Parallel FFTCAP: A Parallel Precorrected FFT Based Capacitance Extraction Program for Signal Integrity Analysis

by

Vivek Bhalchandra Nadkarni

S. B., Massachusetts Institute of Technology (1997)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering
in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

©1998 Massachusetts Institute of Technology. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 20, 1998

Certified by ....
Jacob K. White
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 14 1998

LIBRARIES

Eng

Eng

Parallel FFTCAP: A Parallel Precorrected FFT Based Capacitance Extraction Program for Signal Integrity Analysis

by

Vivek Bhalchandra Nadkarni

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1998, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Estimation of coupling capacitances in complicated three-dimensional integrated circuit interconnect structures is essential to ensure signal integrity in high performance applications. Fast algorithms such as the multipole based FASTCAP and precorrected FFT based FFTCAP have been recently developed to compute these coupling capacitances rapidly and accurately. This thesis shows that the efficacy of FFTCAP can be greatly improved by modifying it to work with a cluster-of-workstations based parallel computer such as the IBM SP2. The issues in parallelizing FFTCAP to balance the computational time and memory usage across the processors, while minimizing interprocessor communication, are examined. Computational results from a parallel implementation of FFTCAP running on an eight processor IBM SP2 are presented, showing a nearly linear parallel speedup for several large examples. The results also show that Parallel FFTCAP can be used on a multiprocessor system to solve significantly larger problems than can be solved on a single processor.

Thesis Supervisor: Jacob K. White
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I thank Professor Jacob White for his guidance and encouragement through the course of my research with him. I thank him especially for taking a chance on me early on in my academic career at MIT, by letting me work with him since the end of my sophomore year. I further thank him for giving me several opportunities to present my work with him in public.

I would like to thank Narayan Aluru who offered me technical guidance throughout this project, and also contributed significantly to several parts of this work. Narayan generously offered to read and give me feedback on this thesis. On a more personal level, Narayan has continued to be a mentor to me, throughout the time I have known him.

I must also thank Tom Korsemeyer, who has always been willing to bounce ideas back and forth, at times for hours on end, until they crystallized into tangible algorithms. Tom also was kind enough to read this thesis and provide valuable detailed feedback on this thesis.

Joel Phillips provided me with a copy of FFTCAP, and explained its nuances to me so that I could get this project off the ground. He also provided me with written descriptions of FFTCAP as well as some figures that I have used in this thesis. George Hadjiyiannis, despite his firm belief that every computer I touched turned to stone, helped me debug the most persistent bugs in my code. Matt Kamon helped me in my struggle with LaTeX, and also gave me his insights into capacitance extraction. I thank them and the other members of the 8th floor of RLE for their informative discussions, their support and their friendship.

I must thank my two apartment-mates Mark Asdoorian and William Lentz, who were always present and willing to listen to me talk incessantly about my thesis project. Many of my ideas were clarified by them through discussions that started around bedtime and continued into the wee hours of the morning. They have both been absolutely wonderful apartment-mates and friends.

Last, but by no means least, I thank my parents for cultivating in me a respect for education. It is only because of their unwavering and earnest encouragement that I made it all the way to and through MIT.

The work needed to produce this thesis was supported in part by DARPA and NSF, and through an NSF fellowship. The IBM SP2 parallel computer on which the algorithm was implemented was provided by IBM.

Contents

Abstract 3

Acknowledgements 5

List of Figures 9

List of Tables 11

1 Introduction 13

2 Capacitance Extraction 15

2.1 Problem Formulation 15

2.2 Solution Using the Precorrected FFT Method 16

2.2.1 Projection of Panel Charges onto the Grid 17

2.2.2 3-D Convolution: Grid Potentials from Grid Charges 18

2.2.3 Interpolation of Grid Potentials onto the Panels 19

2.2.4 Precorrection: Computing Local Interactions Directly 19

2.2.5 Choosing a Grid 20

3 Problem Decomposition across Processors 23

3.1 Approaches to Problem Decomposition 23

3.2 Allocating Grid Points to Processors 24

3.3 Inter-Gridpoint Spacing Selection through Grid Scaling 25

4 **Parallel Implementation of the Precorrected FFT Algorithm** **29**
4.1 Projection of Panel Charges onto a Grid 29
4.2 3-D Convolution in Parallel 30
4.2.1 Rationale for a Custom Parallel 3-D Real DFT 30
4.2.2 Implementation of the Custom Convolution Algorithm 31
4.2.3 Real DFT Extraction and Packing 34
4.2.4 Transpose Operation 36
4.3 Interpolation of Grid Potentials onto Panels 41
4.4 Precorrected Direct Interactions 41

