

To the Graduate Council:

I am submitting herewith a thesis written by Rushian Evan Reynolds entitled "Parallel Fluid Dynamics Grid-Mapping Optimization and Simulation." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

DP Mehta

Dr. Dinesh Mehta, Major Advisor

We have read this thesis
and recommend its acceptance:

Ben Whithy

Alfonso Pujol

Accepted for the Council:

Lawminkel

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature P. E. Pugh

Date 12/2/94

Parallel Computational Fluid Dynamics
Grid Mapping Optimization and Simulation

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Rushian Evan Reynolds

December, 1994

Dedication

I would like to dedicate this thesis to my parents, R. Neil and Elaine Reynolds, for their unwavering support and faith in me, as well as their love.

Acknowledgments

I would like to thank my thesis advisor, Dr. Mehta, for his never-ending support of this paper, as well as Dr. Whitehead for answering my questions at odd hours. I also greatly appreciate Dr. Pujol's advice on writing, as well as his friendship.

I owe a great debt to Calspan Corporation for their Research Assistantship. More specifically, I would like to thank Scott Evans and Bill Dietz for guiding me through Computational Fluid Dynamics and always being there to assist me when I needed it, and to Dr. Benek for hiring me for the assistantship in the first place. Finally, I give my thanks to Chris Humphres for allowing me to use the results of his research in this thesis.

Abstract

Computational Fluid Dynamics (CFD) are an important software algorithmic tool used in the design and analysis of objects that move in a fluid environment. However, due to the large computational times required for CFD problems, parallel CFD grid mapping optimization and simulation are useful tools. These algorithms have been developed to minimize the computational times for these CFD problems and also to provide an approximation of the times required for those problems.

The optimization algorithm, which uses Simulated Annealing to find an mapping of grids to processors which is close to optimum, results in a decrease in execution time ranging from 4-97%. Furthermore, although the problem is NP-Hard, our polynomial-time heuristic found an optimal mapping for the cases tried. The simulation algorithm shows good accuracy at approximating the run times of CFD problems, with under a 7% error.

The results of this study warrant the application of these techniques to other parallel CFD flow solvers.

Table of Contents

Chapter	Page
I. Introduction	1
II. Background and Theory	4
2.1. Computational Fluid Dynamics	4
2.2. Parallel and Distributed Computation	8
2.3. CFD in Parallel	10
2.4. Parallel CFD Simulation	11
2.5. Optimization of Grid Mappings	13
III. CFD Parallel Simulator Design	15
3.1. SYSMOD	15
3.2. PEGMOD	16
3.3. OPTMOD	18
3.4. OBJECT file	18
3.5. SIMMOD	20
IV. CFD Grid Placement Optimization	24
4.1. Problem Definition	24
4.2. Proof of NP-Hardness	26
4.3. Heuristic Solutions	28
4.4. Simulated Annealing	29
4.5. OPTMOD	31

4.6. Performance of the Simulated Annealing Algorithm	34
V. Results	37
5.1. Simulation Results (425 Timesteps).....	37
5.2 Optimization Results (Varied Timesteps)	40
VI. Conclusions	47
List Of References	48
Vita	50

List of Figures

Figure	Page
2.1 Wind Tunnel Model of Guided Bomb Unit modeled by CFD Grids	5
2.2 High Level CFD Algorithm	6
2.3 Two Overlapping Grids	7
2.4 High Level CFD Algorithm with Parallelism	12
3.1 Example SYSMOD Input File	17
3.2 Example SYSMOD Output File	17
3.3 Example PEGMOD Output File	18
3.4 Object Hierarchy	19
3.5 Connected Grids	22
4.1 Algorithm for Simulated Annealing	31
5.1 Simulated vs. Real Execution Times	38
5.2 Simulated vs. Real Execution Times	38
5.3 Bar Chart for Simulated vs. Real Timings	39
5.4 Grid Sizes and Numbering	41

List of Symbols

i	Instance of a processor
P_i	Computational Load of Processor i
S_i	Speed of Processor i
j	Instance of a grid
x_j	Number of Grid Points in Grid j
α_{ij}	Index into a matrix; equals 1 if grid j is mapped into grid i , 0 if not
m	Number of Processors
n	Number of Grids
t	Some task
T	Number of tasks
$l(t)$	Length of time to compute a task
σ	Function to map a task t to a positive integer
k	Boltzmann's constant

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is an important development tool in the design and analysis of any object that moves in a fluid environment, such as air and space vehicles. However, CFD is extremely computationally intensive - a Cray X-MP took over 100 hours to run a 3-D viscous unsteady model for the air flow through just *part* of an axial flow gas turbine engine. [5] The desire to model larger problems with more accurate solutions requires greater processor speed. [9] [5] Parallel computing offers an alternative to achieve the decrease in execution time that CFD requires. However fast a processor can perform, two of those processors can perform better, and ideally do a task in half the time. Adding yet another processor should result in another decrease in execution time.

Calspan Corporation, a CFD research corporation located at Arnold Engineering Development Center, has been using a distributed system of DEC AXP 3000/800 Alpha 200 MHz Workstations to compute CFD in parallel by sending each CFD grid to a processor. Grids are how CFD mathematically models an object - a series of grids are placed about an object and fluid flow is modeled through the grids. If there are more grids than processors, which will be the usual case, each processor will end up with several grids. Each processor will then compute its grids in parallel with the other processors. The only communication that must be sent between processors is the

boundary information that the grids pass back and forth. Due to the inter-relationships between the grids, however, different grid mappings to processors will result in different execution times.

To assist in this project, this thesis presents a program that will find a mapping of grids to processors to achieve a minimum execution time. The algorithm used, Simulated Annealing, models the way metals cool and anneal into a state of minimum energy. While Simulated Annealing is not guaranteed to always find a true optimum, it will almost always find a solution that “cannot be significantly improved upon.” [7 p446] The program is able to find this mapping for any given system of grids and any given distributed systems of any type of processors in polynomial time. Furthermore, the program contains a Discrete Event Simulator that approximates the execution time of that mapping, or any other given mapping of grids to processors. This is primarily useful for determining exactly what effect changes on the system will have on the execution time of CFD problems.

Chapter Two explains the basic ideas behind Computational Fluid Dynamics in more depth. It also presents some concepts of Parallel Computing, and how it applies to CFD. Chapter Three presents the Simulation program, which consists of four modules. Each module will be discussed in a section. Chapter Four presents the Optimization program. The optimization problem is mathematically defined, and it is shown that this problem is

NP-Hard, meaning that the true optimum cannot reliably be found in polynomial time.

The solution used, Simulated Annealing, is then explained. Other heuristics that approximate the solution are introduced. Chapter Five presents the results of the research, and discusses the accuracy of the Simulation program and how well the Simulated Annealing algorithm came to finding an optimum mapping. Finally, Chapter Six presents a brief summary and draws conclusions from this research.

Chapter II

Background and Theory

This chapter begins by presenting the basic ideas behind Computational Fluid Dynamics (CFD), and explaining why greater speed is required to compute CFD problems. The basic ideas behind parallel computing are then presented. The applicability of parallel computing to CFD is then explored, and the rationale for a Parallel CFD Simulator and Optimizer is presented.

2.1 Computational Fluid Dynamics

Computational Fluid Dynamics refers to the modeling of a body in a fluid flow. It is used to test aerodynamic designs for flaws before expensive models or prototypes are built. A set of grids is placed around the body, with fine meshes in places where high gradients are expected (i.e., places where the flowfield quantities will change rapidly over a small area), and coarser meshes where there will not be as many points needed to calculate the fluid flow. An example of these meshes modeling a body can be seen in Figure 2.1. The meshes close to the body being modeled are usually fine, with coarser grids as the meshes move away from the body. A set of equations is solved for each grid point to model the fluid flow and determine the rate of change of conserved quantities of the fluid at that point.

