PERFORMANCE MODELING OF THE FINITE-DIFFERENCE TIME-DOMAIN METHOD ON PARALLEL SYSTEMS*

James E. Lumpp, Jr., Shashi K. Mazumdar,†Stephen D. Gedney
Department of Electrical Engineering
University of Kentucky
Lexington, KY 40506, USA

ABSTRACT. *As high-performance parallel codes are developed or ported to new architectures, it is often difficult to quantify the the causes of performance problems. Models of program performance can provide users with insight into the effect of system and program parameters on performance, can help programmers tune applications, and can help programmers make decisions about processor allocation. This paper introduces a modeling technique applied to the Finite-Difference Time-Domain (FDTD) algorithm. The technique models the performance of an existing application in terms of the size of the problem being solved and the number of processors. The models show that for sufficiently large problem sizes the algorithm performs well. However, for smaller problem sizes or when too many processors are used, the models show that parallel overheads become significant.*

1 INTRODUCTION

Over the last decade, advances in high performance computing have had a significant impact on computational electromagnetics. Specifically, with the decreasing costs of high speed memory, RISC processors, and high speed networks, the size and complexity of practical engineering problems that can be solved using full wave analyses have dramatically increased. Concurrently, the focus of high performance computing has shifted from expensive high-speed single processor computers to relatively low cost multiprocessor computers.

Programming paradigms have changed with the development of parallel systems to optimize the performance of traditional sequential algorithms. Specifically, specialized algorithms have been developed to exploit parallel systems to minimize CPU times and memory usage. Unfortunately, in a parallel computing environment, there are a number of factors that can affect an algorithm's performance that are not obvious to the developer *a priori*. Thus, an invaluable resource for the development of efficient parallel codes is a robust performance analysis tool. Such a utility can enable the programmer to analyze the code and identify inefficiencies for a specific architecture. It can also provide insight needed to fine-tune the algorithm and optimize its performance. Furthermore, for a production level code, a performance analysis tool can provide a close estimation for the optimum number of processors, the optimal decomposition, and the total execution time for a given problem dimension and target parallel system.

Techniques for performance tuning of parallel programs can be broadly categorized into measurement based techniques and modeling based techniques. In measurement based techniques the program is instrumented using hardware or software instrumentation [14, 39, 25, 26]. The instrumented program is executed and information is collected in the form of trace data and this data is analyzed to find performance problems. This process is repeated until the desired level of performance is achieved.

Measurement based techniques suffer from three main disadvantages. First, they require multiple executions of the program on dedicated systems. It can be difficult for programmers to obtain exclusive access to high-performance systems for performance tuning. Second, the instrumentation used during program execution can change the behavior of the program. This can make it difficult to determine how the program would behave without instrumentation. Finally, with measurement based techniques it is difficult to predict the performance if parameters such as the size of the problem, the architecture, or the type of network change.

Modeling based techniques characterize the performance of the program in terms of specific program parameters such as the algorithm, the communication or synchronization behavior, the problem size and system parameters like the number of processors, and network latency. In addition, performance models can be developed for existing and non-existing programs or architectures. Models are useful for comparing the performance of programs on a wide variety of environments as well as different algorithms for a particular program.

In simulation modeling, the salient features of the

*This research was supported by the National Science Foundation under Grants CDA-9502645 and ECS-9624628, the Advanced Research Projects Agency under Grant DAAH04-96-1-0327, and the Army Research Office under Grant DAAH04-94-G-0243.

†Author's current address is: Stratus Computer, Inc., San Jose, CA 95125

program and system are used to drive a simulation [8, 4, 36, 12] . Simulation models offer great flexibility, however, simulations can require orders of magnitude more time to run than the actual program. In addition, accurate simulations require sophisticated and detailed models that can be difficult to generate. For existing programs, direct execution simulation techniques have been developed that greatly improve the simulation time by using the actual programs which has the added benefit of reducing the complexity involved in model creation [36, 4, 31].

Performance models that parameterize program behavior as scalars or mathematical functions can also be developed [29, 10, 38, 13, 1]. These models do not require any time for simulation, however, accurate predictions still require detailed models of the program and computer system[29, 34, 27]. When modeling existing parallel program, the structure of the program can be used as the starting point[3, 9, 7, 33]. Techniques for modeling existing programs typically consist of *static* analysis of source codes to generate the model structure and run-time or *dynamic* analysis to generate the final model.

Several performance modeling tools have been developed that make use of static and dynamic analysis to model parallel programs [3, 9, 7]. The Modeling Kernel [3], which is a part of the AIMS instrumentation toolkit [39], models a program based on the duration of sequential blocks, message lengths and communication phases. APACHE models programs using separate computation models and communication models [7]. APACHE instruments the application to determine the computational requirements, its branching behavior, and the number and size of messages.

In this paper, a performance modeling technique that can help programmers to identify problems in programs or inefficiencies in the interaction between programs and a given architecture is presented. The technique consists of a static modeling process based on the actual program source code and dynamic analysis of the run-time behavior of the program. The dynamic analysis is performed using only the overall execution time of the application. As a result, instrumentation to observe communication patterns or procedures called is not required. This minimizes the intrusion on the run-time behavior of the program.

The performance modeling tool developed is applied to the analysis of the performance of the finite-difference time-domain (FDTD) solution of Maxwell's equations [37, 35]. The FDTD method is based on an explicit time-marching solution and has the advantage that time-variant electromagnetic fields can be accurately and efficiently modeled within inhomogeneous, non-linear, and anisotropic media.

The FDTD algorithm is an excellent algorithm for parallel computer systems. The kernel of the algorithm is a directly addressed sparse matrix-vector multiply and can be efficiently implemented using a simple nested loop structure. Implementation on parallel computers requires message passing only between neighboring processors, and has lead to a highly scalable algorithm [18, 23, 30, 11]. By exploiting parallel architectures based on high-performance RISC processors, the problem sizes that can currently be efficiently solved using the FDTD are orders of magnitudes larger than problems that could have been treated a few years ago.

In this paper the parallel FDTD algorithm is presented. The algorithm is based on the traditional Yee algorithm [37, 40] with a uniaxial PML absorbing media [32, 20, 19]. The performance modeling techniques show that for sufficiently large problem sizes or small numbers of processors, the FDTD algorithm performs quite well. When the problem sizes are small (or number of processors large) the models show that the parallel overheads can become significant and performance decreases. The models also uncovered a performance problem for a particular processor configuration in FDTD. By restructuring loops that communicate boundary information, the performance was substantially improved.