5 **Computational Results** **43**
5.1 Parallel Performance 44
5.2 Effects of Grid Scaling 47
5.3 Solving Large Problems 51

6 **Conclusion** **55**

Bibliography **58**

List of Figures

2-1 2-D Pictorial representation of the precorrected FFT algorithm 18

2-2 Grid Tradeoffs : (a) Fine Grid (b) Coarse Grid 21

3-1 2-D Pictorial representation of the decomposition algorithm 25

3-2 2-D Representation of scaling the distance between grid points 27

4-1 Definition of the cube dimensions 32

4-2 Primary block transpose operation 38

4-3 Movement of z-lines in the primary block transpose 38

4-4 Secondary block transpose operation 39

4-5 Movement of z-lines in the secondary block transpose 39

5-1 Cubic capacitor discretized to 60,000 panels 45

5-2 Parallel scaling of costs 46

5-3 Performance gains through grid scaling 48

5-4 Adaptive grid scaling performs better than forced grid scaling 49

5-5 Adaptive grid scaling trades off speed for memory performance 50

List of Tables

5.1 Large problems solved using Parallel FFTCAP 52

1

Introduction

It is extremely difficult to find signal integrity problems in high performance integrated circuits because the problems are caused by the detailed interactions between hundreds of conductors in the integrated circuit. Simulating these three-dimensional interactions on a conventional scientific workstation is a slow process even when one uses the fastest simulation tools available. To allow such a detailed analysis to be used in optimization instead of just an a posteriori verification step, it is important to reduce the turn-around time for the analysis.

We demonstrate that this reduction in turn-around time for a fast 3-D capacitance extraction program can be achieved by using a cluster-of-workstations based parallel computer such as the IBM SP2. In Section 2 we will present a recently developed fast method for 3-D capacitance extraction, the precorrected FFT accelerated method [6, 5]. In Sections 3 and 4 we describe the methods that we used to parallelize this capacitance extraction algorithm [1]. In Section 5 we give computational results demonstrating the parallel scaling of this new parallel algorithm. We also provide an analysis of the behavior of the parallel efficiency of this algorithm when simulating different integrated circuit geometries, and give examples of large problems that have been solved using Parallel FFTCAP.

2

Capacitance Extraction

2.1 Problem Formulation

Our capacitance extraction program calculates the $m \times m$ capacitance matrix C, which summarizes the capacitive interactions in an m conductor geometry. The jth column of the matrix C is the total surface charge induced on each conductor when the potential of the jth conductor is raised to 1 Volt while the remaining conductors are grounded. This charge on each conductor can be calculated by solving the integral equation [8]

$$\psi(x) = \int_{surfaces} \sigma(x') \frac{1}{4\pi\epsilon_0 \|x - x'\|} da', \quad x \in surfaces, \tag{2.1}$$

where $\psi(x)$ is the known conductor surface potential, which is either 0 or 1 Volt, σ is the surface charge density, which is to be determined, da' is the incremental conductor surface area, x, $x' \in \mathrm{R}^3$, and $\|x\|$ is the Euclidean length of x given by $\sqrt{x_1^2 + x_2^2 + x_3^2}$.

The standard method to solve for σ is to break up the surfaces of the conductors into n small triangles or planar quadrilaterals called panels. The surface charge density on each small panel is assumed to be constant, so that each panel i carries a uniformly distributed charge q_i. The known potential at the center of panel i is $\overline{p}_i$. There is an equation that equates the potential $\overline{p}_i$ to the sum of the contributions to

that potential from the charge distributions on each of the n panels. Thus, a dense linear system

$$Pq = \overline{p} \tag{2.2}$$

is obtained, where q is the vector of panel charges, $\overline{p} \in \mathbf{R}^n$ is the vector of known panel potentials, and $P \in \mathbf{R}^{n \times n}$ is the matrix of potential coefficients, where

$$P_{ij} = \frac{1}{a_j} \int_{panel_j} \frac{1}{4\pi\epsilon_0 \|x_i - x'\|} \, da', \tag{2.3}$$

in which x_i is the center of the ith panel and a_j is the area of the jth panel.