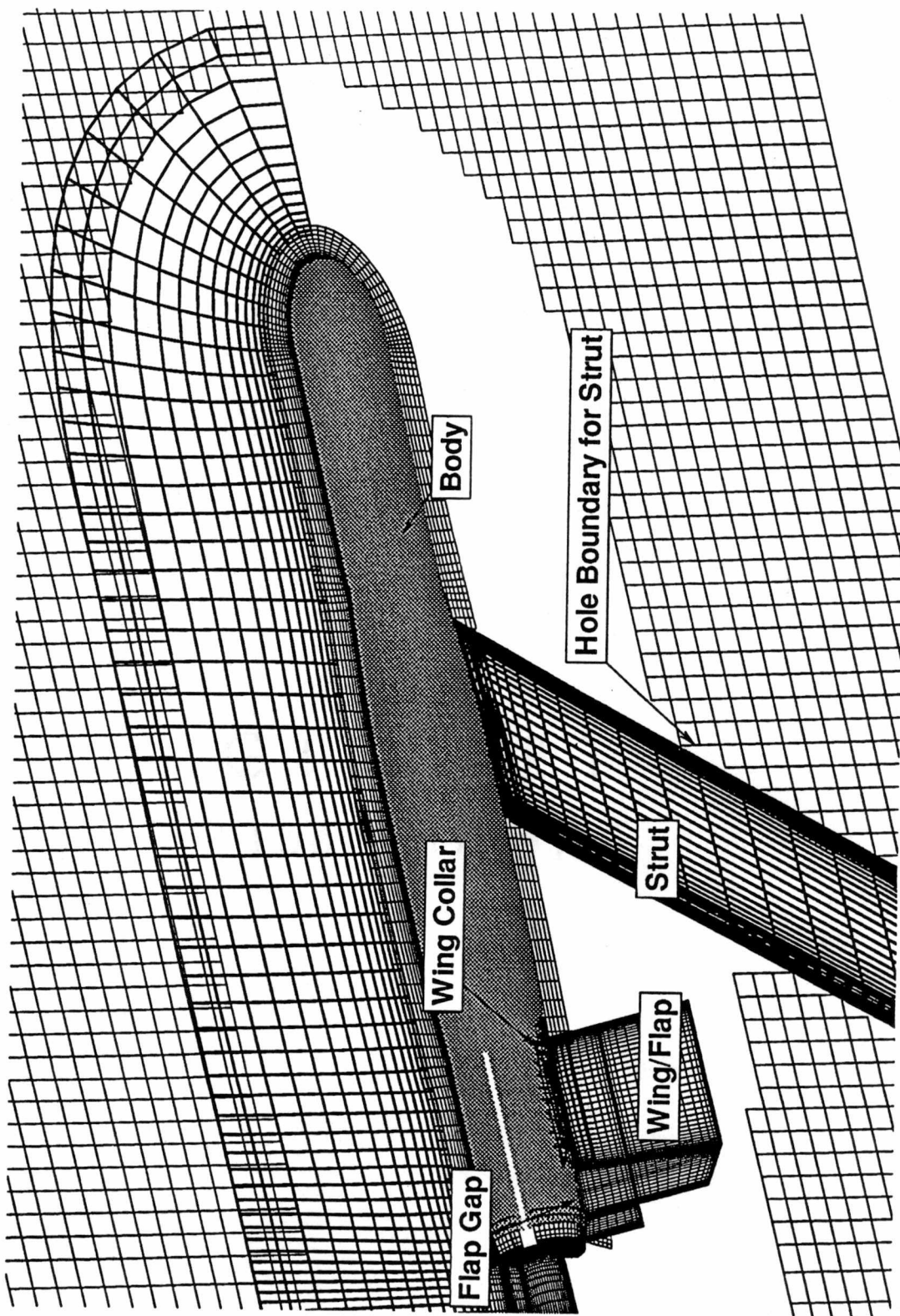
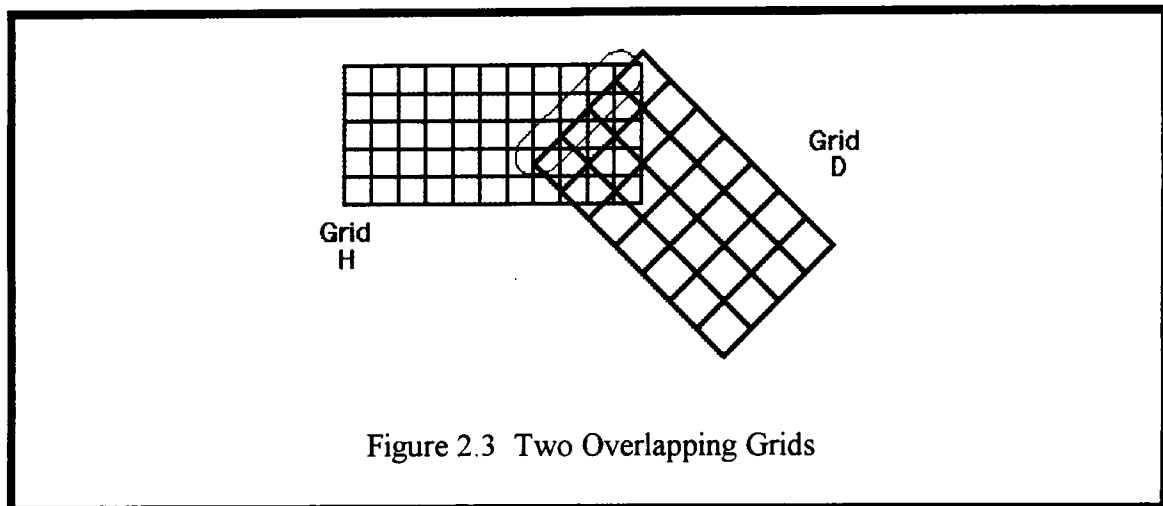


Figure 2.1: Wind Tunnel Model of Guided Bomb Unit modeled by CFD Grids
Source: Calspan Corporation, AEDC operations, CFD Department

The conserved quantities mentioned above refer to the conservation laws. The applicable conservation laws tell us, for example, that if an amount of fluid (such as air) enters an area, then it must also leave that area. If it does not, then the fluid density in that area must change. The same thing applies to energy and momentum. CFD uses these laws, in the form of the Navier-Stokes equations, to compute the fluid flow around a body. The Navier-Stokes equations are partial differential equations which are mathematical representations of the conservation laws. To compute them, each grid point keeps track of several things - the density, the energy, and the momentum in each direction (for a three dimensional mesh, there will be three direction momenta). Each grid computes these things for a number of iterations, called "timesteps". These grids have an overlap in each place where they share a boundary. This overlap is used to send information between bordering grids. When one grid has completed a timestep, the bordering grids will use that data to interpolate their outer edge where that edge borders the finished grid. The grid points lie along the intersections of the mesh. The high level algorithm that performs all this is shown in Figure 2.2.

- **FOR** timesteps 1 to N **DO**
 - **FOR** each grid G_i , $1 \leq i \leq m$
 - **FOR** each stencil point **DO**
DO interpolation
 - **FOR** each point **DO**
Perform CFD computations
 - **FOR** each boundary with other grids **DO**
Send proper boundary points

Figure 2.2 High level CFD Algorithm



lines. To interpolate one of these points, the receiving grid takes all points in the grid quadrilateral (hexahedron for a three dimensional mesh and so on) that the point to be interpolated lies within. The receiving grid then uses linear interpolation to find a value for that point.

Figure 2.3 gives an example of a pair of overlapping grids that must use interpolation to send information to each other. When the horizontal grid, Grid H, finishes a timestep, the diagonal grid, Grid D, will interpolate its top row of points. The row to be interpolated has been circled in the figure for clarity. Each point in Grid D lies inside a square of the mesh of Grid H. The point on Grid D will use the four points on the corners of that square to linearly interpolate information for itself. Once all boundary points have been updated in that fashion for a particular timestep, the grid can compute the rest of its points using the Navier-Stokes equations.

2.2 Parallel and Distributed Computation

While the equations to model a body are known, the computation time is extremely large. For example, it took about 100 hours to run a 3-D viscous unsteady model of just *part* of a gas turbine engine on a Cray X-MP. [5] Furthermore, single processors “appear to be within a single order of magnitude of their speed limits due to physical limitations such as the speed of light and gate switching limits.” [5 p290] Parallel processors avoid those problems, and also offer some advantages over vector supercomputers since not all algorithms can be fully vectorized. [9] Furthermore, if performed on a distributed system of workstations, the large cost of a mainframe is avoided, as is the problem of trying to schedule time for such a huge problem on a large mainframe. The potential for a decrease in execution time is undeniable - if one processor can do a task, two or more processors should be able to do it faster.