2 THE FDTD ALGORITHM

The finite-difference time-domain algorithm is a direct solution of Maxwell's equations for the electric and magnetic field intensities in a finite, piecewise homogeneous space. The algorithm is based on the discretization of Maxwell's curl equations (Ampère's and Faraday's Laws) using central difference approximations of the spatial and time derivatives. This is achieved by projecting orthogonal components of the vector fields onto the edges of a dual, staggered, orthogonal grid. By staggering the vector fields both in space and time, a second-order accurate explicit time-marching solution is obtained [37].

One of the most challenging aspects of the FDTD method is implementing an absorbing boundary condition that can accurately truncate the mesh over broad frequency bands. The perfectly matched layer (PML) absorbing media introduced by J. P. Berenger [2] has been demonstrated to be a highly effective method for the termination of FDTD lattices [6, 28, 22] and can result in reflection errors as minute as -100 dB. Recently, it has been shown that the PML method can be reposed in a Maxwellian form as a uniaxial anisotropic medium [32, 20, 19]. It has also been demonstrated that the uniaxial medium can be perfectly matched to a lossy, inhomogeneous, dispersive, isotropic and anisotropic medium [20]. Most significant is that the extension to such complex media in a FDTD implementation is quite trivial.

The time-dependent electric and magnetic fields

within the uniaxial PML are computed using an explicit time-marching solution scheme, as derived in [20, 19].[1] The uniaxial PML can be easily and efficiently implemented within the framework of an existing FDTD code. For example, posing a uniaxial PML throughout the entire space, the discrete field updates can be expressed as a triple-nested loop (illustrated here in FORTRAN):

```
do 10 k = 1,nz-1
  do 10 j = 2,ny-1
    do 10 i = 2,nx-1
      ds = dz(i,j,k)
      dz(i,j,k) = ay(j)*dz(i,j,k) +
                  by(j)*[hy(i,j,k)-hy(i-1,j,k)-
                  hx(i,j,k)+hx(i,j-1,k)]
      ez(i,j,k) = ax(i)*ez(i,j,k) +
                  bx(i)*[az(k)*dz(i,j,k)-bz(k)*ds]*
                  er_z(i,j,k)
10 continue
```

where

$$ay(j) = \frac{2\varepsilon_0\kappa_{y_j} - \sigma_{y_j}\Delta t}{2\varepsilon_0\kappa_{y_j} + \sigma_{y_j}\Delta t},\ by(j) = \frac{2\varepsilon_0\Delta t}{2\varepsilon_0\kappa_{y_j} + \sigma_{y_j}\Delta t}\cdot\frac{\Delta z}{\Delta x\Delta y},$$

$$ax(i) = \frac{2\varepsilon_0\kappa_{x_i} - \sigma_{x_i}\Delta t}{2\varepsilon_0\kappa_{x_i} + \sigma_{x_i}\Delta t},\ bx(i) = \frac{2\varepsilon_0\Delta t}{2\varepsilon_0\kappa_{x_i} + \sigma_{x_i}\Delta t},$$

$$az(k) = \frac{\kappa_{z_k}}{\Delta t} + \frac{\sigma z_k}{2\varepsilon_0},\ bz(k) = \frac{\kappa_{z_k}}{\Delta t} - \frac{\sigma z_k}{2\varepsilon_0},$$

$$er_z(i,j,k) = \frac{1}{\varepsilon_0\varepsilon_r(i,j,k)}$$

and the fields have been scaled by their edge length (e.g., $E_x = \Delta x E_x$). It is noted that the PML parameters σ_i and κ_i $(i = x, y, z)$ are one-dimensional variables. Specifically, in the interior working volume, it is assumed that $\sigma_i = 0$, and $\kappa_i = 1$, and in the PML regions, they are assumed to have an m-th order polynomial spatial variation along their respective axes. As a result, the update coefficients above are simply one-dimensional coefficients.

Updating the fields over all space has the limitation that the additional storage arrays required to store the flux densities (e.g., Dz) must be stored over all space. However, it does offer the advantage of simplicity in the modification of existing codes. An alternative is to write a triple nested loop for the interior fields, and then write separate loops for the different PML regions (segregating corner regions). Then, only the auxiliary variables need to be stored in the PML regions, leading to memory savings. In this circumstance, the uniaxial PML will require considerably less storage than Berenger's PML, because only the normal fields require dual storage as opposed to the two tangential fields as required by Berenger's PML formulation. Based on this scheme, the FDTD with a uniaxial PML truncation on all 6 boundaries will require

$$6N_xN_yN_z + 8N_{pml}(N_xN_y + N_yN_z + N_zN_x) -$$

$$16N_{pml}(N_x + N_y + N_z) + 24N_{pml}^2$$

real numbers as compared to $12N_xN_yN_z$ real numbers required by a FDTD method with PML everywhere [21].

In this paper, the UPML is assumed to be distributed throughout the grid. While this increases the memory overhead, the overall computational time is slightly better than the case when the grid is split up into multiple regions [21]. Furthermore, load balancing is readily achieved.

3 PARALLEL IMPLEMENTATION OF THE FDTD ALGORITHM

It has been demonstrated that the finite-difference time-domain algorithm is well suited for implementation on tightly coupled distributed memory high performance parallel computers [30, 5, 15, 18]. This is principally due to the regularity of the dual grid and the even distribution of effort throughout the grid during the entire time-marching solution. Furthermore, only the magnetic fields tangential to the shared boundaries needed to be communicated between processors each time iteration [18]. The algorithm presented in this section is primarily focused on distributed memory multicomputers with a single program multiple data (SPMD) paradigm.

The parallel algorithm is based on a spatial decomposition of the regular, orthogonal FDTD lattice. To this end, the original domain is spatially decomposed into contiguous sub-domains. The sub-domains are rectangular in shape, non-overlapping, sharing common surfaces only, and are of equal size. The boundaries, or surfaces, shared by sub-domains are chosen by taking slices along edges of the primary grid along the x, y, and z-directions. Each sub domain is then mapped directly onto independent processors of the parallel computer.

Assume that the lattice has dimension $N_x \times N_y \times N_z$. The lattice is then discretized using a three-way dissection. Essentially, assume that the lattice is mapped onto a three-dimensional grid of processors (P_x, P_y, P_z), where P_x, P_y, and P_z are the number of subsections along the x, y, and z-directions, respectively. (Also, the total number of processors will be $P = P_xP_yP_z$). Then, given a global grid with dimensions N_x, N_y, and N_z, each processor will be assigned a block of the grid with dimensions $N_x/P_x, N_y/P_y$, and N_z/P_z . For most applications, these ratios will be non-integer values, and the sizes of the grids will be slightly uneven, resulting in some load imbalance.

