as a solid object and assign boundary conditions directly to the mesh. Appendix F contains the user manual, where you will find a description of the procedure that a user follows to assign boundary conditions. Chapters 4 and 5 introduce concepts associated with boundary conditions and describe how the FDTD solver itself actually handles boundary conditions.

1.3 The Array Processor

Before presenting the MasPar system architecture, it is necessary to review some history and introduce some related topics. In 1966, Flynn[16] gave the name *SIMD* to a group of proposed computers. The term SIMD is an an acronym for "Single Instruction path, Multiple Data path," which according to Hockey and Jesshope[21] is simply a phrase that describes how one category of computers relates its instructions to the data being processed. The term SIMD refers to any computer in which there is a single program instruction stream associated with directing the actions of processors so that they perform operations on multiple data streams.

The phrase "Single Instruction path, Multiple Data path," while descriptive, falls short of fully characterizing any particular class of machine. The Cray line of supercomputers and Thinking Machines Corp. line of high performance computers are two very different examples, that both can be thought of as having a single instruction path and multiple data paths. To get a better idea of what Flynn had in mind when he wrote his 1966 paper, we must look closer at the details he presented. One machine in particular that Flynn referred to was the proposed SOLOMON project which was superseded to become the Illiac IV, according to Flynn;

> "SOLOMON is the classic SIMD. There are n universal execution units each with its own access to operand storage. The single instruction stream acts simultaneously to the n operands without using any confluence techniques (concurrency, esp. pipelining). Increased performance is gained strictly by using more units. Communications between units is restricted to a predetermined neighborhood pattern and must also proceed in a universal, uniform fashion[Fig. 1.2]."

Flynn's comments regarding SIMD architecture are somewhat useful to us. Central are the notions that there is a collection of processing elements that Flynn refers to as "universal execution units...," that are allowed to communicate amongst themselves. Note that the overall processing power of such a system is enhanced by *simply* inserting more processing elements. Flynn's illustration of a SIMD architecture is reproduced as figure 1.2. While Flynn indicates that each processing element has "its own access..." to data memory, Flynn's illustration presents only a mysterious box that all processing elements are able to somehow share.

In his 1972 paper, Flynn[17] reviewed his computer architecture taxonomy and divided the SIMD category into three subgroups;

1. *The Array Processor:* One control unit and m directly connected processing elements. Each processing element has its own registers and storage, but only obeys commands issued from the control unit.

2. *The Pipelined Processor:* This is an array processor, where each processor

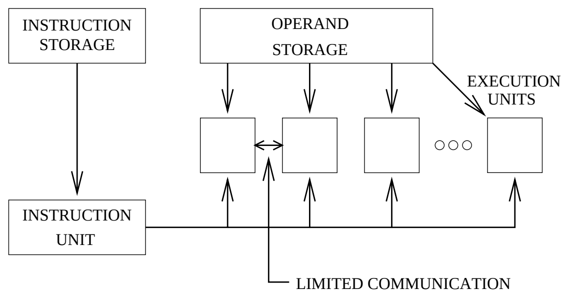


Figure 1.2: Flynn's[16] 1966 Illustration of a SIMD Architecture

element is optimized for a single function type. One processor element might perform floating point addition, another only integer addition, and so on.

3. *The Associative Processor:* This is a variation of the array processor theme, in this case processor elements are not directly addressed by the control unit. Processor elements are activated when a generalized match relationship is found between the contents of a register in the control unit and data in the processor elements. For those designated elements, the control unit instruction is carried out. The other processor elements remain idle.

While Flynn's 1966 paper describes communications between processor elements in a SIMD machine as "restricted" and "predetermined," the associative processor array category presented in Flynn's 1972 paper provides a unique means of coordinating processor elements. Essentially processor elements are either active or idle. Such coordination allows the use of interprocessor communications to be more flexible.

Note that neither of Flynn's papers describes how processing elements are physically organized. Rather than being a "jumble of processing elements," all array processor machines must have their processing elements arranged in some sort of pattern. Such an organization is crucial as it allows a programmer to plan the use of communications between processing elements. Further, note that in both papers, Flynn provides no clue as to how input or output would be handled in a SIMD machine.

According to Hockney and Jesshope[21], Flynn's initial use of the term SIMD was inadequate as it is too broad[3]:

> since it lumps *all* parallel computers except the multiprocessors into the SIMD class and draws no distinction between the pipelined computer

[3]Page 57 of Hockney and Jesshope

> and the processor array which have entirely different computer architectures. This is because it is in effect, a classification by broad function (whether or not explicit vector instructions are provided) rather than a classification of the design (i.e. architecture).

Based on Hockney and Jesshope's observation, it appears that the term SIMD is too ambiguous for our needs, thus the term SIMD is rarely used in this document. The term *array processor* appears to be more apt, and is used throughout this document. While the descriptive phrase *array processor* is actually more general than what we have in mind, in most cases we will assume that the machine has properties of the associative processor.

Lewis and El-Rewini[28][4] provide a reference that serves as a good summary of Flynn's array processor. In such a computer a single control unit is present among many data processors, referred to as *processing elements*. The control unit contains a single copy of the program and a single program counter. The control unit produces a single instruction stream that is *broadcast* to all the processing elements simultaneously. The instruction stream directs the actions of all the processing elements in a *lock-step sequence*. The data however, differs from processor to processor, in a sense there is a multitude of data paths.

To get an even better idea of the specific class of architecture that we have in mind for use in this document, the next section will introduce the reader to one of the machines that Flynn directly referenced, the Illiac IV. As we will see, the Illiac IV can certainly be described as an array processor, and in particular appears to fit the description of an associative processor array. It should be clear later that the Illiac IV serves as a most suitable introduction to the MasPar computer architecture.

1.3.1 The Illiac IV

To get the clearest idea of what an array processor is, we should examine the most famous example of them all, the Illiac IV. The Illiac IV was a one of a kind machine, and was the fourth machine named in honor of the University of Illinois. Hord[23] provides a comprehensive introduction to the Illiac IV, its history, construction, programming and applications. Stevenson[59] focuses on issues of programming the Illiac IV, but also provides a useful introduction. Two things are immediately obvious when reading these texts, first the use of the Roman Numeral 'IV'. It is not

[4]Page 18 of Lewis and El-Rewini

clear why some modern literature, in reference uses the Arabic '4'. Nevertheless, to be consistent with older literature, the Roman numeral 'IV' is used in this document. Also note that since the Illiac IV was actually functional at the time, many of the references cited here are in the present tense. According to Hord;[5]

> "The Illiac IV was the first large scale array computer. As the forerunner of today's advanced computers, it has brought whole classes of scientific computations into the realm of practicality. Conceived initially as a grand experiment in computer science, the revolutionary architecture incorporated a high level of parallelism..."

According to Hord[6], the Illiac IV became operational in November 1975. The Illiac IV actually achieved[7] 200 million operations per second, impressive for that time period. The following is a presentation of the Illiac IV architecture as presented by Hord.[8] The Illiac IV had a single control processor that broadcast instructions to a

> "multitude of processing units termed elements. Each of these processing elements has an individual memory unit... The processing elements execute the same instructions simultaneously on data that differs in each processing element memory."

The Illiac IV had 64 processing elements, Hord continues to state that;

> "In the particular case of the ILLIAC IV, each of the the processing element memories has a capacity of 2,048 word of 64 bit length."

Hord explains how processing elements were organized and how communications between processors was performed.

> "Routing paths are provided... One way of regarding this interconnection pattern is to consider the processing elements as a linear string numbered from 0 to 63. Each processor is provided a direct path to four other processors, its immediate right and left neighbors and the

[5]Page 1 of Hord
[6]Page 1 of Hord
[7]Page 14 of Hord
[8]Page 5 of Hord

> neighbors spaced eight elements away. . . This interconnection structure is wrapped around, so processor 63 is directly connected to processor 0. To transfer values among processors not directly connected, multiple routing steps are required.”

Lastly, Hord explains how the mechanism for coordinating the processing elements worked.

> “The other major control feature that characterizes the Illiac IV is the enable/disable function. While it’s true that the 64 processing elements are under centralized control, each of the processing elements has some degree of individual control. This individual control is provided by a mode value. This mode value for a given processor is either 1 or 0, corresponding to the processor being enabled (“on”), or disabled (“off”). . . mode values can be set independently under program control, depending on the different data values unique to each processing element.”

Thus to review, the Illiac IV had a central controller that broadcast instructions simultaneously to a collection of processing elements. The processing elements followed instructions in a lock-step fashion. Each processing element had its own independent memory and a means of communications between processor elements was present that followed a definite pattern. Lastly the enable/disable function was used as a means of coordinating processing elements. When reading about the MasPar system, try to keep this list of properties in mind.

Lastly, note that the Illiac IV relied on a PDP-11 minicomputer from Digital Corporation to perform key such tasks as compiling source code and providing a connection to the outside world. Nowadays such a computer would be referred to as the *front end*. The Illiac IV was actually used to solve many useful applications. Several applications used the finite difference method, such applications included aircraft fluid dynamics and the seismic study of earthquakes. It is interesting to note that the Space Shuttle was modeled on the Illiac IV.

1.4 The MasPar System

Figure 1.3 presents an illustration of a standard configuration of the MasPar MP–1 computer system. The MasPar MP–1 is a computer system comprised of two units, a high performance UNIX workstation termed the *front end* (FE) and a *data parallel unit* (DPU). The *front end* shown on the table, is a Digital Equipment Corporation workstation. While the system owned by WPI uses a DECstation 5000, older MasPar systems have used the DECstation 3100 or even the VAXstation. Newer MasPar systems use the Alpha workstation. Note that since the front end runs Unix as an operating system, it is sometimes referred to as the *Unix Sub-System* (USS). The DPU is the box shown to the side of the table. The DPU contains an array of processor elements and is where parallel processing is actually performed. A VME connection to the backplane of the front end provides relatively high speed communications with the DPU and serves as a "tight" link. It is important to realize that the front end acts as an intermediary to the DPU, thus normally it is not possible to directly communicate with the DPU. Since the front end is typically networked via Ethernet or the fiber distributed data interface (FDDI), users may access the MasPar system[9] remotely or directly from the attached X-window console.

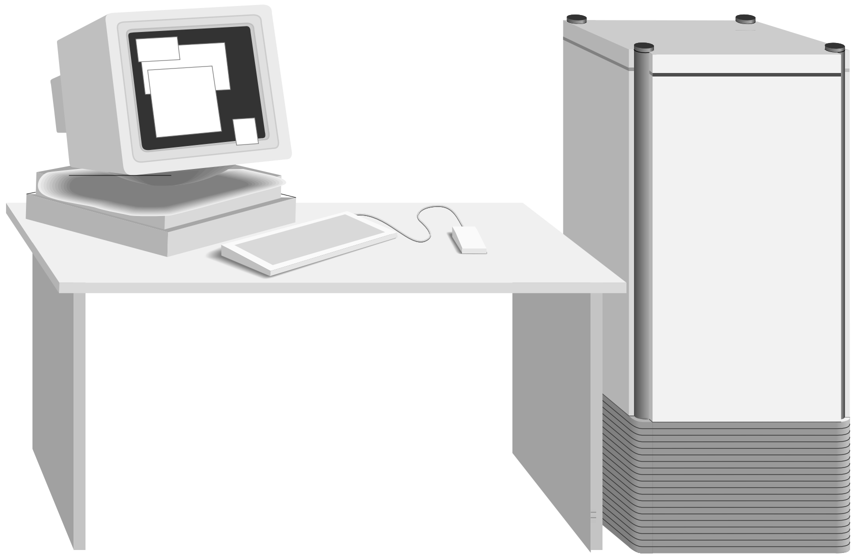


Figure 1.3: Visual of a Standard MasPar Configuration— by permission[39]

For this project all programs were written in MasPar's MPL language as well as ANSI standard C. MPL is actually based on ANSI C, but is considered an extension as it contains certain functions that allow for parallel processing. In addition to MPL, versions of Fortran based on the Fortran/90 standard are available. While it is possible to program the DPU in assembler, MasPar discourages this.

[9]Recall that a MasPar system is composed of the FE and the DPU.

1.5 MasPar DPU Architecture

MasPar documentation[38, 35] provides an ample introduction to the architecture of the DPU, based on this description we can come to the conclusion that the MasPar is another example of an associative array processor. The following can be considered as being paraphrased from the MasPar documentation. We start by presenting the Array Control Unit.

1.5.1 The Array Control Unit (ACU)

The array control unit (ACU) is located in the DPU, it controls the PE array and also performs operations on non-parallel data. The ACU itself is a custom 12 MIPS RISC processor. The ACU was designed with a Harvard type memory configuration with 128k bytes of data memory as well as 1M bytes of program memory, that is virtually mapped as 4 gigabytes. The Array Control Unit essentially acts as the primary control agent in the DPU.

When communications are directed from the ACU to PEs, the ACU is said to '*broadcast* instructions and data to the PEs.' Note that regardless of the number of PEs, all PEs receive broadcasts simultaneously. When PEs simultaneously communicate data to the ACU, the parallel values are said to be *logically reduced* (global OR operation) to a singular value. It is important to remember that while the ACU controls the PE array, the ACU itself is a uniprocessor that is not able to directly receive parallel data, all data must first be converted to singular form.

1.5.2 The PE Array

The MasPar DPU has processor elements (PEs) that are arranged in a two dimensional matrix called the PE array. A system can have 1K, 2K, 4K, 8K or 16K PEs. Note that unless otherwise stated, 'K' refers to multiples of 1024. Each PE is a 1.8 MIPS, four bit, load and store type processor with an arithmetic unit, and its own registers and local RAM. Each PE may have 16K or 64K bytes of physical memory. Because each PE has its own independent memory, in reference to PE memory, the DPU can be referred to as having tightly coupled distributed memory.

In MPL, the variables `nproc`, `nxproc`, and `nyproc` are automatically defined as global variables and contain the number of processors, and the number of columns and rows of processors respectively. Figure 1.4 serves as an example. This array has 64 columns and 32 rows, a total of 2048 PEs.

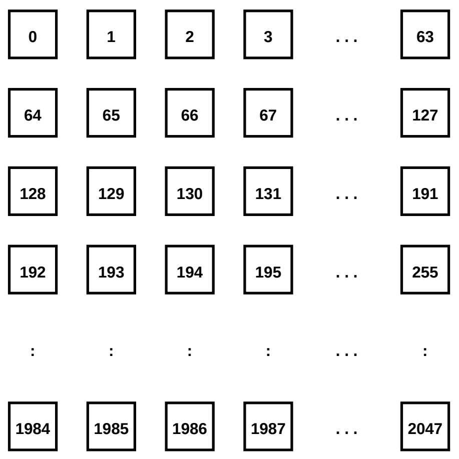


Figure 1.4: PE Array Configuration in 2K System — by permission[36]

Location in the processor array can be represented by row and column, where the corresponding index values are assigned using conventions similar to that of Linear Algebra. The variables `iyproc` and `ixproc` are automatically assigned to each PE and contain the values of the associated row and column, respectively. A PE array with `nxproc` columns will have the left-most column identified as *column 0* and the right-most column will be identified as *column* $(nxproc - 1)$. Likewise, a PE array with `nyproc` rows will have its top row identified as *row 0* and the bottom row identified as *row* $(nyproc - 1)$.

In addition to `iyproc` and `ixproc`, each processor element is automatically assigned a unique identifier named `iproc` that can be determined by the following equation;

$$\texttt{iproc} = \texttt{iyproc} \cdot \texttt{nxproc} + \texttt{ixproc}$$

Figure 1.4 shows inside each PE the associated `iproc` identifier. Having the PEs organized into a simple pattern and being able to identify individual PEs is useful, especially for the purpose of PE coordination, which we discuss in the next section.

In the MasPar system, parallel memory must always be allocated uniformly for all PEs. For this reason, it is useful to think of the variables defined and allocations made in PE memory as being made up of *layers*. Figure 1.5 is an illustration of memory layers in a hypothetical 3×3 processor array.

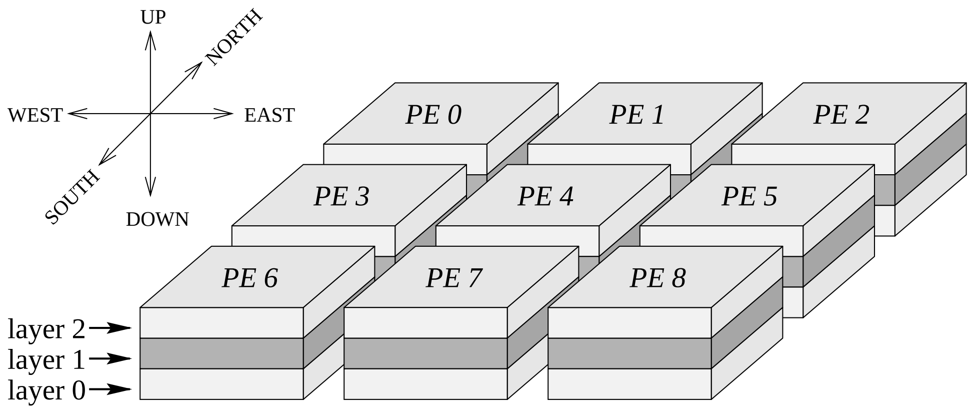


Figure 1.5: PE Memory Layers Allocated

While the gaps between PEs illustrate the fact that each PE has its own independent memory, the shading indicates that the memory can be regarded as *layered.* The layers 0, 1, and 2 might correspond to three plural variables, or perhaps to three entries in a plural array. Note that as more memory is allocated to the user, the layers pile "upwards."

1.5.3 PE Coordination

The MPL Reference Manual[35] provides an introduction to this next important issue, how to coordinate a multitude of processors. During parallel operations, PEs carry out the instructions broadcast from the ACU, in lock step. Note that not all PEs are necessarily involved with parallel computation. The ACU also broadcasts instructions that instruct certain processors to become idle, or alternatively to become active. Such instructions may also tell PEs to evaluate a given logical expression, using parallel data. Processors for which the expression is true become or remain active. Conversely, processors for which the expression is false become or remain idle. This leads to the idea of the *active set.* The active set is simply the set of active processors.

In the following quote from the MPL Reference Manual, the terms *plural controlling expression* and *plural statements* refer to a logical expression that is independently evaluated by each PE.

> The `if`, `switch`, `while`, `do`, and `for` statements of ANSI C are extended in MPL and become plural statements when the controlling expression is plural. You should not think of a plural controlling ex-

pression as being true or false but rather you should think of it as determining which PEs are in the active set.

There are two general principles in the definition of plural control statements:

1. The plural statements control the active set by taking the current active set and further subsetting it by the controlling expression and then restoring the active set following completion of the plural statement.
2. If the active set for a block is empty, then that block is not entered, and conversely, if the active set for a block is non-empty, then that block is entered.

Based on this simple idea of active or idle PEs, the ACU can instruct PEs to simulate various decision patterns. Such control is particularly important for coordinating such operations as communications between PEs as well as for managing the mapping of an application. A simple example from MPL[10] is presented in figure 1.6.

```
if (count < limit)
  {
      /* statements evaluated by active PEs only */
  }
/* previous active set is restored here */
```

Figure 1.6: Example of Processor Coordination

In figure 1.6, the logical expression associated with the `if` statement will be evaluated by all PEs in the current active set. The statements in the block following the `if` statement will only be performed by PEs that evaluate the given logical expression as *true*. In this example, the variable `count` is to be regarded as *plural*, meaning that it is a variable uniformly allocated to each PE. When control exits the given block, the previous active set is restored.

1.5.4 Communications Between PEs

Communications between PEs is an important topic that must be considered. In a MasPar MP–1, two mechanisms are provided to allow PEs to communicate with

[10]MPL is MasPar's programming language. See section1.4 for an explanation.

each other. First, X-Net communications allows processor elements to communicate along straight lines of the processor array. See figure 1.7 for an illustration of the X–Net communication directions. North-South edges are joined, as well as East-West edges. This joining of edges forms the two dimensional mesh into a logical toroid. The toroidal X-Net allows for communications between PEs located along a straight line in any one of eight directions, that is north, northeast, east, southeast, south, southwest, west, and northwest.

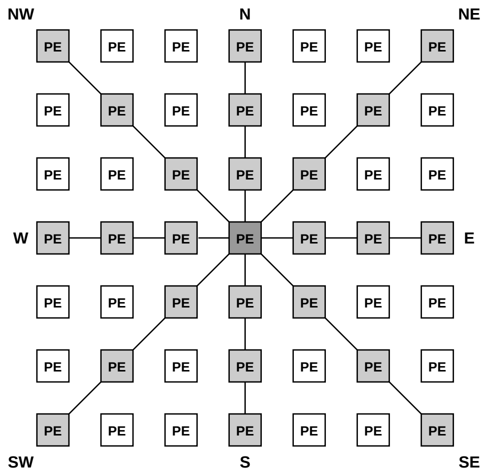


Figure 1.7: Communications Directions for X-Net — by permission[39]

The global router serves as the second mechanism for communications between PEs. The router is a three stage crossbar switch that can connect arbitrary processor elements anywhere in the PE array. For router communications, groups of 16 PEs are organized into clusters, and only one router channel is connected to each PE cluster. Thus only one PE per cluster is allowed to communicate to one PE of another cluster at a time.

The central difference between X-Net and Global Router communications is that the X-Net has "built-in directions of communications" that correspond to the eight primary compass headings, while the Global Router must establish logical connections before data can be communicated. In general, X-Net communications are significantly faster than Global Router communications, but Global Router communications are more flexible.

To fully understand the use of communications between PEs, it is important to understand what is meant by the *connected to* set. X-Net communications allow each active PE to communicate with a PE that is a uniform distance and direction from the active PE. Likewise, the use of the global router allows PEs in the active set to communicate with other PEs. The active set of PEs is said to *initiate* *communications* and access parallel data in the *connected to* set of PEs. Note that PEs in the connected to set may or may not be active PEs.

1.5.5 MasPar Architecture Summary

Figure 1.8 serves as a good summary of the DPU architecture. Note that in this figure the front end (the Unix Sub-System) is simply described as a connected box. As stated earlier, the front end is usually networked thus allowing network access. The Worcester Polytechnic MP–1 is networked via Ethernet and is a client in the local NFS file system. Note the 'ACU-PE' interface between the ACU and the PE array, this interface allows the ACU to broadcast instructions to the PE array, as well as allowing the PE array to return globally reduced data.

Although not shown, the PE array also contains the X-Net PE communications system. The three 'router chip' boxes serve as the global router and also provide a communications path to the I/O subsystem. The I/O subsystem may contain optional I/O and storage devices such as a disk array controller (shown) and a HiPPI bus interface (not shown).

In summary this section provided an introduction to the architecture of the MasPar DPU. For more detail regarding the configuration of MasPar computer systems, see appendix A. Trew and Wilson[66] also provide a useful overview of the MasPar MP–1 and provide some comparison with other modern array processors such as the DAP from AMT and the Connection Machine - 1 from Thinking Machines. All these machines share the sense of having an organized set of processing elements, a single processing element controller, communications between processing elements, and a mechanism of coordination by means of enabling or disabling processor elements.

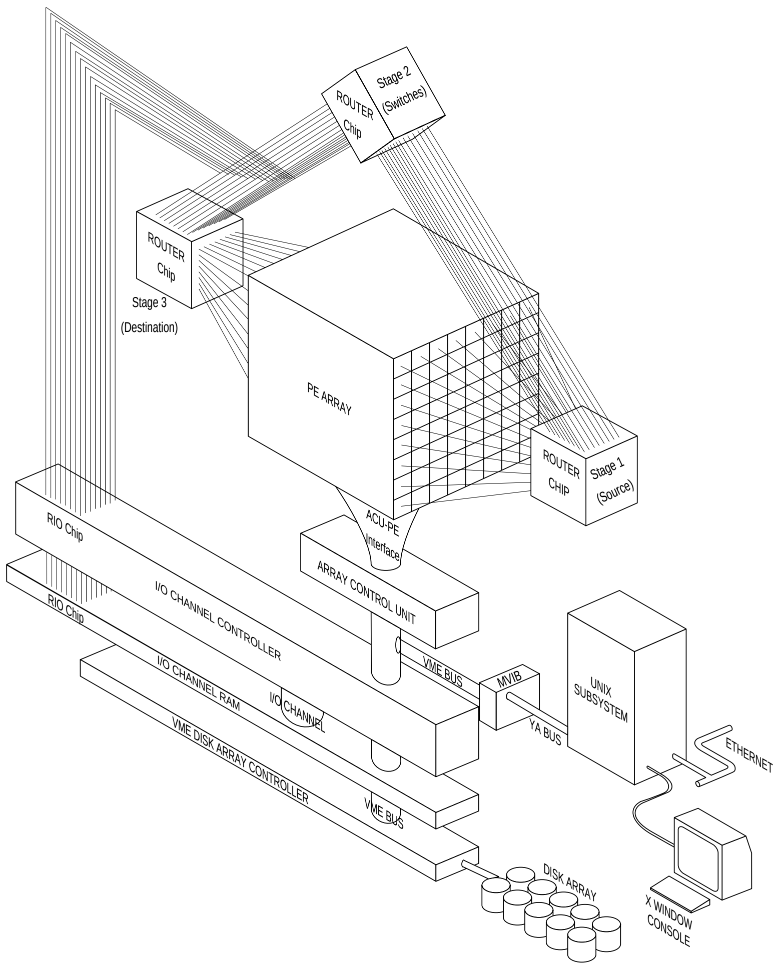


Figure 1.8: Visual of MasPar DPU Architecture — by permission[39]

1.6 The Continuum Model

Stone[60] presents the continuum model as one that is particularly well suited to array processors. Note that this type of model is important to us as it includes the modeling of electromagnetic waves. According to Stone[11];

> "The continuum model accounts for calculations in which time and space are considered to vary continuously, and typical parameters are charge density, temperature and pressure. These are physical measures averaged over regions."

Stone describes the special characteristics associated with continuum problems that allow for a direct solution of differential equations. Based on these characteristics, we can see why processor array machines like the Illiac IV are particularly well suited to solving continuum problems.

> "The nice property of the continuum model is that each point within the continuum acts like an independent autonomous computer. Each point examines its near neighbors to determine their states. Then based on only its current state and the states of its nearest neighbors, each point applies the equation that governs the behavior of the continuum and updates its state."

Stone immediately makes the assertion that if regularly spaced points are defined in a space, a mesh of points is defined. For simplicity, we will assume that each point directly corresponds to a processor in the processor array. Stone continues to clarify the point made;

> "The computations made at each point are made independently and proceed in parallel. So when you visualize convective heat flow, fluid flow, or other physical process, view the dynamics of the process as if each point in the continuum were performing a small computation on local and neighboring data."

In summary, once regularly spaced points are defined in the associated space, it appears that the continuum model is a natural fit for an array type processor like the Illiac IV. Stone continues to state that;

[11]Page 206 of Stone

"The earliest proposals for parallel machines focused on this type of computation, partly because of the natural parallelism inherent in this model, and partly because several important large-scale problems can be solved within a continuum framework."

1.6.1 An Example of a Continuum Model

As an example of a continuum model problem, Stone[12] presents Poisson's equation for the potential in a region as a function of charge density in that region. In two dimensions the equation can be written as:

$$\frac{\partial^2 V(x,y)}{\partial x^2} + \frac{\partial^2 V(x,y)}{\partial y^2} = -C(x,y) \tag{1.1}$$

where $V(x,y)$ is the voltage potential at point (x,y) and $C(x,y)$ is the charge at point (x,y). A solution to this equation in a region depends on the boundary conditions which can be expressed in a variety of ways. Stone chose to assume that we are solving Poisson's equation only for the region $0 \leq x \leq 1$, $0 \leq y \leq 1$ and that we are given the values of $V(x,y)$ on the boundaries of this region.

To determine a direct solution, we must first transform the differential equation into a discrete form that can be treated numerically. In doing so, we represent a continuous region by discrete points as represented in figure 1.9. Each box represents one node in the mesh. The integers in each box correspond to $[i,j]$ and represent its spatial coordinates.

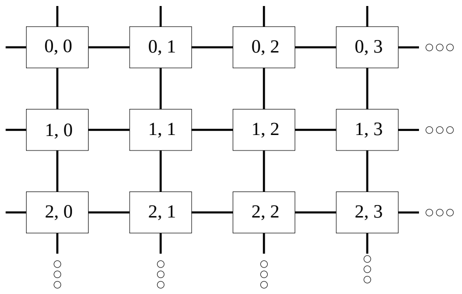


Figure 1.9: A Mesh Representation of a Continuous Space

At the intersection $[i,j]$ in this mesh is a point where we store the values $V[i,j]$ and $C[i,j]$. The indices on i and j can vary from 0 to $N-1$, so the corresponding

[12]See page 212 of Stone

values of x and y are given by[13] $x = i/N$ and $y = j/N$. Stone[14] makes the following point regarding numerical accuracy;

> "If the points in the region are sufficiently close together, we obtain a good approximation of the potential in a continuous region. The fidelity of the discrete version of the problem depends entirely on the mesh spacing. Of course, the number of points grows quadratically with the spacing, so computation time can become very large as spacing diminishes. The user must strike a balance between the resolution of the model and the cost of computation."

This point is particularly significant and will be addressed again with the FDTD solver. Unlike this two dimensional problem, the FDTD solver models a three dimensional space that shows a cubic relationship between the number of points and the mesh spacing.

While the derivation of the solution to the discrete form of Poisson's equation is not presented here, it should be clear that equation (1.1) is converted by modeling differential expressions as finite difference equations. For the complete derivation, the reader is referred to Stone[60]. After some manipulations Stone arrives at a system of linear equations that have the following form:

$$\frac{V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}}{4} = V_{i,j} - \frac{C_{i,j}}{4N^2}$$

$$\text{for } 0 < i < N,\ 0 < j < N$$

Clearly, the potential at each point appears to be the average of the points of its four neighbors, plus a term that reflects the charge located at that point. While it is possible to solve this system of equations by using standard methods from linear algebra, Stone proposes a different method that has great appeal for parallel architectures. The essential idea is to use an iteration of the form:

$$V_{i,j} = \frac{V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}}{4} + K_{i,j} \tag{1.2}$$

Where the $K_{i,j}$ term is defined as follows:

$$K_{i,j} = \frac{C_{i,j}}{4N^2}$$

[13]To be able to maintain this mapping and make reasonable surface contour graphs, we define y, x, V to be a right handed coordinate system.

[14]Page 212 of Stone

By repeatedly applying equation (1.2), the values stored in the mesh will approach the correct solution to the discrete form of Poisson's equation.

A computer architecture for solving the equations given in the form of equation (1.2) is shown in figure 1.10. Each mesh point has an associated processor. Note that each processor in turn has its own memory which it uses to store its own values for the variables 'V' and 'K'. As a means of communication, each processor is somehow connected to its four immediate neighbors. Since all processors execute the same iterative equation, we need only one instruction stream for all processors. Instructions are broadcast by a single control processor and are received and performed by the processors.

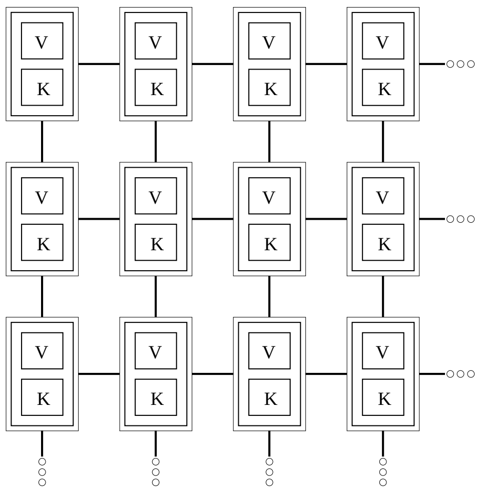


Figure 1.10: An Array of Processors for Continuum Model Calculations

In his example Stone goes so far as to present source code used to perform an iteration of the Poisson solver, as if it were implemented on the Illiac IV. In section 1.6.2, the source code written for the MasPar to solve Poisson's equation is presented. Section 1.6.3 presents results from the Poisson solver.

1.6.2 A MasPar Program Example

This section presents an example of a program written in MPL to provide a direct solution to Poisson's equation. The program is presented along with results. Figure 1.11 illustrates how boundary conditions were assigned in the example. Three boundary edges were assigned a potential of 10. The remaining boundary was assigned a potential of 0. There are 32 PEs along each of the coordinate axes, for a total of 1024 PEs.

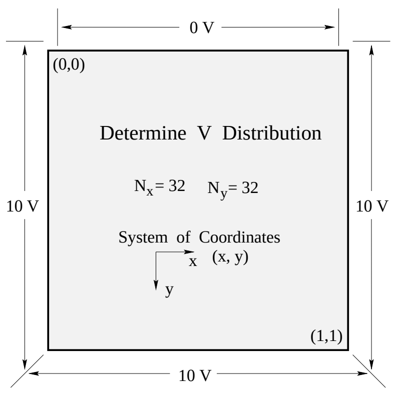


Figure 1.11: Boundary Conditions assigned to Example

Note that because we followed Stone's[60] convention of having the origin in the upper left hand corner of the two dimensional space, we use the y, x, V right handed coordinate system whenever we generate surface contour plots. To make things sane, the usual (x,y) ordered pair is used to refer to points on the surface. To keep your mind clear, its best to associate x with a PE column and y with a PE row.

Figure 1.12 is a listing of the Poisson equation solver, written in MPL. Readers familiar with C should be able to recognize many details in the program listing. As stated in section 1.4, MPL is an extension of ANSI standard C. The following is a discussion of the program listing, for more detail the reader is referred to the MPL Reference Manual[35].

The first thing that you might notice in the listing is the `plural` keyword that is used with the variable declaration. The keyword `plural` states that a variable is to be uniformly defined on all PEs, this implies that there are *nproc* instances of the variable. If the keyword `plural` is not used, a defined variable is assumed to be singular, meaning that it will be stored in ACU memory. Note that in contrast to plural variables, there is only a single instance of a singular variable.

In this situation, the `proc` construct is used inside a loop to assign boundary conditions to certain PEs. The `proc` construct is useful as it allows access to a plural variable on a single PE. The syntax of this use of the `proc` construct is as follows:

```
proc[row][col].plural_expression
```

The terms `row` and `col` refer to the row and column of the referenced PE. Note

```
/****************************************************************************
  Poisson's Equation -- Author: Jonathan Hill

  This program provides a direct solution of Poisson's Potential Distribution
  Equation.  See Chapter 4 of Stone's text for more information.  This program
  was written for the MasPar 1101 with 1024 PEs.  This source code is in MPL.
  A percent difference is used to estimate closeness of convergence.

****************************************************************************/
#include <mpl.h>
#define  TRUE  1
#define  FALSE 0
main (int argc, char *argv[])
{
  plural float voltage = 0.0, nvoltage = 0.0;
  plural float Kcharge = 0.0, temp;
  plural float percentDiff, epsilon = 0.001;
  plural int   not_done = TRUE;
         int   index, cflag = TRUE, limit = 1000;

  /* user enter a different iteration limit? */
  if (argc > 1)
    sscanf(argv[1], "%d", &limit);

  /* Insert boundary conditions into array while still in active set. */
  for (index = 0; index < 32; ++index ) {
    proc[31][index].voltage = 10;
    proc[index][0].voltage  = 10;
    proc[index][31].voltage = 10;
  }
  /* disable boundary processors */
  if ((ixproc != 0) && (ixproc != 31) && (iyproc != 0) && (iyproc != 31 )) {
    for (index = 0; index < limit && TRUE == cflag; ++index) {

      voltage = nvoltage;
      nvoltage = (  xnetN[1].voltage + xnetS[1].voltage
                  + xnetE[1].voltage + xnetW[1].voltage )/4 + Kcharge;

      if ( voltage != 0.0 ) {
        /* Estimate closeness to convergence */
        /* ABSV is a macro that returns the absolute value */
        temp = (nvoltage - voltage)/voltage;
        percentDiff = ABSV( temp );

        if ( percentDiff > epsilon )
          not_done = TRUE;
        else
          not_done = FALSE;
        cflag = globalor(not_done);
      }
    }
  }
  make_Maple_file("test", "A", index, &voltage);
} /* end of Poisson Solver */
```

Figure 1.12: Source Listing of Poisson Solver

that `row` and `col` must be singular expressions. The term `plural_expression` is the plural variable that is accessed on the referenced PE.

It is important to remember that while the ACU is in control of the parallel part of the DPU, the ACU itself is not a parallel machine. Because of this fact, the programmer must be aware of at all times, how the ACU is relating to the PE array. The `proc` construct is useful as it allows the ACU to address a single PE from a multitude of PEs. If the `proc` construct were not used, as in '`voltage = 0.0`' then all instances of `voltage` present in the active set would have zero assigned as a new value. There are times when such an assignment is useful, we used this technique to initialize our plural variables.

While the `proc` construct is used here to assign a value to an instance of a plural variable, the `proc` construct can also be used on the right side of the equal sign to retrieve an instance of data and assign it to a singular variable. This raises an important point, the ACU itself cannot directly relate to plural data, plural data must be *reduced* to singular form before the ACU can relate to it. The `proc` construct is just one of the means of reducing plural data, we will see another way in the listing.

After the boundary conditions are assigned, an `if` statement similar to figure 1.6 is used to disable the PEs containing boundary condition values. The terms `iyproc` and `ixproc` correspond to the PE row and column values, these terms were introduced in section 1.5.2. Note that only PEs related to the interior of the 2D space are allowed to participate in the computation of the potential distribution, thus the active set contains 900 PEs, this is approximately 87.89% of the total number of the PEs.

A `for` loop repeats the equation used to solve for the distribution. The expression, `voltage = nvoltage` stores values from the previous iteration into `voltage`. Note that the boundary condition values stored in `voltage` are never overwritten, because the related PEs are not in the active set. The following statement in the listing is simply equation (1.2), written in a unique way. note that the expression is evaluated simultaneously by all PEs in the active set.

The `xnet` construct is used to access data in a neighboring PE. Although we only use the *plain access* type in the listing, there are actually three different types of xnet communications available. The plain access type was selected as it is the simplest and most commonly used form of the `xnet` construct. The syntax used is:

```
xnetDIR[dist].plural_expression
```

Recall that PEs in the active set initiate the communications, thus the the term `DIR` refers to the access direction, from the point of view of PEs in the active set. In place of `DIR`, the symbols `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, and `NW` are used to represent the eight compass directions. Since the `xnet` construct can be used on either the left or right hand side of an equal sign, to understand the direction that data is actually moving, you must be aware of whether the `xnet` construct is being to make an assignment or to retrieve data. This is to say that data may be *retrieved* from an access, or may be *assigned* in an access. Clearly in an MPL expression, an `xnet` construct to the left of the equal sign is being used to to make an assignment. Conversely an `xnet` construct to the right of the equal sign is used to retrieve data.

The variable expression `dist` determines the distance over which communications will be performed. The variable expression `dist` must be must be singular, and can be a constant. As with the `proc` construct, `plural_expression` refers to the plural variable that is accessed by xnet communications.

In the listing, xnet communications are performed with accesses made to the North, to the South, to the East and to the West. Since these accesses are to the right of the equal sign, the accesses are used solely to retrieve data. Since the communications distance is specified to be one, the accesses are made only with the nearest neighbors. The plural variable being accessed is of course `voltage`, which is used to contain values from the previous iteration.

There are two ways that the program can exit from the iterative loop. First, if the iteration count variable `index` equals or exceeds the set `limit` variable, then the loop will exit. The value of `limit` is set initially to a large value, but the user may specify a different value. Alternatively, the loop will exit if a test indicates that the solution has settled, this test is discussed next.

After each iteration, each PE compares its previous value of `voltage` to its new value stored in `nvoltage` by calculating the absolute value of the percent difference, as shown by equation (1.3).

$$\text{percentDiff} = \left| \frac{\text{nvoltage} - \text{voltage}}{\text{voltage}} \right| \tag{1.3}$$

An `if` statement is included to ensure that no PE attempts to perform division by zero. The result from the equation (1.3) found by each PE is next compared against a threshold to determine on a local level if data appears to be settled. Next, each PE assigns a binary value to an instance of the plural variable `not_done` to indicate the outcome of the comparison. The function `globalor` performs a bitwise OR operations on all instances of the variable `not_done` that are present in the current active set. Thus, the `globalor` function provides another means of *reducing* plural data to singular form. Note that if even one instance of `not_done` is TRUE, then `cflag` will be TRUE. For `cflag` to be FALSE, all instances of `not_done` in the active set must be FALSE. When `cflag` is FALSE, indicating that the solution has settled, the iterative loop will exit.

After the iterative loop exits, the last thing that the program does before execution ends is to write a Maple[15] readable file. The actual function used to write the file is not shown here.

1.6.3 Results from a Continuum Model

This section presents results from the program listing in figure 1.12. This program uses Poisson's equation to solve for the potential distribution in a two dimensional space. Figure 1.11 summarizes how the boundary conditions were assigned to the two dimensional space. As indicated in section 1.6.1, Poisson's equation is solved by relatedly applying equation (1.2). After each iteration, the distribution approaches the solution of Poisson's equation.

Recall that three boundary edges were assigned the potential of 10V, while the remaining boundary edge was assigned the potential of 0V. There were no charges included in this example. Figure 1.13 shows the converging distribution after ten iterations. The program exited with a stable solution after 471 iterations, figure 1.14 shows the resultant potential distribution. Note that the results agree with with what we expected.

1.6.4 Program Performance

To complete the example, we make an estimate of the floating point performance. Remember, the purpose of this simple example is to present the reader with a brief introduction to the concept of performance analysis. Section 1.7 presents performance analysis in a broader sense, and section 1.7.5 in particular provides a more in depth introduction to MFLOPS as a performance metric.

To perform equation (1.2) once calls for 4 add operations and one divide operation, a total of 5 floating point operations. Since 900 PEs are used to perform the solution and a total of 471 iterations were actually performed, equation (1.2) was performed 423,900 times, which corresponds to 2,119,500 floating point operations.

[15]Maple is a software product of the University of Waterloo, Canada

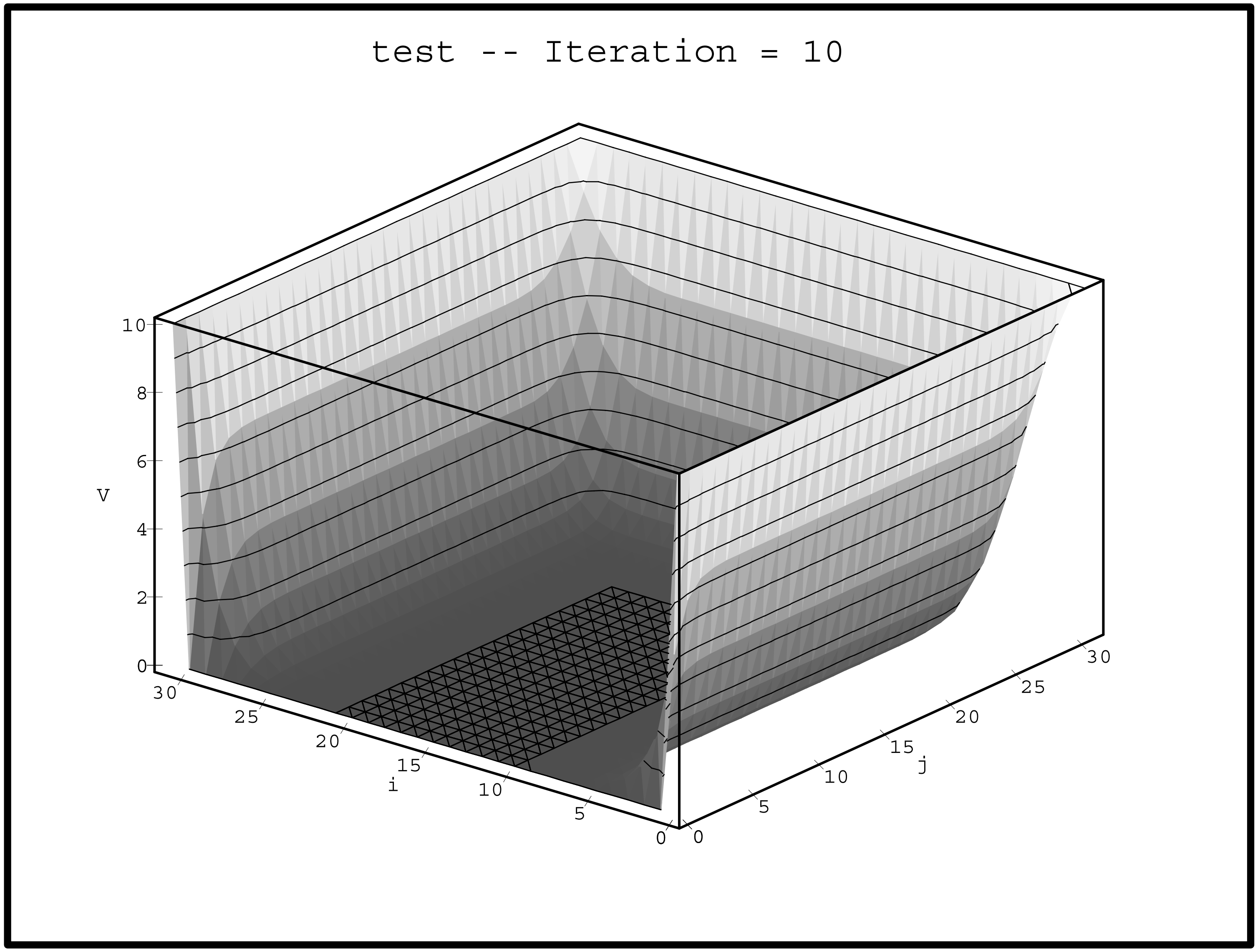


Figure 1.13: Solver Results After 10 Iterations

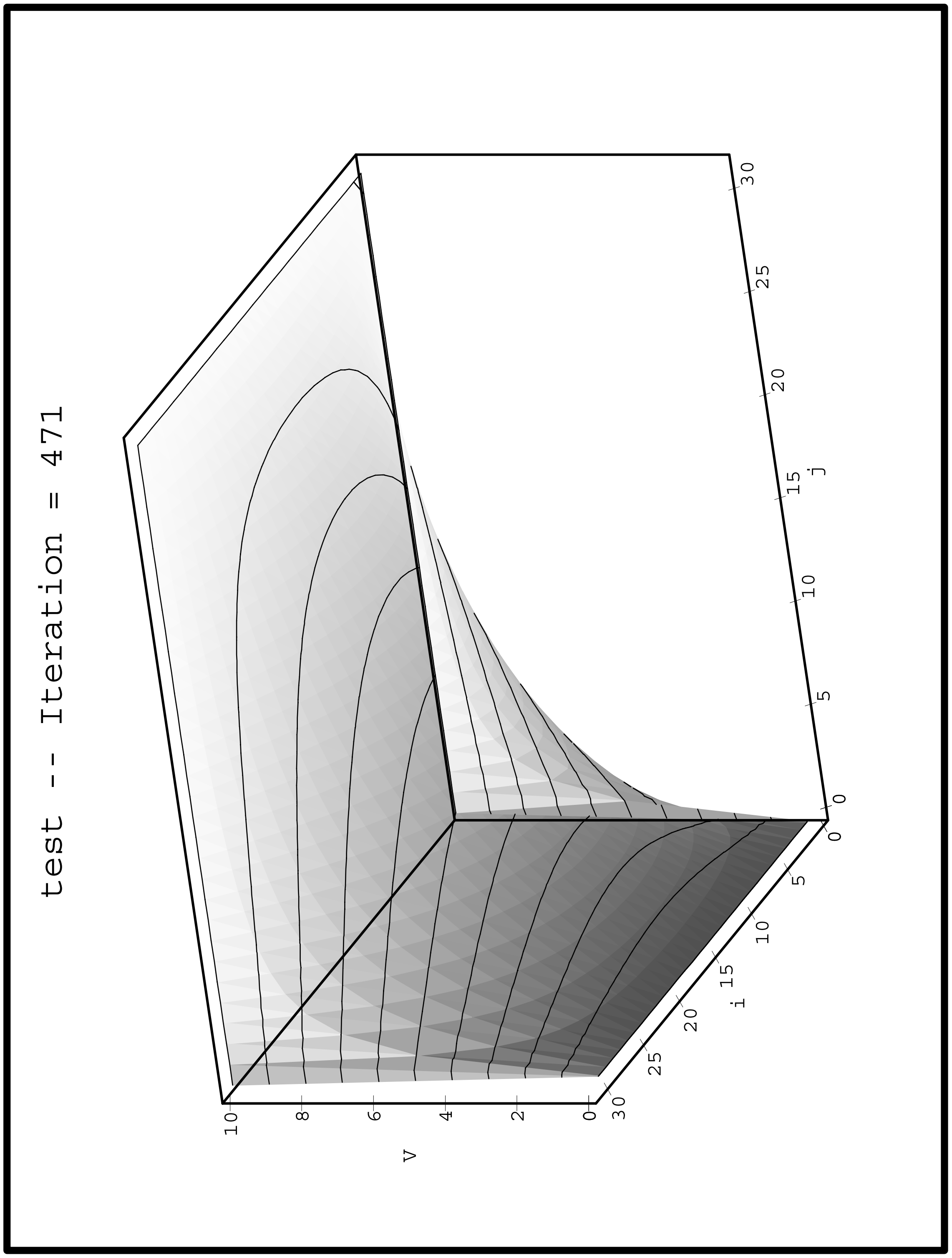


Figure 1.14: Solver Results After 471 Iterations

To determine the number of times that the percent difference equation was performed by each PE, a plural counter was inserted into the program. The `reduceAdd` function was used to reduce by addition, the total from each PE, to an overall total. It was found that the percent difference was actually performed 417,820 times. Since a subtraction and a divide operation are needed to perform a percent difference, this corresponds to 835,640 floating point operations. Together, this is a total of 2,955,140 floating point operations.

To determine the execution time of the Poisson solver itself, the call to the function `make_Maple_file` as well as the plural counter described above were both "commented out" and the MasPar profiler tools were used to determine the CPU time. The Poisson solver was executed five times and the average CPU time was found to be 207.8 milliseconds, which corresponds to a floating point performance of 14.221 million floating point operations per second. Note that such a performance figure by itself has little meaning. To give the performance figure meaning we must examine the context from which the performance figure was derived.

To get a sense of how well the hardware is actually being used, we will determine the floating point rate of a hypothetical model based on the underlying algorithm. We start by determining the amount of time that is required to perform the floating point operations. Equations (1.2) and (1.3) represent the two situations in which floating point operations are performed. Each time equation (1.2) is performed, four additions and one division is completed. To be able to perform these floating point operations, five values must be loaded from parallel memory and one value must be written back to parallel memory. Similarly, each time equation (1.3) is performed, one subtraction and one division operation are performed. Similarly, performing equation (1.3) requires that two values be loaded from memory and one resultant value be written back to parallel memory.

The MasPar MPL user's guide[37] provides the average timings to perform the types of floating point operations that we have been discussing. Based on the given timing information, it is a simple matter to estimate the time during one iteration that the MasPar system should be using to perform the associated floating point operations, see table 1.1.

Note that while the MasPar MPL user's guide gives timing figures in terms of clock cycles, since we know the that the clock period of the MP–1 is 80 nanoseconds, we can immediately determine that the 1944 clock cycles corresponds to 155.52 microseconds. It is important that the reader bear in mind that this timing figure corresponds to seven floating point operations.

Table 1.1: **Floating Point Operations Summary**

Floating Point Operation	MP–1 Clocks	Operation Use	Total MP–1 Clocks
add	126	4	504
subtract	126	1	126
divide	315	2	630
load	76	7	532
store	76	2	152
		Grand Total:	1944

We can easily construct a hypothetical model, just by assuming that such a program requires no input, produces no output, and has no control structures or need for interprocessor communications. Such a program would only contain equations similar to (1.2) and (1.3), but repeated a large number of times. While such a program certainly has no practical use, it does provide a theoretical limit that can serve as a means of comparison. Equation (1.4) presents the limit for this example.

$$\text{MaxRate}_t = \frac{(\text{f.p. operations})\ (nproc)}{\text{f.p. time}} = \frac{7\,(1024)}{155.52 \times 10^{-6}\text{sec.}} = 46.091 \text{ MFLOPS} \tag{1.4}$$

By dividing the observed rate by this theoretical peak figure, we get a number that represents the percentage of the peak performance figure. The actual program floating point performance rate was found to be 30.85% of the peak performance rate that we specified, which is certainly respectable. Upon closer inspection of the program it can be seen that there certainly is some room for improvement. By being clever, this code could be significantly optimized for improved performance. Note however, rather than doing so, this example has already served the purpose of being a vehicle for introducing the continuum model, and the topics of program performance and optimization. All these topics will be addressed more fully in the remainder of this thesis.

1.6.5 Continuum Model Summary

The continuum model was discussed as such problems are readily handled by array processors. In addition to the direct solution of Poisson's equation that was presented here, the FDTD method presented in chapter 4 provides a solution to Maxwell's equations. Both Poisson's equation and Maxwell's equations fall into the continuum category of problems.

A special comment needs to be made in regards to boundary conditions. The handling of boundary conditions are important as they can make a large impact on program performance. In the Poisson solver for example, because of the way that boundary conditions were handled, over 12 percent of the PEs in the processor array were not able to participate in providing a solution. The handling of boundary conditions is a particular problem that shows up with the handling of all continuum models. In reading this thesis, please make note of the special attention given to the handling of boundary conditions.

1.7 Introduction to Performance

Performance is an important issue as it one of the principle ways of determining how well developed a solution is. Patterson and Hennessy[44] provide a useful introduction to the metrics of performance. While Patterson and Hennessy emphasize hardware performance, our emphasis in this document should be to determine how well a given implementation makes use of the given hardware. After all, it is not the goal of this document to compare the MasPar MP-1 to other computer systems. Unfortunately the two topics, hardware performance and hardware use are inseparably linked. It is not really possible to gauge hardware performance without the use of some program implementation. Likewise it is not possible to determine how efficiently hardware is being used unless some optimal situation can be defined, such an optimal situation might be based on the observed asymptotic performance of a program or alternately the model of a theoretically ideal program can developed from data associated with the hardware. Thus the concepts of hardware performance and efficient hardware use are in a sense "married" to each other.

1.7.1 Time as a Performance Metric

In the following discussion[16], Patterson and Hennessy introduce time as the most fundamental metric for measuring performance.

> "Time is the measure of computer performance. . . Program *execution time* is measured in seconds per program. But time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*. This is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead—everything."

Wall clock time is easily measured by use of the Unix `time` command or by use of the `gettimeofday` function in 'C'. By examining the time at the beginning and the end of the program execution, a difference is easily found that is equal to the elapsed time. Note that since computers are often time shared, wall clock time may not provide an accurate description of the performance of any single program

[16]Page 50 of Patterson & Hennessy

in a computer system. Patterson and Hennessy make a comment to further stress this point; Most time shared systems may be tuned for optimizing certain types of performance.

> "the system may be optimizing throughput (amount of data processed) rather than attempting to minimize the elapsed time for executing one program."

In situations where the performance of an individual program needs to be known in the context of the system in which it is used, the use of wall clock time is appropriate. If however the goal is to characterize the performance of the program itself, independent of the rest of the work that the computer is performing, wall clock time is not appropriate. To take care of this discrepancy, Patterson and Hennessy introduce an alternative means to measure time.

> "we often want to distinguish between the elapsed time and the time that the processor is working on our behalf. *CPU execution time* or simply *CPU time*, which recognizes this distinction, is the time the CPU is computing for this task and does not include time spent waiting for I/O or running other programs... CPU time can be further divided into the CPU time spent in the program, called *user CPU time* and the CPU time spent in the operating system performing tasks on behalf of the program..."

The concept of CPU time immediately makes sense for a single processor computer system, in such a system CPU time is simply the amount of time that the CPU is working on our behalf. For serial code there is a function called `getrusage` that accesses CPU time. Unfortunately, CPU time does not immediately make sense in concurrent computers. In an array processor system such as the MasPar for example, because of the ability to disable PEs, the active set of PEs working on our behalf at any moment in time, may actually be only a fraction of the total number of PEs in the system. Thus the concept of CPU time is out of place in this context. Because of this discrepancy, processor time can weighted by the percent of the number of PEs actively used in the array. This introduces the concept of PE use efficiency which is presented later in this section.

To measure the ammount of time that the DPU is working on our behalf, MasPar suggests the use of *DPU time*. According to MasPar literature[17], DPU time is defined as,

[17]For the definition of DPU time see the man page for `mpGetRUsage`

> "the total time that the process has been executing in the DPU. This includes time when the DPU is stalled waiting for the front-end process. It does not include time when some other DPU job is running."

For parallel code the `mpGetRUsage` function accesses CPU time as well as DPU time. Note that DPU time does include time waiting for the front end to perform services for the DPU. In addition note that there is no distinction made here for DPU time in terms of time spent on user code versus the operating system.

To complicate things, we must consider that the DPU is not working alone. The front end serves as host and assists in performing work. Whether a programmer chooses the synchronous process model or the asynchronous process model, it is not unusual for the sum of the CPU time reported by the front end and the DPU time reported by the processor array to exceed the measured wall clock time. This observation comes from the fact that process control is not instantaneously passed back and forth between the front end and the DPU. There is overlap in processes that are performed in the front end and the DPU. This overlap allows the front end and DPU to coordinate activities in which they both must participate. Such activities include page swapping, transferring data, and performing house cleaning chores.

For our purposes, the MasPar profiling tools are particularly useful for providing an overall view of where processes are spending time. The profiling tools provide a breakdown, function by function of how much time is spent by the front end, the DPU, or in performing coordinated activities. The profiling tools provide a useful summary of how and where time is being spent in the MasPar system. If it can be shown by use of the MasPar profiling tools that the time spent executing code in the front end is insignificant compared to the time spent executing code in the DPU, we will ignore the front end CPU time, and we will only be concerned with DPU time.

As with Patterson and Hennessy, we will maintain a distinction between performance based on wall clock time (elapsed time) and performance based on CPU or DPU execution time. We will use the term *system performance* to refer to elapsed time on an **unloaded** system, and *CPU performance* or alternatively *DPU performance* to refer to performance based on these other measures of time.

1.7.2 Time Complexity as a Performance Metric

The goal of time complexity analysis is to find a function that characterizes the relationship between execution time and problem size. Intrinsic to the concept of complexity analysis is the notion that certain properties of such a functional relationship are solely dependent on the underlying algorithm. Time complexity can provide insight to important properties of algorithms that will play a role in determining the performance of an implementation. For this reason, time complexity analysis has been recognized as an important tool for comparing algorithms.

Time complexity analysis can be performed in two ways. From a theoretical viewpoint, time complexity analysis is primarily concerned with the asymptotic relationship between some hypothetical measure of time and problem size, as problems become larger. This type of time complexity analysis is performed by simply examining the algorithm itself. Section 2.6, presents the idea of time complexity as a theoretical tool.

From an experimental viewpoint, time complexity can be performed by fitting an equation to experimental data. One reason for performing this alternate type of time complexity analysis is to support the validity of a theoretical time complexity analysis. The goodness of such a fit provides an indicator of how reasonable our theoretical analysis is. Naturally for such experimental data to have any meaning, the problem size should not be too large as to require a significant amount of page swapping. Remember, the goal of time complexity analysis is to discover characteristics of the execution time that are independent of the hardware actually used.

1.7.3 PE Use as a Performance Metric

The goal of processor use figures is to characterize how well a computer program makes use of a given computer system. Here we consider only a few such processor use figures: processor use due to mapping and processor use due to branching. Also, the percent of the theoretical maximum sustained MFLOPS figure can be thought of as a processor use figure that applies only to floating point intensive programs, see section 1.7.5.

We first consider the processor element use figure due to mapping (PUFM), which is associated with what Flynn[17] referred to as the problem of *vector fitting.* The PUFM describes how well the mapping does at fitting the problem “into” the processor array. In most cases a problem will be unevenly divided among the PEs. Some PEs might receive a smaller chunk of the problem than other PEs, still other PEs may not receive any part of the problem. Naturally, PUFM is a function of the size and geometry of the processor array, the size and geometry of the problem, as well as the effectiveness of the processor mapping used.

A useful way to think of PUFM is that by use of a specific virtualization[18], the processor array is made to appear to the algorithm as a processor array with some other geometry. The virtual array being represented might simply be a larger two dimensional array. In such a virtualization, each processor element will represent one or more virtual processors, where the relationship is defined by the specific virtualization. In this sense, PUFM is the percentage of the total number of available virtual processors that are actually used by the algorithm. Note that the PUFM can always be determined theoretically from first principles, but in certain circumstances can be determined by comparing the execution times for different problem sizes. This is just an introduction, the concept of PUFM is discussed in far more detail in chapters 3 and 5 of this document. In addition, Pickering and Cook[46] present a simple example of a cut-and-stack mapping that should give the active reader some insight into these concepts.

The second processor use figure we consider is the processor use figure due to branching (PUFB), this is what Flynn[17] called the problem of *degradation due to branching.* According to Flynn;

> "When a branch point occurs, several of the executing elements will be in one state, and the remainder will be in another. The master controller can essentially control only one of the two states; thus the other goes idle.

The PUFB term describes how many PEs the algorithm makes use of whenever it encounters a branch statement. Recall from the introduction to the concept of the active set, given in section 1.5.3. When evaluating a plural control statement, it is only necessary for one PE to evaluate the statement as true, for the ACU to start execution in the associated block of code. In such an undesirable situation, the computational power of an entire PE array will be reduced to that of a single PE. While the technique of setting PEs either active or idle is useful as a means of coordination, the technique must be used carefully.

[18]A discussion of specific virtualizations and processor array mapping is presented in appendix section C.3

For most parallel programs, data memory is mapped during program initialization, into a specific configuration that is unchanged until the program terminates. For such a program, once initialized, the PUFM remains constant. Thus, PUFM can be regarded as being as static as the data mapping. In contrast to PUFM, PUFB is dynamic in that it is related to the size of the current active set. PUFB is usually specified as an average and can be specified either at specific control statements, or in a more global manner. If a branch statement is rarely encountered or if the the branch statement is executed very quickly, its impact on the overall performance will be negligible. Note that both PUFM and PUFB are defined to be ratios of numbers of PEs.

1.7.4 Speed-Up as a Performance Metric

Patterson and Hennessy[44] introduce the idea of speed-up in general terms. Speed-up is defined as the measure of how a system performs after some enhancement is made relative to how it performed before. Speed-up is expressed simply as a ratio;

$$\text{speed-up} = \frac{\text{Execution Time Before}}{\text{Execution Time After}} = \frac{T_{\text{before}}}{T_{\text{after}}} \tag{1.5}$$

To make use of the idea of speed-up, consider the following. Suppose that some part of a given program can be performed in a more efficient manner and further, suppose that part of the program accounts for $\psi \cdot 100\%$ of the total execution time. If the part of the program to be improved can actually be performed an *enhancement factor* of E_f times faster, then the new overall execution time should be;

$$T_{\text{after}} = \left(\frac{\psi}{E_f} + (1-\psi)\right) T_{\text{before}} \tag{1.6}$$

With the expression for the new execution time, speed-up can be found in terms of ψ, E_f and T_{before} by simply substituting into the general case speed-up equation. After simplifying, it is found that T_{before} cancels out to yield the following expression;

$$\text{speed-up} = \frac{E_f}{E_f(1-\psi) + \psi} \tag{1.7}$$

Note that the general concept of speed-up can be used to examine nearly any type of performance enhancement. The concept of speed-up is first used in section 2.9 when anticipating how much faster a parallel version of the mesh generator might be in comparison to a serial version. Other uses of speed-up considerd by this document include changing the method used to perform a key task, and changing the number of processors in a system.

Often what is important, is not the performance speed-up related to a single problem size or associated system configuration, but the general relationship between performance, problem size and the number of processors. Essentially the amount that performance *scales* with problem size or the number of processors is often more important than any single value of speed-up. This idea of performance scaling is closely linked with the idea of complexity analysis which is introduced in section 1.7.2.

1.7.5 MFLOPS as a Performance Metric

MFLOPS is simply the floating-point processing rate that a given implementation achieves for a given problem size. A floating-point operation is an addition, subtraction, multiplication, or division operation applied to a pair of numbers in single or double precision floating-point representation. To determine the FLOPS rate, simply divide the total number of floating point operations performed by a program, by the execution time.

Basic Assumption

It is important to realize that the use of MFLOPS as an absolute performance metric comes along with a certain necessary assumption that must be understood. Patterson and Hennessy[19] provide a clear introduction to this assumption;

> "Clearly a MFLOPS rating is dependent on the program. Different programs require the execution of different numbers of floating-point operations. Since MFLOPS were intended to measure floating-point performance, they are not applicable outside that range. Compilers, as an extreme example, have a MFLOPS rating near [zero] no matter how fast the machine is, because compilers rarely use floating-point arithmetic."

The reader must be aware that when examining a MFLOPS rating, an assumption is being made that the only goal of the associated program is to efficiently

[19]page 62 of Patterson & Hennessy

perform floating point operations. Unfortunately, since it is possible to determine the MFLOPS rate of any program, the given assumption may not be observed by the person. By this argument, it should be clear that the meaning and usefulness of a given MFLOPS rating, for the purpose of characterizing the absolute performance of a given program is only as good as the truth of the stated assumption; The only goal of the program is to efficiently perform floating point operations.

The example presented by Patterson and Hennessy above should drive the point home. It should be plainly clear that compilers are useful programs, and accordingly compiler performance must be an important topic. As pointed out by Patterson and Hennessy however, since compilers rarely use floating point operations, the use of MFLOPS is inappropriate for use as a tool for characterizing the performance of compilers. While there are programs for which the use of MFLOPS is certainly useful, there are also cases where the use of MFLOPS is evidently pointless. In between these two extreme cases, there is a certain "gray area" where the use of MFLOPS is at best questionable. To attempt to avoid ambiguity with each use of MFLOPS, this document incorporates a discussion of the validity of its use in context.

Instruction Mix

Patterson and Hennessy provide further insight into the use of MFLOPS as a performance metric.[20] Patterson and Hennessy introduce the concept of floating point instruction mix.

> "the MFLOPS rating changes not only according to the mixture of integer and floating-point operations but also on the mixture of fast and slow floating-point operations. For example, a program with 100% floating-point adds will have a higher rating than a program with 100% floating-point divides."

Depending on how efficiently a computer performs each floating-point operation, a programmer can make decisions on how to perform the necessary floating-point operations. Rather than multiplying by two, often it is more efficient to add a number to itself, for example. Such decisions will effect the MFLOPS rating as well as the overall execution time. Besides the instruction mix the streamlining of a program control structure will alter the percentage of the time that is actually

[20]Page 62 of Patterson & Hennessy

used to perform floating-point operations. Thus reducing the time consumed by other operations can have an effect on the MFLOPS rating as well as the overall execution time.

Sustained Performance Ratio

As indicated earlier, it is not really possible to determine how well hardware is being used unless some optimal situation can be defined. One way to define such an optimal situation is to construct a theoretical model in which the only work a program actually performs is floating point operations. In such a model, we ignore everything except operations directly associated with floating point math. Since MasPar has published the amount of time on average that is required to perform each floating point operation, based on the mixture of floating point operations that a program actually uses, we can define a theoretical maximum floating point rate for any given program. Besides floating point operations, the only allowance made is the time that is required to move floating point arguments from local memory to registers as well as the time to move results from registers to local memory. This allowance is made so that the theoretical maximum rate will correspond to a sustained floating point performance rate.

Clearly such a theoretical sustained maximum performance figure assumes that the only goal of a program is to efficiently perform floating point operations. Thus the meaning of the theoretical sustained maximum floating point performance rate closely follows the fundamental assumption that was stated earlier. Naturally it should be impossible for any useful implementation to actually achieve such a theoretical maximum performance, but still the ratio of the actual sustained MFLOPS rate to the maximum theoretical sustained MFLOPS rate tells us how good a job the program is doing at instructing the hardware to perform floating point operations. We refer to this ratio as the *percent of the theoretical maximum sustained MFLOPS figure*. This document uses the name MFLOPS_%TMS to refer to this performance figure.

To summarize, MFLOPS characterizes the rate at which floating-point operations are performed for a program implementation. The MFLOPS_%TMS figure provides a comparison of a given MFLOPS rate with a defined maximum rate. In comparing implementations, unless the application and selected algorithm can be closely linked to the goal of efficiently performing floating-point operations, the use of MFLOPS as a program performance metric will be questionable. Since the field of computational electromagnetics and the FDTD method in particular are well known to be floating-point computation intensive, the use of MFLOPS is a useful metric for the FDTD solver.

1.7.6 Alternative Metrics

In some cases performance metrics such as FLOPS rate, PUFB, and PUFM do not provide a full characterization of the absolute performance of an application on a specific hardware platform. For the evaluation of such programs, PUFB and PUFM only provide a relative performance measure, and the use of the FLOPS rate may at best be questionable. To get a meaningful measure of the absolute performance, one can resort to an *invented metric.* Such a metric is referred to as "invented" as it only has meaning for evaluating programs that do similar work. You can think of MFLOPS as being an invented metric for example. We used an invented metric to evaluate the absolute performance of our mesh generator, see section 3.8.1.

Reconsider the performance evaluation of a symbolic compiler written for a serial machine. As presented in section 1.7.5, the use of flop rate is pointless for such an application as compilers do not use floating point math operations. As an absolute performance metric, *lines per second* is often used in evaluating compilers. Of course, *lines per second* is an invented metric, but its use should make sense to anyone who has ever used a compiler. While this metric is somewhat dependent on the size and structure of programs being compiled, many different programs can be compiled, and an *average lines per second* value can be determined. Such a measure provides an indicator that may have some meaning to someone who wants to buy a compiler.

1.8 Summary

This chapter presented an introduction to the central topics of this thesis, that is the development of two programs for the MasPar computer system. The first program is a mesh generator that provides a spatial decomposition of three dimensional objects. The second program is an implementation of the Finite Difference Time Domain Method. To make the need for these two programs plain, the application of these two programs to antenna design was presented.

Since the array processor computer architecture may not be familiar to all readers, before presenting the MasPar system, this chapter presented some history of the array processor. The continuum type problem was introduced along with an example that serves as a vehicle for introducing the concept of performance analysis. Following the example, specific metrics used to measure performance on an array processor were given. This material is important as performance analysis is an important part of the content of this thesis.

In chapter 2, the algorithm associated with the mesh generator is presented. Section 2.1 presents a high level introduction that the reader is strongly suggested to read. An important part of the presentation in chapter 2 is the time complexity analysis presented in section 2.7.3. The parallel implementation of the mesh generator is presented in chapter 3. The mapping required by the mesh generator is presented in section 3.2. The performance analysis of the mesh generator starts in section 3.5 and essentially continues to the end of the chapter.

Chapter 4 presents important topics associated with a parallel implementation of the FDTD method. Basic topics associated with FDTD analysis such as the Yee Cell and handling boundary conditions are presented as well. In section 4.5, a discussion presents the theoretical basis for the material we use for a suitable absorbing material for modeling absorbing boundary conditions. Section 4.6 presents the Finite Difference Time Domain Method as an algorithm, complete with a time complexity analysis.

Chapter 5 present the parallel implementation of the FDTD algorithm presented earlier. Of particular importance is the mapping which is presented in section 5.1, and the performance analysis which starts in section 5.4 and essentially continues to the end of the chapter.

Chapter 6 presents the conclusions of this thesis. Suggestions for future research are made, along with general comments regarding the research and lessons that were learned while writing this thesis.

The appendixes contained in this thesis must not be overlooked. Appendix A provides a summary of the MasPar systems that we ran code on for this thesis. Of the three electromagnetic simulations presented, the one presented in Appendix B is the most complete. See section 1.1.2 for details regarding all three of the simulations presented in this thesis.

Appendix C serves as a source of useful information for those who have more than a casual interest in developing an application for the MasPar system. Appendix D presents some formulas that are associated with cross products. These formulas were found to be useful for the development of parallel mesh generator. Appendix E presents the derivation of the finite difference equations that are central to the FDTD method. Appendix F contains the user manual and lastly Appendix G contains the list of references for this thesis.

One last comment needs to be made; This thesis does not contain much source code. Besides short examples that are scattered about, it was decided that due to the volume of the source code associated with this research, it would be unreasonable to include it all here. This thesis will be archived with all related source code, on tape as well as on CD-ROM. You may also contact the author via email at `jmhill@ece.wpi.edu`, or the world wide web at `http://www.wpi.edu/~jmhill`, but please keep your correspondence brief.

Chapter 2

Parallel Mesh Algorithm

This chapter presents the algorithm used to produce the parallel implementation of the orthogonal mesh generator. To aid understanding, the mesh generator is first presented as a serial algorithm. It is important to point out however, that even the serial mesh generation algorithm was developed with a focus on making it suitable for a processor array type computer.

Section 2.1 provides a high level explanation of basic mesh generation concepts. The rest of chapter 2 provides an algorithm that can be derived from the simplistic ideas first presented in section 2.1. It is suggested that the reader quickly peruse section 2.1 before "descending into the depths" of the program algorithm and SIMD implementation and refer back to section 2.1 as needed. Without a mental image of the terrain, you cannot easily "tell the forest from the trees." Section 2.2 presents definitions and in depth details of topics that are just touched upon by section 2.1. Sections 2.3 and 2.4 present the mesh algorithm as a collection of tasks that are repeatedly performed. Section 2.7 presents a time complexity analysis. Section 2.11 is a chapter summary that neatly ties up loose ends.

In developing the mesh generator it was found to be useful to first implement the algorithm in serial form. Remember that it was our goal to produce a parallel implementation. In general, deciding to produce a serial version first, usually does not make much sense. This situation however represents special case as this algorithm allows the parallel version of the mesh generator to be a natural extension of the serial implementation. Chapter 3 is devoted to explaining how the algorithm presented in chapter 2 was implemented in a parallel environment and will provide a better explanation of exactly why the implementation works so well.

2.1 High Level Introduction

It is the goal of this section to provide the reader with a “feeling” of the overall algorithm. With this goal in mind, it was decided to deliberately leave out exacting definitions from this section to help the reader gain a better sense of that “feeling.” The rest of this chapter and the next should provide ample rigor for the active reader.

A solid object is interpreted to be the interior of a given closed surface. Such a closed surface is essentially the shell of an object. The method used to describe such a closed surface is detailed in section 2.2. Files that contain such object descriptions are of a type named *solid files*. A solid file usually has a .sld file name extension. Figure 2.1 is a representation of a four sided object and is an example of an object that can be described using a solid file.

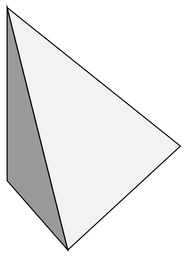


Figure 2.1: Representation of a Simple Four Sided Solid Object

Note that object surfaces are represented by interconnected surfaces. Such surfaces are analogous to the flat surfaces of a cut diamond. As with a diamond, such surfaces are referred to as *facets*. Facets are linked by *edges*, which are lines that simply represent the intersection between surfaces. Detailed definitions for such items as facets are presented in section 2.2.

Next consider what is actually produced by the mesh generation software. The first thing to be defined is the region that will contain the resultant mesh model, the region is said to be contained in a *bounding box*. A bounding box is easily defined by specifying two points, P_{max} and P_{min} in three dimensional space, as shown in figure 2.2. This particular bounding box will be referred to as the *mesh bounding box*. The interior of the mesh bounding box is divided into an array of adjacent, *solid boxes* with regularly spaced edges. Each edge must be parallel to a given coordinate axis.