Parallel computers have several classifications, based on the type of processors being used. The biggest differentiation is whether or not different processors can execute different instructions. There are four major classifications. [8]

The simplest is the SISD machine. SISD stands for single instruction, single data. This is the normal, non-parallel machine. They have one CPU and can only handle one instruction stream.

SIMD machines (single instruction, multiple data) can only handle one instruction at a time, but it can run that instruction on different data simultaneously. It will often have a large number of simple CPUs - some current machines have from 1,024 to 65,536.

Examples of this type of machine are the MasPar and the AMT DAP. Vector processors, such as the Cray mentioned above, are often considered a subclass of SIMD machines. Vector processors contain special hardware that operates on arrays of similar data. They can give one, two, and in special cases three results per clock cycle by computing these arrays in lockstep. However, they can only do this when running in vector mode.

MIMD machines (multiple instruction, multiple data) have different processors that can run different instruction streams, each with their own data. This is divided into two classes depending on the way memory is used.

Shared memory MIMD machines have many CPUs which reference a common memory. The Cray Y-MP is an example. The main problem is of scalability. Each CPU would like a full connection to the memory area. As more CPUs are added, this becomes more and more difficult. Even with alternative connection schemes, this is not scaleable to a large number of CPUs. Usually machines of this type only have 2 to 20 processors.

Distributed memory MIMD machines (DM-MIMD) consist of a number of processors, each with their own memory. The processors are hooked together by a network. All communication between processors is sent over the network. This avoids the scalability problems of shared memory systems, since as the number of processors grows so does the number of communication links. Thus the bandwidth automatically scales up with the processors.

A DM-MIMD system was chosen to be modeled by the Simulation program. The advantage of this over other systems is that each processor can work on one or more grids, so that several grids are being computed at any one time. The only limit on the number of grids being computed simultaneously is the number of processors, which can easily be increased at any time. Also, since any network of workstations can be used for the DM-MIMD system it is a very attractive idea - many institutions already have such a network in place.

2.3 CFD in Parallel

Due to the extreme computation times required for CFD problems, faster computation time is a high priority. The example given earlier, of the turbine engine on a Cray X-MP, shows the need for greater speed. Parallel computing is a logical way to achieve some of that extra speed. In fact, as can be seen in the results chapter, Parallel computing

actually gave us an almost linear decrease in execution time, with four processors solving the problem in almost a fourth of the time.

The method of parallelizing the CFD problem used by this thesis is to map grids to processors. Each processor is sent some number of grids which it is then responsible for computing. The only information sent between processors is sets of stencil points. As a grid is updated for a timestep (one moment in time), it sends updated interpolated values of conserved variables that are required for boundary point updating to all grids that border it. Those grids cannot advance to the next timestep until all their boundaries have been updated to their current timestep. Thus, even if all processors have an equal load, some will spend more idle time than others while they wait for boundary points. The idle time spent waiting will change as the mapping of grids to processors change, and grids get updated in a different order or time and more importantly the processor workloads change. In order to optimize the mapping, both of these must be accounted for.

However, since most grids will vary greatly in size (10,000 and 400,000 are both reasonable sizes), equalizing the load is the most important task. The high level parallel algorithm is shown in Figure 2.4.

2.4 Parallel CFD Simulation

Due to the long computation times of CFD problems, a simulation of that computation becomes a useful tool. Once supplied with the proper data, it allows the user to know

```

• FOR timesteps 1 to N DO
  {Perform following FOR loop in parallel}
    • FOR each grid  $G_i$ ,  $1 \leq i \leq m$ 
      • FOR each stencil point DO
        interpolation
      • FOR each point DO
        Perform CFD computations
      • FOR each boundary with other grids DO
        Send proper boundary points
    {End parallel FOR loop}

```

Figure 2.4 High level CFD Algorithm with Parallelism

how long a particular problem on a particular system will take to compute. This gives such a Discrete Event Simulation two functions.

(1) Since the Discrete Event Simulator will allow the user to approximate the time a problem will take to compute, it will allow better planning of system resources. This will also allow a user to better determine the effects of changing grid mappings and varying workloads on a multi-user system on the solution time of a CFD problem. For example, in the four grid test case (presented in Chapter 5), there was almost no decrease in execution time between two and three processors. This was due to having one processor with two grids, and two with one grid - the last two spent much of their time waiting on the first. Using the Discrete Event Simulator to detect and prevent this by only using two machines would greatly increase the efficiency of the machines if not the execution time - the machines spend much less idle time. Furthermore, the machines with the most

idle time can be determined, and those tasks assigned to the busiest processor. Thus, other jobs performed by that processor will be partially absorbed by the idle time of the CFD task and not effect overall performance as much.

(2) The user can determine the effects of a change in the network of processors. A user could assess the benefits of upgrading the system BAUD rate, adding a processor, etc., before investing in new hardware.

2.5 Optimization of Grid Mappings

Changing the mapping of grids to processors can drastically change the time needed to compute the problem. There are two unrelated causes for this.

(1) If the processor load is unequal, some processors will spend much of their time idle while other processors work continuously. Equalizing the processor load keeps all the processors working as much as possible, with a corresponding reduction in the overall computation time. In an eight grid - two processor test case, about half an hour could be saved just by the mapping of grids to processors, even given the assumption that each processor will be mapped four grids.

(2) The number of boundaries between two grids that must be sent between processors will change as the mapping changes. If two grids that share a boundary are on the same processor, some time will be saved since they do not have to transmit the updated boundary points over the network. How much time depends on the BAUD rate and the number of points to be sent.

The first cause is by far the most important. As current transmission times have become extraordinarily fast, especially with the introduction of fiberoptics, the time for grid information to be sent over a network has become minimal - often under a second.

Furthermore, since all grids are being computed for a number of timesteps, each of these slowdowns becomes magnified since they repeat for each timestep. The number of timesteps will vary, depending on the object being modeled. Grids must be computed until the fluid flow around the modeled object has stabilized. This can take from 10,000 to 100,000 or more iterations, depending on the efficiency of the flow solver and the complexity of the modeled problem.

Chapter III

CFD Parallel Simulator Design

The program was written using C++. The design is very modular, allowing each module to be changed or replaced without affecting other modules. Each module can be compiled into a large program with other modules, or can be a stand-alone program. This allows modifications to be made with a minimum of trouble. The only information passed between modules is in the form of two files created by SYSMOD and PEGMOD, and modified by OPTMOD. Details can be found in the module descriptions.

When all modules are run together, they are controlled by an extra module, called SIM. SIM merely calls each module in the order enumerated below. It also passes to the modules that need them the user input files.

3.1 SYSMOD

The Systems Module (SYSMOD) is responsible for taking a user-specified input file that describes a computer network or networks and turning it into a more computer-friendly form. The file first specifies the number of bytes to be transmitted per grid point when sending stencil points over the network, and then the number of timesteps the grids should be computed. It then specifies, for each network, how many grid points a processor can compute in one second and the RAM size of each processor.

Comments are allowed, using the same format as C. An example input file is given in Figure 3.1.

SYSMOD creates a file called \$\$system.dat and places this information in a series of integers separated by commas. This file is set up so that the program that will read it will find out in advance how many processors and networks there are, before getting the information about them. This allows the reading program to dynamically allocate arrays for those processors and networks without having to read through the file twice. Review of Figure 3.2 and the source file for it, Figure 3.1, will demonstrate this.

3.2 PEGMOD

PEGMOD, the PEGSUS Module, takes as input a user-specified input file. The file must be a PEGSUS Interpolation file. PEGSUS is a grid generation program created by Calspan Corporation for their CFD flow solver. For another type of input file a replacement module can be easily written, as long as the output file it creates follows the same format. The output file, \$\$pegout.dat, specifies the number of grids, information about the grids (such as the size) and finally specifies data about boundary and stencil points in a two dimensional matrix. In the matrix in Figure 3.3, grid 1 sends 144 points to grid 0, and grid 0 sends 114 points to grid 1.

```

/* Comments use the same format as C, starting with a /* and ending with a */

/* Number of bytes to be transmitted per interpolation point: */
    Bytes = 8
/* How many timesteps should the grids run? */
    Steps = 450
/* The two numbers are processor speed, RAM size */
    Network 1 connected to 2; 100 BAUD.
        500,3
        500,5
        500,7
        500,9
/* We'll say network 2 is the central network */
    Network 2 connected to 1,3; 200 BAUD.
        500,11
        500,13
        500,15
    Network 3 connected to 2; 300 BAUD.
        500,25
        500,27
        500,29
        500,31
        500,33
END
    end of file
    Everything past the END will be ignored.