The local grid dimension for each processor, given as $nxp \times nyp \times nzp$, can be uniquely determined using a simple algorithm. Each processor is first assigned a coordinate (p_x, p_y, p_z), where $p_x \in (1, P_x), p_y \in (1, P_y)$, and $p_z \in (1, P_z)$. Then, each processor can uniquely determine its grid dimension nxp along the x-direction using the algorithm

[1] The readers are referred to these references for the general theoretical development of the uniaxial PML.

$$nxp = aint(N_x/P_x)$$
$$if(mod(N_x, P_x).le.p_x)nxp = nxp + 1$$

where, aint() is the FORTRAN intrinsic function which truncates the argument to an integer, and mod() is the modulus FORTRAN intrinsic function which computes the remainder of the quotient N_x/P_x. This algorithm assures that no processor has more than one additional row of the grid than any other processor. The dimensions *nyp* and *nzp* can be computed in a similar manner. The parallel algorithm will then consist of updating all fields assigned to each processor independently, with special consideration for discrete field components that lie on the boundary interfaces shared by two sub-domains.

The spatial decomposition is chosen to slice along the primary lattice grid faces. Specifically, in the boundaries shared between any two processors, the discrete electric field vectors in the planar boundary are tangential to the surface. Thus, the discrete magnetic field vector is normal to the boundary interface. The fields on the shared boundary interface are redundantly stored in memory on both processors sharing that boundary. Due to the decomposition described, multiple processors can share a lattice edge. The discrete electric field vectors associated with the edge are assumed to be stored redundantly on all processors sharing the edge.

The magnetic field vectors normal to the shared interface can be updated independently on all processors sharing the face. The update is proportional to the line integral of the electric field about the edges bounding the face. Because each processor has the updated value of the tangential electric fields on the shared interface, the normal magnetic field can be updated independently on each processor. Therefore, interprocessor communication is not needed when updating the magnetic fields within each sub-domain. Rather, it is much more expedient to simply update the normal magnetic fields in the shared boundary redundantly on each processor sharing the face.

On the other hand, the discrete tangential electric field vectors on the shared boundary interface do not have enough information locally to perform the update. For example, consider an edge shared by two processors. Three of the four magnetic field vectors needed for the update are stored in local memory. The fourth magnetic field vector, which is tangential to the interface, is in memory on the adjacent processor. Hence, this data must be communicated to the local processor before the update can be completed.

The parallel algorithm performing the parallel FDTD algorithm is illustrated in Figure 1. The first step is to update the magnetic field intensity in all space. The next step is to send the magnetic field vectors tangential to the shared boundary (one-half cell removed) to the neighboring processor, while receiving the complimentary magnetic fields across the shared face and storing them in a local vector. To reduce the effects of message passing overhead, all the discrete magnetic fields to be communicated to a given processor are first combined into a single buffer and then sent to the adjacent processor. The electric fields are then updated, including those on the shared boundary. Note that the electric fields on the shared boundaries are updated redundantly on each processor to avoid additional communication.

```
initialize e,h to zero
do it = 1, max_iterations
        call source_update
        call h_update
        call communicate_h_field
        call e_update
enddo
```

Figure 1: Parallel FDTD Algorithm

It will be shown that the algorithm described is quite scalable. However, because the interprocessor communication must be performed at each time step, the efficiency of the parallel algorithm will ultimately stagnate as the number of processors is increased for a fixed problem size, as predicted by Amdahl's law. It will be shown that the performance analysis tool described in the following section can predict the optimum number of processors for a given problem size.

4 MODELING TECHNIQUE

Our approach to model the performance of parallel programs is by analyzing the structure of programs and then measuring the performance from actual program runs as key factors of the application and architecture are varied. This information is used to create a closed-form expression for the execution time of the program in terms of those factors.

The performance of a parallel program depends on many different attributes from architecture details to the structure of the algorithm. Some of these factors are under the control of the user and can be changed to improve performance. However, there are other factors that either can not be changed or the changes cannot be controlled by the user. While modeling techniques similar to those presented here can be used to model other factors affecting program performance (e.g., processor speed), the goal of the modeling technique presented here is to assist users in modeling programs primarily in terms of factors that can be controlled by an end user of a given parallel machine.

Examples of factors that are constant or change very rarely include the speed of the processors, the type of

interconnection network or its topology, or the size and speed of memory. While these factors directly influence performance, they can not be changed in a typical end-user environment. These factors are assumed to be constant and are included implicitly in our models.

Factors affecting performance that can change, but may not be under the control of the user, include the external load on individual machines or external load on the network (which can impact latency and available bandwidth). If the parallel machine is dedicated to a single user, these factors will not be a concern. However, for a typical distributed system these factors can greatly impact performance. These factors are implicitly included in the models if the dynamic analysis can be done with these factors at values similar to those that will be present when the program is used.

Typical factors that the user can control include the size of the problem, the number of processors used, and the algorithm used. The modeling approach presented here analyzes the program structure in terms of the control flow, loop structures and the communication and synchronization events.

During the static analysis phase, the program source is analyzed to generate a model template for the execution time. The model is based on identifying the basic blocks, loops, function calls, communication, and synchronization events. By determining the bounds on loops and the number and size of messages in terms of the problem size (S) and the number of processes (P), it is possible to determine the terms that will appear in the execution time model. A static model for execution time in terms of S and P is generated.

The final model of execution time is found by determining the coefficients of the terms in the static model. These unknown coefficients are estimated using regression on execution time data gathered during run-time. The following sections describe the approach in detail.

4.1 Static Analysis

The execution time of a parallel program can be divided into three categories. These are:

- **Computation Time:** This is the portion of time spent in performing actual computation.
- **Parallel Overhead Time:** This is the portion of time spent in exchanging information between the processes.
- **Synchronization Time:** This is the amount of time that the processes spend in coordinating their activities.

The computation time is the time spent performing actual computations that are required in the program. It depends on the number of operations performed in the program, and is often referred to as the *order* or *complexity* of the computation. The computational complexity of a program depends on the number of loops and routines that are present.

In a parallel program, there is always some overhead associated with transferring information between processes. The amount of time it takes to format and prepare to send a message or to initiate an access to a remote memory location is defined as the *Parallel Overhead Time.* This measures only the local time to process and send a message or to make a remote memory access and not the time it takes for messages to propagate between processes because the transmission time may overlap with computation or parallel overhead time.