The mesh model is produced by assigning an identifier corresponding to each

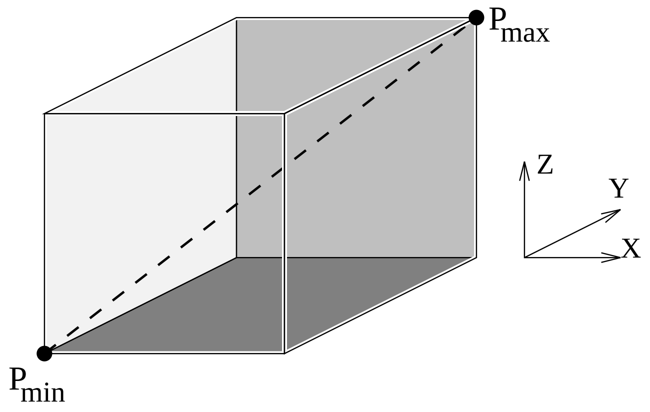


Figure 2.2: A Three Dimensional Bounding Box

object being modeled, to boxes in the mesh such that the shape of the original objects are approximated. The identifier corresponding to each object is referred to as a *material identifier* and indirectly describes what the solid object is made of. Several objects may have the same material identifier. The idea of constructing a model with plastic toy bricks such as LEGO[1] brand toy bricks is a useful analogy.

Since the shape of the mesh array is understood implicitly, a list of the mesh boxes completely describes the resultant mesh. Boxes in the mesh that have not been assigned material identifiers take on the default material identifier which is zero. The figure below is a visualization of a resultant mesh made to represent the four sided object presented earlier. Such visualization files can be generated to help the user to "see" the resultant mesh.

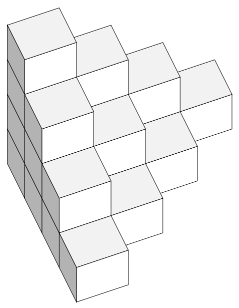


Figure 2.3: Visualization of a *(4x4x4)* Mesh Model

Mäntylä refers to output as in figure 2.3 as a *decomposition model* and explicitly refers to this type of decomposition model as an *exhaustive enumeration*.[2] Hoffman

[1]A trademark of LEGO systems.
[2]Mäntylä[31], page 60.

uses alternative jargon to refer to such output as *spatial decomposition by uniform subdivision of space.*[3] In either case it is clear that such a mesh of boxes is an approximate representation of the shape of three dimensional solid objects and that the "goodness" of any such model is dependent upon the mesh subdivision size. Use of the phrase subdivision size in this sense is similar to that of grain size in photography where reference is made to the size of small bits of silver derived from silver halide compounds. Adams[4] explains how photographic grain size is related to image sharpness. Theoretically if the size of each mesh box were to shrink to an almost infinitesimal size then such a decomposition model could represent a perfect likeness. It should also be obvious that any attempt to approach such a limit would be impractical and would require nearly an infinite amount of computer memory.

While this document does discuss memory use as well as memory allocation, it is beyond the scope of this document to discuss in general the relationship between mesh density and mesh "goodness." This particular topic is left to the discretion of the program user. Note however, it is anticipated that the primary use of the mesh model software will be to produce models for use with the accompanying finite difference time domain (FDTD) solver software. For this reason, section 4.1.4 provides advice to the FDTD solver user in the choice of an appropriate mesh density.

The issue of how to assign property identifiers to mesh boxes amounts to deciding which boxes in an array are owned by which object being modeled. Exactly in the center of each box is a special point that is referred to as a *grid point.* A special point must be defined outside the volume containing objects being modeled. The special point is referred to as the, *search origin.* A line segment can be drawn from the search origin to an arbitrary grid point. The line segment is referred to as the *search line.*

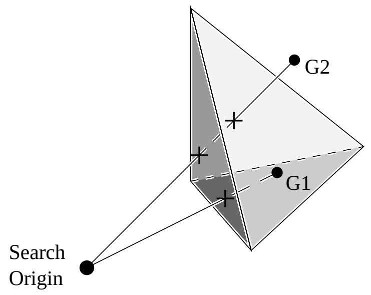


Figure 2.4: Search Lines Examining a Solid Object

The mesh generator algorithm is based on the following basic idea: If the number

[3]Hoffman[22], page 63.
[4]Adams[1], page 19.

of intersections between a search line and an object surface is odd, the grid point associated with that search line must be located inside the object. Kalay[24] refers in general terms to this as the *parity count method*; This identifiable name will be used throughout this document to refer to this basic idea. Given that the parity count method can be used to determine if a grid point is inside a solid object, the box in the mesh model corresponding to that grid point will be said to *belong* to that object.

In figure 2.4 the search line associated with grid point G1 has one intersection with the object surface. Since one is an odd number, G1 must be inside the object. The box associated with G1 must belong to the object. The search line associated with grid point G2 has two intersections with the object surface. Since two is not an odd number, G2 must be outside the object. The box associated with G2 cannot belong to the object. With the ability to determine if an arbitrary point is inside a solid object and the ability to define a regular grid of arbitrary points, it is a simple matter to define a suitable mesh, as in figure 2.3

While in most cases the search line is to be interpreted as a finite length line segment, there are cases when it is helpful to see a search line as being unbounded in both directions, that is both ends extend to infinity. In such situations the phrase *extended search line* or *unbounded search line* is used. The phrases actually indicate that we are referring to an unbounded line that is coincident with the search line. Thus every search line has an associated extended search line. Do not be confused however, the search line itself is simply a straight line segment that extends from the search origin to a given grid point and thus the search line has finite length.

While the parity count method may appear "obvious," it is deceptively simple and far more powerful than first apparent. Unfortunately such a simple idea also presents certain pitfalls, thus implementing such a simple idea really was not a simple task. First off, since the parity count method only describes how to examine one shape, provisions must be made to allow for the modeling of multiple shapes. Issues related to handling multiple objects and shapes are discussed in section 2.2.3.

Edges also present a particular problem. As Kalay[24] points out, the problem arises whenever a search line is coincidental with *shape elements* such as edges. Such coincidence may result from one of two different cases illustrated in figure 2.5 below: (1) The case represents a transition from an internal region of the object to an external region in 3D space (or vice versa). For the parity count method to produce correct results, such a coincidence should be counted as a surface intersection. (2) The case may represent a tangency condition in which the object surface is not penetrated and no inside to outside transition occurs. Clearly this second type of coincidence should not be counted as a surface intersection. Kalay refers to both types of coincidence as *singularities*. Misinterpretation of such singularities may result in a wrong intersection count, turning the model inside-out at such corresponding points.

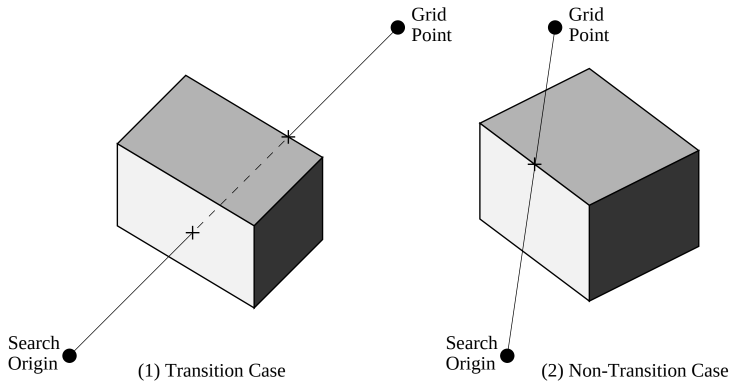


Figure 2.5: Possible Cases for Singularities

Kalay points out that avoiding singularities in a 3D analysis has “undesirable side effects” which have to be solved somehow at a cost which is possibly higher than of solving the singularities themselves. According to Kalay, singularities can be avoided two ways;

1. Use another search line that does not experience the same singularities.
2. Displace the object description in some predetermined manner that will eliminate the singularities.

Kalay goes to great lengths to explain how such singularities can be handled. It was decided that such techniques are far too computationally expensive, especially for an array processor type environment where multitudes of processors all execute the same instructions in lock-step. Requiring nearly all processors to be idle so that a single processor can resolve a singularity is unreasonable. A simple method for handling singularities was devised. The regular spacing of grid points is a particularly useful property that is easily used to handle singularities.

According to our technique, which we refer to as the resolver method, when a search line encounters a singularity the corresponding grid point is referred to as an *unresolved grid point* and will not be processed further until all facets that describe the object are processed. Grid points that never experience a singularity are referred to as *immediately resolvable grid points*, and are easily handled using the parity count method. Unresolved grid points are handled next. An unresolved grid point inside the object should notice that its neighbors are inside the object. Conversely, an unresolved grid point outside the object should notice that neighbor grid points are not inside the object. By making such an observation of neighbors, a decision of insideness can easily be made for every unresolved grid point.

In a sense no technique is used at all to eliminate or even solve singularities, the technique directly determines whether grid points that experience singularities are inside or outside the object. For the resolver method to work however, it is necessary that the probability of encountering singularities be small so that neighboring grid points can provide a reliable indication of insideness. A simple technique for reducing the probability of encountering singularities was devised. Such a technique reduces the possible number of singularities encountered but cannot possibly eliminate all singularities and corresponding unresolved grid points. The techniques used for reducing the probability of encountering singularities and the technique of handling unresolved grid points are introduced in section 2.4.6. Section 2.4.5 provides techniques for detecting singularities.

One last comment regarding singularities. Singularities are not only encountered when using the parity count method to analyze three dimensional objects, singularities also may also be encountered when using the parity count method to analyze two dimensional structures. The topic of handling singularities when performing the parity count method in a two dimensional analysis will be discussed later in this document.

After all objects have been examined and all mesh boxes appropriately assigned material identifier numbers, the task of the entire algorithm is nearly complete. The property identifier assignment list is saved in a specified file, visualization files may be generated, lastly the program terminates.

2.2 High Level Details

This section expounds several of the items that were just touched upon by the high level introduction. First a few definitions.

2.2.1 Definitions of Geometry

Most standard geometric definitions encountered in this document are formally defined in the field of topology. Blackett[11] wrote a popular book on topology. In addition geometric definitions are given in many books. The following definitions are paraphrased from Kalay[24]. In his presentation, Kalay refers to Behnke, Bachmann, Fladt and Kunle[9] as well as Giblin, Chapman & Hall[19].

Definition 1: The Polygon

A *polygon* is the figure formed by choosing a sequence of n arbitrary points $P_1, \ldots, P_n$, joining each point with the next by a line segment, and joining the last point with the first one. The points $P_1, \ldots, P_n$ are the *vertices* of the polygon, and the segments $P_1P_2, \ldots, P_nP_1$ are its *sides.* A vertex bounds exactly two sides. Figure 2.6 is a general polygon.

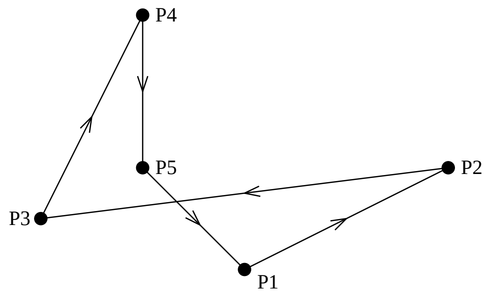


Figure 2.6: A General Polygon

A polygon is said to be *planar* if all its sides lie in one plane, otherwise it is said to be *skew.* If sides intersect only at common bounding vertices, then it is said to be *simple.* It is said to be *straight* if all its sides are segments of straight lines. Note that the given definition of a polygon does imply an *ordering of the sides,* fixed by the sequence of joining vertices in P.

Figure 2.7 illustrates examples of simple, straight, planar polygons that are convex as well as non-convex. A polygon is said to be *convex* if any straight line, coincident with a side of the polygon, does not intersect any other side of the polygon. In the figure, associated straight lines are shown as dashed lines. In the non-convex case, lines, each coincident with a side also intersect a different side. The side intersections are illustrated as circled dots.

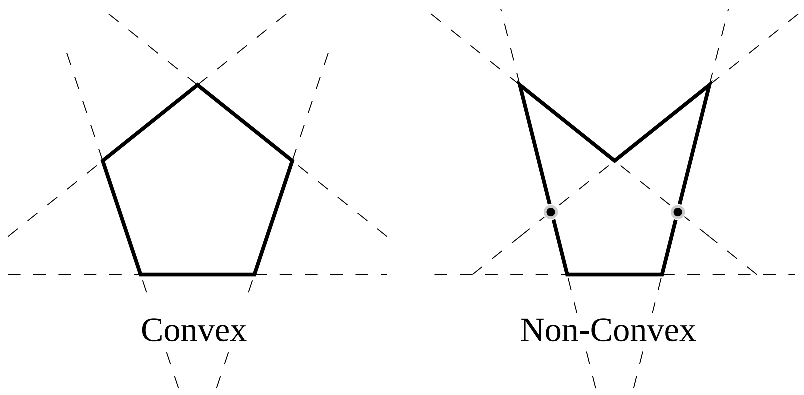


Figure 2.7: Simple Polygons

Definition 2: The Shape

A *shape* is a system of simple polygons with the property that every side of each polygon is also a side of exactly one other polygon in the system. The polygons are called the *faces* of the shape, their sides are its *edges,* and their vertices are its *vertices.* The system as a whole is called the *surface* of that shape.

In this document, only a special class of faces are considered. Shapes are constructed only with planar, simple, straight polygons. In this document such planar, simple, straight polygons are called *facets.* The definition of a shape easily can accommodate facets, since they are just one class of faces.

A corollary of definition 2 is that every edge is bounded by exactly two vertices and belongs to exactly two faces. The definition implies closedness of the surface, but does not imply connectivity or orientability. *Connectivity* means that there exists a path on the surface from any vertex to any other vertex of the shape. Not imposing this requirement means that a shape could be composed of multiple *disjoint shells*, each consisting of a *connected* set of faces. However, to be able to establish a containment relationship between a point in space and the shape, it is necessary to add the *orientability* condition, so that each shell of the shape will partition the 3D space into two disjoint domains, exactly one of which is unbounded, commonly denoted *outside*, and the other is bounded and denoted *inside.* Points which are members of the inside domain are said to be *contained* by the shape.

Orientability of a surface can be defined in different ways, for example by the Möbius Edge Rule. The orientation of each face is simply the order of the defining vertices. Since each side of a face is bounded by two vertices, an *orientation* is easily assigned to each side, as a vector that indicates the order of the defining vertices. Consider an edge common to two adjacent faces. If the two sides that form the single edge are oppositely oriented, then we say that the faces are *coherently oriented*. For shapes this means that the normals to all their faces point to the same domain of the 3D space, as partitioned by their surface.

Shapes whose surfaces are made of connected and orientable shells, that have the property that the points shared by any pair of faces are only those of their common edge or vertices (that is, they do not self intersect), are said to be *well-formed*. As in Kalay's paper, only well-formed shapes are considered by this document. It is important to note that well-formedness is a *global* characteristic of the shape. It cannot be determined with local data alone by simple data manipulation. Figure 2.8 illustrates why the non-self-intersection requirement is not easily checked. Shapes A and B are topologically the same, but geometrically they differ in the location of vertex V relative to the rest of the shape, making A a well-formed shape and B an ill-formed one.

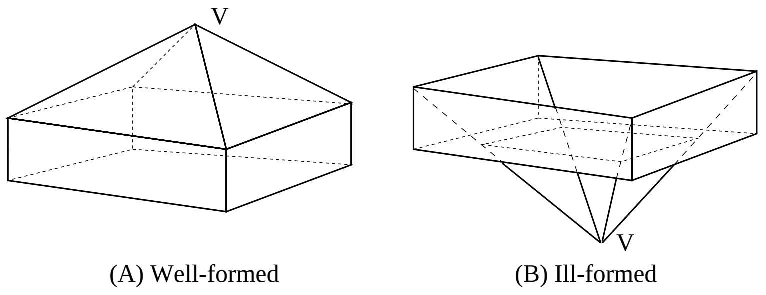


Figure 2.8: Globality of the Self-Intersection Condition

Clearly since a shape can only have an associated inside and outside, shapes by themselves are not able to immediately describe all *objects* that might be encountered in the real world. In this document the following condition applies, shapes can only be loosely referred as objects if it is implicitly understood that the interior of such an object is homogeneous with uniform material properties throughout.

2.2.2 Clarifying Closedness

The importance of the requirement that a shape surface be closed in exactly the way stated in the previous section is self evident in figure 2.4 on page 49. The assumption that every shape to be analyzed by this algorithm, must be described by a closed surface that contains some finite volume, is intrinsic to the task of defining the insideness or conversely the outsideness of grid points in the model.

If a shape does not contain some finite volume, then it is not possible to correctly determine if a point can be contained by that shape.

2.2.3 Analyzing Multiple Shapes

The next significant topic can be considered a major pitfall. The issue was first touched upon when considering how it might be possible to analyze an arbitrary number of shapes. Problems arise as the parity count method really only describes how to analyze individual shapes. These problems are complicated by the fact that individual shapes only readily describe objects that are solid and uniform. During the development of the algorithm it became clear that only analyzing uniformly solid objects presents a handicap. We discovered the need to analyze more complex objects.

Since by basic definition a closed surface can only imply the sense of exterior and interior, a constructive method of describing more complicated objects was proposed, that of *overlapping shapes*.[5] The method of overlapping shapes is particularly useful for constructing descriptions of objects that are not solid or are conglomerations of several materials. The basic idea can be described in the following simple terms; A hollow object can be described by taking a shape representing an empty interior and enclosing it inside a shape representing a solid object. The resultant description can be said to have walls with certain thickness. Any object described in this way is said to be *hollow*.

It is important to point out that even if it is not the user's intention to overlap shapes as a means to constructively describe objects a problem still exists. A problem may arise when shapes are closely spaced, particularly when shapes are supposed to share common faces. If special care and caution is not observed when describing a geometry, closely spaced shapes may actually overlap. Unlike physical reality, shapes are simply abstract descriptions and can easily occupy the same space, thus there is no danger of breaking any natural laws. In any circumstance, a method of handling such a situation is needed, otherwise the results of the parity count method may be unpredictable whenever the situation arises.

[5]The term *overlapping shapes* is confusing. While it is not possible for real three dimensional objects to occupy the same physical space, shapes can be considered as more basic abstractions of objects. The term *overlapping* was first introduced to describe a condition that exists for shapes that are closely spaced, which is explained in the next paragraph.

Enclosed Method

The following is an introduction to a method that was considered for handling the problem of overlapping shapes. It turns out that this method is too complicated and an implementation was not attempted. Intrinsic to this method is the notion of one shape enclosing another shape, for this reason the name *enclosed method* was given. A shape is said to be enclosed by another if one shape contains all the vertex points of another shape. In general any set of points can be said to be enclosed by a shape if every point in the set is contained by that shape. The method proceeds as follows, when analyzing intersections between search lines and shape surfaces compare the distance from each intersection to the search origin. Given a situation where one shape contains all the vertex points of another shape, intersections associated with a contained shape will be further from the search origin than other intersection points. Thus this simple observation can be used to determine the relative enclosure of shapes.

In situations where several shapes claim containment of a given grid point, the enclosed method provides a rule. All shapes that are enclosed have privilege over the shapes that they are contained by. Such an enclosed shape would be assigned all the grid points it contains, leaving the remaining grid points to other shapes.

Even though the enclosed method can be used to determine which shape is enclosed by which shape, it does not solve the larger problem. First, the enclosed method does not solve the problem of closely spaced shapes. Second, the enclosed method becomes far more complicated for shapes which are only partially enclosed by other shapes, figure 2.9 represents just one such example.

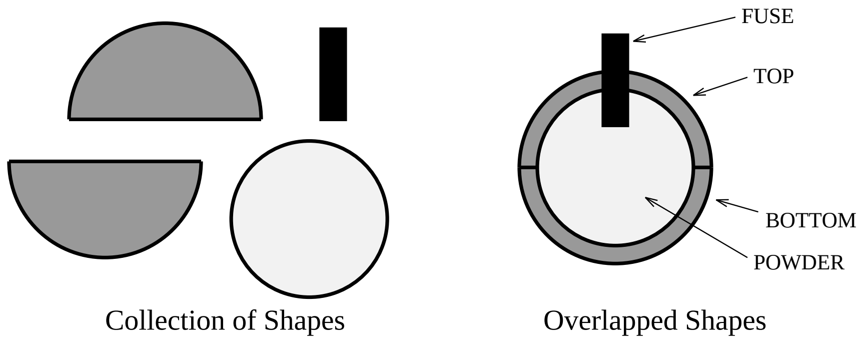


Figure 2.9: Construction of Bomb by Overlapped Shapes

Using the enclosed method, the resultant mesh model would depend on how the search line is oriented relative to the bomb. In one particular orientation, parts of the fuse would be detected as being enclosed but other parts of the fuse would not be detected as being enclosed by the bomb. In a sense only part of the fuse is enclosed. While this simple fact makes sense to humans, it is not readily handled by the simple enclosed method. Likewise since the bomb shell is made of a top part and a bottom part, it is unclear if the powder would be detected as being enclosed by the shell or if the shell would be detected as being enclosed by the powder, a statement that may at first appear like nonsense.

While the enclosed method could be modified to handle such situations, it soon becomes unwieldy and complex. It is important to realize that for a high performance implementation, the required solution must be simple. The choice of the above example was only made to graphically illustrate the severity of the overlapping shapes pitfall. The enclosed method was not implemented, a more simple technique was found to be satisfactory.

Precedence Method

The first method implemented to address the issue of how to analyze multiple shapes was exceedingly simple. Each shape was individually analyzed, one at a time by using the parity count method. After a shape had been analyzed, all grid points contained by that shape were assigned to that shape. In a sense, shapes were given the opportunity to steal all grid points away from shapes that had been previously analyzed. This method presents two problems. First, the resultant mesh will be dependent upon the order in which shapes are analyzed. Second, the handling of such a situation can quickly become cumbersome and complex.

An improvement was devised, making this the method of choice for handling multiple shapes. By taking the magnitude of the material property identifier assigned to each shape into account, a simple order of precedence is achieved, thus the technique is referred to as the precedence method. The idea is analogous to the concept of layer depth that is commonly used with two dimensional mechanical drawing software. In such software,[6] drawings can be thought of as being made on layers of mylar film. Drawings on top layers may cover up parts of drawings made on lower layers. This method is applied in the mesh generation software as follows; Shapes are analyzed one at a time by using the parity count method. Shapes may take ownership of grid points that are contained, if the grid points satisfy one of the following conditions; (1) The grid points have not yet been assigned to a shape yet. (2) The current shape has a larger material identifier than the shape(s) that

[6]xfig for the Unix operating system is one such mechanical drawing package

had previously claimed the grid points.

The precedence method provides a predictable means of handling the situation that arises when shapes are intentionally or unintentionally made to share the same space. The *precedence method* is somewhat flexible in that resultant meshes will not be dependent upon the order in which shapes are analyzed. Implementation of this method is discussed in section 2.4.7.

2.2.4 A Word on Notation

In this document any variable that is capped with an arrow, such as $\vec{v}$, is meant to be interpreted as a vector. Variable names that are capitalized but not capped with an arrow are almost always points in two or three dimensional space. In *all* cases, variables in P imply points in three dimensional space. Similarly, *all* variables in B imply points in two dimensional space. The notation of Anton and Rorres[4] is used to express vector norms, for example $\|\vec{v}\|$ = the norm of $\vec{v}$. Both vectors and points refer to either ordered triples or ordered pairs of real numbers. Variable names in lower case are most often just ordinary scalar quantities. For example t could refer to a non-integer or an integer, which should be understood in context.

2.2.5 The Orthogonal Mesh

In this section, we example the variables used to define an orthogonal mesh. Figure 2.10 was provided to help in visualizing such a mesh. To form a mesh, along each edge of the mesh bounding box place equally spaced marks to form brick shaped subdivisions. There are N_x, N_y, and N_z brick shaped subdivisions along each edge corresponding to the x, y and z coordinate axes. The brick shaped subdivisions are referred to as *mesh boxes* or simply as *boxes.* It will be noted that marks placed along each respective edge of the bounding box are spaced apart a distance Δx, Δy and Δz, corresponding to;

$$\begin{aligned}
\Delta x &= \frac{P_{max_x} - P_{min_x}}{N_x} \\
\Delta y &= \frac{P_{max_y} - P_{min_y}}{N_y} \\
\Delta z &= \frac{P_{max_z} - P_{min_z}}{N_z}
\end{aligned}$$

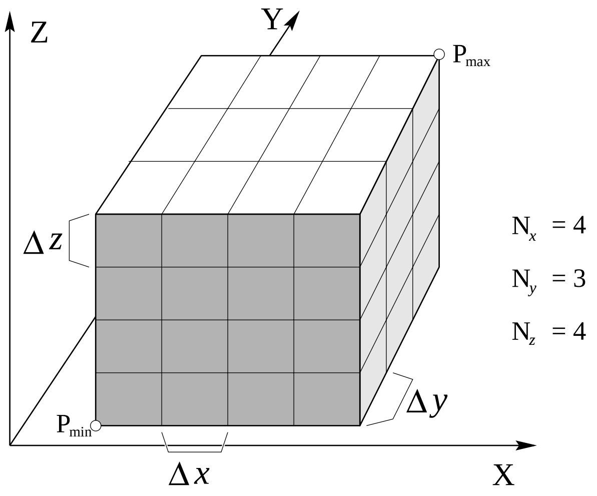


Figure 2.10: Example of Orthogonal Mesh

Starting at P_{min}, the marks made along each edge of the bounding box are each assigned an index value such that the index values i, j, k expressed as an ordered triple $[i, j, k]$ can reference any box corner point in the mesh. Each index is valid in each of the following sequences.

$$\begin{aligned} i &= 0, 1, 2, \ldots, N_x \\ j &= 0, 1, 2, \ldots, N_y \\ k &= 0, 1, 2, \ldots, N_z \end{aligned}$$

The square braces associated with index terms are meant to imply that the ordered triple is composed of integers rather than real physical dimensions. To determine the physical location (x, y, z) corresponding to $[i, j, k]$, use the following formula;

$$\begin{aligned} x &= i\,\Delta x + P_{min_x} \\ y &= j\,\Delta y + P_{min_y} \\ z &= k\,\Delta z + P_{min_z} \end{aligned} \tag{2.1}$$

A convention is followed that each box in the overall mesh will be referenced by the corner point that is closest to P_{min}. Such a reference point is referred to as a *referenced mesh box corner*. It is important to point out that while a bounding box edge parallel to, say the x axis has N_x subdivisions, these subdivisions were formed between $N_x + 1$ marks placed on that edge. Thus when referencing boxes in the overall mesh, i, j, and k are valid on the following sequences;

$$i = 0, 1, 2, \ldots, N_x - 1$$

$$\begin{aligned} j &= 0, 1, 2, \ldots, N_y - 1 \\ k &= 0, 1, 2, \ldots, N_z - 1 \end{aligned}$$

Thus the box that has P_{min} as a corner point is referenced as $[0, 0, 0]$, while the box that has P_{max} as a corner point is referenced as $[N_x - 1, N_y - 1, N_z - 1]$. Lastly, it is important that we be able to determine the actual physical coordinate values of grid points in the mesh. Since a grid point is located at the center of the corresponding mesh box, equation (2.1) allows us to immediately state that;

$$\begin{aligned} x_{gp} &= (i + 0.5)\Delta x + P_{min_x} \\ y_{gp} &= (j + 0.5)\Delta y + P_{min_y} \\ z_{gp} &= (k + 0.5)\Delta z + P_{min_z} \end{aligned}$$

These equations are rewritten in the following form;

$$\begin{aligned} x_{gp} &= i\,\Delta x + (P_{min_x} + 0.5\,\Delta x) \\ y_{gp} &= j\,\Delta y + \left(P_{min_y} + 0.5\,\Delta y\right) \\ z_{gp} &= k\,\Delta z + (P_{min_z} + 0.5\,\Delta z) \end{aligned}$$

To simplify these last equations, the coordinate values of the grid point in box $[0, 0, 0]$ are assigned to a unique variable;

$$\begin{aligned} \text{start}_x &= P_{min_x} + 0.5\,\Delta x \\ \text{start}_y &= P_{min_y} + 0.5\,\Delta y \\ \text{start}_z &= P_{min_z} + 0.5\,\Delta z \end{aligned} \tag{2.2}$$

Thus at last the coordinates of grid points referenced as $[i, j, k]$ can be calculated by the following equations;

$$\begin{aligned} x_{gp} &= i\,\Delta x + \text{start}_x \\ y_{gp} &= j\,\Delta y + \text{start}_y \\ z_{gp} &= k\,\Delta z + \text{start}_z \end{aligned} \tag{2.3}$$

2.2.6 The Search Origin

The search origin was introduced in section 2.1 as being a special point located outside the mesh bounding box. The following explanation is more exact. The location of the search origin is defined relative to two points, P_{max} and P_{min} that are specified in a given solid file along with the shape descriptions. These two points contain the extreme coordinate values that are used to define the mesh bounding box. Thus the search origin is defined relative to the mesh bounding box. In matrix notation the search origin is defined as;

$$S_o = \text{ diag } \{udso_x, udso_y, udso_z\}(P_{max} - P_{min}) + P_{min}$$

Where diag $\{a, b, c\}$ refers to a diagonal square matrix filled with the terms a, b, and c on the diagonal. The equation is equivalent to;

$$\begin{aligned} S_{o_x} &= udso_x \, (P_{max_x} - P_{min_x}) + P_{min_x} \\ S_{o_y} &= udso_y \, (P_{max_y} - P_{min_y}) + P_{min_y} \\ S_{o_z} &= udso_z \, (P_{max_z} - P_{min_z}) + P_{min_z} \end{aligned}$$

The terms $udso_x$, $udso_y$ and $udso_z$ are user defined constants. For the search origin to be guaranteed to be outside the mesh bounding box, the following restrictions must be enforced;

$$\begin{array}{lclclcl} udso_x & < & 0 & \text{or} & udso_x & > & 1 \\ udso_y & < & 0 & \text{or} & udso_y & > & 1 \\ udso_z & < & 0 & \text{or} & udso_z & > & 1 \end{array}$$

This particular approach to defining the search origin was suggested by a good friend and colleague, Surender Mohan. Clearly by using this equation the search origin can be placed anywhere relative to the mesh bounding box. During program development this equation was found to be particularly useful. Further details of the significance of this definition and the suitable selection of the constants $udso_x$, $udso_y$ and $udso_z$ are discussed in sections 2.4.4 and 2.4.6.

2.2.7 Describing Surfaces

To avoid confusion, a few special descriptive terms were developed for use in this document. As described earlier, a particular facet is defined by a finite set of coplanar three dimensional points, thus *facet* refers to the subregion of an infinite plane. The phrase *facet plane* refers to the infinite plane that the facet is actually a subregion of. It is implicitly understood that a given set of coplanar points are used to define both a facet and facet plane. Lastly, note that *facet* and *facet plane* are most often used in three dimensional spatial contexts.

The phrase *bounded region* also refers to a facet, but is properly used in context only in a certain restricted sense, that of two dimensional space. Paliouras and Meadows[43] provide a clear introduction to the Jordan Curve Theorem. Since the edges of a facet must represent a simple closed path, according to the Jordan Curve Theorem the facet plane is divided by a curve formed by the sides, into three mutually disjoint sets as follows:

1. The curve itself, denoted *C*.
2. The interior of the curve, denoted *Int(C)*, which is an open and bounded set.
3. The exterior of the curve, denoted *Ext(C)*, which is an open and unbounded set.

The truth of the Jordan Curve Theorem should be intuitively obvious, its simple geometric meaning is illustrated in figure 2.11. Following the clear terminology presented by Paliouras and Meadows it makes sense to refer to facets in the context of a two dimensional space as being a *bounded region*, or alternately as being a *bounded set*.

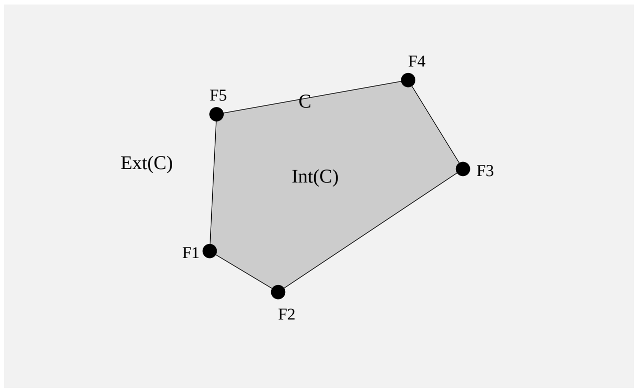


Figure 2.11: Facet Plane Regions

The two dimensional space being referred to above is called *facet space.* In the sense of linear algebra, a facet space is a vector space. Strang[61] as well as Anton and Rorres[4] provide clear explanations of vector spaces. A given *facet space* is simply a two dimensional subspace that completely contains a facet plane. Note that every facet has a uniquely associated facet space. Section 2.4.2 provides details of how a given facet space is defined. Lastly, there should be no confusion as to which facet space corresponds to each facet. In the algorithm, facets are processed one at a time thus there should a clear understanding as to *the facet space* which corresponds to the *current facet.*

2.3 Mesh Algorithm Overview

The parity count method described in section 2.1 effectively reduces the job of generating a three dimensional orthogonal mesh to the simpler problem of identifying and counting intersections. The job of actually counting intersections is trivial, thus the remaining problem is the identification of intersections with a shape surface.

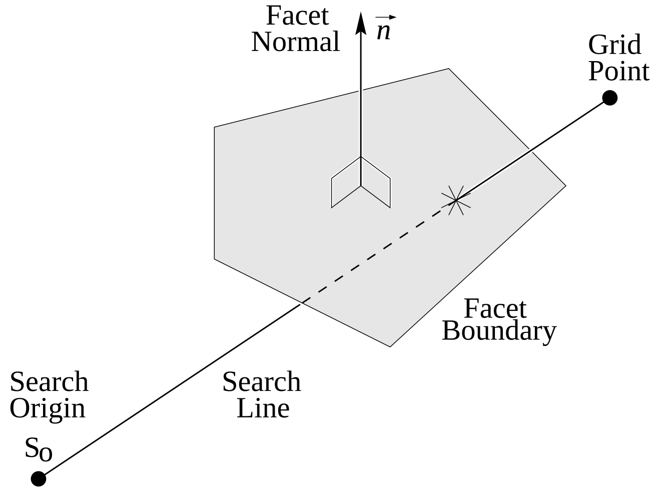


Figure 2.12: Search Line Intrersecting a Facet

In section 2.2, we showed how the surface of a shape is described by using a set of interconnected facets. Thus it should make sense that we must count the number of intersections a search line makes with all the facets that are used to describe the shape. Given a search line and an arbitrary facet, like the example shown in figure 2.12, there are only a small number of possible scenarios that are possible. We consider each of them now;

1. – The search line may not reach the facet. Remember that a search line extends only from the search origin to its associated grid point, thus has finite length.

2. – The search line may be orthogonal to the facet normal vector, $\vec{n}$. What this means is that either the search line cannot intersect the facet, or that the search line intersects the facet an infinite number of times. (That is, the search line in "inside" the facet space.)

   A simple test can be used to determine if the search line is orthogonal to the facet normal vector. If the test passes, we simply ignore the facet. If the search line is inside the facet space, the search line will also intersect the shape element of a connected facet, which we can detect.

3. – The search line may intersect the facet plane once. Given that a facet is a closed region of the facet plane, in this scenario it is possible for the search line to intersect the facet, at most once. Special care must be made to check if the search line intersects a shape element. If this happens the associated grid point must be declared as *unresolved.*

With this list in mind, it is now safe to state a simple yet powerful assertion. The job presented in section 2.1 of counting the number of times a search line intersects with a shape surface amounts to totaling the number of facets that experience a single intersection with the given search line. This assertion is intrinsic to the entire algorithm.

Given that a search line intersects a facet plane once, it is not immediately obvious whether the intersection appeared inside the facet. Clearly, determining if a search line intersects a facet is the "crux of the problem." Thus the task of counting intersections is dependent on the solution of a two dimensional problem. The two dimensional problem is simply stated; Determine if an 'arbitrary point' (the intersection point between the facet plane and a search line) is located in the interior of a 'two dimensional bounded region' (the facet). One possible solution to the two dimensional problem is presented in section 2.4.5, along with a discussion of other possible solutions. Once all of the facets have been examined and all the intersections with search lines counted, if the tally is odd then the grid point must be inside the shape.

An introductory diagram of the algorithm used to count intersections for a single shape is presented in figure 2.13. Note that the algorithm simply loops repeatedly until all the facets describing the current shape have been examined. Note in figure 2.13, if an intersection is seen between a search line and a facet, it is appropriate to simply increment a counter by one.

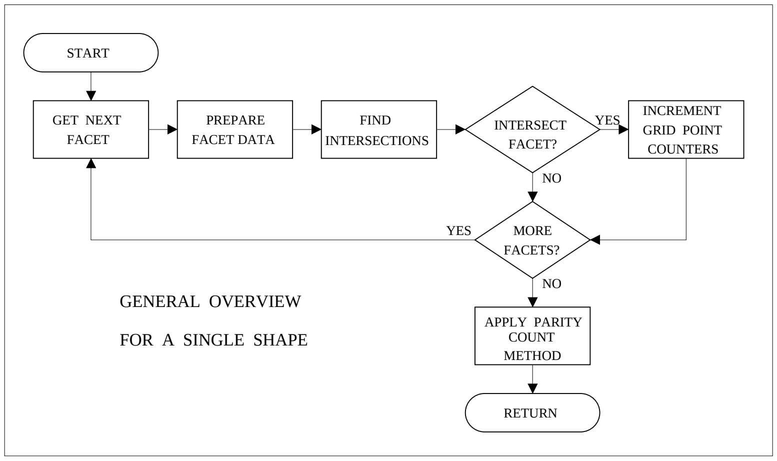


Figure 2.13: Counting Intersections for Parity Count Method

2.4 The Mesh Algorithm

In this section we present the entire mesh generation algorithm. The algorithm was broken into smaller tasks that individually are more easily defined. The flowchart in figure 2.14 illustrates the proper sequence in which tasks are performed to implement the algorithm. Each square block in the flowchart is equivalent to a task listed in table 2.1. Note that depending on how the flowchart is interpreted, either a serial or a parallel implementation of the algorithm will result.

Table 2.1: **List of Algorithm Tasks**

Item		Section		Name of Task
1	–	2.4.1	–	Initialize Analysis
2	–	2.4.1	–	Initialize Shape
3	–	2.4.2	–	Facet Server
4	–	2.4.3	–	Pick Next Grid Point
5	–	2.4.4	–	Intersection Finder
6	–	2.4.5	–	Facet Solver
7	–	2.4.6	–	Check Insideness
8	–	2.4.6	–	Resolve Singularities
9	–	2.4.7	–	Copy Out
10	–	2.4.8	–	Store Results in Files

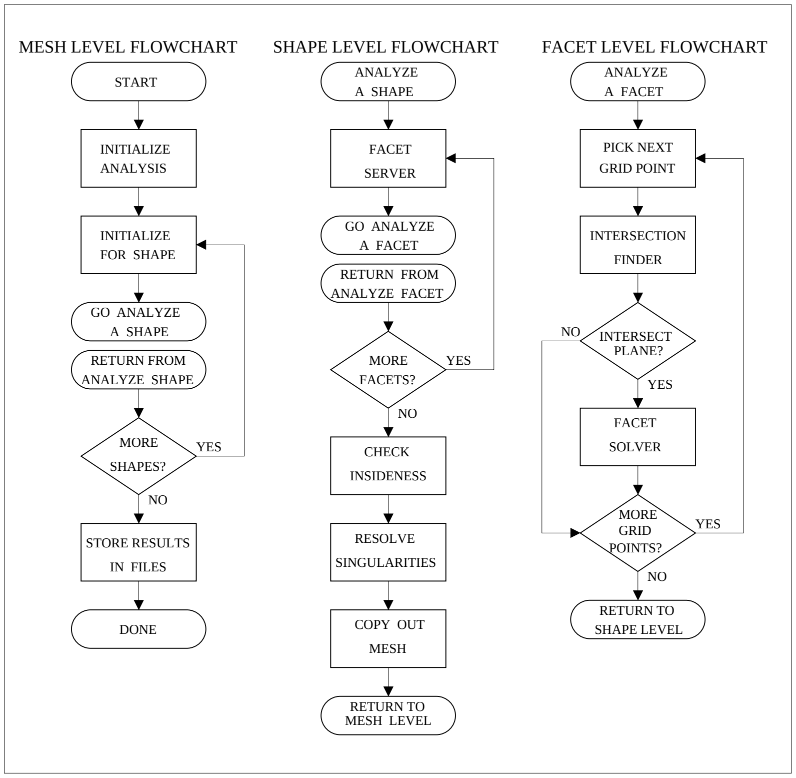


Figure 2.14: Flowchart of Mesh Generation Algorithm Tasks

The overall flow chart was divided into three smaller flowcharts that are linked as layers. The Mesh level flowchart is where the algorithm starts and ends and is performed once. The shape level flowchart is performed for each and every shape that is analyzed. The facet level flowchart is performed for each and every facet that is processed. Next is an in depth discussion of each task.

2.4.1 Initialization Tasks

Initialization was split into two tasks. Before the algorithm can be started, a certain amount of initialization is required. In addition, a certain amount of initialization is required before each shape can be analyzed, the list of vertex points associated with the shape to be analyzed are loaded into memory and is searched to find the largest and smallest coordinate values. The extreme coordinate values are used to define a workspace for the facet.

It is important to realize that it is not necessary that every grid point be checked for containment by every shape. Since the largest and smallest coordinate values associated with each shape are known, it is a simple matter to define a subregion of the overall mesh. Such a subregion is easily defined by using a three dimensional bounding box inside the overall mesh. Clearly in examining a given shape it is only necessary to check for containment, the grid points in a subregion where the shape is known to be, thus it is appropriate to name such a subregion the *analysis subregion*. Thus part of the job of preparing a workspace is defining the associated analysis subregion. With the workspace set aside, the list of vertex points associated with the current shape loaded, and the anylysis subregion defined, the next task can be performed.

2.4.2 Facet Server Task

To make a facet more usable, calculations are performed to result in three items of useful information. First, the parameters used to define the facet plane are determined. These planar parameters are needed by the Intersection Finder Task to determine if intersections exist between the facet plane and given search lines. Second, the facet server produces two orthogonal unit vectors that serve as basis vectors for facet space. These unit vectors will be needed also by the intersection finder to map any intersections found, to facet space. Last, the facet server produces the list of boundary points that define the facet in facet space. This list of boundary points is needed by the facet solver to determine point containment. The following sections provide a detailed description of the facet server and its duties.

Selecting From Facet Points

A set of three dimensional points are used to define each facet.

$$\{F_i : i = 1, 2, \ldots, n\}$$

For this discussion, it should be acceptable to the reader that somehow a set of three dimensional points used define each facet are automatically provided to the facet server. An outline of how this is actually done is presented in section 3.4. The number n, is simply the total number of points used to define a facet.

From the set of points used to define a facet, the first two points are selected and named P_0 and P_1. A third point, named P_2 is arbitrarily selected from the remaining points used to define the facet. It is important to note that P_2 is selected such that, if possible, P_0, P_1 and P_2 are not collinear, or form a single straight line. Using the three selected points, two vectors $\vec{u}$, and $\vec{\ell}$ are defined. Note that both vectors originate from point P_0.

$$\vec{\ell} = (P_2 - P_0)$$

$$\vec{u} = (P_1 - P_0)$$

To determine that P_2 is an acceptable choice we must show that $\vec{\ell}$ and $\vec{u}$ are not parallel. It is simple enough to use the following formula to determine the cosine of the angle between the two vectors.

$$\cos(\alpha_{lu}) = \frac{\vec{\ell} \cdot \vec{u}}{\left\|\vec{\ell}\right\| \, \|\vec{u}\|} \tag{2.4}$$

Once the cosine of the angle between the two vectors has been determined, its absolute value is compared against the cosine of an arbitrary constant angle near 0^o. Thus we define $\alpha_{coll} = 0^o + \epsilon$ where ϵ is a small number. If it is true that;

$$\alpha_{coll} < \alpha_{lu} < (180^o - \alpha_{coll})$$

then it must also be true that;

$$|\cos(\alpha_{lu})| < \cos(\alpha_{coll})$$

To perform such a test, an arbitrary small angle can easily be selected for α_{coll}. The actual test performed is stated as follows;

$$\frac{|\vec{\ell} \cdot \vec{u}|}{\left\|\vec{\ell}\right\| \, \|\vec{u}\|} < \cos(\alpha_{coll}) \tag{2.5}$$

Suppose we pick α_{coll} to be 10^o, then $\cos(\alpha_{coll}) = 0.984808\ldots$ All angles that are greater than 10^o, and less than 170^o will have $|\cos(\alpha_{lu})| < \cos(10^o)$, for example $\cos(60^o) = 0.5$. Thus, all such angles that satisfy equation (2.5) are associated with vectors that are clearly not parallel.

Basis for Facet Space

Formally, the next job that the facet server performs is to define a unique set of two dimensional coordinates for the facet space. The new two dimensional coordinate system is defined such that P_0 corresponds to the new origin and that two unit length vectors, $\vec{u}_n$ and $\vec{a}_n$ act as basis vectors for the new coordinate axes. Half the job of defining the two basis vectors is nearly already done. The basis vector $\vec{u}_n$ is simply the normalized $\vec{u}$ vector.

$$\vec{u}_n = \frac{\vec{u}}{\| \vec{u} \|}$$

Only a small bit of work will be needed find the other basis vector, $\vec{a}_n$.

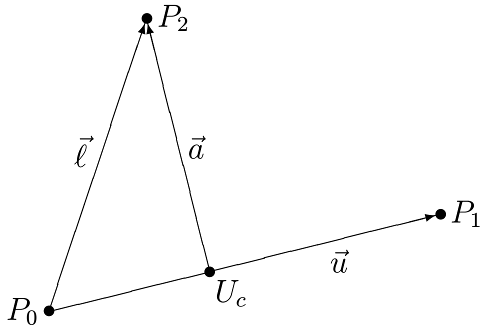


Figure 2.15: Facet Related Vectors

Line U is defined as a parametric equation, with respect to $\vec{u}_n$ and P_0, as shown. Also, refer to figure 2.15.

$$U = \vec{u}_n \, t + P_0 \tag{2.6}$$

Vector $\vec{a}$ is defined to be orthogonal to $\vec{u}$. The length of $\vec{a}$ represents the distance from point U_c on line U to point P_2. The basis vector $\vec{a}_n$ is simply the normalized $\vec{a}$ vector. Point U_c is found as follows; First determine the parametric variable t_c that corresponds to U_c, via the dot product.

$$t_c = \vec{\ell} \cdot \vec{u}_n$$

Next use equation (2.6) to determine U_c. With U_c in hand, determine $\vec{a}_n$.

$$U_c = \vec{u}_n \, t_c + P_0$$

$$\vec{a} = (P_2 - U_c)$$

$$\vec{a}_n = \frac{(P_2 - U_c)}{\| P_2 - U_c \|}$$

Thus $\vec{u}_n$ and $\vec{a}_n$ are orthogonal basis vectors that define a two dimensional subspace. It is interesting to note that basis vectors in general are not necessarily orthogonal or unit length. It was simply an arbitrary decision to select basis vectors that are orthogonal and of unit length. Some simplification of the algorithm is possible, for example $\vec{u}$ and $\vec{\ell}$ could be used directly as basis vectors. The fundamental requirement that basis vectors be linearly independent is guaranteed since P_0, P_1 and P_2 were selected such that they are are not collinear. In regards to the actual implementation, the task of defining a two dimensional system of coordinates really does not represent any additional work other than defining two linearly independent vectors. The point P_0 was arbitrarily selected to be mapped as the new two dimensional origin.

Defining Planar Parameters

To define the infinite plane that a given facet is a subregion of, it is useful to start with the point-normal form then determine the general form of a plane. Anton and Rorres[4] provide a full introduction to the point normal form. Two data items are needed to produce a point-normal form of an infinite plane. The first item needed is just any vector $\vec{n}$ that is normal to the facet surface. Such a vector is easily found by performing a cross product;

$$\vec{n} = \vec{u}_n \times \vec{a}_n \tag{2.7}$$

The second item needed is just any point on the facet, such as P_0.

The basic idea behind the point-normal form is that a plane can be defined as the locus of all points such that a line drawn from P_0 to any point in the plane, must be orthogonal to $\vec{n}$. Thus;

$$\vec{n} \cdot \overrightarrow{P_0P} = 0 \tag{2.8}$$

Given that an arbitrary point in the facet plane is given as $P = (x, y, z)$ and that point $P_0 = (x_0, y_0, z_0)$ it follows that;

$$\overrightarrow{P_0P} = (x - x_0, y - y_o, z - z_0)$$

Further, given that $\vec{n} = (a, b, c)$, equation (2.8) can be rewritten as;

$$a\,(x - x_0) + b\,(y - y_0) + c\,(z - z_0) = 0$$

This last equation is the point-normal form of a plane. The general form of a plane is;

$$a\,x + b\,y + c\,z + d = 0 \tag{2.9}$$

In equation (2.9), d is easily defined as;

$$d = -(a\,x_0 + b\,y_0 + c\,z_0) \tag{2.10}$$

Thus with $\vec{n}$ and d in hand, a plane is completely defined in a most useful manner.

Mapping to 2D Space

Each point in two dimensional space has two coordinate values that are uniquely arranged in an ordered pair, such as;

$$B_i = (u_i, a_i)$$

To map each vertex point from three dimensional space to the current two dimensional facet space, use the following equation;

$$\begin{aligned} u_i &= (F_i - P_0) \cdot \vec{u}_n \\ a_i &= (F_i - P_0) \cdot \vec{a}_n \end{aligned} \tag{2.11}$$

The reader will notice that until this point all vertex points have been labeled with the following conventional form: P_{ID}. The associated capital P strictly implies that the given point named *ID* belongs to a set of three dimensional points used to define a facet. Similarly, the following conventional form B_{ID} will be used to identify boundary points. For this document, the term *boundary points* refers to two dimensional points in facet space that are mapped from three dimensional vertex points.

Facet Server Task Summary

Figure 2.16 contains a flowchart that helps to provide a summary of the Facet Server Task. Note that in nearly all cases the first choice of P_2 is acceptable. When the 'A' circle is reached, points P_0, P_1 and P_2 will be selected from the list of points that define the facet. After defining the facet plane and facet space, the remaining loop iterates through the list of points that define the facet and maps them to facet space.

To summarize, three significant items are produced by the facet server. First, two orthogonal unit vectors, $\vec{u}_n$ and $\vec{a}_n$ are produced that serve as basis vectors for the facet space. Second, planar parameters $\vec{n}$ and d that define in three dimensional space the infinite facet plane. Third and last, a list of the boundary points that were mapped from the three dimensional vertex points.

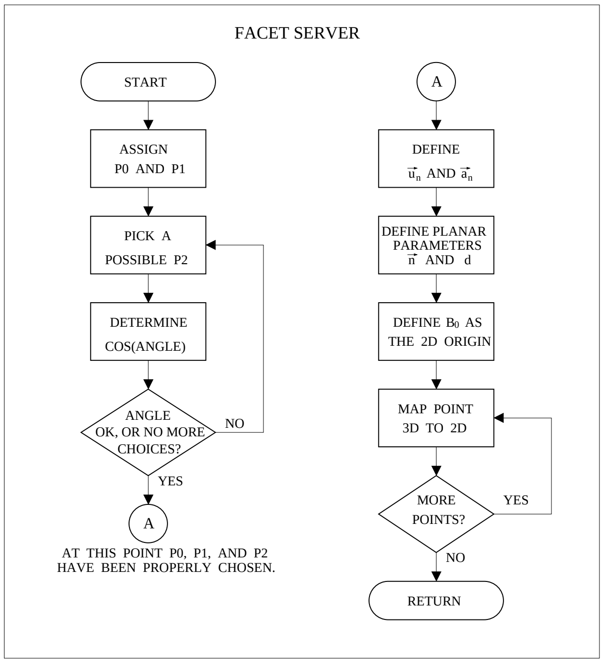

FACET SERVER
START
ASSIGN P0 AND P1
PICK A POSSIBLE P2
DETERMINE COS(ANGLE)
ANGLE OK, OR NO MORE CHOICES?
NO
YES
A
AT THIS POINT P0, P1, AND P2 HAVE BEEN PROPERLY CHOSEN.
A
DEFINE $\vec{u}_n$ AND $\vec{a}_n$
DEFINE PLANAR PARAMETERS $\vec{n}$ AND d
DEFINE B_0 AS THE 2D ORIGIN
MAP POINT 3D TO 2D
MORE POINTS?
YES
NO
RETURN

Figure 2.16: Flowchart of Facet Server

2.4.3 Pick Next Grid Point Task

It was pointed out by the ‘Initialize For Shape’ task in section 2.4.1 that it is not necessary that every grid point be examined for containment by every shape. This task picks from the overall mesh only those grid points that will be checked. Note again that depending how the flowchart in figure 2.14 is interpreted, either a parallel or serial implementation will result. For example if a single grid point were to be selected each time this task is called, a serial implementation would result.

Consider the following as an alternative scenario; Each time this task was called, if a set of grid points were selected all at once, and simultaneously each associated search line was checked for an intersection with the current facet, then the implementation could be thought of as being “parallel.” In a sense, many cases of the Facet Level Flowchart would be performed simultaneously, in parallel. Continuing the same thought process, yet another scenario could have multiple instances of the shape level flowchart performing in parallel. Such scenarios easily lead to different levels and types of parallelism. This topic will be considered further with the discussion of time complexity in section 2.7.

2.4.4 Intersection Finder Task

The purpose of the intersection finder is to determine if and where there exists an intersection between a facet plane and a search line. There are two cases that result in no intersection between a search line and a facet plane, first as described in section 2.1, a search line is simply a finite length line segment and may not reach the facet plane. It is also possible for the search line to be orthogonal to the facet plane normal vector. In this second case there is either no intersection, or an infinite number of such intersections. In any circumstance, the intersection finder must only report an intersection between a given search line and given facet plane if and only if there is exactly one intersection present.

The first job performed by the Intersection Finder Task is to define the search line as a parametric equation.

$$L = \vec{l}\, t_s + S_o\ ;\ \ 0 \leq t_s \leq 1 \tag{2.12}$$

The variables are defined as;

L	$=$ A Point on The Search Line
t_s	$=$ The parametric term
G_i	$=$ Grid Point i
S_o	$=$ The Search Origin
$\vec{l}$	$= (G_i - S_o)$

Note that in defining the search line as a line segment, the parametric term t_s is restricted to a set of values that range from zero to one, inclusive.

Near Orthogonal Test

The next job is to test if the search line is orthogonal to the plane normal vector. Because of the way that floating point numbers are stored in digital computers, simply taking the dot product and testing the result for equality with zero, will not be useful in checking for vectors that are exceedingly close to being orthogonal.

Two methods of checking for near orthogonal vectors were considered. The first method is to use an equation similar to equation (2.5) to determine the cosine of the angle between the vectors, the equation is shown next.

$$ttest = \cos(\alpha_{ln}) = \text{abs}\left(\frac{\vec{l}\cdot\vec{n}}{\left\|\vec{l}\right\|\|\vec{n}\|}\right)$$

Since $\vec{n}$ is equal to the cross product of two orthogonal unit length vectors, $\vec{n}$ must also be a unit length vector, thus the equation can be simplified to;

$$ttest = \cos(\alpha_{ln}) = \text{abs}\left(\frac{\vec{l}\cdot\vec{n}}{\left\|\vec{l}\right\|}\right) \tag{2.13}$$

External to the program, the cosine of some arbitrary angle near 90^o can easily be defined as a constant. Thus, if *ttest* is smaller than the given arbitrary constant, the two vectors are said to be, *close to being orthogonal.* Conversely, if *ttest* is larger than the arbitrary constant then the vectors are not orthogonal.

While equation (2.13) is reasonably efficient, performing such a calculation still requires a significant amount of computer resources. Determining the length of a search line calls for the use of the square root function which is computationally expensive, especially since equation (2.13) must be performed once for every grid point in the mesh, for every facet processed. Rather than looking for a complicated "fix," such as calculating the square of the cosine of the angle between vectors instead, if it is realized that such an exacting test is not really required, then a very simple test can be designed. An alternative method is presented next.

Consider one definition of the dot product[4], where θ is the angle between two arbitrary vectors vectors $\vec{u}$ and $\vec{v}$. Note that the following equation was selected as a means to present the following topic from a clear conceptual standpoint. Since it was decided that all vectors would be stored in a component wise fashion, a more efficient method was actually selected to calculate dot products.

$$\vec{u}\cdot\vec{v} = \begin{cases} \|\vec{u}\|\,\|\vec{v}\|\,\cos(\theta) & \text{if } \vec{u}\neq 0 \text{ and } \vec{v}\neq 0 \\ 0 & \text{if } \vec{u}=0 \text{ or } \vec{v}=0 \end{cases}$$

Since the search origin must be outside the mesh bounding box and grid points must be inside the mesh bounding box, clearly $\vec{l}$ which is associated with the search line should never be of zero length. Also, $\vec{n}$ is normal to the facet plane and known to be a unit length vector. Making suitable substitutions, and given that α is the angle between $\vec{l}$ and $\vec{n}$, the dot product for these two vectors can be expressed as follows;

$$\vec{l}\cdot\vec{n} = \left\|\vec{l}\right\|\cos(\alpha)$$

Note that since P_{max} and P_{min} define the extreme coordinate values of the mesh bounding box, $\|P_{max} - P_{min}\|$ is equal to the length of the diagonal of the mesh bounding box. See figure 2.2 on page 48 for an illustration of the mesh bounding box, including one such diagonal. Using the last math expression, the following constant can be defined for testing for nearly orthogonal vectors;

$$\begin{gathered} K_{test} = \|P_{max} - P_{min}\| \cos(\alpha_k) \\ \alpha_k = 90^o - \epsilon, \text{ where } 0^o < \epsilon \ll 90^o \end{gathered} \tag{2.14}$$

The constant K_{test} represents a quantity equal to the length of the diagonal of the mesh bounding box, multiplied by the cosine of an arbitrary constant angle that is just under 90^o.

To test if $\vec{l}$ and $\vec{n}$ are nearly orthogonal, simply check if the absolute value of the dot product between the two vectors is less than K_{test}. If the following mathematical statement is true, it should be possible to guarantee that the two associated vectors are not orthogonal;

$$|\vec{l} \cdot \vec{n}| > K_{test} \tag{2.15}$$

This test is particularly appealing since the dot product between $\vec{l}$ and $\vec{n}$ is actually an intermediate value that may be required later in the algorithm, thus performing the approximate test for nearly orthogonal vectors is almost at no cost. While the test is certainly simple, it is not so clear as to what angles will pass or fail this test or how we can guarantee that the test will always work, so let's clear up the mystery.

By combining equation (2.13) and equation (2.14), the angle at which test 2.15 just indicates that vectors $\vec{l}$ and $\vec{n}$ are nearly orthogonal is equal to;

$$\alpha_{orth} = \arccos\left(\frac{\|P_{max} - P_{min}\|}{\left\|\vec{l}\right\|} \cos(\alpha_k) \right) \tag{2.16}$$

By examining this last equation, it is clear that $\left\|\vec{l}\right\|$ is the only independent variable. By placing the search origin at a distance from the mesh bounding box, the range of values that $\left\|\vec{l}\right\|$ and α_{orth} may have are easily controlled. The length of $\vec{l}$ must always fall into the following range;

$$\left\|\vec{l}\right\|_{\text{smallest}} < \left\|\vec{l}\right\| < \left\|\vec{l}\right\|_{\text{largest}}$$

The largest range of values that $\left\|\vec{l}\right\|$ can have results for a highly dense mesh when the search origin is placed at a location that is in line with a diagonal of the mesh bounding box. For our purposes it is reasonable to place the search origin at a distance from the mesh bounding box, approximately equal to the length of the diagonal. Thus, given that the search origin is in fact placed in line with a diagonal such that the distance from the mesh bounding box is equal to the length of a diagonal, then the following extreme values are defined as;

$$\left\|\vec{l}\right\|_{\text{smallest}} = \|P_{max} - P_{min}\|$$

$$\left\|\vec{l}\right\|_{\text{largest}} = 2\,\|P_{max} - P_{min}\|$$

This last statement along with equation (2.16) will be used to determine the range of angles that are guaranteed to pass or fail the near orthogonality test defined by equation (2.15). Figure 2.17 is presented next to help the reader to visualize the range of angles that is being discussed.

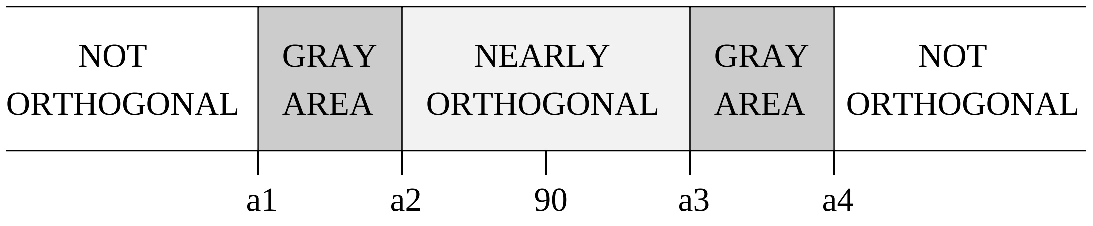


Figure 2.17: Diagram of Associated Angles

The angles associated with the near orthogonality test are computed as follows. Note that ϵ is a small positive number. As ϵ approaches zero the following approximations become equalities.

$$\text{a1} = \arccos(\cos(90^o - \epsilon)) = 90^o - \epsilon$$

$$\text{a2} = \arccos\left(\frac{\cos(90^0 - \epsilon)}{2}\right) \approx 90^o - \frac{\epsilon}{2}$$

$$\text{a3} = \arccos\left(\frac{\cos(90^0 + \epsilon)}{2}\right) \approx 90^o + \frac{\epsilon}{2}$$

$$\text{a4} = \arccos(\cos(90^o + \epsilon)) = 90^o + \epsilon$$

Consider the following example, suppose that $\epsilon = 0.1^o$, then a1 $= 89.9^o$, a2 $= 89.95^o$, a3 $= 90.05^o$, and a4 $= 90.1^o$. For this example, this situation is equivalent to measuring angles with an absolute accuracy of $\pm 0.025^o$ and then stating that the angles are nearly orthogonal if the measured value is greater than 89.925^o and less than 90.075^o. In any event, we can guarantee in this example that angles that are between 89.95^o and 90.05^o will be detected as nearly orthogonal. Likewise, we can guarantee that angles greater than 90.1^o or less than 89.9^o will not be detected as being nearly orthogonal.

Finally, since ϵ is defined before the program is executed and the placement of the search origin is arbitrarily selected relative to the mesh bounding box, the tolerance of the test can easily be made tighter than was demonstrated in the above example.

The Intersection Point

Given that the normal of the current facet plane is not orthogonal to a given search line, at most one intersection point may exist between the search line and the facet plane. The next job is to try to find that intersection point. First consider an extended search line to be of infinite length. The extended search line is defined as;

$$E = \vec{l}\, t_e + S_o \tag{2.17}$$

Where $\vec{l}$ is the familiar search line orientation vector, t_e is the parametric term, S_o is the search origin and E is a point on the extended search line. Given that vectors $\vec{l}$ and $\vec{n}$ are not nearly orthogonal, one and only one intersection point must exist between the extended search line and the facet plane.

Equation (2.9) is the general form of the facet plane. The definition of the extended search line is given as equation (2.17). Both equations are easily solved to determine the parametric term for the extended search line that corresponds to the intersection point.

$$t_e = \frac{-(d + (a\, S_{o_x} + b\, S_{o_y} + c\, S_{o_z}))}{\vec{l} \cdot \vec{n}} \tag{2.18}$$

As before, $\vec{n} = (a, b, c)$ Note that since $\vec{l}$ and $\vec{n}$ are non-zero and non-orthogonal, $\vec{l} \cdot \vec{n}$ must be nonzero.

The next step is to decide if the intersection point found on the extended search line indicates that the search line cut through the facet plane. While equation (2.12) says that the parametric term for the search line is valid over,

$$0 \leq t_s \leq 1$$

this range does not suggest an efficient or useful test. For a search line to possibly experience an interior to exterior transition, the tip of the search line must cut through the facet plane. In addition since the search origin is known to be outside the mesh bounding box and grid points are known to be inside the mesh bounding box, there is no reason to test if $0 \leq t_e$, since this condition is guaranteed. Thus if the following is true,

$$t_e < 1 \tag{2.19}$$

the search line is guaranteed to cut through the facet plane. Next, the value of the parametric term t_e is substituted back to find the coordinates of the intersection point. Lastly the intersection point is mapped to the facet space.

Intersection Finder Task Summary

Figure 2.18 contains a flowchart that is helpful in summarizing the Intersection Finder Task. Note that the entire Intersection Finder Task summarized in figure 2.18 matches the 'Intersection Finder' and 'Intersection Found?' blocks in figure 2.14.

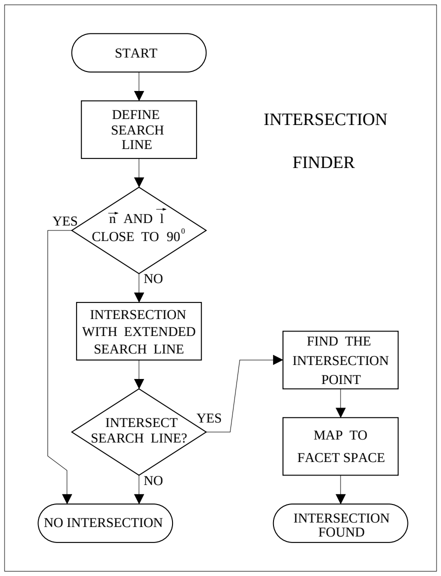


Figure 2.18: Flowchart of Intersection Finder

The intersection task first defines the search line and equivalently the extended search line by defining terms for equation (2.12) and (2.17). Before attempting to find an intersection point it is necessary to check that a single intersection point is guaranteed to exist between the extended search line and a given facet, this test is performed by evaluating equation (2.15). The next action performed is to actually find the parametric term corresponding to the intersection point between the extended search line and the facet plane by using equation (2.18).

The intersection finder must next decide if the search line cuts through the facet plane, the decision is made by evaluating equation (2.19). If equation (2.19) is true, the intersection point is found by evaluating equation (2.17), and lastly the intersection point is mapped to the facet space, by using equation (2.11). If equation (2.15) or equation (2.19) are not satisfied, then the intersection finder simply reports that no intersection was found between the search line and facet plane.

2.4.5 Facet Solver

At this point it is useful to think of a facet as being a bounded region in two dimensional facet space, as was presented in section 2.2.7. If an intersection point is indeed found between a search line and a facet plane, this task has the job of examining where the intersection point is located relative to the facet. As introduced in section 2.3, this task essentially attempts to answer a "true or false" question, that of whether an arbitrary point is clearly inside a given bounded region. Unfortunately as presented in section 2.1, it may not be possible to determine in a clear sense whether or not the arbitrary point is inside the bounded region, this task also detects such situations.

The job performed by the facet solver can be thought of as being similar to the job performed by the entire mesh generation program. While the Facet Solver Task decides in a two dimensional sense if a given point is contained by a given two dimensional bounded region, the core objective of the mesh generation program as a whole is to decide if arbitrary points are contained by a given three dimensional bounded region. Essentially the three dimensional problem of generating an orthogonal mesh was reduced to solving the two dimensional containment problem. The solution to the two dimensional containment problem presented in this document is named the *lines and intersections method*, and is presented next. While only one method of solving the two dimensional containment problem is presented here in detail, several references are made in the following facet solver summary. Hansen and Levin[20] present a simple technique that can be extended to solve the two dimensional containment problem, but is not considered to be a satisfactory solution as it can only solve for the special case of convex facets. This alternate technique is discussed only in passing in section 2.10.

Introduction to Lines and Intersections

The lines and intersections method of solving a facet actually implements the same parity count method introduced in section 2.1, but in a two dimensional sense. A two dimensional facet search origin is defined for each and every facet to be outside the bounded region. As with the three dimensional analysis, a search line segment is first drawn from the search origin to some arbitrary point. In this case the arbitrary point is the point mapped from the intersection between the three dimensional search line and the facet plane, to see why this is true see section 2.4.4. If the number of intersections between the sides of the bounded region and the two dimensional search line is odd, then the arbitrary point must be inside the bounded region. Conversely, if the number of intersections is even, then the arbitrary point must be outside the bounded region.

The two dimensional parity count method is easily implemented using a technique analogous to that used in the three dimensional analysis. Since the sides of a bounded region are actually straight line segments, if the facet search line is not parallel to any side then at most one intersection may exist between the facet search line and any given side. Given that such a case is true and further given that the intersection is not coincident with either of the side endpoints and lastly is not coincident with the end[7] of the facet search line then a *clear intersection* is said to exist. If all intersections between a search line and the sides of a facet are clear intersections then the total number of intersections is equal to the total number of side elements that have intersections with the facet search line. The subsection titled 'Handling the Intersection,' discusses in greater detail how intersections are accounted for. Special emphasis is placed on how intersections that are not clear intersections are handled.

Facet Search Origins

In defining a facet search origin, the first thing required is a list of the largest and smallest coordinate values of boundary points used to describe the facet. The job of finding these boundary values is usually an additional task that the facet server performs. The largest and smallest coordinate values are assigned to two variables, together the variables define a two dimensional bounding box.

[7]Clearly like all line segments the facet search line must have two endpoints, but we are only concerned with the end that can possibly be inside the bounded region. The reader can think of a search line as being a ray that extends to infinity.

B_{max}	—	maximum coordinate values
B_{min}	—	minimum coordinate values

A facet search origin is defined by the following equation. Note that $udso_u$ and $udso_a$ are constant values defined such that the facet search origin is always outside the bounded region.

$$\begin{aligned} B_{so_u} &= udso_u\,(B_{max_u} - B_{min_u}) + B_{min_u} \\ B_{so_a} &= udso_a\,(B_{max_a} - B_{min_a}) + B_{min_a} \end{aligned} \tag{2.20}$$

It is important to clarify the following point; The single point referred to as the *search origin* is a three dimensional point that is specifically defined relative to the three dimensional mesh bounding box. The *search origin* is to be regarded as being defined by constant value coordinates thus are not allowed change during program runtime. The unique symbol S_o is assigned to the search origin.

In contrast to the search origin, each facet has an associated *facet search origin.* It is to be understood that there is a one to one correspondence between each facet and its facet search origin. A single facet origin is defined for the current facet being studied. While many facet origins may exist, there should be no confusion regarding the unique symbol B_{so}, which is assigned to the current facet search origin.

Facet Search Line

A search line is defined as a parametric equation as follows;

$$B_s = \vec{v}_s\, t_s + B_{so}\ ;\ \ 0 \leq t_s \leq 1 \tag{2.21}$$

The variables are defined next. Note that in defining the facet search line as a line segment, the parametric term t_s is restricted to a set of values that range from zero to one, inclusive.

B_s	= A point on the facet search line
$\vec{v}_s$	= $(B_I - B_{so})$
B_I	= Point mapped from 3D intersection
B_{so}	= The facet search origin
t_s	= The parametric term

Facet Sides

A side of the bounded region is defined as a parametric equation as follows; The points B_i and B_{i+1} are the end points of a given facet side element and B_{mid} is the point exactly between the two endpoints. Note that in defining a facet side element in this way, the parametric term is restricted to a set of values that range from minus one to one, inclusive.

$$B_b = \vec{v}_b\, t_b + B_{mid}\ ; \quad -1 \le t_b \le 1 \tag{2.22}$$

The variables are defined as;

B_b	$=$ A point on the facet search line
$\vec{v}_b$	$= (B_{i+1} - B_{mid})$
B_{mid}	$= \frac{1}{2}(B_{i+1} + B_i)$
B_i, B_{i+1}	$=$ Two boundary points in order
t_b	$=$ The parametric term

Intersection Solution

In the following we consider the solution of equations (2.21) and (2.22). The coordinates of the actual intersection point between a side element and a facet search line need not be determined to check if an intersection exists. The solutions of the facet search line equation (2.21) and the facet side equation (2.22) are presented in terms of parametric terms. First for the search line;

$$t_s = \frac{v_{b_a}(B_{mid_u} - B_{so_u}) - v_{b_u}(B_{mid_a} - B_{so_a})}{v_{s_u} v_{b_a} - v_{s_a} v_{b_u}} \tag{2.23}$$

Next, for the facet side;

$$t_b = \frac{v_{s_a}(B_{mid_u} - B_{so_u}) - v_{s_u}(B_{mid_a} - B_{so_a})}{v_{s_u} v_{b_a} - v_{s_a} v_{b_u}} \tag{2.24}$$

In using equations (2.23) and (2.24), steps must be taken to avoid division by a number close to zero, as such a situation would lead to an overflow condition. To avoid such an overflow condition, it may at first make sense to compare denominators to some fraction of the numerators, unfortunately this is not a good idea in general as the situation becomes more complicated when a numerator is also close to zero. Such a situation corresponds to the indeterminate condition of zero divided by zero. Note that the use of the subtraction operator is significant here. Whenever we subtract two numbers that are close in value, we expect the difference to be small, unfortunately the smaller the difference is, the worse the precision of that value. Such cases of reduced precision must be handled carefully.

A certain insight can be obtained by making an investigation from a geometric point of view. By examining the denominators of equations (2.23) and 2.24, it is seen that the denominators are identical and have a particular significance.

$$\text{denominator} = v_{s_u} v_{b_a} - v_{s_a} v_{b_u} = 2A \qquad (2.25)$$

The variables are defined as;

$\lvert 2A \rvert$	=	Twice the triangular area
v_{s_u}, v_{s_a}	=	Components of $\vec{v}_s$
v_{b_u}, v_{b_a}	=	Components of $\vec{v}_b$

The absolute value of equation (2.25) is equal to twice the area of a triangle formed from $\vec{v}_s$ and $\vec{v}_b$ when their tails are joined at a common point and a line segment is drawn between the tips of the vectors, see figure 2.19. It is important to note that figure 2.19 does not show the ordinary relationship between the facet search line and the side of a facet. Something similar to figure 2.19 is easily formed when any two vectors are joined in just this way. Hansen and Levin[20] presented equation (2.25) in a more general determinant form. Equation (2.25) is actually derived in appendix D.

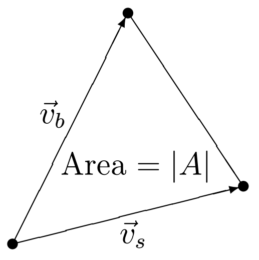


Figure 2.19: Vectors and Triangular Area

If the 'denominator' term which corresponds to the triangular area between $\vec{v}_b$ and $\vec{v}_s$ is measurably not zero, then these vectors cannot be parallel, and thus no more than one intersection may exist between the corresponding facet side and facet search line. Conversely, if the denominator term is zero then $\vec{v}_s$ and $\vec{v}_b$ must be parallel vectors. While informative, what we really need to know is if the facet search line and facet side are coincident, or rather are they parallel and represent an inconsistent system of equations. If the search line and given facet side are parallel, but not coincident then no intersection can possibly exist between them. If however the lines coincide, then an infinite number of intersections will exist.

$$\text{numerator} = v_{b_a}(B_{mid_u} - B_{so_u}) - v_{b_u}(B_{mid_a} - B_{so_a}) \tag{2.26}$$

It is easily shown that if the numerator of equation (2.23), given above as equation (2.26) is also zero, then the facet search line and facet side must be coincident. First, the numerator of 2.23 gives twice the area of a triangle formed by points B_{so}, B_{mid}, and B_{i+1}. Given that these points should actually be three separate, non-coincidental points, the area of the triangle can be zero, only if the points are collinear. See figure 2.20. Remember that B_{mid} and B_{i+1} are two points on the facet side. Since the facet search line also originates from B_{so}, if the facet search line is parallel to a facet side, implying that equation (2.25) is zero, and at the same time equation (2.26) is determined to be zero then the search line and facet side must be coincident with the line drawn from B_{so} to B_{mid}.