```

FIGURE 3.1 Example SYSMOD input file

8,450,4,4,3,5,4,500,3,500,5,500,7,500,9,500,11,500,13,500,15,500,25,500,27,500,29,500,31,500,33,500,17,500,19,500,21,500,23,1,100,0,3,2,200,1,3,300,1,2,400;END.

1st number = number of bytes to transmit per grid point
 2nd number = number of timesteps to compute grids
 3rd number = number of networks
 4th - 7th number = number of processors in each network
 Remaining numbers are processor information, and when all processors in a network have been specified, the network BAUD rate is given.

FIGURE 3.2 Example SYSMOD output file, \$\$system.dat, for figure 3.1, with explanation of numbers.

4	-----Number of grids
144 114 114 1 40 5 50	-----Grid data.
228 228 342 115 54 5 40	
168 189 531 343 60 5 30	
75 84 615 532 50 5 27	
0 144 0 0	-----Stencil and Boundary point
114 0 114 0	matrix
0 84 0 84	Rows show boundary points
0 0 75 0	Columns show stencil points

FIGURE 3.3 Example PEGMOD output file, \$\$pegout.dat.

3.3 OPTMOD

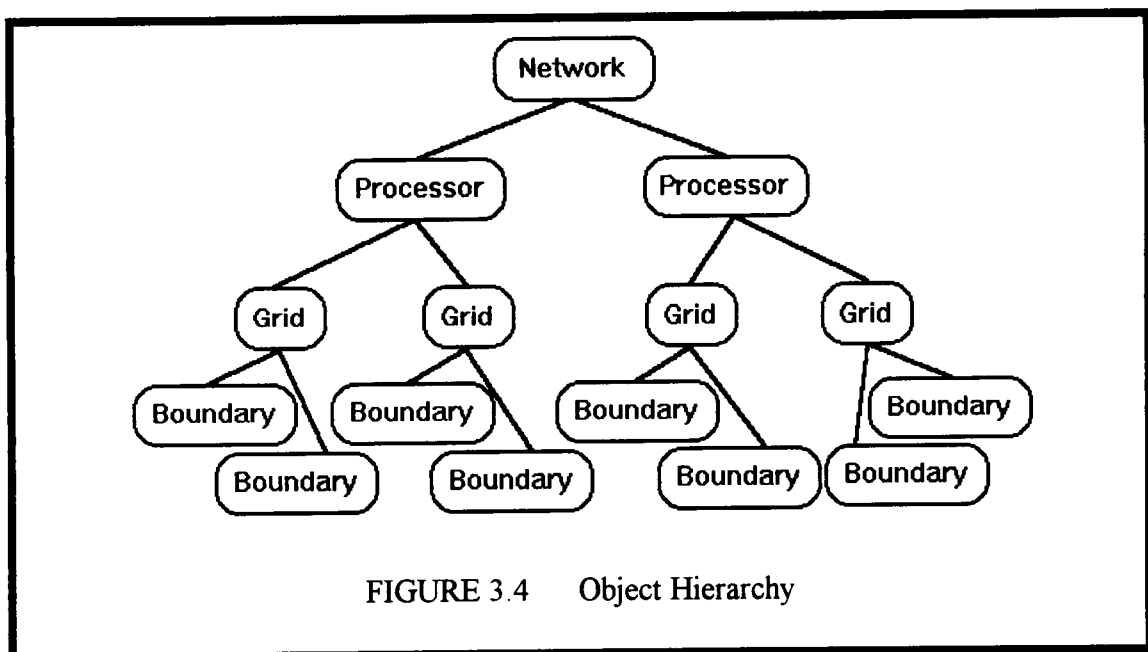
OPTMOD is the optimization module. If it is to be run, it will usually be run at this point. It will find the mapping of grids to processors that will take the least amount of time for the processors to compute the problem. It will be discussed in detail in Chapter Four.

3.4 OBJECT file

This file contains the object declarations. The objects are arranged in a strict hierarchy, as shown in Figure 3.4. The controlling program has access to a dynamically declared array of pointers to Network objects. To reference any other object, the call must be sent through the Network object. Each Network contains some number of Processor

objects. The Processors have Grids, which have Boundary objects. Each object can only reference objects one level down in its subtree, as drawn in Figure 3.4. To reference objects further down the subtree, calls must be sent down one layer at a time. For example, a Processor could query a Grid, which queries a Boundary, which sends information back to the Grid, which gives it back to the processor. This keeps modularity present in the design, as well as making it easier to use the objects - Processors never have to worry about anything besides Grids, etc.

Each object has a set of variables which keep track of that instance of an object. No information about the objects is kept outside the objects, but must be gotten from the objects. Furthermore, all object manipulation is performed within the objects. This simplifies programming, as any operation can be executed for all instances of an object by merely placing it in a loop consisting of one function call.



The purpose of each object is kept simple. The Network objects keep track of data such as traffic and network BAUD rates. Processor objects keep track of their RAM size and processor speed. Grids know their ID number and size, while their Boundary objects keep track of the grids stencil points, destination of those points, and boundary points, as well as how many points there are. The object functions perform these tasks, and are called and controlled by the Discrete Event Simulator, SIMMOD.

3.5 SIMMOD

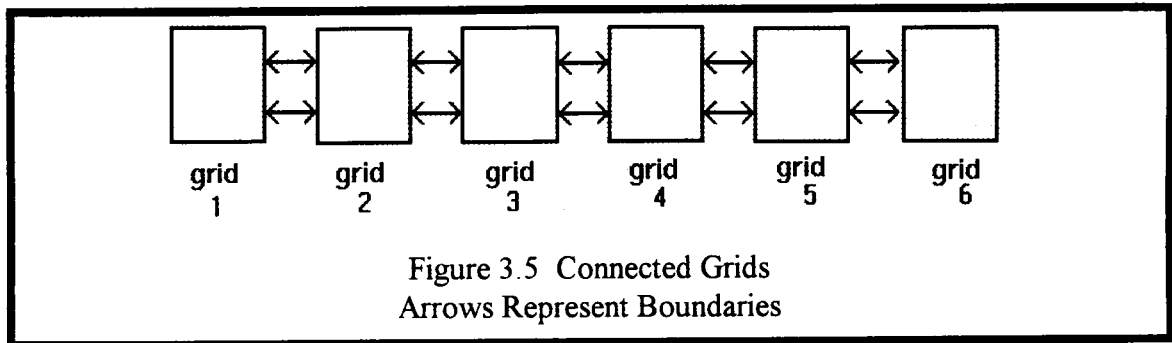
SIMMOD, the Simulation Module, is a Discrete Event Simulation. It first creates and initializes the Objects, and then runs the simulation using an event list. There are four events that control the simulation. The first is a one time only event, which is the start of the simulation. When the discrete event simulation begins, each processor that has a grid begins to compute a grid. Grids are selected by first looking for the grid that has been updated the least, or is on the lowest timestep. For each of those, the processor will take the grid with the most stencil objects, since that means that the most other grids are waiting for that grid to be updated. If there is a tie, it will take the smallest grid. Grid selections for other events are computed that way as well.

The remaining events occur during the simulation run. The first, Event 'A', tells the Discrete Event Simulator that a processor is finished computing a grid. The processor

object then computes at what time each stencil packet will arrive, and puts that on the event list as Event 'B'. The processor object also computes what time it will be finished sending stencil points and puts that on the event list as well, as Event 'C'. When stencil packets arrive, Event 'B', the receiving grid updates the proper boundary. When Event 'C', a processor coming free, occurs, the Discrete Event Simulator merely starts that processor on another grid. If there are no grids ready to be updated, the processor flags itself idle and waits for a grid to come ready. The only way for this to occur is for the boundary points of a grid to be updated. Thus all testing for this is performed when boundary points are received.