Some form of synchronization between the processes is required to coordinate the activities in a parallel program. This is usually achieved by implementing locks, semaphores or barriers in shared memory systems or synchronous (blocking) communication calls in message passing systems. The time spent by a process blocked on a synchronization event is defined as *synchronization time.*

To accurately model the execution time of a parallel program, each of the three categories must be modeled. The following sections explain the techniques used to model each of these.

4.1.1 Computation Time

Computation time models are generated by dividing the program into *basic blocks* which are the largest consecutive blocks of instructions between control flow constructs in the program. The structure of each loop in the program is analyzed to determine the number of times loops will iterate to find the number of times each basic block will execute. If it is assumed that the execution time of a basic block is constant, then the execution time for a loop is the execution time of the basic block multiplied by the number of times the loop iterates.

For example, if the execution time for a given basic block in a loop is k then the total time for the basic block for n iterations of the loop would be $k*n$. For many loops it is possible to statically determine the number of iterations by analyzing the upper and lower bounds of the loop. Consider a loop l with an upper bound of u, a lower bound b, and the stride through the loop is s, then the number of iterations of the loop can be expressed as:

$$I_l = \frac{u-b}{s}$$

and the execution time for loop l can be expressed as:

$$E_l = k * \frac{u-b}{s} = k * I_l$$

If u, b and s are expressed in terms of the problem size S and the number of processors P, the execution time can then be expressed as:

$$E_l(S,P) = k * I_l(S,P)$$

Nested loop structures in programs must also be modeled. Consider a loop g with $I_g(S,P)$ iterations that is nested within a loop l. For each execution of the outer loop, the inner loop iterates $I_g(S,P)$ times. Thus, the total execution time of the outer loop can be written as:

$$E_l(S,P) = I_l(S,P)(k_0 + k_1 * I_g(S,P))$$

This can be generalized to any number of loops within l as follows:

$$E_l(S,P) = I_l(S,P)(k_0 + \sum_n k_n * I_n(S,P)$$

where $I_n(S,P)$ is the number of iterations of the nth nested loop. It is also possible for procedure calls to be nested within a loop. If the execution time of a routine i is R_i, the overall execution time of a loop, including all the nested loops and the procedure calls, is then be expressed as:

$$E_l(S,P) = I_l(S,P)(k_0 + \sum_n k_n * I_n + \sum_i R_i(S,P))$$

where $R_i(S,P)$ is the execution time of the ith procedure call within the loop.

The execution time of a routine or a procedure is the sum of the execution times of all the loops present, as well as the execution times of all other calls made within the scope of the particular routine. The execution time of the routine can then be expressed as:

$$R_i(S,P) = \sum E_j(S,P) + \sum R_k(S,P)$$

where E_j is the execution time of the jth loop and R_k is the execution time of the kth called function within the routine. The overall execution time for any application then becomes the execution time of the main routine.

4.1.2 Parallel Overhead Time

Parallel programs must exchange information throughout program execution. The time to manage and initiate transfers can constitute a significant portion of the execution time for a program. The overhead time depends on the number of messages sent and the amount of information passed.

Communication time consists of two parts, a fixed overhead and a variable overhead portion. The fixed overhead is the time that a process takes to initialize message buffers, set up a network connection with the receiver, etc. The variable overhead is proportional to the size of the message, for example, the time to copy messages buffers and the time it takes to propagate the message to the network. In our models, the constant overhead can be attributed to the basic block in which the communication call is made. Therefore, the only additional term necessary in the model is proportional to the size of the message. These terms are added to the computation time model before dynamic analysis.

4.1.3 Synchronization Time

In a parallel program, processors must synchronize to coordinate their activities. It is not possible to model the synchronization time through static analysis of the program, because synchronization time depends not only on the problem size S and the number processors P, but also on non-deterministic factors like the load on each processor and the contention in the network.

Synchronization in message passing parallel programs is achieved with barriers or synchronous communication calls. In a barrier, all processes block at a particular point during their execution and wait for all other processors to block. Barriers and global reductions, such as summing vectors, are implemented with $\log P$ algorithms. To model this type of behavior, $\log P$ and $P * \log P$ terms are included in the static models.

4.1.4 Static Analysis Tool

A tool to automatically generate the static model described in the previous section was developed. The tool, based on the Sage++ parsing toolkit [17], parses the program to identify all routines, control flow constructs, communication, and synchronization calls. Each loop in the program is identified and the associated execution time model is determined based on the loop bounds. For loops where it is not possible to statically identify the number of iterations (e.g., a loop bound that depends on input data), the user is prompted for the value if the user does not know the number of iterations, a heuristic is used where the loop number of iterations is assumed to match the number of iterations of some other loop in the program. The calculation of overhead times for messages is also done automatically by analyzing the communication calls. If the size of the message can not be determined statically, again a heuristic can be employed or the user can supply the expected size. The tool can determine the execution time of any particular routine. Analysis of the main routine provides a model for the overall execution time.

4.2 Dynamic Analysis

The unknown coefficients $(k_0, k_1, \cdots)$ in the static model are determined by dynamic analysis to produce the final model for execution time in terms of S and P. Consider the equation

$$E_{time} = k_0 + k_1 * (f_1) + k_2 * (f_2) + \cdots + k_n * (f_n)$$

where, f_n's are polynomials of (S, P) and $K = (k_0, k_1, \cdots, k_n)$ are the unknown coefficients. Let this equation be the execution time model for a hypothetical parallel program. This equation has $n + 1$ terms including a constant term. To determine the coefficients for this expression, it is necessary to have at least $n + 1$ equations. Hence, at least $n + 1$ runs of the program must be performed to generate the final expression. The system of equations can be represented in a matrix form as follows:

$$F(S, P) * K_n = E_n$$

where $K_n = (k_0 k_1 k_2 \cdots k_n)^T$ and $E_n = (e_0 e_1 e_2 \cdots e_n)^T$. E_n represents execution times for unique values of (S, P). The coefficient vector K_n can be determined from $F(S, P)$ and E_n using:

$$K_n = F(S, P)^{-1} * E_n.$$

All possible combinations of the terms present in the static model are checked to determine which set of factors produce the best model. The measure for each model is the *coefficient of determination* (R^2) which is defined as the fraction of the variation that is explained by the model[27].