This problem is solved to yield the charge vector q corresponding to the case in which the panels of the jth conductor are raised to 1 Volt while the panels corresponding to the other conductors are grounded. By adding up the charges induced on the panels belonging to each conductor we obtain the total charge induced on each conductor. This new vector of total charge induced on each conductor corresponds to the jth column of the capacitance matrix that we are trying to calculate.

2.2 Solution Using the Precorrected FFT Method

The costs of solving the linear system (2.2) are quantified using two metrics. The first metric is the computational time measured in the number of operations required to solve the equation. The second metric is the amount of memory, measured in megabytes, that the algorithm allocates to solve the equation. $O(n^2)$ memory and $O(n^2)$ operations are required to form the dense matrix P. The computational time of solving (2.2) using Gaussian elimination is $O(n^3)$ operations. By using an iterative algorithm like GMRES[9] to solve this system of equations, the computational time is reduced to $O(n^2)$ operations, corresponding to the cost of computing a dense matrix vector product Pq for each GMRES iteration. The $O(n^2)$ time and memory costs of explicitly forming P and multiplying by it to solve (2.2) can be avoided, and the costs of solving the equation can be reduced to $O(n)$ in memory and $O(n)$ or $O(n \log n)$ in number of operations by using matrix sparsification techniques such as fast multipole

algorithms described in [3] and [4] or precorrected FFT methods described in [6]. The total costs of formulating and solving (2.2) can be reduced to $O(n)$ in memory and $O(n \log n)$ in number of operations using the precorrected FFT method to compute this matrix vector product.

The precorrected FFT method is explained here, following the development of the algorithm in [6]. In this algorithm, once the three-dimensional conductor geometry has been discretized into panels, a three-dimensional grid containing $j \times k \times l$ cubes is superimposed onto the geometry, so that each cube contains a small number of panels. The interactions of panels that are near each other, that is, in the same or neighboring cubes, are found by computing the corresponding portions of the product Pq directly. The distant panels interactions are approximated by representing them as the interactions between the grid points of the cubes within which the panels lie. The entire matrix vector product Pq can be approximated accurately in this manner. Specifically, Pq may be approximated in $O(n \log n)$ operations in four steps:

1. Project the panel charges onto a uniform grid of point charges.
2. Compute the grid potentials due to grid charges using an FFT for the convolution.
3. Interpolate the grid potentials onto the panels.
4. Directly compute local interactions.

This four-step process is summarized in Figure 2-1. In this process, the calculation of the grid potentials due to the grid charges is used to approximate the calculation of the panel potentials due to the panel charges. Section 2.2.4 describes the computation of the local interactions between panels that are close to each other, shown by the shaded gray region in the figure.

2.2.1 Projection of Panel Charges onto the Grid

Projecting of panel charges onto the grid means that the charges that are assigned to the grid points should induce the same potential at any distant point as the panel charges. To project the panel charges, test points called collocation points

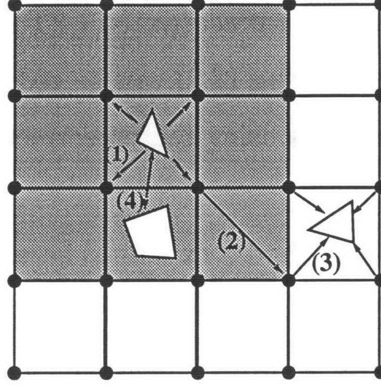


Figure 2-1: 2-D Pictorial representation of the precorrected FFT algorithm. Interactions with nearby panels (grey area) are computed directly, interactions between distant panels are computed using the grid. Figure obtained from [5]

are selected outside the cube in which the panel lies. The grid charges are chosen so that the potential at these test points due to the grid points matches the potential at these points due to the panel charge distribution. Since such collocation equations are linear in the charge distribution, this projection operation which generates a subset of the grid charges, denoted q_a^g, can be represented as a matrix, W_a, operating on a vector representing the panel charges in cube a, q_a, giving rise to the equation