It is worth proving that the grids that share a boundary will be within one timestep of each other. If they were not, the available buffer space could easily be overloaded as the grid that got ahead of its neighbor sent boundary information to the slower grid. While that could be dealt with, it will be shown that it could not happen due to the boundary points. A grid cannot advance to timestep x until all of its boundaries have been updated to timestep $x-1$.

Theorem. If two grids share a boundary, they will always be within one timestep of each other. If the shortest path between two grids is of length l , where a path is found by moving along grids connected by a boundary, the grids will be at most $(l-1)$ timesteps apart.



Proof. Pick any group of connected grids. Draw from them the longest path possible, such as that shown in Figure 3.5. Other connections between grids in Figure 3.5 will actually mean that the grids must have closer timesteps, so this proof can be extended to any group of grids. Let grid one be on timestep x . It would have then updated the boundary to grid 2 for timestep $x-1$. Therefore grid two cannot be on a higher timestep than x , since it does not have the updated boundaries. Furthermore, it cannot be on a lower timestep than x , since grid one is on timestep x , meaning that its boundaries have been updated to timestep $x-1$. Thus grid two has to either be waiting for a boundary from grid three to start on timestep x , be working on timestep x , or have finished timestep x and is waiting to boundary points to come in so it can start on timestep $x+1$. If it is working on timestep x , then grid three has the same possibilities as grid two had. However, if grid two is finished working on timestep x , grid three could actually be working on timestep $x+1$. Also, if grid two is waiting for boundary points to start on

timestep x , grid three could be on timestep $x-1$. The worst case for grid four would then be if it was on timestep $x-2$, grid five on timestep $x-3$, and grid six on timestep $x-4$. This can be extended to show that if the shortest path between two grids has length l , where the path between grid four and grid one in Figure 3.5 is of length 3, the grids could be $(l-1)$ timesteps apart. That is a pathological example, but even in that case no boundaries are more than one apart.

When a Processor has computed all of its Grids for all timesteps, it flags itself idle and will not be put on the event list again. Thus the discrete event simulation ends when the event list is empty. At that point, the Discrete Event Simulator reports the amount of idle time it computed each processor to have, as well as how long the CFD problem will take to run.

Chapter IV

CFD Grid Placement Optimization

The actual function to minimize will be defined and shown to be NP-Hard. Heuristics to solve it will be reviewed, and the reason they were not used explained. A near-optimal solution to the function will then be found in polynomial time using Simulated Annealing.

4.1 Problem Definition

The final module is called OPTMOD, for Optimization Module. It is responsible for optimizing the mapping of n grids to m processors. The computational load of each processor for one timestep, without accounting for the sending and receiving of grid points or the inter-relationships of grids, is represented by the equation

$$P_i = \alpha_{i1}x_1 + \alpha_{i2}x_2 + \dots + \alpha_{in}x_n \quad (4.1)$$

where i is an instance of a processor, P_i is the load of processor i , x_j is the number of grid points in grid j , and $\alpha_{ij} = 1$ if grid j is mapped to processor i , and is zero if not.

Therefore, P_i equals the number of grid points mapped to that processor. (It should be noted here that the amount of work a processor must perform is directly proportional to the number of grid points it must compute.) The Optimizer begins by mapping grids to processors in a way that equalizes the processor load as much as possible. However, to truly equalize the load, processor speed must be accounted for. This is accomplished by extending equation 4.1 to

$$P_i = \frac{\alpha_{i1}x_1 + \alpha_{i2}x_2 + \dots + \alpha_{in}x_n}{S_i} \quad (4.2)$$

where S_i is the speed of processor P_i . This scales each processor load so that they can be compared. To equalize the loads at the lowest level, the equation

$$\max_{i=1}^m (P_i) \quad (4.3)$$

must be minimized, where P_i is found by the equation in 4.2. Equation 4.3 written out in full would then be

$$\max_{i=1}^m \left(\frac{\alpha_{i1}x_1 + \alpha_{i2}x_2 + \dots + \alpha_{in}x_n}{S_i} \right) \quad (4.4)$$

It should be noted that the solution space to equation 4.4 is discrete. Furthermore, each equation (as presented in 4.2) will be discrete. This follows from the very nature of the CFD problem. When a grid is mapped to a processor, the load of the processor will not change smoothly, but jumps to the new load. This is accomplished in Equation 4.2 by changing some α_{ij} from 0 to 1 or 1 to 0. This will change P_i by an amount x_j . Since x_j is an integer that can never be smaller than 1 (grid sizes are not 0 or negative) P_i must increase (or decrease) by at least one, and in most cases considerably more. Furthermore, x_j must always be an integer - it represents the number of grid points a processor computes, and a processor will never be mapped just part of a grid point. Therefore, the solution space of Equation 4.2 must be discrete. Since Equation 4.4 takes the maximum of i discrete integer equations, it cannot be continuous either.

Equation 4.4 still does not account for the inter-processor communication times of boundary conditions. These times only need to be computed for the processor that will take the longest to compute its grids - the other processors will send their traffic while they are waiting for that processor to finish. The inter-processor time, IP, for that processor i , can be calculated as

$$IP = \frac{\text{Total number of bytes sent from processor } i}{\text{BAUD rate of processor } i} \quad (4.5)$$

It should be noted that the solution to equation 4.5 will change as the grid mappings change. This combines with Equation 4.4 to produce the final equation to minimize.

$$\max_{i=1}^m \frac{\alpha_{i1}x_1 + \alpha_{i2}x_2 + \dots + \alpha_{in}x_n}{S_i} + IP \quad (4.6)$$

4.2 Proof of NP-Hardness

The complexity of an algorithm is a function of its input size, n . If this is a polynomial function of n , then the algorithm is said to have polynomial computation time. However, there exists a large class of problems for which no polynomial time solutions have been discovered. This class of problems also has the interesting property that if any one problem in it is solvable in polynomial time, then all of them can be solved in polynomial time. These problems are called NP-Complete. An NP-Hard problem is an even more complex version of an NP-Complete problem. Therefore no polynomial time solutions

have been discovered for NP-Hard problems. For more information, see the book Fundamentals of Computer Algorithms, by Ellis Horowitz and Sartaj Sahni. [6]

The solution to Equation 4.4 is NP-Hard, and since Equation 4.6 is an extension of Equation 4.4 it is NP-Hard as well. This will be proved by comparison of a restatement of Equation 4.4 with a problem that is known to be NP-Complete:

Problem 4.1: Restatement of Equation 4.4

Each grid for each timestep is a task $t \in T$ which takes some length of time $l(t) \in \mathbb{Z}^+$ to compute, where $l(t)$ is found by Equation 4.1. There is also some number $m \in \mathbb{Z}^+$ of processors.

What is the m -processor schedule (in this case the schedule consists of a mapping of grids to processors) that minimizes the overall computation time C ? In other words, what is the function $\sigma : T \rightarrow \mathbb{Z}_0^+$ s.t. for all $u \geq 0$, the number of tasks $t \in T$ for which $\sigma(t) \leq u < \sigma(t) + l(t)$ is no more than m (i.e., there will not be more than m tasks running at any time u) and, for all $t \in T$, $\sigma(t) + l(t)$ is minimized?

The following similar problem is known to be NP-Complete.

Problem 4.2: INSTANCE: Set T of tasks, number $m \in \mathbb{Z}^+$ of processors, length $l(t) \in \mathbb{Z}^+$ for each $t \in T$, and a deadline $D \in \mathbb{Z}^+$.
 QUESTION: Is there an m -processor schedule for T that meets the overall deadline D , i.e., a function $\sigma : T \rightarrow \mathbb{Z}_0^+$ such that, for all $u \geq 0$, the number of tasks $t \in T$ for which $\sigma(t) \leq u < \sigma(t) + l(t)$ is no more than m and such that, for all $t \in T$, $\sigma(t) + l(t) \leq D$?

Problem 4.1 is the optimization of Problem 4.2. Since the optimization of a NP-Complete problem is NP-Hard, Equation 4.4 is NP-Hard.

4.3 Heuristic Solutions

All known optimal solutions to Problem 4.1 “require computation time that grows exponentially with the number of tasks.” [2 p1] However, some heuristics have been found which solve the problem in polynomial time, but only with “near-optimal” solutions.