We have developed a dynamic analysis tool that automates the process of model generation. The *Dynamic Analysis Tool* generates models for all possible combinations of the terms using least squares curve fitting. The coefficient of determination for each of the model is calculated and compared. Plots of execution time versus problem size or the number of processors can be generated allowing the user to visually compare the models. Plots of percentage error versus the problem size of the number of processors are also generated.

5 PERFORMANCE MODELING OF THE FDTD PROGRAM

The Finite Difference Time Domain (FDTD) program is implemented in Fortran using the Parallel Virtual Machine (PVM) message passing library [24], however the same analysis would result if another message passing systems was used, e.g., MPT[16]. FDTD was modeled on two platforms typical of current high-performance architectures: an SGI Power Challenge and a cluster of Sun Hyper-Sparc workstations on a 100 Mbps ethernet.

The FDTD program consists of approximately 158 loops (this includes all the initialization loops, field update loops, and loops to support interprocessor communication). The majority of the computation takes place within the triply nested field update loops. However, it was found that the doubly nested loops used for interprocessor communication can be non-negligible. As explained in section 2, the algorithm is based on a spatial decomposition of the regular, orthogonal FDTD lattice of dimensions N_x, N_y and N_z. This lattice was then mapped onto a three dimensional grid of processors (P_x, P_y, P_z) to allocate the work to individual processors. The problem size of the algorithm can be changed by altering the dimensions of the lattice. It is also possible to change the configuration of the processors in the grid and thereby changing the manner in which the workload is distributed. As an example, Figure 2 shows different processor configurations possible when using 8 processors. In this paper, it will be assumed that $N_x = N_y$ and N_z is constant. It will also be assumed that $P_z = 1$. Subsequently, the grid of processors will be two dimensional (i.e., $(P_x, P_y, 1)$). This is typical of microwave circuit analysis, where $N_z << N_x, N_y$ [18]. While the performance analysis tool is not restricted to this, it will greatly simplify the presentation of data.

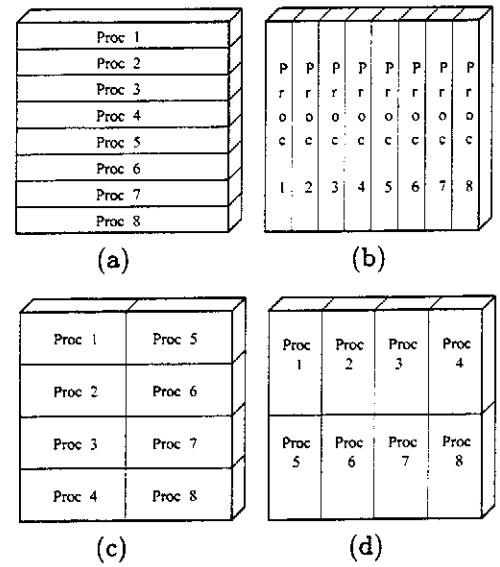

Figure 2: Configurations for Eight Processors; (a) (1,8,1), (b) (8,1,1), (c) (2,4,1), (d) (4,2,1).

The static analysis tool described in section 4.1.4 was first used to model the FDTD algorithm. Under the constraints defined above ($N_x = N_y$ and N_z = a constant), the problem size S was set to N_x. The model was then found to be:

$$E_{time} = k_0 + k_1 * P + k_2 * N_x + k_3 * N_x/P + k_4 * N_x^2/P + k_5 * log(P) + k_6 * P * log(P)$$

where P is the number of processors ($P = P_x P_y P_z$). This model includes terms proportional to the size of the problem $k_2 * N_x$, the number of processors $k_1 * P$. There are also two terms that show the computation time proportional to the problem size divided by the number of processors $k_3 * N_x/P$ and the problem size squared divided by the number of processors $k_4 * (N_x^2/P)$. These terms correspond to the computation being divided among the processors. It is anticipated that the squared term will dominate, since for a fixed N_z, the field update computations will be proportional to $N_x * N_y = N_x^2$.

Each communication in the program is modeled by a constant term and a term proportional to the message size. Because each communication call is made within some basic block, and each basic block is modeled with a constant term k_i, the constant term of the communication call is included in the model of the basic block. To include the term proportional to the message size, the communication call is modeled as a call to a routine with the appropriate execution time. The size of the message is determined and expressed in terms of N_x and P to generate the model for communication. As described in section 4.1.3 the static analysis tool also included the terms $k_5 * \log P$ and $k_6 * P * \log P$ typical of synchronization behavior in parallel programs.

Models in terms of the problem size or the number of processors alone can also be generated by simplifying the overall model. The model in terms of problem size is:

$$E_{time} = k_0 + k_1 * N_x + k_2 * N_x^2$$

and the model based on the number of processors is:

$$E_{time} = k_0 + k_1 * P + k_2 * 1/P + k_3 * log(P) + k_4 * P * log(P).$$

Dynamic analysis experiments were run on both the SGI Power Challenge and the Sun network. On both the platforms PVM 3.3.11 was used for message passing. On the SGI, which is a shared memory machine, the PVM shared memory port was used so that messages were efficiently routed through shared memory. In the cases illustrated, run-time information was collected for different problem sizes varying N_x and N_y from 31 through 131 and the number of processors was varied from 2 to 8. In all cases, $N_z = 21$, and $N_x = N_y$. The static and dynamic models include both the set up time and the explicit field updates. The simulations were run for 1000 time iterations.

While experimenting with 8 processors, models were also generated for all the different possible processor configurations namely (1,8,1), (8,1,1), (2,4,1), and (4,2,1). The one-way dissections (i.e., (1,8,1) or (8,1,1)) require interprocessor communication in one-direction only. Whereas, two-way dissections (2,4,1) or (4,2,1)) require communication in two-directions. This poses a trade off in the sense that the amount of data communicated may be reduced; however, the number of interprocessor communications per processor is increased. The loop dimensions also vary with the geometry of the decomposition which will also have an affect on the code's performance.

Initially, models in terms of problem size were generated by keeping the number of processors constant at 4, 6 and 8. Based on the static analysis, and a dynamic analysis for small problem sizes ($N_x = N_y = 51$ and $N_x = N_y = 61$), models for the SGI Power were derived based on the formulations in Section 5. Both models were based on a simulation with 1000 time iterations. Table 1 shows the models for $P = 4$, 6, and 8.

# of processors	k_0	k_1	k_2
4	6.257	0.000	0.021
6	-1.067	0.000	0.016
8	2.845	0.000	0.011

Table 1: Coefficients for the model $E_{time}(N_x) = k_0 + k_1 * N_x + k_2 * N_x^2$ on the SGI Power Challenge.