$$q_a^g = W_a q_a \tag{2.4}$$

2.2.2 3-D Convolution: Grid Potentials from Grid Charges

Once the charge has been projected to a grid, computing the potentials at the grid points due to the grid charges is a three-dimensional convolution. This is described by the expression

$$\psi_g(i,j,k) = \sum_{i',j',k'} h(i-i',j-j',k-k')q_g(i',j',k'), \tag{2.5}$$

where i,j,k and i',j',k' are triplets specifying the grid points, ψ_g is the vector of grid potentials, q_g is the vector of grid charges, and $h(i-i',j-j',k-k')$ is the inverse distance between grid points i,j,k and i',j',k'. This convolution can be computed in $O(N \log N)$ time, where N is the number of grid charges, by using the FFT.

2.2.3 Interpolation of Grid Potentials onto the Panels

Once the grid potentials have been found they can be interpolated onto the panels in each cube. This is the dual operation of projecting the panel charges onto the grid points. Consider V_a to be the operator projecting a point charge at the centroid of a panel onto the cube. Then the transpose of the projection operator V_a^T is the interpolation operator, which interpolates the grid potentials to give the panel potentials [2, 5]. Note that this is not the same as the transpose of W_a for cube a. This is because W_a projects the panel charge distribution whereas V_a projects a point charge. The three steps, projection, followed by convolution, followed by interpolation, can be represented as

$$\psi_{fft} = V^T H W q, \tag{2.6}$$

where q is the vector of panel charges, ψ_{fft} is an approximation to the panel potentials, W is the concatenation of the W_a's for each cube, V is the concatenation of the V_a's for each cube and H is the matrix representing the convolution in (2.5).

2.2.4 Precorrection: Computing Local Interactions Directly

In ψ_{fft} of (2.6), the portions of Pq associated with neighboring cube interactions have already been computed, though this close interaction has been poorly approximated in the projection/interpolation. To accurately model these interactions we need to subtract out the effect of these poorly approximated nearby interactions from the product Pq, representing the panel potentials, and add in the contribution to the panel potentials due to the true interactions. The interactions of panels near each other are calculated by explicitly evaluating (2.3) numerically.

Consider two panels a and b which belong to neighboring cubes. Denoting $P_{a,b}$ as the portion of P associated with the interaction between neighboring cubes a and b, $H_{a,b}$ as the potential at grid points in cube a due to grid charges in cube b, ψ_a and q_b as the panel potentials and charges in cubes a and b respectively, a better

approximation to ψ_a is

$$\psi_a = \psi_{a_{fft}} + \left(P_{a,b} - V_a^T H_{a,b} W_b\right) q_b \tag{2.7}$$

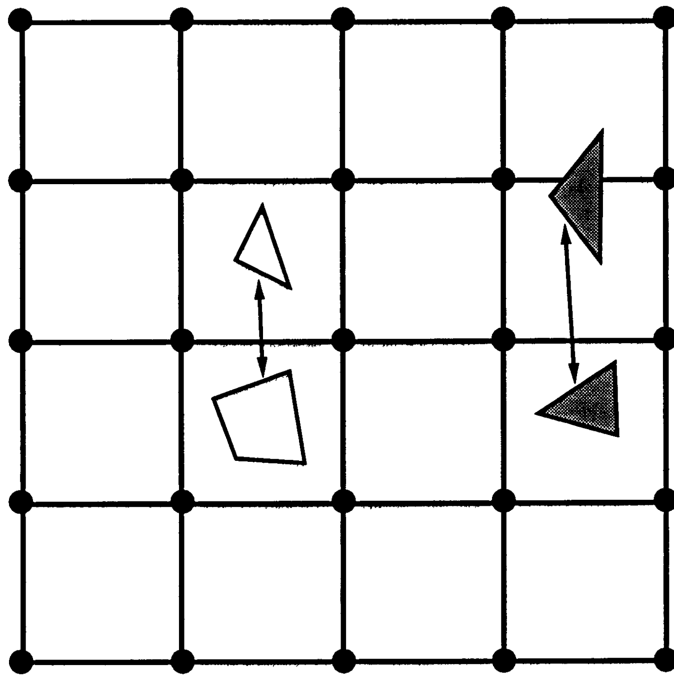
where $P_{a,b}^{cor} \equiv P_{a,b} - V_a^T H_{a,b} W_b$ is the precorrected direct interaction operator.When used in conjunction with the grid charge representation $P_{a,b}^{cor}$ results in exact calculation of the interactions of nearby panels.