The most common of these heuristics are known as Largest Processing Time first (LPT) algorithms. The basic algorithm, as introduced by R.L. Graham [4] and reviewed by Donald Friesen [3] consists of m identical processors and n tasks. These tasks are sorted in nonincreasing order of execution time, and then sent off to the first free processor for computation. When that processor finishes, it sends the job back and is sent another task (the largest of those available.) The major drawback is that LPT algorithms do not map tasks to processors, and due to the size of the CFD grids this thesis began with the assumption that grids must be mapped to processors to save networking time. The second drawback is that the upper bound on the solutions this algorithm finds is about 33% longer than the optimal solution. Given the exceeding long execution time of a CFD problem, this can cause quite a delay. Furthermore, where this algorithm has been applied to processors of differing speeds as is the case in this thesis, the worst case (as

found by Friesen) is between 1.52 and 1.67 times optimal - this is improved from 1.5 to 2 times optimal in previous solutions. [3]

E.G. Coffman found a solution using Bin-Packing methods that can find a solution within 1.220 of optimum. [2] However, this solution requires processors of identical speed. Essentially, he creates several bins with a given load capacity. The tasks are sorted in nonincreasing order, and each task in succession is placed into the lowest indexed bin in which it will fit without violating the capacity constraint. The algorithm then attempts to find the lowest capacity such that the minimum number of bins required is less than or equal to the number of processors. There are again two problems with this algorithm. The first, the "identical processor" restraint, can be overcome by setting the number of bins to the number of processors at the start and assigning each bin a scaling factor that reflects the processor speed. The task is then multiplied by the scaling factor before it is placed into the bin. The remaining difficulty is then that this only guarantees a solution within 1.220 of the optimum.

4.4 Simulated Annealing

Simulated Annealing is a relatively new method, but has had good results at solving some very difficult problems, such as the traveling salesman problem. It is capable of

sifting through many local minima to find the global minima, and of doing this in a factorially large discrete solution space.

Simulated Annealing is drawn from thermodynamics. It mimics the way metals cool and anneal. As the metal cools, the formerly active molecules slow down. They often line themselves up in a ordered crystal that is the state of minimum energy for the system. However, the system must be cooled slowly for this to happen. The Boltzmann probability distribution, Equation 4.7, “expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E .” [7 p445]

$$\text{Prob}(E) \sim \exp(-E/kT) \quad (4.7)$$

The variable k in the equation is Boltzmann’s constant, which relates temperature to energy. [7] The equation means that there is always a chance that the system is in a high energy state, but that chance becomes smaller as the system cools. Being in a high energy state allows the system to make a change that does not necessarily go toward a minimum, which also allows it to break out of local minima. Essentially, if the system is in a very high energy state virtually all changes suggested will be made, whether they are good or not. As the energy state decreases, the probability of making a “bad” change decrease with it. [1] The algorithm is shown in Figure 4.1.

```

BEGIN
  t=t0;
  i=istart; /* initial grid mapping */
  REPEAT
    FOR lcv = 1 TO num_steps DO /* Try up to num_steps temperature steps
                                  */
      BEGIN
        GENERATE (new grid mapping j from old grid mapping i);
        IF (COST(j) ≤ COST(i)) THEN i = j;
        ELSE
          IF  $\exp\left(\frac{\text{cost}(i) - \text{cost}(j)}{t}\right) > \text{random}[0,1)$  THEN i = j;
          /* if an improvement, make the change.
             If not, make the change with the given probability */
        END;
        t = t * TFACTOR; /* scale t down by some factor */
      UNTIL stopcriterion;
  END;

```

Figure 4.1 Algorithm for Simulated Annealing [1] [7]

4.5 OPTMOD

The Optimization Module, OPTMOD, uses the Metropolis Algorithm of Simulated Annealing. The Metropolis Algorithm makes use of four problem specific pieces.

According to Numerical Recipes in C, page 446, these are:

1. A description of possible system configurations.
 2. A generator of random changes in the configuration; these changes are the “options” presented to the system.
 3. An objective function E (analog of energy) whose minimization is the goal of the procedure.
 4. A control parameter T (analog of temperature) and an *annealing schedule* which tells how it is lowered from high to low values, e.g., after how many random changes in configuration in each downward step in T taken, and how large is that step. The meaning of “high” and “low” in this context, and the assignment of a schedule, may require physical insight and/or trial-and-error experiments.
- [7]

The system configuration used is simply the way in which grids are mapped to processors. The objective function to be minimized is Equation 4.7, shown above, and is implemented as a “cost function” (see below). The rate with which T is lowered depends upon the number of grids - the more grids, the more slowly T is lowered (to allow more time for the procedure to improve the mapping for more complicated problems). The final element is thus how to generate a change in the system.

4.5.1 Generator for New Mappings

OPTMOD has two possibilities when it generates a change in the grid mapping, with a 50% possibility for each choice. The first choice is to swap two grids between processors. For example, this could take grid x on processor A and grid y on processor B and change it to grid y on processor A and grid x on processor B. The other option is to change just one grid from one processor to another - taking grid x on processor A and mapping it to processor B instead. The grids are chosen randomly, and the previous

choice and current mapping have no effect on the grid selection process. Thus, the probability of making a transition is equal to the probability of making the reverse transition. Furthermore, the probability of a grid moving from processor i to processor k is equal to the probability of that grid moving from processor i to processor j multiplied by the probability of that grid then moving from processor j to processor k . In addition, the temperature is scaled down in each iteration, so that the limit of the temperature as the iterations continue is zero. These are the conditions specified by Theorem 3.3 of Simulated Annealing and Boltzman Machines for asymptotic convergence to the set of globally optimal solutions. [1, p.42] This change generator can take any mapping and change it into any other mapping, given enough moves. This is easily shown using the second change generator. To make any mapping into another mapping, just use the second change generator to move each grid from its original processor to its final processor. The first change generator adds one major ability - swapping two grids is less likely to overload one processor than merely giving a processor an extra grid. However, this by itself would be prone to finding only local minima, since a global minima may very well have some processors with many grids and other processors with no grids at all (for very fast and very slow processors respectively). The first change generator alone would not find that solution while in combination they have found such solutions to example problems.

4.5.2 Cost Function

Once the grid generator has come up with a new mapping, there must be a way to determine if that mapping is better or worse than the old mapping. The cost function performs this task. It does this by evaluating Equation 4.6 for the new mapping. If this gives a smaller value, the change is kept, if not the change is not kept. However, even if the change is for the worse, it might be kept with the probability given in Boltzmann's probability distribution (Equation 4.7). That gets the algorithm out of local minima.

The procedure begins by finding the maximum amount of time any processor is idle. It then determines which network has the most processors. For all processors on some other network, it assigns a one second penalty. One second was chosen since it was the smallest integer that would still have a positive effect. This prevents the Optimizer randomly selecting processors over several networks. A penalty was originally given to putting grids that share a boundary on separate processors, to prevent extra network traffic time, but the program that was modeled by the Discrete Event Simulator sent all boundaries over the network to a controller which then returned them to the receiving boundary. Thus it made no difference if the two were on the same processor or not.

4.6 Performance of the Simulated Annealing Algorithm

Some of the other heuristics, such as the bin packing method, would at first appear to make a viable alternative to Simulated Annealing. However, Simulated Annealing was

selected for two reasons. The first is that it does find an optimum, which was the goal of this thesis. While it may not find the true optimum, it will (almost always) find a solution that "cannot be significantly improved upon." [7 p446] The second is that this computation is performed within polynomial time. As implemented, it runs for a number of loops equal to 100,000 times the number of grids. This will increase the execution time of the algorithm by quite a bit as the number of grids increases, but it will increase linearly. Given that the solution space for n grids and p processors is p^n this is not too bad. However, due to a stop criterion that stops the algorithm if no further improvements are made over an iteration, this may perform slightly better than $(100,000 * n)$ for small n .

Another excellent reason for the selection of Simulated Annealing is the ease of modifying the Cost function (explained in 4.5.2) so that any changes that may be made to the algorithm being optimized can be handled quickly and easily. No other heuristic discussed is quite as viable.