This analysis was repeated for the 100 Mbps network of Sun Hyper-Sparcs. Table 2 shows the size models on the network of Hyper-Sparcs for P = 4,6, and 8.

# of processors	k_0	k_1	k_2
4	51.050	0.000	0.113
6	82.515	0.000	0.097
8	47.069	0.000	0.083

Table 2: Coefficients for the model $E_{time}(N_x) = k_0 + k_1 * N_x + k_2 * N_x^2$ on the Sun Hyper-Sparcs.

Figures 3 and 4 show the models on the SGI Power Challenge and the Sun workstations respectively. From the models it can be seen that the execution time varies as square of the problem size. This can be attributed to the fact that the program has a significant amount of computation that is performed within the triply nested loops with loop bounds that depend on N_x, N_y and N_z, where N_z is a constant.

From Figure 3, it can be seen that the predicted times matched closely with the actual execution times. The original models were generated using small values of problem sizes such as $N_x = N_y = 51$ and $N_x = N_y = 61$. The models were then used to compute the anticipated execution time as N_x and N_y were varied from 1 to 200. This is compared to actual execution times recorded for various values of N_x and N_y. The errors in predictions ranged from about 5 to 8 percent. Similarly, from Figure 4, the error in the predictions for the network of Sun

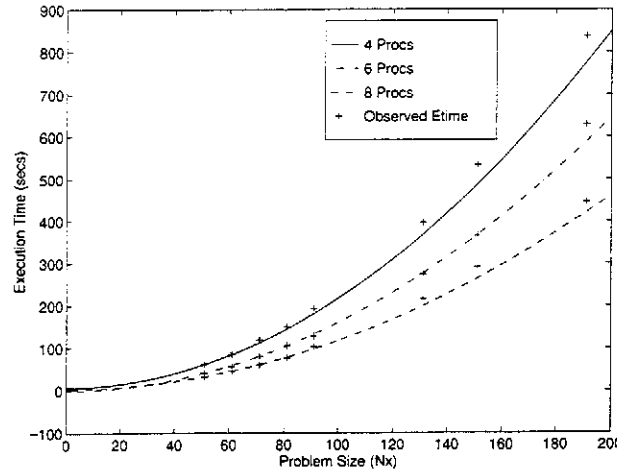

Figure 3: Problem Size Models on SGI Power Challenge, where $N_y = N_x$, $N_z = 21$ (1000 time iterations).

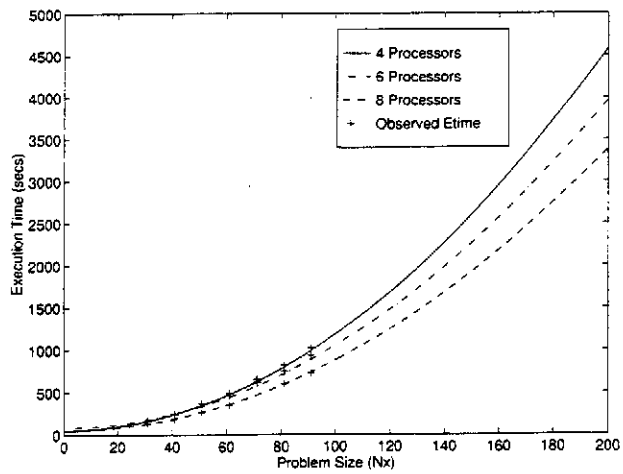

Figure 4: Problem Size Models on Sun Hyper-Sparcs, where $N_y = N_x$, $N_z = 21$ (1000 time iterations).

Hyper-Sparcs was slightly higher than on the SGI and ranged from 6 to 10 percent.

It can also be observed from the models that the performance of the SGI Power Challenge is an order of magnitude greater than the network of workstations. This is because the MIPS R10000 processors in the SGI Power Challenge machine are much faster than the Sun Hyper-Sparcs. Runs with a single processor configuration on each architecture showed that the MIPS R10000 processor to be approximately 8 times faster that the Sun Hyper-Sparc processor.

On the SGI Power Challenge, processors configurations of (1,8,1), (8,1,1), (2,4,1), (4,2,1) were also modeled and compared. Table 3 shows the models for the different configurations. Figure 5 shows the models for

Configuration	k_0	k_1	k_2
(1,8,1)	2.845	0.000	0.011
(8,1,1)	-7.947	1.037	0.013
(2,4,1)	2.918	0.000	0.012
(4,2,1)	-1.290	0.000	0.016

Table 3: Coefficients for the model $E_{time}(N_x) = k_0 + k_1 * N_x + k_2 * N_x^2$ on the SGI Power Challenge.

the four configurations. It can be seen that as the problem size increases, the (8,1,1) and (4,2,1) configurations perform significantly worse than the (1,8,1) and (2,4,1) configurations.

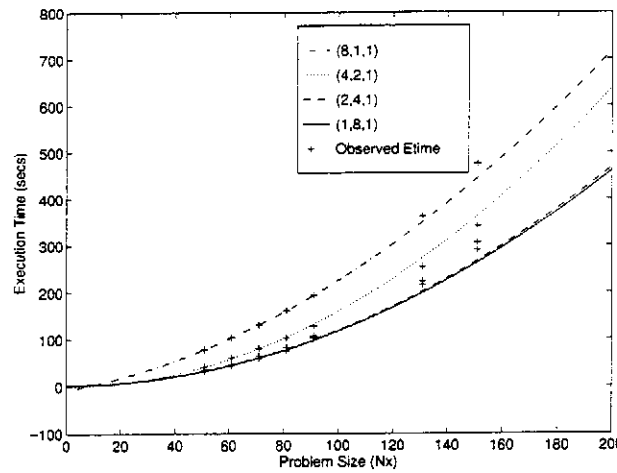

Figure 5: Models for different processor configurations on the SGI Power Challenge, where $N_y = N_x$, $N_z = 21$ (1000 time iterations).

To quantify this behavior additional models were created. Models for the message size and the number of messages showed that the number and size of messages being passed were the same for both the (8,1,1) and (1,8,1) configurations. Thus, the problem was not com-

munication overhead. The next hypothesis was that there was a problem in the computation phase of the program. Because the amount of work done in both the configurations should be equal, interaction with the memory system was suspected. The number of page faults generated by each configuration was measured and is shown in Figure 6 versus the problem size for the (1,8,1) and (8,1,1) configurations. The figure shows that the number of page faults for the (8,1,1) configuration is much higher. The likely cause for page fault problems is irregular accesses to memory.