2.2.5 Choosing a Grid

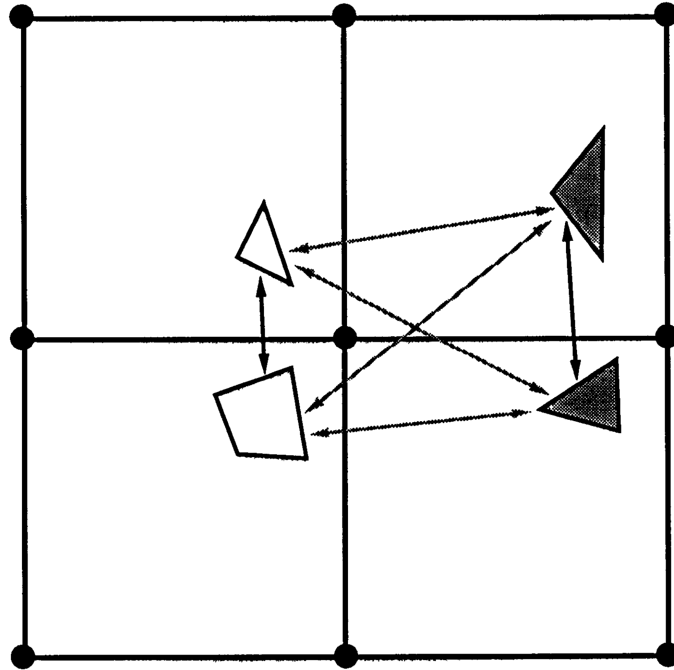
In this algorithm, a uniform grid of point charges is superimposed on the problem domain so that the long range interactions of the panels can be approximated using the grid to grid interactions. There are some tradeoffs associated with choosing this grid that need to be considered before parallelizing the algorithm. Some of these concerns are tied to the load balancing of the algorithm across processors, which becomes important in the parallel case, but is not relevant in the single processor case. The dominant concern in choosing a grid in both the parallel and the single processor algorithms is to determine the inter-gridpoint spacing, that is to determine how fine the grid should be.

Consider Figure 2-2(a), which shows a fine grid superimposed on the panels of the structure being simulated. The panels lying in adjacent cubes of the grid are shaded with the same color. The interactions between panels of the same color are shown in the figure by arrows. These interactions need to be computed directly because the panels lie very close to each other, and their interactions cannot be accurately approximated by the grid interactions. The interactions between the gray and white panels, however, are adequately represented by the grid approximation because the gray and white panels do not lie in adjacent cubes. The fineness of the grid ensures that there are not very many local panel interactions that need to be computed directly, but the large number of grid points in the fine grid implies that the cost of Fourier Transforming the grid is relatively high in terms of processor time.

The alternative is to have a coarse grid superimposed on the panel structure



(a)



(b)

Figure 2-2: Grid Tradeoffs : (a) Fine Grid (b) Coarse Grid

as shown in Figure 2-2(b). Now the cost of the FFT is reduced as the number of gridpoints has decreased. In this coarse grid, however, the gray and white panels lie in adjacent cubes. Now each gray panel interacts with each white panel, and these interactions have to be computed directly. This is in addition to the interactions between the pairs of panels of the same color as in the fine grid case. This increases the cost of the local interactions, both in terms of processor time and memory that needs to be allocated to compute and store the additional panel interactions. The more important cost of the direct interactions is the cost of memory.

The optimal grid minimizes the total cost measured as either the sum of the local interaction time and the FFT time, or sum of the local interaction memory and the FFT memory. The measure that we choose to minimize is the expected time-memory product, since the dominant cost in the FFT is the time, and the dominant cost in computing local interactions is memory. It has been shown in [5] that optimizing on expected time or expected memory usually yield the same size grid, and when they differ, the time-memory product for both grids is very similar.