Some timings were made of this algorithm as implemented in Section 4.4 on an Gateway 2000, Intel Pentium 90 MHz computer running under DOS 6.2. All timings were made for a two-processor system. For a four grid problem, the algorithm ran in .38 seconds. It took 1.09 seconds for an 8 grid case. Thus, for a problem that was twice the size, it actually took 2.87 times as long to run. For further comparison, a twelve grid case was

timed, taking 1.98 seconds. This took 1.8 times longer to run than the 12 grid problem, and was 1.5 times the size. It also took 5.21 times longer than the four grid problem, but is only three times the size. This included a statement that stopped the algorithm if no further improvements were made over all iterations on one temperature, which improves the execution time of the algorithm but throws the comparisons off. For a better comparison, therefore, that break was taken out so that a “worst case” comparison could be made. The four grid case took 4.45 seconds, compared to 11.19 for the eight grid case and 17.74 for the 12 grid case. In this case, it took 2.5 times as long to compute a problem that was twice the size, and 3.99 times as long to compute one that was three times the size. The 8 to 12 grid increase, a 1.5% increase, took 1.58 times as long to compute. For demonstration purposes, timings with the break statement were made for a four processor system, which squares the size of the solution space. Timings for the 4, 8 and 12 grid problems were .44, 1.48, and 3.84. The large jump for the 12 grid problem were due to the break statement being activated later. Without the break statement, the timings became 4.95, 11.37, and 19.12. This shows the linear increase predicted.

Chapter V

Results

This program simulated and optimized the mapping of grids to processors. The results were compared to trial runs of a four grid case and an eight grid case. These runs were run on four DEC AXP 3000/800 200 MHz workstations connected by a gigaswitch by Chris Humphres. In all cases, the Discrete Event Simulator closely matched the actual execution times, and the Optimizer came up with optimized grid mappings.

In this chapter, “real” refers to the actual execution of a CFD problem on some system, and “simulated” refers to the modeling of some run of a CFD problem on some distributed memory MIMD system by the Discrete Event Simulator.

5.1 Simulation Results

Figures 5.1 and 5.2 present some real execution times of a four grid nozzle problem and compare them to simulated runs of the same problem. Each grid consisted of 54,000 points - the original problem was adjusted to make this the case. Figure 5.1 has timesteps that conform to a real run of the problem, while Figure 5.2 has example and trial runs.

The largest error, 10.7% in the two processor case in Figure 5.1, was due to another job being performed on the workstations while the CFD problem was being computed. This slowed the run down and skewed the results. Even so, the Discrete Event Simulator

	1 processor 425 timesteps	2 processors 425 timesteps	4 processors 425 timesteps
Real Execution Time	4:55:47	2:38:42	1:15:06
Simulated Execution Time	4:43:20	2:21:40	1:17:55
Percent Error	4.20	10.70	3.75

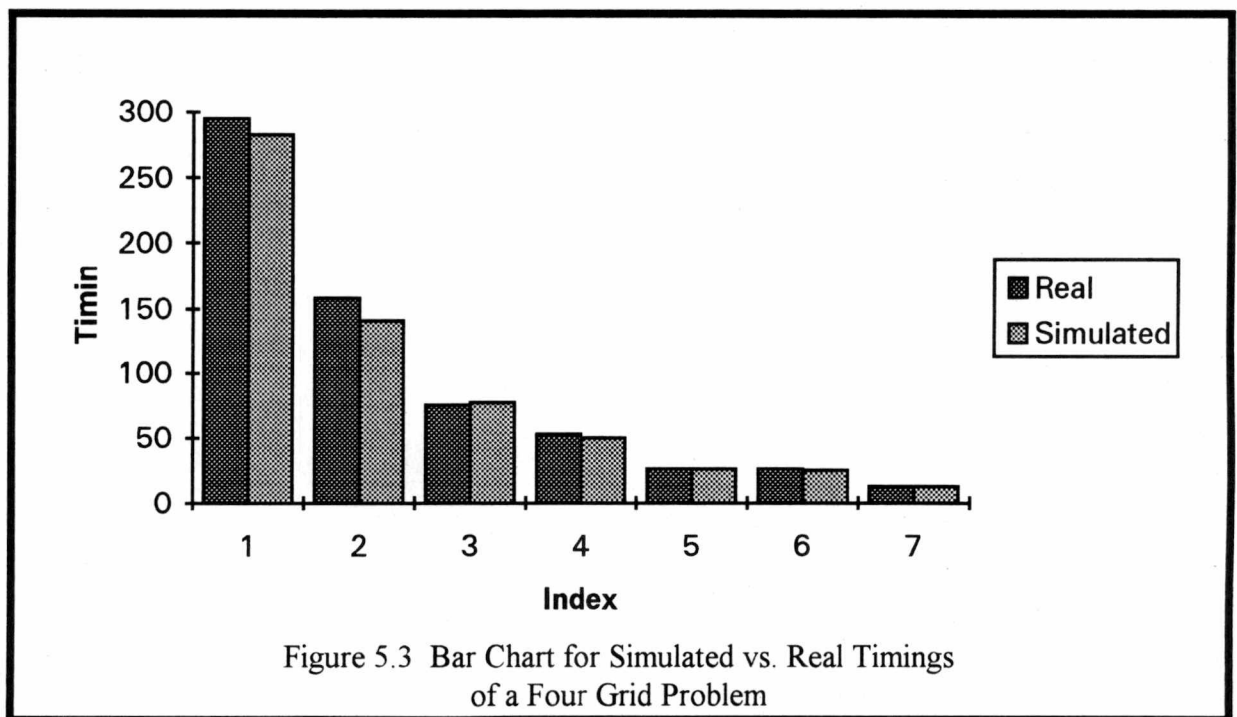
Figure 5.1 Simulated vs. Real Execution Times (425 Timesteps)

	2 Processors 150 Timesteps	2 Processors 75 Timesteps	3 Processors 75 Timesteps	4 Processors 75 Timesteps
Real Execution Time	0:53:33	0:26:53	0:26:26	0:13:43
Simulated Execution Time	0:50:00	0:25:00	0:26:15	0:13:45
Percent Error	6.63	7.01	.69	.0024

Figure 5.2 Simulated vs. Real Execution Times (Varied Timesteps)

performs quite well in approximating the execution times of these problems. The percentage of error is within an acceptable range, especially considering that processors may be performing other jobs. The timings between simulated and real runs remain close, as shown in the graphical comparison in Figure 5.3. The given timings correspond to hours:minutes:seconds.

It should also be noted that there was almost a linear decrease in execution time by using extra processors. Two processors took 53% of the time one processor took - but this number may be off due to the skewed results of that two processor run explained above. Four processors took just 25.4% of the time one processor took, and 47.5% of the two



Index to Figure 5.3

- 1: 1 processor, 425 timesteps
- 2: 2 processors, 425 timesteps
- 3: 4 processors, 425 timesteps
- 4: 2 processors, 150 timesteps
- 5: 3 processors, 75 timesteps
- 6: 2 processors, 75 timesteps
- 7: 4 processors, 75 timesteps

processor run. The simulated runs also show a linear decrease in execution time, with two processors performing in 49.8% of the time taken by one processor. Four processors took 54.6% of the two processor run, and 27.2% of the one processor run. These are almost exactly the results hoped for.

Only one run of the eight grid case has been completed. That was run on one workstation for 75 timesteps. The resulting time was 1:48. The simulated time was 1:42, giving a percent error of 5.55%. However, it is believed that this run was slightly slow due to a program inefficiency which is currently being corrected. If so, the error is actually smaller than 5.55%.

5.2 Optimization Results

Three problems were available to work the Optimizer on. One was a four grid nozzle problem - this was the original problem that the problem actually run was derived from. That derived problem is trivial to optimize, since all of its grids are the same size, and

only the original problem will be discussed. The second is an eight grid nozzle problem, which was basically an extension of the derived four grid problem, but with the extra grids being different sizes. The final problem is a twelve grid problem of a store and holding rod. It is a CFD problem actually worked by Calspan Corporation. Grid sizes are shown in Figure 5.4. It should be noted that none of these examples have, at the time of this writing, been run in parallel. All results and improvements that will be

	Four Grid Problem	Eight Grid Problem	Twelve Grid Problem
Grid 0	10,000	54,000	89,460
Grid 1	10,800	54,000	79,375
Grid 2	9,000	54,000	66,675
Grid 3	6,750	54,000	52,877
Grid 4		64,800	75,625
Grid 5		36,000	171,479
Grid 6		54,000	171,479
Grid 7		67,500	171,479
Grid 8			28,886
Grid 9			28,886
Grid 10			28,886
Grid 11			28,886
TOTAL	35,550	438,300	993,993

Figure 5.4 Grid Sizes and Numbering

shown are based solely on the Discrete Event Simulator. Each example given was simulated for 425 timesteps. All "worst case" scenarios presented assume that each

processor will have the same number of grids (otherwise each worst case scenario would just be all grids on one processor).