Further examination of the code revealed that the cause was a set of loops in the interprocessor communication subroutine that accessed a matrix row-wise fashion. In Fortran, row-wise accesses cause non-unit strides through memory that can increase the number of pages touched and, as a result, the number of page faults. The code was modified by changing the loop structure so that one instance of the non-unit stride in the program was removed. Figure 7 shows the performance models for the original version of the (1,8,1) and (8,1,1) configurations and the optimized version of the (8,1,1) configuration. The figure shows that there was a significant improvement in the performance of the modified version when compared to the original program. Although the number of operations performed in the double-nested loops was insignificant compared to the three-dimensional field update loops, it had a dramatic affect on the overall program performance. This illustrates how performance models can be used to uncover unexpected behavior in applications.

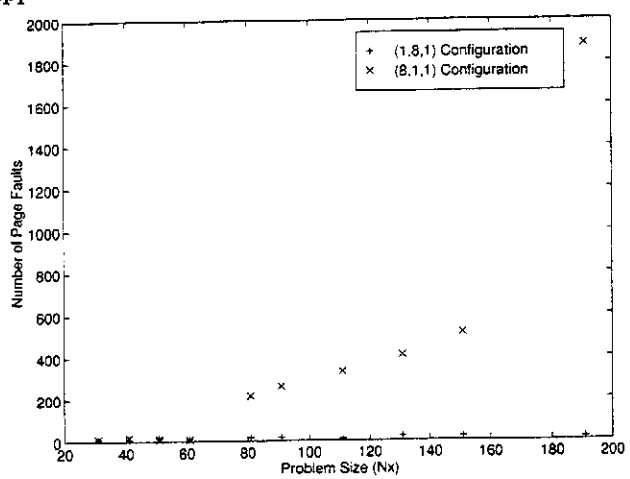

Figure 6: Number of page faults on the SGI Power Challenge for two configurations.

Models in terms of the number of processors were also generated on both platforms. These models were created by changing the number of processors while keeping the problem size fixed. Table 4 shows the different P models for $N_x, N_y = 31$, $N_x, N_y = 81$, and $N_x, N_y = 151$.

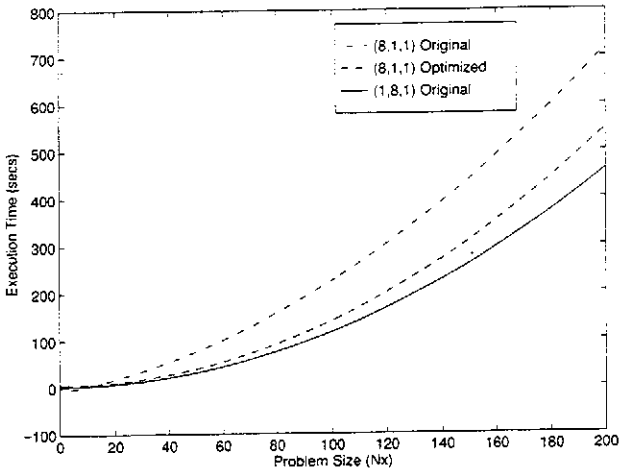

Figure 7: Comparison of original program to the modified version.

Size	k_0	k_1	k_2	k_3	k_4
31	73.081	8.951	0.000	63.704	0.000
81	-106.964	0.000	780.309	44.899	0.000
151	2.945	0.000	2122.639	0.000	0.000

Table 4: Coefficients for the model $E_{time}(P) = k_0 + k_1 * P + k_2 * 1/P + k_3 * \log P + k_4 * P * \log P$ on the SGI Power Challenge.

The different models are shown in Figure 8. From the figure it can be seen that as the number of processors increase the performance, in terms of execution time, improves. As the number of processors increase there is a decrease in the performance gain. This is expected since as the number of processors increase the parallel overhead also increases and this affects the performance. From the figure it can be seen that for problem size of $N_x = N_y = 31$ and $N_z = 21$, the execution time drops gradually until 9 processors and as more processors are increased the execution time increases. This is because at this point the processors are spending more time exchanging information than performing the computation. The optimum number of processors can be determined from the models.

Figure 9 shows the models for the execution time in terms of the number of processors on the network of Sun Hyper-Sparcs. The models were generated for $N_x = N_y = 31$, $N_z = 21$ and $N_x = N_y = 91$, $N_z = 21$. Similar to the models on the SGI Power Challenge, the performance of the program improves sharply at first and then gradually reduces as the number of processors are increased. The coefficients for the models are shown in Table 5.

Performance models for the FDTD application in terms of both the problem size and the number of processors were also generated. The model for the execution

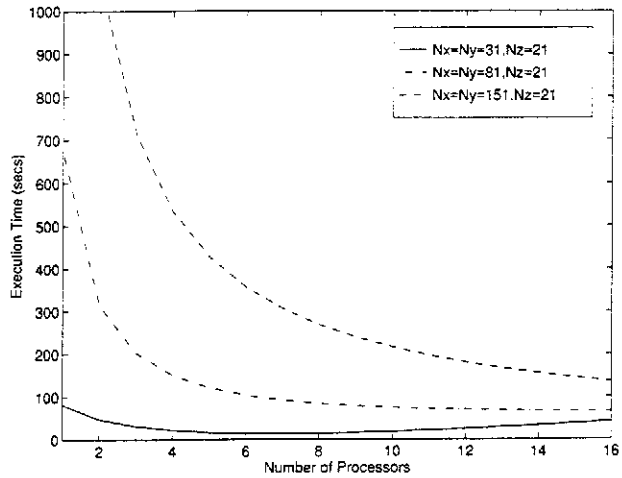

Figure 8: Processor Models on the SGI Power Challenge (1000 time iterations).

Size	k_0	k_1	k_2	k_3	k_4
31	101.144	0.000	503.707	0.000	0.000
61	195.574	0.000	1478.746	0.000	0.000
91	314.110	0.798	4268.800	0.000	0.000

Table 5: Coefficients for the model $E_{time}(P) = k_0 + k_1 * P + k_2 * 1/P + k_3 * \log P + k_4 * P * \log P$ on the Network of Sun Hyper-Sparcs.