FFTCAP adaptively chooses the optimal grid depth to minimize the time-memory product. An additional factor is used to choose the best inter-grid spacing in Parallel FFTCAP. This additional element of the adaptive algorithm to choose the best inter-grid spacing is described in Section 3.3, along with its implications for parallel performance of the algorithm.

3

Problem Decomposition across Processors

Parallelizing FFTCAP involves efficiently parallelizing the application of the matrix H to the vector Wq in (2.6). This matrix vector product is the matrix representation of the convolution, which is the most expensive step measured in computational time. In terms of memory usage, the most expensive step is to generate and store the precorrection terms in (2.7). We have to decompose the problem across processors so that we balance the estimated memory usage and processor usage, and at the same time minimize interprocessor communication. This section describes the possible approaches to this decomposition, and the approach that we chose. Chapter 4 describes the implementation of a parallel algorithm for approximating the matrix vector product in (2.7) based on the chosen problem decomposition.

3.1 Approaches to Problem Decomposition

A problem decomposition which balances memory and processor usage, and at the same time minimizes interprocessor communication, is difficult to find. From a functional standpoint, we need to effectively parallelize each of the steps mentioned in Section 2.2. Balancing the direct computation implies balancing the number of nearby interactions, balancing the projection/interpolation implies associating the

same number of panels with each processor, and balancing the grid convolution implies associating the same number of grid points to each processor.

One approach to resolving this difficulty is to consider separate decomposition algorithms for each part of the precorrected-FFT algorithm, but the advantage of the better load balancing might be lost due to additional communication costs associated with realigning the problem decomposition. In this algorithm, we take the approach of picking a single decomposition which best fits the convolution algorithm, that of balancing the number of grid points per processor. We make this choice because the convolution is the most expensive step in terms of computational time, and also because the time taken by the convolution exhibits $O(n \log n)$ growth while the time and memory required by all of the other steps exhibit $O(n)$ growth. If the problem is sufficiently homogeneous, this also results in a relatively well balanced memory usage. Knowing this decomposition method, we can rescale the inter-grid point spacing to load balance the direct interactions spatially across the processors. In the algorithm, this rescaling needs to be done before the decomposition that best distributes the convolution can be chosen.

3.2 Allocating Grid Points to Processors

The decomposition algorithm simply allocates an equal number of planes of grid points, which we refer to as grid planes, to each processor. The partitioning is performed along the z direction or the third FFT dimension and the number of planes allocated to each processor is computed as

$$\text{number of planes} = \frac{\text{number of } z\text{-direction grid points}}{nproc} \tag{3.1}$$

where *nproc* is the number of processors and the number of z-direction grid points is a power of two.

Figure 3-1 illustrates a two-dimensional example with 3 squares and 8 lines of grid points for convolution. Each square contains 9 grid points and a two-way

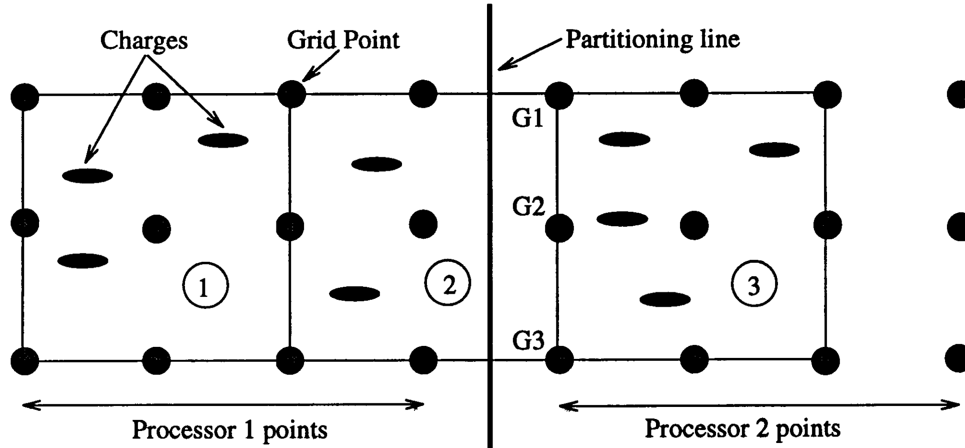


Figure 3-1: 2-D Pictorial representation of the decomposition algorithm

partitioning of the problem puts 4 lines of points per processor, splitting the second square (or cube in 3-D) into two parts. The charges in cube 1 and cube 2 are associated with processor 1 and the charges in cube 3 are associated with processor 2. The first line (plane in 3-D) of points in each processor (except the first processor) is shared by cubes belonging to processors i and $i + 1$. The communication associated with this sharing is described in the Section 4.1 on projection.