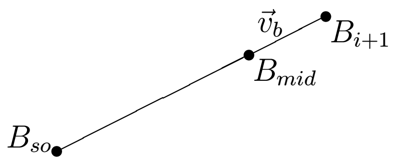


Figure 2.20: Collinear Points

For our purposes will will state that equation (2.25) or equation (2.26) are not considered zero if their absolute value is larger than an arbitrarily small fraction of the area formed by the current two dimensional bounding box.

Note that the case of zero divided by zero for equations (2.23) and 2.24 only arises when the facet search line is coincident with a facet side. Whenever this unique case is detected, it is important to determine if the arbitrary point B_I is on the facet side. This is accomplished by determining where B_I is located relative to the facet side;

$$t_b = \frac{\vec{v}_b \cdot (B_I - B_{mid})}{\|\vec{v}_b\|} \tag{2.27}$$

If $-1 - \epsilon < t_b < 1 + \epsilon$, then B_I is considered to be on the facet side, otherwise B_I is not on the facet side.

Thus to summarize what we have covered in this section so far, first we have defined the facet search origin, facet search line, and facet side. Before we can perform equations (2.23) and (2.24), we must perform equation (2.25) to show that $\vec{v}_s$ and $\vec{v}_b$ are not parallel. If these two vectors are considered parallel then it is necessary to perform equation (2.26) to determine if the search line and facet side are coincident. For the case that the lines are not coincident, no intersection can exist. If the lines coincide then equation (2.27) is used to determine whether B_I is on the facet side, if this is the case then we have detected a singularity. This topic will be discussed further in this section. If B_I is not on the facet side, then we continue and examine the next facet side.

Handling the Solution

Given that $\vec{v}_s$ and $\vec{v}_b$ are not parallel, equations (2.23) and (2.24) provide the parametric terms that serve as the single solution. Table 2.2 is used to determine what relationship exists between the facet search line and facet side element.

Table 2.2: **Intersection Cases**

Search	Side Element t_b		
Line t_s	$\lvert t_b\rvert < 1-\epsilon$	$1-\epsilon \le \lvert t_b\rvert \le 1+\epsilon$	$\lvert t_b\rvert > 1+\epsilon$
$t_s > 1+\epsilon$	MISS	MISS	MISS
$1-\epsilon \le t_s \le 1+\epsilon$	CASE 1	CASE 3	MISS
$t_s < 1-\epsilon$	HIT	CASE 2	MISS

The columns of the table correspond to three ranges of value that a side element's parametric term may have. The rows of the table correspond to three ranges of values that the facet search line's parametric term may have. The term ϵ is simply an arbitrary small number that we define. The ranges selected may appear odd at first glance, but recall from equation (2.21) and equation (2.22) that constraints were placed on the associated parametric terms. For a side element the parametric term was restricted to $-1 \le t_b \le 1$. Likewise for a facet search line the parametric term was restricted to $0 \le t_s \le 1$. Note that since the facet search origin should be outside the facet bounded region, there is no need to test if $0 \le t_s$.

The MISS entries in Intersections Case Table indicate that no intersection exists between the facet search line and side element. The HIT entry in the table indicates that a *clear intersection* exists. The remaining entries are discussed in the following paragraphs.

The entry CASE 1 corresponds to a situation that arises when the intersection point happens to be coincident with the end point of the facet search line. Several topics that have already been introduced merge at this point. First, a facet search line was defined explicitly to be a straight line segment that extends from the associated facet search origin which is outside the bounded region, to an arbitrary point which might be inside the bounded region. Since the arbitrary point is the end of the search line that we are concerned with, it should be obvious that this situation implies that the arbitrary point is coincident or nearly coincident with a point in a side element. Next recall that the arbitrary point corresponds to the intersection between a three dimensional search line and a facet plane. This correspondence is based on the one to one mapping established by equation (2.11) in section 2.4.2. It should be clear that the situation of having a facet search line coincident with a side element corresponds to the cases of singularities that was introduced in section 2.1, and was illustrated by figure 2.5. Thus by detecting CASE 1, Kalay's[24] singularities are detected. Because such a situation is associated with analysis performed on three dimensional shapes, the situation might be referred to as *3D singularities.*

The entry CASE 2 corresponds to a situation that arises when the intersection between a facet search line happens to be coincident with an end point of a side element, see figure 2.21. In such a situation the intersection could imply: (1) A transition from an internal region of the bounded region to an external region (or vice versa) in 2D space. (2) A tangency condition in which the facet boundary is not penetrated and thus no inside to outside transition occurs. Kalay refers to both types of coincidence as singularities. Since this situation arises from the application of the parity count method to two dimensional structures, such singularities might be referred to as *2D singularities.*

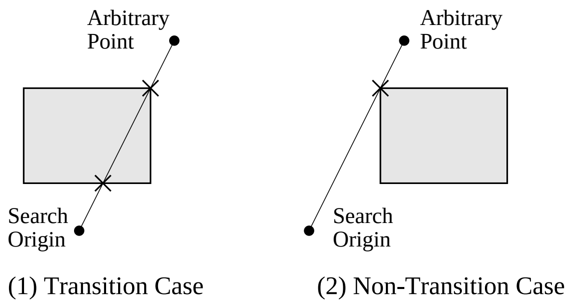


Figure 2.21: Cases of Singularities for 2D Analysis

The entry CASE 3 corresponds to a situation that arises when not only the intersection is coincident with the end of the search line but also the intersection is also coincident with an end point of a side element. Clearly, following CASE 1 and CASE 2, this situation corresponds to a singularity that arises from two sources.

As with all cases of singularities the resolver method was selected to handle them. The associated grid points are marked as unresolved and will not be analyzed further till all facets associated with the current shape have completed processing. It is suggested that the reader review the flowchart in figure 2.14 on page 67 at this point. The mesh resolver is located in the Shape Level Flowchart. The resolver task is discussed in section 2.4.6.

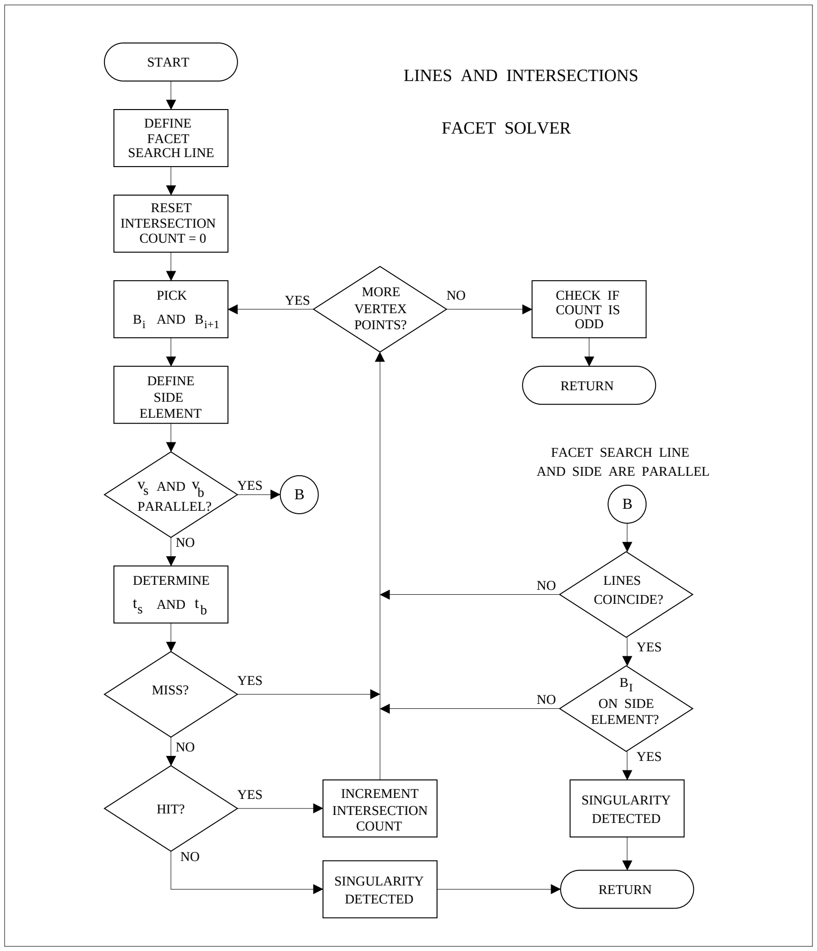


Figure 2.22: Facet Solver Flowchart

Facet Solver Summary

This section provides an overview of the two dimensional containment problem and presented one solution in detail. The lines and intersections method was presented and is summarized by figure 2.22. This section also provided a discussion on detecting singularities. Note that singularities are rarely encountered, thus most execution takes places in the left part of the flowchart. In the flowchart the parity count method is applied in a two dimensional sense, if the arbitrary point B_I is found to be contained in the bounded region then counters associated with surface intersections are incremented.

In addition to the lines and intersection method, other methods exist for solving the two dimensional containment problem. Kalay[24] refers to several methods, one of which is based on the idea of summing angles around a facet. Lee and Preparata[27] present other solutions to the two dimensional containment problem and refer to techniques presented elsewhere by Ketelsen as well as Shamos. Bentley and Carruthers[10] also present techniques along with references to Shamos as well as Lipton and Tarjan. Many of these techniques are summarized by Preparata and Shamos[47], one technique in particular requires on average $2\log(n)$ operations to solve a facet. Unfortunately these and other references were not available when the overall mesh generation algorithm was developed. These references provide solutions to the two dimensional containment problem, that appear to be less computationally intensive then what was presented here. If we had been aware of these references, most likely a different algorithm would have been selected for the Facet Solver Task.

The dependence of the overall performance on the facet solver is examined by complexity analysis in section 2.7. In addition, section 2.10 provides more insight by examining the determinant method, which unfortunately can only handle the special case of convex facets. Based on the information presented, we can get a feel for how much overall performance improvement an alternate facet solver might provide.

2.4.6 Insideness and Resolver Tasks

At this point there should be no surprise regarding the function of the Insideness Task. As introduced way back in section 2.1, after all intersections between search lines and the facets associated with the current shape have been counted, it is time to determine whether immediately resolvable grid points are contained by the shape. Following the parity count method, immediately resolvable grid points which counted an odd number of intersections must be contained by the shape surface. Alternatively, immediately resolvable grid points which counted an even number of intersections must not be contained by the shape. With immediately resolvable grid points taken care of, if any unresolved grid points remain, they are tallied and the count is reported to the user. The resolver is given the task of taking care of any remaining unresolved grid points.

Unresolved grid points are handled by the resolver method, which is performed by making a simple observation; Unresolved grid points inside the shape should notice that neighbor grid points are contained by the shape. Conversely, unresolved grid points outside the shape should notice that neighbor grid points are not contained by the shape. While it is true that a grid point deep inside the mesh bounding box has six nearest neighbors or twenty six neighbors in total, only two neighbors are selected from opposite directions. See figure 2.23.

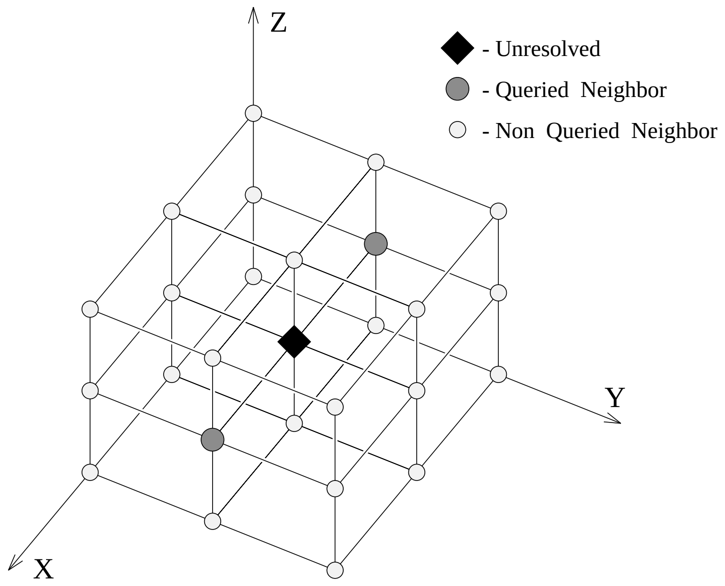


Figure 2.23: Two Point Resolver Neighbors

The resolver handles all unresolved grid points, effectively all at the same time. To do this, the facet solver temporarily stores the state of neighboring grid points before applying the precedence method, in this way there is no possibility of having a pattern “propagated” by the facet solver. Table 2.3 lists the possible cases that might arise while performing the two point resolver method.

The first two cases listed in table 2.3 should characterize most of the unresolved grid points that are ever encountered, and based on the information already

Table 2.3: **Two Point Resolver Cases**

1)	Both neighbors queried are contained by the shape.
2)	Neither neighbor is contained by the shape.
3)	Only one neighbor is contained by the shape.
4)	One neighbor is not contained, the other is unresolved.
5)	Both neighbors are unresolved.

presented should be self explanatory.

The third and forth cases represents a weakness of the resolver, as it only queries two neighboring grid points. The third case usually represents a situation where the unresolved grid point is physically close to the shape surface. For this case the “other” neighbor most likely would not be contained by the shape, but could possibly be another unresolved grid point. The third case is handled by arbitrarily declaring the unresolved grid point in question as being contained by the current shape.

While in the fourth case the unresolved grid point in question is most likely not contained by the shape, it could potentially be physically close to the shape surface. The fourth case is handled by arbitrarily declaring the unresolved grid point in question, as being not contained by the shape.

Assuming that the event of having adjacent unresolved grid points is exceedingly rare, then such arbitrary decisions as declaring a grid point as being ‘contained’ or ‘not contained’ should not be a problem. In comparison to an entire orthogonal mesh, the presence or absence of extra mesh boxes near the surface of shapes, in exceedingly small numbers (fewer than one, on average) should not be a problem.

The fifth case represents a particular problem where the two point resolver “falls apart.” Such a grid point is referred to as an *unresolvable grid point.* For the two point resolver to provide reasonable results, the probability of encountering singularities must be comparatively small so that the probability of having three unresolved grid points as in-line neighbors would be nearly impossible. The resolver method handles unresolvable grid points arbitrarily by declaring them as not contained by the current shape.

Future work could investigate ways of enhancing the resolver method, for example more neighbors could be queried. A decision of containment could be made based on the average of the responses. Such an enhancement would better handle cases four, five, and six.

Probability of Singularities

As pointed out, for the resolver method to provide reasonable results, the probability of encountering a singularity must be small. A simple technique for reducing the probability of encountering singularities is based on the special property that linear combinations of irrational numbers or rational and irrational numbers that do not result in zero, are irrational. While it is not possible to represent an irrational number exactly by using finite precision floating point numbers, it is noted that such a representation is "close to irrational." Both versions of the existing code use double length floating point variables, which provide a better approximation of irrational numbers than single precision floating point numbers. If the term *unique* is used to describe the finite precision approximation of an irrational number, then the modified statement will nearly always be true for the situations that we have in mind.

Next, note that in nearly all situations, the coordinate values associated with vertex points are *typical*, that is rational or closely related to rational numbers[8]. This is important as every point on every edge is expressed by using parametric equations, thus is a linear combination of the coordinate values of vertex points.

Recall that a singularity is caused when any given search line intersects the edge of a shape. Clearly, if we wisely choose a *unique location*[9] for the search origin, then except for the associated grid point, every point on the search line will have unique coordinate values. If floating point numbers had infinite precision then for special cases it would be possible to guarantee that singularities would never be encountered.

Experience has shown that even with the use of finite precision numbers, and with typical examples, by selecting a *unique location* for the search origin, the probability of encountering singularities is in fact, greatly reduced. Experience has also shown that any extreme corner of the bounding box represents a particularly "bad" location for the search origin. Such a location virtually guarantees that singularities will be encountered.

In summary, the reason for selecting a *unique* location for the search origin is

[8]The number π or the square root of an integer are often encountered as component factors in *typical* numbers.

[9]Implies that the coordinates of the search origin are unique

to minimize the number of unresolved grid points that will be encountered. While the selection of such a location is essentially arbitrary, it is important to remember the point made in section 2.4.4 that the search origin should be at least a distance from the bounding box, equal to the long diagonal of the bounding box. Refer to section 2.2.6 for the notation and equations used to define the search origin.

2.4.7 Copy Out Mesh Task

After it is known which grid points are contained by a shape, it is the duty of the Copy Out Task to assign material property identifiers to boxes in the overall mesh. Since the workspace that was prepared for the current shape is separate from the overall mesh, and given that a shape must be uniform in composition, a bitmap conveniently indicates if each associated grid point is contained by the current shape. For each bit in the bit map, *true* indicates that the corresponding grid point is contained by the current shape. A value of *false* indicates the converse. To implement the precedence method introduced in section 2.2.3, for each bit that is *true*, the Copy Out Task compares the material identifier associated with the current shape to that of the corresponding box in the overall mesh. The precedence method says that a mesh box can only be assigned a new material identifier if one has not yet been assigned, or if the previously assigned material identifier has a numerically smaller value.

The Copy Out Task essentially copies results out of the current workspace, hence the name. After the Copy Out Task has completed its job, if another shape is going to be analyzed, the workspace is returned to the Initialize Shape Task.

2.4.8 Store Results In Files Task

This task has the job of writing the finished mesh to files. The files written are selected by the user in a dialog fashion with the program. Of the file types that can be written, text and Ideas files were intended primarily for visualization purposes. Patran files are used for visualization and analysis. The mesh file format is read only by the FDTD solver.

2.4.9 The Mesh Algorithm Summary

This section provided an in depth discussion of the mesh generation algorithm. While it has not yet been formally presented how the algorithm was used in a parallel environment, section 2.4.3 introduced some interesting ideas. This next section presents a time complexity analysis and will attempt to present a compelling argument for one particular type of parallel implementation.

2.5 The Detection of Singularities

The detection of singularities presents a collection of important issues; What is the largest threshold value ϵ (see section 2.4.5) that we can use and still guarantee that the resolver method will provide reasonable results? Alternately, what is the smallest possible threshold value that we can use and still guarantee that singularities will be detected? Is it possible to make such absolute guarantees? Unfortunately such questions can only be answered in the realm of numerical analysis and discrete probability theory. While these topics are important, for two primary reasons it was decided to not perform a comprehensive treatment of these topics. First, such topics are beyond the scope of this thesis. The development of the mesh generator and FDTD solver has already been a sizable software development. Second, while a comprehensive treatment may be a useful intellectual activity, it is not needed to obtain useful results from the mesh generator program.

To determine a reasonable threshold value, a shortcut based on experimental data was used. While it is certainly true that the number of unresolved grid points is dependent on the geometry of the problem, the number of facets, and the number of vertex points, it was decided that the small sphere would serve as our model. The shortcut made was based on the assumption that the number of unresolved grid points encountered during an analysis is proportional to the number of grid points in a mesh.

To examine the relationship between the number of unresolved grid points, the number of mesh boxes, and and the threshold (ϵ), three mesh sizes were considered: 20x20x20, 30x30x30, and 50x50x50. For several ranges of threshold values, the number of unresolved grid points encountered was recorded. Note that for each mesh size, threshold values were selected to be large enough so that unresolved grid points would actually be seen. For each mesh, the numbers of unresolved grid points encountered was divided by the number of mesh boxes. If there is an approximately linear relationship between the number of unresolved grid points and the number of mesh boxes in a mesh, we should see it in a graph of the data. Figure 2.24 is a graph showing the relationship for each mesh size, between the number of unresolved grid points and the threshold values. The closeness of the curves implies that there definitely is a correlation between the number of mesh boxes and the number of unresolved grid points in a mesh.

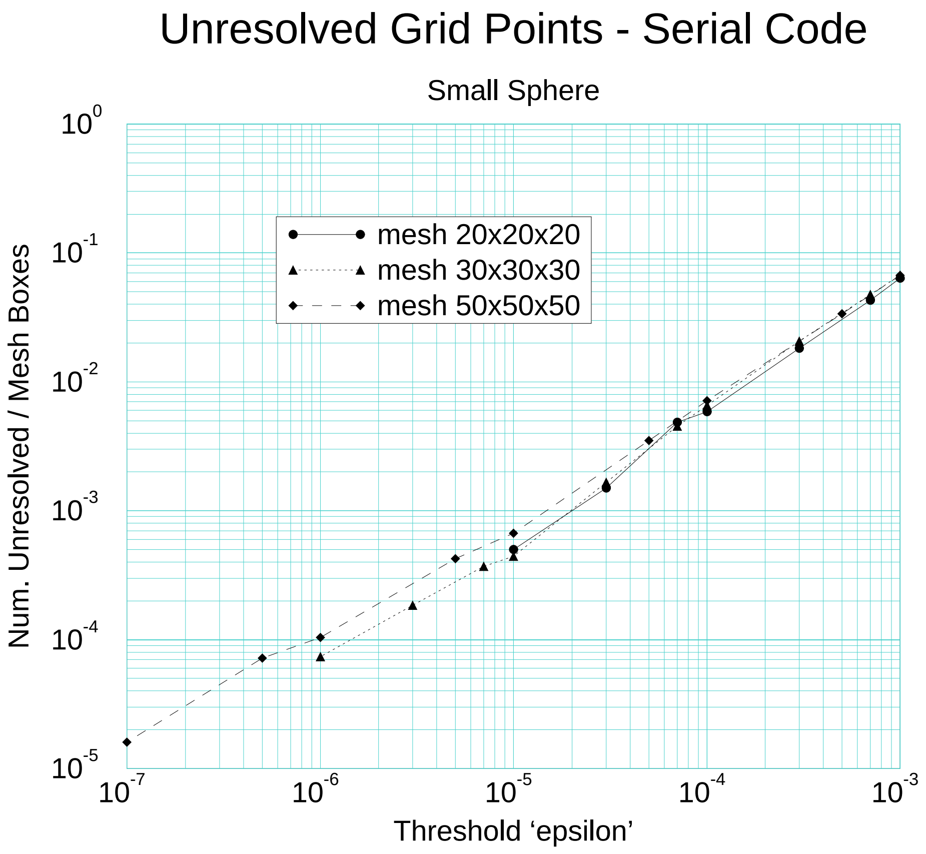


Figure 2.24: Detection Data for Three Mesh Sizes

It was arbitrarily decided that an acceptable threshold value would produce approximately one unresolved grid point in a 100x100x100 mesh model of the small sphere, which corresponds to a detection ratio of 10^{-6}. To estimate the required threshold value, the data from the 50x50x50 mesh was used. By simply drawing a straight line on the curve as shown in figure 2.25, it appears that a threshold value of approximately 10^{-8} should provide the desired performance. Thus it was decided to use 10^{-8} as the threshold (ϵ) value.

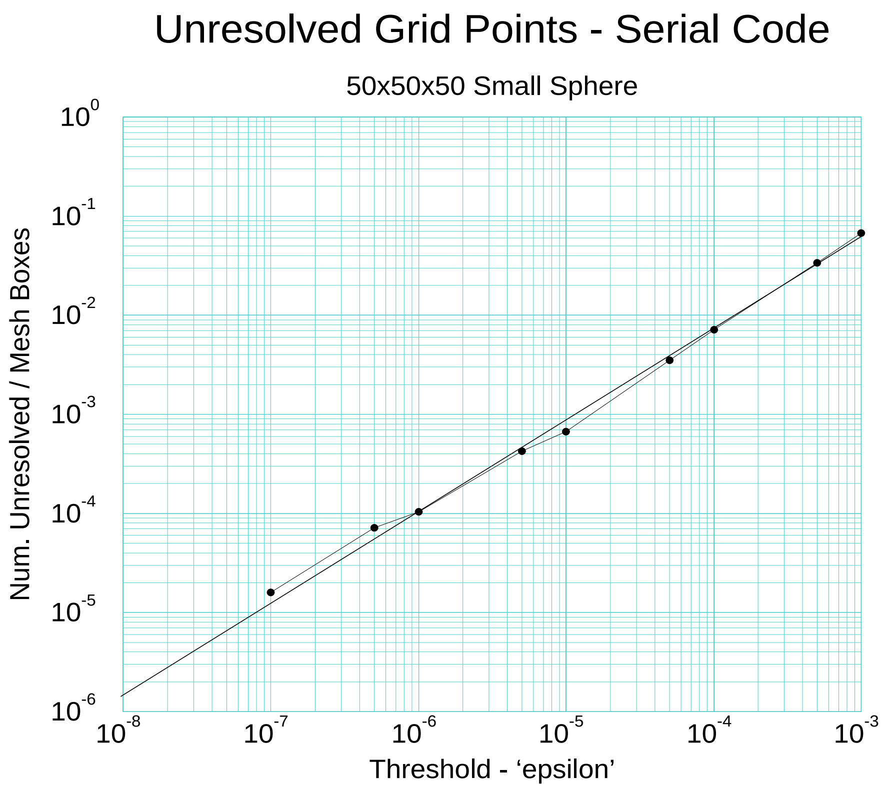


Figure 2.25: Estimate of Detection Threshold

2.6 Introducing Complexity Analysis

One of the classic texts in regards to complexity analysis was written by Aho, Hopcroft and Ullman[2]. The following is a quote[10] from from their text, the quote serves as a useful introduction to this section.

> "Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We would like to associate with a problem an integer, called the *size* of the problem... For example, the size of a matrix multiplication problem might be the largest dimension of the matrices to be multiplied. The size of a graph problem might be the number of edges. The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm."

While many problems can adequately have *size* specified by a single integer, this is not generally true. For most problems, it is usually necessary to state additional assumptions or conditions that apply. For example, consider the mail sorting problem where the goal is to route a set of letters received all at once, such that each letter is correctly placed in a matching mailbox. Suppose that we wish to describe the ammount of time that is reqired to perform this task, based solely upon the number of letters received. We might first suspect that there exists a simple linear relationship between time and the number of letters received, but consider the following: First, it can be easily shown that if a large stack of received letters is first sorted, then the average time required to find the correct mailbox for a given letter can be made much smaller than for the case that if only a small number of letters were received. Second, a moderately large mailroom may have additional staff that can be called on to process large mail shipments. Only if we *assume* that a fixed number of mail personell will follow a strict protocol in handling mail, then we might be able to justify such a linear time complexity relationship. The point being made here is that the cost of being able to describe problem size with a single number, is that we must encumber ourselves with the burden of such assumptions.

Time complexity analysis is most often used as a theoretical tool. While Aho, Hopcroft and Ullman point out that the goal of time complexity analysis is to characterize the time or space needed to solve a problem, based on problem size.

[10]page 2 of Aho, Hopcroft and Ullman[].

they continue to point out that time complexity is usually examined in a special way;

> "The limiting behavior of the complexity as size increases is called the *asymptotic time complexity.*"

What this means is that we are less interested in determining the exact ammount of time it takes to solve a problem, but that we have a general rule that characterizes how time scales with problem size. Asymptotic time complexity is often represented by using a standard *big - oh* notation, which Aho, Hopcroft and Ullman describe;

> "If an algorithm processes inputs of size n in time cn^2 for some constant c, then we say that the time complexity of that algorithm is $O(n^2)$, read "order n^2". More precisely, a function $g(n)$ is said to be $O(\,f(n)\,)$ if there exists a constant c such that $g(n) \leq c\,f(n)$ for all but some finite (possible empty) set of non-negative values for n."

Knuth[25][11] provides a far more in depth explanation of big-oh notation that the reader is referred to. In addition Baase[5][12] provides a list of properties associated with big-oh notation. The reader is referred to Bentley[7][8], who has written two papers that serve particularly well as introductions to the field of complexity analysis.

Aho, Hopcroft, and Ullman[13] point out, one of the most important motivations for complexity analysis and one of the reasons why formal models of computation are developed is to;

> "discover the inherent computational difficulty of various problems. We would like to prove lower bounds on computation time… to show that there is no algorithm to perform a given task in less than a certain amount of time… "

[11]See page 104 of Knuth
[12]See page 34 of Baase
[13]See page 4 of Aho et. all.

This last quote is in agreement with Kronsjö[26] who states that[14]

> "The two major questions of arithmetic complexity—what is the minimum number of arithmetic operations needed to perform the computation and how can we obtain a better algorithm…"

This document does make use of techniques that are commonly used in the field of complexity analysis. Such techniques are used in presenting the parallel algorithm. Where appropriate, this document will consider more efficient algorithms.

2.7 Time Complexity

In this section we will examine the time complexity of the mesh generation algorithm. In examining algorithm complexity, it was found to be useful to study the serial model of the algorithm. With a serial model, we won't be so concerned with the actual computer that we run the code on. Also, if it is possible to construct parallel and serial implementations based on a single algorithm, both implementations should perform the same type and number of opertions. For now the reader will have to be content with the notion that this section provides important definitions and clues that will serve as an introduction to the parallel implementation of the orthogonal mesh generator.

2.7.1 Problem Size

In the following, parameters that describe problem size are introduced. First, the parameters associated with the size of an input solid file are presented in table 2.4. Shapes are described separately and each is represented by a system of facets. In turn each facet is defined by a sequence of referenced vertex points.

The parameters presented in table 2.5 are associated with the resultant mesh. Note that since every grid point in the overall mesh may not be examined for containment by every shape, the number of mesh boxes in the analysis subregion of each shape can be expressed as a ratio of the total number of boxes. In addition, the term as_i is defined to be the set of all grid points in the analysis subregion associated with shape i.

[14]See page 7 of Kronsjö

Table 2.4: **Solid File Size Parameters**

Term		Description
N_s	–	Number of shapes
N_f	–	Total number of facets
$N_{f(i)}$	–	Number of facets associated with shape i
$\tilde{N}_{fs}$	–	Average number of facets per shape
N_v	–	Total number of vertices
N_{vi}	–	Number of vertices in shape i
$\tilde{N}_{vs}$	–	Average number of vertices per shape
$N_{v(i,j)}$	–	Number of vertices on facet j associated with shape i
$\tilde{N}_{vf}$	–	Average number of vertices per facet

Table 2.5: **Mesh Size Parameters**

Term		Description
N_x N_y N_z	–	Number of boxes along each axis of a complete mesh
N_m	–	Total number of boxes in the overall mesh = $N_x\,N_y\,N_z$
N_{bi}	–	Number of boxes in the shape i analysis subregion
R_i	–	Ratio of N_{bi} to N_m

2.7.2 Time Complexity by Task

This subsection contains a discussion of the time complexity. The contribution of each task is examined and a final time complexity is then determined.

Initialization Tasks

For our purposes, since the Initialization Analysis Task is performed once and is nearly independent of problem size, we shall assign its contribution a time complexity of;

$$\mathrm{CX}_{IA} = O(\,1\,)$$

Likewise, since the Initialize Shape Task is performed for each shape and has the effect of reading in every vertex point, we shall assign it a complexity dependent on the total number of vertex points;

$$N_v = \sum_i N_{v(i)} = N_s \left(\frac{1}{N_s}\right) \sum_i N_{v(i)} = N_s\,\tilde{N}_{vs}$$

$$\mathrm{CX}_{IS} = O\,(\,N_v\,) = O\left(\,N_s\,\tilde{N}_{vs}\,\right)$$

Facet Server

By examining figure 2.16 it is seen by inspection that the facet server itself has a time complexity of $O\left(\,N_{v(i,j)}\,\right)$. Since the facet server is performed for each facet, where a given facet has $N_{v(i,j)}$ vertex points, we can determine the total number of references to vertex points.

$$\mathrm{ref}_{FS} = \sum_{i,j} N_{v(i,j)} = N_s \left(\frac{N_f}{N_s}\right) \frac{1}{N_f} \sum_{i,j} N_{v(i,j)} = N_s\,\tilde{N}_{fs}\,\tilde{N}_{vf}$$

Thus from the facet server the total contribution to the execution time is;

$$\mathrm{CX}_{FS} = O\left(\,N_s\,\tilde{N}_{fs}\,\tilde{N}_{vf}\,\right)$$

Pick Next Grid Point

This task amounts to a typical loop control structure. In determining time complexity, this task can be thought of as being part of the Intersection Finder Task.

Intersection Finder

By examining figure 2.18, it can be seen by inspection that the intersection finder itself is $O(1)$. For each shape analyzed there is a corresponding analysis subregion as_i. The intersection finder is performed for each facet, for every grid point in the current analysis subregion. Based on the number of times the intersection finder is called, we can determine the contribution to the execution time.

$$\mathrm{calls}_{XF} = \sum_i N_{bi} N_{fi} = N_m \sum_i R_i N_{fi} = N_m N_s \left(\frac{1}{N_s} \sum_i R_i N_{fi} \right)$$

$$\mathrm{calls}_{XF} = N_m N_s \tilde{F}_s$$

The variable $\tilde{F}_s$ is related to the average number of facets per shape, for example if $R_i = 1$ for all i, then $\tilde{F}_s = \tilde{N}_{fs}$. Note that in general, $\tilde{F}_s$ can be expressed as follows;

$$\tilde{F}_s = \hat{R}_{XF} \tilde{N}_{fs}$$

Since R_i is constrained to the range $0 < R_i \leq 1$ for all i, it can easily be shown that $0 < \hat{R}_{XF} \leq 1$. Thus the number of times that the Intersection Finder Task is called will be;

$$0 < \mathrm{calls}_{XF} \leq N_m \, N_s \, \tilde{N}_{fs}$$

Thus the contribution of the Intersection Finder Task to the overall execution time is;

$$\mathrm{CX}_{XF} = O\left(N_m \, N_s \, \tilde{N}_{fs} \right)$$

Facet Solver

By following the assumption that the total number of unresolved grid points encountered is negligible in comparison to the number of grid points examined, it can seen by inspection of figure 2.22 that the facet solver itself has a time complexity of $O\left(N_{v(i,j)}\right)$. Each time the intersection finder is called there is a certain probability that the facet solver will also be called. This probability is dependent on the grid point location in the overall mesh given by index values x, y, z, and is also dependent on the shape and facet i, j. The term as_i defines the analysis subregion associated with shape i. Thus the total contribution from the facet solver is roughly proportional to;

$$\mathrm{loops}_{RX} = \sum_{i,j} \sum_{x,y,z}^{\mathrm{as}_i} P_{\substack{v(i,j) \\ g(x,y,z)}} \, N_{v(i,j)} = \sum_{i,j} N_{v(i,j)} \sum_{x,y,z}^{\mathrm{as}_i} P_{\substack{v(i,j) \\ g(x,y,z)}}$$

Note that x, y, and z are restricted to those points in analysis subregion as_i associated with shape i. To simplify things we define;

$$\sum_{x,y,z}^{\mathrm{as}_i} P_{\substack{v(i,j)\\ g(x,y,z)}} = N_{bi}\tilde{P}_{v(i,j)}$$

Note that we find that;

$$\tilde{P}_{v(i,j)} = \frac{1}{N_{bi}} \sum_{x,y,z}^{\mathrm{as}_i} P_{\substack{v(i,j)\\ g(x,y,z)}}$$

Thus the term $\tilde{P}_{v(i,j)}$ can be regarded as the average probability that a three dimensional search line will intersect facet j on shape i. Clearly $\tilde{P}_{v(i,j)}$ must be a positive number that is less than or equal to unity.

Thus we return to the analysis to find that;

$$\mathrm{loops}_{RX} = \sum_{i,j} N_{v(i,j)} N_{bi} \tilde{P}_{v(i,j)} = N_m \sum_{i,j} N_{v(i,j)} R_i \tilde{P}_{v(i,j)}$$

$$\mathrm{loops}_{RX} = N_m N_s \left(\frac{N_f}{N_s}\right) \left(\frac{1}{N_f} \sum_{i,j} N_{v(i,j)} R_i \tilde{P}_{v(i,j)}\right)$$

To simplify further we will state that;

$$\tilde{V} = \frac{1}{N_f} \sum_{i,j} N_{v(i,j)} R_i \tilde{P}_{v(i,j)} = \tilde{N}_{vf} \hat{R}_{RX}$$

Since R_i and $\tilde{P}_{v(i,j)}$ are positive numbers less than or equal to unity we can state that $\tilde{V}$ is related to the number of vertex points per facet, such that $0 < \hat{R}_{RX} \leq 1$. Next we obtain the following;

$$\mathrm{loops}_{RX} = N_m N_s \tilde{N}_{fs} \tilde{N}_{vf} \hat{R}_{RX}$$

Thus the total number of loops performed in the facet solver can be expressed as;

$$0 < \mathrm{loops}_{RX} \leq \left(N_m\, N_s\, \tilde{N}_{fs}\, \tilde{N}_{vf}\right)$$

Lastly, the total contribution from the Facet Solver Task to the overall execution time is found to be the order of;

$$\mathrm{CX}_{RX} = O\left(\, N_m\, N_s\, \tilde{N}_{fs}\, \tilde{N}_{vf}\,\right)$$

Check Insideness

This task is performed once per shape. Each time this task is called, every grid point in the corresponding analysis subregion is checked for containment by the current shape.

$$\text{checks}_{CI} = \sum_i N_{bi} = N_m \sum_i R_i = N_m N_s \left(\frac{1}{N_s} \sum_i R_i \right)$$

$$\text{checks}_{CI} = N_m \, N_s \, \hat{R}_{CI}$$

It should be clear that $0 < \hat{R}_{CI} \leq 1$ thus the contribution from the Check Insideness Task is;

$$\text{CX}_{CI} = O\,(\,N_m \, N_s\,)$$

Resolve Singularities

Following the assumption made earlier, we will assume that singularities are rarely encountered. Thus the contribution from this task is considered negligible and is not included in the complexity analysis.

Copy Out

Like the Check Insideness Task, this task is performed once per shape, for every grid point in the current analysis subregion. Thus the contribution from this task should be;

$$\text{CX}_{CO} = O\,(\,N_m \, N_s\,)$$

Store Points in Files

The contribution from this task is particularly difficult to characterize as it is heavily dependent on data transfer rates. The contribution from this task can be significant as some selected file types can be huge. For this reason it was decided that execution time in the overall program is to be reported with and without contributions from this task. For the complexity analysis we will ignore the contribution from this task.

2.7.3 Overall Time Complexity

Table 2.6 provides a summary of the contribution from each task, where each contribution is expressed in terms of big-oh notation.

Table 2.6: **Time Complexity by Task**

Task Name	Term	=	Expression
Initialize Analysis	CX_{IA}	=	$O\,(\,1\,)$
Initialize Shape	CX_{IS}	=	$O\left(\,N_s\,\tilde{N}_{vs}\,\right)$
Facet Server	CX_{FS}	=	$O\left(\,N_s\,\tilde{N}_{fs}\,\tilde{N}_{vf}\,\right)$
Intersection Finder	CX_{XF}	=	$O\left(\,N_m\,N_s\,\tilde{N}_{fs}\right)$
Facet Solver	CX_{RX}	=	$O\left(\,N_m\,N_s\,\tilde{N}_{fs}\,\tilde{N}_{vf}\,\right)$
Check Insideness	CX_{CI}	=	$O\,(\,N_m\,N_s\,)$
Copy Out	CX_{CO}	=	$O\,(\,N_m\,N_s\,)$

To get a feeling for what table 2.6 is implying, consider the following line of thought. Based on experience it has been found in general that $\tilde{N}_{fs}$ tends to be less than 10, also $\tilde{N}_{vs}$ and $\tilde{N}_{vf}$ tend to be on the order of 100 for contoured shapes but may be on the order of 10 for simpler shapes. In contrast, since N_m is equal to the product of three integer values, that is $N_m = N_x N_y N_z$, it should be clear that N_m can potentially be a large number and in most cases will be the largest factor.

To obtain a better understanding, we consider the modeling of a small sphere. For the sphere, $\tilde{N}_{fs} = 256$, $\tilde{N}_{vs} = 242$, $\tilde{N}_{vf} = 3.875$, and $N_s = 1$. A 22 by 22 by 22 mesh was chosen for the model so that $N_m = 10648$, note that such a mesh size is considered modest. For the given model, the contribution from each task model is summarized in table 2.7.

This example shows that the complexity terms that include N_m can easily be made much larger than all other terms. By comparing the contributions from each task, it should be obvious that the intersection finder and facet solver should make the largest contributions to the overall execution time. Thus in general terms, given the results of the 'big-oh' complexity analysis, we might be so bold as to present an expression for the asymptotic behavior of the entire algorithm;

$$\mathrm{CX}_{overall} = \mathrm{CX}_{RX} = O\left(\,N_m\,N_s\,\tilde{N}_{fs}\,\tilde{N}_{vf}\,\right) \tag{2.28}$$

Table 2.7: **A Small Example Case**

Task Name	Term	=	Expression	≈	Order
Initialize Analysis	CX_{IA}	=	$O(1)$	≈	$O(1)$
Initialize Shape	CX_{IS}	=	$O(242)$	≈	$O(10^2)$
Facet Server	CX_{FS}	=	$O(992)$	≈	$O(10^3)$
Intersection Finder	CX_{XF}	=	$O(2725888)$	≈	$O(10^6)$
Facet Solver	CX_{RX}	=	$O(10562816)$	≈	$O(10^7)$
Check Insideness	CX_{CI}	=	$O(10648)$	≈	$O(10^4)$
Copy Out	CX_{CO}	=	$O(10648)$	≈	$O(10^4)$

As a "back of the envelope" exercise, we can use the arguments from table 2.7 to make an educated guess of where the algorithm will spend most of its time. Based on the 'Expression' column we might guess that the Intersection Finder and Facet Solver together will consume 99.83% of the total execution time.

To get an even better idea of how the algorithm is actually spending its time, the example of the small sphere was performed by the serial version of the mesh generator. As before, the mesh produced is 22 by 22 by 22. The execution time was profiled with the MasPar profile tools, the results are summarized in table 2.8. To obtain a meaningful profile, it is important to note that the mesh generator executable was not optimized under compilation.

In table 2.8, the time attributed to each task is listed in milliseconds, in the column 'Time-ms'. The last entry in the 'Time-ms' column provides the total execution time, also in milliseconds. The column 'Percent' gives the percent of the total execution time that was consumed by each task. The column 'Coef.' gives the ratio of the time consumed by each task to the complexity argument given in the column 'Expression' of table 2.7. With these coefficients, given the problem of modeling the small sphere, the execution time can be estimated for any mesh size.

In examining table 2.7, we might expect the facet solver to require more time than the intersection finder task. Yet in examining table 2.8, it is seen that the facet solver is actually consuming slightly less time than the intersection finder. Two things are going on here, first the amount of time consumed by facet solver is dependent on the number of intersections that the intersection finder actually finds. This is an important point as it states that the overall execution time is dependent not only on the problem size but also on the configuration of the

Table 2.8: **Profile Results**

Task Name	Time-ms	Percent	Coef.
Initialize Analysis	≈ 1	0.001	1.00×10^{0}
Initialize Shape	58	0.033	2.40×10^{-1}
Facet Server	150	0.086	1.51×10^{-1}
Intersection Finder	92021	52.495	3.38×10^{-2}
Facet Solver	83007	47.352	7.86×10^{-3}
Check Insideness	29	0.017	2.72×10^{-3}
Copy Out	30	0.017	2.82×10^{-3}
total	175296		

shapes that are being modeled. For shapes that fill the bounding box more fully, the facet solver will be called more often. Clearly in most cases, equation (2.28) will be misleading, since more than half of the execution time time may easily be consumed by the intersection finder. In addition to the dependence on the facet solver on the configuration of the shape being modeled, recall that the time consumed by the Pick Next Grid Point Task was also assigned to the intersection finder. Since the Pick Next Grid Point Task is simply a loop control structure, this additional term is less significant.

In comparing the facet solver and the intersection finder, since the time complexity of the facet solver only has one additional factor, namely $\tilde{N}_{vf}$, a better asymptotic expression can be presented. Since $\tilde{N}_{vf}$ is generally less than ten, and is 3.875 for the case of the small sphere, this term can be incorporated into the assumed coefficient. Thus a better expression for the complexity would include contributions from both the intersection finder and the facet solver. Such an expression can be expressed as;

$$\mathrm{CX}_{overall} = O\left(N_m \, N_s \, \tilde{N}_{fs} \right) \tag{2.29}$$

To summarize this section, complexity analysis provides useful tools for comparing algorithms as well as for characterizing the execution time of a program as a function of problem size. In examining the algorithm, note that 99.85 percent of the total execution time was consumed by the intersection finder and the facet solver routines. This is considered to be very close to the "back of the envelope" estimate that was made earlier.

2.8 Presenting the Parallel Algorithm

A conclusion that is directly raised from the complexity analysis and the profile data given is that nearly all of the execution time is accounted for by two tasks. Rather than writing the entire algorithm in parallel, a sizable speed up should be realized by performing key tasks in parallel. In particular it was found to be a great benefit to implement the entire Facet Level Flowchart in parallel, thus the intersection finder and the facet solver are implemented in the DPU. Figure 2.26 gives an outline of which parts of the algorithm are performed in parallel rather than in serial form; The parts of the algorithm in the shaded region are implemented in parallel in the DPU, while the parts in the unshaded region are implemented in serial in the front end. The following discussion explains each part of this figure.

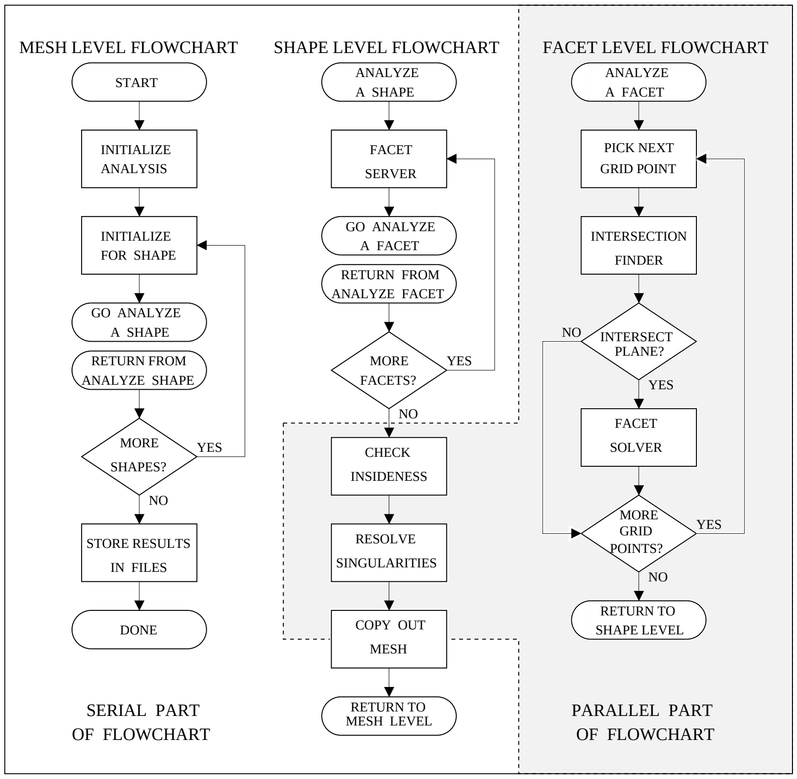


Figure 2.26: Division of Mesh Generation Algorithm Tasks

It was a logical choice to perform the facet server in the front end. The work of the facet server is inherently composed of serial operations that are best handled by a high performance uniprocessor. The facet server performs such activities as reading a serial data stream, searching through a small list for coordinate values, and performing a small list of calculations for each facet, as needed. While the

ACU is certainly capable of performing this task, the ACU is not optimized for such serial operations.

From an abstract viewpoint, for each shape that is modeled, the facet server prepares facets and the Facet Level Flowchart consumes facets. For each shape that is modeled in turn, a workspace is established in the DPU. In such a scenario, the entire Facet Level Flowchart is performed in parallel in the DPU. In one sense, a multitude of copies of the Facet Level Flowchart are performed simultaneously. In addition, since the workspace established in the DPU to model each shape is inherently parallel, the Check Insideness and Resolve Singularities Tasks are best performed in parallel as well.

It was an arbitrary decision to store the overall mesh in the front end. In such a scenario the DPU only needs to produce a bitmap indicating the containment of every grid point by the current shape. Essentially only one bit, in a yes/no fashion is needed to specify if each grid point is contained by the current shape. Use of a bitmap was chosen primarily as a means of reducing the overhead of communications from the DPU to the front end. Thus the Copy Out Task takes on the additional role of organizing each bitmap and converting it from parallel to serial form. It was found to be most useful to implement parts of the Copy Out Task on the DPU and front end. The parallel part of the Copy Out Task organizes the bitmaps. The serial part of the Copy Out Task actually performs the precedence method that assigns material identifiers to boxes in the overall mesh. Lastly the remaining tasks and the entire Mesh Level Flowchart are performed in serial.

2.9 Anticipating Speed-Up

Lewis and El-Rewini[28] present one form of the speed-up equation that was presented in section 1.7.4 of this document. The equation presented by Lewis and El-Rewini was made famous because of a statement made by Amdahl[15] when he expressed his skepticism of computer parallelism. Known as Amdahl's law, the formula first assumes that some fraction ψ of a program is "naturally parallel" and as such the program can be improved by evenly dividing that ψ fraction of the program among N_p processors. Thus the execution time T_{after} is found to be;

$$T_{\text{after}} = \left(\frac{\psi}{N_p} + (1 - \psi)\right) T_{\text{before}}$$

[15]Gene Amdahl, 1967, see page 31 of Lewis & El-Rewini

The expression for T_{after} is substituted into the expression for speed-up, which is simplified to yield Amdahl's law;

$$\text{speed-up} = \frac{N_p}{N_p\,(1-\psi)+\psi}$$

There are a few additional assumptions that were made while forming Amdahl's law, we discuss a few of these in turn. We assumed that all the processors used are identical and that the "naturally parallel" part could be evenly divided among N_p processors without cost. First off, array processors typically provide parallelism by implementing a multitude of simple processors. The MasPar MP–1 series DPU uses four bit processors. What we are calling T_{before} is the execution time of the serial code on the front end. Clearly there is a huge disparity between the performance of a DECstation 5000 and an individual PE in the DPU. Since in general terms the front end is a 24 MIPS machine and a PE is a 1.8 MIPS machine, some scaling of the N_p number of processors is in order, an approximation might be;

$$N_p^{scale} = N_p \frac{1.8\ \text{MIPS}}{24\ \text{MIPS}}$$

Clearly, there will be some overhead in dividing the ψ part of the program among a group of processors. Characterizing this system overhead is not easy, not all PEs are always used, the ACU requires some time to coordinate PEs, the front end requires some time to coordinate processes, and communications between the front end and the DPU requires time to actually move data as well as ACU instructions. For now we will ignore these terms.

Another major assumption made by Amdahl's law is that the amount of parallelism in a program is independent of problem size. Lewis and El-Rewini[28] point out that this assumption is most often false. Lewis and El-Rewini present the Gustafson-Barsis Law[16], which is an alternative to Amdahl's law. The Gustafson-Barsis Law makes the point that the problem size can be used to increase the "naturally parallel" part of a computer program. Relative to Amdahl's law, the Gustafson-Barsis law gives a more optimistic view of parallelism. The Gustafson-Barsis law is not presented here, readers are referred to Lewis and El-Rewini.

Lastly it is not clear what is meant by "naturally parallel", this definition is crucial as Amdahl's law become ever more sensitive to changes in ψ, as ψ approaches unity. In the example of the small sphere in section 2.7.3, the job of producing a modest size mesh had 99.85% of its execution time attributed to only two tasks.

[16]Page 32 of Lewis and El-Rewini

For the sake of simplicity we consider two values for ψ, we pick $\psi_1 = 0.9985$ and pick $\psi_2 = 0.90$. We find that speed-up$_1 = 68.96$ and speed-up$_2 = 8.95$. Note that even though ψ was varied by approximately ten percent, the anticipated speed-up changed an amount that is many times that amount.

Thus to summarize, Amdahl's law is a famous equation. We use Amdahl's law here only as an indicator that a parallel version of the mesh generator should have a significant speed-up in comparison to the serial version. Based on the data presented, we might guess that the speed up would be at least ten times.

2.10 An Alternate Facet Solver

The following is the overview of an alternate solution to the two dimensional containment problem. The following solution was named the *determinant method*. Unfortunately the determinant method is not considered to be satisfactory as it can only handle the case of convex facets. It is for this reason alone that the determinant method was not selected for our use. The determinant method was implemented in the serial code but not in the parallel code.

2.10.1 The Determinant Method

Appendix D introduces equation (D.7), which can be used to determine if a sequence of three arbitrary points in 2D space follow a counter-clockwise (CCW), clockwise (CW) circular path, or a straight line (SL) path. Such an equation is particularly useful and can be used to form a solution to the two dimensional containment problem, but is suitable only for the special case of convex facets.

The algorithm is presented as follows; Take two boundary points B_j, B_{j+1}, at a time and along with the arbitrary point B_I, substitute into equation (D.7), which is written as equation (2.30).

$$\mathrm{dmy}_j = \begin{vmatrix} 1 & B_{j_u} & B_{j_a} \\ 1 & B_{j+1_u} & B_{j+1_a} \\ 1 & B_{I_u} & B_{I_a} \end{vmatrix} \tag{2.30}$$

The variable dmy_j is only used to temporarily hold the result of the expression. The following table indicates the path direction followed by the point sequence B_j, B_{j+1}, B_I.

Table 2.9: Point Sequence Direction

dmy_j	Direction
positive	Counter-Clockwise
zero	Straight Line
negative	Clockwise

If it is found that for all values of j of the current facet, that all the paths formed follow a clockwise path, or alternatively all paths formed follow a clockwise path, then the arbitrary point must be contained by the facet. If the path direction changes from clockwise to counter-clockwise, or alternately from counter-clockwise to clockwise for any value of j then the arbitrary point is not contained by the facet.

If for any value of j, the associated value of dmy_j is found to be zero, then the path followed is along a straight line, implying that the points B_j, B_{j+1} and B_I are all collinear. In such a situation if it is found that B_I is between B_j and B_{j+1}, or if B_I is coincident with B_j or B_{j+1}, then it is said that a singularity has been detected, in which case the resolver will handle the associated grid point later. If however, B_j, B_{j+1} and B_I are all collinear, but B_j is not between or coincident with the other points, then the arbitrary point is clearly not contained by the facet.

To calculate the determinant given in equation (2.30), the following formula is actually used. The following formula is used rather than actually calculating the determinant as the formula is much more efficient for this special case,

$$\begin{aligned} \mathrm{dmy}_j \quad = & \quad B_{j_u} \left(B_{j+1_a} - B_{I_a} \right) \\ + & \quad B_{j+1_u} \left(B_{I_a} - B_{j_a} \right) \\ + & \quad B_{I_a} \left(B_{j_a} - B_{j+1_a} \right) \end{aligned} \tag{2.31}$$

Note that in performing equation (2.31), two adds, three subtracts, and three multiply operations are performed on floating point data, a total of eight floating point operations.

2.10.2 Close Examination

The flowchart in figure 2.27 presents a summary of the determinant method of solving facets. It can be seen by examining the flowchart that generally the number of iterations will be less than the number of facet boundary points, this is a certain advantage over the lines and intersection method which must examine every edge associated with a facet. To character the complexity the time complexity of this facet solver however, it is actually $O\left(N_{v(i,j)}\right)$ for each facet.

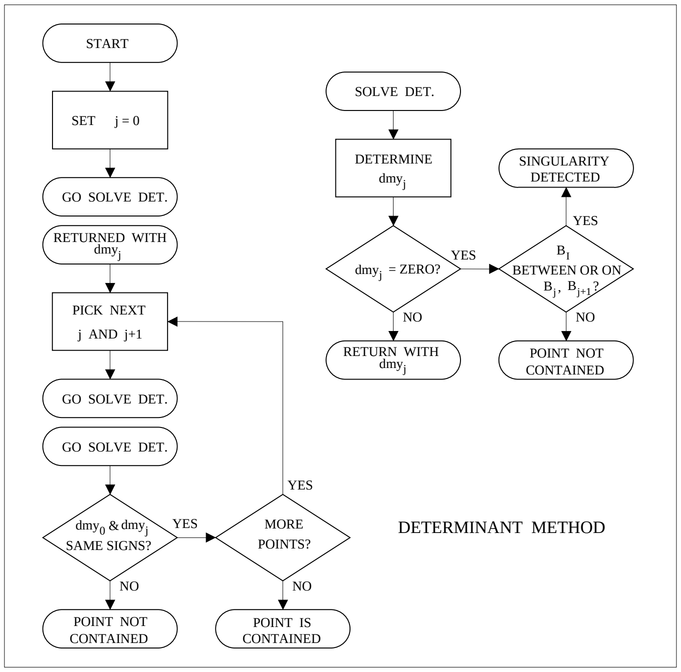


Figure 2.27: Flowchart of Determinant Method Facet Solver

In comparing flowcharts, the determinant method is much simpler than the the lines and intersection method. Each iteration inside the determinant method requires fewer floating point operations. Assuming that no singularities are encountered, the determinant method requires 8 floating point operations per iteration, compared to the lines and intersection method which requires 19 floating point operations. Based on these figures it is expected that the determinant method should yield a significant improvement in the overall performance.

2.10.3 Comparing Performance

To compare the facet solver algorithms, two versions of the mesh generator were timed by using the MasPar profile tools. Both executables were not optimized and produced a 22 x 22 x 22 orthogonal mesh representation of the small sphere as was performed in section 2.7.3. The total CPU execution time associated with each version of the mesh generator is listed in table 2.10.

Table 2.10: **Comparing Serial Mesh Generator Versions**

Version	CPU Time (msec)
lines & intersections	175,250
determinant method	105,199

In examining table 2.10, it is seen that the overall speed-up is 1.66589. To determine how much faster the determinant method is than the lines and intersections method, the speed-up equation (1.7) was solved for the enhancement factor E_f.

$$E_f = \frac{\psi \text{ speed-up}}{1 - \text{speed-up}(1 - \psi)}$$

The percent of the total execution time consumed by the lines and intersection method is ψ, and was given by table 2.8 as 0.47352. After substituting values into the above equation, it is found for this example that the determinant method is 6.41629 times faster.

2.10.4 So What is Possible?

The overall time complexity summary presented in section 2.7.3 indicates that for large mesh models of objects like the sphere, approximately fifty percent of the total execution time can be attributed to the facet solver. Section 2.10.3 is significant as it clearly shows that a measurable improvement is possible, even with less than one order of magnitude performance improvement in the facet solver itself. Unfortunately as stated earlier, the primary failing of the determinant method is that it can only handle the special case of convex facets. Given that the facet solver consumes 50% of the total execution time, for this type of problem the limiting factor on the overall speed-up over the existing algorithm is approximately 2.0.

2.11 Parallel Mesh Algorithm Summary

Along with many details associated with orthogonal mesh generation, this chapter presents the mesh algorithm used to implement a parallel realization the mesh generator. The complexity analysis in section 2.7 provides important insight into the mesh generation algorithm. An outline of the parallel algorithm is presented in section 2.8. Chapter 3 will serve to clarify the parallel algorithm further. Section 2.9 provides a sense of the speed-up that we expect in the parallel implementation. The performance of alternate facet solvers was also considered. Lastly, sections 2.4.4 and 2.4.6 explain why the correct placement of the search origin, relative to the overall bounding box is so important.

Chapter 3

Mesh Generator Implementation

This chapter discusses both the parallel and serial implementations of the orthogonal mesh generator. Important details associated with the implementations are discussed, an outline of the implementations is presented, examples are presented and performance is characterized. Note that although both the serial and parallel implementations are discussed, the parallel version is emphasized.

In the following few sections the most significant implementation details are presented. Such details are fundamental and deserve an appropriate presentation. The details discussed include minimizing data traffic between the front end and the DPU, mapping data, and a scheme for allocating memory. Note that the topic of mapping memory for this implementation is broken down into five related topics: Organization of the overall mesh, defining an analysis subregion, organization of the counters used to implement the parity count method, and mapping memory to the PE array.

3.1 Transferring Data with the DPU

During program run-time, some data structures must be updated for every facet, or for each shape. Still other data structures can be regarded as constants. For the parallel version of the mesh generator, this observation is particularly useful as it immediately suggests a means by which data traffic between the DPU and the front end can be controlled. In examining the algorithm presented in section 2.8, four opportunities were found where communications between the front end and the DPU are most convenient for transferring data that needs to be updated for each shape, for each facet, or is considered as constant. Table 3.1 summarizes these opportunities, describes how often the data needs to be transferred, as well as which direction the data is transferred.

Table 3.1: **Data Transfer Opportunities**

	Opportunity	Transfer Frequency	Transfer Direction
1)	Initialize Analysis	Constant Variables	To & From DPU
2)	Initialize Shape	Changes with Shape	To DPU
3)	Facet Server	Changes with Facet	To DPU
4)	Copy Out	Changes with Shape	From DPU

The Initialize Analysis task produces constants that both the front end and the DPU must be aware of. Being constants, it is necessary to have the data transferred only once. An example of such a constant is the search origin. In preparation for a new shape, the Initialize Shape Task will change the values of certain variables. All the variables that could change are transferred by the Initialize Shape Task. An example of data that changes with each shape is the definition of the current analysis subregion. For every facet that is processed by the facet server, new data is produced, and is transferred to the DPU by using a combination of formal parameter passing as well as DMA style copying. Lastly, after the containment of every grid point in the current analysis subregion is determined, DMA style copy operations are used to move a compressed bitmap from the DPU back to the front end. Because an 'unsigned int' type variable is 32 bits long, each entry in an array of 'unsigned int' variables can represent the containment of 32 grid points. This compression of data provides a sizable savings in communications overhead.

3.2 Presenting the Mapping

In the following the mapping scheme is introduced. Note that the scheme presented applies to both the parallel and serial versions of the mesh generator. For clarity, the overall mapping scheme was broken down into four related topics, that are each discussed in turn.

3.2.1 The Overall Mesh

The array 'grid[]' is a linear array of character variables used to store the overall mesh. A character variable can store an integer value ranging from 0 to 255. Since 0 is defined as the default material identifier, users are free to assign up to 255 additional material types. Recall from section 2.2.5 that N_x, N_y, and N_z are the number of brick shaped subdivisions along each edge of the overall bounding box. It should be clear that such a linear array will require $N_{\text{entries}} = N_x \, N_y \, N_z$ entries to store the overall mesh.

As in section 2.2.5, each *referenced mesh box corner* is indexed with an ordered triple of integers $[i, j, k]$. Since the overall mesh is actually stored in a linear array, each ordered triple is mapped to an index value by using the following formula;

$$\text{index}_m \;=\; i + j\,N_x + k\,N_x\,N_y \;=\; i + (j + k\,N_y)N_x \tag{3.1}$$

Since this is a one to one mapping, given an index value from the linear array the following formula will produce the corresponding ordered triple.

$$\begin{aligned} k &= \lfloor \text{index}/\,(N_x\,N_y) \rfloor \\ \text{temp} &= \text{index}\,\%\,\,(N_x\,N_y) \\ j &= \lfloor \text{temp}/N_x \rfloor \\ i &= \text{temp}\,\%\,N_x \end{aligned} \tag{3.2}$$

In the previous expressions, the floor function $\lfloor \cdot \rfloor$ and the divide operator / are used to return the integer part of a quotient, while the modulo operator % returns the integer remainder. The term 'temp' is used simply a temporary dummy variable.

3.2.2 The Analysis Subregion

The idea of an analysis subregion was first introduced back in section 2.4.1 as being a subregion of the overall mesh. This section presents the means by which an analysis subregion is defined, along with additional concepts associated with an analysis subregion.

Defining an Analysis Subregion

Recall from section 2.2.5 that the overall mesh is formed by placing equally spaced marks along each edge of the overall bounding box. An extension of this simple idea is used to first introduce the idea of a mesh subregion. By selecting a range of marks on each edge of the overall bounding box, a smaller bounding box contained by the overall bounding box is easily defined. The set of index values corresponding to such a range of marks is called the *analysis range*. For this smaller bounding box, we again define a mesh by placing equally spaced marks along its edges. The smaller mesh is said to be a *mesh subregion*. The heavily shaded part of figure 3.1 is an illustration of a mesh subregion.

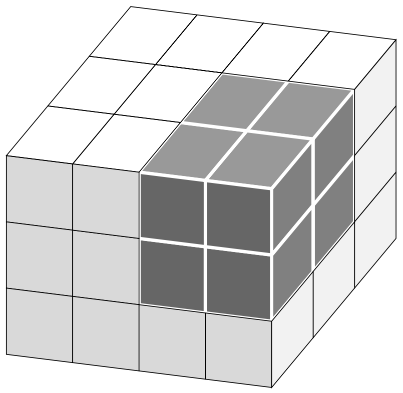


Figure 3.1: Visualization of an Overall Mesh and Subregion

The whole point behind the idea of an analysis subregion is that by using the extreme coordinate values of a shape, an analysis range can be defined that provides a "ball-park idea" of where the shape currently being modeled is located in the overall bounding box. With such information, we don't have to check every grid point in the overall mesh for containment by the shape. It will only be necessary to check those grid points in the analysis subregion.

To define an analysis range, the index values corresponding to the extreme index values are found. To find the maximum and minimum index terms corresponding to the x axis, the following formulas are used. Note that formulas for the y and z axes are formed by a simple change of variables.

$$\text{AR\_min}_i = \left\lfloor \frac{\text{shape\_min}_x - P_{min_x}}{\Delta x} + \text{AS\_OFFSET} \right\rfloor \tag{3.3}$$

$$\text{AR\_max}_i = \left\lceil \frac{\text{shape\_max}_x - P_{min_x}}{\Delta x} - \text{AS\_OFFSET} \right\rceil \tag{3.4}$$

The $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ symbols correspond to the familiar floor and ceiling functions. The associated variables are defined as follows;

AR_min	–	Minimum index terms in analysis range
AR_max	–	Maximum index terms in analysis range
P_{min}	–	Minimum coordinate values of bounding box
shape_min	–	Minimum coordinate values of current shape
shape_max	–	Maximum coordinate values of current shape
Δx	–	Mesh discretization size along x axis
AS_OFFSET	–	Offset coordination term

The term 'AS_OFFSET' is arbitrarily selected by the programmer at compile-time, such that the analysis subregion will always correspond to an actual subregion of the overall mesh and that grid points exceedingly close to the current shape are not excluded from being tested for containment by the current shape. After all it is supposed to be the job of the parity count method to determine whether grid points are contained by the current shape. For AS_OFFSET, the value 0.4 has been found to be effective.

Figure 3.2 illustrates how a range of index values is determined, note that the shape minimum coordinate value is in the neighborhood of 'a', and that the shape maximum coordinate value is in the neighborhood of 'b'. The dashed outline indicates which value is assigned for any arbitrary spot on the given axis.

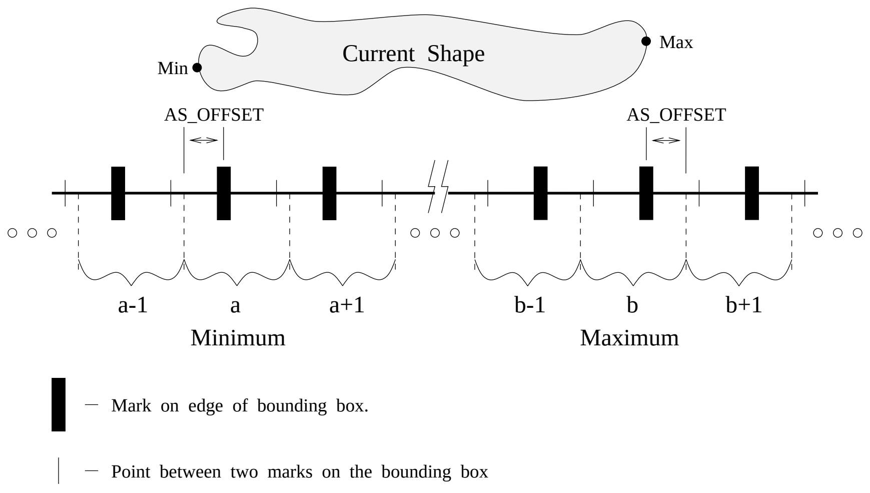

Current Shape
Min
Max
AS_OFFSET
AS_OFFSET
a-1
a
a+1
b-1
b
b+1
Minimum
Maximum
– Mark on edge of bounding box.
– Point between two marks on the bounding box

Figure 3.2: Determining Range of Index Values

Bounding Box for Analysis Subregion

To determine the extreme coordinate values of the bounding box associated with the analysis subregion, we substitute AR_max and AR_min into equation (2.1), to determine the following;

$$\begin{aligned}\text{box\_max}_x &= \text{AR\_max}_i\,\Delta x + P_{min\_x}\\ \text{box\_max}_y &= \text{AR\_max}_j\,\Delta y + P_{min\_y}\\ \text{box\_max}_z &= \text{AR\_max}_k\,\Delta z + P_{min\_z}\end{aligned} \tag{3.5}$$

and further;

$$\begin{aligned}\text{box\_min}_x &= \text{AR\_min}_i\,\Delta x + P_{min\_x}\\ \text{box\_min}_y &= \text{AR\_min}_j\,\Delta y + P_{min\_y}\\ \text{box\_min}_z &= \text{AR\_min}_k\,\Delta z + P_{min\_z}\end{aligned} \tag{3.6}$$

Subdividing the Analysis Subregion

In defining the analysis subregion, we define a smaller mesh that is part of the overall mesh. Inside the analysis subregion, given an ordered triple $[i, j, k]$, each of the index values is valid in the following sequences.

$$\begin{aligned}i &= \text{AR\_min}_i, \text{AR\_min}_i + 1, \ldots, \text{AR\_max}_i\\ j &= \text{AR\_min}_j, \text{AR\_min}_j + 1, \ldots, \text{AR\_max}_j\\ k &= \text{AR\_min}_k, \text{AR\_min}_k + 1, \ldots, \text{AR\_max}_k\end{aligned}$$

By simply taking differences, it is seen that there are N_AS_x, N_AS_y, and N_AS_z boxes along each of the corresponding x, y, and z axes. The terms are defined as;

$$\begin{aligned}\text{N\_AS}_x &= \text{AR\_max}_i - \text{AR\_min}_i\\ \text{N\_AS}_y &= \text{AR\_max}_j - \text{AR\_min}_j\\ \text{N\_AS}_z &= \text{AR\_max}_k - \text{AR\_min}_k\end{aligned}$$

It is preferable to use an alternate index scheme to refer to points in the analysis subregion. Such a scheme simplifies references by considering the analysis subregion as being separate from the overall mesh. In the alternate scheme, an ordered triple $\langle i_a, j_a, k_a\rangle$ of integers is defined by first defining AR_min as being $\langle 0, 0, 0\rangle$. Thus each of the alternate index terms is valid in the following sequences;

$$\begin{aligned}i_a &= 0, 1, 2, \ldots, \text{N\_AS}_x\\ j_a &= 0, 1, 2, \ldots, \text{N\_AS}_y\\ k_a &= 0, 1, 2, \ldots, \text{N\_AS}_z\end{aligned}$$

The angled braces associated with alternate index terms are first meant to imply that the ordered triple is composed of integers rather than physical dimensions, and second that a given reference is different from a reference made with square brackets.

Following a convention similar to what is presented in section 2.2.5 for the overall mesh, boxes in the analysis subregion are referenced by the extreme corner closest to $\langle 0,0,0\rangle$. Note that while the smaller bounding box edge parallel to, say the x axis has $\mathrm{N\_AS}_x$ subdivisions, these subdivisions were formed from $\mathrm{N\_AS}_x+1$ marks on that edge. Thus when referencing boxes in the analysis subregion, the alternate index terms are valid in the following sequences;

$$\begin{aligned} i_a &= 0,1,2,\ldots,\mathrm{N\_AS}_x-1 \\ j_a &= 0,1,2,\ldots,\mathrm{N\_AS}_y-1 \\ k_a &= 0,1,2,\ldots,\mathrm{N\_AS}_z-1 \end{aligned}$$

The correspondence between $\langle i_a, j_a, k_a\rangle$ and $[i,j,k]$ references in the analysis subregion is given by the following formula;

$$\begin{aligned} i &= i_a + \mathrm{AR\_min}_i \\ j &= j_a + \mathrm{AR\_min}_j \\ k &= k_a + \mathrm{AR\_min}_k \end{aligned} \tag{3.7}$$

Note that the correspondence between references is restricted to the analysis subregion, since references in $[i,j,k]$ that are made outside the analysis subregion do not map to points in $\langle i_a, j_a, k_a\rangle$.

Coordinates in the Analysis Subregion

The simplest way to determine the physical coordinates of references made in the analysis subregion as $\langle i_a, j_a, j_a\rangle$ is to write equations analogous to those given in section 2.2.5. In particular, to determine the physical coordinates of grid points in the analysis subregion, the analogue of equation (2.3) is written as;

$$\begin{aligned} x_{gp} &= i_a\,\Delta x + \mathrm{start\_AS}_x \\ y_{gp} &= j_a\,\Delta y + \mathrm{start\_AS}_y \\ z_{gp} &= k_a\,\Delta z + \mathrm{start\_AS}_z \end{aligned}$$

The term 'start_AS' contains the physical coordinates of the grid point associated with box $\langle 0,0,0\rangle$, and appropriately is the analogue of 'start' which was defined by equation (2.2).

$$\begin{aligned} \text{start\_AS}_x &= \text{box\_min}_x + 0.5\,\Delta x \\ \text{start\_AS}_y &= \text{box\_min}_y + 0.5\,\Delta y \\ \text{start\_AS}_z &= \text{box\_min}_z + 0.5\,\Delta z \end{aligned} \tag{3.8}$$

Using the equations from section 2.2.5, the following equations are derived. To determine the physical location (x, y, z) corresponding to $[i, j, k]$, we use the following formula;

$$\begin{aligned} x &= i\,\Delta x + P_{min_x} \\ y &= j\,\Delta y + P_{min_y} \\ z &= k\,\Delta z + P_{min_z} \end{aligned}$$

To determine the physical coordinates corresponding to $\langle i, j, k\rangle$, we substitute the correspondence equations into the previous formula and simplify to find that;

$$\begin{aligned} x &= i_a\,\Delta x + (\text{AR\_min}_i\,\Delta x + P_{min_x}) \\ y &= j_a\,\Delta y + \left(\text{AR\_min}_j\,\Delta y + P_{min_y}\right) \\ z &= k_a\,\Delta z + (\text{AR\_min}_k\,\Delta z + P_{min_z}) \end{aligned}$$

Lastly, it is important to be able to determine the actual physical coordinate values of grid points in the analysis subregion. Since a grid point is located in the center of its corresponding mesh box, the following is determined from the previous expression;

$$\begin{aligned} x_{gp} &= (i_a + 0.5)\,\Delta x + (\text{AR\_min}_i\,\Delta x + P_{min_x}) \\ y_{gp} &= (j_a + 0.5)\,\Delta y + \left(\text{AR\_min}_j\,\Delta y + P_{min_y}\right) \\ z_{gp} &= (k_a + 0.5)\,\Delta z + (\text{AR\_min}_k\,\Delta z + P_{min_z}) \end{aligned}$$

To simplify this expression, the coordinate values of the grid point in box $\langle 0, 0, 0\rangle$ are assigned to a unique variable;

$$\begin{aligned} \text{start\_AS}_x &= (\text{AR\_min}_i\,\Delta x + P_{min_x}) + 0.5\,\Delta x &&= \text{box\_min}_x + 0.5\,\Delta x \\ \text{start\_AS}_y &= \left(\text{AR\_min}_j\,\Delta y + P_{min_y}\right) + 0.5\,\Delta y &&= \text{box\_min}_y + 0.5\,\Delta y \\ \text{start\_AS}_z &= (\text{AR\_min}_k\,\Delta z + P_{min_z}) + 0.5\,\Delta z &&= \text{box\_min}_z + 0.5\,\Delta z \end{aligned}$$

Thus the same equation results for determining the physical coordinates of grid points in the analysis subregion.

$$\begin{aligned} x_{gp} &= i_a\,\Delta x + \text{start\_AS}_x \\ y_{gp} &= j_a\,\Delta y + \text{start\_AS}_y \\ z_{gp} &= k_a\,\Delta z + \text{start\_AS}_z \end{aligned} \tag{3.9}$$

3.2.3 A Linear Array of Counters

It was arbitrarily decided that character variables would be used to store intersection count used in the parity count method. Since the value '-1' is reserved to indicate that a grid point is unresolved, any given search line may have as many as '254' intersections with an individual shape surface before the associated counter overflows. To make the mapping of the counters simple, we mapped the counters associated with the parity count method as a linear array. Mapping the counters in this way offers a certain benefit because of the fact that the overall mesh is stored in the front end as a linear array. Such a situation makes the job of determining the correspondence between entries in the overall mesh, and associated counters in the analysis subregion, particularly simple.