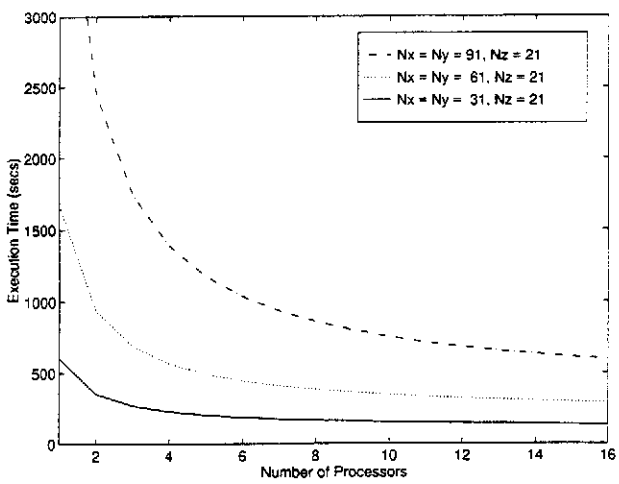

Figure 9: Processor Models on the Network of Sun Hyper-Sparcs (1000 time iterations).

time in terms of the number of processors P and the problem size N_x is:

$$E_{time} = 5.538 - 0.227 * N_x + 1.366 * P$$
$$-1.639 * (N_x/P) + 0.119 * (N_x^2/P)$$

The average error in prediction for this model was about 10 percent. Figure 10 shows model for the program as both a function of problem size and number of processors.

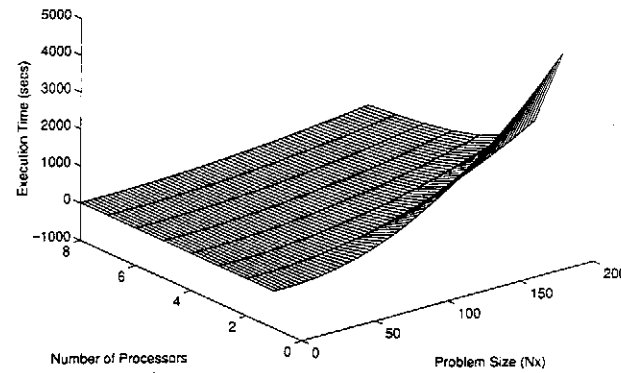

Figure 10: The Problem Size and Processor Model on the SGI Power Challenge, where $N_y = N_x$, $N_z = 21$ (1000 time iterations).

6 SUMMARY

An approach to modeling the performance of parallel programs was presented and applied to modeling the FDTD algorithm. The modeling technique is based on analyzing the structure of an existing program and measuring the performance of the program during actual runs on the target architecture as key factors of the application and architecture are varied. This information is used to create a closed-form expression for the execution time of the program in terms of those factors. The expression can then be used to predict the performance of the application over a wide range of problem size and number of processors.

The models showed that the FDTD application performed well when there was a sufficiently large problem size resulting in sufficient parallelism. The models showed that the execution time varied as a square of the problem size. This is due to the fact that the majority of the work was done within a triply nested loop, with the outer loop dimension (N_z) held constant. The models also showed a degradation in the performance in one of the configurations as the problem size increased. The reason for this was found to be non-unit stride accesses to memory. This was corrected by interchanging the

rows and columns. This change resulted in a 30 percent performance improvement for $N_x = 131$.

7 REFERENCES

[1] V. D. Agrawal and S. T. Chakradhar. Performance estimation in a massively parallel system. In *Proceedings of Supercomputing' 90*, pages 306–313, 1990.

[2] J. P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114:185–200, October 1994.

[3] R. J. Block, P. Mehra, and S. Sarrukai. Automated performance prediction of message-passing parallel programs. In *Proceedings of Supercomputing' 95*, 1995.

[4] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[5] R. Calalo, T. Cwik, N. Jacobi W. Imbriale, P. Liewer, T. Lockhart, G. Lyzenga, and J. Patterson. Hypercube parallel archtecture applied to electromagnetic scattering analysis. *IEEE Transactions on Magnetics*, 25:2898–2900, July 1989.

[6] W. C. Chew and W. H. Weedon. A 3d perfectly mached medium from modified maxwell's equations with strecthed coordinates. *IEEE Microwave and Guided Wave Letters*, 7:599–604, September 1994.

[7] M. J. Clement, M. R. Steed, and P. E. Crandall. Network performance modeling for pvm clusters. In *Proceedings of Supercomputing' 96*, 1996.

[8] R. Covington, S. Dwarakadas, J. Jump, S. Madala, and J. Sinclair. Efficient simulation of parallel computer systems. *International Journal on Computer Simulation*, 1(1):31–58, June 1991.

[9] M. E. Crovella and T. J. Leblanc. Parallel performance prediction using the lost cycles analysis. In *Proceedings of Supercomputing' 94*, pages 600–610, 1994.

[10] D. Culler, R. Karp, and D. Patterson. Logp: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Conference on Parallel Programming Practice and Experience*, pages 1–12, 1993.

[11] D. B. Davidson and R. W. Ziolkowski. A connection machine(cm-2) implementation of a three-dimensional parallel finite-difference time-domain code for electromagnetic field simulation. *International Journal of Numerical Modelling*, 8:221–232, August 1995.

[12] P. M. Dickens, P. Heidelberger, and D. M. Nicol. Parallelized direct execution simulation of message-passing parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1090–1105, October 1996.

[13] M. A. Driscoll and W. R. Daasch. Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25:16–30, 1995.

[14] R. Kroger F. Lange and M. Gergeleit. Jewel: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems 3*, pages 657–671, November 1992.

[15] R. D. Ferraro. Solving partial differential equations for electromagnetic scattering problems on coarse-grained concurrent computers. In T. Cwik and J. Patterson, editors, *Computational Electromagnetics and Supercomputer Architecture*, number 7, pages 111–154. EMW Publishing, Cambridge, MA, July 1993.

[16] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department CS-94-230, University of Tennessee, Knoxville, TN, 1994.

[17] D. Gannon, F. Bodin, S. Srinivas, N. Sunderasan, and S. Narayana. Sage++, an object oriented toolkit for program transformations. Technical report, Department of Computer Science, Indiana University, 1993.

[18] S. D. Gedney. Finite-difference time-domain analysis of microwave circuit devices on high performance vector/parallel computers. *IEEE Transactions on Microwave Theory and Techniques*, 43:2510–2514, October 1995.

[19] S. D. Gedney. An anisotropic perfectly matched layer absorbing media for the truncation of fdtd lattices. *IEEE Transactions on Antennas and Propogation*, 44:1630–1639, December 1996.

[20] S. D. Gedney. An anisotropic pml absorbing media for fdtd simulation of fields in lossy dispersive media. *Electromagnetics*, 16:399–415, July/August 1996.