3.3 Inter-Gridpoint Spacing Selection through Grid Scaling

A relatively homogeneous problem may be distributed unevenly across processors using the decomposition described in Section 3.2, if the distance between adjacent grid points is not chosen carefully. In the single processor FFTCAP code, the grid spacing is chosen by first deciding the grid depth d, such that the number of grid points along the longest dimension of the input structure is 2^d. These 2^d grid points are then spaced equally along this dimension of the structure. The number of grid points in the other dimensions is the smallest power of two that contains the input structure in each of those dimensions. In the serial code this gives rise to the most compact grid for each grid depth. An adaptive algorithm then chooses the optimal grid depth, which is the depth of the grid with the lowest expected cost time-memory

product as described in Section 2.2.5.

In the parallel code, the input structure is distributed across the processors along the z dimension. If the z dimension is not the longest dimension, then the grid along the z dimension may look like Figure 3-2 (a). The extent of the panels of charge is just a bit further than will fit in a smaller power of two, which in the figure is 8 grid points. Therefore, a larger power of two has to be chosen for the grid, shown as 16 grid points in the figure, in which a large part of the grid is empty. This is a problem in the parallel code, because the grid points are equally distributed across processors, but the panels may not be equally distributed amongst the grid points, as shown in the figure. Processor 2 gets fewer panels than processor 1 in this case, and the direct interaction calculations are badly load balanced. Note that this problem can be completely avoided if it is ensured in the input structure that the z dimension is the largest dimension, or if the input file can be rotated as a pre-processing step to make the z dimension the longest dimension. We do not wish to put such restrictions on the allowable input structures, however, and try to work around this problem as described below.

This problem can be caught during the problem decomposition, by checking to see if the non-empty cubes in processor 2 all lie close to processor 1. It can be corrected, by increasing the inter-grid spacing as shown in Figure 3-2(b) so that the smaller power of two grid points spans the entire z dimensional extent of the input structure. When the problem is divided across processors after the rescaling, the panels are more evenly distributed across the processors, leading to more load balanced direct interaction calculations.

Grid scaling gives the benefits of reducing the size of the FFT grid by a factor of two and load balancing the direct interaction calculations across processors, while it increases the total computation and memory required for the direct interaction calculations. If the increase in the number of direct interaction computations is large, then the increase in computation cost and memory cost incurred can overshadow the two benefits derived by rescaling the grid. In the parallel algorithm, we choose the grid spacing which we estimate will have a lower total cost, based on the magnitude