In the parallel version of the mesh generation algorithm there is little need for communications between PEs. The two point resolver represents the rare case when communications between PEs is needed. Since the PE communications requirements of the two point resolver are simple, such a linear mapping of the counters is adequate. By first mapping the counters as a linear array, we will be able to use a single dimensional virtualization, which generally offers better use of PEs and PE memory than higher dimensional virtualizations.

Since the list of counters associated with the parity count method is stored as a linear array, each ordered triple $\langle i_a, j_a, k_a \rangle$ is mapped to an index value by using the following formula;

$$\begin{aligned} \text{index}_c &= i_a + j_a\,\text{N\_AS}_x + k_a\,\text{N\_AS}_x\,\text{N\_AS}_y \\ &= i_a + (j_a + k_a\,\text{N\_AS}_y)\,\text{N\_AS}_x \end{aligned} \tag{3.10}$$

Since this is a one to one mapping, given an index value from the linear array of counters, the following formula will produce the corresponding ordered triple.

$$\begin{aligned} k_a &= \lfloor \text{index}/\,(\text{N\_AS}_x\,\text{N\_AS}_y) \rfloor \\ \text{temp} &= \text{index}\,\%\,\,(\text{N\_AS}_x\,\text{N\_AS}_y) \\ j_a &= \lfloor \text{temp}/\text{N\_AS}_x \rfloor \\ i_a &= \text{temp}\,\%\,\text{N\_AS}_x \end{aligned} \tag{3.11}$$

In the previous expressions, the floor function $\lfloor\cdot\rfloor$ and the divide operator / are used to return the integer part of a quotient, while the modulo operator % returns the integer remainder. The term 'temp' is used simply a temporary dummy variable.

3.2.4 One Dimensional Cut and Stack

The means by which a programmer incorporates the concepts of PE identity and memory layering determines the specific virtualization[1] that will be used to map data to PE memory. As we shall see, the phrase "One Dimensional Cut and Stack," provides a clear description of this specific virtualization. To understand the following discussion, you may need to review the following subjects; The organization of the PE array and how to determine the 'iproc' value assigned to each PE, these topics start on page 13 of section 1.5. In addition the reader should be familiar with the way in which PE memory can be thought of as being layered, this concept is also introduced in section 1.5.2. Lastly, the reader should be familiar with the techniques used to define and describe an analysis subregion, these topics are presented in section 3.2.2.

The first step in initializing this virtualization is to determine how many layers of PE memory will be needed. Given that the linear array to be mapped has a nonzero number of entries equal to 'n_entries', the following formula is used;

$$\text{N}_{\text{layers}} = 1 + \left\lfloor \frac{\text{n\_entries}}{\text{nproc}} \right\rfloor \tag{3.12}$$

As before, the $\lfloor\cdot\rfloor$ symbols represent the floor function, and `nproc` is the number of PEs in the array. Immediately after the required number of layers is determined, the required memory is allocated as a plural array.

[1]If you are not familiar with the concept of *virtualization*, see appendix section C.3.

The next step in initializing the virtualization is to assign each entry in each layer of the allocated plural array, an integer index that corresponds to its location in the linear array. Note that if 'n_entries' is less than nlayers$\cdot$nproc, then the top-most memory layer will only be partially filled. This assignment is made by each PE based on its associated 'iproc' value as well as the index value that corresponds to the associated 'layer' in the plural array. The actual formula used to make this assignment is;

$$\text{index}_c = \text{layer} \cdot \text{nproc} + \text{iproc} \tag{3.13}$$

For the virtualization to be useful, it must be possible to determine which combination of 'iproc' and 'layer' correspond to a given entry 'index$_c$', in the linear array. The following formula is used;

$$\begin{aligned} \text{layer} &= \lfloor \text{index}_c / \text{nproc} \rfloor \\ \text{iproc} &= \text{index}_c \,\%\, \text{nproc} \end{aligned} \tag{3.14}$$

As before, the $\lfloor \cdot \rfloor$ and / symbols represent the floor function and the divide operator, together they return the integer part of a quotient. The % symbol returns the integer remainder of that quotient.

To illustrate the virtualization, consider the following hypothetical example. We are given nproc = 9 and n_entries = 13, thus we find that n_layers = 2. Figure 3.3 shows how index$_c$ is assigned to each entry in our hypothetical example. Note that five entries in 'layer 1' are unused.

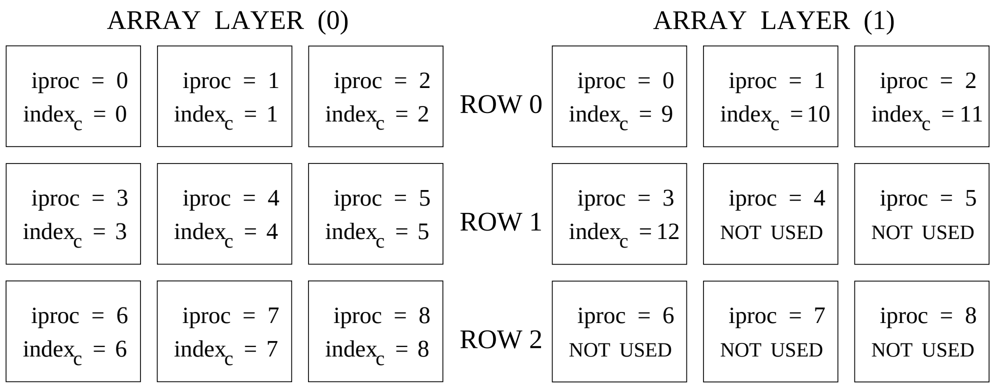


Figure 3.3: Hypothetical Example of Virtualization

If each layer in figure 3.3 is first "cut" into rows, and then the each of the rows is "stacked" in sequence, end to end, then the construction will form a linear array. The result of "cutting" and "stacking" the layers of our hypothetical example is shown in figure 3.4. At this juncture, the phrase "one dimensional cut and stack," should make sense to the reader.

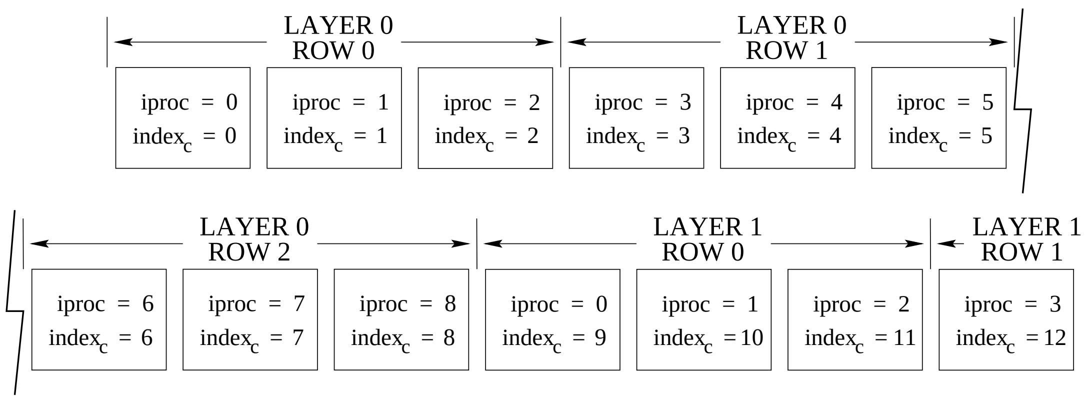


Figure 3.4: Virtualization Presented as Linear Array

To summarize the virtualization, the parallel mesh generation code maps data in a linear array form that is best described as *One-Dimensional Cut-and-Stack Data Mapping*[2]. Note that even though data being processed is associated with three dimensional space, data is mapped to the two dimensional processor array memory in a way that is characterized as one dimensional. While such a mapping improves the use of PEs and PE memory, it also complicates use of the X-Net. This type of mapping is adequate for mesh generation for one reason in particular. In the mesh generation algorithm there is very little need for communications between PEs. In the rare case where such communications is needed, it will only be needed along one dimension of the mesh. Thus the mesh generation code trades off the ability to easily perform communications between PEs in favor of improved PE use.

3.2.5 Mapping Summary

In examining the previous sections, the reader will note that the mapping equations are all related and are meant to be used together. For example, to determine the physical coordinates of a grid point, equation (3.9) provides the answer in terms of a $\langle i_a, j_a, k_a \rangle$ reference. Alternately equation (2.3) on page 61 also provides the physical coordinates but in terms of a $[i, j, k]$ reference. Equation (3.7) provides the correspondence between references in $[i, j, k]$ and references in $\langle i_a, j_a, k_a \rangle$. Equation (3.1) is used to find a $[i, j, k]$ reference in the overall mesh. Equation (3.10) is used to find reference $\langle i_a, j_a, k_a \rangle$ in the linear array of counters. And lastly equation (3.14) is used to find where an entry in the linear array of counters is actually stored in PE memory.

[2]See page 2-9 of MasPar MPDDL Reference Manual[33]

Note that the primary tradeoff of the mapping scheme was to decide that coordinates associated with each mesh box would not be stored in memory. What this means is that for each iteration of the facet level flowchart, index values must be reverse mapped so that the coordinates associated with grid points can be determined. Thus the cost of the tradeoff is a slight loss in performance. The benefit from this tradeoff is that memory is better used for storing the mesh, this allows for exceedingly large meshes to be modeled by the mesh generator.

Figure 3.5 provides a visual summary of the mapping scheme. The overall mesh is mapped to a linear array of material identifiers. The analysis subregion inside the overall mesh is mapped to an array of counters used to perform the parity count method. There is a unique correspondence between the array of material identifiers which are stored in the front end, and the array of counters which is stored in the DPU. This correspondence is important as once the containment of grid points is determined, the Copy Out Task must be able to read the resulting bitmap, and based on the precedence method, insert the shape's material identifier into the array of material identifiers used to represent the overall mesh. In the parallel version of the mesh generator, the linear array of counters is stored in PE memory according to the one dimensional cut and stack virtualization.

3.3 Allocating Memory

For both the parallel and serial versions of the mesh generator, adequate memory must be allocated for two general purposes. First, memory must be allocated for storing the overall mesh. Second, a workspace must be available for the analysis of each shape. As we shall see, setting up and maintaining such a workspace requires that memory be allocated in different ways. Following a linear model of memory, the following discussion considers only large blocks of memory that are allocated.

For the parallel version of the mesh generator, the arrangement of having the overall mesh stored in the front end and having counters associated with the parity count method stored in the DPU has at least two significant outcomes: First, better use can be made of front end memory. Because the counters associated with the parity count method are not stored in the front end, more memory should be available for the overall mesh. While the front end does have the advantage of efficient demand paging, by minimizing its use it is possible to store meshes that are ever larger.

In addition, because the overall mesh is not stored in the DPU, use of memory

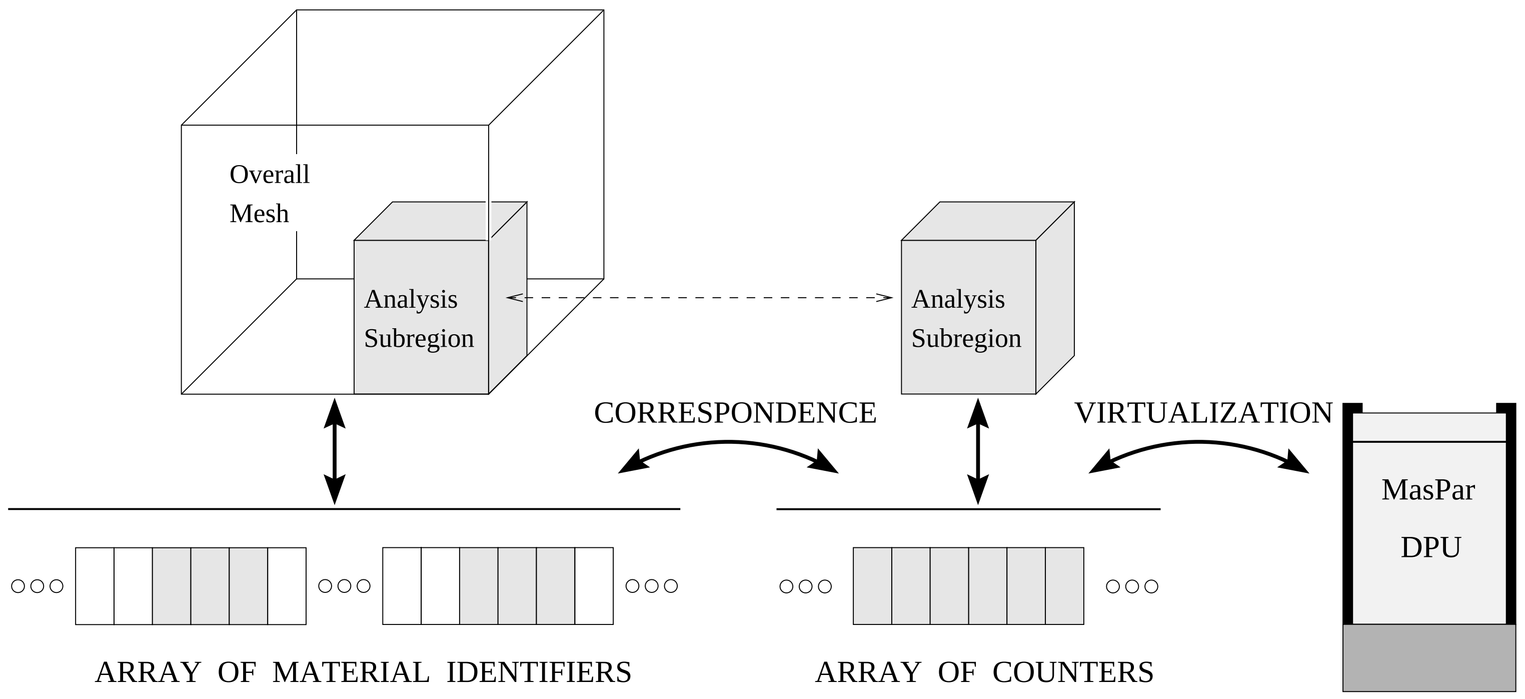


Figure 3.5: Summary of Mapping for Mesh Generator

in the DPU can be very simple. Since shapes are modeled one at a time by using the parity count method, memory requirements in the DPU have been simplified to the point where it is only necessary to ensure that the largest shape to be modeled can "fit into" DPU memory.

Even with differences in memory use between the parallel and serial versions of the mesh generator, the decision to store the overall mesh in the front end guarantees that data structures in both versions will be similar. In developing the parallel code, this similarity provided a significant savings in time.

For the following, please refer to figures 3.6 and 3.7. Figure 3.6 illustrates how memory is allocated for the serial version of the mesh generator. Figure 3.7 is an illustration of how memory is allocated in the parallel version. In both figures, the '`char`' variable type, the 'unsigned int' variable type, and the use of '`*`' to indicate pointers should be familiar to the 'C' programmer. The terms 'point' and 'point_2d' refer to defined data structure types that store the three or two dimensional coordinates of points, respectively, along with an associated point identification number.

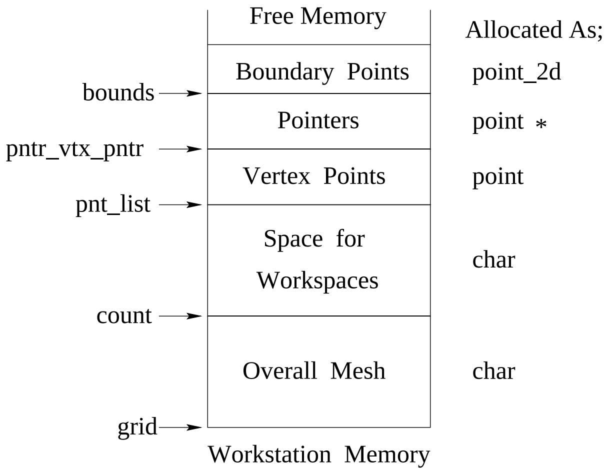


Figure 3.6: Memory Allocation in Serial Version

For both versions of the mesh generator, the first block of memory allocated is assigned to a character variable pointer named `grid` and is used to store the overall mesh. For the serial version, the next block of memory allocated is assigned to a character variable pointer named `count` and is used by the current workspace to apply the parity count method. For the parallel version the next block of memory is assigned to an 'unsigned int' pointer, also named `count` but is used as a buffer for receiving data from the DPU. Note that for both the serial and parallel versions, the first two blocks of memory are allocated by the Initialize Analysis Task. These

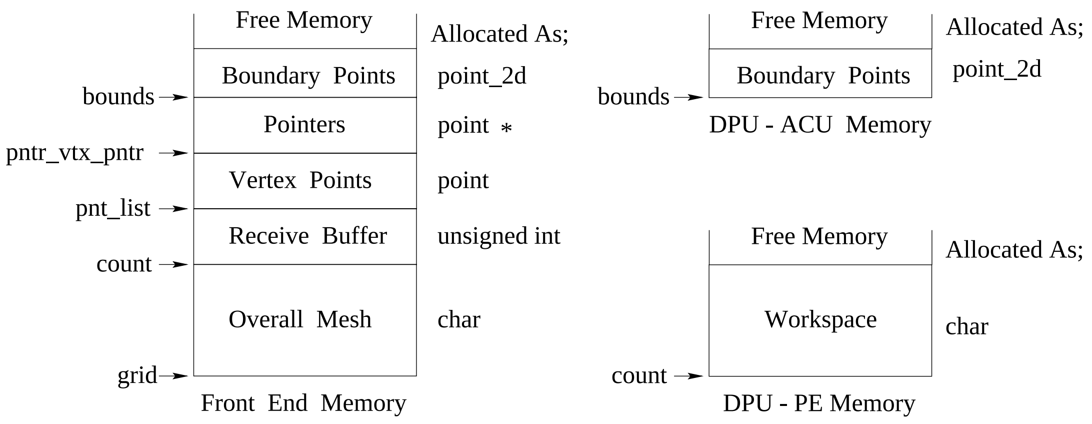


Figure 3.7: Memory Allocation in Parallel Version

first two blocks are not returned to free memory until the program terminates.

For both the parallel and serial versions of the mesh generator, the third, fourth and fifth blocks of memory are allocated in the front end. The third block of memory allocated is assigned to a pointer named `pnt_list` and is used to store a list of vertex points for each shape. The fourth block of memory is assigned to a pointer named `pntr_vtx_pntr` and is used to store a list of pointers to vertex points. Such a list of pointers is used to indirectly define the current facet. The fifth block of memory is assigned to a pointer named `bounds` and is used to store the two dimensional boundary points that define the current facet in facet space.

Whenever a new shape is initialized, if more memory is needed for storing vertex points, the last three blocks of memory allocated in the front end are first returned to free memory, then the memory needed to store the vertex points is allocated as one complete block. The list of vertex points must be allocated as one complete block, as a binary search is used to identify individual vertex points. Once the number of vertex points associated with a facet is known, a decision can be made as to whether more memory should be reallocated for storing pointers and boundary points.

In the parallel version, memory must also be allocated in the DPU. For each shape that is initialized, the function `simd_init` allocates the required amount of PE memory to store counters for applying the parity count method. As stated earlier, memory for the overall mesh is not allocated in the DPU, only the required memory is allocated to model the current shape. The memory allocated for counters used to implement the parity count method, is assigned as an array of plural character variables to a pointer named `count`.

For each facet processed, the function `simd_solve_a_facet` ensures that adequate memory is allocated in the ACU for receiving the boundary points used to define the current facet. The memory used to receive boundary points is assigned to a pointer named `bounds`. Note that since the ACU and PE memories are disjoint, there is no possibility of interaction. For example, the memory allocated to `count` may grow as much as it needs to, but there never will be a possibility of overwriting the memory assigned to `bounds`.

3.4 Implementation Outline

This section presents an outline of the implementation of the parallel mesh generator. Note that this section does not discuss details regarding the algorithm, the complete algorithm is actually presented in chapter 2. Nor does this section dive into great detail regarding the implementation, the actual code itself serves this purpose. The purpose of this section is to provide a rough sketch that will assist readers who may wish to become more familiar with the serial or parallel implementations, in a sense it provides a guided tour of the mesh generator codes. Section 3.4.1 adds to this section as it provides an outline of how the software modules are organized.

3.4.1 Organization of Software Modules

In developing the parallel and serial versions of the mesh generator, software modules were organized into groups. Figure 3.8 illustrates these module groups and shows how the module groups are organized to produce two versions of the mesh generator. Modules used by both the parallel and serial versions are referred to as *common modules.* The common modules perform the majority of the initialization tasks as well as the Facet Server Task, the Store Results Task, and a collection of the vector operations. Note that in the parallel and serial versions, the common modules are essentially identical, in fact in the case of the parallel version, the common modules actually execute in the front end as serial code.

Since the common modules are meant to be used by both the serial version and the parallel version, some additional serial code is needed to manage transactions with the DPU, thats the job of the interface module. The interface module is

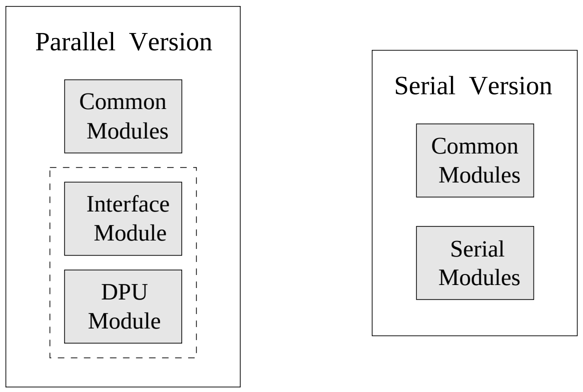


Figure 3.8: Software Modules in Generator Versions

essentially a kind of liaison to the DPU. The DPU module executes only in the DPU and performs the parallel versions of the intersection finder, facet solver, Check Insideness Task, Resolve Singularities Task, and half of the Copy Out Mesh Task.

Lastly, modules used only by the serial version are referred to as the serial modules, which perform the serial versions of the intersection finder, facet solver, Check Insideness Task, Resolve Singularity Task, and the entire Copy Out Mesh Task.

3.4.2 Mesh Initialization

The term *initialized* is defined in this context to be that point in time when facets are actively being read. In comparison, the term *invoked* is used to indicate that execution of the mesh generator has begun. The reader will recall from section 2.4.1 that initialization is divided into two tasks. The Initialize Analysis Task is performed first. The Initialize Shape Task is performed once for each shape.

Initialize Analysis

The Initialize Analysis Task is started the moment that the mesh generator is invoked and may or may not include an interactive component, as it is possible to start the mesh generator by use of an ‘autostart file’. An autostart file contains all the required information that a user would normally enter manually.

When invoking either version of the mesh generator, the user may optionally include a file name, if such a file name does not contain an extension, the extension '`.aum`' will automatically be appended. Note that the '`.aum`' extension is usually associated with autostart files. If however a user does not provide a file name when invoking the mesh generator, the user will be prompted for a file name. In this second case, if the file name given does not have an extension, the extension '`.sld`' will automatically be appended. Note that the '`.sld`' extension is usually associated with solid files. In both cases if an extension is given as part of a filename, an extension will not be appended.

After a file name has been acquired, and the associated file has been opened, the first few characters are examined to determine the file type. If the first four characters match with the word '`auto`', then the autostart file type is assumed, in which case the mesh generator will immediately attempt to load the rest of the file. One item always specified in an autostart file is a solid file name, thus after loading the autostart file, the mesh generator will open the specified solid file,

Whenever a file is opened and the first five characters match the word '`solid`', the solid file type is assumed. If an autostart file was not loaded, then more data will be needed. The user is asked a series of questions, starting with the requested number of mesh box divisions along each coordinate axis. Next the user is asked which output files should be produced. Output file types are discussed in the user's manual, see section F.3. Lastly the user is asked if an autostart file should be written. In writing an autostart file, the file contains a summary of all the user responses along with the specified solid file name.

Lastly note, if the first few characters of a file do not match either '`auto`' or '`solid`', then the mesh generator will immediately alert the user that the file type is unrecognizable and will then abort execution.

After all necessary information has been gathered either directly from the user or from the autostart file, and the solid file has been successfully opened and identified as a solid file, the next job is to load the solid file header. The *solid file header* is four lines of text that contains the unit conversion factor, dimensions of the overall bounding box and the number of shapes in the file. It is important that the solid file contain at least the stated number of shapes or the mesh generator will terminate later in an abnormal fashion.

After the solid file header has been loaded, a function called '`set_up_grid`' is called to define initial constants and allocate two blocks of memory. The first block of memory is used to store the overall mesh. For the serial version the second block of memory is used to store counters, in the parallel version the second block is used as a receive buffer. For more details about the allocation of these two blocks of memory, see section 3.3. The constants defined by '`set_up_grid`' include the search origin, the constant for detecting nearly orthogonal vectors, and the coordinates of the grid point in the first mesh box. In the parallel version, '`set_up_grid`' sends the constants it defined to the DPU by calling '`simd_pag`'. The function '`simd_pag`' returns the number of groups of 1024 processors that are present in the PE array. After '`set_up_grid`' returns control to 'main', the time of day is stored and is used later to determine the system execution time. Initialization is now partially complete. The next section describes the remaining initialization that needs to be done.

Initialize Shape

The section of a solid file that is used to store a shape is referred to as a *shape description*, and has three parts: The shape header, a list of vertex points, and a list of facets. The shape header contains a unique integer shape label, the material identifier, the number of vertex points and the number of facets used to describe the shape. The list of vertex points defines for each vertex point, a correspondence between an integer used to identify the vertex point and a set of three dimensional coordinates. Note that for each shape the values given as vertex identification numbers usually start at zero or one. Since a binary search is used to locate individual vertex points, the only restrictions made regarding the sequence of vertex point identifiers is that each value be unique and that the sequence be ascending. Lastly each shape description has a section containing facets. Each facet is defined by referencing a list of vertex points by their identification numbers.

To initialize a shape, the shape header is first loaded. If space is needed to store vertex points, it is allocated. See section 3.3 for more details regarding the memory allocation. Next, a function called '`gather_points`' is called. In this function a loop loads all of the vertex points associated with the shape into memory. While the vertex points are being loaded into memory, they are compared to yield the extreme coordinate values. These extreme coordinate values are stored in two data structures, '`shape_min`', and '`shape_max`'.

The function '`prepare_scratch`' is next called to prepare the workspace that will be used to apply the parity count method. This function starts by defining the analysis subregion. Section 3.2.2 describes exactly how an analysis subregion is defined. For the serial version, since space has already been allocated for use as counters, it is only necessary to fill the memory locations associated with the analysis subregion with zeros.

For the parallel version, since the shape will be modeled in the MasPar DPU and memory has not yet been allocated there, some additional work will be needed. In the front end, after the analysis subregion is defined, a bounding box containing the associated mesh subregion is defined. Since the workspace bounding box must "fit in" the overall mesh, the coordinates of the extreme corners of the new bounding box are assigned to the data structures '`obj_box_min`' and '`obj_box_max`'. The number of mesh box subdivisions along each edge of the workspace bounding box is assigned to the data structure named '`obj_boxes`', the receive buffer named '`count`' is cleared by filling it with zeros. Lastly, '`simd_init`' is called in the DPU to perform the memory allocation and store the data structures associated with the analysis subregion.

After entering a '`for`' loop control structure, the next thing encountered is a `scanf` that will read in part of the description of a facet. Initialization of the mesh generator is now complete.

Model the shape

For each facet that is examined, memory may need to be allocated for data structures needed to store the facets. Shapes are examined one at a time. Until it is time to start writing out data files, the program follows the algorithm specified in chapter 2. Note that while a unit conversion factor is included in the solid file header, it is not applied to the data when the shapes are actually being modeled. Only the actual output file data is scaled by the unit conversion factor. Thus rather than scaling the coordinate values of every vertex point, only the coordinates associated with the overall bounding box are scaled. It is felt that this approach provides a savings in performance as well as a possible improvement in the accuracy of a model.

Since the ownership of each mesh box by a single shape can be expressed by the value of a single bit value, it was possible to compress the description of groups of 32 mesh boxes into an array of long integer variables. The generation of a bit map and the compression of the bit map into an array of long integers was chosen as a means to efficiently move the resultant mesh out of the DPU and into the front end. The precedence method[3] is implemented in the front end to write the mesh associated with the current shape into the overall mesh.

[3]For more information regarding the precedence method see section 2.2.3

3.5 Characterizing Performance

In the following sections we will consider performance of the mesh generator code. We will start by presenting the shape referred to in this document as being the *small sphere*. The type of model represented by the small sphere is unique as it has a relatively large number of facets where each facet represents a small part of the surface area of the overall shape. In addition the facet planes are all oriented relative to each other at unusual angles. Because of these properties the small sphere model is considered somewhat rigorous and computationally intensive.

The small sphere model is comprised of 242 vertex points, which are used to define a system of 256 facets. The facets are constructed of either three or four vertex points and on average each facet is defined by 3.875 vertex points. One property to note is that while the analysis subregion contains the entire bounding box, the sphere does not fill the entire space.

Figure 3.9 is an illustration of a mesh representation of the small sphere. The illustration was produced by Patran, a commercially available software package. Note that the sphere was was defined to be constructed of a solid material and all the surrounding material was removed. The overall mesh was defined to be of size 30x30x30.

In the sections that follow, we characterize the performance of both the serial and the parallel mesh generator programs. CPU time refers to user time spent performing serial code and DPU time refers to time spent performing parallel code. Both CPU time and DPU time were introduced in section 1.7.1.

In gathering execution time data from the serial code, two cases were always considered, non-optimized code as well as fully optimized code. The RISC `cc` compiler produced non-optimized serial code when the `-g` flag was specified, conversely fully optimized code was produced when the `-O3` flag was specified. Note that the parallel code was not optimized. While the MasPar compiler does have an option to produce optimized code, it does not always work and has bugs, some of which are known and documented. In our situation the optimized parallel code that the MasPar compiler produced was non-functional, thus we were limited to non-optimized parallel code.

In gathering execution time data from the parallel version, two different machines were used to perform the parallel code. The first machine was the MasPar system owned by Worcester Polytechnic Institute. This machine is an MP1101, known as node 'goofy'. In writing the parallel code, special care was taken to

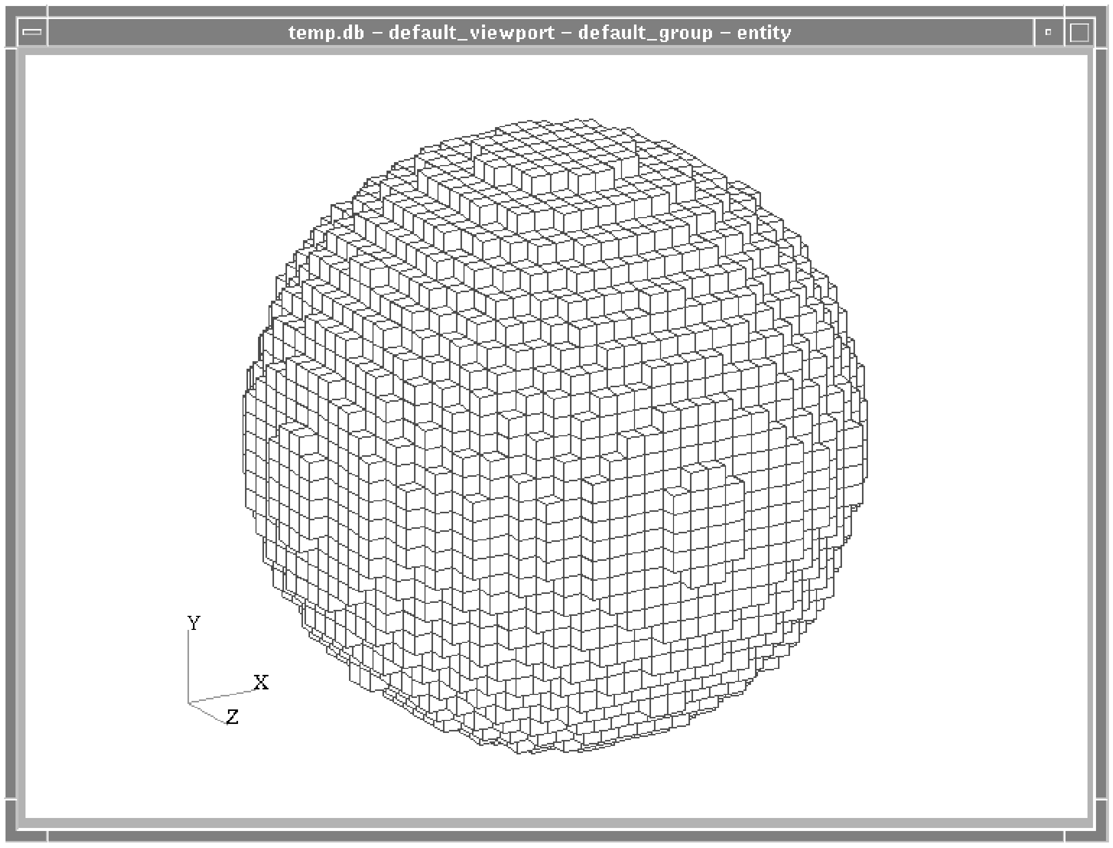


Figure 3.9: 30x30x30 Mesh of the Small Sphere

ensure that no assumptions were made regarding the size of the PE array. Upon startup, the parallel code self adjusts the mapping to the size of the PE array. This adjustment at startup is provided to ensure that the parallel code will work as efficiently as possible in any MasPar system configuration. We made good use of this software feature as MasPar graciously provided us with access to a larger MasPar system, an MP 1208 with 8192 PEs. This MasPar system has eight times as many PEs as the system owned by WPI. Section A provides a complete description of the configurations of these two machines.

3.6 Parallel Time Complexity Analysis

In section 2.7.2 a complexity analysis modeled the asymptotic behavior of the serial version of the mesh generator. The following presents the time complexity equations in the context of the parallel implementation. The reader should note that the big-oh equations developed for the serial code are also valid for modeling the asymptotic behavior of the parallel version of the mesh generator. This is not unusual, as the time complexity equations are supposed to be characteristic of the underlying algorithm.

According to section 2.8, from an algorithmic point of view, the primary differences between the parallel and serial versions of the mesh generator are that the Pick Next Grid Point, Intersection Finder, Facet Solver and Check Insideness tasks are implemented in parallel rather than in serial form. In addition, the Copy Out Task is implemented partially in parallel and partially in serial code. We will consider each of these tasks in turn. A copy of figure 2.26 is presented as figure 3.10, as a convenience to the reader.

The Pick Next Grid Point Task amounts to a for-loop control structure, along with some calculations. As introduced in section 3.2.4, the array of grid point counters is distributed across many memory layers. Appropriately, the memory associated with grid point counters is indexed by a singular memory layer index value.

Before the Intersection Finder Task can be performed, for each memory layer, each PE must determine its location in the overall mesh by using equation (3.13) and 3.11. With the index terms $\langle i_a, j_a, k_a \rangle$, the corresponding coordinates of a grid points are found by using equation (3.9). Note that the index terms and coordinate values are only stored temporarily and must be recalculated for each iteration. While the cost of such a tradeoff is increased computation, the benefit

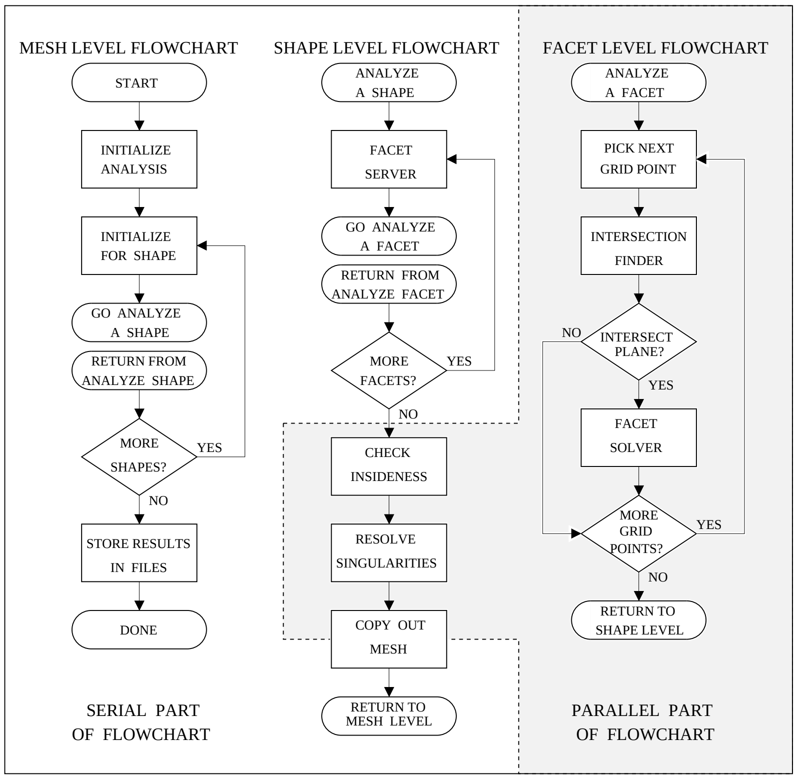


Figure 3.10: Division of Mesh Generation Algorithm Tasks

is that better use is made of parallel memory, allowing us to model larger meshes than would otherwise be possible. The serial version of the mesh generator also makes use of this tradeoff.

For the serial version of the mesh generator, for each facet processed, N_m grid points[4] must be examined, one at a time. For the parallel version of the mesh generator, since at most `nproc` grid points are examined simultaneously, there will be significantly fewer iterations of the facet level flowchart. With the idea of the analysis subregion in mind, each facet processed will require this many iterations in the facet level flowchart;

$$\text{serial iterations} = O\left(N_m\right)$$

$$\text{parallel iterations} = O\left(\, \lceil N_m/\texttt{nproc} \rceil \,\right)$$

In the above equations, N_m is the number of grid points in the overall mesh and `nproc` is the number of PEs, and $\lceil \cdot \rceil$ refers to the ceiling function. For the

[4]For background of the related variables, see section 2.7.1

purposes of a time complexity analysis, we will make the following approximation;

$$\lceil N_m/\texttt{nproc} \rceil \approx \frac{1}{\texttt{nproc}} N_m$$

Thus we can state that;

$$\text{parallel iterations} = O\left(N_m\right) \tag{3.15}$$

To review, while we know that for each facet processed, the number of iterations in the facet level flow chart is much smaller for the parallel version than the serial version, both grow asymptotically in a first order fashion.

The code associated with the Initialize Analysis Task is performed once, and as with the serial implementation, has a time complexity of $O(1)$. The identical code is used in the parallel and serial versions to perform the Initialize Shape Task. Likewise, the identical code is used to perform the Facet Server Task. In this section however, we will follow the assumption made in section 2.7.3 that led to equation (2.29). The assumption is that the average number of vertex points per facet $\tilde{N}_{vf}$ is a relatively small number. Based on this assumption, we restate the overall contribution from the facet server to be $O\left(N_s\,\tilde{N}_{fs}\right)$.

As with the serial version of the intersection finder, each call to the parallel intersection finder is $O(1)$. Given N_s shapes where each shape has an average number of facets equal to $\tilde{N}_{fs}$, the overall contribution from the parallel intersection finder is the same as the serial version.

$$\text{CX}_{XF} = O\left(N_m\,N_s\,\tilde{N}_{fs}\right)$$

As with the serial version of the facet solver, each call to the parallel facet solver is $O\left(N_{v(i,j)}\right)$. Following the same argument given in section 2.7.3, we will assume that the average number of vertex points per facet $\tilde{N}_{vf}$ is a relatively small number and thus is incorporated into the assumed constant, as in equation (2.29). Thus the overall contribution from the parallel facet solver is given as;

$$\text{CX}_{RX} = O\left(N_m\,N_s\,\tilde{N}_{fs}\right)$$

Since the parallel versions of the intersection finder and facet solver were written into the same function, it is difficult to separate their timing values in a timing profile. For this reason we define the combined term to contain the contributions from the facet solver and intersection finder.

$$\text{CX}_{PC} = O\left(N_m\,N_s\,\tilde{N}_{fs}\right)$$

As with the serial code, the contribution from the Pick Next Grid Point is incorporated into the intersection finder and is referred to here as the *parallel core* of the mesh generator. In this case CX$_{PC}$ contains contributions from Pick Next Grid Point, Intersection Finder and Facet Solver tasks.

Lastly, the Check Insideness Task and Copy Out Task have overall contributions of $O\,(\,N_m\,N_s\,)$. Whenever we report execution time profile data for the parallel Copy Out Task, the times used to execute the parallel and serial parts of the program are reported separately and combined.

Table 3.2 provides a summary of the contributions from each task. Of these tasks, the Initialize Analysis, Initialize Shape, and Facet Server tasks were implemented in serial form. For more information, the reader is referred back to section 2.7.2.

Table 3.2: **Revised Time Complexity by Task**

Task Name	Term	=	Expression
Initialize Analysis	CX$_{IA}$	=	$O\,(\,1\,)$
Initialize Shape	CX$_{IS}$	=	$O\left(\,N_s\,\tilde{N}_{vs}\right)$
Facet Server	CX$_{FS}$	=	$O\left(\,N_s\,\tilde{N}_{fs}\,\right)$
Parallel Core	CX$_{PC}$	=	$O\left(\,N_m\,N_s\,\tilde{N}_{fs}\right)$
Check Insideness	CX$_{CI}$	=	$O\,(\,N_m\,N_s\,)$
Copy Out	CX$_{CO}$	=	$O\,(\,N_m\,N_s\,)$

After examining table 3.2, a few points can be considered. First, since the parallel core has the most number of terms in its argument, it has the potential of being the largest contribution to the overall execution time. This is essentially the same conclusion that was made back in section 2.7.3.

To get a better idea of the size of the contributions from each task, a 22x22x22 mesh representation was generated, as was done back in section 2.7.3, but this time execution was also performed on the MP 1101 and MP 1208. The MasPar tools were used to determine the timings associated with each task, and are listed in table 3.3. Note that on the parallel machines the check insideness task and parallel core are much faster than the serial code. The parallel version of the copy out task is slower than the serial code as the parallel version has the additional overhead of data transfer back from the DPU to the front end. Note that the MP 1208 we used was able to perform the transfer faster than the MP 1101, this may be evidence of the fact that the MP 1208 had a high performance parallel VME connection. The MP 1101 only had a serial VME connection.

The last row of table 3.3 indicates the percent of the total execution time that is represented by the parallel core. Note that even though the total execution times vary drastically, the contribution represents more than 90% of the total execution time. Since a 22x22x22 mesh is considered to be small, with knowledge of the time complexity it should be clear that as the mesh size increases, the parallel contribution will remain large, even for large MasPar configurations.

Table 3.3: **Execution Time(mSec) by Task**

Task	Comp.		Serial		
Name	Term	=	Code	MP 1101	MP 1208
Initialize Analysis	CX_{IA}	=	1	1	1
Initialize Shape	CX_{IS}	=	58	54	50
Facet Server	CX_{FS}	=	150	133	147
Parallel Core	CX_{PC}	=	175028	15756	3195
Check Insideness	CX_{CI}	=	29	6	3
Copy Out	CX_{CO}	=	30	74	43
	Task Totals	=	175296	16024	3438
	Percent Parallel Core	=	99.85%	98.33%	92.93%

During the development of the mesh generator we considered the possibility of running the front end and DPU in an asynchronous fashion as a means of improving the overall efficiency of the parallel code. We reasoned that if the front end and DPU each spent a significant amount of time waiting for the other to complete tasks, a performance gain could be realized by allowing both machines to work simultaneously. This is probably the primary reason why the facet server was put in the front end, rather than in the DPU. Serving facets in the front end provides a secondary means of reducing the impact of such serial operations on the parallel algorithm. This scenario represents an opportunity, since the front end unit and the DPU can be run, *asynchronously.* After the first facet of a shape has been delivered to the DPU, asynchronous operation allows the front end to immediately start preparing the next facet. Assuming that the DPU takes more time to consume a facet than the front end needs to prepare one, a newly prepared facet will always be available for the DPU to immediately process. We suspected that the net result would be of partially hiding the serial operation in the front end from the parallel execution in the DPU, thus improving the performance of the algorithm.

The data in table 3.3 appears to indicate that in this case, if the MasPar system was run in an asynchronous fashion, there might be an improvement in performance, but the improvement would be small. At the time that this data was collected, it was decided that the effort to convert the code from its current synchronous operation to that of asynchronous operation was not worth the effort. For future work it would be worthwhile to consider the circumstances where such asynchronous operation would provide a significant savings in performance, and to assess those savings.

Parallel Time Complexity Summary

The goal of the time complexity analysis is to show that an approximately linear relationship exists between the execution time and the terms used to define the problem size. A few of the terms used to define the problem size are the number of mesh boxes that will be in the resultant mesh, and the number of facets in a shape, see section 2.7.1 for a listing of the terms used to define the problem size. The knowledge of such a relationship is important as it provides a means to compare the algorithm with other possible algorithms. The dependence of the execution time on the problem size is particularly useful in this respect as a figure of merit. The timing results given in section 3.7, along with curves fitted to the data serves to verify the expected linear dependence of the execution time on the number of boxes in a mesh.

3.7 Timing Results

Figure 3.11 provides a graph of the execution time for each case that was considered. As described earlier, we considered optimized and non-optimized versions of the serial code. Also parallel code was executed on two different MasPar systems. The mesh sizes that were examined were cubic, thus to determine the number of mesh boxes along each axis simply take the cube root of the number of mesh boxes for a given mesh.

By visual inspection of the data presented in figure 3.11 it can be inferred that there is an linear relationship between the execution time and the number of mesh

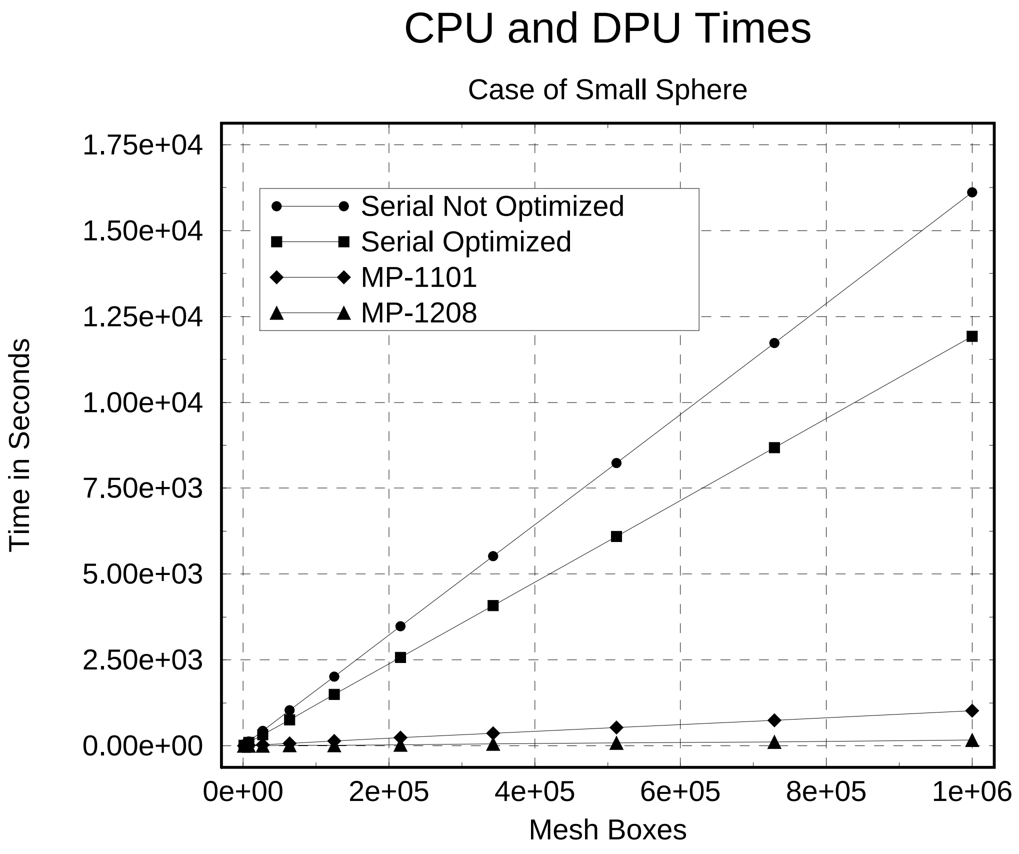


Figure 3.11: CPU and DPU Times for Small Sphere

boxes in a resultant model. To better quantify this relationship, lines were fit to the data. To draw the graphs, a program called xmgr was used. To fit the lines to the given data, xmgr provides a regression analysis tool. Table 3.4 is a summary of the of the curve fit data that xmgr produced, note that xmgr refers to goodness of fit as being the *correlation coefficient*, which in this case appears to be the correlation between the data sets represented by the execution time and the number of boxes in the resultant mesh.

Table 3.4: **Results Fitted for Small Sphere**

Case Description	Slope	Intercept	Correlation Coef.
Serial - Not Optimized	1.610523×10^{-2}	-1.291225	0.9999990
Serial - Optimized	1.192160×10^{-2}	-1.026041	0.9999998
MP 1101 - Not Optimized	1.013866×10^{-3}	10.668460	0.9985120
MP 1208 - Not Optimized	1.638132×10^{-4}	1.525807	0.9999717

In table 3.4, the correlation coefficient is used to describe the “goodness of fit” of each linear fit to actual data. By noting the closeness of the correlation coefficients to unity, it can be said in general that there is an approximately a linear relationship between the execution time and the number of boxes in the mesh.

3.7.1 Speed Up Characteristics

The concept of speed-up was introduced in general terms, back in section 1.7.4. Basically, speed-up provides a means of characterizing how much faster a program becomes when the program is enhanced in some way. Starting with non-optimized serial code, the first enhancement was to use the optimized serial code instead. In both instances, the front end workstation of the MasPar system owned by WPI served as the platform for the serial code. The second enhancement made was to perform certain tasks in parallel form on the MasPar parallel array processor, rather than in serial form on the front end. Section 2.8 provides a detailed explanation of which tasks were performed in parallel form. The MasPar MP 1101 owned by WPI served as the platform for evaluating this second enhancement. The third enhancement was to perform the parallel code on a larger MasPar system, a MP 1208 system (node ‘maspar’) at MasPar corporation. The configuration of both MasPar systems is summarized in section 1.5.5. As before, the parallel code was not optimized by the parallel compiler. In the following we examine the speed-up that resulted from each of these enhancements. Note that the results used in the following speed-up characterization correspond to the generation of a mesh model of the small sphere.

Serial Versions Compared

Figure 3.12 provides an illustration of the speed-up that resulted in selecting fully optimized serial code The speed-up figure appears to be close to 1.35, corresponding to a 35% increase in the performance.

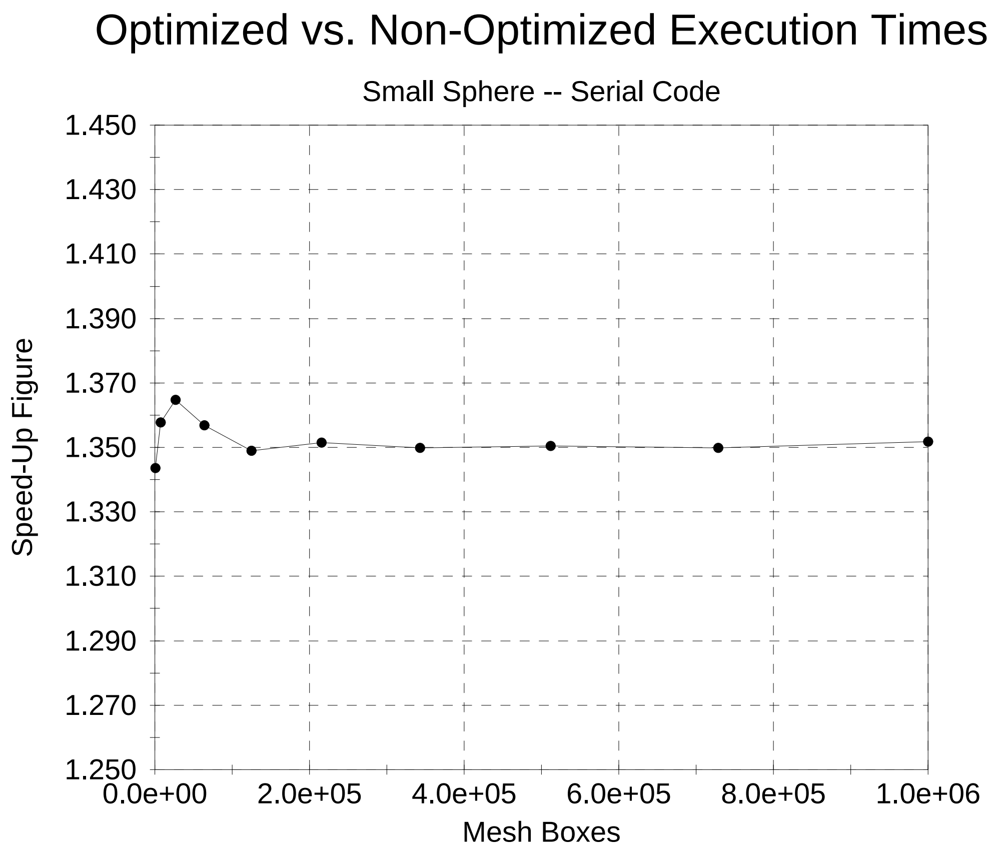


Figure 3.12: Comparing Serial Execution Times for Small Sphere

Parallel Code on the MP 1101

Figure 3.13 provides an illustration of a comparison of the parallel code performed by the MP 1101 system owned by WPI (Node 'goofy') and serial code performed only on the front end. For very small meshes (1000 mesh boxes) the speed-up is the smallest; Note that as the problem size increases the speed-up increases as well. This observation can be interpreted as the fact that a parallel hardware platform requires a large problem to make the best use of the hardware. This topic is investigated in greater depth in the discussion of processor element use. Note that as the problem size increases, an asymptotic limiting behavior appears to be present. Compared to the non-optimized code, the speed-up figure appears to be asymptotically approaching the value 16. Compared to the optimized code, the speed-up figure appears to be asymptotically approaching a value just under 12.

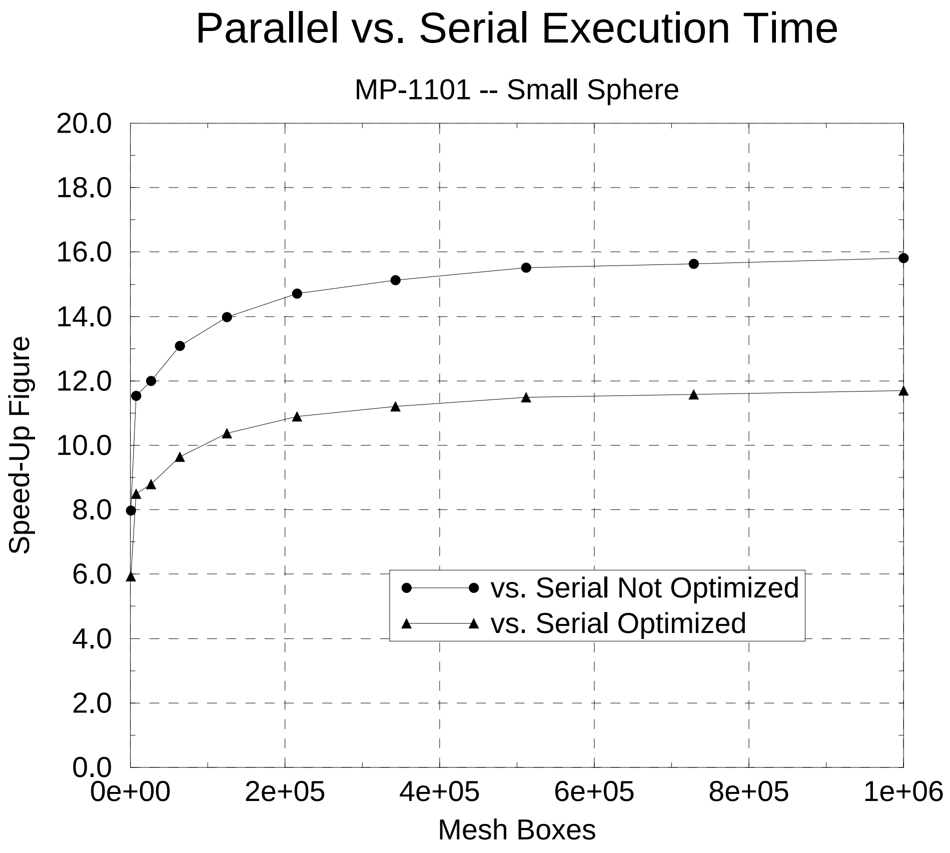


Figure 3.13: Speed-Up, MP 1101 vs. Serial

Parallel Code on a MP 1208

Figure 3.14 provides an illustration of the parallel code performed on a MP 1208 versus optimized as well as non-optimized serial code. As before, for very small meshes the speed-up is the smallest, and in general appears to improve as the problem size becomes larger. In comparison to the non-optimized code, as the problem size becomes ever larger the speed-up appears to asymptotically approach a value near 98. In comparison to the optimized code, speed-up appears to asymptotically approach a value just above 72.

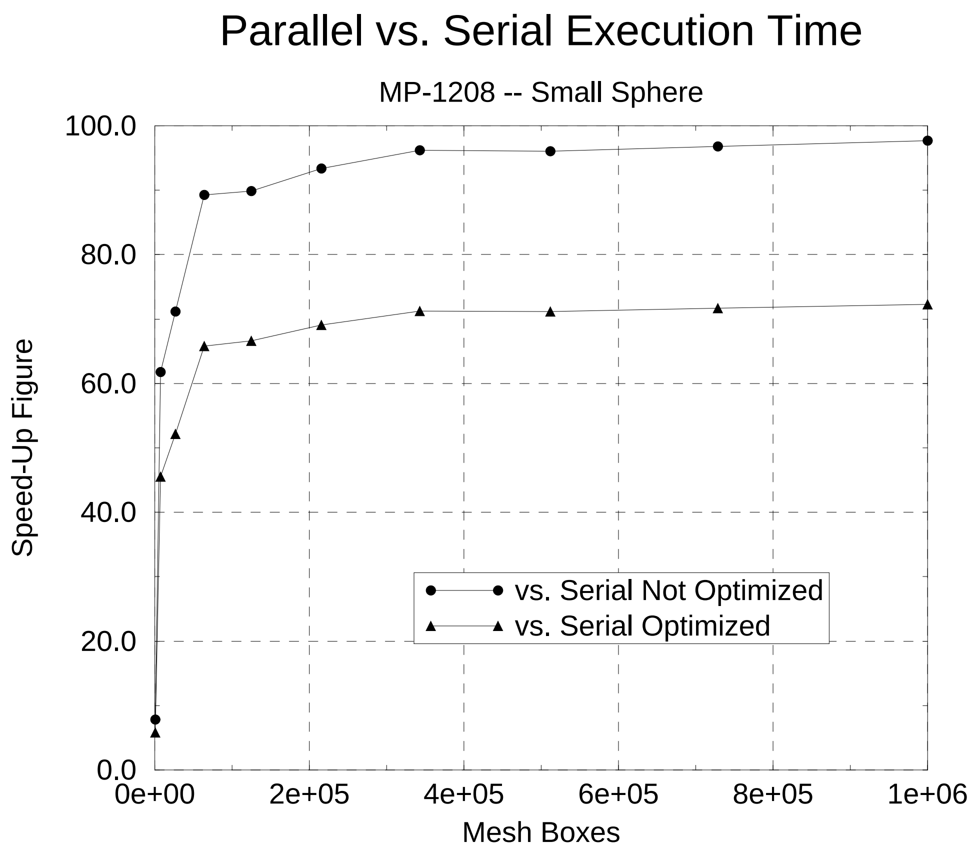


Figure 3.14: Speed-Up, MP 1208 vs. Serial

Comparing Parallel Systems

Figure 3.15 provides a speed-up comparison between the two parallel machines. Note that for the smallest mesh size the speed up factor is unity. For a 10x10x10 size mesh, the MP 1101 and the MP 1208 each use a single memory layer in the DPU to store the array of counters, so naturally for this case the speed up should be unity. For the parallel versions, the speed-up peaks to almost seven and appears to settle near 6.15 for large meshes. Unfortunately it is not clear why the speed up peaks for a 40x40x40 mesh. It may be because of the odd way that the memory layers fill. The size of peak seems to indicate the definite presence of some phenomenon.

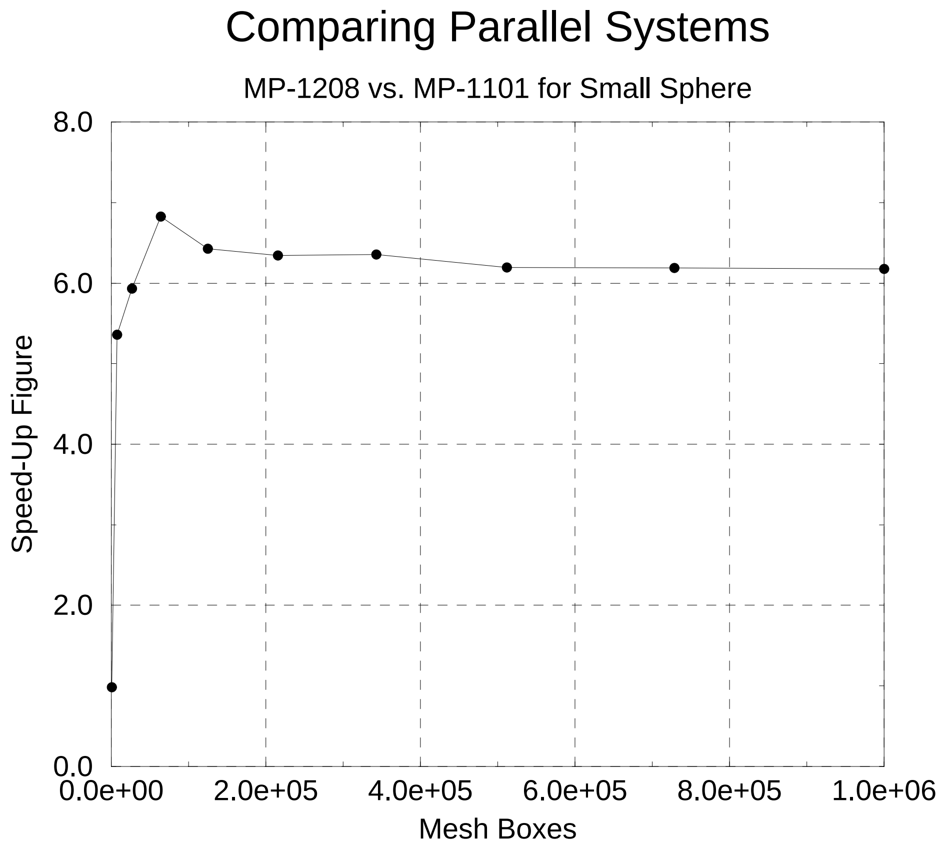


Figure 3.15: Speed-Up, MP 1208 vs. MP 1101

By increasing the number of PEs in the array by a factor of eight, the reader might expect a corresponding speed-up of eight times. According to Amdahl's law which was presented in section 2.9, such a direct relationship between execution time and the number of PEs does not exist. According to Amdahl's law, the "sequential part" of the implementation will serve as a limiting term that limits the maximum improvement in execution time that can be achieved by simply increasing the number of PEs. Equation (1.7) is easily solved to yield an expression that determines the percent of the execution time (ψ) that corresponds to the parallel part of the program.

$$\psi = \frac{E_f(\text{speed-up} - 1)}{(E_f - 1)\ \text{speed-up}} \tag{3.16}$$

The serial part of the program is simply equal to $1 - \psi$, after substituting in equation (3.16) the expression simplifies to the following expression. To determine the size of the serial component in terms of percent, simply multiply the speed-up by 100%.

$$\text{serial-part} = 1 - \psi = \frac{E_f - \text{speed-up}}{(E_f - 1)\ \text{speed-up}} \tag{3.17}$$

Note that for the speed-up to be equal to the enhancement factor E_f, which is in this case eight, then the numerator of equation (3.17) would be zero, thus implying that the serial part of the program is insignificant. Given the observed speed-up between the two parallel machines, equation (3.17) was used to determine the size of the serial part of the program execution time as a function of problem size. The results are presented in figure 3.16. Note that the data point corresponding to the 10x10x10 size mesh was not included, as there was no speed-up. Earlier we had explained that for this unique mesh size, both the MP 1101 and MP 1208 each require only one memory layer to store the counters used to implement the parity count method. For this unique situation, since the MP 1208 is to a large extent underutilized, there is no point in considering Amdahl's equation.

One thing is immediately obvious from the graph, it appears that what Amdahl's law refers to as the "serial part" of the program execution is not necessarily just that part of the mesh generation code that executes in the front end. As the mesh becomes larger and larger, the amount of time executing parallel code in the DPU should become more and more significant in comparison to the front end serial code, see section 3.6. According to figure 3.16, the serial part of the execution time is not falling as we would expect.

3.7.2 PE Use Figure

In examining the flowchart of figure 3.10, it should be clear that the facet solver task represents a hazard in terms of branching. A better PE use figure will result if a larger number of PEs participate in performing the facet solver. One theory is that as the problem size becomes ever larger, each memory layer in the PE array will correspond to a relatively smaller part of the overall mesh, implying that the

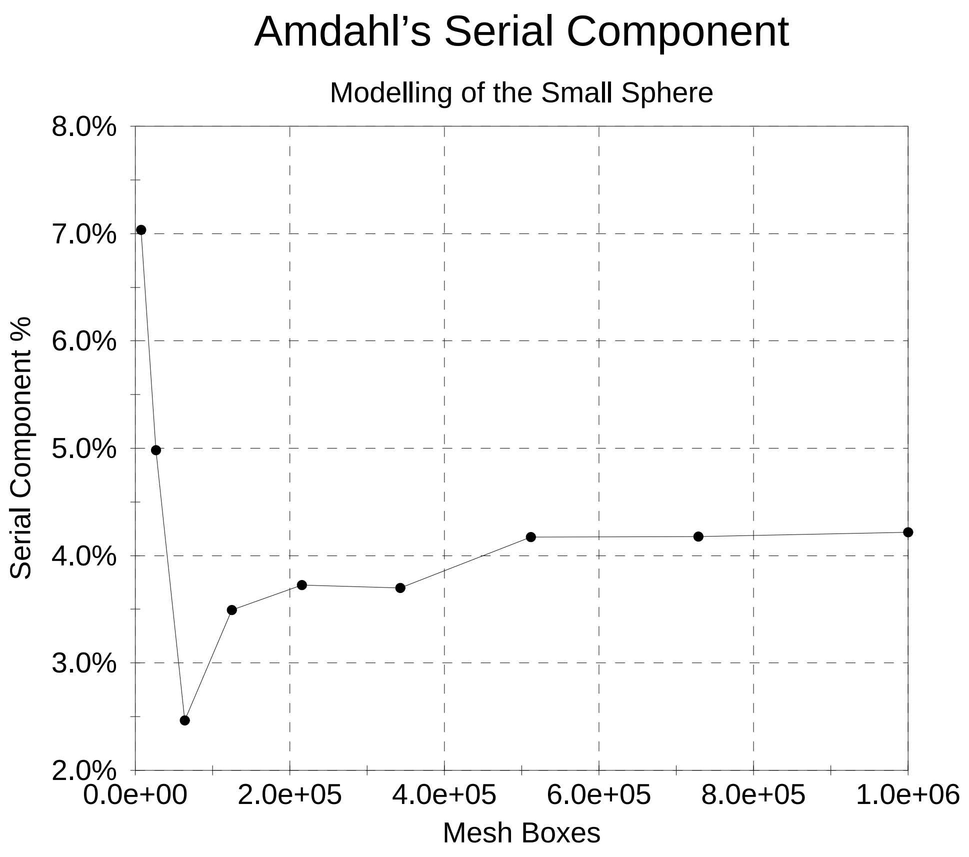


Figure 3.16: Serial Component of Mesh Generation

conditions experienced by each grid point in the memory layer will be more closely related. Thus it can be said in general that use of parallelism should improve with problem size.

To get a better understanding of the processor element use, two sets of of counters were inserted into the parallel facet solver code. The first set only contains a singular counter that is referred to as the facet solver entry counter (FS_e), which is used to count the number of times the ACU enters the code block associated with the facet solver. Recall that the ACU will only enter the facet solver code if at least one PE finds an intersection with the current facet plane. Thus for a shape being modeled, the final count in the facet solver entry counter (FS_e) does not have to equal the number of facets used to describe the shape.

The second set of counters inserted into the facet solvers is a plural variable referred to as the facet solver PE counter, which is used to count the number of times each PE participates in the facet solver code. After a shape has been modeled, all the facet solver PE counters are summed together to yield the total facet solver PE count (PE_{pe}). The following formula is used to determine the average overall PE use in performing the facet solver.

$$\mathrm{PE}\text{use} = \frac{\mathrm{FS}_{pe}}{\mathrm{FS}_e\,\mathtt{nproc}} \tag{3.18}$$

Table 3.5 provides a summary of the facet solver entry counter, total facet solver PE count, and PE use values for a range of mesh sizes. Figure 3.17 is an illustration of the corresponding PE use values.

Since the overall processor use figures that we have determined can be thought of as being the product of the PUFM and PUFB values, by independently determining PUFM, we can easily determine the values for PUFB. The processor element use figure due to mapping (PUFM) can easily be determined theoretically for a single shape. As indicated in section 3.2.4, the following equation gives the number of memory layers that are allocated for storing the counters associated with the current analysis subregion.

$$\mathrm{n\_layers} = 1 + \left\lfloor \frac{\mathrm{N\_AS}}{\mathtt{nproc}} \right\rfloor$$

As presented in section 3.2.2, N_AS is the number of mesh boxes in the analysis subregion and `nproc` is the number of PEs in the PE array. Since each memory layer represents `nproc` virtual processor elements, , the PUFM is found to be;

$$\mathrm{PUFM} = \frac{\mathrm{N\_AS}}{\mathrm{n\_layers}\,\mathtt{nproc}}$$

Table 3.5: **Overall PE Use Data**

Mesh	MP 1101			Node 'MP 1208'		
Boxes	FS_e	FS_{pe}	PE_{use}	FS_e	FS_{pe}	PE_{use}
1×10^3	256	128378	0.489723	256	128378	0.061215
8×10^3	1761	1028522	0.570367	256	1028522	0.490438
27×10^3	5891	3471639	0.575500	911	3471639	0.465186
64×10^3	12887	8229714	0.623639	1795	8229714	0.559668
125×10^3	23124	16073850	0.678824	3549	16073850	0.552871
216×10^3	37443	27776009	0.724435	5978	27776009	0.567184
343×10^3	57148	44107769	0.753727	9292	44107769	0.579450
512×10^3	82501	65840382	0.779351	13908	65840382	0.577880
729×10^3	116223	93745938	0.787699	19679	93745938	0.581513
1×10^6	157380	128595474	0.797951	26743	128595474	0.586983

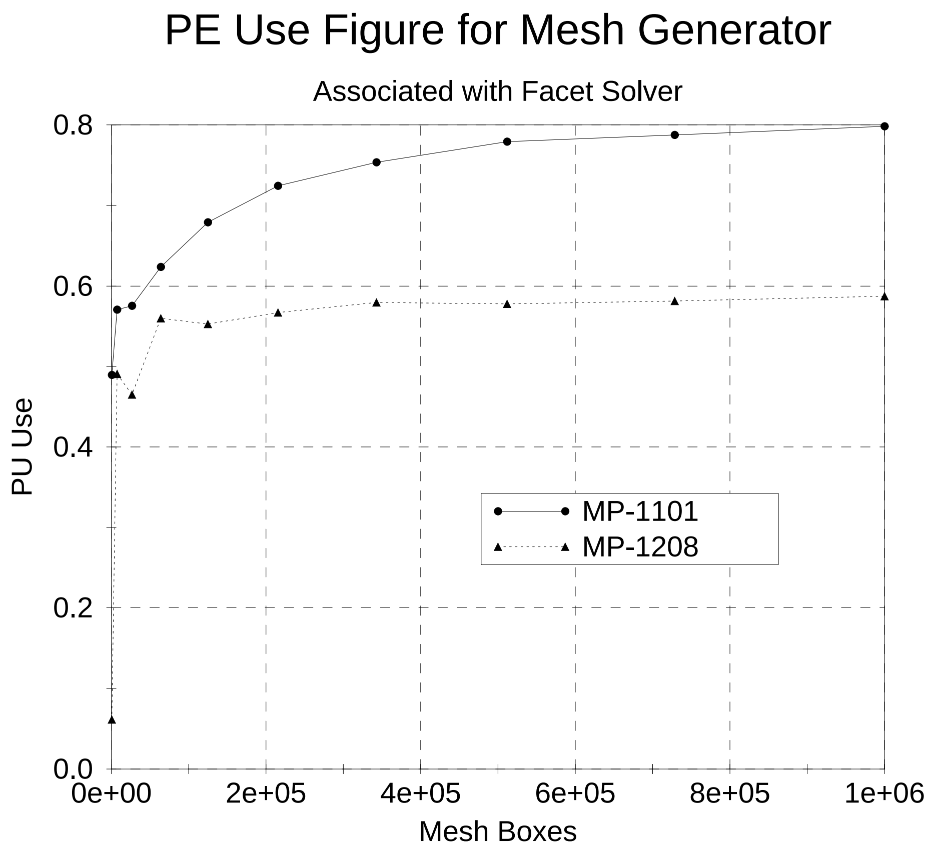


Figure 3.17: Overall PE Use Figure

Figure 3.18 is an illustration of the processor element use figure due to mapping for the given range of mesh sizes. Note that PUFM quickly improves with size of the mesh and hovers around unity.