5.2.1 Four Grid Case

The four grid case was simulated on two processors of the same speed (5000 grid points per second). Processors of the same speed were chosen so that processor loads could be compared. The Optimizer mapped grids 1 and 3 to processor A, and grids 0 and 2 to processor B. This gave A 48.0% of the work load, and B 52.0%. This resulted in an simulated execution time of 21 minutes and 15 seconds. The worst case scenario is to have grids zero and one on processor A, and grids 2 and 3 on processor B. That gives A 56.9% of the load, and B 43.1%, and is estimated to take 28 minutes and 20 seconds to run. The optimized output gives a 25% faster throughput. This is a trivial case, but it shows the value of optimization.

5.2.2 Eight Grid Case

This was first simulated on two processors, and then on four processors. For the two processor case, the Optimizer mapped grids 0,2,3,and 6 to processor A, and grids 1,4,5, and 7 to processor B. Processor A thus had 49.3% of the work load, while processor B had 50.7%. This gave an simulated execution time of 5 hours, 4 minutes and 35 seconds. The worst case scenario run mapped grids 2,3,5, and 6 to processor A and 0,1,4, and 7 to processor B, giving them 45.2% and 54.8% of the workload, respectively. This gave a simulated execution time of 5 hours, 18 minutes, and 45

seconds. The optimized mapping was thus 4.44% faster. While this increase is not very significant, it should also be noted that the grid sizes in this case do not vary much - and thus optimization will not help too much either, although it did a bit better in the four processor case.

For that four processor case, the Optimizer mapped the grids as follows.

<u>Processor</u>	<u>Grids</u>	<u>Work Load</u>	<u>Percentage of Total Load</u>
0	3,6	108,000	24.6
1	1,4	118,800	27.1
2	5,7	103,500	23.6
4	0,2	108,000	24.6

This resulted in a simulated execution time of 2 hours, 42 minutes, and 55 seconds.

The worst mapping was as follows.

<u>Processor</u>	<u>Grids</u>	<u>Work Load</u>	<u>Percentage of Total Load</u>
0	3,6	108,000	24.6
1	1,5	90,000	20.5
2	4,7	132,300	30.2
4	0,2	108,000	24.6

This resulted in a simulated execution time of 2 hours, 57 minutes, and 5 seconds. The improvement due to optimization was 8%. This is not a huge improvement, but again the example problem is an almost trivial example and even in the worst case scenario the work loads do not vary by more than 9.7%.

5.2.3 Twelve Grid Case

For completeness, and to show the value of parallelization, this was simulated on one, two, three, and four processors. In each case, the Optimizer came up with a mapping that was a true optimum. For one processor, the Discrete Event Simulator estimated an execution time of 22 hours and 40 minutes. For two processors, the estimated execution time for an optimized mapping dropped to 12 hours, 3 minutes, and 8 seconds. Three

and four processors were 8:51:15 and 6:50:50, respectively. The optimized mappings were as follows (refer back to Figure 5.4 for the individual grid sizes.)

Two processors

<u>Processor</u>	<u>Grids</u>	<u>Work Load</u>	<u>Percentage of Total Load</u>
0	0,1,3,4,7,8	497,702	50.0
1	2,5,6,9,10,11	496,291	49.9

Three Processors

<u>Processor</u>	<u>Grids</u>	<u>Work Load</u>	<u>Percentage of Total Load</u>
0	1,3,7,8	332,617	33.5
1	4,5,9,10,11	333,762	33.6
2	0,2,6	327,614	33.0

Four Processors

<u>Processor</u>	<u>Grids</u>	<u>Work Load</u>	<u>Percentage of Total Load</u>
0	0,1,4	244,460	24.6
1	3,6,9	253,242	25.5
2	2,5	238,154	24.0
3	7,8,10,11	258,137	26.0

Three other mappings were simulated for each processor system, a worst-case run and two others by mapping grids to processors in some order. These timings were as follows.

<u>Run</u>	<u>Two Processors</u>	<u>Three Processors</u>	<u>Four Processors</u>
Optimized	12:03:08	8:51:15	6:50:50
Worst Case	19:21:40	15:13:45	13:27:30
1	12:30:50	13:41:40	7:26:15
2	12:45:17	15:13:45	9:33:45

The Optimizer found the fastest mapping in each case, and it can be seen that a significant decrease in execution time will thus be gained. In the four processor case, a estimated 50% decrease in execution time is gained from the worst case scenario, while a 8% and 29% decrease were estimated from runs 1 and 2 respectively.

Chapter VI

Conclusions

The goal of this research was to minimize the computational time of Parallel CFD flow solvers by optimizing the mapping of grids to processors, and approximate the computational time of any given mapping. The program to implement this goal appears to perform both these functions well. Each approximation made by the Discrete Event Simulator closely matched the actual execution times of the CFD flow solver.

Furthermore, the application of Simulated Annealing to the optimization problem resulted in a globally optimized mapping for each problem. The decrease in simulated execution time gained by this optimization has ranged from 4.44% to 29%, and 97% for a worst-case scenario. The smaller decreases in execution time were largely due to the CFD problem utilizing generally equally sized grids, while a decrease in execution time of up to 97% appears to make this program a valuable tool in this application of Parallel CFD.

Further research can test the usefulness of these algorithms by applying them to other flow solvers and distributed systems of workstations. More data points to verify the accuracy of the Discrete Event Simulator would be useful. Finally, the Optimizer should be analyzed and modified until it can be guaranteed that it will produce results within some fixed or probabilistic percentage of the true optimum.

List of References

List of References

- [1] Aarts, Emile and Jan Korst. Simulated Annealing and Boltzmann Machines. New York: John Wiley and Sons, 1989.
- [2] Coffman Jr., E.G., M.R. Garey, and D.S. Johnson. "An Application of Bin-Packing to Multiprocessor Scheduling." SIAM Journal on Computing. Vol. 7, No. 1, February 1978.
- [3] Friesen, Donald K. "Tighter Bounds for LPT Scheduling on Uniform Processors." SIAM Journal on Computing. Vol. 16, No. 3, June 1987.
- [4] Graham, R.L. "Bounds on Multiprocessing Timing Anomalies." SIAM Journal on Applied Mathematics. Vol. 17, No. 2, March 1969.
- [5] Gropp, William D and Edward B. Smith. "Computational Fluid Dynamics on Parallel Processors." Computers and Fluids Vol. 18, No. 3 (1990): pp. 289-304.
- [6] Horowitz, Ellis and Sartaj Sahni. Fundamentals of Computer Algorithms. Rockville, MD: Computer Science Press, 1984.
- [7] Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical Recipes in C, Second Edition. New York: the Press Syndicate of the University of Cambridge, 1992.
- [8] Roose, D. and R. Van Driessche. "Distributed Memory Parallel Computers and Computational Fluid Dynamics." von Karman Institute for Fluid Dynamics Lecture Series 1993-04, March 15-19, 1993.
- [9] Schreck, E. and M Peric. "Computation of Fluid Flow with a Parallel Multigrid Solver." International Journal for Numerical Methods in Fluids Vol. 16 (1993) pp. 303-327.

VITA

R. Evan Reynolds, son of R. Neil and Elaine Reynolds, was born in Tallahassee, Florida on August 30, 1969 and raised in Waterford, Virginia. He graduated from Loudoun Valley High School in June, 1987. In May, 1992 he received a Bachelor of Science degree for Math/Computer Science from Mary Washington College in Fredericksburg VA. The following August he entered the University of Tennessee Space Institute and received a Master of Science degree in Computer Science in December of 1994.

Evan is currently known for telling peanut jokes ("Two peanuts crossed the street and one was assaulted") and for having spliced a Nintendo Power Glove into his PC for a virtual reality hand. He humbly apologizes for not having thought of a way to work that into his thesis. Computational Fluid Dynamics needs a Nintendo Power Glove.