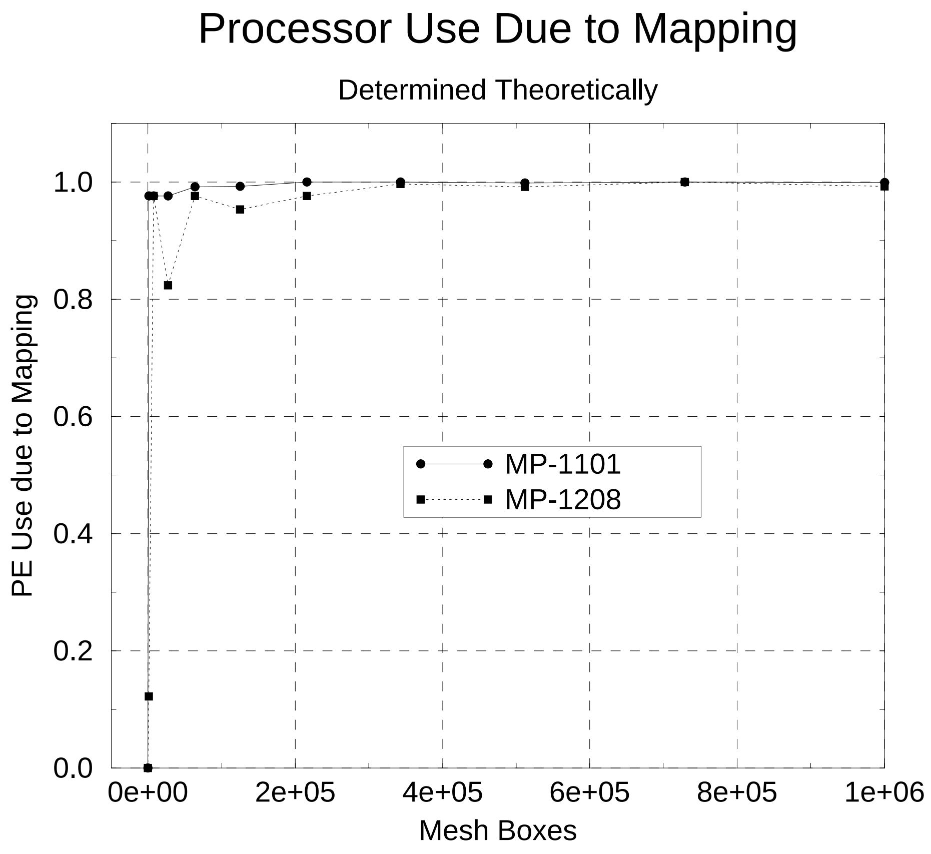


Figure 3.18: PE Use due to Mapping

To determine the processor element use figure due to branching (PUFB), it is only necessary to divide the overall processor element use figure by the processor element use figure due to the mapping (PUFM).

$$\mathrm{PUFB} = \frac{\mathrm{PE_{use}}}{\mathrm{PUFM}}$$

Figure 3.19 shows an illustration of the processor element use figure due to branching, associated with the calling of the facet solver function. For the smallest mesh considered (10x10x10), the PUFB was approximately 50% for both the MP 1101 and the MP 1208. As expected, as the mesh becomes larger, the PUFB value becomes larger. For the largest mesh considered, the MP 1101 had PUFB at almost 80% (actually 79.8309%), while the MP 1208 had PUFB at almost 60%

(actually 59.1453%). Thus it should be clear that while the MP 1208 is much faster than the MP 1101 the parallel version of the mesh generator runs more efficiently on the MP 1101.

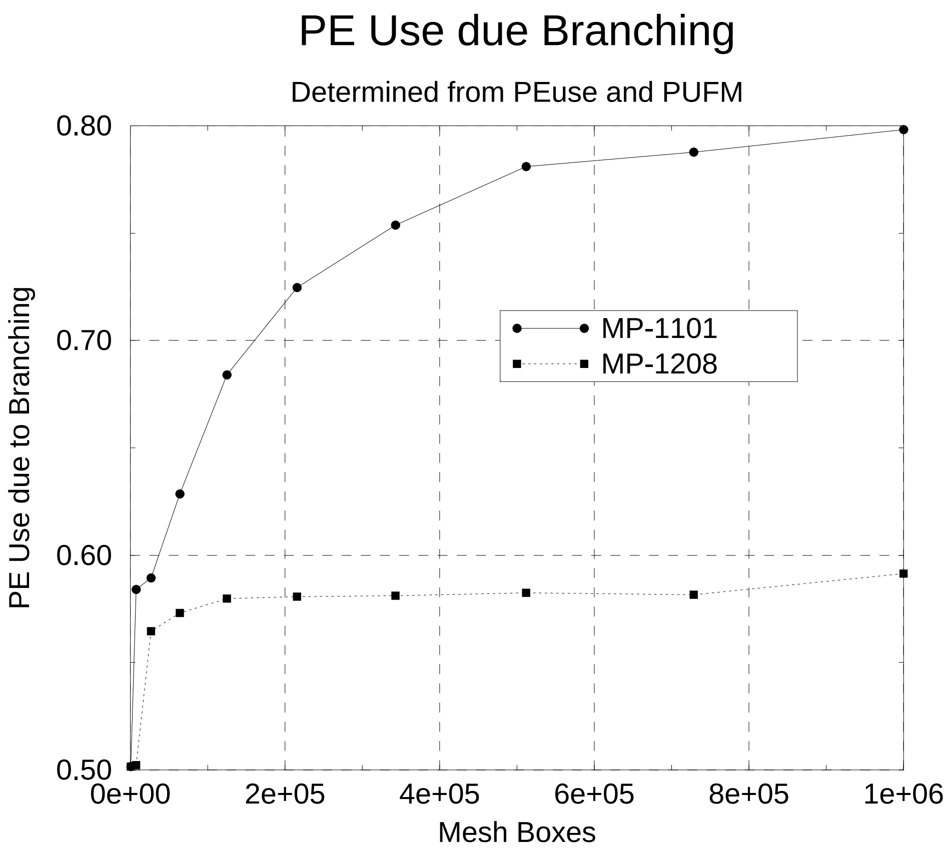


Figure 3.19: PE Use due to Branching

It is suspected that the reason for this variation in the use of the two machines is that for a larger PE array, the conditions experienced by each grid point will be more varied and result in a lower PE utilization than that of a smaller PE array. The data summarized in figures 3.19 and 3.17 helps as it is used to explain why a speed-up of approximately six was seen for large meshes, rather than the anticipated speed-up of eight. While the PE use figures that we have been examining are not for the overall program, since the execution time associated with the facet solver makes up a significant part of the execution time (see section 2.7.3 for representative data for the serial code), we can use the PE use figure associated with the facet solver to gain some insight into the overall program.

Consider the following, if we take the ratio of the overall PE use figure for the MP 1208 and the overall PE use figure for the MP 1101, and multiply the quotient by 8, the resulting graph (see figure 3.20) looks much like the speed up graph that compares the two machines. The reason for scaling by 8 is that the MP 1208 has that many times more PEs. Clearly there appears to be an approximate relationship between the PE use figure, the number of PEs, and the execution time of the array processor.

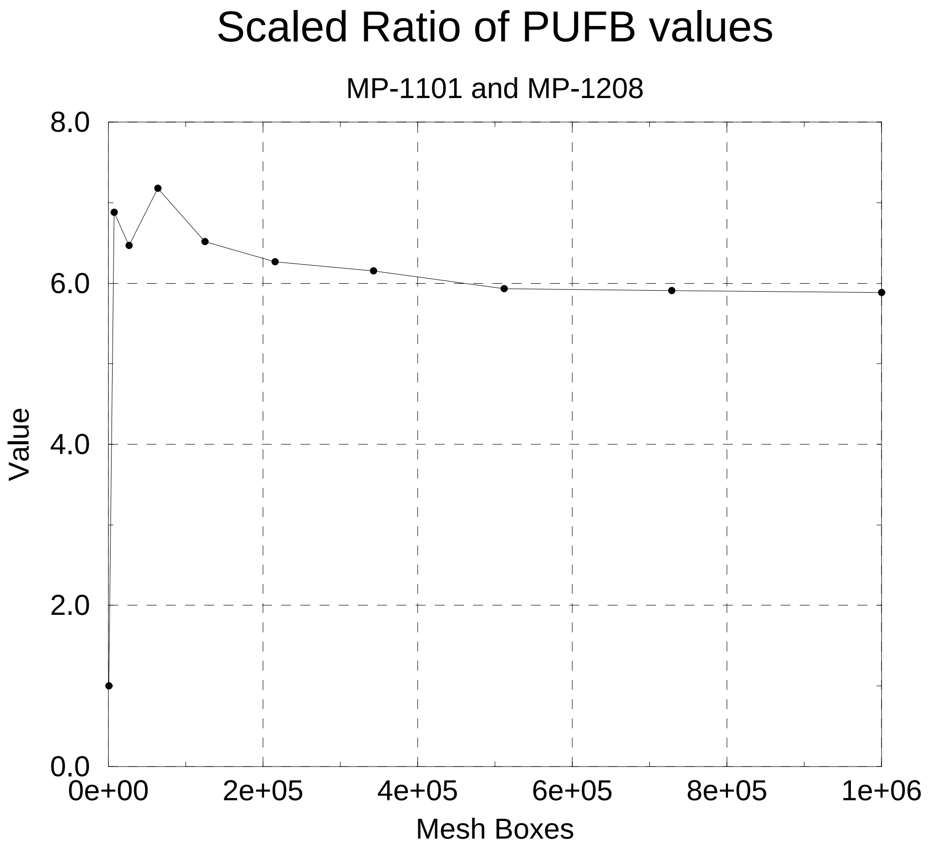


Figure 3.20: Scaled ratio of PUFB values

3.7.3 MFLOPS Performance

While it is true that the algorithm selected for the mesh generator is floating point operation intensive, it is also true that a less floating point intensive algorithm can be developed and implemented, section 2.10 serves as a testament of that fact. The point being made is that because of the nature of the mesh generator concept, the MFLOPS rate does not serve as a useful tool in comparing implementations of the mesh generator based on different algorithms. While we will be using the MFLOPS rate to compare our program implementations, it is important to realize that a different algorithm could easily produce code with a lower MFLOPS rate, yet execute in less time then the implementations presented here. Thus in examining the following, MFLOPS will not be considered as an absolute indicator of program performance.

Square Root Function

One of the issues we faced in determining the MFLOPS rates is the use of the square root function. As explained in section 1.7.5, a floating point operation is an addition, subtraction, multiplication, or division operation. While a square root function is assumed to be programmed as a power series that incorporates floating point operations, it is not easy to make an estimate of how many such floating point operations are actually used. In determining the MFLOPS rate we will assume that our algorithm encounters no singularities, which in turn implies that the only use of the square root function is in the front end.

To get a better understanding of the square root function, a test program was written, see figure 3.21. The program first times the execution of an empty loop. This timing is used to measure the overhead associated with the `for` construct. The double precision operation we are using is just a simple increment operation that uses a register variable. By measuring the time to execute one million such double length addition operations, we estimate the peak MFLOPS rate. Further, by taking the ratio of the time to perform one million double precision square root operations, to the time to perform one million double precision additions, we estimate the number of addition operations that execute in the same amount of time as a single square root operation.

Note that the goal of the program listed in figure 3.21 is not to determine the actual number of floating point operations that are associated with a square root operation, but to set a limit on the number of floating point operations that we are to associate with a square root operation. The double precision addition was specifically chosen as it should be the simplest and fastest to perform. In other words we may assume that fewer than an estimated number of floating point operations are actually associated with a square root operation. Based on this estimate we will be able to show that the contribution from the square root function to the total number of floating point operations is negligible.

In executing the test program listed in figure 3.21, the MFLOPS rate was estimated to be 8.46 MFLOPS. The equivalent number of adds per square root is determined by our test program to be approximately 50 floating point operations.

```
/*****************************************************************
 Estimate Floating Point Operations in Square Root

 Except for minor changes, this program was written by Mitch Mason,
 member of the support staff at MasPar corporation.  Mitch can be
 reached at mitch@mpppac.maspar.com.au -- Tue Aug  1 01:30:28 1995
 Compile this with cc -o target source.c -lm

 The program first determines the amount of time to execute one
 million iterations of an empty loop.  This timing figure is
 subtracted from later loop structures.  The second loop executes
 one million iterations of a simple addition operation.  The last
 loop executes one million iterations of a square root operation.
 The ratio of the times to execute the square root operations to
 the add operations gives an estimate of the number of addition
 operations that execute in the same amount of time as a single
 square root operation.
*****************************************************************/
#include "stdio.h"
#include "time.h"
#include "math.h"

main()
{
  unsigned int i, j, k, l1, l2;
  register double a;
           double d, e;
  a = 0.0;
  d = 10101.010101;

  clock(); /* Start system Clock. One tick = 1 micro sec. */
  for(i=0; i<1000000L; i++)
    {       /* Time the empty loop */
    }
  k=clock(); /* Loop overhead constant */
  printf("Loop Time Overhead Constant = %u \n", k);

  j=clock();
  for(i=0; i<1000000L; i++)
    { /* How long for 1Meg floating point add operations? */
      a = a + (double)1.0;
    }
  l1=clock()-j-k;
  printf("1 Meg. Double Float ops takes %u Micro Sec.\n", l1);
  printf("Double Float Add %f DOUBLE PRECISION MFLOP/Sec.\n",
         (float)1000000/l1);

  j=clock();
  for(i=0; i<1000000L; i++)
    {       /* Now, how long for 1Meg floating adds? */
      e=sqrt(d);
    }
  l2=clock()-j-k;
  printf("1 Meg. Double Float Sqrts takes %u Micro Sec.\n", l2);
  printf("Therefore one Double Float Sqrt takes %f D.P. OPS.\n",
         ((float)l2)/l1);
```

Figure 3.21: Source Listing of Square Root Test

Counting Floating Point Operations

To determine the MFLOPS rate, a special consideration has to be taken care of. Since the execution of the facet solver task is dependent on whether the intersection finder actually finds intersections, in examining the algorithm it is not immediately obvious what the number of floating point operations actually performed is. To remedy the situation it was decided that we would insert integer type counters into the code to keep track of the number of times certain events take place. For the serial code, the counter named FS_c keeps track of the number of times the facet solver was called. Also since facets may be defined with any arbitrary number of vertex points greater than three, another counter called LP_c was used to keep a total of the number of loop iterations that were actually performed inside the facet solver.

For the parallel code, the situation is just a little bit more complicated. For each iteration of the intersection finder task, `nproc` intersections can be detected by the PE array. If there is even one intersection found, the block of code associated with the facet solver must be entered by the ACU. As introduced in sections 1.5.3 and 1.7.3, in such situations the ACU must broadcast all the instructions that are required to perform the facet solver, even if only one PE is participating. The counter FS_e counts the number of times that the facet solver is entered by the ACU. Each PE is also assigned a counter that totals the number of times it performs the facet solver. To determine the number of instances in which the facet solver was performed, the PE counters are reduced to a singular value by use of addition and is assigned to the variable FS_{pe}. The variables FS_e and FS_{pe} are also useful for examining PE use, see section 3.7.2.

In a manner analogous to the serial code, counters are used in the parallel code to keep track of the looping performed inside the facet solver. A singular counter named LP_e keeps track of the total number of times that the ACU iterates the loop inside the facet solver. Also a plural counter keeps track of the total number of times each PE participates in the loop inside the facet solver. The instances of this plural counter are reduced by addition to a singular value named LP_{pe}. Table 3.6 summarizes the remaining variables used to characterize the MFLOPS rate. Note that the values given refer to the small sphere example.

The variable N_{rv} is associated with the actual number of references to vertex points made by the facet solver. The variable N_{rv} is defined as follows;

$$N_{rv} = \sum_{i=1}^{N_s} \sum_{j=1}^{N_{vi}} N_{v(i,j)}$$

Table 3.6: **Some Variables used in Determining MFLOPS**

Symbol		Description		Value
N_s	–	number of shapes being modeled	=	1
N_f	–	total number of facets in solid file	=	256
N_{rv}	–	vertex point reference number	=	992
N_{as}	–	grid points in analysis subregion		
N_{layers}	–	memory layers allocated		

Organization of Software Modules

To examine the MFLOPS performance closely, we must understand where floating point operations are performed in the code. Section 3.4.1 provides an introduction to the organization of the software modules. Figure 3.22 was produced by modifying figure 3.8 in section 3.4.1. The figure was modified to help in the discussion presented in the following sections. In this figure, the block 'Common Modules' refers to modules shared by the parallel and serial versions of the mesh generator. The number of floating point operations performed by the common modules is referred to by the variable 'FP$_{com}$'.

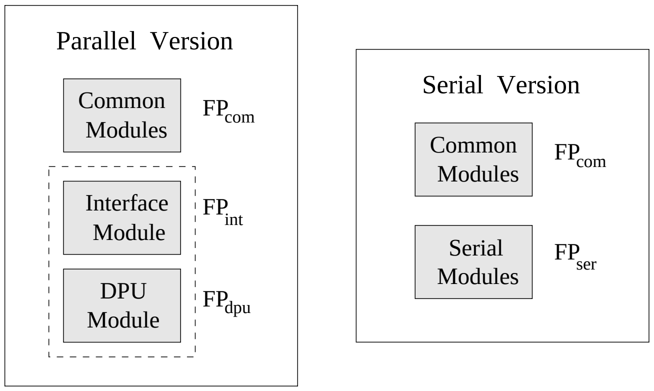


Figure 3.22: Software Modules in Generator Versions

The 'Interface Module' interfaces the parallel part represented by 'DPU module', and serial parts of the parallel version of the code. The number of floating point operations performed by the interface module and DPU module are referred to by the variables 'FP$_{int}$', and 'FP$_{par}$', respectively. Note that the interface module and DPU module are used only by the parallel implementation of the mesh generator. The number of floating point operations performed by the serial modules is referred to as ‘FP$_{ser}$’.

Common Module Floating Point Operations

After examining the implementations, equation (3.19) was devised to give the number of floating point operations that are performed by the common modules. The term FP$_{sqrt}$ refers to the actual number of floating point operations that are used to perform a square root operation. Note that in examining equation (3.19), it is noted that FP$_{com}$ is solely dependent on the input solid file. For the case of the small sphere if FP$_{sqrt} = 50$, then FP$_{com} = 70300$ floating point operations.

$$\mathrm{FP}_{com} = (74 + 3\,\mathrm{FP}_{sqrt})N_f + 13\,N_{rv} + 54\,N_s + 6 \tag{3.19}$$

Serial Module Floating Point Operations

Equation (3.20) was devised to determine the number of floating point operations that are performed in the serial modules that are only associated with the serial version of the mesh generator. Table 3.7 provides the parameters and number of floating point operations performed in the serial module for each given mesh size. The mesh sizes were all cubic with N_b mesh boxes along each coordinate axis, thus for each mesh $N_{as} = N_b^3$.

$$\mathrm{FP}_{ser} = 4N_f\left(N_b + N_b^2 + N_b^3\right) + 21N_{as}N_f + 13\mathrm{FS}_c + 19\mathrm{LP}_c + 30N_s + \mathrm{FP}_{sqrt} + 30 \tag{3.20}$$

Serial Version MFLOPS Rate

To determine the MFLOPS rate for a single given mesh size, divide the total number of floating point operations by the corresponding execution time. For the serial version the total number of floating point operations is the sum of FP$_{com}$ and FP$_{ser}$, see equation (3.21). Table 3.8 lists the actual MFLOPS rates. Note that the optimized and non-optimized serial versions performed the same number of floating point operations but had slightly different execution times, resulting in slightly different MFLOPS rates. Also see figure 3.23 for an illustration of the MFLOPS rates.

Table 3.7: **Variables Associated with Serial Version**

N_b	N_{as}	FS_c	LP_c	FP_{ser}
10	1×10^3	1.2838×10^5	4.9751×10^5	1.76343×10^7
20	8×10^3	1.0285×10^6	3.9847×10^6	1.40711×10^8
30	27×10^3	3.4716×10^6	1.3450×10^7	4.74426×10^8
40	64×10^3	8.2297×10^6	3.1883×10^7	1.12403×10^9
50	125×10^3	1.6074×10^7	6.2271×10^7	2.19472×10^9
60	216×10^3	2.7776×10^7	1.0761×10^8	3.79175×10^9
70	343×10^3	4.4108×10^7	1.7088×10^8	6.02033×10^9
80	512×10^3	6.5840×10^7	2.5507×10^8	8.98567×10^9
90	729×10^3	9.3746×10^7	3.6318×10^8	1.27930×10^{10}
100	1×10^6	1.2860×10^8	4.9818×10^8	1.75476×10^{10}

$$\mathrm{MFLOPS}_{ser} = \frac{\mathrm{FP}_{com} + \mathrm{FP}_{ser}}{\mathrm{Execution\_Time}} \tag{3.21}$$

Interface Module Floating Point Operations

Equation (3.22) gives the number of floating point operations performed by the interface module. In examining the equation it should be noted that the results are independent of the size of the resultant mesh and is only dependent on the number of shapes in the input solid file. By combining the expressions for FP_{com} and FP_{int}, we can see that for the example of the small sphere, the front end performs a total of 70434 floating point operations.

$$\mathrm{FP}_{int} = 54N_s + \mathrm{FP}_{sqrt} + 29 \tag{3.22}$$

DPU Floating Point Operations

Equation (3.23) was devised to determine the number of floating point operations that are performed in the DPU. Table 3.9 provides the parameters and number of floating point operations performed in the MP 1101 (node 'goofy'), for a range of mesh sizes generated of the small sphere. Table 3.10 provides the same data,

Table 3.8: **Floating Point Rate – Serial Version**

Mesh Size			FLOP Rate	
N_b	N_{as}	FP$_{total}$	Not Optimized	Optimized
10	1×10^3	1.7705×10^7	1.0984×10^6	1.4757×10^6
20	8×10^3	1.4078×10^8	1.0943×10^6	1.4858×10^6
30	27×10^3	4.7450×10^8	1.0909×10^6	1.4889×10^6
40	64×10^3	1.1241×10^9	1.0902×10^6	1.4792×10^6
50	125×10^3	2.1948×10^9	1.0907×10^6	1.4713×10^6
60	216×10^3	3.7918×10^9	1.0887×10^6	1.4715×10^6
70	343×10^3	6.0204×10^9	1.0909×10^6	1.4725×10^6
80	512×10^3	8.9857×10^9	1.0908×10^6	1.4730×10^6
90	729×10^3	1.2793×10^{10}	1.0912×10^6	1.4729×10^6
100	1×10^6	1.7548×10^{10}	1.0887×10^6	1.4716×10^6

but for the MP 1208 (node 'maspar'). Note that because of the way that the algorithm adjusts to different PE array sizes, the number of floating point operations performed in the MP 1101 will be similar but will not necessarily be identical to the number of floating point operations performed in the MP 1208.

$$\text{FP}_{par} = \Big(13N_{as} + 4N_{\text{layers}}\Big)N_f + 21\text{FS}_{pe} + 11\text{LP}_e + 8LP_{pe} + 5N_f + 12N_s \quad (3.23)$$

Parallel Version MFLOPS Rate

To determine the MFLOPS rate for a single mesh size, as before divide the total number of floating point operations by the associated execution time. For the parallel versions, the total number of floating point operations is the sum of FP$_{com}$, FP$_{int}$, and FP$_{par}$, see equation (3.24). Table 3.11 lists the actual MFLOPS rates for the MP 1101 and MP 1208, which are illustrated in figure 3.23.

$$\text{MFLOPS}_{par} = \frac{\text{FP}_{com} + \text{FP}_{int} + \text{FP}_{par}}{\text{Execution\_Time}} \quad (3.24)$$

Table 3.9: **Variables Associated with Node MP 1101**

N_{as}	N_{layers}	FS_{pe}	LP_e	LP_{pe}	FP_{par}
1×10^3	1	1.2838×10^5	9.92×10^2	4.9751×10^5	1.0017×10^7
8×10^3	8	1.0285×10^6	6.792×10^3	3.9847×10^6	8.0185×10^7
27×10^3	27	3.4716×10^6	2.2706×10^4	1.3450×10^7	2.7064×10^8
64×10^3	63	8.2297×10^6	4.9731×10^4	3.1883×10^7	6.4149×10^8
125×10^3	123	1.6074×10^7	8.9358×10^4	6.2271×10^7	1.2528×10^9
216×10^3	211	2.7776×10^7	1.4488×10^5	1.0761×10^8	2.1648×10^9
343×10^3	335	4.4108×10^7	2.2131×10^5	1.7088×10^8	3.4376×10^9
512×10^3	500	6.5840×10^7	3.1976×10^5	2.5507×10^8	5.1312×10^9
729×10^3	712	9.3746×10^7	4.5065×10^5	3.6318×10^8	7.3059×10^9
1×10^6	977	1.2860×10^8	6.1048×10^5	4.9818×10^8	1.0022×10^{10}

Table 3.10: **Variables Associated with MP 1208**

N_{as}	N_{layers}	FS_{pe}	LP_e	LP_{pe}	FP_{par}
1×10^3	1	1.2838×10^5	9.92×10^2	4.9751×10^5	1.0017×10^7
8×10^3	1	1.0285×10^6	9.92×10^2	3.9847×10^6	8.0114×10^7
27×10^3	4	3.4716×10^6	3.516×10^3	1.3450×10^7	2.7040×10^8
64×10^3	8	8.2297×10^6	6.924×10^3	3.1883×10^7	6.4096×10^8
125×10^3	16	1.6074×10^7	1.3685×10^4	6.2271×10^7	1.2519×10^9
216×10^3	27	2.7776×10^7	2.3049×10^4	1.0761×10^8	2.1633×10^9
343×10^3	42	441078×10^7	3.5824×10^4	1.7088×10^8	3.4352×10^9
512×10^3	63	658404×10^7	5.3618×10^4	2.5507×10^8	5.1278×10^9
729×10^3	89	937459×10^7	7.5868×10^4	3.6318×10^8	7.3011×10^9
1×10^6	123	128595×10^8	1.0316×10^5	4.9818×10^8	1.0015×10^{10}

Table 3.11: **Floating Point Rate – Parallel Versions**

Mesh Size		MP 1101		MP 1208	
N_b	N_{as}	FP_{total}	FLOP Rate	FP_{total}	FLOP Rate
10	1×10^3	1.0088×10^7	4.9860×10^6	1.0088×10^7	4.9003×10^6
20	8×10^3	8.0255×10^7	7.1913×10^6	8.0184×10^7	3.8513×10^7
30	27×10^3	2.7071×10^8	7.4702×10^6	2.7047×10^8	4.4272×10^7
40	64×10^3	6.4156×10^8	8.1371×10^6	6.4103×10^8	5.5478×10^7
50	125×10^3	1.2529×10^9	8.7083×10^6	1.2520×10^9	5.5934×10^7
60	216×10^3	2.1649×10^9	9.1481×10^6	2.1633×10^9	5.8009×10^7
70	343×10^3	3.4376×10^9	9.4258×10^6	3.4353×10^9	5.9862×10^7
80	512×10^3	5.1312×10^9	9.6653×10^6	5.1279×10^9	5.9800×10^7
90	729×10^3	7.3059×10^9	9.7406×10^6	7.3012×10^9	6.0259×10^7
100	1×10^6	1.0022×10^{10}	9.8330×10^6	1.0015×10^{10}	6.0694×10^7

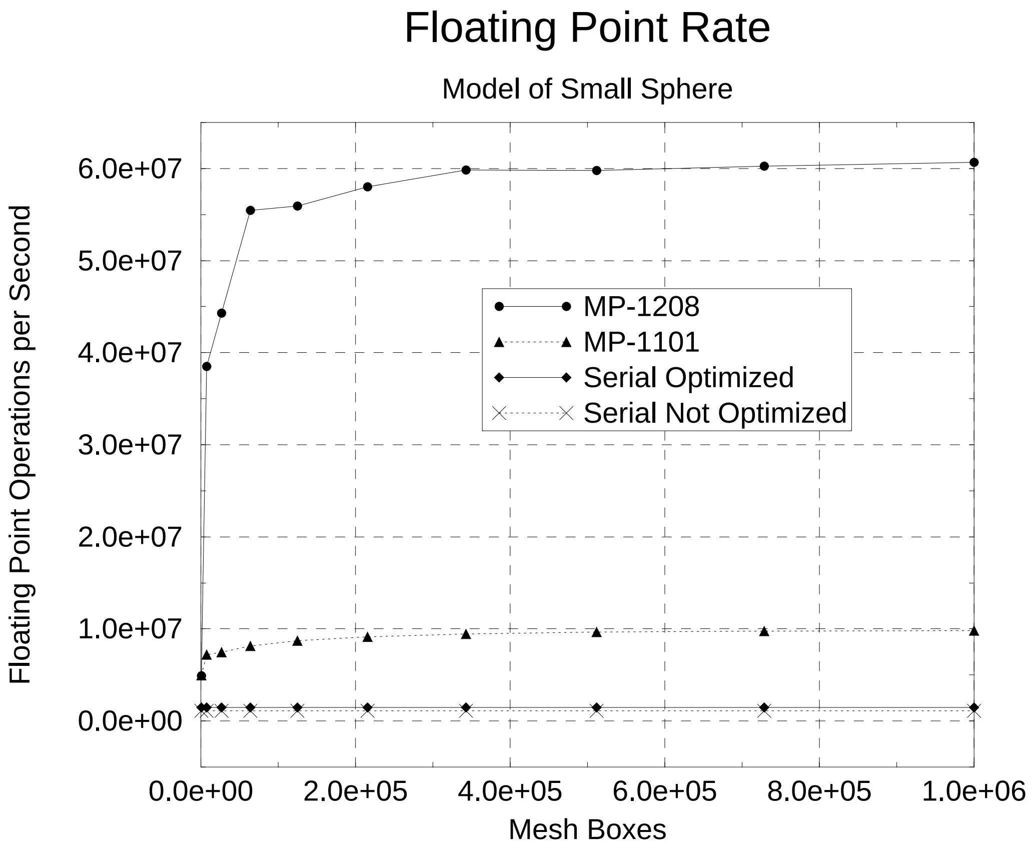


Figure 3.23: Floating Point Rate for Small Sphere

Review of Contribution from Square Root Function

As promised earlier, we will show that the number of floating point operations associated with our use of the square root function is insignificant in comparison to the total number of floating point operations that are actually performed. In examining the previous equations it is seen that the following equation correctly gives the total number of times the square root function is called in each of the implementations.

$$\mathrm{CALLS}_{sqrt} = 1 + 3N_f$$

Given that each call to the square root function corresponds to FP_{sqrt} floating point operations, we can say that our use of the square root function corresponds to this many floating point operations.

$$\mathrm{FP}_{tot\_sqrt} = (1 + 3N_f)\,\mathrm{FP}_{sqrt}$$

Further given that for the small sphere that $N_f = 256$ and assuming that $\mathrm{FP}_{sqrt} = 50$, then $\mathrm{FP}_{tot\_sqrt} = 38450$ floating point operations. This total is less than 0.4% of the total number of floating point operations performed for even the smallest mesh size that we considered. For larger mesh sizes, the contribution from our use of the square root function is even less significant.

Summary of MFLOPS Performance

In comparing the MFLOPS rates of the parallel versions with the serial versions, it should be very clear that while the parallel version does perform slightly fewer double precision floating point operations, the parallel version performs floating point operations much faster than the serial version. In fact the comparisons are remarkably similar to the speed-up curves. While encouraging, what really should be catching the reader's eye is how small the MFLOPS rates really are! For instance the program we used to examine the square root function indicated a FLOP rate of 8.46×10^6, which is significantly larger than the performance of approximately 1.5×10^6 that we observed with the serial mesh generators. Likewise, MasPar claims that a MP 1101 should have a peak double precision floating point rate of 34 MFLOPS, and a MP 1208 should have a peak double precision floating point rate of 275 MFLOPS. For a large mesh we observed the MP 1101 having a floating point rate of 10 MFLOPS and the MP 1208 at 60 MFLOPS. The point being made is that there is an apparent discrepancy here.

To understand this apparent discrepancy we must review a statement that was made earlier. While these implementations do perform floating point operations, it is NOT the goal of these implementations to only perform such operations. In a sense in reviewing the algorithm, floating point operations amount to the necessary drudgery that needs to be performed before solving the problem. Recall that the parity count method is really based on the principle of counting an integer number of intersections. Recall that in section 1.7.5, it was stated that the use of MFLOPS as an absolute performance metric is only valid if it is the sole purpose of a program to perform floating point operations.

While in this situation the use of MFLOPS does not make sense as an absolute measure of performance, we can use it as a relative measure of performance to compare similar implementations. This point will be discussed again in section 3.8. The point being made can be presented in another fashion; We have shown that there are other methods of performing the necessary drudgery of floating point operations, such algorithms use fewer floating point operations than our algorithm. The point is that the use of a relative performance metric has no meaning in comparing implementations based on very different algorithms.

3.8 Performance Summary

There are two goals of our performance analysis. The first goal is to show how efficiently the hardware platform is being used. The second goal is to show how effectively the implementation actually solves the problem. While these two goals are often seen as being linked, this is not necessarily the case. It is a simple matter to write a program that will effectively make full use of computer hardware, yet produce no useful results. Thus, in reviewing each of the performance metrics, we will consider each of these goals.

The speed-up curves clearly shows that the parallel implementation is much faster than the serial implementation, but the speed-up curves do not provide a clear indication of how effectively hardware is being used. Since the MP 1208 has eight times as many PEs as the MP 1101, we might hope for a speed-up of eight times between the two machines. For a large mesh size a speed-up of approximately six was actually seen, thus we might be led to the qualitative conclusion that node MP 1208 is not being used as effectively as it could be. Clearly in this situation a more quantitative conclusion is desired. Lastly, while the speed up curves clearly state that the parallel code more effectively generates a mesh than the serial code, the speed-up curves do not provide an absolute indicator of what the performance really is.

The PE use figures are useful as they provide insight into how effectively the hardware is being used. The idea of PE usage statistics is fundamental in an array processor as the available compute power is primarily related to the number of PEs that are involved in computation. While PE use figures do provide insight into how well the hardware is being used, the metric does not provide a very clear understanding of how effectively a mesh is actually being generated.

It was hoped that a determination of the MFLOPS rate would provide an indication of how effectively meshes are being produced. In floating point intensive type problems, the primary goal is typically limited to the need to quickly perform floating point operations, so naturally a performance metric like MFLOPS is useful. The whole point of the MFLOPS metric is to tell you how often floating point operations are actually being performed. Unfortunately, characteristics of the mesh generation algorithm make it pointless to use MFLOPS as an absolute indicator of how effectively a mesh is being produced.

First and foremost, while the mesh generation algorithm we selected does perform floating point operations, a large part of the algorithm involves the manipulation of binary data. Floating point operations are needed primarily to make decisions regarding whether certain binary counters should be incremented or whether certain flags should be set or cleared. A large part of the algorithm is involved in manipulating those flags and counters. Consider further that if we had selected a more effective facet solver, it would most likely use fewer floating point operations than our current algorithm. While the use of MFLOPS allows us to see that the parallel implementation is much faster than the serial implementation, it is important to realize in this instance that the use of MFLOPS does not provide a meaningful indication of how fast meshes are actually being produced.

3.8.1 An Alternative Metric

To provide an absolute indicator of how quickly meshes are being generated, a unique metric was devised. Since it is known from the complexity analysis that the execution time is proportional to the product of the number of facets in a solid file and the number of mesh boxes in the resultant mesh, a unique metric was based on this knowledge. The metric is called, *mesh boxes per facet second*, and describes the efficiency at with a mesh is produced. For a given mesh size, simply divide the total number of mesh boxes in the mesh by the product of the execution time and the number of facets in the associated solid file. While the metric does have some dependence upon the problem geometry, by dividing by the number of facets, it should be possible to compare the metric for meshes generated from different solid files.

The justification for this metric is very simple, just as compilers rate performance in terms of lines of source code compiled per second, it is useful to characterize the rate at which mesh boxes are generated. For the case of the small sphere there were 256 facets in the associated solid file. See figure 3.24 for a graph of the metric for various mesh sizes.

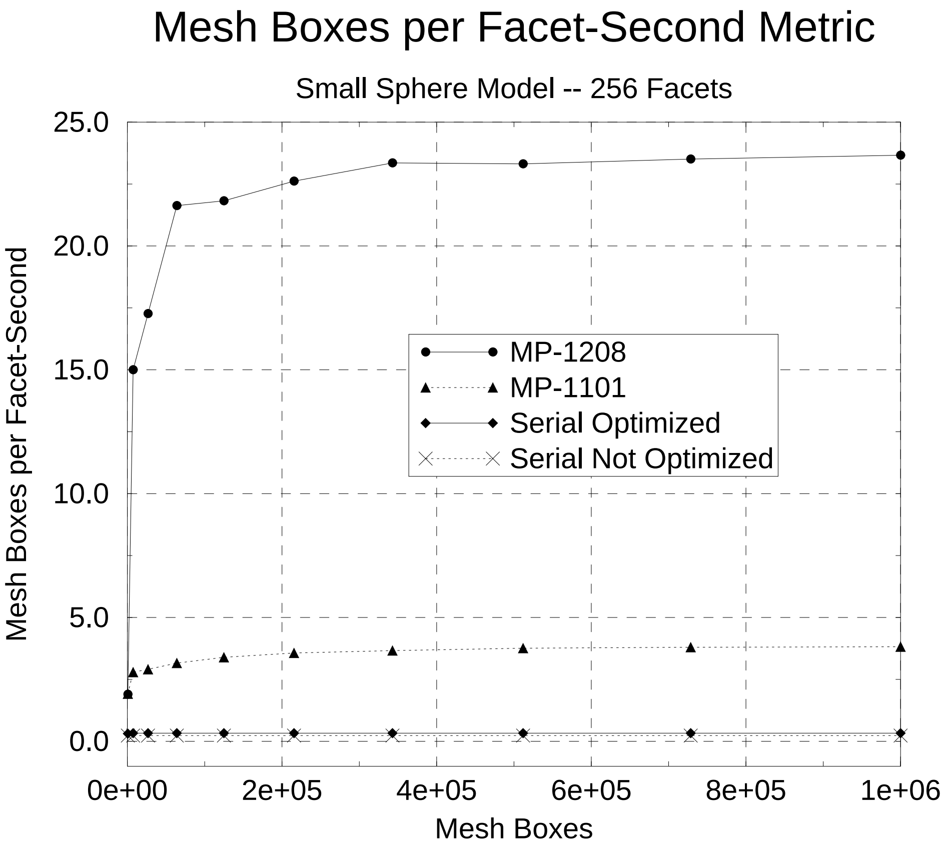


Figure 3.24: Absolute Performance Metric

It should be clear in examining figure 3.24 that there is not much of a difference in performance between the two serial implementations. While the boxes per facet second performance metric may at first just appear to resemble other performance metrics that we have considered, the real value in this metric is that it should allow us to compare in a very reasonable way, the performance of this implementation with other orthogonal mesh generators.

3.9 Summary

Both the serial and parallel implementations of the orthogonal mesh generator were presented. Important details such as transferring data to the DPU and data from the DPU, the mapping, the allocation of memory were introduced. The issue of time complexity was reviewed, this time as a parallel algorithm. Performance was examined by considering execution time, speed-up, the rate of performing floating point operations, the processor element use figure, and the rate at which mesh boxes were generated. For this type of algorithm we found that the processor use figure due to branching played an important role in describing how the implementation scales with PE array size. By examining the data, it appears that for moderately sized meshes, the front end consumes a very small amount of the overall execution time. Thus it appears that with this algorithm the asynchronous process model would offer an almost no improvement in performance.

Chapter 4

FDTD Algorithm

This chapter introduces the finite difference time domain technique as an algorithm that could be implemented on nearly any workstation. Important topics associated with FDTD analysis are first introduced. Practical issues associated with the actual use of the FDTD method are discussed as well. By allowing a magnetic loss term into the FDTD equations, materials can be modeled that allow for the development of a simple absorbing boundary condition that is uniquely suited for parallel computation. A derivation of suitable discrete difference equations is presented in Appendix E. The last section of this chapter formally presents the FDTD method as an algorithm.

4.1 FDTD Topics

This section introduces basic topics associated with FDTD analysis, starting with the FDTD mesh. The Yee Cell is introduced along with a description of its implications, especially the handling of materials and the reporting of field components. Central to FDTD analysis is the idea of time step *leap-frogging* which is explained. Lastly, the topic of FDTD stability is introduced.

4.1.1 Mesh Topics

In the following, we first review some ideas associated with the orthogonal mesh. While these topics were first presented in chapter 2, they are are presented again here for the convenience of readers who may be less interested in the mesh generation algorithm. Readers already familiar with this material may wish to skip to section 4.1.2 on page 175.

First we show how to define a bounding box. A bounding box having edges parallel to the coordinate axes is easily defined by interpreting P_{max} and P_{min} to be the extreme points on a diagonal. Figure 4.1 is an illustration of a bounding box.

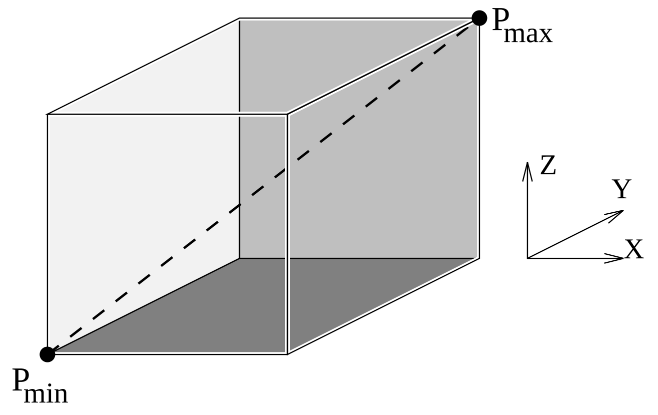


Figure 4.1: Three Dimensional Mesh Bounding Box

To form a mesh, along each edge of the bounding box place equally spaced marks to form brick shaped subdivisions that we call *cells* or *boxes*. There are N_x, N_y, and N_z subdivisions along each edge corresponding to the x, y and z coordinate axes. It will be noted that marks placed along each respective edge of the bounding box are spaced apart a distance Δx, Δy and Δz, corresponding to;

$$\begin{aligned}
\Delta x &= \frac{P_{max_x} - P_{min_x}}{N_x} \\
\Delta y &= \frac{P_{max_y} - P_{min_y}}{N_y} \\
\Delta z &= \frac{P_{max_z} - P_{min_z}}{N_z}
\end{aligned}$$

Points in the mesh can be associated with a unique index that can be expressed as an ordered triple of positive integer values. Note that i, j, and k are integers that correspond to real values x, y and z in the conventional right handed three dimensional coordinate system.

$$\text{INDEX} = [i, j, k]$$

Each index is valid in each of the following sequences. It is important to point out that while a bounding box edge parallel to the x axis has N_x divisions, for example, these subdivisions were formed between $N_x + 1$ marks placed on that edge.

$$\begin{aligned} i &= 0, 1, 2, \ldots, N_x \\ j &= 0, 1, 2, \ldots, N_y \\ k &= 0, 1, 2, \ldots, N_z \end{aligned}$$

The square braces associated with an index imply that the ordered triple is composed of integers rather than real physical dimensions. Also, parenthesis imply real physical coordinates. To determine the physical location (x, y, z) corresponding to $[i, j, k]$, we use the following formula:

$$\text{LOCATION} = (x, y, z)$$

$$\begin{aligned} x &= i\,\Delta x + P_{min_x} \\ y &= i\,\Delta y + P_{min_y} \\ z &= i\,\Delta z + P_{min_z} \end{aligned}$$

Given a brick shaped box, the corner with the smallest coordinate values (x, y, z) is called the *mesh box corner*. The significance of a mesh box corner is that it is the point that we use to reference any given box in the mesh. Since each box has a unique mesh box corner, each box must be uniquely identified by specifying a set of index values, $[i, j, k]$. Given that there are N_x, N_y, and N_z mesh boxes along each respective axis, when referencing mesh boxes inside the mesh, index values are valid on the following sequences.

$$\begin{aligned} i &= 0, 1, 2, \ldots, N_x - 1 \\ j &= 0, 1, 2, \ldots, N_y - 1 \\ k &= 0, 1, 2, \ldots, N_z - 1 \end{aligned} \tag{4.1}$$

4.1.2 The Yee Cell

Rather than defining all electric and magnetic fields at individual points in space, the Yee Cell distributes electromagnetic field components in a manner that makes the application of the discrete form of Maxwell's equations rather simple. The Yee Cell is presented graphically in figure 4.2. In the Yee Cell, electric field components are distributed along the edges such that each edge is parallel to the assigned component. Magnetic field components are defined to be normal and in the center of the cell faces.

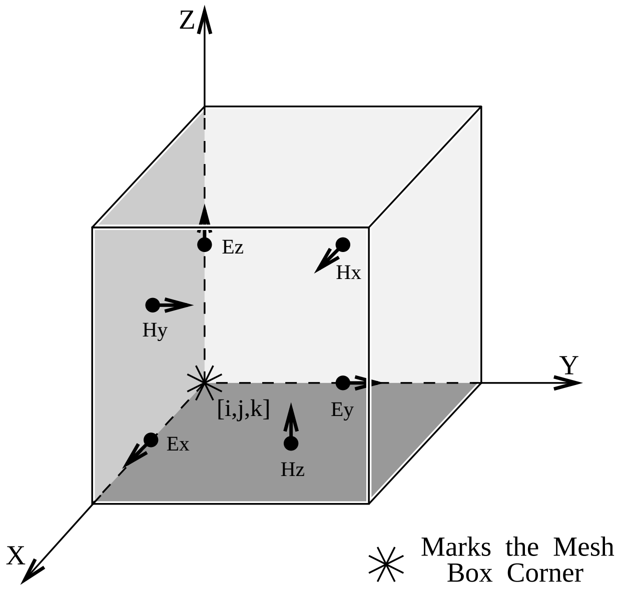


Figure 4.2: Three Dimensional Yee Cell

A standard notation is used to refer to field components in the mesh. This notation is similar to that used by Taflove and Umashankar[64]. The terms i, j, and k are simply index terms, as opposed to x, y, and z which correspond to real physical dimensions. This notation was selected as it should improve the readability of equations.

$$\begin{aligned}
E_x(x+\tfrac{1}{2}\,\Delta x, y, z) &\Leftrightarrow E_x[i,j,k] \\
E_y(x, y+\tfrac{1}{2}\,\Delta y, z) &\Leftrightarrow E_y[i,j,k] \\
E_z(x, y, z+\tfrac{1}{2}\,\Delta z) &\Leftrightarrow E_z[i,j,k]
\end{aligned}$$

$$\begin{aligned}
H_x(x, y+\tfrac{1}{2}\,\Delta y, z+\tfrac{1}{2}\,\Delta z) &\Leftrightarrow H_x[i,j,k] \\
H_y(x+\tfrac{1}{2}\,\Delta x, y, z+\tfrac{1}{2}\,\Delta z) &\Leftrightarrow H_y[i,j,k] \\
H_z(x+\tfrac{1}{2}\,\Delta x, y+\tfrac{1}{2}\,\Delta y, z) &\Leftrightarrow H_z[i,j,k]
\end{aligned}$$

A good mental tool that can be used to keep track of this use of notation is to think of a named component as referencing a field component, of the Yee Cell in the mesh box marked by mesh box corner $[i, j, k]$. For example, $E_x[1, 2, 3]$ refers to the E_x component in the Yee Cell associated with mesh box $[1, 2, 3]$.

Material Handler

While each mesh box is defined to have certain material properties, the Yee Cell defines the electric and magnetic fields as being located on the surfaces of mesh boxes. What this implies is that at the interfaces between mesh boxes, some mechanism is needed to properly handle material properties. To more fully understand what is being described, see figure 4.3 which illustrates a cross section of the interface between materials 'A', 'B', 'C', and 'D'. To be able to determine the field components, we must be able to define the material properties where field components are actually located.

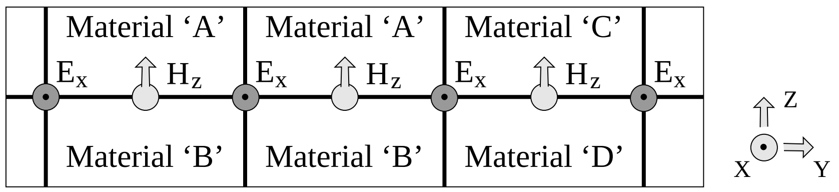


Figure 4.3: Interface Between Materials

Since field components are defined to be at interfaces between mesh cells, some means of determining the material properties at the interfaces between mesh cells is required. If adjacent mesh cells have identical material properties, then there is no question as to what material property to assign. If however, there is a difference in material properties between mesh cells, then a *material discontinuity* is said to exist. To handle such a situation, Li, Tassoudji, Shin, and J. Kong[29] suggest that we assign material properties at nodes by taking the algebraic average of adjacent mesh cells.

Since magnetic field components are centered between mesh boxes, to estimate the material properties required to update a given magnetic field component, we must average two materials. For example in figure 4.3, the magnetic permeability $\mu_z[i, j, k]$ is one of the material properties that we use to update magnetic field component $H_z[i, j, k]$. In reviewing figure 4.2, it should be clear that to determine the value of the needed magnetic permeability, the following applies;

$$\mu_z[i,j,k] = \frac{1}{2}\left(\mu_{zz}[i,j,k] + \mu_{zz}[i,j,k-1]\right)$$

While $\mu_{zz}[\cdot]$ refers to the z axis permeability assigned to each mesh cell, it is $\mu_z[\cdot]$ that is actually associated with updating $H_z[\cdot]$. For isotropic materials we can state that;

$$\mu_{xx}[i,j,k] = \mu_{yy}[i,j,k] = \mu_{zz}[i,j,k] = \mu[i,j,k]$$

While in most cases we will be using isotropic materials, it is important to remember that nodal properties are associated with the averages of the properties of adjacent mesh cells. Thus in general, even for isotropic materials:

$$\mu_x[i,j,k] \neq \mu_y[i,j,k] \neq \mu_z[i,j,k]$$

We handle the material properties associated with updating the electric fields in a similar fashion. Electric field components are located on the edges of the Yee Cell, thus each electric field component is centered between four mesh boxes. To estimate the material properties needed to update a given electric field component, we must average four materials. For example, in figure 4.3, the electric permittivity $\epsilon_x[i,j,k]$ is one of the material properties needed to update $E_x[i,j,k]$. In reviewing figure 4.2, it should be clear that to determine the needed permittivity value, the following applies;

$$\begin{aligned}\epsilon_x[i,j,k] = \tfrac{1}{4}(\quad & \epsilon_{xx}[i,j,k] & + & \quad \epsilon_{xx}[i,j-1,k] + \\ & \epsilon_{xx}[i,j,k-1] & + & \quad \epsilon_{xx}[i,j-1,k-1])\end{aligned}$$

Remember that while $\epsilon_{xx}[\cdot]$ corresponds to a material property associated with each mesh cell, it is $\epsilon_x[\cdot]$ that is actually associated with updating $E_x[i,j,k]$. The remaining material properties are presented in section 4.3 are handled in a similar fashion.

Estimating Components

While the Yee Cell simplifies the application of the discrete form of Maxwell's equations, it complicates the reporting of meaningful data. What we want to report to the user is the electric and magnetic field components at the points $[i, j, k]$, as defined in section 4.1.1. The problem is that while electric and magnetic field components are defined in the Yee Cell, they are distributed about the cell.

In examining the mesh closer, we can see that it is possible to estimate the electric and magnetic field components at the mesh box corners, by averaging adjacent field components. Figure 4.4 shows that to estimate electric field components at a point, we can average the electric field components along the edges that intersect that point. Note that the use of the hat symbol is meant to imply two things, first that the associated variable is estimated, and second that the estimate is located at the point $[i, j, k]$.

$$\begin{aligned}\hat{E}_x[i,j,k] &= \frac{1}{2}\left(E_x[i,j,k] + E_x[i-1,j,k]\right)\\ \hat{E}_y[i,j,k] &= \frac{1}{2}\left(E_y[i,j,k] + E_y[i,j-1,k]\right)\\ \hat{E}_z[i,j,k] &= \frac{1}{2}\left(E_z[i,j,k] + E_z[i,j,k-1]\right)\end{aligned}$$

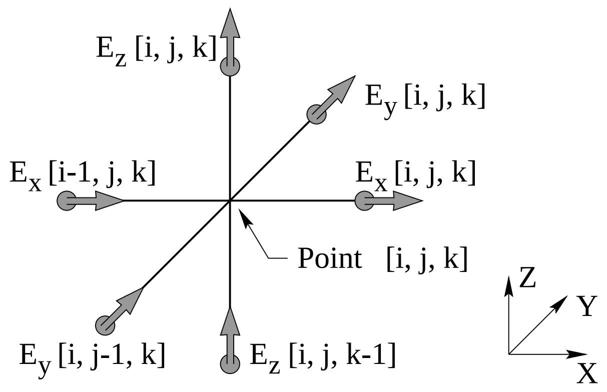


Figure 4.4: Estimating E Field Components

In a similar fashion, magnetic fields are estimated by averaging the four magnetic field components that surround a point. Figure 4.5 shows how point $[i, j, k]$ is surrounded by four H_x components. The situation is similar for the other magnetic field components. Lastly, to estimate components on the boundary surface requires special handling that is discussed further in section 4.4.2.

$$\hat{H}_x[i,j,k] = \frac{1}{4}\left(H_x[i,j,k] + H_x[i,j-1,k] + H_x[i,j,k-1] + H_x[i,j-1,k-1]\right)$$

$$\begin{aligned}
\hat{H}_y[i,j,k] &= \frac{1}{4}\left(H_y[i,j,k] + H_y[i-1,j,k] + H_y[i,j,k-1] + H_y[i,j-1,k-1]\right) \\
\hat{H}_z[i,j,k] &= \frac{1}{4}\left(H_z[i,j,k] + H_z[i-1,j,k] + H_z[i,j-1,k] + H_z[i-1,j-1,k]\right)
\end{aligned}$$

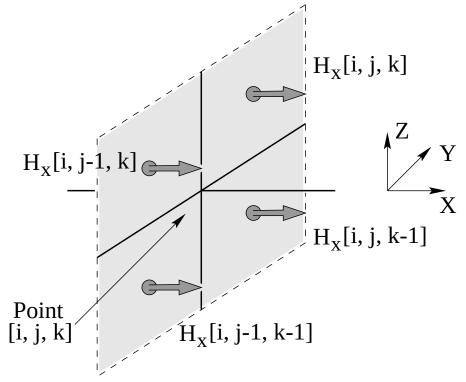


Figure 4.5: Estimating H_x Field Component

4.1.3 Time Stepping

To indicate time, Yee's[67] use of a superscript is used. The superscript n is related to time t by the following equation. The variable Δt is referred to as the *time-step* size.

$$t = n\,\Delta t$$

In the FDTD method, electric field values are only defined at whole number time steps, while magnetic field values are defined only at half time steps. Thus relative to time step n, two arbitrary electric field values can be expressed as;

$$E^n[i,j,k] \text{ and } E^{n+1}[i,j,k]$$

Likewise, relative to time step n, two magnetic field values can be expressed as;

$$H^{n+\frac{1}{2}}[i,j,k] \text{ and } H^{n-\frac{1}{2}}[i,j,k]$$

The benefit of defining electric and magnetic field components in this way in regards to time is that it allows electric and magnetic field component updates in a FDTD analysis to be interlaced in time. Electric field values are first calculated at each time step, then the magnetic field values are calculated at half time steps. Figure 4.6 provides an illustration of why the term leap-frogging is commonly

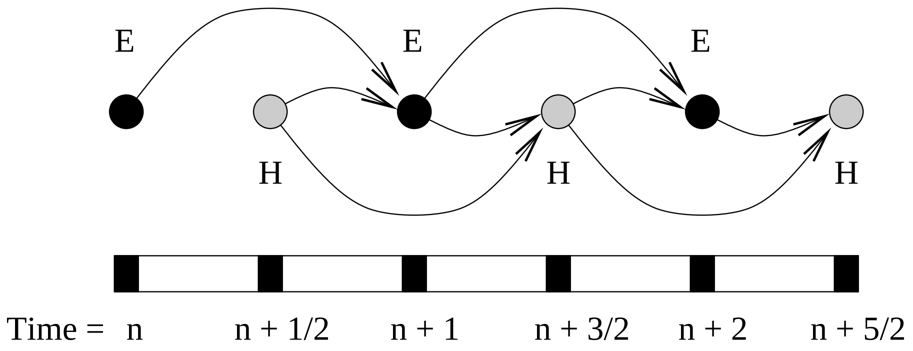


Figure 4.6: Time Step Leap-Frogging

used. Note that in updating a field component, only past electric and magnetic field components are required.

In cases where an electric field value is needed at a half time step, a value can be approximated by averaging electric field values between whole time steps. Likewise, in cases where a magnetic field value is needed at a whole time step, a value can be approximated by averaging magnetic field values between half time steps.

4.1.4 Stability and Accuracy

Since the primary goal of this thesis is to introduce a unique new implementation of the well known FDTD technique, rather than justify the FDTD technique, it seems appropriate that this section should simply introduce the reader to some basic guidelines regarding stability and accuracy of the FDTD technique. Regarding stability for an isotropic FDTD analysis, most authors refer to a paper by Taflove and Brodwin[62], which provides a derivation of the Courant criterion. According to Taflove and Brodwin, for a FDTD analysis based on central differencing to remain stable the following mathematical statement must be true;

$$v_{max}\Delta t \leq ((\Delta x)^{-2} + (\Delta y)^{-2} + (\Delta z)^{-2})^{-\frac{1}{2}}$$

Where v_{max} is the maximum wave phase velocity expected in the model, Δt is the time step size, Δx, Δy, and Δz refer to the mesh discretization size along each coordinate axis. For a simple implementation where cube shaped mesh cells are used,

$$\Delta x = \Delta y = \Delta z = \delta$$

thus the stability criterion can be restated as;

$$v_{max}\, \Delta t \leq \frac{\delta}{\sqrt{3}}$$

The choice of the mesh density and the time step size is motivated by the need for analysis stability as well as accuracy. According to Li, Tassoudji, Shin and Kong[29], in general the length of an edge of a mesh cell must be a small fraction ($\sim \frac{1}{10}$) of either the smallest wavelength in any media expected in the model or the smallest dimension of an object being modeled. For a detailed discussion on the topic of stability and FDTD accuracy conditions, the reader is referred to Bayliss, Goldstein and Turkel[6], Petropoulos[45], as well as Taflove and Umashankar[64].

In one sense, these requirements are analogous to Nyquist's criterion which is well known in digital signal processing. Nyquist's criterion states that the sampling interval must be less than half the period of the highest frequency component of the signal being sampled. The way that the Yee cell defines field components at specific places in a mesh corresponds to discrete sample locations. Also the size of a mesh box corresponds to the discrete spatial sampling interval. The whole point of the stability criterion and requirements for accuracy is that a wave be must large in comparison to any individual mesh cell and that the distance an electromagnetic wave travels in one unit of time must be small relative to individual mesh cells.

The stability criterion sets a minimum limit in the relationship between the time step size, mesh cell size, and propagation speed. Likewise the need for accuracy sets a limiting relationship between the quality of results and the overall mesh size. While this discussion may seem to be purely academic at this point, it will be revisited in the next chapter when the topic of memory allocation is discussed for the FDTD implementation.

4.2 Frequency Domain Analysis

While the FDTD method is conducted in the time domain, it has become standard practice to take samples from fixed locations in a FDTD mesh, and use a Fast Fourier Transform to post process the results. Oppenheim and Schafer[42] provide a clear introduction to the concept of the Fast Fourier Transform algorithm. Such post processing allows for the estimation of scattering parameters and feed-point impedance of a FDTD model over a wide range of frequencies. Zhang and Mei[68] as well as Sheen, Ali, Abouzahra, and Kong[56] present the details on how such an analysis is performed. In a few instances we used these techniques to post-process the results from our research.

4.3 Finite Difference Equations

This section introduces the equations used to derive the finite difference equations used to perform the FDTD method. Note that since this chapter is supposed to present an algorithm, rather than including the derivation of FDTD equations here, the derivation is presented in appendix E. This section only goes so far as to present the general form of the finite difference equations. The equations we use apply to three dimensional space and assume linear materials in a source-less region. Further, all variables are real valued.

According to Li, Tassoudji, Shin and Kong[29] Maxwell's divergence equations are always satisfied in the FDTD scheme by ensuring that initial and boundary conditions are correctly applied. Thus the actual difference equations are based solely upon Maxwell's curl equations and the constitutive relations which have already been applied.

$$\nabla \times \vec{H} = \epsilon \frac{\partial \vec{E}}{\partial t} + \sigma \vec{E} \tag{4.2}$$

$$\nabla \times \vec{E} = -\mu \frac{\partial \vec{H}}{\partial t} - \sigma_m \vec{H} \tag{4.3}$$

Using the MKS system of units, table 4.1 summarizes the variables that are central to Maxwell's Equations. Note that except for the term σ_m which is fictitious, the remaining terms in this presentation of Maxwell's curl equations should be familiar. Li, Tassoudji, Shin and Kong[29] use these same equations and identical notation. Taflove and Umashankar[64] use these same equations but only with slightly different notation. Taflove and Umashankar explain that the term σ_m is provided to yield symmetric curl equations and to allow for the possibility of a magnetic loss mechanism. They refer to this loss mechanism as *equivalent magnetic resistivity*, see page 293 of their book. Note however that since we follow the notation from Li et al[29], we will be consistent with their use and refer to this loss mechanism as *magnetic conductivity.* While the magnetic loss term σ_m is fictitious, its use will allow us to develop a simple material that can be used immediately to render an effective absorbing boundary condition. The details of this special material are presented in section 4.5.

Table 4.1: **Terms in Maxwell's Equations**

Term		Description		Units
$\vec{E}$	–	Electric Field Intensity	–	Volts / Meter
$\vec{H}$	–	Magnetic Field Intensity	–	Amperes / Meter
ϵ	–	Electric Permittivity	–	Farads / Meter
μ	–	Magnetic Permeability	–	Henrys / Meter
σ	–	Electric Conductivity	–	Siemens / Meter
σ_m	–	Magnetic Conductivity	–	Ohms / Meter

As you will see in examining the derivation in appendix E, from an algorithmic point of view, equations used to update electric fields, or update magnetic field values look very similar and generally fit the following general form;

$$F^{\text{new}} = \text{KF}_\ell \cdot F^{\text{previous}} + \text{KF}_c \left(\frac{G_{b+}^{\text{previous}} - G_{b-}^{\text{previous}}}{\Delta a} - \frac{G_{a+}^{\text{previous}} - G_{a-}^{\text{previous}}}{\Delta b} \right)$$

The terms KF_ℓ and KF_c can be thought of as proportionality constants. The letter F that is part of the names KF_ℓ and KF_c, indicates a symbolic association with the variable F. The variable F is either an electric (E) or a magnetic field (H) component in the Yee Cell. The variables Δa, Δb, and Δc are mesh discretization dimensions that correspond to Δx, Δy, and Δz, but not necessarily in that order. The remaining terms are the field components, either electric or magnetic that are needed to perform the given update.

Two sets of equations similar to this generic form are used to perform the field updates. One set of equations updates electric field components, the other set updates magnetic field components. In the derivation of the actual difference equations, the reader should keep this general form in mind.

4.4 Mesh Boundaries

In the following, the handling of mesh boundaries is examined. From algorithmic as well as implementation viewpoints, our interest in mesh boundaries is motivated by the fact that a FDTD mesh must be finite in size. While the form of the finite difference equations presented in section 4.3 assumes an infinite space, we must remember that all computers have finite memory. Unfortunately, the finite difference equations will not allow us to simply "chop up" a theoretically infinite mesh, to produce a finite size mesh. For the finite difference equations to remain valid, special handling is needed near those places where the FDTD mesh will end. (i.e. near the surfaces of the bounding box.) We first consider some practical aspects of mesh boundaries, next we consider the modeling of certain mesh boundary conditions.

4.4.1 Practical Mesh Boundaries

This section addresses some practical aspects of mesh boundaries. While the mesh concepts presented in section 4.1.1 and the idea of the Yee Cell presented in section 4.1.2 makes sense from a conceptual standpoint, only a small amount of thoughtful reflection is needed to realize that these concepts do not provide a fully useful explanation of what mesh boundaries really are, or what the FDTD mesh really is. Since this section discusses boundaries, this is the most appropriate place to discuss such an issue in detail.

In examining the Yee Cell illustrated in figure 4.2, we can see that the Yee Cell defines electric field components on only three edges of the mesh box and that magnetic field components are defined only on three surfaces of the mesh box. Note however that when Yee Cells are placed side by side, and on top of each other, the Yee Cells actually "fill in" a three dimensional space. What this means is that all of the needed electric and magnetic field components in the interior of the mesh are properly defined. Figure 4.3 should help give the reader some sense of this point. While all this should make sense, the attentive reader will ask, "But what about the north, east and top surfaces of the of the bounding box, are electric and magnetic field components defined there?" For the reasons presented next, the answer to this question is 'yes'.

The FDTD algorithm was written for this application to allow the mesh to be augmented. The variable `offset_pos` defines the number of mesh layers that are added to the north, east, and top sides of the mesh. In the current implementation, `offset_pos` is defined to be one. Thus the actual mesh that we use in the FDTD analysis, referred to as the *augmented mesh* has more mesh boxes than the mesh produced by the mesh generator code. We refer to the mesh produced by the mesh generator as being the *standard mesh*. To be able to reference mesh boxes that are outside the standard mesh, but inside the augmented mesh, we will have to be able to use index values that are outside the sequences given by equation (4.1). Thus for mesh boxes in the augmented mesh, index values are valid on the following sequences;

$$\begin{aligned} i &= 0, 1, 2, \ldots, N_x \\ j &= 0, 1, 2, \ldots, N_y \\ k &= 0, 1, 2, \ldots, N_z \end{aligned} \tag{4.4}$$

The variable `offset_neg` defines the number of mesh layers that are to be added to the west, south, and bottom sides of the standard mesh. In the current implementation, `offset_neg` is defined to be zero, and because of this we will not be concerned with `offset_neg`. Figure 4.7 illustrates a 2x2x2 standard mesh, along with the additional layers needed to form a 3x3x3 augmented mesh.

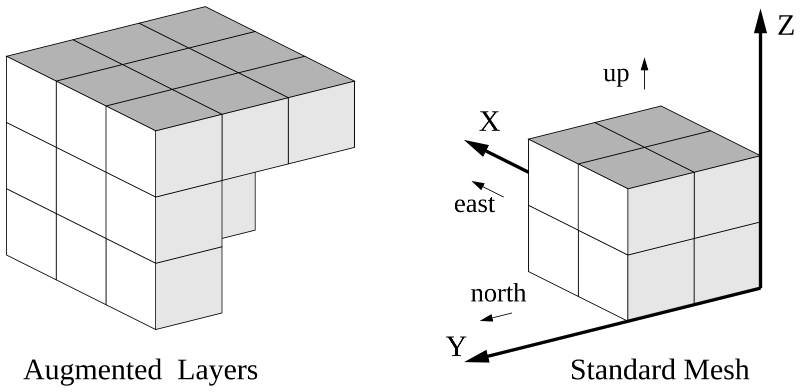


Figure 4.7: Construction of Augmented Mesh

Figure 4.8 provides an illustration of the cross section of an augmented mesh, along with electric field components from the associated Yee Cells. In this illustration the cross section of the standard mesh (shaded) has 3 columns and 2 rows, thus the overall cross section of the augmented mesh has 4 columns and 3 rows. Note that in the augmented layers, only the electric field components against the standard mesh are shown, this illustrates the fact that in the augmented layers of the mesh, we only use the electric and magnetic field components that are against the east, north, and top of the standard mesh. What we call the *FDTD mesh* is actually the Yee Cells in the standard mesh, along with the parts from the augmented layers that we need to define the boundary conditions.

To summarize, the need for an augmented mesh is that it allows us to define electric and magnetic field components where we need them on the boundaries. While this section does provide a slight taste of the implementation, the next chapter dives far deeper into the implementation.

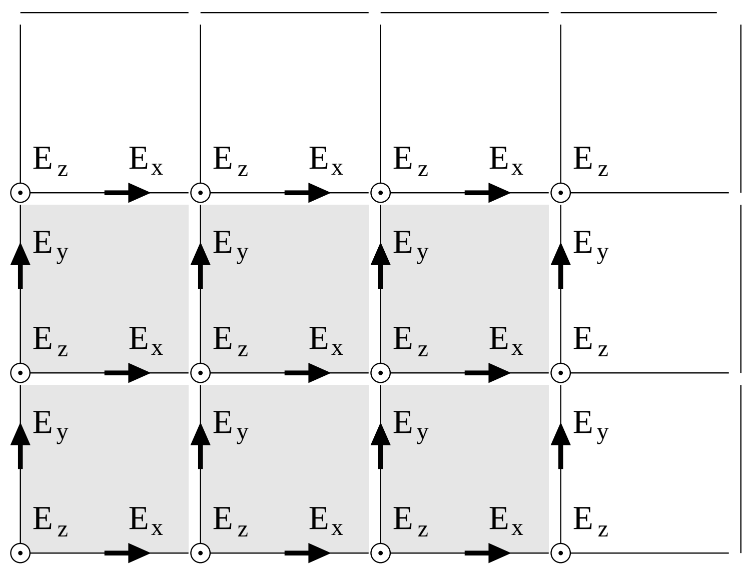


Figure 4.8: Augmented Mesh with Standard Mesh Shaded

Estimating Components Along the Mesh Boundary

For situations where a mesh point is on the boundary surface of the FDTD mesh, it is not possible to use the equations presented in section 4.1.2, (page 177), for estimating field components, because points needed to perform the equations lie outside the FDTD mesh and are thus undefined. To handle the situation, the field estimation equations are modified to average only those field components that are available. In instances where only one electric field component is available, that component will be taken as the estimate. In instances where only two magnetic field components are available, the average of the two components will be used as the estimate. Lastly in instances where only one magnetic field component is available, it will be used as the estimate.

Handling Materials Along the Mesh Boundary

For situations where an edge of a Yee Cell is on the boundary surface of the FDTD mesh, it is not possible to use the equations presented in section 4.1.2, (page 176) for averaging material properties, because material properties needed lie outside the mesh and are thus undefined. To handle the situation, material properties associated with magnetic field updates that are available will be used. Instances in which the material properties associated with electric field updates exist only in one adjacent Yee Cell, the average will be used, otherwise the available property will be used.

4.4.2 Boundary Conditions

By handling field components along the mesh boundaries in certain ways, we will model certain electromagnetic phenomena associated with surfaces. The handling of field components along the mesh boundary forces neighboring field components to behave as is if there were actually a surface along the boundary of the mesh. For example we can model the effect of a perfectly conductive metal surface. Because a special handler separate from the FDTD difference equations is directly manipulating field components, we use the phrase *applied boundary condition* to describe the handling of the mesh boundary conditions. In a sense we handle boundary conditions by "applying" specific field values directly to the mesh.

In the following sections we will introduce several methods of handling mesh and spatial boundaries by first describing the effect that each method models. Li, Tassoudji, Shin and Kong[29] provide a meaningful explanation of PEC as well as PMC boundary conditions, their paper is cited as a reference for those two topics.

PEC Boundary Conditions

The term PEC is an acronym for *perfect electric conductor*, and is used to model a perfectly conductive metal surface. It is interesting to note that what Kane Yee[67] referred to as a perfectly conducting surface in his seminal paper, is what we refer to as PEC boundary conditions. The boundary conditions at a perfect electric conductor are such that the electric field components tangential to the surface must be zero, stated mathematically where $\vec{n}$ is a surface normal vector;

$$\vec{n} \times \vec{E} = 0$$

By examining the Yee Cell in figure 4.2, it should be clear that the electric fields calculated at points on the surface of a perfect electric conductor are always tangential to the surface. Thus by using the Yee Cell in the finite difference time domain scheme, the boundary condition at the surface of a perfect electric conductor can be satisfied by simply setting these electric field components equal to zero at every time step.

In addition to being used on the boundaries of the FDTD mesh, PEC type conditions can be assigned to the surface of Yee Cells that are inside the mesh. The use of PEC surfaces inside the mesh allows us to model perfectly conductive metal surfaces of minute thickness. The use of such conditions helps in the modeling of strip-lines and other such structures.

PMC Boundary Conditions

The term PMC refers to a perfect magnetic conductor. The boundary conditions at a perfect magnetic conductor are such that the magnetic field components tangential to the surface must be zero, or rather;

$$\vec{n} \times \vec{H} = 0$$

It should be clear by looking at figure 4.2 that on the surface of a Yee Cell, there are no tangential magnetic fields specified. To handle this situation, Li, Tassoudji, Shin and Kong recommend use of the image method. As we shall see, using the image method implies a certain restriction. Shen and Kong[57] provide a clear introduction to the image method. Unlike the typical application to electrostatics, but like magnetostatics we apply the image method to discrete magnetic field components. Figure 4.9 part (a) illustrates the interface between media and a perfect magnetic conductor. Part (b) is an illustration of the image problem. The

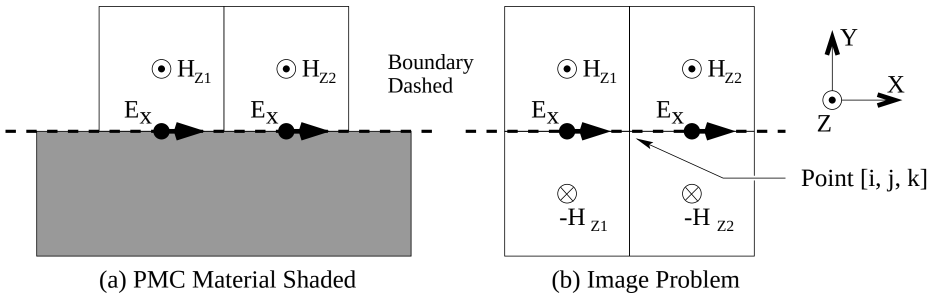


Figure 4.9: PMC Boundary Conditions

image problem is easily solved by inspection, clearly the *phantom* image fields take on values opposite the values of corresponding field values in the media. In doing so if we were to estimate the magnetic field tangent to the PMC boundary at point $[i, j, k]$, (See section 4.1.2 on page 177.) we would average adjacent magnetic field values. Since the effective adjacent field values are opposite, the average must be zero.

PMC surfaces are often used in the modeling of a symmetric region. The idea is that if a model and its excitation can be shown to be symmetric about a given surface, then instead of simulating the whole model, it is only necessary to simulate one of the symmetric parts of the model. Thus by symmetry, we can realize a significant savings in computer resources (time, computer memory, data storage) and will be able to simulate a model that would not otherwise fit into the computer‘s memory.

By now the reader may have an idea as to the restriction implied by the image method. Note the image method implies a sense of *image sides*, that is, there is an *in-side* and a *phantom image side*. This sense of image sides does not present a problem if we wish to make a surface associated with the mesh bounding box appear as a perfect magnetic conductor. Clearly the mesh bounding box has an associated *inside* and *outside*, thus there is no question as to how to solve the image problem. A problem arises if a user were to wish having a PMC boundary associated with an arbitrary object placed inside the mesh bounding box. Such a PMC boundary would be presumed to be of minute thickness. Because of the problem of image sides, it is unclear how to solve the image problem inside the FDTD mesh. Thus, PMC boundaries are to be restricted to the mesh bounding box surface. For future work it may be worthwhile to consider how PMC boundaries could be implemented in the interior of the mesh bounding box.

Absorbing Boundary Conditions

Absorbing boundary conditions (ABCs) are supposed to allow the modeling of fields in an unbounded region. Such modeling is particularly useful for examining the free space characteristics of antennas. We considered two kinds of ABCs, the first is an applied boundary condition, the second is based on the idea of an impedance matched material placed inside the mesh. We consider applied ABCs first.

The principle behind applied ABCs is that by using some function to estimate what the electric field components should be on the surface of the mesh boundary, electromagnetic fields should appear to be “absorbed” into the boundary surface. As a simple example, Taflove and Brodwin[62] presented an applied ABC that works by averaging adjacent electric field components inside the mesh. Clearly, in developing an applied ABC, a fundamental decision is deciding which function should be used to model the ABC.

We considered Mur’s[41] second approximation to absorbing boundary conditions. Taflove and Umashankar[64] indicate that Mur’s second approximation to absorbing boundary conditions is based on the two term Taylor series expansion of radiation boundary conditions (RBCs). Mur refers to Engquist and Majda[13] as presenting a similar ABC. We selected Mur's second approximation as we wanted an ABC that would fulfill a range of general uses. According to Mur, "The second and higher approximations are less subject to reflection problems, especially for fields grazing the outer boundary…" We anticipated that an ABC suitable for general use should be able to handle such grazing fields.

As an alternative to applied ABCs, we considered an absorbing boundary condition based on the idea of an impedance matched lossy material (MABC) that we place inside the mesh. This is not an applied absorbing boundary condition in the same sense as any applied ABC, but is incorporated with PEC type boundary conditions. The absorbing material is simply placed in layers against the PEC boundary surface, see figure 4.10. Section 4.5 presents a discussion of the material in some detail.

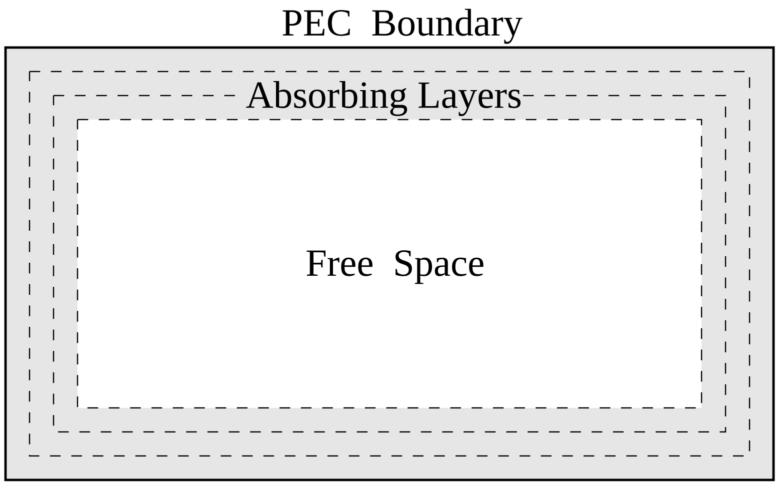


Figure 4.10: Matched Absorbing Boundary Conditions (MABC)

The use of lossy material to model absorbing boundary conditions has been used by researchers, for example Reineix and Jecko[51] used such absorbing boundary conditions for their modeling of patch antennas. Rappaport[49], as well as Rappaport and Bahrmasel[50] report the use of similar lossy material in FDTD analysis. From our point of view, Mur's ABC provided us with a standard of comparison for our MABC. If our MABC could be shown to be as good as Mur's ABC, then we would feel comfortable in using it.

4.4.3 Implementing Boundary Conditions

Next we briefly consider the implementation of boundary conditions. Since an array processor must execute the same instructions across all processors simultaneously, the most straightforward way to handle boundary conditions is to mask out processors alternately. Thus one part of the code calculates the interior of the problem while the boundary processors are switched off, and vice versa. Since the boundaries form a small percentage of the actual problem domain, this method is inefficient and can have a significant impact on the overall execution time. Since the majority of the processor elements will be idle when boundary conditions are handled, any time spent processing boundary conditions, directly degrades the average processor use figure.

As one might suspect, there are at least two caveats that will allow us to implement boundary conditions, and still be able to achieve high performance in our parallel implementation. First, if the time required to process boundary conditions can be made to be small in comparison to the time required to update field components, then the average processor use figure can still be made to be large. Second, if it can be shown that the relative overhead of handling boundary conditions becomes smaller as the problem size becomes larger, then we may still be able to realize a benefit in using a massively parallel machine, such as our MasPar system. In the following, we examine in an abstract way how each of the boundary conditions we presented was implemented.

The handling of PEC boundary condition requires that certain electric field components be set to zero. While masking is used as a mechanism to handle PEC boundary conditions, the handling amounts to assigning an initial value to field components in the mesh, and assuring that the field components associated with the PEC surfaces remain unchanged. This handling amounts to setting an active set that excludes the PEC field components. Clearly, in comparison to the time required to update interior field components, the time required to set up an active set is negligible. What this means is that the average processor use figure due to branching can still be good and that the use of PEC boundary conditions will not impact the overall performance in any significant way.

PMC type boundary conditions are taken care of by an exception handler. Before the finite difference equations are performed, field component values are fetched from memory. Processor elements that are handling PMC boundaries use an interior electric field component twice, appropriately handling the signs of the magnetic field component. The point is that a simple exception handler allows the same equations used to update interior field components, to update PMC boundary components. Since only a very simple exception handler is used, the use of PMC boundary conditions should not impact the overall performance in any significant way.

From an implementation viewpoint, Mur's absorbing boundary conditions are computationally intensive and because of that they present a problem. When Mur's ABCs are being handled, the vast majority of the array processor is left idle for a significant amount of time, representing a waste of computational resources. Rather than using an applied ABC, which is inherently inefficient, we need an absorbing boundary condition that does not rely heavily upon the technique of masking out processor elements. Use of MABC is particularly appealing to a parallel implementation and is considered to be a *natural fit* between the algorithm and the machine architecture.

Table 4.2 is a comparison of the performance of Mur's absorbing boundary conditions and the impedance matched lossy material (MABC). As a test model we performed an analysis similar to that presented by Mur[41]. Rather than being two dimensional, our analysis was three dimensional. All interior surfaces were made to be absorbing with either Mur's ABC or MABC. Excitation was applied by simply exciting one point in the FDTD mesh near one corner of the mesh.

Table 4.2: **Comparing ABCs**

Problem Size Cells	FDTD Cells (Mur)	CPU Time (Mur)	FDTD Cells (MABC)	CPU Time (MABC)
80x80x25	82x82x27	86.40sec	90x90x35	61.23sec
62x62x25	64x64x27	51.81sec	72x72x35	61.20sec

Since the MABC results are from a mesh that used an additional 10 layers for each dimension, a certain amount of overhead associated with MABC is seen. For smaller problems this additional overhead made the MABC analysis slower than an implementation using Mur's ABC. However, as the problem size grows, this overhead becomes negligible in comparison to the rest of the mesh. Eventually MABC outperforms Mur's ABC. The results in table 4.2 were obtained from the MP 1101, node 'goofy' at WPI.

In addition to the possible improvement in performance, the results from the analysis incorporating MABCs were noticeably better than those performed with

Mur's ABC. Based on these two observations, improved execution time and improved data quality, we decided to dismantle Mur's ABC. For future work it may be worthwhile to reconsider how and when it would be useful to use an applied ABC such as Mur's ABC.

4.4.4 Assigning Boundary Conditions

From a user's point of view, boundaries are assigned to surfaces of the FDTD mesh. A user might select certain parts of the outer surface of the FDTD mesh to model a PMC boundary condition, leaving the remaining outer surfaces as PEC type. PEC and PMC surface types were presented in section 4.4.2

From an algorithmic point of view, to properly handle applied boundary conditions as presented here, we are primarily concerned with the handling of electric field components. For PEC surfaces, we assign electric field components tangent to the PEC surface a value of zero. For PMC surfaces we use the image method to determine the magnetic field components needed to update electric field components. Likewise, for applied ABCs, it is the electric field components that we are concerned with properly updating by using some arbitrary function. This observation, that the proper handling of electric field components on the surface of the FDTD mesh is responsible for the modeling of surface types is important as it raises a fairly obvious rhetorical question, that is how to properly handle adjacent boundary conditions that are of different types. Consider figure 4.11, which shows the surfaces of Yee Cells that have been assigned different types of boundary conditions.

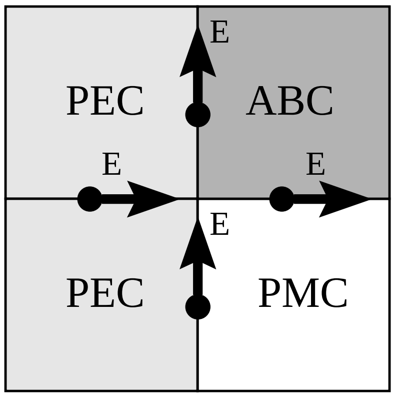


Figure 4.11: Adjacent Boundary Conditions

The problem illustrated by figure 4.11 is associated with the interface between different boundary condition types. Because electric field components are defined to be on the edges of the Yee Cell, it is not immediately obvious how to properly handle the electric field updates. To properly handle electric field components, an order of precedence was defined so that when different boundary condition types are adjacent, the boundary type with precedence will the used to update the electric field component along the common edge. The order was set by reviewing situations in which a user might assign each boundary condition type.

PEC surfaces are given precedence over PMC and ABC surfaces. Figure 4.12 shows cases when PEC is adjacent to PMC. For the case when PEC and PMC are adjacent at a corner, the PEC surface should be handled so that it is modeled as extending continuously across the PMC boundary. For the case when PEC and PMC do not meet at a corner, it is important that the edge between the two types appear as PEC, otherwise the PEC edge will not be modeled as straight. This second case is common for the modeling of strip-lines. For this same reason, whenever PEC is adjacent to ABC, the PEC handling should have precedence.

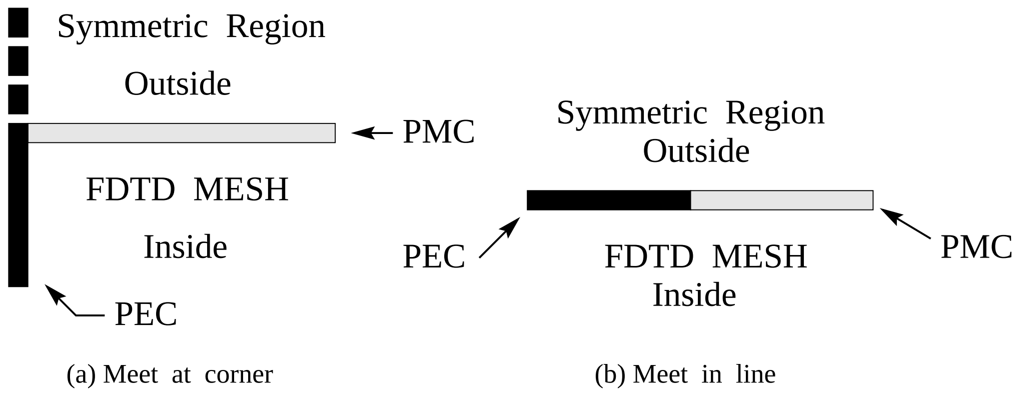


Figure 4.12: PEC abutting PMC

One situation in which PMC and ABC surfaces meet is at a corner. In such a situation the applied ABC should appear to be continuous across the PMC boundary, thus the ABC is given precedence over PMC. This point is for your information only, since our applied ABC has been dismantled. Table 4.3 summarizes the order of precedence for the surface types we considered. PEC type is given first and is considered to have the highest precedence, followed by ABC, then PMC.

One last comment, since some kind of boundary conditions must be specified for every part of the outside surface of the FDTD mesh, it was decided that a default boundary condition would be assigned before the user assigns boundary conditions. Wherever a user specifies a boundary condition, the default is overridden. Currently, if a user does not specify a boundary condition type on any

Table 4.3: **Precedence of Surface Types**

Entry	Surface Type
1	PEC
2	PMC
3	ABC

part of the outer surface of the FDTD mesh, the PEC type will be assigned.

4.5 Conductive Materials

This section provides a more fundamental discussion of material properties used to describe conductive materials. A special case is discussed that leads to a non-physical material suitable for a simple boundary condition.

4.5.1 Electric Conductivity

Shen and Kong[57] provide a useful discussion of plane waves in dissipative media. As is usual in this kind of analysis we assume complex sinusoidal excitation and use the harmonic form of Maxwell's equations. In this analysis we assume that σ and σ_m are positive, real valued constants. A conductor is characterized by conductivity σ and is governed by Ohm's law. For isotropic conditions Ohm's law states that the conduction current is;

$$\vec{J}_c = \sigma \vec{E}$$

The unit of conductivity σ is Siemens per meter, alternatively called mhos per meter. From Ampère's law;

$$\nabla \times \vec{H} = j\,\omega\,\vec{D} + \vec{J}$$

We see that the current density $\vec{J}$ can in fact embody two kinds of current, source current $\vec{J}_o$ and conduction current $\vec{J}_c$, thus;

$$\vec{J} = \vec{J}_c + \vec{J}_o$$

For an isotropic dielectric we apply the following constitutive relationship: $\vec{D} = \epsilon\,\vec{E}$. Thus, Ampère's law can be rewritten as follows,

$$\nabla \times \vec{H} = j\,\omega\,\epsilon\,\vec{E} + \sigma\,\vec{E} + \vec{J}_o$$

to include the conductivity as a complex term;

$$\nabla \times \vec{H} = j\,\omega\,\left(\epsilon - j\,\frac{\sigma}{\omega}\right)\vec{E} + \vec{J}_o$$

Thus a new permittivity is defined to contain an imaginary part derived from the material conductivity.

$$\tilde{\epsilon} = \epsilon - j\frac{\sigma}{\omega}$$

Since $\tilde{\epsilon} = \epsilon_o\tilde{\epsilon}_r$, we can say that;

$$\tilde{\epsilon} = \epsilon_o\left(\epsilon_r - \frac{j\sigma}{\epsilon_o\omega}\right) \tag{4.5}$$

As pointed out by Luebbers[30], when applying the FDTD method to a material, the usual approach is to specify a real permittivity and a constant electric conductivity. The corresponding frequency dependence is given by equation (4.5). Luebbers discusses alternative methods for characterizing lossy dielectrics that the active researcher may have some interest in.

Lastly, in a conductive media devoid of sources, we can write Ampère's law as follows;

$$\nabla \times \vec{H} = j\,\omega\,\tilde{\epsilon}\,\vec{E}$$

4.5.2 Magnetic Conductivity

Although Shen and Kong don't present this next derivation, we can rewrite Faraday's law, equation (E.2) in harmonic form and follow the same thought process.

$$\nabla \times \vec{E} = -j\,\omega\,\mu\,\vec{H} - \sigma_m\vec{H}$$

In this case the magnetic conductivity term is incorporated into the permeability term.

$$\nabla \times \vec{E} = -j\,\omega\left(\mu - j\frac{\sigma_m}{\omega}\right)\vec{H}$$

$$\nabla \times \vec{E} = -j\,\omega\,\tilde{\mu}\,\vec{H}$$

While there may be speculation surrounding the idea of magnetic conduction current, allowing such a phenomenon in our simulation proves to be useful. Thus a new permeability is defined as;

$$\tilde{\mu} = \mu - j\frac{\sigma_m}{\omega}$$

$$\tilde{\mu} = \mu_o \left(\mu_r - \frac{j\ \sigma_m}{\mu_o\ \omega} \right)$$

4.5.3 A Special Case

The intrinsic impedance of such an isotropic media can now be expressed as a complex quantity. Substituting into the impedance formula shows that in general the intrinsic impedance is dependent on ω.

$$\eta = \sqrt{\frac{\tilde{\mu}}{\tilde{\epsilon}}} = \left(\frac{\mu - j\frac{\sigma_m}{\omega}}{\epsilon - j\frac{\sigma}{\omega}} \right)^{\frac{1}{2}} = \left(\frac{\omega^2\mu\epsilon + j\omega\mu\sigma - j\omega\epsilon\sigma_m + \sigma\sigma_m}{\omega^2\epsilon^2 + \sigma^2} \right)^{\frac{1}{2}}$$

From the previous expression, a special case can easily be noted. If $\mu\sigma = \epsilon\sigma_m$ then the complex terms cancel. This special case is better stated as a ratio,

$$\frac{\sigma_m}{\sigma} = \frac{\mu}{\epsilon} \tag{4.6}$$

Assuming the special case of equation (4.6), we continue simplifying the intrinsic impedance equation.

$$\eta = \left(\frac{\omega^2\mu\epsilon + \sigma\sigma_m}{\omega^2\epsilon^2 + \sigma^2} \right)^{\frac{1}{2}} = \left(\frac{\mu}{\epsilon} \frac{\left(\omega^2\epsilon + \frac{\sigma\sigma_m}{\mu}\right)}{\left(\omega^2\epsilon + \frac{\sigma^2}{\epsilon}\right)} \right)^{\frac{1}{2}} = \sqrt{\frac{\mu}{\epsilon}}$$

We discover that assuming the special case not only makes the intrinsic impedance real valued but also sets the impedance of the material independent of ω. In fact if $\epsilon = \epsilon_o$ and $\mu = \mu_o$ we can define a conductive media that has the same intrinsic impedance as free space.

Such a conductive media can be used to implement an absorbing boundary condition. As Rappaport[49] as well as Sacks, Kingsland, Lee and Lee[53] point out, the following condition is referred to as the free space *impedance matching* condition.

$$\frac{\mu_o}{\epsilon_o} = \frac{\mu_1}{\epsilon_1} = \frac{\sigma_m}{\sigma}$$

The *impedance matching* condition corresponds to maintaining constant frequency domain wave impedance. Such a case implies no reflections for normal wave incidence.

Returning to Shen and Kong[57], conductive media can be classified by severity of loss according to what is called the loss tangent which is $\sigma/\omega\epsilon$. A highly conductive media will have $\sigma/\omega\epsilon \gg 1$, in which case the skin depth can be approximated as;

$$\delta_p \simeq \sqrt{\frac{2}{\omega\mu\sigma}} \tag{4.7}$$

It is important to remind the reader that this document is not supposed to be an in-depth presentation of electromagnetic field theory, let alone the FDTD method itself. The purpose of this chapter is to present the FDTD method as an algorithm, the purpose of this complete document is to present a unique application that uses the FDTD method. For more details regarding lossy media, the reader is referred to Shen and Kong[57].

4.6 The FDTD Algorithm

This section serves as a summary to this chapter by formally presenting the FDTD method as an algorithm. Figure 4.13 provides a flowchart of the algorithm that we used to implement the FDTD method. The algorithm starts by performing an initialization operation. Initialization is performed by first determining the mapping of data structures, then loading the standard mesh into the FDTD mesh, estimating material properties, adjusting mapping flags so that the simulation will properly handle boundary conditions, open files for data, and configure simulation parameters according to user requests.

After initialization the algorithm enters the main loop of the flowchart. After each iteration of handling boundary conditions, updating fields by using the finite difference equations, and incrementing the time step counter, a test is performed to see if the simulation has been completed. At appropriate times the loop takes a detour to write data to files. While the FDTD equations used to update electromagnetic field values can be considered to be the core of the algorithm, this aspect is actually one of the easier parts of the algorithm to implement. In comparison, the mechanisms used to write data to files is far more complicated than the FDTD equations. The mechanisms used to initialize the simulation are even more complicated than the mechanisms used to write data to files.

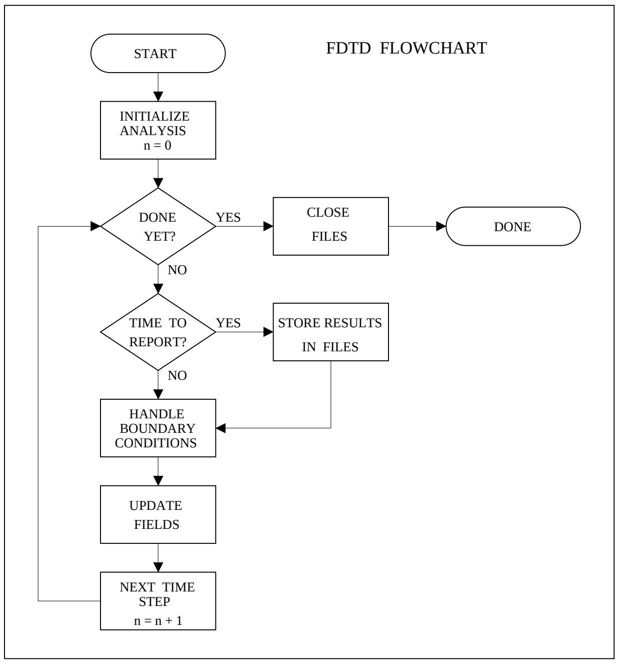


Figure 4.13: Flowchart of FDTD Algorithm

Determining the time complexity of the FDTD algorithm is relatively straightforward. While the task of initializing the analysis is complicated, it is performed only once per FDTD analysis, thus we can assign the initialization a time complexity of $O(1)$[1]. We will assume that the time to handle boundary conditions is small in comparison to the time required to update the fields. We will further assume that as the problem size becomes larger, the time required to update field values, becomes even larger in comparison to the time needed to handle boundary conditions. Lastly, we will assume that data is rarely written to files. After making these assumptions and examining figure 4.13, we arrive at the conclusion that the time complexity of the overall algorithm is primarily dependent on the time needed to update the electromagnetic field vales.

Since there are N_m boxes in a FDTD mesh, the time to update the electromagnetic fields once is $O(N_m)$. Since N_{step} time steps are required in an analysis, we will assign the FDTD algorithm an overall time complexity of $O(N_m\ N_{step})$. In section 5.5.2, the time complexity is examined by using experimentally measured data.

This chapter presented the FDTD method as an algorithm. Topics central to the FDTD method such as the FDTD mesh itself, the Yee Cell, time stepping,

[1]For a review of time complexity analysis, and the use of 'big-oh' notation, see section 2.6.

stability and accuracy, and mesh boundary conditions were discussed. The idea of using a Fast Fourier Transform to perform frequency domain analysis was just touched on. The form of the finite difference equations used to perform the FDTD method was introduced. The actual finite difference equations used to perform the FDTD method are derived in appendix E. A discussion of conductive materials was presented along with a special case that we use to derive a special absorbing material that we can use to model absorbing boundary conditions. Lastly, this chapter presented the FDTD method in the form of an algorithm, complete with a flowchart and a discussion of its time complexity.

Chapter 5

FDTD Implementation

In this chapter we examine the implementation of the finite difference time domain code. Perhaps the most important decision that was made during the development of the FDTD code was how to map the FDTD mesh to PE array memory. Because of its importance, this topic is addressed first. Next, a general outline of the FDTD code is presented. Lastly, the performance of the code is examined in detail.

5.1 The FDTD Mapping

This section introduces the mapping used to store a three dimensional FDTD mesh. The first section presents useful directional terms. The second section presents how a three dimensional mapping is constructed by layering two dimensional hierarchical maps. The third action introduces data structures used to manage the mapping.

5.1.1 Directional Terms

Figure 5.1 illustrates the terms used to describe directions relative to the FDTD mesh. While these terms were actually introduced back in chapter 1, for your convenience are presented here as well. The terms *north*, *south*, *east*, *west*, *up*, and *down* were defined as an attempt to ease the task of the writer as well as reader. For example the phrase "Take a specified value from the nearest Yee cell found in a direction of increasing i index values but constant j and k index values..."

is better stated as "Take a specified value from the nearest cell to the east..." Clearly the simple idea expressed is more clearly stated in the second sentence fragment.

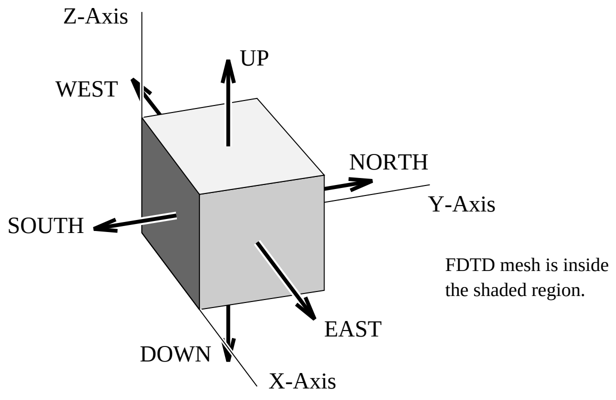


Figure 5.1: The Six Basic Directional Terms and Coordinate Axes

In implementing the mapping and associated virtualization on the MasPar, coordinate axes were aligned so that the terms *north*, *south*, *east* and *west* would agree with terms established by MasPar literature. Also index values were coordinated so that *up* refers to increasing index values and *down* refers to decreasing index values. These simple notions saved much confusion. Implementation of the three dimensional mapping is introduced next.

5.1.2 Introduction to the Mapping

Figure 5.2 provides an illustration of how a 4x4x4 mesh is stored in a hypothetical 2x2 PE array. It is suggested that as you read the following, refer to the figure as needed. Note that individual mesh boxes are referred to by using a simple $[i, j, k]$ notation. To produce the mapping, a three dimensional mesh is first divided into sheets. A sheet is simply a two dimensional array of mesh boxes that correspond to a single k index value. Each sheet is mapped to a virtual two dimensional processor array by assigning each virtual processor a single Yee cell from the sheet. In this example the virtual array processor has sixteen PEs, numbered from 0 to 15.

It is important that the reader remember, that in this example there are only four in the processor array, and that a virtual processor array is only a handy abstract concept that we use. The virtual two dimensional processor array is mapped to the physical PE array by using a mapping equivalence. To illustrate this equivalence, the virtual array is first divided into squares that we call *sub-sheets*. Each sub-sheet is assigned to a PE. In figure 5.2, the PEs are numbered

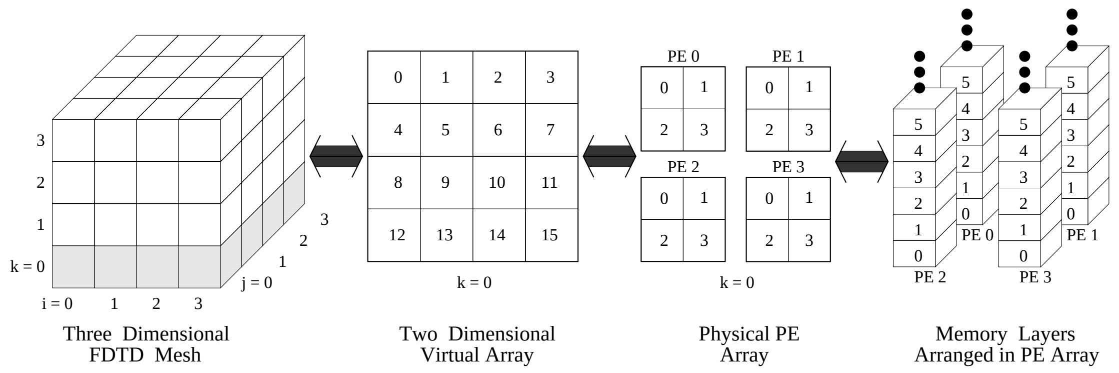


Figure 5.2: Introduction to Layered 2D Hierarchical Mapping

PE0 to PE3, and each PE is assigned four Yee cells. To get a better sense of this, consider the following analogy; If you fold a sheet of paper into half and then fold into quarters and so forth, then when you unfold the sheet and lay it flat, the creases in the paper will mark boundaries between squares. A pattern of squares is selected that will best match a two dimensional array of PEs.

Except for sub-sheets near the edges of a sheet, each sub-sheet will have the same number of Yee cells assigned. The right-most part of figure 5.2 shows four Yee cells are mapped in each PE to memory layers, numbered 0 thorugh 3. The symbol *lps* is an acronym for the phrase *layers per sheet*. We use this number to describe the number of memory layers that are needed per PE to store a sub-sheet of Yee cells.

It is important to realize that by dividing each sheet into sub-sheets, a hierarchy is formed. Each sheet is a two dimensional array constructed from a two dimensional array of smaller two dimensional arrays(sub-sheets). Because of the way that each sheet is mapped, the sheet mapping is referred to as *two dimensional hierarchical*. Since the overall mesh is constructed by layering these two dimensional mappings, we state that for the FDTD solver, a three dimensional mesh is mapped according to a *layered two dimensional hierarchical mapping*.

Unlike the mesh generation algorithm, where in most cases each mesh box can independently decide if it is inside or outside a given solid object, the FDTD solver performs finite difference equations that require data from adjacent Yee cells. Note that the term *adjacent* is used to describe mesh boxes, or equivalently Yee cells that share a common surface in the FDTD mesh. Once mapped to PE memory, adjacent Yee cells may actually be stored in different PEs.

Since each sheet is mapped to memory in a similar fashion, moving data up and down[1] the mesh is simple. The mesh box $[i, j, k]$ for instance is in the same PE as mesh box $[i, j, k-1]$, but is *lps* layers higher in memory. Moving data across the mapping is not as simple however. If a mesh box is on the edge of a sub-sheet, to send data to an adjacent mesh box on a different PE, the use of interprocessor communications is required. Performing interprocessor communications requires time, so naturally, we try to minimize the use of interprocessor communications. The time required to perform communications between adjacent PEs is roughly equivalent to half the time it takes a PE to perform a floating point addition.

An important feature that a hierarchical mapping has over a cut-and-stack mapping, like the two dimensional cut-and-stack mapping presented by Pickering and Cook[46], or the one dimensional cut-and-stack used by the mesh generation code, is that a hierarchical mapping can store entities that are adjacent in the problem domain, in the same PE. What this means for an application like the FDTD solver that requires data to be shared between adjacent Yee cells, is that use of a hierarchical mapping will produce a program that uses interprocessor communications less often, than if a cut-and-stack mapping were used.

Like all design decisions there is an associated trade-off. On the down side, for certain problem sizes this mapping will not make the best use of all the PEs. For certain problem sizes all the PEs will be able to participate in the simulation, but for other problem sizes a sizable fraction of the PE array will be unusable and will represent wasted resources. On the plus side however, such a mapping makes the transfer between adjacent Yee cells very simple, and actually minimizes the use of communications between PEs. For such an application, these benefits are particularly appealing and were the reason why this mapping was selected.

5.1.3 Mapping Data Structures

To be able to perform the finite difference equations, it was decided that key information associated with the mapping would be stored in data structures. The data structures are used like *look up tables*, that is at any moment of the simulation, to determine how to find data in the mapping, or how to handle boundary conditions, only a simple reference to the data structures is needed. These data structures, summarized in table 5.1 are set up during program initialization, and remain unchanged until the program terminates. To set up these data structures, macros were written to identify special cases associated with the mapping.

[1]See section 5.1.1 for an introduction to these terms.

Table 5.1: **Mapping Reference Elements**

`mark[k]`	–	Array stored in Processor Array Memory, describes at the mesh cell level whether a given Yee cell is against the overall mesh boundary.
`b_control[k]`	–	Array stored in Processor Array Memory, indicates how to update electric field components.
`eq_choice[l]`	–	An array of data structures, stored in ACU memory. This array globally describes if and how the X-NET will be used for communications.

The `mark` array is particularly useful in identifying the boundaries of the overall mesh. The `b_control` array is an array of flags used to indicate how to process field components. It is by use of the flags in this array that we simulate PEC or PMC surfaces. The `eq_choice` array is used to identify how to transfer data between Yee cells. To make the actual transfer easier, a set of macros were written to handle array indexes. These data structures are used together in concert to give the appearance of the intended mapping.

5.2 Implementation Outline

The following amounts to being a high level guided tour of the FDTD code. While this is high level, it still contains many details. It is suggested that before your first reading of this section, glance through the headings to get a sense of the bigger picture. The FDTD code goes through a long drawn out initialization procedure that takes a bit of space to describe. The finite difference equations actually represent the core of FDTD method and are derived in appendix E. The discussion of the floating point performance in section 5.6 actually lists the finite difference equations.

Program execution starts in the DPU. The very first thing the program concerns itself with is checking if it has a valid mesh file name. The point of a mesh file is that an FDTD simulation should be performed without any direct interaction with the user. To make things simple, the format of the mesh file was designed to contain all the information needed to perform a simulation.

If the user does not provide a file name in the command line used to start the program, the user will be asked to provide a file name. Before a FDTD simulation can begin, the program must be initialized. Once initialized, the rest of the FDTD simulation is relatively straight forward. The program just loops through the finite difference equations to advance the time step with each iteration. At regular intervals, data is extracted from the mesh, and is stored in files.

5.2.1 FDTD Initialization

The initialization procedure centers on gathering data stored in a mesh file and processing the data so that the simulation can proceed quickly. A mesh file is broken down into five parts that are given in order. The initialization procedure was organized in a manner that directly mirrors this file format. The first part of a mesh file is referred to as the mesh file header, which is just three lines that describe how many mesh boxes are contained in the mesh, and what the size of each mesh box is. The second part of a mesh file is referred to as the commands section which describes how the simulation will proceed, what type of excitation function to use, and what kind of output data will be produced. The third part contains the standard mesh produced by the mesh generator program. The fourth part is the materials section which describes the material properties of every material used in the simulation. The last section contains data used to describe mesh boundary conditions.

The practice of opening and reading files in the front end, then passing data in formatted data structures to the DPU appears to provide the most efficient way to load serial data into the MasPar system, and was used by this implementation. The mesh file is actually loaded by several functions. Each of these functions takes its turn once, to process specific information in the mesh file. As you might suspect, there is a parser to load the commands section. There is a special loader that is used to move the standard mesh into PE array memory. Another loader gathers materials from the mesh file to produce a *materials table*. One step of the initialization is to take the materials table and standard mesh, and produce all the coefficients needed to perform the FDTD method. The last step of the initialization is to set up flags to represent boundary conditions.

Opening a Mesh File

The mesh file is opened in the front end by a small function called `open_mesh_xyz1`. The first few lines of the standard mesh contain the number of mesh boxes that are along the x, y, and z coordinates, as well as the coordinates of the point $[0, 0, 0]$, as well as the mesh discretization sizes. These lines are read in the front end and are passed to the DPU by calling the function `define_const`. With the size of the standard mesh and the dimensions of the PE array known, the DPU can set the first constants associated with the selected memory mapping. Think of a standard mesh as being made up of many sheets that are stacked, in a manner similar to the stacking of paper. To prepare for storing the mesh, we need to consider how we can cut each sheet into sub-sheets, such that each sub-sheet will fit into a single PE. Such a mapping is referred to as hierarchical, see section 5.1.2 for a graphical introduction to this mapping. The variable `lps` (layers per sheet) defines the number of PE memory layers that are to be associated with a single mesh sheet. Another variable associated with the mapping is the total number of memory layers that we will associate with storing the mesh in PE memory. This variable, referred to as `alayers` is important as it tells us how many mesh sheets need to be stored in PE memory.

Simulation Directives

The `command_parse` function is called to load the user defined simulation directives. The directives specify such important simulation parameters as the output file types, output file names, the simulation time step size, the number of time steps to simulate, and the type of excitation required. For an explanation of how a user enters simulation directives, see appendix section F.5.

Before returning to `open_mesh_file`, the `command_parse` function performs a quick check to ensure that the minimum requirements for a simulation have been specified, that is the time step size, number of time steps to simulate, some kind of output type and file name, and some kind of excitation. Given that the required items are present, a pointer to a data structure called `RData` that summarizes the simulation directives is returned so that the DPU can copy out the data from the front end.

Loading the Standard Mesh

The next step of program initialization is to load the standard mesh from the mesh file into PE array memory. The standard mesh is represented by a sequence of integer values, where each integer corresponds to exactly one mesh cell. The value of each integer is used to indicate the material type of each mesh box.

Before the standard mesh can be loaded into memory however, PE array memory must be allocated. PE array memory is actually allocated in three steps. The first memory allocation step involves the arrays listed in table 5.2. In the table, array names are each listed with a short description. The column "Element Size" gives the size in bytes of a single entry in each of the allocated arrays.

Table 5.2: First Allocation in PE Array

Array Name	Array Description	Element Size
`b_control`	boundary conditions marker	4 bytes
`mark`	bounding box marker	1 byte
`found`	flags a match for material	1 byte
`ID`	integer material index	1 byte

The standard mesh is loaded one sheet at a time into front end memory and then is moved to PE memory. To be able to transfer the data, the number of memory layers associated with the actual transfer must be calculated. Because the `blockIn` command is used to perform the actual transfer from the front end to the DPU, memory layers must be allocated for the transfer in multiples of four bytes.

The `setup_load` function is called to allocat memory in the front end for loading a single sheet of the standard mesh at a time, as well as to set up a few constants needed to configure the loader function `load_sheet`. Following the call to `setup_load`, each call to `load_sheet` in the front end loads the next mesh sheet from the mesh file. Thus a mesh is loaded into the array `ID` by repeatedly calling `load_sheet` to load a single sheet at a time and then calling `blockIn` to copy the mesh sheet from the front end to PE array memory.

After the sequence of integers is loaded into PE memory, each integer is matched with its corresponding entry in the materials property table.

Initializing the Materials

The function `material_file_reader` next loads the entire materials table from the mesh file, into front end memory. The function `check_mat_table` is called to move the materials table to the DPU. Checks are performed to ensure that for each material type present in the standard mesh, at least one set of material properties is actually present in the materials table. Also, a check is made to ensure that the real parts of the relative permeability and relative permittivity are not equal to zero. The values of relative permeability and permittivity are each multiplied by the appropriate physical constant to yield the actual values of permeability and permittivity. A stability criterion number is calculated as an approximate test to see if the simulation will be stable.

Handling Materials

Each mesh cell integer used to describe a mesh cell simply is a reference into the materials table. With the materials table loaded and checked, PE array memory is reconfigured by calling `field_memory_allocation`. Since the boundary control array `b_control` has not yet been assigned any values, we temporarily transfer the `ID` values to the `b_control` array and free up the `ID array`. We also free up the `found` array. Memory is allocated for KE_l, KE_c, KH_l, and KH_c finite difference coefficients, as well as arrays for storing electric and magnetic field components. All these new arrays are allocated to store single precision floating point numbers.

The last block of memory allocated by `field_memory_allocation`, is allocated in ACU memory. The allocated memory, named `eq_choice` is configured by `setup_eq_choice` to be a look-up table for managing the finite difference equations used to implement the hierarchical mapping.

To calculate the finite difference coefficients `KH` and `KE`, the mesh cell properties are first temporarily assigned to the `KH` and `KE` arrays. As described in section 4.1.2, adjacent mesh cell property values are averaged together to define nodal material properties. The average values are temporarily stored in electric and magnetic field component arrays. Using the equations presented in appendix E, the `KH` and `KE` finite difference coefficients are calculated by using the nodal material property values. With the `KH` and `KH` arrays defined, the materials table in ACU memory can be freed up and the electric and magnetic field component arrays are cleared by filling them with zeros.

Setting Boundary Conditions

As section 4.4.4 points out, as far as the user should be concerned, boundary conditions are assigned to surfaces of the FDTD mesh. The abstraction presented to the user is that the mesh is made up of many uniformly shaped boxes. Note that no sense of the Yee Cell is directly presented to the user. Inside the FDTD solver however, boundary conditions are handled by manipulating electric field values associated with the edges of Yee Cells. The details of how boundary conditions are actually handled is presented in section 4.4.3. except the case when boundary conditions of different types are adjacent, the assignment of boundary conditions is simple. Table 4.3 provides an order of precedence for resolving situations when different boundary types are adjacent.

The function `set_b_control` is called in the DPU to set the flags in the array `b_control`. The array `b_control` serves as a look-up table by the code that actually handles boundary conditions. The following is a list of the steps that are followed in assigning the flags to the array `b_control`.

1. Memory is made available in the mesh to temporarily store descriptions of surface types. By placing a value in one of these memory locations, it is said that "a surface has been assigned to the mesh."
2. Default surface types are assigned to the surfaces of the mesh that are associated with the mesh bounding box.
3. The description of surface types provided by the user is compiled into a list that will make sense in the context of the mesh.
4. The compiled list of surface types is assigned to the mesh, overwriting any default surface types that may already be present.
5. By examining the assigned surface types and following the order of precedence for surface types, flags are assigned to the array `b_control`.

Note that while the FDTD solver has default boundary types defined, the user is free to assign boundary types. Of course, user defined boundary types over-ride the default boundary types.

Final Memory Allocation

Table 5.3 summarizes the parallel memory allocation for the FDTD simulator, when initialization is complete. Note that the table groups variables together in a meaningful way, for example the `Ex`, `Ey`, and `Ez` arrays are used to store electric field intensity values. Together, these three arrays describe the entire electric field being simulated. The allocation value given on this table describes the amount of memory required to properly store the values associated with a single Yee cell. With the flags in the `b_control` array defined, the program initialization is almost complete.

Table 5.3: **Final Allocation in PE Array**

Array Names	Composite Description	Bytes per PE
`Ex`, `Ey`, `Ez`	Electric Field Intensity	12
`Hx`, `Hy`, `Hz`	Magnetic Field Intensity	12
`KElx`, `KEly`, `KElz` `KEcx`, `KEcy`, `KEcz`	Difference Coefficients for Updating Electric Fields	24
`KHlx`, `KHly`, `KHlz` `KHcx`, `KHcy`, `KHcz`	Difference Coefficients for Updating Magnetic Fields	24
`b_control`	Boundary Control Flags	4
`mark`	Mesh Limits	1
	PE Allocation per Yee Cell:	77

We can think of the layered hierarchical two dimensional mapping as storing Yee cells in parallel memory layers, where each memory layer has at most `nproc`[2] Yee cells. Each memory layer is 77 bytes thick. This number will be important in section 5.3.1, where we more closely examine parallel memory allocation.

Finishing Initialization

The last step in program initialization is calling `setup_timewise` to define constants that are associated with the user selected field excitation function, and then calling either `sampler_prep` or `setup_data_output` to configure the output handlers.

[2]`nproc` is the number of PEs in the DPU

5.2.2 Simulation Run Time

After everything that needs to be initialized has been taken care of, the FDTD simulation begins. The code that actually performs the finite difference equations is stored in a file called '`fdtd_solve_3d.m`'. The function '`FDTD_solver`' is responsible for keeping track of simulated time, it calls the functions that perform the finite difference equations and calls the functions to output data as necessary. The function `update_Elayer` performs the finite difference equations used to update the electric field components. The function `update_Hlayer` performs the finite difference equations used to update the magnetic field components. The details of how data is output is described in section 5.2.3.

5.2.3 Producing Output Data

The FDTD code provides two mechanisms for the output of data from a simulation. The first mechanism is handled by the function '`sampler`', it can take samples from as many as two electric field components in the mesh, and will do so at every time step. Sampled data is formatted into files that can be immediately processed by a fast Fourier transform. No special processing is performed to handle data that is sampled by the first mechanism. At every time step, electric field components are written to files.

The second mechanism is handled by the function '`output_data`', it is used to report all the electric field components from a range of specified sheets in the FDTD mesh, at regular intervals of simulated time. The data produced by this second method is formatted into files that are readable by Patran. Patran is the tool that we use to visually post process output data. Because this second mechanism is capable of producing large amounts of data that all must be properly formatted, this second method of producing data could potentially impact the performance of the FDTD simulation. To minimize this problem, it was decided that the work associated with transforming raw data into formated files would be performed asynchronously in the front end. To be able to output data, the DPU must pause from the simulation so that it can estimate field component values, and then it sends the data produced directly to the front end. Note that by not having the DPU format data into files and not requiring the DPU to directly handle file I/O, the DPU can immediately return to the task of performing the simulation.

Next we consider how asynchronous processing is organized, please refer to figure 5.3. When the DPU is ready to send data to the front end, it proceeds to the sync-lock mechanism. What happens next depends on what the front end is up to. If the front end is able to format data and write files before the DPU produces new data, the front end will be idle when the DPU arrives at the sync-lock. In such a situation, data is immediately transferred to the front end, and the DPU returns immediately back to performing the simulation. If however the front end is not able to "keep up" with the DPU, the DPU will have to wait until the front end is ready to accept new data. The sync-lock mechanism guarantees two things, first that the DPU will only transfer data when the front end is ready to receive new data, and secondly that the front end will not begin to process data until it has actually be transferred from the DPU. The net result is that one process will be constantly busy, leaving the other process idle at regular intervals. Determining which process experiences idle time is related to both mesh size, the number of mesh sheets that are reported each time output is produced, and the frequency that data needs to be produced.

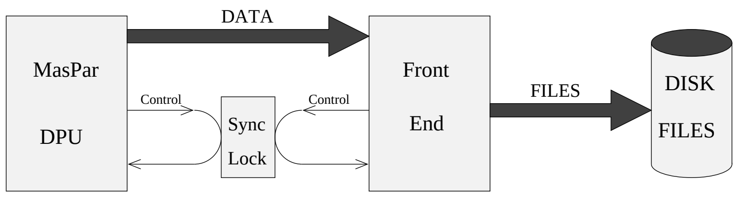


Figure 5.3: Asynchronous Processing Model

5.2.4 Closing the Simulation

Before exiting, any files that are open are closed. If Patran files were produced, the front end is instructed to shut down asynchronous processing and return full process control to the DPU. Lastly, the run time statistics are reported,

5.3 Parallel Memory Use

Table 5.4 summarizes the most important variables that are associated with determining the memory requirements of the mapping being used to store the FDTD mesh in PE array memory. The variables `nxproc` and `nyproc` are associated with the actual size of the PE array, see section 1.5.2 for details. The variables N_x, N_y, and N_z describe the size of the standard mesh, see section 4.1.1.

As discussed earlier in section 5.1.2, each sheet from the FDTD mesh is divided

Table 5.4: **Parallel Memory Allocation**

Variables	Description
`nxproc`, `nyproc`	columns, rows in PE array
N_x, N_y, N_z	size of standard mesh
`nxbox_pe`, `nybox_pe`	size of sub-sheet
lps	memory layers per sub-sheet
`nlayers`	total number of memory layers
`mem_lay_size`	allocation for a memory layer
`mem_overhead`	excess allocation required

into sub-sheets, where for each sheet, each sub-sheet is assigned to a single PE. Each sub-sheet consists of a two dimensional array of mesh boxes having `nxbox_pe` columns and `nybox_pe` rows. The variables `nxbox_pe` and `nybox_pe` are determined by taking the ceiling integer values of the following ratios:

$$\texttt{nxbox\_pe} = \left\lceil \frac{N_x + 1}{\texttt{nxproc}} \right\rceil \tag{5.1}$$

$$\texttt{nybox\_pe} = \left\lceil \frac{N_y + 1}{\texttt{nyproc}} \right\rceil \tag{5.2}$$

The variable *lps* describes the density of the mapping of mesh sheets to PE array memory, *lps* indicates exactly how many memory layers are needed to store each mesh sub-sheet in a PE. The variable *lps* is simply equal to the product of `nxbox_pe` and `nybox_pe`, or rather:

$$lps = \texttt{nxbox\_pe} \cdot \texttt{nybox\_pe} \tag{5.3}$$

Lastly, the variable `nlayers` describes the total number of memory layers needed to store a mesh in PE array memory. `nlayers` equals to the product of the sheet mapping density and the number of sheets in the mesh, plus one mesh layer or:

$$\texttt{nlayers} = lps \cdot (N_z + 1) \tag{5.4}$$

5.3.1 PE Memory Allocation

When discussing memory allocation in the PE array, it is important to remember that while we talk of the allocation on each individual PE, the allocation must always be uniform for all PEs. For the selected mapping, the variable `mem_lay_size` describes the amount of memory in bytes that each PE will use to store a single mapped memory layer. Section 5.2.1 describes in detail what the contents of each memory layer is and gives the actual number of bytes consumed each memory layer. For the FDTD mapping the value of `mem_lay_size` was found to be 77 bytes.

The following expression gives the total number of bytes that each PE will be allocated when an FDTD mesh is stored in PE array memory. The variable `mem_overhead` is simply the required memory that is not directly associated with the mesh. Currently there is an overhead of approximately 24 bytes required for the mapping.

$$\text{PE\_allocation} = \texttt{mem\_lay\_size} \cdot \texttt{nlayers} + \texttt{mem\_overhead} \tag{5.5}$$

Appendix section C.1.3 presents a discussion, showing how you can estimate the amount of parallel memory that you can use in your DPU. For a MP 1101 with 16K of RAM per PE for example, when the PE array is fully allocated, 15137 bytes of memory is consumed by each PE. Given that a machine has '`PE_mem`' bytes of memory available per PE, we can estimate that the total number of memory layers available to your mapping will be:

$$\text{Memory\_Layers} = \left\lfloor \frac{\texttt{PE\_mem} - \texttt{mem\_overhead}}{\texttt{mem\_lay\_size}} \right\rfloor \tag{5.6}$$

In the previous equation, the $\lfloor\ \rfloor$ symbols refer to the mathematical floor function. For a MP 1101, 196 memory layers are available using this mapping. With this many memory layers available, we can just fit a 95x95x20 mesh, which has $lps = 9$ and requires 180 memory layers.

5.3.2 Processor Use Figure Due to Mapping

The processor use figure due to mapping (PUFM), introduced in section 1.7.3, describes how well a given mapping does at fitting a problem "into" the processor array. For a static mapping, PUFM can always be calculated from what is known about the problem size and the PE array size. Using the layered, two dimensional hierarchical mapping to store a mesh having sheets N_x boxes wide and N_y boxes long, this mapping models a virtual PE array that has lps times as many PEs as the actual PE array.

Since PUFM is just the ratio of the number of virtual processors used, to the total number of virtual processors available in the mapping, we express PUFM below. Note that `nproc` is equal to the total number of PEs in the actual PE array.

$$\text{PUFM} = \frac{(N_x + 1) \cdot (N_y + 1)}{\texttt{nproc} \cdot lps} \cdot 100\% \tag{5.7}$$

For this mapping, as the problem size grows, PUFM periodically approaches 100% and then drops. A graph of PUFM versus problem size produces the familiar saw tooth waveform that is typically associated with the efficiency of array type processors. The worst case PUFM corresponds to a mesh without any mesh boxes, but such a trivial case is of no interest to us. In section 5.5, PUFM is examined for a range of mesh sizes.

5.4 Performance

To examine the performance of the FDTD code, we will be concerned with execution time, time complexity, processor use due to mapping (PUFM), processor use due to branching (PUFB), speed-up, and MFLOPS metrics. Our interest in PUFB is primarily associated with the handling of PMC boundary conditions. Also our interest in speed-up is to examine how well our FDTD code scales with the number of PEs in the processor array.

From a performance standpoint, the key to speed is being able to execute the finite difference equations as fast as possible. The finite difference equations form the core of the algorithm. As soon as the required constants have been determined and the simulation has been initialized, the remainder of the run time should be spent simply executing the finite difference equations. To make the execution of the finite difference equations faster, a decision was made that if any values were known in advance to not change, they would be calculated once during program initialization, and stored in memory. What this means is that we deliberately traded the memory needed to store these constant values in favor of improved execution time. The cost of this tradeoff is that memory is not used as effectively as it would be otherwise.

5.4.1 A Model for Performance

To be able to examine the performance of the FDTD solver, we needed a simple model that we could use. Since we planned on using a wide range of mesh densities, detailed structures inside the mesh were undesirable. Sacks[52] provided us with just such a model. Sacks used the finite element time domain method to model a resonant cavity. For his simulation Sacks used a cavity that was 3 meters by 4 meters by 5 meters. By exciting the cavity with a Gaussian pulse, sampling fields inside the cavity, and applying th fast Fourier transform, Sacks was able to estimate the frequencies of the resonant modes. Sacks was able to compare his results with well known analytical results and found that his estimates of the resonant modes were in agreement.

There are two things that make Sack's model desirable for our use. First as stated above, the model is simple, there are no detailed structures inside the cavity. Second, the test produces results that are easy to compare with well known analytical results. Table 5.5 lists five of the modes that should be observed in such a resonant cavity. The frequencies given were calculated analytically, see Sacks[52].

Table 5.5: **Resonant Modes for Test Cavity**

Resonant Mode	Exact Frequency - MHz
011	47.990
101	58.269
110	62.457
111	69.279
012	70.706

It was decided that to prove that the FDTD solver was working correctly, we would examine one mesh model in detail. The mesh model selected was 30 boxes, by 40 boxes, by 50 boxes, making each mesh box 0.1 meters along each edge. While this mesh density is approximately three times finer than that used by Sacks, it is important to point out that the mesh density selected was completely arbitrary. Again, our first goal was only to prove that the simulator works.

For excitation, a single Yee cell was chosen and energy was added to individual electric field components, using the technique described by Taflove and Brodwin[62]. The excitation function given by Sacks is given as equation (5.8).

$$f(t) = e^{\frac{-(t-5\sigma)^2}{\sigma^2}} \tag{5.8}$$

The 3dB frequency of the excitation is approximately given by equation (5.9).

Using this equation, the shape of the excitation was selected to be band-limited to approximately 100MHz. ($\sigma = 3$nsec)

$$f_{3dB} \approx \frac{1}{\pi\sigma} \tag{5.9}$$

The simulation was run three times, each time exciting the E_x, E_y, and E_z field component of the selected Yee cell, respectively. Each time the simulation was run, ten thousand time steps were used to allow the system to settle. A time step size of 0.1nsec. was used, which corresponds to a total simulated time of 1μsec.

Electric field components were sampled at a suitable distance from the excitation. After the simulation, the data was padded with zeros to produce 32768 sample points per sampled field component. Padding with zeros is a common practice in DSP, used to improve the resolution of the fast Fourier transform. A simple radix-2 fast Fourier transform was used. Figures 5.4 through 5.6 show the magnitudes of the transforms.

In figure 5.4, the E_x field component was excited and the E_x field component was sampled; Note that modes appear at approximately 47.8MHz and 70.6MHz. In figure 5.5, the E_y field component was excited and the E_y field component was sampled; Note that modes appear at approximately 58.3MHz and 69.3MHz. In figure 5.6, the E_z field component was excited and the E_z field component was sampled; Note that modes appear at approximately 62.5MHz and 69.3MHz. All the resonant frequencies were found to be well within 1% of the correct values. Thus in summary, it appears that our FDTD solver performs simulations that are correct.

5.4.2 Target Machines

To examine the performance of the FDTD code, two different MasPar systems were used: a MP 1101 at Worcester Polytechnic Institute, and a MP 1104 at the University of Oregon. Unfortunately, the MP 1208 at MasPar headquarters has been having severe technical problems and was not available for our use. See appendix A for a summary of the configurations of these machines. In general terms it can be said that the MP 1104 is four times larger than the MP 1101. By running code on machines of different sizes, it is possible to do speed-up performance estimates, to examine the scalability of the program.

Figure 5.4: **Excite E_x, Sample E_x**

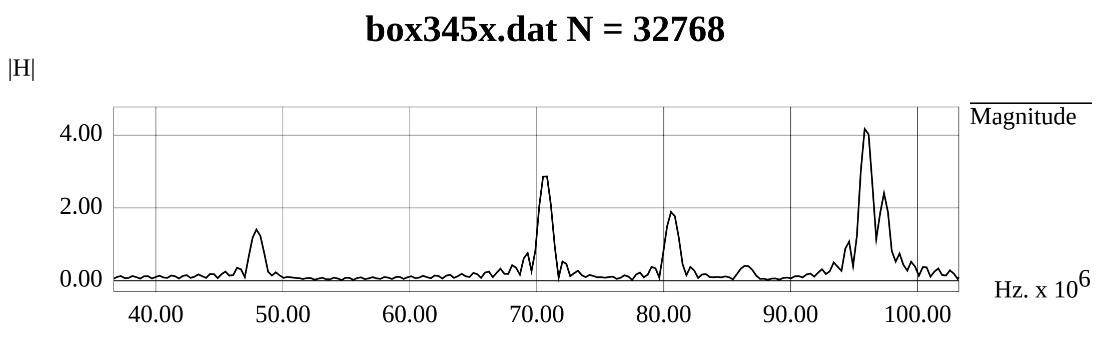


Figure 5.5: **Excite E_y, Sample E_y**

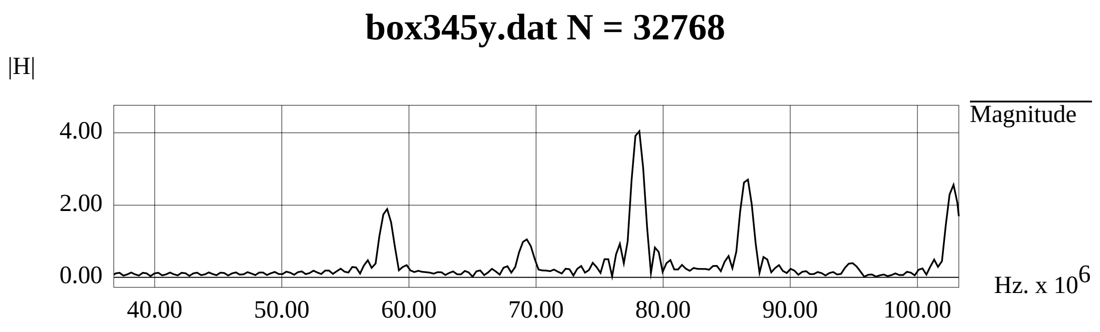
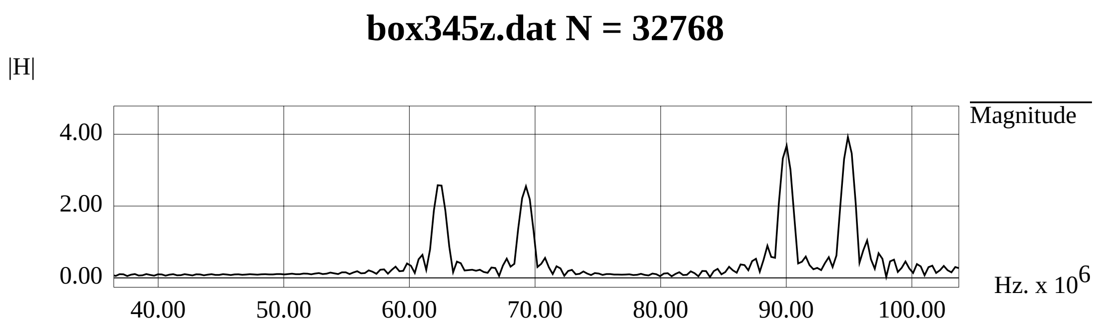


Figure 5.6: **Excite E_z, Sample E_z**



5.5 Execution Time

To measure time, DPU time[3] as well as wall clock time were measured. The parts of the code responsible for performing the FDTD difference equations as well as providing field excitation are referred to as the *FDTD core code*. Time associated with the initialization of the program will be referred to as *initialization time*. Conversely, time associated with performing the FDTD core code will be referred to as *simulation time*.

In performing these time measurements, the FDTD core code was run with excitation as described in section 5.4.1, but without outputting any data. Remember that it was our goal to determine the processing rate of the MasPar system while running this program, not to determine the speed of our NFS file server system. In section 5.4.1 we have already shown that the program works for a specific mesh size. For each mesh size used here however, the program was run for one thousand time steps. A time step size of 10 picoseconds was selected to guarantee that all the simulations would be numerically stable.

5.5.1 Mesh Sizes and PUFM

Table 5.6 provides a summary of the mesh sizes used to examine execution time. Mesh sizes were specifically chosen to provide a range of PUFM values. The mesh sizes selected have *lps*[4] figures of 1, 2, 4, 6, 8, and 9 on the MP 1101. The configuration of the meshes with *lps* equal to 4 and 8 were specifically chosen and should provide the best possible performance on the MP 1104.

The values of PUFM in table 5.6 were calculated by using equation (5.7) from section 5.3.2. Note that in table 5.6 the worst case PUFM for a mesh having *lps* larger than one is not shown, the worst case for the MP 1101 is actually 26.59%, which corresponds to a mesh having $N_x = 32$, and $N_y = 32$. The values of PUFM for the MP 1101 and MP 1104 are illustrated in figure 5.7.

[3]See section 1.7.1 for an explanation of these measures of time.

[4]See section 5.1.2 for an explanation of mapping variables.

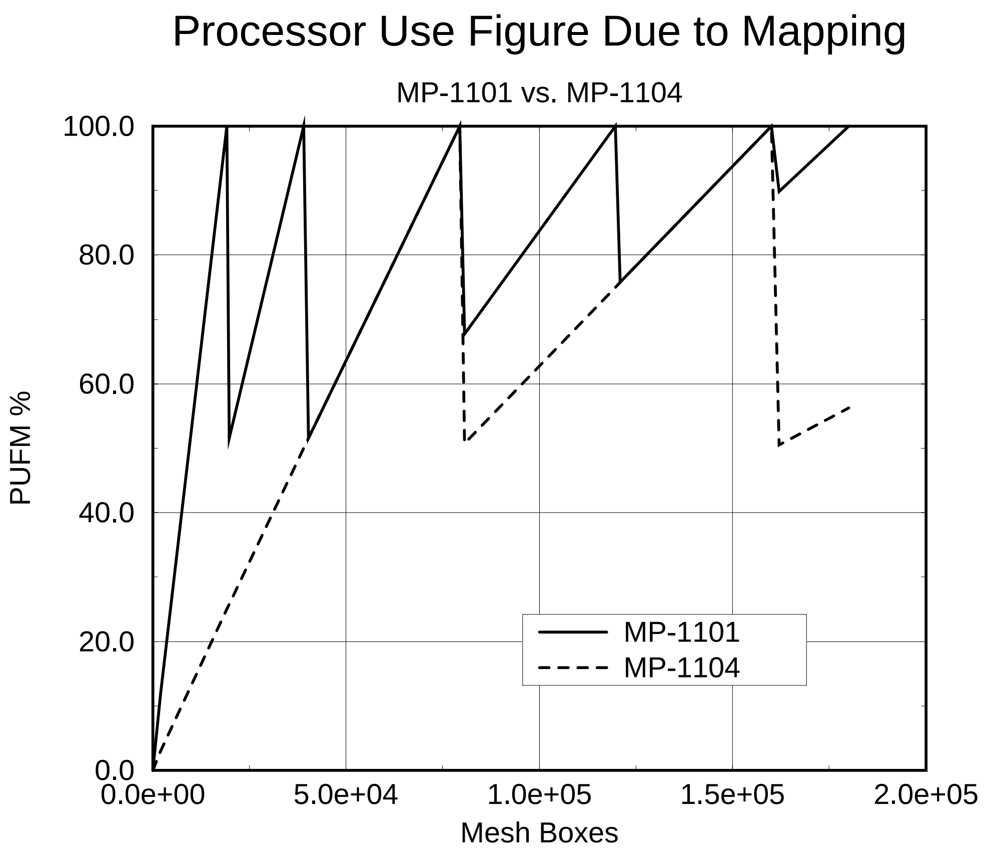


Figure 5.7: PUFM for MP 1101 and MP 1104

Table 5.6: **FDTD Mesh Sizes**

Size	Mesh	MP 1101	MP 1101	MP 1208	MP 1208
N_xxN_yxN_z	Boxes	*lps*	PUFM%	*lps*	PUFM%
10x10x20	2000	1	11.82	1	2.95
31x31x20	19220	1	100.00	1	25.00
32x31x20	19840	2	51.56	1	25.78
63x31x20	39060	2	100.00	1	50.00
32x63x20	40320	4	51.56	1	51.56
63x63x20	79380	4	100.00	1	100.00
64x63x20	80640	6	67.71	2	50.78
95x63x20	119700	6	100.00	2	75.00
96x63x20	120960	8	75.78	2	75.78
127x63x20	160020	8	100.00	2	100.00
90x90x20	162000	9	89.85	4	50.54
95x95x20	180500	9	100.00	4	56.25

5.5.2 Execution Time on a MP 1101

Figure 5.8 shows that for the MP 1101, there is an approximately linear relationship between DPU initialization time and mesh size. The straight line fit to the data shown has a slope of approximately 14.8 microseconds per mesh box. Figure 5.9 shows that for a MP 1101 the DPU simulation time is most apparently stair shaped, where each stair step corresponds to a different mapping value of *lps*. While the shape of figure 5.9 is certainly non-linear, the data was fit to a straight line. Figure 5.10 provides the summary of the curve fit that was performed.

The curve fit data presented in figure 5.10 should support the argument given in section 4.6, that for very large changes in mesh size, the asymptotic time complexity[5] of the FDTD solver should be approximately linear. For the MP 1101 the slope is approximately 1.3 milliseconds per mesh box for one thousand time steps, or 1.3 microseconds per mesh-box, per time step.

[5]See section 2.6 for an explanation of time complexity.

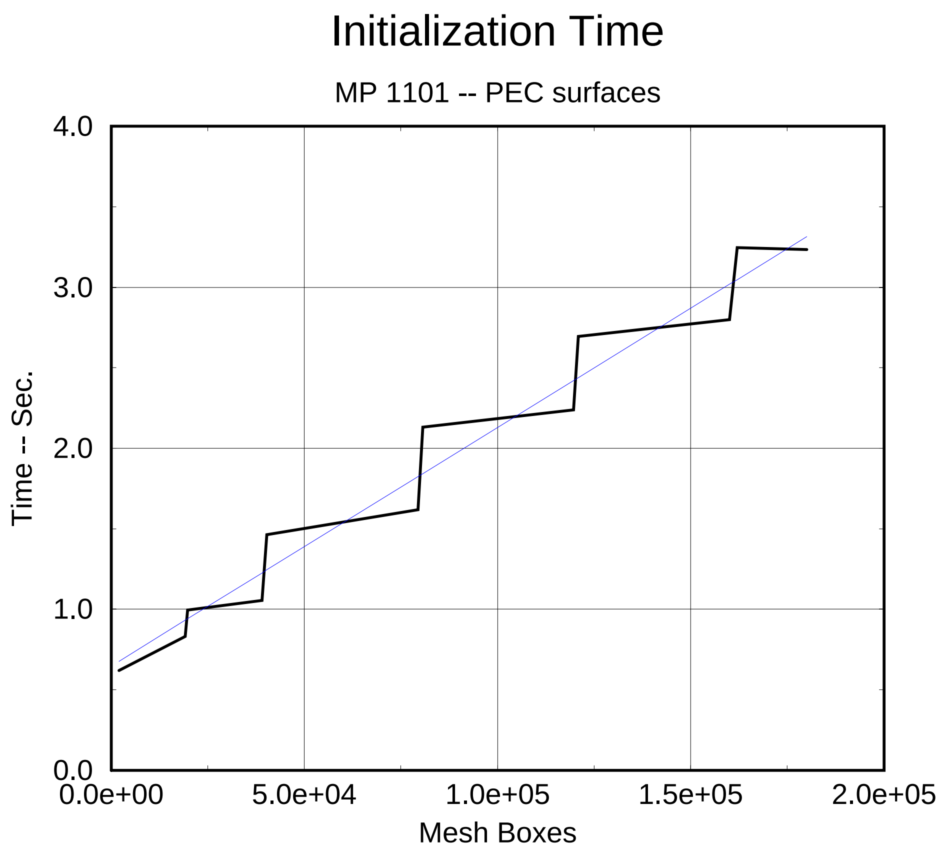


Figure 5.8: DPU Initialization Time for MP 1101

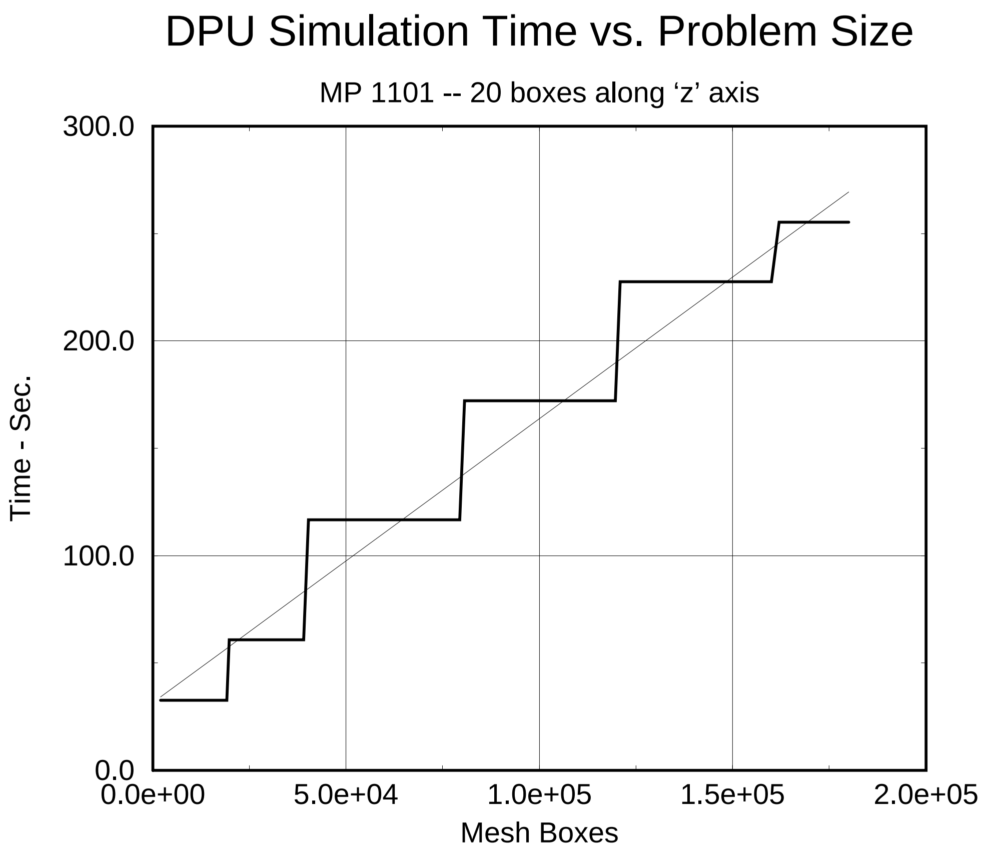


Figure 5.9: DPU Simulation Time for MP 1101

Figure 5.10: **Report of Curve Fit for DPU Simulation Time**

```
Regression of set 2 results to set 3
Number of observations          = 12
Mean of independent variable    = 85260
Mean of dependent variable      = 144.1777
Standard dev. of ind. variable  = 62304.42
Standard dev. of dep. variable  = 85.49991
Correlation coefficient         = 0.9632146
Regression coefficient (SLOPE)  = 0.001321813
Standard error of coefficient   = 0.0001166188
t - value for coefficient       = 11.33447
Regression constant (INTERCEPT) = 31.48001
Standard error of constant      = 12.13487
t - value for constant          = 2.594177

Analysis of variance
Source     d.f Sum of squares Mean Square     F
Regression   1   74605.37      74605.37    128.4702
Residual    10    5807.213       580.7213
Total       11   80412.59
```

5.5.3 Speed-Up Examined

The point of speed-up analysis in this sense is to see how well the FDTD code scales with processor array size. The argument that is often made is that if a program is able to make full use of an array processor, regardless of the number of PEs in the system, then the performance of that program should be proportional to the number of PEs in the array. For the mesh generator, we saw that when we compared the MP 1208 to the MP 1101, there was a speed-up of six, which is somewhat smaller than the expected speed-up of eight.

Figure 5.11 presents the execution times on the MP 1101 and MP 1104, for the mesh sizes that we have been considering. One thing interesting to note about the MP 1104 is that for $lps = 1$, the simulation time is approximately 31.7 seconds, but for $lps = 4$ the simulation time increased to 113.4 seconds. Taking the ratio of these times gives a value of 3.58, which is somewhat less than the expected ratio of 4.00.

Figure 5.12 provides the actual speed-up comparison between the MP 1101 and MP 1104. Note that depending on the problem size, speed-up can be one, almost two, almost three, or almost four. The largest speed-up actually observed was

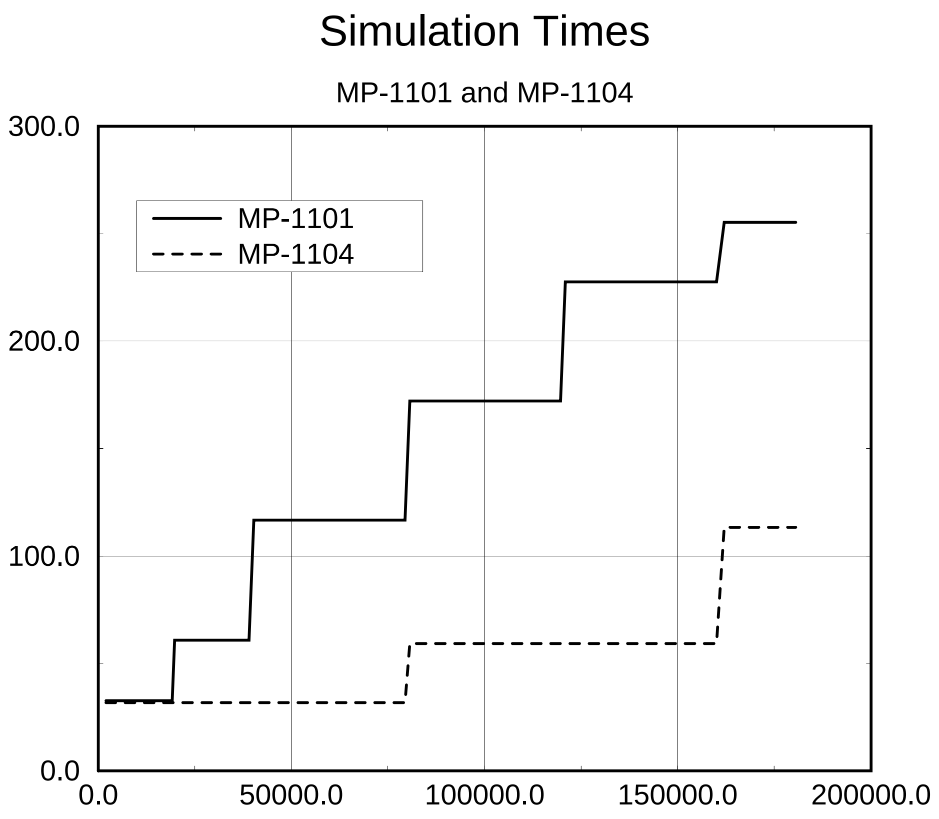


Figure 5.11: DPU Simulation Times MP 1101 and MP 1104

3.85, which is within four percent of the anticipated speed-up of four.

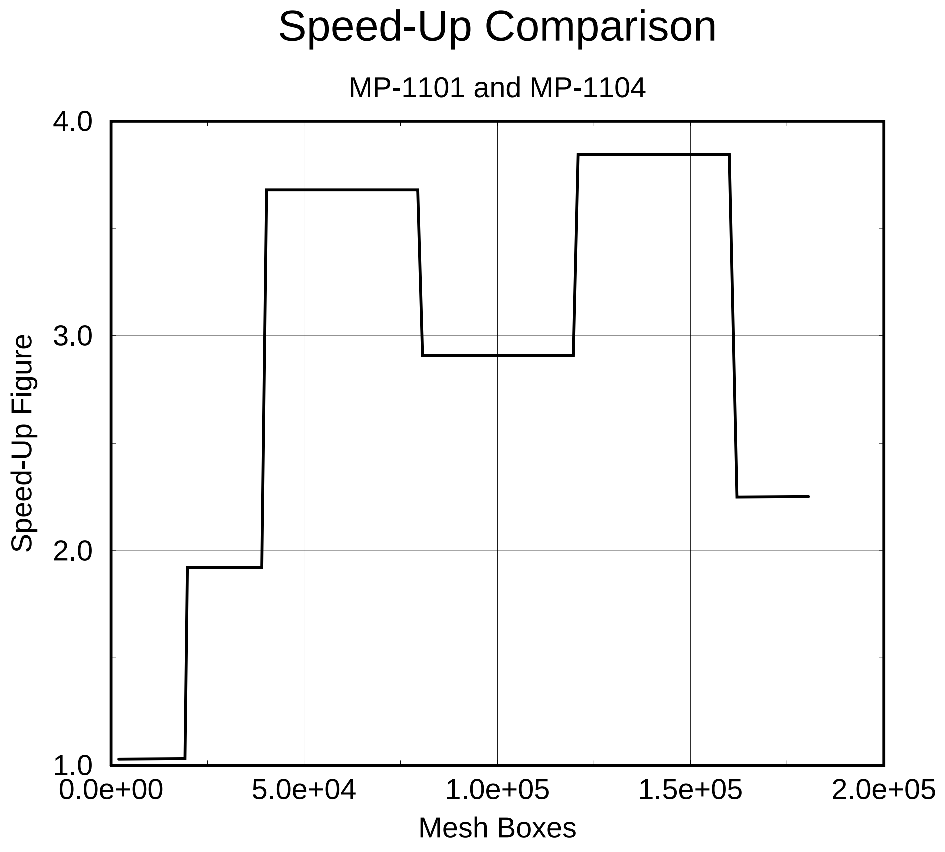


Figure 5.12: FDTD Speed-Up Comparison for MP 1101 and MP 1104

In comparing figure 5.12 with table 5.6, an interesting observation is made; The largest speed-up is seen for those mesh sizes where the MP 1101 and MP 1104 have the same PUFM figures. Note that while the MP 1104 is faster than the MP 1101, in many cases the PUFM is better for the MP 1101 than for the MP 1104. What this means is that in most cases, the program is able to make better use of the MP 1101 than the MP 1104. In other words, wherever the speed-up seen is not approximately four, the utilization is better for the smaller machine.

5.6 MFLOPS Performance

To estimate the MFLOPS rate, we will only be concerned with that part of the execution that we associate with the core of the FDTD simulation. The core of the FDTD simulator has three parts, they are:

- Finite Difference Equations
- Field Excitation Equations
- Mechanisms for Reporting Results

Note that we are not including the time required to initialize the FDTD simulator. The finite difference equations are responsible for performing the FDTD simulation, and are central to the floating point rate analysis. Based on the finite difference equations we estimate the theoretical sustained maximum performance figure for our instruction mix. The contribution from the field excitation equations is discussed, and is found to be negligible in the FLOPS analysis. Lastly, while the simulator was set to produce no results during the MFLOPS analysis, its impact on overall performance is examined when the program does produce output.

5.6.1 Finite Difference Equations

The finite difference equations used to update electric and magnetic field components are presented in appendix E. In the source code, the finite difference equations are actually expressed as shown in table 5.7. The KE and KH terms are proportionality constants that correspond to material properties, these constants are defined during program initialization. To implement the equations presented in appendix E, some slight changes were made. Rather than performing division, the terms $\frac{1}{\Delta x}$, $\frac{1}{\Delta y}$, and $\frac{1}{\Delta z}$, are stored as constants and are multiplied by the difference terms. Also, before performing difference equations, a small amount of code retrieves field components and stores values in local variables.

In examining these finite difference equations, a few things should be self evident. First of all, to update a single field component requires 4 multiplication operations and 4 addition/subtraction type operations. To perform these operations, a total of five electromagnetic field components and two proportionality constants must first be fetched from PE array memory. Also, two of the mesh density variables (KIx, KIy, KIz) must be fetched from ACU memory. After these operations are performed, the updated field value must be stored back into PE array memory. This information is summarized in table 5.8.

By examining table 5.8 we find that to update a single field component, a total of 8 floating point operations are required. Table 5.8 also specifies the number of clock cycles that are required to perform each of these operations. By adding the totals given for each operation, we find that to perform one field component

Table 5.7: **Finite Difference Equations**

$$\begin{aligned}
\mathrm{Ex}[\cdot] &= \mathrm{KElx}[\cdot] * \mathrm{Ex}[\cdot] + \\
&\quad \mathrm{KEcx}[\cdot] * (\mathrm{KIy} * (\mathrm{Hz\_ijk} - \mathrm{Hz\_ijmk}) - \mathrm{KIz} * (\mathrm{Hy\_ijk} - \mathrm{Hy\_ijkm})) \\
\mathrm{Ey}[\cdot] &= \mathrm{KEly}[\cdot] * \mathrm{Ey}[\cdot] + \\
&\quad \mathrm{KEcy}[\cdot] * (\mathrm{KIz} * (\mathrm{Hx\_ijk} - \mathrm{Hx\_ijkm}) - \mathrm{KIx} * (\mathrm{Hz\_ijk} - \mathrm{Hz\_imjk})) \\
\mathrm{Ez}[\cdot] &= \mathrm{KElz}[\cdot] * \mathrm{Ez}[\cdot] + \\
&\quad \mathrm{KEcz}[\cdot] * (\mathrm{KIx} * (\mathrm{Hy\_ijk} - \mathrm{Hy\_imjk}) - \mathrm{KIy} * (\mathrm{Hx\_ijk} - \mathrm{Hx\_ijmk})) \\
\mathrm{Hx}[\cdot] &= \mathrm{KHlx}[\cdot] * \mathrm{Hx}[\cdot] + \\
&\quad \mathrm{KHcx}[\cdot] * (\mathrm{KIz} * (\mathrm{Ey\_dz} - \mathrm{Ey}[\cdot]) - \mathrm{KIy} * (\mathrm{Ez\_dy} - \mathrm{Ez}[\cdot])) \\
\mathrm{Hy}[\cdot] &= \mathrm{KHly}[\cdot] * \mathrm{Hy}[\cdot] + \\
&\quad \mathrm{KHcy}[\cdot] * (\mathrm{KIx} * (\mathrm{Ez\_dx} - \mathrm{Ez}[\cdot]) - \mathrm{KIz} * (\mathrm{Ex\_dz} - \mathrm{Ex}[\cdot])) \\
\mathrm{Hz}[\cdot] &= \mathrm{KHlz}[\cdot] * \mathrm{Hz}[\cdot] + \\
&\quad \mathrm{KHcz}[\cdot] * (\mathrm{KIy} * (\mathrm{Ex\_dy} - \mathrm{Ex}[\cdot]) - \mathrm{KIx} * (\mathrm{Ey\_dx} - \mathrm{Ey}[\cdot]))
\end{aligned}$$

update requires a total of 2132 clock cycles. Since the 12xx series of MasPar DPUs has a clock period of 80 nanoseconds, we can estimate that to perform a single field component update would nominally require 170.56 microseconds.

To estimate the theoretical sustained maximum performance figure we just need to multiply the total number of floating point operations by the number of PEs in the PE array, then divide by the required time, which we found to be 170.56 microseconds. Thus we find that the theoretical sustained maximum performance figure for the MP 1101 is 48.03 million floating point operations per second. The MP 1104 should have a theoretical sustained maximum performance figure of

Table 5.8: **Operations for Single Field Component Update**

Floating Point Operation	Clock Cycles per Operation	Use per Operation	Total Clock Cycles per Op.
PE Memory Load	76	7	532
PE Memory Store	76	1	76
ACU Memory Load	6	2	12
PE F.P. Add/Sub.	126	4	504
PE F.P. Multiply	252	4	1008

192.12 million floating point operations per second.

5.6.2 Field Excitation Equations

To provide Gaussian pulse excitation, the exponential function was used. To be able to set a limit on the number of floating point operations that are associated with the exponential function that we use to calculate our Gaussian excitation, we used the program shown in figure 5.13. The point of this exercise is to prove that the number of floating point operations is small in comparison to the number of floating point operations that are required to update the FDTD mesh once. For such proof, an exact value is not needed, an approximate estimate will be adequate.

In the program shown, there are three loops, each loop is performed `NLOOP` times. The first loop is used to estimate the overhead associated with the `for` loop structure. We used the `mpTimerElapsed()` function to determine elapsed time as measured by the UNIX system clock. Time is returned in milliseconds. The second loop is used to measure the time to perform the addition operation. The floating point values are stored as register variables. The last loop is used to measure the time to perform the `f_exp()` function.

The output of this program is shown in figure 5.14. Remember that time is printed in values of milliseconds. By dividing the number of additions by the time, converted to seconds, a floating point rate is estimated. The first thing that you might notice in the results shown in figure 5.14 is that the floating point rate is not very impressive. It turns out that the ACU is not capable of performing floating point operations, such operations are actually performed by one of the PEs. Thus, what we are seeing for the floating point rate is roughly the floating point performance of a single PE. There is some overhead associated with moving the data to a selected PE, and returning the results back to the ACU.

To determine the number of floating point operations that we will associate with the `f_exp()` function, we simply divide the time to perform the `f_exp()` function by the time to perform the additions, this gives us a value of approximately 9.01546. Since we are really only looking for a reasonable limit, we round this to 10 floating point operations.

To produce the excitation at a single time-step, we use equation (5.10). To evaluate the argument of the exponential function, we perform four floating point operations. The exponential operation requires approximately ten, producing a

```
/****************************************************
 * exptime
 *
 * The goal of this program is to set an upper limit on
 * the number of floating point operations that are to
 * associate with the exponential function f_exp().
 * jmhill@ee.wpi.edu -- Sat Jun  1 20:51:58 EDT 1996
 ***************************************************/
#include <mpl.h>
#include <mp_libc.h>
#include <math.h>
#include <time.h>
#include <mp_time.h>
#include <stdio.h>

#define NLOOP  100000
#define KVAL   1.234

main()
{
  unsigned int t0, t1, t2, t3, tval;
  unsigned int i, tloop, tsum, texp;
  register float sum, eval;

  sum  = 0.0;
  eval = 1.0;
  mpTimerStart();

  t0 = mpTimerElapsed();
  for (i = 0; i < NLOOP; ++i)
    { }

  t1 = mpTimerElapsed();
  for (i = 0; i < NLOOP; ++i)
    { sum += KVAL; }

  t2 = mpTimerElapsed();
  for (i = 0; i < NLOOP; ++i)
    { eval = f_exp( -eval ); }

  t3 = mpTimerElapsed();

  tloop = t1 - t0;
  tsum  = t2 - t1 - tloop;
  texp  = t3 - t2 - tloop;
  printf("NLOOP %d\ntime loop %6d\n", NLOOP, tloop);
  printf("time sum  %6d\ntime exp  %6d\n", tsum, texp);
  printf("FLOP rate %e\n", (NLOOP*1.0e3)/(float)tsum);

} /* end of exptime.m */
```

Figure 5.13: Program `exptime.m`

```
goofy.WPI.EDU> a.out
NLOOP 100000
time loop    277
time sum    4592
time exp   41399
FLOP rate 2.177700e+04
goofy.WPI.EDU>
```

Figure 5.14: Results from `exptime.m`

count of fourteen floating point operations.

$$\text{exc\_val} = \exp\left(\frac{(\text{time\_step\_size}\cdot\text{time\_step} - \text{time\_delay})^2}{\text{variance}}\right) \tag{5.10}$$

In comparison to the total number of floating point values given in section 5.6.1, that are needed to perform a single update, the number of floating point operations needed to calculate the excitation at a single time step is very small. In calculating the FLOPS rate of the overall program, the number of floating point calculations required to calculate the excitation will be ignored.

5.6.3 Actual FLOPS Rate

To determine the FLOPS rate, we divide the total number of floating point operations performed by the simulation time. Since the simulation time is known for the cases presented in section 5.5, our first order of business here is to calculate the associated number of floating point operations.

For this analysis we use the data presented in section 5.5. For such a cavity with PEC surfaces, equation (5.11) gives the number of electric field updates that are performed at each time step. Equation (5.12) gives the number of magnetic field updates that are performed at each time step. Recall from earlier in this section that eight floating point operations are required to perform each component update.

$$\begin{aligned}\text{E field updates} &= 3N_xN_yN_z + N_x + N_y + N_z \\ &\quad -2\left(N_xN_y + N_yN_z + N_xN_y\right)\end{aligned} \tag{5.11}$$

$$\text{H field updates} = 3(N_x+1)(N_y+1)(N_z+1) \tag{5.12}$$

Table 5.9 summarizes the total number of field component updates that are required to perform a single time step. With the knowledge that 8 floating point operations are required to perform a single field component update, and that the simulation times shown in figure 5.9 correspond to one thousand time steps, we can readily calculate the floating point operation rates on the MP 1101. These rates are illustrated in figure 5.15. Of the mesh sizes considered for the MP 1101, the largest FLOPS rate was estimated to be 34.37 million floating point operations per second. The smallest was estimated to be 3.1 million floating point operations per second.

Table 5.9: **Time Step – Field Update Totals**

Mesh Size N_xxN_yxN_z	Mesh Boxes	E Field Updates	H Field Updates	Total Updates
10x10x20	2000	5040	7623	12663
31x31x20	19220	53340	64512	117852
32x31x20	19840	55099	66528	121627
63x31x20	39060	109628	129024	238652
32x63x20	40320	113243	133056	246299
63x63x20	79380	225208	258048	483356
64x63x20	80640	228923	262080	491003
95x63x20	119700	340988	387072	728060
96x63x20	120960	344603	391104	735707
127x63x20	160020	456668	516096	972764
90x90x20	162000	462800	521703	984503
95x95x20	180500	516060	580608	1096668

To get a better sense of the absolute performance of the program implementation, the FLOPS rate was divided by the theoretical maximum sustained FLOPS rate, to give the percent of maximum FLOPS utilization graph shown in figure 5.16.

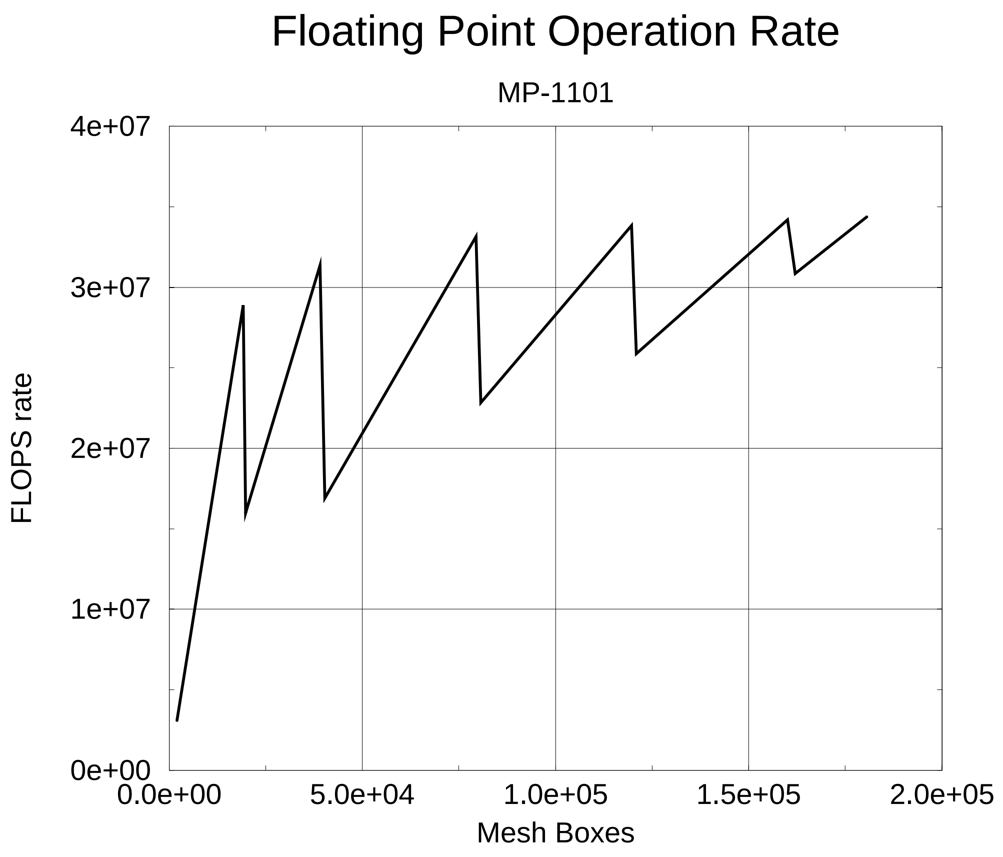


Figure 5.15: FDTD FLOPS Rate for MP 1101

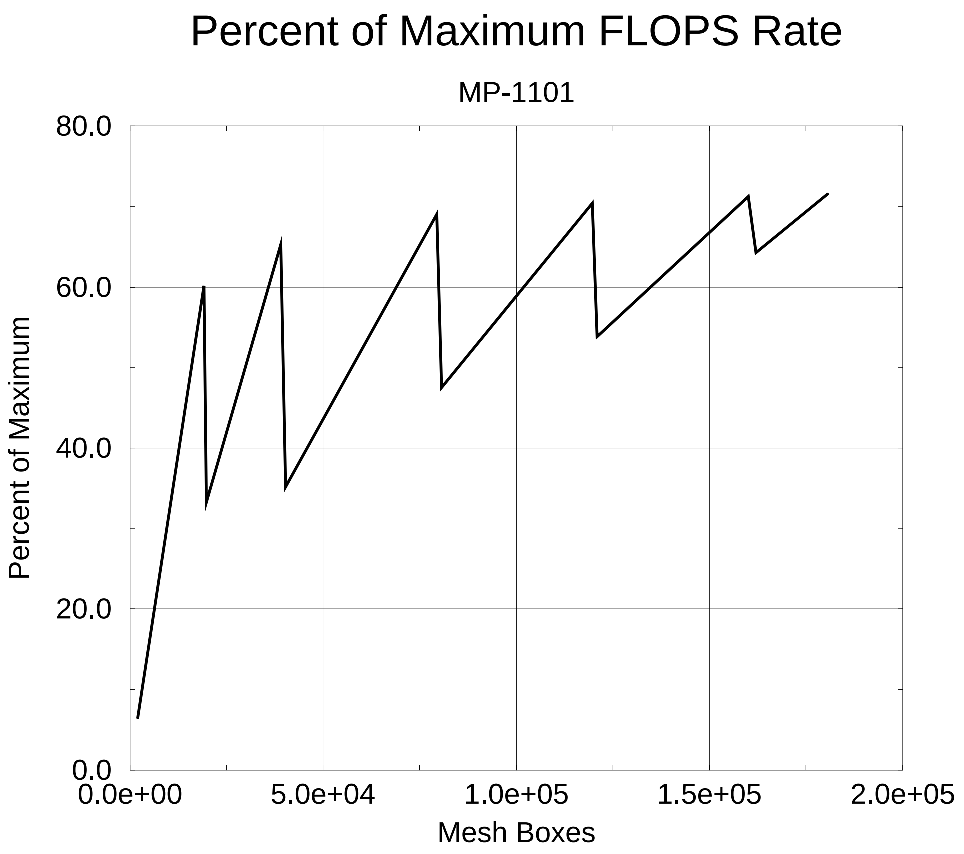


Figure 5.16: Percent of Maximum Theoretical FLOPS Rate

5.7 PMC Boundary Conditions

To minimize the impact of PMC boundary conditions on the performance of the FDTD simulator, the use of the magnetic image method was used, see section 4.4.2. To get a sense of the actual impact of PMC boundary conditions on performance, a test was performed. The same model used to examine FLOPS performance was used with a 30x40x50 mesh. The simulation was run for 1000 time steps, without producing any output. A first simulation was run without making any changes to the boundary condition (All PEC). A second simulation was run after replacing the north surface with PMC boundary conditions. The results of these simulations are summarized in table 5.10.

Table 5.10: **PMC Test Summary**

Simulation Name	Initialize Time(sec.)	Simulation Time(sec.)
PEC Only	2.3	149.4
PMC One	7.0	153.9

Note that the time to required to initialize each simulation increased when one boundary surface was changed to PMC type. This increase is due to the fact that the PEC surfaces were set by default. To set a boundary surface to PMC type, indicators had to be inserted into the mesh file to describe the PMC boundary. Thus the increase in initialization time is associated with the handling of these indicators.

To implement the magnetic image method, a handler fetches magnetic field component values and applies a sign change. The point is that while the same finite difference equations are used to perform the updates to PMC boundary components as the rest of the mesh, it is just the data input to the difference equations that makes a component appear as if its part of a PMC surface.

When PMC surfaces are used, there is a slight increase in the simulation time. This increase is due to the Processor Use Figure due to Branching (PUFB). While the PMC handler has a very low PUFB, (approximately 10%) it performs its action very quickly. Between the two simulations that we considered, the simulation time increased by only three percent. This change, while present is not considered to be significant.

5.8 Data Output

So far, all performance testing has assumed that the program produced no output data. To get a sense of how producing data impacts simulation performance, a test was performed. The same model as used to evaluate FLOPS performance was used. The simulation used a 30x40x50 mesh, was run for 1000 time steps, and produced ten files. For each run, the amount of data produced was varied by changing the number of mesh sheets that was reported each time a file was written. Since there was a pronounced difference between DPU time and wall clock time, wall clock time was used to measure performance, see figure 5.17.

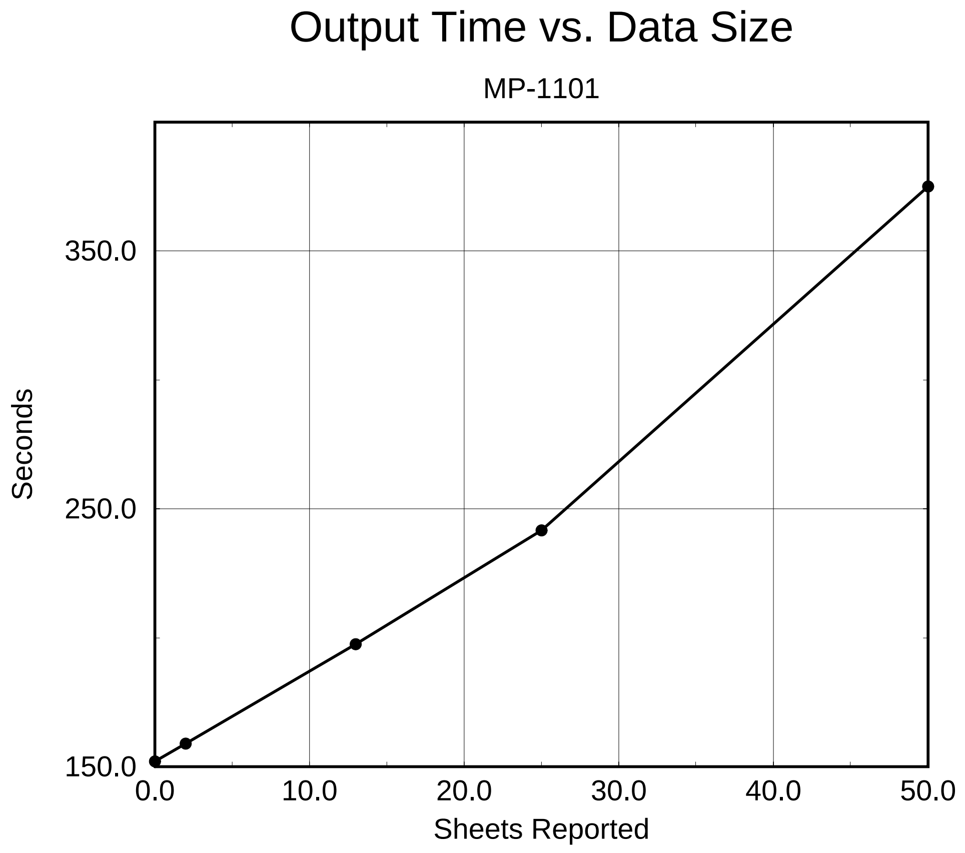


Figure 5.17: Time vs. Output Data Size

The shape of the curve in figure 5.17 is significant, note that it is exactly linear in appearance. Because of the asynchronous handling of output data, if only a small amount of data needs to be output, performance is not greatly impacted. Midpoint in the curve, the time the front end and the DPU each spend on handling file output is roughly equal. When more output is produced, the DPU spends more time waiting for the front end to format output data.

5.9 Summary

From the simulations presented it appears that the FDTD simulation code works. It is clear from the speed-up analysis that large simulations should be run on large MasPar systems. While a larger machine may be faster in executing programs, most often a smaller machine will have a better utilization than a larger machine. On the MP 1101 it appears that the code has a peak absolute performance figure of approximately 34 million floating point operations per second. This figure is impressive as it represents 70.8 percent of the theoretical sustained peak performance figure. The Processor Use Figure due to Mapping (PUFM) provides an explanation of why the FLOPS graph is square in appearance. The impact of PMC boundary conditions and the output of data, on the overall performance of the FDTD code was examined quickly.

Chapter 6

Thesis Conclusion

This chapter brings this thesis to a close by drawing from all other chapters in this thesis, to present the lessons learned and summarize the nuggets of knowledge that we revealed. The material in this chapter is presented in several forms. First this chapter reviews a few of the highlights from this thesis. Second, a discussion of lessons learned in programming the MasPar system is presented. And last, suggestions for future research are discussed.

6.1 Thesis Highlights

Chapter 1 presented an introduction to the central topics of this thesis, that is the development of two programs for the MasPar computer system. The first program is a mesh generator that provides a spatial decomposition of three dimensional objects. The second program is an implementation of the Finite Difference Time Domain Method. To make the need for these two programs plain, the application of these two programs to antenna design was presented.

Before presenting the MasPar system, chapter 1 presented some history of the array processor. The continuum type problem was introduced along with an example that serves as a vehicle for introducing the concept of performance analysis. Following the example, specific metrics used to measure performance on an array processor were given. This material is important as performance analysis is an important part of the content of this thesis.

In chapter 2, the algorithm associated with the mesh generator was presented.

The complexity analysis in section 2.7 provided important insight into the mesh generation algorithm. An outline of the parallel algorithm was presented in section 2.8. To gain more insight into the mesh generator, a serial implementation was produced from the algorithm. By using preliminary results from the serial version, we attempted to predict the performance of the parallel version of the mesh generator.

In chapter 2 we also showed that while the algorithm works, we can do better. The time complexity analysis presented in section 2.7.3 indicates that for large mesh models, approximately fifty percent of the total execution time can be attributed to the facet solver. Section 2.10.3 is significant as it clearly showed that a measurable improvement is possible, even with less than one order of magnitude performance improvement in the facet solver. Given that the facet solver consumes 50% of the total execution time, then the limiting factor on the overall speed-up over the existing code should be approximately 2.0.

Chapter 3 presented the parallel version of the mesh generator and the serial version of the mesh generator in more detail. Important details such as transferring data to and from the DPU, the mapping, and allocation of memory were introduced. The issue of time complexity of the mesh generator was reviewed, this time as a parallel algorithm. Performance was examined by considering execution time, speed-up, the rate of performing floating point operations, the processor element use figure, and the rate at which mesh boxes were generated. For this type of algorithm we found that the processor use figure due to branching played an important role in describing how the implementation scales with PE array size. Lastly, by examining the data, it appears that for moderately sized meshes, the front end consumes a very small amount of the overall execution time. Thus it appears that with this algorithm the asynchronous process model would offer little improvement in performance.

Chapter 4 presented the FDTD method as an algorithm. Topics central to the FDTD method such the FDTD mesh were reviewed. Topics including the Yee Cell, time stepping, analysis stability and accuracy, and mesh boundary conditions were discussed. The idea of using a fast Fourier transform to perform frequency domain analysis was just touched on. The form of the finite difference equations used to perform the FDTD method was introduced. The actual finite difference equations used to perform the FDTD method are derived in appendix E. A discussion of conductive materials was presented along with a special case that we used to derive a special absorbing material that we can use to model absorbing boundary conditions. Lastly, this chapter presented the FDTD method in the form of an algorithm, complete with a flowchart and a discussion of its time complexity.

Chapter 5 presented the parallel implementation of the FDTD algorithm. From the simulations presented it appears that the FDTD simulation code works. It is clear from the speed-up analysis that large simulations should be run on large MasPar systems. While a larger machine may be faster in executing programs, most often a smaller machine will have a better utilization than a larger machine. On the MP 1101 it appears that the code has a peak absolute performance figure of approximately 34 million floating point operations per second. This figure is impressive as it represents 70.8 percent of the theoretical sustained peak performance figure. The Processor Use Figure due to Mapping (PUFM) provides an explanation of why the FLOPS graph is jagged in appearance. The impact of PMC boundary conditions and the output of data, on the overall performance of the FDTD code was examined quickly.

The appendixes contained in this thesis must not be overlooked. Appendix A provides a summary of the MasPar systems that we ran code on for this thesis. Of the three electromagnetic simulations presented, the one presented in Appendix B is the most complete. See section 1.1.2 for details regarding all three of the simulations presented in this thesis.

Appendix C serves as a source of useful information for those who have more than a casual interest in developing an application for the MasPar system. Appendix D presents some formulas that are associated with cross products. These formulas were found to be useful for the development of parallel mesh generator. Appendix E presents the derivation of the finite difference equations that are central to the FDTD method. Appendix F contains the user manual and lastly Appendix G contains the list of references for this thesis.

6.2 Lessons Learned While Programming

This section discusses a few of the lessons that were learned while programming the MasPar system. To start with, I refer to a few of these lessons as being *universal,* as they will always be useful, regardless of the platform used. The first lesson is that the concept of mapping equivalence presented in appendix section C.3 is central to the use of any computer system. While the idea of mapping equivalence was central to the development of specific virtualizations for the DPU, we also used the idea of mapping equivalence to explain how the overall mesh was stored in the front end of the MasPar system.

The next lesson was the generality of algorithms. While an algorithm can be devised so that an implementation of it will take advantage of certain features in a computer system, there are certain qualities that algorithms have that a programmer cannot "shake off." Things like the time complexity of an algorithm will be somewhat obvious, whether implemented on a serial machine or a parallel machine.

As pointed on in the beginning of chapter 1, just having high performance computer hardware will not bring you high performance results. For the best results, high performance computers demand that programmers devise better, more sophisticated algorithms. This lesson should especially hearten those who do not have access to a machine like the MasPar. By devising better algorithms, better performance can be achieved on any computer system. A large part of this thesis is simply the discussion of the algorithms that were derived for this research. In one particular case, an example was given to prove that a better algorithm must exist. For more details regarding this example, see the suggestions for future research.

The next lesson learned is that clearly, while the DPU is not a general purpose machine, the front end is. Clearly while some applications, or parts of applications run well on an array processor, others do not. Many applications, or parts of applications actually run better on a uniprocessor system. When writing applications for the MasPar system it is important to not forget the front end processor. In both of the programs presented in this thesis, the front end served as an important concurrent processor element, along with the DPU.

The fourth lesson learned is that moving data between the front end and the DPU is most efficiently performed by using data structures. One of the biggest headaches in writing the mesh generator program was finding a way to share data between the front end and the DPU in an effective way. In writing the FDTD code, it was found that data structures provide a simple and clean way to pack and move data between the two machines by using the DMA transfer mechanism. I must admit that in programming uniprocessors I viewed the 'C' `struct` tool as simply a convenience. In programming the MasPar however I have learned the wisdom that normally only systems developers are aware of. The wisdom is that there are often very good reasons for such things as the `struct` tool.

The fifth lesson learned is that serial file I/O is slow and is best managed directly in the front end. For the DPU to perform any kind of file operation, the DPU makes a system call, which leads to a series of events, which in turn leads to the eventual indirect call of functions in the front end to perform the requested file I/O operation. Rather than having the operating system "rattle about" it is far more efficient to write code yourself to implement exactly what needs to be done.

A significant improvement in initialization time was seen when the code associated with reading the FDTD mesh file was moved to the front end.

The use of data structures was helpful with file I/O as well. By having the front end read data, and pack the data into data structures before sending the data to the DPU, additional time savings was realized. The reason for this improvement is that since there is a fairly slow communications link between the DPU and front end, the use of data structures which provide compression and the use of DMA transfers have a noticeable impact on performance. Likewise, by having the DPU organize resultant data into data structures before sending to the front end, improvements in performance were also seen. The mesh generator used a bitmap scheme that provided a compression of at least eight times. Consider also that numbers are more efficiently stored in binary form than in ASCII form. In the FDTD solver, by sending numbers in binary form and formatting the output in the front end, a compression of four was sometimes seen.

The issue of I/O handling came up again and again. In any real application, significant amounts of data must be transferred to and from files. By having the MP 1101 owned by WPI mounted as an NFS client a tremendous negative impact on system performance was observed. Also, the serial link between the DPU and front end had a negative impact on performance. Having a parallel disk array present in the DPU may be very useful as a cache, helping to ease the slowness of I/O handling. While the MP 1101 can be considered to be a relatively fast machine, I/O represents a terrible shortcoming for this type of system, and deserves much attention.

6.3 Suggestions for Future Research

The following are few comments that are necessary, regarding possible future research. To start with, while the mesh generator works correctly and exhibits what could be described as good performance, it can be improved. Shamos and Preparata [47] present a better solution to the two dimensional containment problem. Since this problem is at the core of our mesh generator algorithm, it would be worthwhile to investigate how Shamos's algorithm could improve the mesh generator program. Since Shamos's algorithm requires some setup time for each facet, this could potentially be a good application of the asynchronous process model. In such an application, the front end and DPU act as independent concurrent processors.

The FDTD solver represents a situation where overlay type techniques could effectively be used. Once memory is allocated to the FDTD array, there is no change in the memory configuration during run-time. The core of the FDTD solver is a set of equations that are repeatedly performed in a very predictable way. For future work it would be worthwhile to examine how a memory-recycling technique could be applied to allow larger FDTD meshes to be studied.

For the FDTD solver, while Mur's ABC was tried, it was disabled. In the end, PEC and PMC were the only applied boundary conditions that were implemented. It would be worthwhile to try implementations of other boundary conditions. In particular other ABCs should be investigated.

While the mapping selected for the FDTD solver has advantages over other mappings, to be honest, I was never really completely satisfied. The biggest problem I see with the mapping is that the performance exhibits a large dependence not only on the size of the mesh, but also on the shape of the mesh. It would be worthwhile to consider other mapping schemes for FDTD solvers. A three dimensional hierarchical mapping might be a worthwhile to research, for example.

The measurement of time is important, according to Patterson and Hennessy[44] time is the most fundamental measure of performance. In regard to the issue of measuring time, in a concurrent processor it is not so obvious as to what is actually being measured. Wall clock time is usually only used to characterize the performance of programs in the context of an overall system. Traditionally, in a multiprogrammed uniprocessor system, CPU time is used to characterized the performance of any single program. In a multiprogrammed concurrent processor system however, the concept of CPU time is exceedingly vague as a process may be shared by multiple processors, each of which may or may not be overlapping processing time. For most of the cases we encountered, most of the processing took place in the DPU. To examine performance I was able to ingeniously sidestep the larger issue by using DPU time as the measurement of overall process time. I believe however that the measurement of processor time for concurrent processing in general is an important topic that deserves some research.

6.4 Summary

It should be clear from all this discussion that the FDTD code and mesh generator both work. This document presents three examples of electromagnetic simulations. In section 5.4.1 a simple model is presented that we use to examine the performance of the FDTD solver. In appendix B an example of a microstrip structure is presented, and the user manual in appendix F the example of a waveguide is given. A large part of this thesis was the discussion of the algorithms used in the programs, and the performance of the programs. Chapter 2 presents the mesh generator algorithm and chapter 4 presents the FDTD method as an algorithm. Chapters 3 and 5 present the implementations, show how well both programs run and present absolute indicators of program performance.

Lastly, this this thesis will be archived with all related source code, on tape as well as on CD-ROM. You may contact the author by email at `jmhill@ece.wpi.edu` or the world wide web at `http://www.wpi.edu/~jmhill`, but please keep your correspondence brief.

Appendix A

MasPar System Configuration

In this appendix we review the configuration of each machine that we executed code on for this thesis. Appendix C provides some key topics that a new programmer would want to know, about how to actually program the MasPar system. We start here with some general comments regarding possible MasPar configurations.

The MP–1 was the first series of parallel computers from MasPar. At the time of this writing, MasPar has long released the newer MP–2 series. Due to circumstances however, we were not able to run code on any of the newer machines. In a MP–1 system, processor array boards can be added to increase the system processing power. Each array board is used to contain a set of 1024 PEs. Depending on the size of the power supply and chassis, there are two possible system groupings. The MP 1100 series systems support one, two, or four array boards and have five I/O slots. The MP 1200 systems support one, two, four, eight, or sixteen array boards and have fifteen I/O slots.

The model number of a MasPar system is derived from the system grouping, by indicating the number of array boards that the system has. The machine owned by WPI is referred to here by its node name, 'goofy'. Note however in the rest of this thesis the machine is referred by its model name, 'MP 1101'. Node 'goofy' has one array board, with 1024 PEs.

MP 1100 and MP 1200 systems may have as little as 16K bytes per PE, but may have as much as 64K per PE. Other hardware options are also possible. To determine the actual configuration of the DPU in a MasPar system, simply login and issue the `mpconfig` command at the user prompt. The `mpconfig` command can only be issued from the front end of a MasPar system. The following is a `mpconfig` report from node 'goofy';

```
goofy> mpconfig
Copyright (c) 1989-1993 MasPar Computer Corp.  All rights reserved.
Version MP3.2.0
MasPar DPU Model MP-1101 (32 rows, 32 columns)
ACU IMEM size: 1 MByte
ACU CMEM size: 128 KBytes
PE memory size: 16 KBytes
goofy>
```

Node 'goofy' represents a minimum MasPar configuration. From the report given, we can see that there are indeed 1024 PEs present, arranged an array, 32 rows by 32 columns. We can also see from the report that each PE has 16 Kbytes of its own memory. The ACU has 128 Kbytes of data memory (Referred to as CMEM) and 1 MBytes of instruction memory (Referred to as IMEM). Lastly, note that node 'goofy' contains no hardware options, thus all I/O and disk storage must be handled by the front end. If a parallel disk array were present for example, a reference to it would be present in the `mpconfig` report.

To get an idea of what a `mpconfig` report from a more impressive machine looks like, consider node 'maspar' which is owned by MasPar Corporation. While all of our software development was performed on the machine owned by WPI, MasPar was helpful in our research effort by allowing us to have access to one of their computer systems.

```
maspar> mpconfig
Copyright (c) 1989-1993 MasPar Computer Corp.  All rights reserved.
Version MP3.2.0
MasPar DPU Model MP-1208 (64 rows, 128 columns)
ACU IMEM size: 1 MByte
ACU CMEM size: 128 KBytes
PE memory size: 64 KBytes
PVME in I/O slot 8 (8 MBytes)
IORAM allocation: MPFS, 0 MBytes; IORAMFS, 0 MBytes; other, 7 MBytes
maspar>
```

The processor array in node 'maspar' is formed from 64 rows and 128 columns of PEs, a total of 8192 PEs. Each PE has 64 KBytes of its own memory. Note that besides the familiar ACU memory allocation, node 'maspar' has two special options. There is a reference to a 'PVME' card, this is a high performance parallel VME card that provides improved communications between the DPU and the front end. There is another reference to 'IORAM,' this is just additional memory that is present to assist in handling input/output operations, see figure 1.8 on page 19.

The last machine we consider is node 'beauty', which is located at the University of Oregon. Special thanks go to Prof. John Conery, who is with the University of Oregon, and helped to make node 'beauty' available to us. It is interesting to note that unlike the other machines we ran code on, the front end to node 'beauty' is a DECstation 3100. The configuration for this machine is shown below, note that this is a typical MasPar configuration with 4096 PEs and no special hardware options.

```
beauty% mpconfig
Copyright (c) 1992 MasPar Computer Corp.  All rights reserved.
Version 3.1
MasPar DPU Model MP-1104 (64 rows, 64 columns)
ACU IMEM size: 1 MByte
ACU CMEM size: 128 KBytes
PE memory size: 64 KBytes
beauty%
```

Thus to summarize, for this thesis I ran code on three different MasPar MP–1 systems. Table A.1 summarizes the configurations of the machines used.

Table A.1: **Machines Used in Performance Analysis**

	MP 1101	MP 1104	MP 1208
Item	'goofy'	'beauty'	'maspar'
PE rows	32	64	64
PE columns	32	64	128
PEs total	1024	4096	8192
PE memory	16K	64K	64K
ACU CMEM	128K	128K	128K

Appendix B

A Complete Example

This appendix presents an example that was performed from beginning to end. The example starts with a solid model of a microwave band eliminator, which is simulated to produce data in the time domain. After processing the data with a fast Fourier transform, a transfer characteristic is produced to show clearly that the example is indeed a band eliminator. The results are compared to well known published results.

Figure B.1 is a mechanical drawing of a microwave band eliminator based on the use of a planar stripline structure. The top and bottom surfaces of the structure are plated to provide a symmetric wave-guide structure. The metallic material sandwiched inside the structure is assumed to be of minute thickness.

The goal of this simulation was to provide the transfer characteristic of the structure for the range of frequencies from 2.0 GHz to 4.5 GHz. Bonetti and Tissi[12] provided this example, they indicate that the transfer characteristic should have a significant dip in magnitude at a frequency just above 3.0 GHz.

To provide a 50 ohm feed-point match, the stripline width was selected to be 0.5 cm. A Gaussian pulse was used to excite the system. The width of the pulse was selected to band-limit the excitation to approximately 10 GHz.

To perform the analysis, two different models were used. The first model, shown in figure B.2 was used to take samples from the incident wave. The second model, shown in figure B.3 is of the actual stripline structure. Surrounding the exterior of each model is a lossy material used to absorb waves. Special attention was given at the feed-points to minimize discontinuities, and hence possible reflections. For

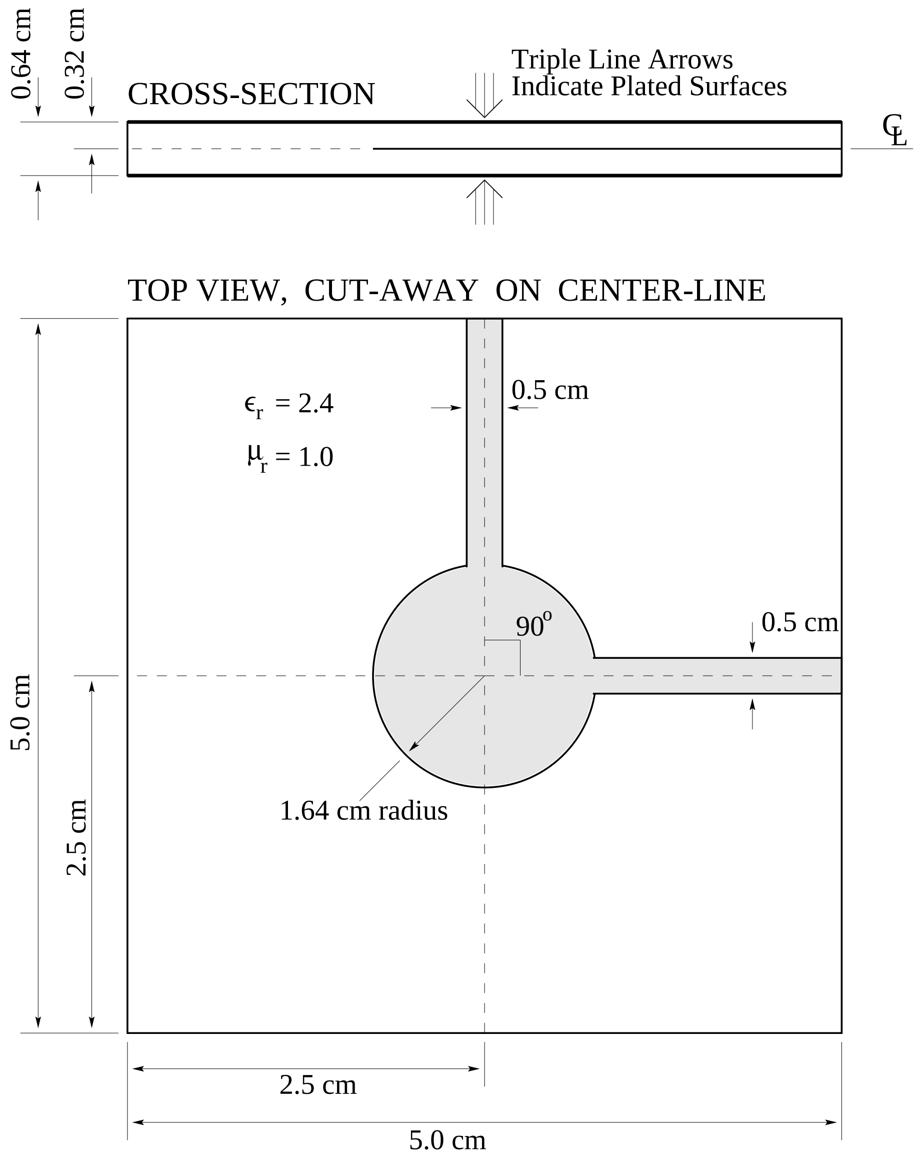


Figure B.1: Microwave Band Eliminator Structure

each model, a 125x125x8 mesh was used.

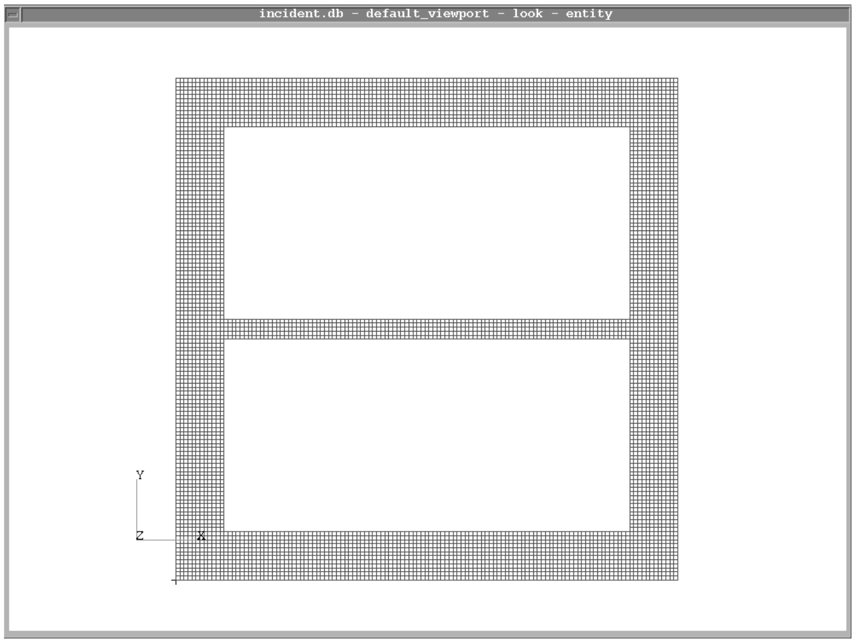


Figure B.2: Incident Wave Test Model

The simulation was run with a time-step size of two picoseconds, for a total of five thousand time-steps. The simulation was run several times. During the first simulation, Patran data files were produced so that the propagation of the electromagnetic waves could be seen. Results are shown at several time steps, see figures B.4, B.5, B.6, and B.7. The figures show the incident wave reaching the circular region, then bouncing inside the circular region. Waves are seen exiting the circular region, through both ports.

During the second and third simulation, the incident, reflected, and output waves were sampled. The incident and output waves were examined to produce the transfer characteristic shown in figure B.8. As reported by Bonetti and Tissi, the dip appeared exactly where they stated it should be.

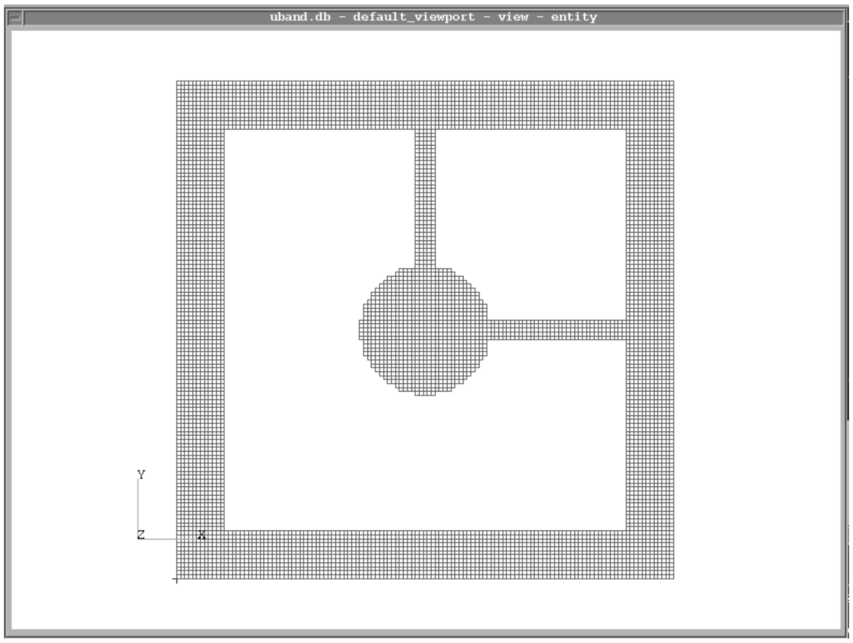


Figure B.3: Band Eliminator Test Model

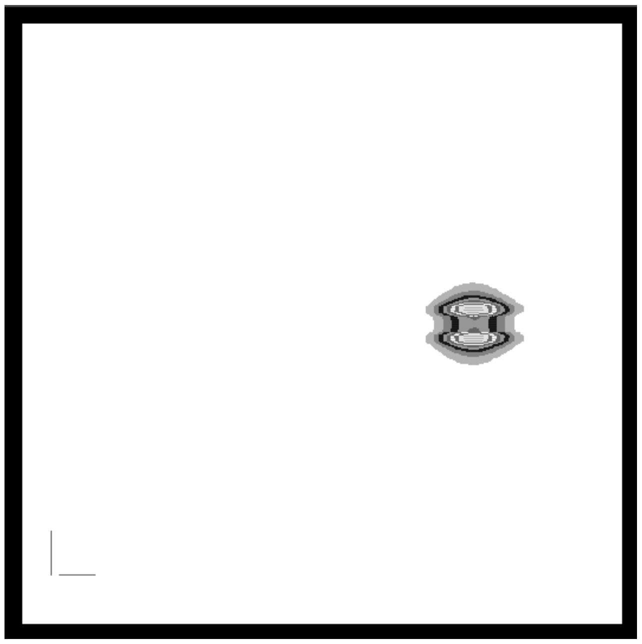


Figure B.4: Simulation After 100 Time Steps

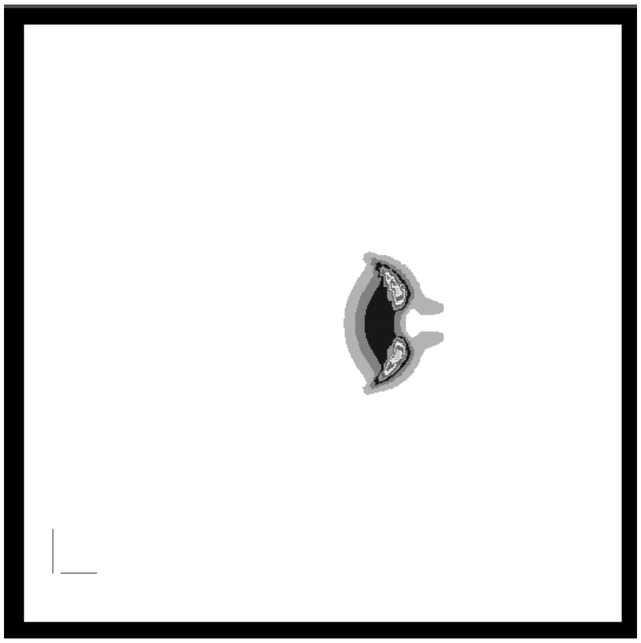


Figure B.5: Simulation After 150 Time Steps

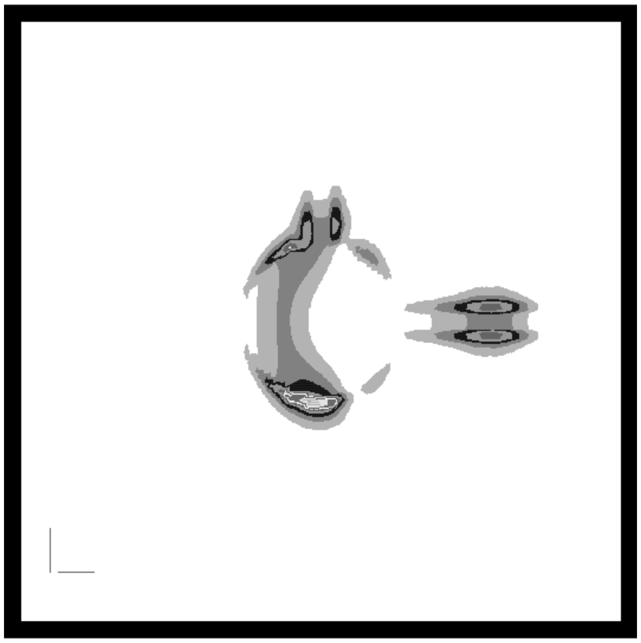


Figure B.6: Simulation After 200 Time Steps

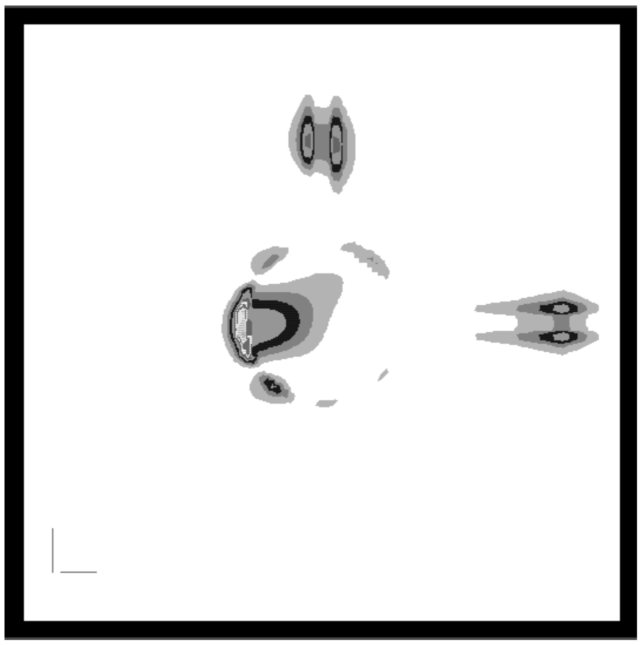
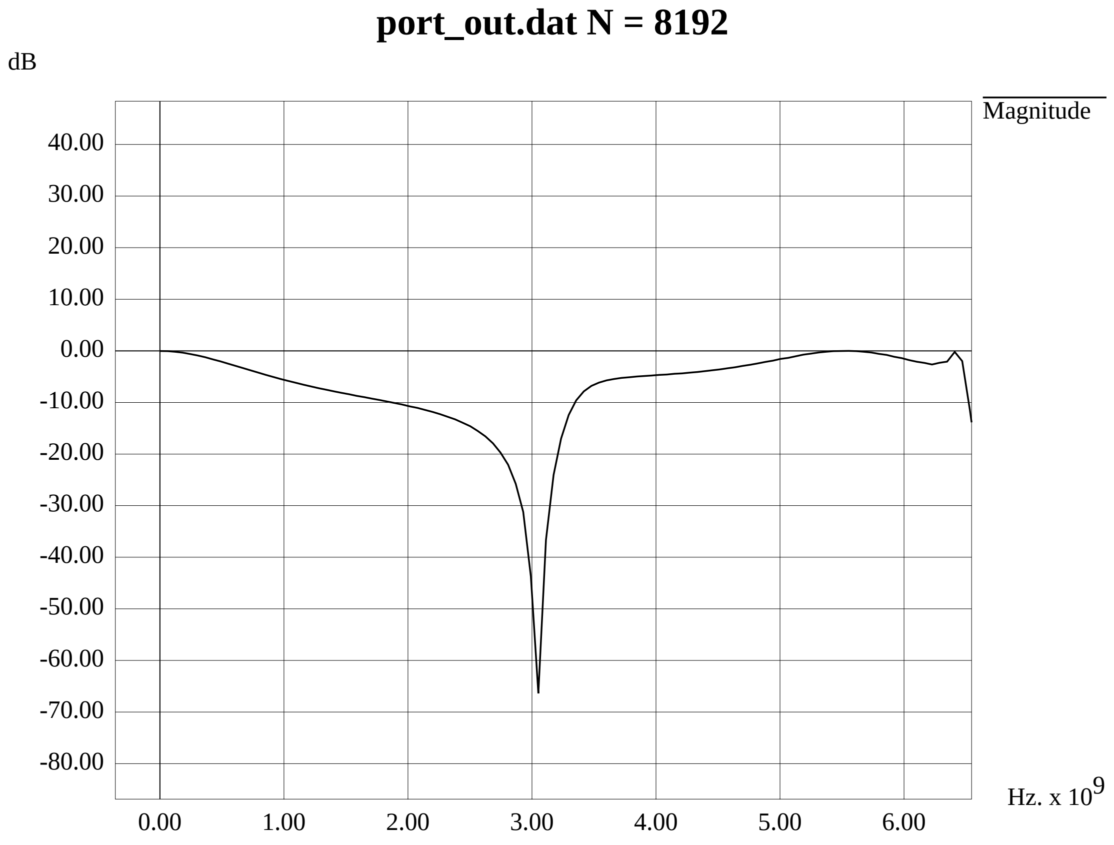


Figure B.7: Simulation After 250 Time Steps

Figure B.8: **Band Eliminator Transfer Characteristic**



Appendix C

Using the MasPar System

In this appendix, we assume that the reader has more than a general interest in the MasPar system. Here we "get down to serious business" and consider a few of the more important aspects of programming the MasPar system. In this appendix we first consider the memory in a MasPar system. Next we consider process control between the front end and DPU, and lastly we discuss the important topic of mapping an application to the MasPar system.

C.1 MasPar Memory Topics

In a MasPar system, sharp divisions are present in the memory systems. The front end and the DPU have entirely separate memories, data must be explicitly moved as formal arguments or copied between the machines. The operating system and hardware provide DMA mechanisms just for copying memory between the DPU and front end. Inside the DPU, memory is divided between the ACU and the PE array. Memory allocated in the PE array is allocated uniformly for all PEs. The term *plural* is used to describe variables stored in PE array memory, or temporarily in PE registers. The use of memory is extremely important in the MasPar system. In the following we consider a few memory related topics.

C.1.1 Front End Memory Configuration

The front end of the MasPar system owned by WPI is a DECstation 5000/200 workstation, we will use this machine here to serve as an example system. In writing applications for a MasPar system, the memory in the front end will play important roles in the final implementation. In our research for example, the amount of memory in the front end played a role in determining the size of the largest problem that can be handled. The first command we consider reports the amount of memory that is allocated for use by virtual memory as well as the amount that is currently available to the user.

```
goofy> /etc/pstat -s
```

Since memory is constantly being assigned and de-assigned to and from processes, the best that this command can do is report statistics at a given moment in time. Currently, the front end workstation at WPI has 64 Mbytes allocated for use as virtual memory. To determine how much memory is represented by a logical page, use the `pagesize` Ultrix command. In a DECstation 5000 a page of virtual memory represents 4096 bytes.

One way to determine the amount of physical memory present in the front end, is to look into the machine's error log. Every time a DECstation boots, a report is made that contains the physical memory configuration. To examine the last entry at startup, use the following command at the Unix prompt:

```
goofy> /etc/uerf -R  more +/START—
```

Currently it appears that node 'goofy', the machine that serves as the front end to WPI's MasPar system is configured with 24 MBytes of physical RAM, of which 18.73 MBytes is available for use.

C.1.2 Plural Declarations

The MasPar MPL programmer's reference manual[35] provides a good introduction to the use of the *plural*[1] qualifier that is be used when a programmer defines variables.

[1]Page 2-3 of MPL Ref. Manual

> "The *plural* type qualifier tells the MasPar system that the variable being defined is a variable to be allocated on each PE, not a variable to be allocated in the ACU.... Storage allocation is uniform across all PEs. Therefore, plural declarations affect all PEs, even PEs that are not active when the declaration is encountered."

Variables that are not stored in PE memory, but stored in ACU memory are referred to as *singular* variables. The use of pointers with plural data allows an unusual flexibility[2]. While in most cases a *plural pointer* is actually a singular variable that points to plural data, it is also possible for a pointer to be stored as a plural variable. In such situations, such a pointer can be declared to point to singular data or to plural data. These last two options are particularly interesting, since a pointer can be regarded as data, by allowing pointers to be stored in PE memory, this allows each of the PEs to independently address its own memory.

C.1.3 Available PE Memory

Since the MasPar system is meant to be used in a multiuser environment, memory is partitioned to allow for multiprocessing. Memory is assigned to users as multiples of a predefined partition size. The size of a memory partition is set by the system administrator. The DPU manager[3] automatically provides a program with enough memory for its static needs, but a user is allowed to request more memory. In addition to the static requirements, if a program allocates memory at runtime, the user must ensure that additional memory is available. The `mplimit` command[4] advises the DPU manager as to how much memory should be made available to a program. The `mplimit` command is particularly useful for programs that dynamically allocate memory.

The `mpsize` command may be useful in determining the amount of memory that a user should request. The `mpsize` command reports the size of an executable as well as its static memory requirements. In addition, the MasPar Command Reference Manual[32] points out that in addition to the user requirements, the DPU requires some PE memory for housekeeping, and gives an estimate of 384 bytes.

To provide an absolute indicator as to how much memory can be allocated by

[2]Page 2-5 of MPL Ref. Manual

[3]The DPU manager is a service provided the operating system.

[4]`mplimit` and related topics are presented in the MasPar Commands Reference Manual[32]

a user, the program in figure C.1 was written. The program works by repeatedly assigning then freeing up memory. After the program was compiled, `mpsize` reported that the executable required 472 bytes of static PE memory.

```
/****************************************************************
 pmemtest

 This program determines the available amount of PE memory and reports
 to the user.  Use this program with the 'mplimit' command to see
 how much memory is actually made available to your process.
****************************************************************/
#include <stdlib.h>
#include <stdio.h>

main(int argc, char *argv[])
{
  plural char *char_data;
  unsigned int nbytes;
  for (nbytes=1; nbytes<16385; nbytes++) {
    if ((char_data = p_malloc(nbytes)) == NULL) {
      printf("Out of Memory (%d)!\n", nbytes);
      exit(0); }
    p_free(char_data);
  }
}
```

Figure C.1: PE Memory Size Test Program

The 'pmemtest' program was run several times, the results are presented in table C.1 For each run the `mplimit` command was used to change the amount of memory requested. The amount of PE memory requested is listed in the column 'Request'. The column 'Actual' is the amount of available PE memory reported by the 'pmemtest' program. Lastly, the 'Difference' column lists the difference between the amount requested and the amount reported. Note that since 472 bytes of static PE memory was reported by `mpsize`, and the MasPar Commands Reference Manual indicates that 382 bytes of overhead exists, it was expected that the largest deficit would be 854 bytes. Note however that the largest deficit is always 1247 bytes, which is accounted for by memory being allocated in the DPU for stack space.

Lastly note that the operating system scheduler makes use of data from the 'mplimit' command. By examining the user's request for memory, the scheduler can take precautions to ensure that multiple users can share the available memory.

Table C.1: **pmemtest Results**

Request	Actual	Difference
4096	3041	-1055
4288	3041	-1247
4289	7073	+2784
8320	7073	-1247
8321	11105	+2784
12352	11105	-1247
13353	15137	+1784
16384	15137	-1247

C.1.4 Demand Paging and Other Techniques

The MasPar PE array does not implement any kind of demand paging. What this means is that unless special techniques similar to overlays are used, the amount of physical memory available immediately determines the amount of data that can be stored in the DPU. The paging that is performed in the DPU is limited to program instructions associated with the Array Control Unit. Unfortunately, user access to instruction memory is not allowed.

Silberschatz, Peterson and Galvin[58] as well as Patterson and Hennessy[44] both provide a useful introduction to overlays as well as to the idea of demand paging. Both techniques essentially reuse memory by cycling data to and from disk storage. This reuse of memory allows a computer to run programs that cannot otherwise fit all at once into physical memory. There are at least two significant disadvantages that all overlay type techniques share, the first is complexity. Unlike demand paging, the operating system provides no support to overlays. All data movement to and from disk must be organized by the application programmer. The task of writing code to manage such transactions can be a sizable feat, just by itself. In comparison, demand paging is performed by the operating system in a way that is nearly transparent to the user and programmer alike.

The second disadvantage of overlays is that unlike demand paging, there is always an associated loss of performance. Any time a program spends on manipulating overlays is time that cannot be used to perform useful work. In contrast, in a demand paged system, the operating system and the hardware have special features that help to minimize the cost of page swapping. Without direct support from the lower system layers, overlay techniques cannot help but be slow.

Clearly the use of such techniques as overlays is best to be avoided. There are situations where the use of overlay type techniques cannot be avoided however, for example the processing of extremely large matrices. The MasPar Math Library[34] provides routines that are known as *out of core* matrix solvers. These routines use special techniques that allow the DPU to process huge matrices that could never fit all at once into PE memory. While there is some loss of performance due to the cycling of data between PE memory and disk, the loss can be eased by the use of a parallel disk array.

The current version of the parallel mesh generation program does not implement any such memory overlay technique. By storing the overall mesh in the front end, the mesh generator makes use of the demand paging available there. Also, by having the overall mesh in the front end it is possible to make better use of the available memory in the DPU. In the current situation, the mesh generator can produce meshes much larger than could ever be accommodated by the FDTD solver.

The current version of the FDTD solver does not use any kind of memory recycling technique either. For the FDTD solver the size of the largest FDTD mesh that can be studied is directly limited by the amount of physical memory in the DPU. The FDTD solver represents a situation where overlay type techniques could effectively be used. Once memory is allocated to the FDTD array, there is no change in the memory configuration during run-time. The core of the FDTD solver is a set of equations that are repeatedly performed in a very predictable way. For future work it would be worthwhile to examine how a memory-recycling technique could be applied to allow larger FDTD meshes to be studied.

C.2 Process Control

Because the MasPar MP–1 computer system is actually comprised of two machines, a DPU and a front end that are meant to work together, provisions were made in MPL to coordinate the two machines. The MPL reference manual[35] provides information as to how process control is handled.

> "The design concept uses a simple subroutine calling convention. All state changes occur through subroutine requests, execution, and returns. No other forms of communication are allowed or needed. For example, the DPU user process can call a routine that prints a message to the screen(a front end activity) or the front end user process

> can call a routine that will do an image processing transformation in the DPU…routines for moving large amounts of data make a system call to initiate a more efficient block DMA-style move."

Thus for one machine to initiate execution in the other machine, something akin to a typical function call is used. MPL allows for two execution models, a synchronous model as well as an asynchronous model, each is described next;

> "In the synchronous model, either the front end user process or the DPU user process is running at any given time; they are not allowed to run simultaneously…In contrast to the synchronous model, the asynchronous execution model allows both the front end user process and the DPU user process to run concurrently."

MPL provides functions for synchronizing the asynchronous execution model. Naturally both processors have to be running so that synchronization points can be reached. For this project the synchronous model found the most use, however the asynchronous model was used as well. An important point needs to be made, distinguishing which processor "calls the shots" is an important matter;

> "The simple protocol used in these routines requires that there be only one calling master that may alternate between the front end and the DPU. Note that this does not preclude either side from doing asynchronous operations as long as it does not require interaction from the other user process."

When the front end as master calls the DPU, the DPU becomes the new master. The DPU then can either return from the subroutine or call another routine in the front end. Both these actions return master control to the front end. Process control is most important as it provides a tight link between the DPU and front end.

C.3 The Mapping Topic

A few issues are always faced anytime an array processor application is written, one such issue is called the mapping of the application to the computer system.

The topic of *mapping an application* is actually a collection of basic issues that determine, in a fundamental way, how resources will be used. *Resources* are features of a computer architecture that can be used in a program. Such resources typically include the PE array, the ACU, the front end, communications between the front end and DPU as well as interprocessor communications between the PEs themselves and the ACU. The task of mapping an application is actually composed of two related activities. First consider the mapping of processes, which basically involves deciding where and when functions will be called and performed. Since the MasPar system is actually composed of two machines, this activity is not in general as easy as it might first appear.

In most introductory programming examples, "main" is the only function that is referenced. For such a program the task of process assignment is trivial. The activity of mapping processes is usually not introduced to student programmers until later, after they have mastered more advanced programs. For without such experience, one really cannot appreciate the complexity of the topic. We will follow this apparent tradition by first discussing the topic of memory system mapping. The topic of mapping processes is discussed later in this section.

C.3.1 Mapping Data Structures to Hardware

The second activity involved in mapping an application involves the mapping of data structures to physical hardware. To understand this second task, three associated ideas must be clearly understood:

1. mapping equivalence
2. domain decomposition
3. processor virtualization

We will start the discussion by presenting the idea of mapping equivalence. In the most basic form used here, a mapping is a mathematical function[5] that defines a correspondence between sequences of numbers. Flanders[15] has a simple example that we present here. Consider the simple case of mapping N data items given in a sequence to a linear array of N PEs such that exactly one data item is

[5]According to Anton[3](See page 53), a function is a rule that assigns to each element in a set A (the domain), one and only one element in set B (the range).

held in each PE. The number P identifying the PE in which data item I is stored is given by:

$$P = \mathit{mapfun}(I)$$

where *mapfun* is a one to one[6] mapping of I onto P over the range 0 to $N-1$. Since we must not only be able to assign values according to a mapping, but must be able to determine, according to the mapping, where data came from, it is important that we be able to determine a reverse mapping that maps P onto I over the same range. What this implies is that a mapping should be one to one.

$$I = \mathit{mapfun}^{-1}(P)$$

The idea of mapping equivalence is actually quite general. Besides being used to map data to the PE array, it is also used to define any array in the front end. The next concept we introduce is referred to as *domain decomposition.* According to Fox[7],

> "Each complex system [to be solved] is assumed to have associated with it a domain of objects or data. The first step in concurrent computation consists of *decomposing* this domain into several grains... made up of 'atomic' or basic entities which cannot be usefully decomposed further."

The analogy presented by Fox is that like certain cast metals, close inspection of a sample reveals small metallic regions that are referred to as *grains.* We refer to the basic entities as *grains*, or *members* of the domain. In problems that satisfy partial differential equation, such a Maxwell's equations, space serves as the domain and is *discretized* into individual points that represent small regions in the space. Thus individual points in space serve as *grains*, or basic members in the domain.

The last topic to introduce before reviewing the concept of mapping data structures, is the concept of processor virtualization. At this point, the physical architecture of the MasPar system should be very familiar to the reader. While the physical hardware configuration is pretty much fixed, the programmer can take steps to make the physical hardware appear differently to the problem space. Rather than mapping domain members directly to processor elements, it has been found to be more effective to map domain members to a virtual representation of the computer architecture. While the PE array in the MasPar system is two dimensional, memory can be thought of as providing a third axis. By use of mapping

[6]According to Anton[3](See page 453), a function is *one to one* if it does not take on the same value at two distinct points in its domain.

[7]Page 43 of Fox et. all

equivalence, processor elements and memory can be made to appear as a different configuration.

The mapping of memory and processor elements to an alternative configuration provides a layer of abstraction between the decomposed problem and the actual hardware that the MasPar Display Library[33] refers to as a *specific virtualization.* Figure C.2 is one such example of a specific virtualization. In this example an n by n processor array is mapped to appear to an algorithm as being a $2n$ by $2n$ processor array. It is important to point out that the mapping essentially determines how the computer will appear in a sense that is very similar to that of how "memory" appears to a program implementation on a demand paged computer system. It is important that the reader understand that in the MasPar system, it is the program implementation that performs this virtualization mapping.

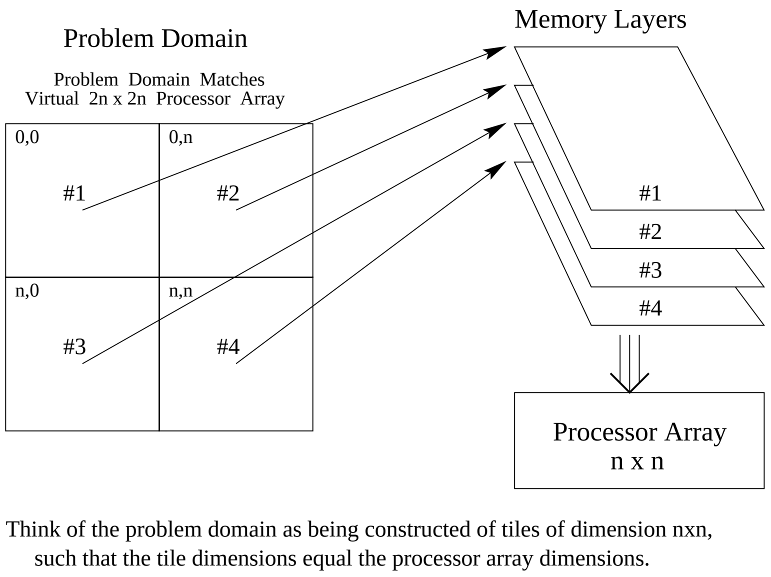

Problem Domain
Problem Domain Matches Virtual 2n x 2n Processor Array
0,0
0,n
n,0
n,n
#1
#2
#3
#4
Memory Layers
#1
#2
#3
#4
Processor Array n x n
Think of the problem domain as being constructed of tiles of dimension nxn, such that the tile dimensions equal the processor array dimensions.

Figure C.2: Two Dimensional Cut and Stack Virtualization

Prins and Smith[48] wrote a sort application for the MasPar system that provides a good example of the use of a virtualization. According to Prins and Smith;

> "To sort arrays that have more elements than there are processors, a virtualization technique must be employed. Unlike the [Connection Machine], virtualization is not implemented in hardware; instead the physical machine size is exposed in the MPL language, so that different virtualization techniques can be programmed. This can be done efficiently because MPL programs the [ACU]."

Prins and Smith also point out that as an alternative to MPL, automatic virtualization can be provided during compilation of other higher level languages, such as the Fortran dialect supported by MasPar. As Prins and Smith point out, the advantage of using MPL is that the programmer has full access and full control over the virtualization. A case in point is that Prins and Smith found it most effective to use the processor virtualization of a hypercube, "as it preserves the hypercube structure of the original algorithm. . . ." Jacobsen[18] as well as Tilton[65] also provide other examples of specific processor virtualizations as well.

Figure C.3 serves as a good summary as it illustrates the mapping of a set of data structures used to store a problem that has been suitably decomposed. While the physical hardware configuration is pretty much fixed, the *mapfun()* function provides a match between the hardware and the decomposed problem. The algorithm provides a theoretical means of solving a problem that has been domain decomposed. In the figure, the puffy cloud represents the theoretical aspects of a problem that are actually being addressed by the algorithm. Note that since different MasPar systems may have different numbers of PEs, the mapping should also serve to provide a virtualization that will make the best use of the actual hardware.

C.3.2 Assigning Processes to the FE and DPU

The job of mapping processes involves deciding how tasks will be assigned to the front end and the DPU, as well as which type of process control model will be used. Except for the simplest programs, most applications have some kind of division between the front end and the DPU. While much emphasis has been placed on the DPU, we must not lose sight of the front end. First consider situations where serial computations must be performed, while the ACU is certainly capable, the front end will do a much better job. In comparison to the ACU which is a 12 Mips machine, the front end is a 24 Mips machine. Also, the front end is better optimized for performing serial operations.

Next, consider that the asynchronous model allows for concurrent processing[8] with the DPU and the front end. By allowing the front end to process data concurrently with the DPU, the front end provides a means to boost the performance beyond what is capable by the DPU alone. Such a boost in performance is best achieved when an algorithm has "naturally serial" and "naturally parallel" parts

[8]Fox[14] (See page 2) defines *concurrent processing* as the use of several working entities, working together toward a common goal.

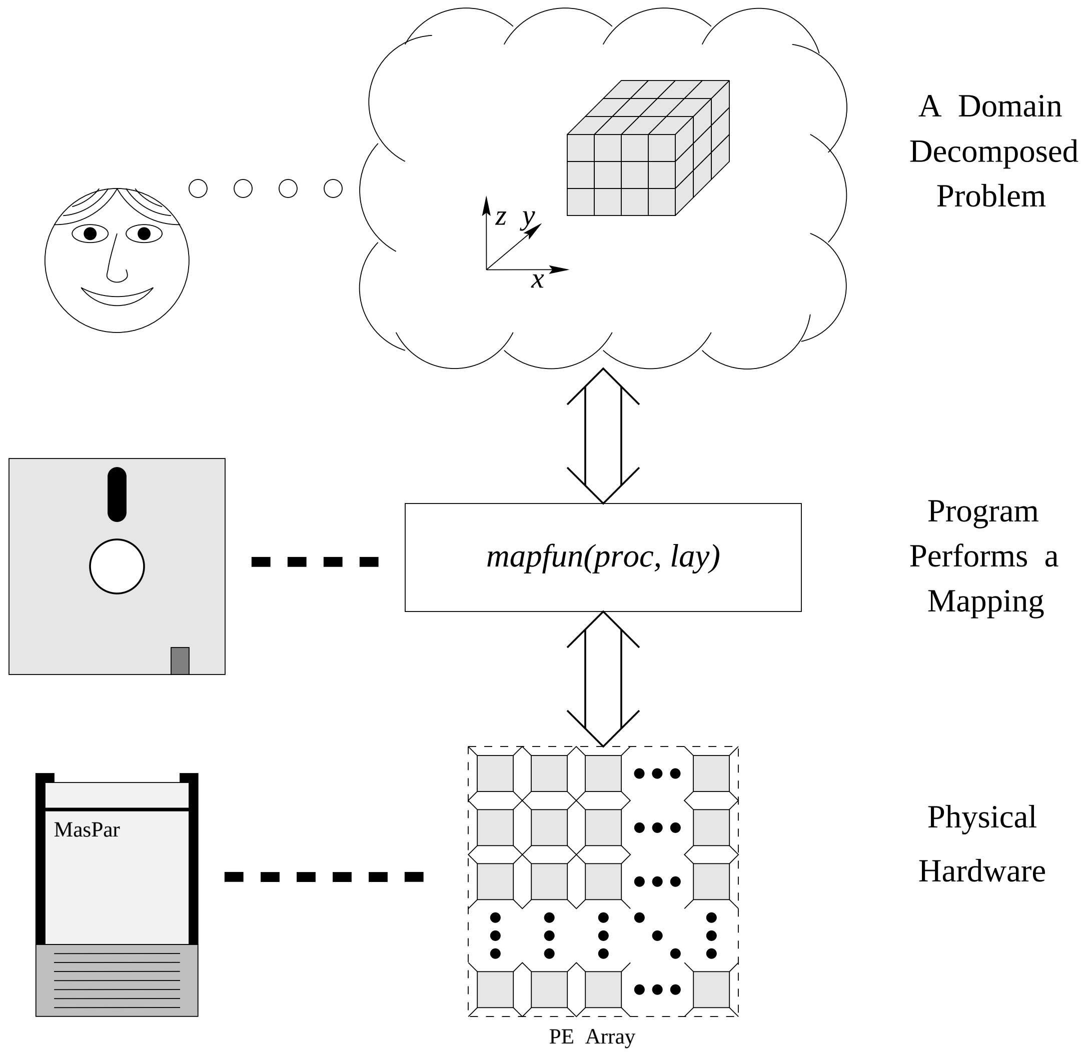


Figure C.3: Illustration of Data Structure Mapping

that can be performed simultaneously.

Lastly, note that no matter how parallel an application may be, there will always be some work that the front end must perform[9]. By taking control of such details directly from the front end, it is often possible to achieve better performance. Consider the following example, a function call such as `fopen` made in the DPU just calls a library function in the DPU that in turn calls a library function in the front end. Unfortunately, library functions must be capable of handling many possible scenarios. By performing library functions in the front end, a layer of library functions is eliminated.

C.3.3 Making a Decision Regarding Mapping

Almost always a decision involving mapping involves some type of tradeoff between resources, for example one virtualization might make the best use of PEs and PE memory, but may complicate use of interprocessor communications. Alternatively, a virtualization that makes the best use of one form of interprocessor communications may not make the best use of all the PEs. For certain problem sizes all the PEs might be used, but for other problem sizes a sizable fraction of the PE array would be unusable. Likewise, memory use plays a role in determining which virtualization to select, as we shall see later. In the following we consider the issue of transferring data between the front end and the DPU, this is an important issue that must be examined before choosing a particular overall mapping scheme.

C.3.4 FE and DPU Data Transfers

The efficient handling of communications between the front end and the DPU is an important issue that complicates the task of mapping processes to the MasPar system. During execution, processes in the front end and the DPU must often "share" data. Whenever data changes, it is important that processes in the DPU and front end have updated versions. Because the DPU and front end memories are disjoint, in physically separate machines, some method must be used to ensure that shared data values are properly updated. The developers of MPL appear to have had the viewpoint that requiring the MasPar system itself to automatically update shared data would be from a performance standpoint, "too expensive." Thus the task of ensuring data integrity has been delegated to the programmer.

[9]Such jobs as disk and terminal I/O are tasks that only the front end can perform, for example.

In MPL, the programmer must be fully aware of how and where data is being used, and must select an appropriate means and opportunity for transferring data between the machines. In a sense, in MPL there really is no such thing as "shared data", the phrase "transferred data" is more apt.

There is an obvious need to be able to efficiently transfer data between the front end and the DPU, this presents a real challenge to the programmer. The transfer of large amounts of data back and forth between the front end and the DPU can potentially result in reduced system performance. Rather than *sloshing*[10] data between the front end and the DPU, it is best that a programmer incorporate as part of the algorithm, techniques for minimizing the amount of data traffic. One such technique is to only transfer data at certain opportunities, such that the data traffic can be minimized. Another technique that applies when opening files in the front end is to neatly organize data from files into data structures and eliminate superfluous text that was used to format the data in the files. Remember that a floating point number stored as a `float` requires only four bytes, but as an ASCII string may require ten or more bytes. By properly organizing data into data structures, data is most easily transferred to the DPU.

To complicate matters, certain data structures can only be moved by using a DMA style copy. Since a DMA copy in this sense is a strict memory to memory copy, DMA copies require that memory be properly allocated in both machines so that transfers can take place. Thus an appropriate solution to the problem of mapping processes must also consider the proper allocation of memory in both machines.

C.3.5 Mapping Overview

The specific virtualization and process assignments selected for an implementation are essentially arbitrary. The MPL programming language is most flexible as it allows a programmer to make choices as well as implement efficient mappings. Because of this flexibility, MPL was chosen for our application. Other high level programming languages such as Fortran automatically provide a virtualization but do not provide as much flexibility. When programming in MPL, the programmer first considers the desired performance of the implementation. The programmer next examines the algorithm and decides how the algorithm can be performed by the DPU and the front end and considers such things as memory use, the need

[10]The term *sloshing* is defined by the MasPar Commands Reference Manual[32] as the inefficient transfer of large amounts of data back and forth, between the DPU and the front end. The meaning of the word *sloshing* should be clear to the reader by its use here in context.

for interprocessor communications and the use of input/output operations. Based on these topics the programmer decides which virtualization will be used, which processor will perform each part of the algorithm, and what mappings will be used.

Appendix D

Triangular Area

Anton and Rorres[4] present a useful introduction to the cross product, in particular they provide a geometric interpretation[1] that leads to an equation for determining the area of triangles formed between vectors. A special case is well suited for two dimensional space and has a unique property that will be introduced. The following is an excerpt from Anton and Rorres[2];

> "If $\vec{\kappa}$ and $\vec{\nu}$ (shown in figure D.1) are nonzero vectors in 3-space, then the norm of $\vec{\kappa}\times\vec{\nu}$ has a useful geometric interpretation. Lagrange's identity (see theorem 3.4.1 in Anton & Rorres)... states that
>
> $$\|\vec{\kappa} \times \vec{\nu}\|^2 = \|\vec{\kappa}\|^2 \, \|\vec{\nu}\|^2 - (\vec{\kappa} \cdot \vec{\nu})^2 \tag{D.1}$$
>
> If θ denotes the angle between $\vec{\kappa}$ and $\vec{\nu}$, then $\vec{\kappa} \cdot \vec{\nu} = \|\vec{\kappa}\| \, \|\vec{\nu}\| \cos(\theta)$, so that (D.1) can be rewritten as
>
> $$\begin{aligned} \|\vec{\kappa} \times \vec{\nu}\|^2 &= \|\vec{\kappa}\|^2 \, \|\vec{\nu}\|^2 - \|\vec{\kappa}\|^2 \, \|\vec{\nu}\|^2 \cos(\theta) \\ &= \|\vec{\kappa}\|^2 \, \|\vec{\nu}\|^2 \left(1 - \cos^2(\theta)\right) \\ &= \|\vec{\kappa}\|^2 \, \|\vec{\nu}\|^2 \sin^2(\theta) \end{aligned}$$
>
> Thus,
>
> $$\|\vec{\kappa} \times \vec{\nu}\| = \|\vec{\kappa}\| \, \|\vec{\nu}\| \sin(\theta) \tag{D.2}$$

[1]See page 129 in Anton & Rorres

[2]Anton and Rorres use alternate notation for vectors. To follow the same arrow notation as the rest of this document, without reusing any previous symbols, $\vec{\kappa}$ and $\vec{\nu}$ were selected to replace the variables $\boldsymbol{u}$ and $\boldsymbol{v}$ respectively, that Anton and Rorres use.

But $\|\vec{\nu}\|\,\sin(\theta)$ is the altitude of the parallelogram determined by $\vec{\kappa}$ and $\vec{\nu}$. Thus, from equation D.2, the area of the parallelogram is given by

$$A_p = (\text{base})(\text{altitude}) = \|\vec{\kappa}\|\,\|\vec{\nu}\|\,\sin(\theta) = \|\vec{\kappa}\times\vec{\nu}\|$$

In other words, *the norm of $\vec{\kappa}\times\vec{\nu}$ is equal to the area of the parallelogram determined by $\vec{\kappa}$ and $\vec{\nu}$.*"

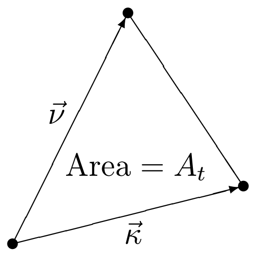


Figure D.1: Vectors and Triangular Area

Since the area A_t of the triangle formed between $\vec{\kappa}$ and $\vec{\nu}$ is half that of the area A_p of the parallelogram, we can state that;

$$2\,A_t = \|\vec{\kappa}\times\vec{\nu}\| \tag{D.3}$$

Note that since Anton and Rorres define θ as the smallest positive angle between $\vec{\kappa}$ and $\vec{\nu}$, $\sin(\theta)$ can only be positive or zero. For the following discussion we define θ_r according to the right hand rule, such that θ_r is the angle from $\vec{\kappa}$ to $\vec{\nu}$ when their tails are joined. Since θ_r is allowed to be positive as well as negative, in terms of θ_r, equation D.2 can be rewritten as follows;

$$\|\vec{\kappa}\times\vec{\nu}\| = \|\vec{\kappa}\|\,\|\vec{\nu}\|\,|\sin(\theta_r)| \tag{D.4}$$

D.1 A Special Case

Next, we consider a special case that will lead to a useful equation. Suppose that $\vec{\kappa}$ and $\vec{\nu}$ are still nonzero vectors, but that their z components are zero. In such a situation $\vec{\kappa}$ and $\vec{\nu}$ correspond to two dimensional vectors drawn on the x-y plane. Thus, $\vec{\kappa}$ and $\vec{\nu}$ can be expressed component-wise as follows;

$$\begin{aligned}\vec{\kappa} &= (\kappa_x,\,\kappa_y,\,0) &&= \kappa_x\hat{\imath}+\kappa_y\hat{\jmath}\\ \vec{\nu} &= (\nu_x,\,\nu_y,\,0) &&= \nu_x\hat{\imath}+\nu_y\hat{\jmath}\end{aligned}$$

To closer examine this special case we first determine $\vec{\kappa} \times \vec{\nu}$;

$$\vec{\kappa} \times \vec{\nu} = \begin{vmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ \kappa_x & \kappa_y & 0 \\ \nu_x & \nu_y & 0 \end{vmatrix} = (\kappa_x \, \nu_y - \kappa_y \, \nu_x) \, \hat{k}$$

Thus we find that $\vec{\kappa} \times \vec{\nu}$ only produces a component along the z axis. If we are only concerned with the value of the z component, we can state that;

$$z \text{ component} = (\vec{\kappa} \times \vec{\nu}) \cdot \hat{k} = \kappa_x \nu_y - \kappa_y \nu_x = \|\vec{\kappa}\| \, \|\vec{\nu}\| \, \sin(\theta_r)$$

Or more simply;

$$z \text{ component} = \kappa_x \nu_y - \kappa_y \nu_x \tag{D.5}$$

Thus for the special case where the z components of $\vec{\kappa}$ and $\vec{\nu}$ are zero, we can express the area of the triangle between these vectors as;

$$A_t = \frac{1}{2} \left| \kappa_x \nu_y - \kappa_y \nu_x \right| \tag{D.6}$$

Note that equation D.5 is of particular significance, not only is its absolute value equal to twice the area of a triangle formed between $\vec{\kappa}$ and $\vec{\nu}$, but the sign associated with the result tells us something about how $\vec{\kappa}$ and $\vec{\nu}$ are oriented. Following the right hand rule, θ_r is defined as the angle from $\vec{\kappa}$ to $\vec{\nu}$, when their tails are joined. If θ_r is positive then equation D.5 will produce a positive value. Alternatively if θ_r corresponds to a negative angle, then equation D.5 will produce a negative value. Equation D.5 is useful and is used in two significant parts of this document.

D.2 Hansen and Levin

Hansen and Levin[20] provide an equivalent form of equations D.5, but in determinant form. First the determinant provides the z component;

$$z \text{ component} = \begin{vmatrix} 1 & c_x & c_y \\ 1 & d_x & d_y \\ 1 & e_x & e_y \end{vmatrix} \tag{D.7}$$

Thus the area A_t is expressed as;

$$A_t = \frac{1}{2} \operatorname{abs} \left(\begin{vmatrix} 1 & c_x & c_y \\ 1 & d_x & d_y \\ 1 & e_x & e_y \end{vmatrix} \right) \tag{D.8}$$

Note that the coordinates (c_x, c_y), (d_x, d_y), and (e_x, e_y) correspond to the three points C, D, and E. Only a small amount of work is necessary to show that equations D.8 and D.6 are equivalent. We start by evaluating the determinant and then solve for A_t.

$$A_t = \frac{1}{2} \left| \begin{array}{l} \;\; d_x\, e_y + c_x\, d_y + e_x\, c_y \\ -d_x\, c_y - e_x\, d_y - c_x\, e_y \end{array} \right| \tag{D.9}$$

Next we define $\vec{\kappa}$ and $\vec{\nu}$ as follows,

$$\begin{aligned} \vec{\kappa} &= D - C \\ \vec{\nu} &= E - C \end{aligned}$$

and substitute into equation D.6 to find that;

$$A_t = \frac{1}{2} \left| \begin{array}{l} \;\; d_x\, e_y + c_x\, d_y + e_x\, c_y \\ -d_x\, c_y - e_x\, d_y - c_x\, e_y \end{array} \right| \tag{D.10}$$

In comparing equations D.10 and D.9, we find that the expressions are identical, thus the expression given by Levin and Hansen is equivalent.

Note, since Hansen and Levin gave their equations in terms of the coordinates of points, if the points C, D, and E are taken as a sequence, then the sign of equation D.7 can be interpreted in the following way. If the result is found to be positive, then the sequence C, D, E follows a circular path in the *counter-clockwise* direction, see figure D.2. Conversely, if the result of equation D.7 is found to be negative, then the sequence C, D, E follows a circular path in the *clockwise* direction. Lastly, note that if equation D.7 is found to be zero, the points C, D, and E must be collinear and can be thought of being on a circle of infinite radius.

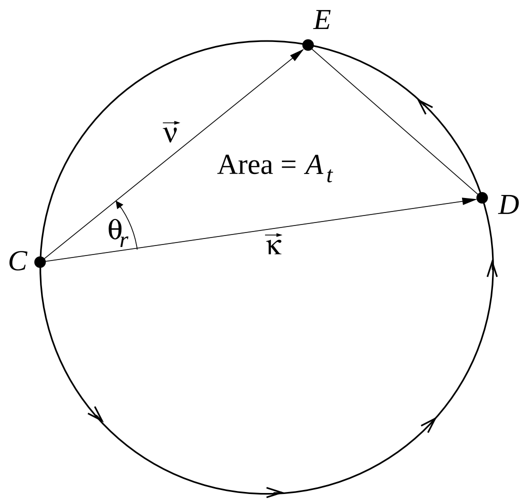


Figure D.2: Points on Circular Path

Appendix E

Derivation of FDTD Equations

This appendix contains a derivation of the FDTD equations that we used. The equations are for three dimensions and assume linear materials in a sourceless region. Further, for this derivation all variables are real valued. According to Li, Tassoudji, Shin and Kong[29] Maxwell's divergence equations are always satisfied in the FDTD scheme by ensuring that initial and boundary conditions are correctly applied. Thus the actual difference equations are based solely upon Maxwell's curl equations. As is typically done in this type of derivation, Maxwell's curl equations are first expressed in differential form, the constitutive relations have already been applied.

$$\nabla \times \vec{H} = \epsilon \frac{\partial \vec{E}}{\partial t} + \sigma \vec{E} \tag{E.1}$$

$$\nabla \times \vec{E} = -\mu \frac{\partial \vec{H}}{\partial t} - \sigma_m \vec{H} \tag{E.2}$$

The MKS system of units is used, see section 4.3. In performing this derivation, material properties in each cell are defined to be the diagonal terms of a matrix. This use of notation arose from the handling of material properties, see page 176 in section 4.1.2.

$$\underline{\underline{\epsilon}} = \begin{bmatrix} \epsilon_x & 0 & 0 \\ 0 & \epsilon_y & 0 \\ 0 & 0 & \epsilon_z \end{bmatrix} \qquad \underline{\underline{\sigma}} = \begin{bmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_z \end{bmatrix}$$

$$\underline{\underline{\mu}} = \begin{bmatrix} \mu_x & 0 & 0 \\ 0 & \mu_y & 0 \\ 0 & 0 & \mu_z \end{bmatrix} \quad \underline{\underline{\sigma_m}} = \begin{bmatrix} \sigma_{mx} & 0 & 0 \\ 0 & \sigma_{my} & 0 \\ 0 & 0 & \sigma_{mz} \end{bmatrix}$$

After incorporating the materials, equations E.1 and E.2 are now expressed as;

$$\nabla \times \vec{H} = \underline{\underline{\epsilon}} \frac{\partial \vec{E}}{\partial t} + \underline{\underline{\sigma}} \vec{E} \tag{E.3}$$

$$\nabla \times \vec{E} = -\underline{\underline{\mu}} \frac{\partial \vec{H}}{\partial t} - \underline{\underline{\sigma_m}} \vec{H} \tag{E.4}$$

In proceeding in the derivation, Maxwell's equations are first rewritten in the following form;

$$\frac{\partial \vec{E}}{\partial t} = -\underline{\underline{\epsilon}}^{-1} \, \underline{\underline{\sigma}} \vec{E} + \underline{\underline{\epsilon}}^{-1} \left(\nabla \times \vec{H} \right) \tag{E.5}$$

$$\frac{\partial \vec{H}}{\partial t} = -\underline{\underline{\mu}}^{-1} \, \underline{\underline{\sigma_m}} \vec{H} - \underline{\underline{\mu}}^{-1} \left(\nabla \times \vec{E} \right) \tag{E.6}$$

Schwab's[55] book provides a standard definition of the *curl* operator which is easily written in vector notation.

$$\nabla \times \vec{Q} = \left\{ \begin{array}{ccc} \frac{\partial}{\partial y} Q_z & - & \frac{\partial}{\partial z} Q_y \\ \frac{\partial}{\partial z} Q_x & - & \frac{\partial}{\partial x} Q_z \\ \frac{\partial}{\partial x} Q_y & - & \frac{\partial}{\partial y} Q_x \end{array} \right\}$$

The given expression for *curl* is used next to produce Maxwell's equations in a component-wise form, equation E.5 is rewritten as;

$$\frac{\partial}{\partial t} E_x = -\frac{\sigma_x}{\epsilon_x} E_x + \frac{1}{\epsilon_x} \left(\frac{\partial}{\partial y} H_z - \frac{\partial}{\partial z} H_y \right) \tag{E.7}$$

$$\frac{\partial}{\partial t} E_y = -\frac{\sigma_y}{\epsilon_y} E_y + \frac{1}{\epsilon_y} \left(\frac{\partial}{\partial z} H_x - \frac{\partial}{\partial x} H_z \right) \tag{E.8}$$

$$\frac{\partial}{\partial t}E_z = -\frac{\sigma_z}{\epsilon_z}E_z + \frac{1}{\epsilon_z}\left(\frac{\partial}{\partial x}H_y - \frac{\partial}{\partial y}H_x\right) \tag{E.9}$$

Likewise, equation E.6 is rewritten in a component-wise form as follows;

$$\frac{\partial}{\partial t}H_x = -\frac{\sigma_{mx}}{\mu_x}H_x + \frac{1}{\mu_x}\left(\frac{\partial}{\partial z}E_y - \frac{\partial}{\partial y}E_z\right) \tag{E.10}$$

$$\frac{\partial}{\partial t}H_y = -\frac{\sigma_{my}}{\mu_y}H_y + \frac{1}{\mu_y}\left(\frac{\partial}{\partial x}E_z - \frac{\partial}{\partial z}E_x\right) \tag{E.11}$$

$$\frac{\partial}{\partial t}H_z = -\frac{\sigma_{mz}}{\mu_z}H_z + \frac{1}{\mu_z}\left(\frac{\partial}{\partial y}E_x - \frac{\partial}{\partial x}E_y\right) \tag{E.12}$$

Maxwell's equations will be expressed by using a finite difference approximation of partial derivatives. The following finite difference approximations will be used for updating x component field values. The term *central differencing* is sometimes used to describe this overall FDTD technique, since we apply the concept of central differences in time and space. The use of central difference equations allows the FDTD method to have an accuracy on the order of Δ^2

For Updating Ex

The following are the difference equations that were used to develop the equations for updating the x components of electric fields.

$$\frac{\partial E_x^{n+\frac{1}{2}}[i,j,k]}{\partial t} \approx \frac{E_x^{n+1}[i,j,k] - E_x^{n}[i,j,k]}{\Delta t}$$

$$\frac{\partial H_z^{n+\frac{1}{2}}[i,j,k]}{\partial y} \approx \frac{H_z^{n+\frac{1}{2}}[i,j,k] - H_z^{n+\frac{1}{2}}[i,j-1,k]}{\Delta y}$$

$$\frac{\partial H_y^{n+\frac{1}{2}}[i,j,k]}{\partial z} \approx \frac{H_y^{n+\frac{1}{2}}[i,j,k] - H_y^{n+\frac{1}{2}}[i,j,k-1]}{\Delta z}$$

For Updating Hx

The following are the difference equations that were used to develop the equations for updating the x components of magnetic fields.

$$\frac{\partial H_x^{n}[i,j,k]}{\partial t} \approx \frac{H_x^{n+\frac{1}{2}}[i,j,k] - H_x^{n-\frac{1}{2}}[i,j,k]}{\Delta t}$$

$$\frac{\partial E_y^n[i,j,k]}{\partial z} \approx \frac{E_y^n[i,j,k+1] - E_y^n[i,j,k]}{\Delta z}$$

$$\frac{\partial E_z^n[i,j,k]}{\partial y} \approx \frac{E_z^n[i,j+1,k] - E_z^n[i,j,k]}{\Delta y}$$

Using the above expressions, equation E.7 is next converted to finite difference form. While the expressions are presented for the x directed components, equivalent expressions are easily made for the other components by simply changing the appropriate subscripts. Note that in the following equations, to reduce clutter, the subscript x in σ_x, ϵ_x, σ_{mx}, and μ_x is not written. The use of the subscript x should nevertheless be understood. In returning to the derivation, the central difference equations are first substituted into equation E.7 to yield;

$$\begin{aligned}\tfrac{E_x^{n+1}[i,j,k]-E_x^n[i,j,k]}{\Delta t} &= -\tfrac{\sigma}{\epsilon}E_x^{n+\frac{1}{2}}[i,j,k] \\ &+\tfrac{1}{\epsilon}\left(\tfrac{H_z^{n+\frac{1}{2}}[i,j,k]-H_z^{n+\frac{1}{2}}[i,j-1,k]}{\Delta y} - \tfrac{H_y^{n+\frac{1}{2}}[i,j,k]-H_y^{n+\frac{1}{2}}[i,j,k-1]}{\Delta z}\right)\end{aligned}$$

Note that since electric field components are not actually defined at half time steps, E_x is approximated as;

$$E_x^{n+\frac{1}{2}}[i,j,k] \approx \frac{1}{2}\left(E_x^{n+1}[i,j,k] + E_x^n[i,j,k]\right)$$

The updated term is brought to the left side of the equal sign, the previous value is moved to the right;

$$\begin{aligned}\left(1+\tfrac{\sigma\,\Delta t}{2\epsilon}\right)E_x^{n+1}[i,j,k] &= \left(1-\tfrac{\sigma\,\Delta t}{2\epsilon}\right)E_x^n[i,j,k]+ \\ &\tfrac{\Delta t}{\epsilon}\left(\tfrac{H_z^{n+\frac{1}{2}}[i,j,k]-H_z^{n+\frac{1}{2}}[i,j-1,k]}{\Delta y} - \tfrac{H_y^{n+\frac{1}{2}}[i,j,k]-H_y^{n+\frac{1}{2}}[i,j,k-1]}{\Delta z}\right)\end{aligned}$$

This last expression is simplified to produce the following expression,

$$\begin{aligned}E_x^{n+1}[i,j,k] &= \tfrac{2\epsilon-\sigma\,\Delta t}{2\epsilon+\sigma\,\Delta t}E_x^n[i,j,k] \\ &+\tfrac{2\,\Delta t}{2\epsilon+\sigma\,\Delta t}\left(\tfrac{H_z^{n+\frac{1}{2}}[i,j,k]-H_z^{n+\frac{1}{2}}[i,j-1,k]}{\Delta y} - \tfrac{H_y^{n+\frac{1}{2}}[i,j,k]-H_y^{n+\frac{1}{2}}[i,j,k-1]}{\Delta z}\right)\end{aligned}$$

The constants KE_ℓ and KE_c are defined as follows;

$$\mathrm{KE}_\ell = \frac{2\epsilon - \sigma\,\Delta t}{2\epsilon + \sigma\,\Delta t} \tag{E.13}$$

$$\mathrm{KE}_c = \frac{2\,\Delta t}{2\epsilon + \sigma\,\Delta t} \tag{E.14}$$

Note that KE_ℓ can be rewritten along these lines to another form;

$$\mathrm{KE}_\ell = \frac{2\epsilon_r\epsilon_o - \sigma\,\Delta t}{2\epsilon_r\epsilon_o + \sigma\,\Delta t} = \frac{2\epsilon_r - \sigma\frac{\Delta t}{\epsilon_o}}{2\epsilon_r + \sigma\frac{\Delta t}{\epsilon_o}} = \frac{2\epsilon_r - \sigma\sqrt{\frac{\mu_o}{\epsilon_o}}\frac{1}{\sqrt{\mu_o\epsilon_o}}\Delta t}{2\epsilon_r + \sigma\sqrt{\frac{\mu_o}{\epsilon_o}}\frac{1}{\sqrt{\mu_o\epsilon_o}}\Delta t}$$

$$\mathrm{KE}_\ell = \frac{2\epsilon_r - \sigma\eta_o\Delta\tau}{2\epsilon_r + \sigma\eta_o\Delta\tau} \tag{E.15}$$

Where clearly the following is defined as;

$$\Delta\tau = \frac{1}{\sqrt{\mu_o\epsilon_o}}\Delta t$$

$$\eta_o = \sqrt{\frac{\mu_o}{\epsilon_o}}$$

Similarly, KE_c can be rewritten along these lines;

$$\mathrm{KE}_c = \frac{2\,\Delta t}{2\epsilon_r\epsilon_o + \sigma\,\Delta t} = \frac{2\,\Delta t}{2\epsilon_r\sqrt{\frac{\epsilon_o}{\mu_o}}\sqrt{\mu_o\epsilon_o} + \sigma\,\Delta t} = \left(\sqrt{\frac{\mu_o}{\epsilon_o}}\right)\frac{2\frac{\Delta t}{\sqrt{\mu_o\epsilon_o}}}{2\epsilon_r + \sigma\frac{\Delta t}{\sqrt{\mu_o\epsilon_o}}\sqrt{\frac{\mu_o}{\epsilon_o}}}$$

$$\mathrm{KE}_c = \eta_o\frac{2\,\Delta\tau}{2\epsilon_r + \sigma\eta_o\,\Delta\tau} \tag{E.16}$$

Equations E.15 and E.16 are the same exact constants as presented by Li, Tassoudji, Shin and Kong[29]. In any case, equations E.15 and E.16 are actually equivalent to E.13 and E.14.

Returning to the derivation, the equation for updating E_x is presented with equations used to update the remaining electric field components, which are derived in a similar fashion.

$$\begin{aligned} E_x^{n+1}[i,j,k] = \;& \mathrm{KE}_{\ell x}E_x^n[i,j,k] + \\ & \mathrm{KE}_{cx}\left(\frac{H_z^{n+\frac{1}{2}}[i,j,k] - H_z^{n+\frac{1}{2}}[i,j-1,k]}{\Delta y} - \frac{H_y^{n+\frac{1}{2}}[i,j,k] - H_y^{n+\frac{1}{2}}[i,j,k-1]}{\Delta z}\right)\end{aligned} \tag{E.17}$$

$$\begin{aligned} E_y^{n+1}[i,j,k] = \;& \mathrm{KE}_{\ell y}E_y^n[i,j,k] + \\ & \mathrm{KE}_{cy}\left(\frac{H_x^{n+\frac{1}{2}}[i,j,k] - H_x^{n+\frac{1}{2}}[i,j,k-1]}{\Delta z} - \frac{H_z^{n+\frac{1}{2}}[i,j,k] - H_z^{n+\frac{1}{2}}[i-1,j,k]}{\Delta x}\right)\end{aligned} \tag{E.18}$$

$$\begin{aligned} E_z^{n+1}[i,j,k] = {} & \mathrm{KE}_{\ell z} E_z^n[i,j,k] + \\ & \mathrm{KE}_{cz}\left(\tfrac{H_y^{n+\frac{1}{2}}[i,j,k]-H_y^{n+\frac{1}{2}}[i-1,j,k]}{\Delta x} - \tfrac{H_x^{n+\frac{1}{2}}[i,j,k]-H_x^{n+\frac{1}{2}}[i,j-1,k]}{\Delta y}\right) \end{aligned} \tag{E.19}$$

For the above equations, constants associated with arbitrary axis q are defined as follows;

$$\mathrm{KE}_{\ell q} = \frac{2\epsilon_q - \sigma_q\,\Delta t}{2\epsilon_q + \sigma_q\,\Delta t}$$
$$\mathrm{KE}_{cq} = \frac{2\,\Delta t}{2\epsilon_q + \sigma_q\,\Delta t}$$

Next we will finish deriving the equations for updating the magnetic fields. Using the central difference approximations, equation E.10 is next converted to a usable finite difference form. Equations E.11 and E.12 are handled in a similar way. Remember that to reduce clutter in this derivation, the subscript x associated with μ_x and σ_{mx} is not written. In converting equation E.10, substitutions are first made;

$$\begin{aligned} \tfrac{H_x^{n+\frac{1}{2}}[i,j,k]-H_x^{n-\frac{1}{2}}[i,j,k]}{\Delta t} = {} & -\tfrac{\sigma_m}{\mu}H_x^n[i,j,k] \\ & +\tfrac{1}{\mu}\left(\tfrac{E_y^n[i,j,k+1]-E_y^n[i,j,k]}{\Delta z} - \tfrac{E_z^n[i,j+1,k]-E_z^n[i,j,k]}{\Delta y}\right) \end{aligned}$$

Note that since magnetic field values are not defined for whole time steps, we can approximate the value of H_x^n as follows;

$$H_x^n[i,j,k] \approx \frac{1}{2}\left(H_x^{n+\frac{1}{2}}[i,j,k] + H_x^{n-\frac{1}{2}}[i,j,k]\right)$$

Next we move the updated term to the left of the equal sign and the old term to the right of the equal sign to get;

$$\begin{aligned} \left(1+\tfrac{\sigma_m\,\Delta t}{2\mu}\right)H_x^{n+\frac{1}{2}}[i,j,k] = {} & \left(1-\tfrac{\sigma_m\,\Delta t}{2\mu}\right)H_x^{n-\frac{1}{2}}[i,j,k] \\ & +\tfrac{\Delta t}{\mu}\left(\tfrac{E_y^n[i,j,k+1]-E_y^n[i,j,k]}{\Delta z} - \tfrac{E_z^n[i,j+1,k]-E_z^n[i,j,k]}{\Delta y}\right) \end{aligned}$$

The last expression is simplified to yield;

$$\begin{aligned} H_x^{n+\frac{1}{2}}[i,j,k] = {} & \tfrac{2\mu-\sigma_m\,\Delta t}{2\mu+\sigma_m\,\Delta t}H_x^{n-\frac{1}{2}}[i,j,k] \\ & +\tfrac{2\,\Delta t}{2\mu+\sigma_m\,\Delta t}\left(\tfrac{E_y^n[i,j,k+1]-E_y^n[i,j,k]}{\Delta z} - \tfrac{E_z^n[i,j+1,k]-E_z^n[i,j,k]}{\Delta y}\right) \end{aligned}$$

The constants KH_ℓ and KH_c are defined as follows;

$$\mathrm{KH}_\ell = \frac{2\mu - \sigma_m\,\Delta t}{2\mu + \sigma_m\,\Delta t} \tag{E.20}$$

$$\mathrm{KH}_c = \frac{2\,\Delta t}{2\mu + \sigma_m\,\Delta t} \tag{E.21}$$

Note that KE_ℓ can be rewritten along these lines to another form;

$$\mathrm{KH}_\ell = \frac{2\mu_r\mu_o - \sigma_m\,\Delta t}{2\mu_r\mu_o + \sigma_m\,\Delta t} = \frac{2\mu_r - \sigma_m\frac{\Delta t}{\mu_o}}{2\mu_r + \sigma_m\frac{\Delta t}{\mu_o}} = \frac{2\mu_r - \sigma_m\sqrt{\frac{\epsilon_o}{\mu_o}}\frac{\Delta t}{\sqrt{\mu_o\epsilon_o}}}{2\mu_r + \sigma_m\sqrt{\frac{\epsilon_o}{\mu_o}}\frac{\Delta t}{\sqrt{\mu_o\epsilon_o}}}$$

$$\mathrm{KH}_\ell = \frac{2\mu_r - \sigma_m\Delta\tau/\eta_o}{2\mu_r + \sigma_m\Delta\tau/\eta_o} \tag{E.22}$$

Following similar lines, KE_c is rewritten to another form as well;

$$\mathrm{KH}_c = \frac{2\,\Delta t}{2\mu_r\mu_o + \sigma_m\,\Delta t} = \frac{2\,\Delta t}{2\mu_r\sqrt{\frac{\mu_o}{\epsilon_o}}\sqrt{\mu_o\epsilon_o} + \sigma_m\,\Delta t} = \frac{2\frac{\Delta t}{\sqrt{\mu_o\epsilon_o}}}{\sqrt{\frac{\mu_o}{\epsilon_o}}\left(2\mu_r + \sigma_m\frac{\Delta t}{\sqrt{\mu_o\epsilon_o}}\sqrt{\frac{\epsilon_o}{\mu_o}}\right)}$$

$$\mathrm{KH}_c = \frac{2\,\Delta\tau}{\eta_o\left(2\mu_r + \sigma_m\,\Delta\tau/\eta_o\right)} \tag{E.23}$$

Equations E.22 and E.23 are identical to the constants defined by Li, Tassoudji, Shin and Kong[29]. In any instance, equations E.22 and E.23 are equivalent to E.20 and E.21.

Returning to the derivation, the equations used for updating H_x is presented with equations used to update the remaining magnetic field components, which have been derived in a similar fashion.

$$\begin{aligned} H_x^{n+\frac{1}{2}}[i,j,k] = {} & \mathrm{KH}_{\ell x} H_x^{n-\frac{1}{2}}[i,j,k] + \\ & \mathrm{KH}_{cx}\left(\tfrac{E_y^n[i,j,k+1] - E_y^n[i,j,k]}{\Delta z} - \tfrac{E_z^n[i,j+1,k] - E_z^n[i,j,k]}{\Delta y}\right) \end{aligned} \tag{E.24}$$

$$\begin{aligned} H_y^{t+\frac{1}{2}}[i,j,k] = {} & \mathrm{KH}_{\ell y} H_y^{n-\frac{1}{2}}[i,j,k] + \\ & \mathrm{KH}_{cy}\left(\tfrac{E_z^n[i+1,j,k] - E_z^n[i,j,k]}{\Delta x} - \tfrac{E_x^n[i,j,k+1] - E_x^n[i,j,k]}{\Delta z}\right) \end{aligned} \tag{E.25}$$

$$\begin{aligned} H_z^{n+\frac{1}{2}}[i,j,k] = {} & \mathrm{KH}_{\ell z} H_z^{n-\frac{1}{2}}[i,j,k] + \\ & \mathrm{KH}_{cz}\left(\tfrac{E_x^n[i,j+1,k] - E_x^n[i,j,k]}{\Delta y} - \tfrac{E_y^n[i+1,j,k] - E_y^n[i,j,k]}{\Delta x}\right) \end{aligned} \tag{E.26}$$

Constants are defined for the equations that update magnetic field components. For an arbitrary axis q;

$$\mathrm{KH}_{\ell q} = \frac{2\mu_q - \sigma_{mq}\,\Delta t}{2\mu_q + \sigma_{mq}\,\Delta t}$$

$$\mathrm{KH}_{cq} = \frac{2\,\Delta t}{2\mu_q + \sigma_{mq}\,\Delta t}$$

E.1 Special Case of Free Space

For the special case of free space, KE_ℓ, KE_c, KH_ℓ and KH_c all simplify since the magnetic and electric conductivities of free space is implicitly zero. Reviewing equations E.13, E.14, E.20, E.21, E.16, and E.23, the following is noted for the special case of free space;

$$\mathrm{KE}_\ell = 1$$

$$\mathrm{KH}_\ell = 1$$

$$\mathrm{KE}_c = \frac{\Delta t}{\epsilon_o} = \eta_o\,\Delta\tau$$

$$\mathrm{KH}_c = \frac{\Delta t}{\mu_o} = \frac{\Delta\tau}{\eta_o}$$

Where as before;

$$\Delta\tau = \frac{\Delta t}{\sqrt{\mu_o\epsilon_o}}$$

$$\eta_o = \sqrt{\frac{\mu_o}{\epsilon_o}}$$

E.2 Appendix Summary

In summary, equations E.17, E.18 and E.19 are the difference equations for updating electric field components at whole time steps. Equations E.24, E.25 and E.26 are the difference equations for updating magnetic field components at half time steps.

Appendix F

Users Manual

Research was performed with the overall goal of developing a high performance package for performing finite difference time domain (FDTD) analysis of electromagnetic phenomenon. The research resulted in a software package consisting of an orthogonal mesh generator, a FDTD analysis program, several utility programs, and file based interface software for using two commercially available programs. Ideas and Patran are two commercially available programs that are particularly useful for generating models of solid objects as well as for visualizing results. It should be pointed out that this user manual is primarily concerned with software written at Worcester Polytechnic Institute. It is assumed that the user is already somewhat familiar with Ideas and Patran. This manual only goes so far as to provide hints regarding use of Ideas and Patran.

Apart from the commercial software, the software package is a collection of serial as well as parallel code. Serial code was written in ANSI standard 'C'. The serial code should execute on any platform that supports an ANSI compatible compiler and provides adequate memory. The parallel code was written explicitly for the MasPar SIMD environment and was only tested on a MasPar, DECstation combination. Parallel as well as serial versions of the mesh generator were developed. At this point only the parallel FDTD analysis software is included in the package.

This user manual introduces topics in an order as would be encountered in a "typical" FDTD analysis. For a first reading it is strongly suggested that a user quickly read through the entire document. Figure F.1 presents a helpful outline that the user should examine closely. For a second reading it is suggested that the user read each section with a particular example in mind. Section F.9 contains an example that should be helpful.

Credit is given to Shipeng Li, who wrote the first version of the universal to solid file conversion program ‘fconv2’. Credit is also given to Guanghua Peng, who wrote and maintains the boundary condition assignment program ‘bccode’. Jonathan Hill wrote the mesh generation, parallel FDTD solver and other utility programs described in this document.

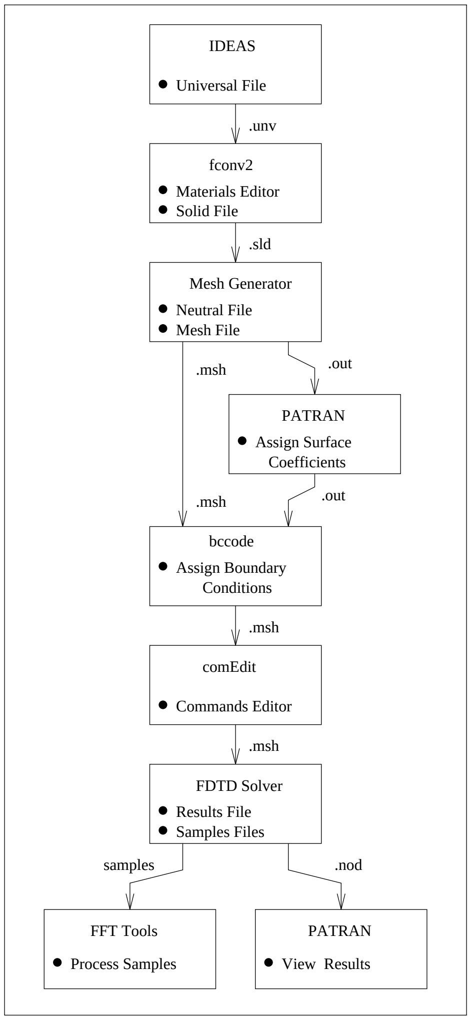


Figure F.1: Diagram of Outline

F.1 Managing Ideas

Ideas is currently the program we use to make solid models. A program called ‘fconv2’ converts from the Ideas universal format to the particular solid format associated with the orthogonal mesh generator. The only reason we currently use

Ideas is that the software for converting from universal format is already available. Patran is also capable of solid modeling but unfortunately Patran does not produce *universal* files.

Inside Ideas, solid modeling will most likely be performed in the *solid modeling* family, and most likely by the *object modeling* task. Note that objects must be stored in bins. This point is significant; If you create a model and don't use the bins, when you write a universal file it will not contain your model. Menus associated with bins are found under the *manage stored* menu. This is also where you find the *write* menu to request that a universal file be written.

The names that you assign to bins will later be interpreted as material property names, thus the user is strongly urged to select meaningful names that will be easily recognizable. Unfortunately the default bin 'MAIN' cannot have its name changed. Users are not required use the default bin however. Lastly it is suggested that when storing objects, users pick object names that will be recognizable.

In addition to a bin name, every bin has an associated bin number. The bin number will later be referred to as the *material identifier.* It is important to note that the mesh generator code uses a method of precedence in determining which material identifier to assign to each mesh box. The precedence method is simply stated, if two objects are drawn in Ideas so that they share a common space, the object with the larger material identifier will be assigned the boxes in the region where both objects coincide. It is strongly suggested that users keep this point in mind.

While the precedence method employed by the mesh generator was implemented as a means to guarantee that results be predictable, the precedence method is easily used to describe a constructive geometry. For example to describe a hollow box, enclose one block inside another block.

F.2 Conversion to Solid Format

As described earlier, the program 'fconv2' converts from universal to solid. The program fconv2 will accept either complete file names or the base of a file name. For example 'sample.unv' is a file name, while 'sample' is the base associated with that file name. If you specify the filename base, fconv2 will automatically attach the '.unv' extension. Unless you specify a second filename, the solid file produced will have the same filename base but will have the '.sld' extension to indicate that the file is of solid type. Start fconv2 by entering 'fconv2' or by entering 'fconv2 filename' at the system prompt.

A second task performed by the conversion software is the handling of material properties. In the current package implementation, the only indication of materials inside Ideas is the unique bin names that users assign. Objects in a given bin are assumed to be made of a material identified by the bin number and associated bin name. The conversion program 'fconv2' has a material editor to allow the user to enter constants associated with each bin name. A user enters values for relative permittivity, relative permeability, electric conductivity and magnetic conductivity.

The user will note that the material editor will always have a default material property with material identifier zero, this default material identifier is interpreted by the mesh generator to correspond to free space. The material editor can readily handle isotropic or diagonal tensor materials. Permittivity and permeability materials can have real or complex values. All materials are initially given default constant values equivalent to that of free space, each material is updated as the user enters values.

The material editor is driven by single character commands. For an informative description of all commands the user simply enters the '?' command at the 'choice:' prompt. The user can exit the material editor at any time with the 'x' command.

F.3 Mesh Generation

Two versions of the mesh generator are provided, a serial version as well as a parallel version. A makefile will construct either version for you. To start the serial version with interactive mode enter the command 'mesh_ser'. Likewise to start the parallel version with interactive mode enter the command, 'mesh_par'. In the interactive mode the mesh generator first announces that it is running, tells the user whether the version is serial or parallel then asks a series of questions. A typical session with the serial mesh generator is presented.

```
vlsi5> mesh_ser
Orthogonal Mesh Generator; Serial version
Please Enter a File Name or Base Name of a Solid File
Name: duo.sld
Enter the number of uniform boxes that you are
requesting along each respective axis.
X boxes: 10
Y boxes: 10
Z boxes: 10
```

```
Next you will be asked a series of questions, unless you are
specifying a file name, answer each question with ‘y’ for yes
or ‘n’ for no.
Do you want a text visualization? (y/n): y
Accept file name ‘duo_vis.txt’?
(y/n): y

Do you want a universal file visualization? (y/n): n

Do you want a Patran file generated? (y/n): y
Would you like your Patran file made solid for analysis?
If you specify no, the model produced will be ‘hollow’.
(y/n): n
Include the bounding box in your visualization? (y/n): y
Accept file name ‘duo_phm.out’?
(y/n): y

Do you want a mesh file?
(y/n): y
Accept file name ‘duo.msh’?
(y/n): y

If you would rather not start mesh generation now but make an autostart
file, please indicate with ‘y’ for yes.
(y/n): y
Accept file name ‘duo.aum’?
(y/n): y
auto file ‘duo.aum’ written
Mesh program exiting
vlsi5>
```

The mesh generator also allows users to optionally enter only the base-name of a file. If the base name is entered in the command line as a formal argument, then the mesh generator will automatically append the ‘.aum’ extension which is usually associated with an *autostart file*. Alternatively, the user can enter a full name including any arbitrary extension in which case the ‘.aum’ extension will not be added. If the user does not provide a formal argument when the mesh generator is called, the user will be prompted for a file name. After the prompt, if the user provides the base of a file name, the ‘.sld’ extension will automatically be added which usually corresponds to a solid file. Again, the user is free to enter a complete file name in which case the ‘.sld’ extension will not be added.

The mesh generator starts by examining the first line of a specified file, to determine the file type. If the file specified is an autostart file, the mesh generator loads the rest of the autostart file then proceeds in the non-interactive mode. Conversely, if the file specified by the user is a solid file, then the mesh generator enters the interactive mode. In the interactive mode the mesh generator asks the user a series of questions, starting with the requested number of mesh boxes along each coordinate axis. This information is used to determine the required mesh density.

Next the mesh generator asks a series of questions to determine what output files should be generated. The text and universal visualization files are meant solely for visualization purposes. The text visualization file is particularly useful for visualizing reasonably sized meshes. Patran neutral files are particularly useful, in addition to helping a user to visualize a mesh, Patran files are useful for assigning FDTD boundary conditions and visualizing results. For assigning boundary conditions it is recommended that the user select a hollow model but include the default material bounding box, as illustrated in the example. By producing a *hollow file*, the resulting neutral file and Patran database will be significantly smaller in size than if a complete neutral file were to be generated. A mesh file produced will have components added later and will become the input file to the FDTD solver.

Eventually the mesh generator will ask if the user wishes to have an autostart file written. An autostart file saves results from questions that the user replied to and is particularly useful if you plan to run the mesh generator later, perhaps in a batch type environment. An autostart file contains all the parameters needed to perform a mesh analysis. If you choose to have an autostart file generated, the mesh generator exits after the autostart file is written.

The next example illustrates how the mesh generator can be started in the non-interactive mode. In this case the parallel version was executed. Note that all returning comments were written to 'stderr'.

```
goofy> mesh_par duo.aum
number unresolved = 0
number unresolved = 0
Text visual 'duo_vis.txt' written
Mesh file 'duo.msh' written
writing neutral file 'duo_phm.out'
23-Dec-94   16:26:08
Analysis took 0 seconds.
Including file write time is 2 seconds.
goofy>
```

It is important to point out that the parallel version of the mesh generator can only be executed on a workstation that has an attached MasPar data parallel unit. For both versions of the mesh generator, the *number unresolved* figure reported refers to the number of unresolved mesh boxes that were encountered during the mesh generation process, a report is made for each object that is analyzed. It is not unusual to have one, two or perhaps three unresolved boxes in the mesh. The mesh generator has a special function that can take care of small numbers of unresolved mesh boxes. A large number of unresolved boxes would exceed say one percent of the total number of boxes in a mesh. Such a situation indicates that there is a serious problem in your mesh analysis. The normal number of unresolved boxes is typically zero.

It has been found that is useful to report the analysis execution time separately from the total execution time. File write time can be very significant as neutral files can be exceedingly large.

F.4 Assigning Boundary Conditions

As indicated in section F.3, when running the mesh generator a Patran neutral file can be generated. A Patran file is used as a vehicle for Patran to describe boundary conditions. Unfortunately since the orthogonal mesh generator does not preserve contours, it is not possible to define boundary conditions at the same time that solid modeling is performed. Thus the earliest opportunity that we have to assign boundary conditions is after the orthogonal mesh has been generated.

One point needs to be made, since the FDTD solver must have boundary conditions specified, default boundaries are assigned by the FDTD solver. In other words, wherever the user does not specify boundary conditions on the bounding box, the parallel FDTD solver will assign an arbitrary boundary condition. The arbitrary value is by default that of a perfectly electric conducting (PEC) surface. Clearly any circumstance where boundary conditions are not specified on the bounding box represents an indeterminate state in the FDTD solver. The incorporation of default boundary condition guarantees that the bounding box will always have boundary conditions specified.

The user is free to assign boundary conditions to object surfaces inside the bounding box or to surfaces of the bounding box. Note that boundary conditions corresponding to perfectly magnetic conductive (PMC) material can only be assigned to surfaces of the bounding box. The activity of assigning boundary conditions to the bounding box over-rides default values that would otherwise be assigned. The use of the default boundary condition can be seen as a time saver, since the default boundary condition is in effect "already there" on the bounding box.

To assign boundary conditions you will need to start Patran, create a new database, then import the Patran neutral file that the mesh generator produced. Boundary conditions are marked by assigning convection coefficients to surfaces of the Patran model. You must select an analysis preference that supports convection coefficients, one such analysis code is ‘P3/FEA’. If you select this preference, make sure to select the ‘thermal’ analysis type.

The ‘load/BC’ menu is used to assign the convection coefficients. Make sure to select the convection object. In the ‘Input Data’ sub-menu it is recommended that you select arbitrary whole numbers for each boundary condition type that you mark. You might select say, 50 for excitation and 100 for PMC type boundary conditions. In the ‘Select Application Region’ make sure to select ‘FEM Geometry Filter’, likewise in the select menu click on the ‘visible entities’ switch. Lastly make sure that the icon corresponding to ‘Select a Free Face of an Element’ is highlighted.

Note that the ‘groups’ and ‘select’ menus are particularly useful for separating a model into components by material type. This is useful if you need to assign a boundary condition to an object surface inside your model.

Note that since convection packets are writable and readable, such files can be saved and allow for changes to convection coefficients later. Once you exit Patran make sure to rename the neutral file. The program that actually assigns boundary conditions is expecting the ‘.out’ extension, rather than the ‘.out.1’ or ‘.1’ extension that Patran sticks on.

The program ‘bccode’ is responsible for actually assigning boundary conditions to your mesh model. To call bccode use the following syntax;

```
bccode neutral_base mesh_base unit_conversion
```

Note that bccode will automatically append the ’.out’ extension to the neutral file base name and the ’.msh’ extension to the mesh file base name. The unit conversion is a remnant from Ideas. If you stored your universal file in MKS units then the unit conversion is simply ‘1.0’.

The program bccode first reads the neutral file, then asks what type of boundary condition corresponds to each of the convection coefficients found in the neutral file. Note that in addition to specifying that a surface is to be excited you must also indicate the type of surface. It is perfectly acceptable for an excitation surface to correspond to free space, but such a surface is only allowed inside the mesh model.

Note that the current parallel FDTD solver does not support Mur type absorbing boundary conditions. At one point Mur absorbing boundaries were supported.

Some work would be needed to re-introduce Mur absorbing boundaries. In the absence of Mur absorbing boundaries, an appropriately chosen isotropic absorbing material can be easily selected to provide absorbing boundary conditions. Such an isotropic absorbing material has permittivity and permeability values corresponding to free space, but includes electric and magnetic conductivities. The conductivities are selected such that the material has the same impedance as free space.

F.5 Simulation Directives

The last bit of work needed to prepare for a FDTD analysis is to assign simulation directives to the mesh file. Simulation directives are used to describe the number of time steps to perform, the time step size, the type of excitation to use and how to report results. For excitation, the current parallel code supports real sinusoidal and real Gaussian pulse excitation. Results can be posted to Patran readable files or samples of a field component can be dumped to files.

Simulation directives are entered by using the commands editor. The commands editor is interactive in the sense that it asks a series of questions that a user responds to. The commands editor follows a script in a file named 'com_script.txt', the script must be available for the commands editor to function. To start the commands editor, enter;

```
comEdit file-name
```

The file-name may be the entire file name or just the base part of the file name. If a file name is not given the commands editor will ask for a file name. Note that the commands editor automatically makes a back-up file.

If you should need to exit the commands editor, the easiest way is to enter a bogus entry for a number, 'bogus' for example, then reply that you do not wish to try again. Note that when starting the commands editor your mesh file is first renamed to 'file-name.bak'. The commands editor first makes a new mesh file with your original mesh file name. By exiting the commands editor in the way described above, the new file will be deleted and 'file-name.bak' will be renamed such that it again has the original file name. If you were to exit the commands editor by simply issuing control-c, then the new mesh file will be left in an unfinished state. You will have to rename 'file.bak' manually, otherwise the file may be lost.

Note that if a file already has had commands inserted, there is no danger in running the command editor again; Before inserting new commands the old commands are neatly removed, thus a user can use the commands editor to repeatedly make changes to the same file. Unfortunately however after commands are inserted, the file format is changed just slightly so that 'bccode' will not work. A user should run 'bccode' before running the commands editor.

F.6 Parallel FDTD Solver

Start the FDTD solver by entering, 'FDTDsolve'. The solver will ask for a file name, make sure to enter the entire file name with extension. Note that the parallel FDTD solver can only be executed on a machine with an attached MasPar data parallel unit.

F.7 Results for Patran

Patran files are automatically given the standard '.nod' extension. Data is easily imported into Patran using the 'E_VEC.res_tmpl' template file that is included with the FDTD tools.

Before a results file can be imported into Patran, a corresponding finite element model must be loaded into Patran. There are at least three ways to accomplish this. The most direct method is to instruct Patran to make a finite element model. Examples of this type of procedure can be found in the Patran documentation.

The second method to produce a finite element model is somewhat obvious. The mesh generator can be instructed to produce a neutral file that will contain a complete finite element model. The neutral file is simply imported before loading the results files. The third method is similar to the second, the program 'make_neutral' was specifically written to generate such neutral files. The program 'make_neutral' is interactive and is particularly useful for quickly generating a small number of layers of a mesh. The program make_neutral is found with the FDTD tools and is started by entering 'make_neutral' at the system prompt.

F.8 Examining Samples

While running the commands editor, at most two locations in the mesh file can be specified for sampling data. For each location a single component value is reported to a file for every time step. The file names specified will not have any extension added. To process sample files, several small useful programs were written and are summarized in the following table.

Sampling Tools

no.	name and use	description
1	adjlength newlength [file]	Adjust length of data set, if made larger then inserts zeros.
2	radix2 [file]	A fast Fourier transform
3	plotreal [file]	Produce plot of real part, readable by xgraph
4	plotmag [file]	Produce magnitude plot, readable by xgraph
5	dlog Afile [Bfile]	Produce plot of ratio dB, Bfile divided by Afile, readable by xgraph.
6	makeCos	Produces samples of a cosine signal.

One reason why a user might wish to change the length of a data set would be to alter the resolution of a discrete Fourier transform. Note that besides ‘adjlength’, the fast Fourier transform will automatically adjust the data set to have a length that is a power of 2. Both ‘plotreal’ and ‘plotmag’ can be used to make plots of sample data or plots of discrete transform data. Note if ‘file’ is not specified, these programs will expect data on standard input. Note that for ‘dlog’, ‘Afile’ must be specified. If ‘Bfile’ is not specified then input will be expected on standard input.

All of the programs listed in the above table produce output on standard output which can easily be redirected to a file or piped to another application. With the exception of ‘MakeCos’, if a file name is not specified then these programs will expect an input file to appear on standard input, thus these programs can easily be made to work together. For example to adjust the length of an input data file, take the fast Fourier transform and then produce a magnitude plot readable by xgraph a user might call each program, one at a time. Alternatively all three programs can be called in a single line. In this example a file named ‘file.dat’ is adjusted to have 1024 entries, then is transformed. A magnitude plot is made from the transform data. Lastly the magnitude plot is saved in ‘file.xgf’, which incidentally is the only file created.

```
adjlength 1024 file.dat | radix2 | plotmag > file.xgf
```

The program ‘makeCos’ was included primarily for verification, to show that the fast Fourier transform program works correctly. Note that ‘radix2’ is based on a decimation in time fast Fourier transform written in ‘C’. Jonathan Hill translated the transform from the Fortran code presented by Oppenheim and Schafer on page 608 of *Discrete–Time Signal Processing,* published by Prentice Hall, copyright 1989. Lastly, ‘xgraph’ is a simple plotting package written by David Harrison, at the University of California.

F.9 An Example

For this introductory example, a sinusoidal pulse is applied at one end of a rectangular tube. Absorbing material at the opposite end should remove most of the excitation.

F.9.1 The Test Chamber

Figure F.2 is an illustration of the test chamber used in this example. Note that the test chamber is constructed using two blocks, each of different material. The properties of each material is summarized in the table below. Note that such material properties are to be regarded as real and isotropic. Also note that ‘permit.’ refers to relative permittivity, ‘permea.’ refers to relative permeability, ‘sigE’ refers to electric conductivity and that ‘sigM’ refers to magnetic conductivity.

Material Properties

material	permit.	permea.	sigE	sigM
mat#1	1.0	1.0	0.0000	0.0000
mat#2	1.0	1.0	2.6525e-03	3.7700e+02

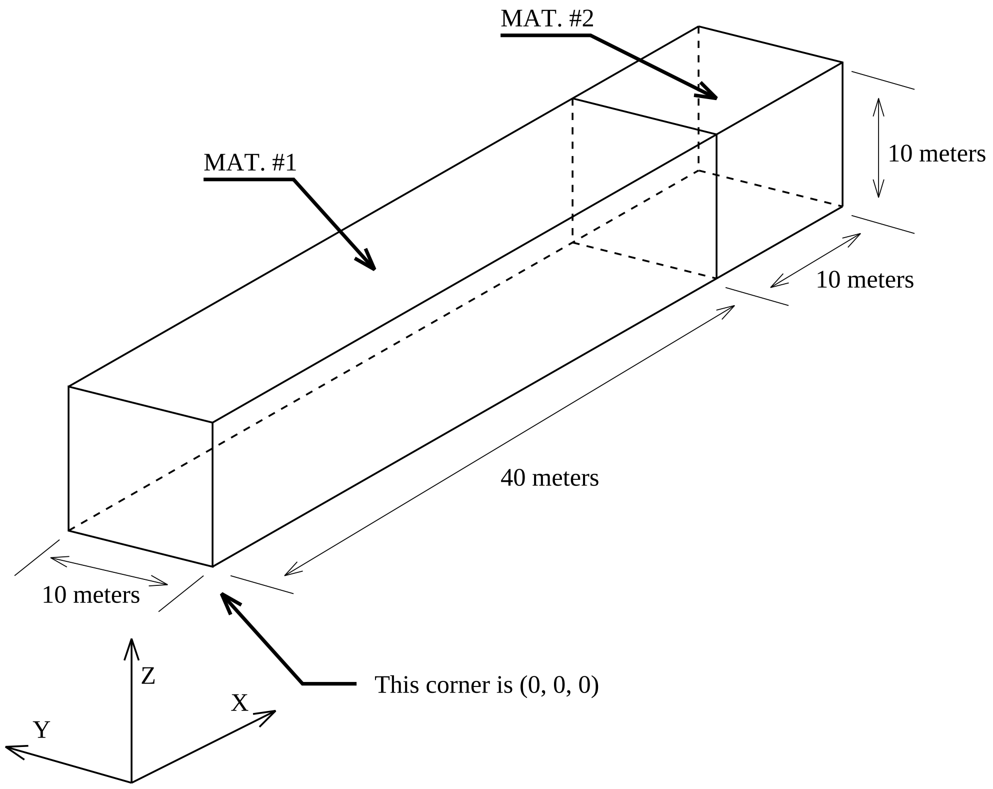


Figure F.2: Diagram of Test Chamber

The long rectangular block has been assigned a material equivalent to free space. Likewise the cube shaped block has been assigned a material with absorbing properties.

F.9.2 Mesh and Boundaries

An arbitrary mesh configuration is given: 200 by 10 by 10 boxes along the x, y and z coordinate axes respectively. Since the problem we are considering is basically one dimensional, that is wavefronts are expected to propagate along the x axis, the mesh is made most dense along the x axis.

To assign boundary conditions consider the following; Excitation is placed on the surface furthest from the absorbing material, corresponding to $x = 0$ and should be specified as a PEC surface otherwise. Note that when you specify a surface as being associated with excitation, 'bccode' will also ask for an *alias type.* which is PEC in this case. Surfaces associated with $y = 0$ and $y = 10$meters should specified as being PMC. All remaining surfaces should be left as PEC. Note that we will be apply energy to z components of the electric field, so we will be exciting a TEM mode.

F.9.3 Analysis Parameters

Tables were provided to help the user with this introductory example. Note that the parameters selected in most cases are arbitrary, remember that the point of this exercise is to become familiar with the software. Note that 800 time steps was arbitrarily selected since this should allow the user to “see” the most interesting parts of the analysis. A step size of 0.4 nano-seconds provides an ample margin to guarantee stability. In free space such an excitation should have a wavelength of 10 meters.

Run-Time Parameters

name	value
Time Steps	800
Step Size	4.0×10^{-10}sec

The step size can easily be entered from the keyboard as 4.0e-10. Note also that there is no need to specify a material center frequency.

Excitation Parameters

name	value
Type	Real Sinusoid
Component	Ez
Amplitude	1.0
Frequency	30×10^{6}Hz
Life Time	133.3×10^{-9}sec

For reporting results, a user most likely will want to start by using Patran files. Once convinced that the analysis works, you may want to select one or two sample points and try using sample files. In total the analysis should produce 21 Patran files. Patran reports are organized in x-y layers. Since in this example all layers should all look identical, there is no need to report all values from the mesh. For the arrangement presented, when you run ‘make_neutral’, request a mesh that is 200 by 10 by 1 along the x, y , and z components respectively. Note that even though such a mesh has a thickness of one box, it has two layers, that is a top and a bottom. Of course, the Patran file name given is arbitrary.

Report Parameters

name	value
Report Type	Patran Files
First Report	0 step
Steps Between	40 steps
First Layer	layer 5
Number Layers	2 layers
File Name	first

F.10 Summary

It is hoped that this manual gave the user an adequate introduction to the parallel FDTD solver package. This manual provides hints and directions on how to use the software and provides a simple introductory example.

Appendix G

References

This part of this document contains a list of all references made.

1. A. Adams, *The Negative*, The New Ansel Adams Photography Series/Book 2, New York Graphics Society, Published by Little, Brown & Company, Boston, 1981.

2. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Coputer Algorithms,* Addison-Wesley Publishing Company, copyright 1974.

3. H. Anton, *Calculus with Analytic Geometry*, John Wiley and Sons, Copyright 1980.

4. H. Anton and C. Rorres, *Elementary Linear Algebra, Applications Version*, John Wiley and Sons, Copyright 1991.

5. S. Baase, *Computer Algorithms, Introduction to Design and Analysis*, Second Edition, Addison-Wesley Publishing Company, copyright 1988.

6. A. Bayliss, C. I. Goldstein, and E. Turkel, "On accuracy conditions for the numerical computation of waves," Journal of Computational Physics, vol 59, pp. 396-404, 1985.

7. J. Bentley, "An Introduction to Algorithm Design," *Computer*, published by the IEEE, pp. 67-78, February 1979.

8. J. Bentley, "Programming Pearls, Algorithm Design Techniques," *Communications of the ACM*, pp. 865-871, vol. 27, no. 9, September 1984.

9. H. Behnke, F. Bachmann, K. Fladt and H. Kunle, Eds., *Fundamentals of Mathematics, volume II*, MIT Press, Cambridge Press, 1974.

10. J. L. Bentley and W. Carruthers, “Algorithms for Testing the Inclusion of Points in Polygons,” Proceedings, Eighteenth Annual Allerton Conference on Communication, Control, and Computing, Oct. 8 - 10, 1980, Sponsored by the University of Illinois - Urbana - Champaign, 1980.

11. D. W. Blackett, *Elementary Topology, A Combinatorial and Algebraic Approach,* Academic Press, New York, 1967.

12. R. Bonetti and P. Tissi, “Analysis of Planar Disk Networks,” IEEE Transactions on Microwave Theory and Techniques, vol. 26, no. 7, July 1978, pp. 471-477.

13. B. Engquist and A. Majda, “Absorbing Boundary Conditions for the Numerical Simulation of Waves,” Mathematics of Computation, vol. 31, no. 139, July 1977, pp. 629-651.

14. G. Fox, M. Johnson, Lyzenga, S. Otto, J. Salmon, D. Walker, *Solving Problems on Concurrent Processors,* Volume 1, Prentice Hall, Copyright 1988.

15. P. M. Flanders, “The Effective Use of SIMD Processor Arrays,” International Specialist Seminar, The Design and Application of Parallel Digital Processors, The Institution of Electrical Engineers, Copyright 1988, pp. 143 - 147.

16. M. J. Flynn, “Very High-Speed Computing Systems,” proceedings of the IEEE, vol. 54, no. 12, pp. 948-960, December 1966.

17. M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” IEEE Transactions on Computers, vol. 21, no. 9, September 1972.

18. K. P. Jacobsen, “Data Mapping Strategies for Effieient Implementation of a CFD Program on a Massively Parallel Computer,” presented at the 4th International Symposium on Computer Fluid Dynamics, Sept. 9 - 12, 1991, Davis CA.

19. P. J. Giblin, Chapman & Hall, *Graphs, Surfaces and Homology,* Mathematical Series, Halsted Press, New York, 1977.

20. A. J. Hansen and P. L. Levin, “On Conforming Delaunay Mesh Generation,” Advances in Engineering Software, vol. 14, pp. 129-135, 1992.

21. R. W. Hockney and C. R. Jesshope, *Parallel Computers 2: Architecture, Programming and Algorithms*, IOP Publishing Limited, Copyright 1988.

22. Christoph M. Hoffman, *Geometric & Solid Modeling*, Morgan Kaufmann Publishers, Inc., San Mateo, CA 1989.

23. R. M. Hord, *The Illiac IV, The First Supercomputer*, Computer Science Press, Copyright 1982.

24. Y. E. Kalay, "Determining the Spatial Containment of a Point in General Polyhedra," Computer Graphics and Image Processing, vol. 19, pp. 303–334, 1982.

25. D. Knuth, *The Art of Computer Programming*, Volume 1, Second Edition, Addison-Wesley Publishing Company, copyright 1973.

26. L. Kronsjö, *Computational Complexity of Sequential and Parallel Algorithms,* John Wiley & Sons, copyright 1985

27. D. T. Lee and F. P. Preparata, "Location of a Point in a Planar Subdivision and it's Applications," SIAM J. Comput., vol. 6, No. 3, September 1977.

28. T. Lewis and H. El-Rewini, *Introduction to Parallel Computing,* Prentice-Hall, Copyright 1992.

29. K. Li, K. Tassoudji, R. Shin, and J. Kong, "Simulation of Electromagnetic Radiation and Scattering Using a Finite Difference-Time Domain Technique," Computer Applications in Engineering Education, published by John Wiley & Sons, vol. 1, september/october 1992, pp. 45 - 63.

30. R. Luebbers, "Lossy Dielectrics in FDTD," IEEE Transactions on Antennas and Propagation, vol. 41, no. 11, Nov. 1993, pp. 1586-1588.

31. Martti Mäntylä, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, MD, 1988.

32. *MasPar Commands Reference Manual, Software Version 3.2,* Part Number 9300-0300, Rev. A9, May 1993.

33. *MasPar Data Display Library (MPDDL) Reference Manual*, Software Version 3.0, Part Number 9302-0200, Rev. A6, July 1992.

34. *MasPar Mathematics Library (MPML) Referencs Manual, Software Version 2.0,* Part Number 9302-0400, Rev. A5, July 1995.

35. *MasPar Parallel Application Language (MPL) Reference Manual, Software Version 3.2*, Part Number 9302-001, Rev. A4, May 1993, MasPar Computer Corporation.

36. Copyright 1993, MasPar Computer Corporation, *MasPar Parallel Application Language (MPL) Reference Manual*[35], Used by permission.

37. *MasPar Parallel Application Language (MPL) Users Guide, Software Version 3.2*, Part Number 9302-0101, Rev. A5, July 1993.

38. *MasPar System Overview*, Part Number 9300-0100, Rev. A5, July 1992, MasPar Computer Corporation.

39. *MasPar System Overview*[38], Copyright 1992, Illustrations used by permission.

40. Raj Mittra and Jin-Fa Lee, "Direct Maxwell's Equation Solvers in Time and Frequency Domains – A Review," *Directions in Electromagnetic Wave Modeling*, Plenum Press, New York, 1991.

41. Gerrit Mur, "Absorbing Boundary Conditions for the Finite-Difference Approximation of the Time-Domain Electromagnetic-Field Equations," IEEE Transactions of Electromagnetic Compatibility, vol. 23, no. 4, Nov. 1981.

42. A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing,* Prentice-Hall, Inc, 1989.

43. J. D. Paliouras and D. S. Meadows, *Complex Variables for Scientists and Engineers, second edition*, Macmillan Publishing Company, 1990.

44. D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface,* Morgan Kaufman Publishers, Copyright 1993.

45. P. Petropoulos, "Phase Error Control For FD-TD Methods," *Ultra-Wideband Short-Pulse Electromagnetics,* Plenum Press, 1993.

46. R. Pickering and J. Cook, *Parallel Processing: A Self-Study Introduction*, A Report in a series of parallel processing tutorials from parallab, Dept. of Informatics, University of Bergen, N-5020 Bergen, Norway. Copies can be obtained by sending electronic mail to `adm@parallab.uib.no`.

47. F. P. Preparata and M. I. Shamos, Computational Geometry, An Introduction, Springer-Verlag, Copyright 1985.

48. J. F. Prins and J. A. Smith, "Parallel Sorting of Large Arrays on the MasPar MP-1," The 3rd Symposium on the Fontiers of Massively Parallel Computation, IEEE Computer Society Press, Copyright 1990, pp. 59 - 64.

49. C. Rappaport, "Preliminary FDTD Results from the Anechoic Absorber Boundary Condition," IEEE Antennas and Propagation International Symposium, 1992 Digest, vol. 1, pp. 544 - 545.

50. C. M. Rappaport and L. J. Bahrmasel, "An Absorbing Boundary Condition Based on Anechoic Absorber For EM Scattering Computation," Journal of Electromagnetic Waves and Applications, vol. 6, no. 12, 1992, pp. 1621-1633.

51. A. Reineix and B. Jecko, "Analysis of Microstrip Patch Antennas Using Finite Difference Time Domain Methods," IEEE Transactions on Antennas and Propagation, vol. 37, no. 11, Nov. 1989, pp. 1361-1369.

52. Zachary S. Sacks, *A Finite Element Time Domain Method for Microwave Cavities*, Major Qualifying Project Report, Worcester Polytechnic Institute, Project Number EE-JFL-9302, Library Resorces Number: 94D139M, May 2, 1994.

53. Z. Sacks, D. Kingsland, J. Lee and R. Lee, "A Perfectly Matched Anisotropic Absorber for use as an Absorbing Boundary Condition," Currently Unpublished.

54. M. Sadiku, *Elements of Electromagnetics,* Saunders College Publishing, a division of Holt, Rinehart and Winston, Inc., copyright 1989.

55. A. J. Schwab, *Field Theory Concepts,* Springer-Verlag, New York, 1988.

56. D. M. Sheen, S. M. Ali, M. D. Abouzahra, and J. A. Kong, "Application of the Three-Dimensional Finite-Difference Time-Domain Method to the Analysis of Planar Microstrip Circuits," IEEE Transactions on Microwave Theory and Techniques, vol. 38, no. 7, July 1990, pp. 849-856.

57. L. Shen and J. Kong, *Applied Electromagnetism*, PWS Publishers, Boston, Massachusetts, 1983.

58. A. Silberschatz, J. Peterson, P. Galvin, *Operating Systems Concepts, Third Edition*, Addison - Wesley Publishing Company, Copyright 1991.

59. D. K. Stevenson, *Programming the Illiac IV*, Carnegie-Mellon University, Department of Computer Science, November 1975.

60. H. S. Stone, *High-Performance Computer Architecture*, Addison - Wesley Publishing Company, Copyright 1987.

61. G. Strang, *Linear Algebra and its Applications, second edition*, Academic Press, New York, 1980.

62. A. Taflove and M. E. Brodwin, "Numerical Solution of Steady-State Electromagnetic Scattering Problems Using the Time-Dependent Maxwell's Equations," IEEE Transactions on Microwave Theory and Techniques, vol. 23, no. 8, August 1975, pp. 623 - 630.

63. Taflove and Umashankar, "Review of FD-TD Numerical Modeling of Electromagnetic Wave Scattering and Radar Cross Section," Proceedings of the IEEE, vol. 77, no. 5, May 1989, pp. 682 - 699.

64. Taflove and Umashankar, "The Finite-Difference Time-Domain Method for Numerical Modeling of Electromagnetic Wave Interactions with Arbitrary Structures," *PIER2 Progress In Electromagnetics Research, Finite Element and Finite Difference Methods in Electromagnetics Scattering*, Elsevier, copyright 1990.

65. J. C. Tilton, "Porting an Iterative Parallel Region Growing Algorithm from the MPP to the MasPar MP-1," The 3rd Symposium on the Fontiers of Massively Parallel Computation, IEEE Computer Society Press, Copyright 1990, pp. 170 - 173.

66. A. Trew and G. Wilson, *Past, Present, Parallel: A Survey of Available Parallel Computer Systems,* Springer-Verlag, Coyright 1991.

67. Kane S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," IEEE Transactions on Antennas and Propagation, vol. 14, no. 3, May 1966. pp. 302 - 307.

68. X. Zhang and K. K. Mei, "Time-Domain Finite Difference Approach to the Calculation of the Frequency-Dependent Characteristics of Microstrip Discontinuities," IEEE Transactions on Microwave Theory and Technique, vol. 36, no. 12, December 1988, pp. 1775-1787.