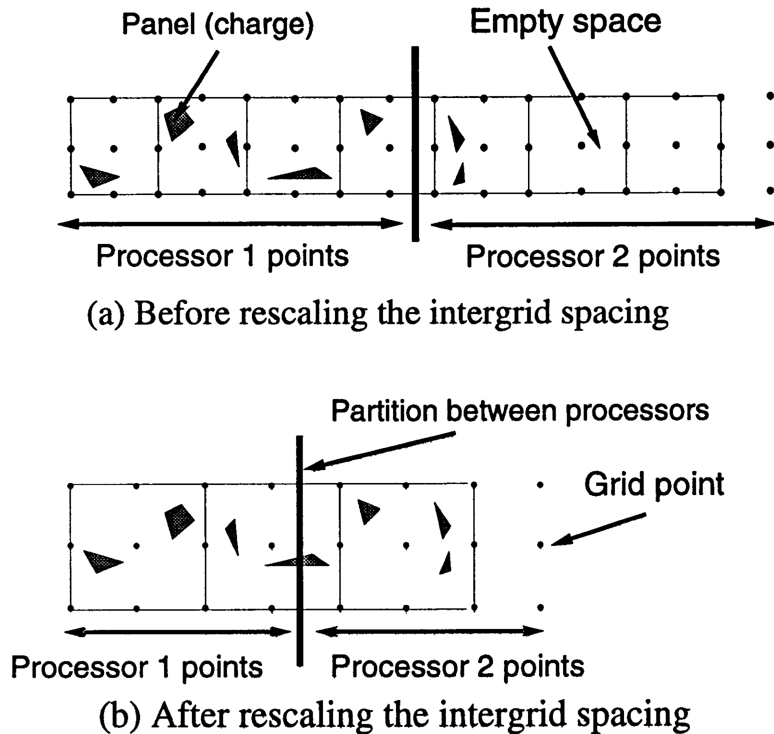


Figure 3-2: 2-D Representation of scaling the distance between grid points

of the grid-size scaling factor required to exactly fit the problem in a power of 2 grid points. This adaptively chosen inter-grid spacing for each grid depth is then fed back into the adaptive algorithm for determining the optimal grid depth, as described in Section 2.2.5. A high level description of the complete grid selection algorithm follows:

1. Set grid depth d to minimum allowed grid depth $- 1$
2. Set *maxlength* to max(length,width,height) of structure
3. **While** time, memory and time-memory product costs at depth $d-1$ are not all less than those at depth d
   (a) Increment d
   (b) Superimpose a cubic grid of length *maxlength* and 2^d gridpoints per side on structure.
   (c) Compute *zscale* the factor by which the inter-grid spacing would have to be scaled, for the z dimension of the structure to exactly fit inside a power of 2 gridpoints.

(d) If *zscale* is smaller than the ratio by which the original z dimension of the grid extends beyond the structure, scale the grid.

(e) Estimate the time, memory and time-memory product that will be required by the algorithm to compute the capacitance matrix.

4. Choose the grid with minimum time-memory product cost.

This grid can now be used for the precorrected FFT based capacitance extraction.

4

Parallel Implementation of the Precorrected FFT Algorithm

To implement the precorrected FFT algorithm in parallel we need to parallelize each of the four steps described in Section 2.2.

4.1 Projection of Panel Charges onto a Grid

The charges in a cube can be projected onto local representations of the grid, but some interprocessor communication is required to complete the global grid representation because of the distribution of grid z-planes across processors. To illustrate the problem, consider Figure 3-1 where the problem is decomposed between two processors. The grid points (lines in 3-D), identified as $G1, G2$ and $G3$, are the interface points, and are shared by cubes 2 and 3 where cubes 2 and 3 belong to processors 1 and 2 respectively. Grid points $G1, G2$ and $G3$ are assigned to processor 2 to balance the FFT computation and these grid points are not known to processor 1. However, processor 1 stores an extra line (or plane) to maintain information about these interface points. Denoting the projected charges at grid point $G1$ in processors 1 and 2 by q^1_{G1} and q^2_{G1} respectively, we obtain the global value for the charge at grid point $G1$ using the equation $q_{G1} = q^1_{G1} + q^2_{G1}$. To obtain the global values, each processor i (except the last processor) sends the extra plane of data it stores for the interface

points to processor $i+1$. Processor $i+1$ receives and adds the data to its local data to obtain the global values for the interface plane. Processor i, at this stage, does not need the global values for the extra plane as the interface points are not involved in the convolution operation in processor i.

4.2 3-D Convolution in Parallel

The three-dimensional convolution to compute grid potentials involves a forward 3-D DFT computation of the convolution kernel and the grid point charges, point-wise multiplication of the kernel and the grid point charges in the Fourier domain, and an inverse 3-D DFT of the point-wise multiplied data. The kernel is a fixed set of data and is Fourier transformed and stored. The grid point charges, however, change during each iteration of the GMRES algorithm and must be Fourier transformed for each iteration.

4.2.1 Rationale for a Custom Parallel 3-D Real DFT

The simplest way to perform the convolution is to take an off-the-shelf parallel 3-D FFT algorithm and use it to Fourier transform the kernel and the grid charge data. However, there are two drawbacks to this approach.

The first drawback is that the kernel and grid charge data are real and the DFT of the data is complex. Therefore, it would take twice as much memory to store the Fourier transformed data as it takes to store the original data. In addition to this, the Fourier Transform of a real sequence is conjugate symmetric. There is a redundancy introduced in the Fourier transformed data if this symmetry is not taken into account while computing the DFT. Since half the data is redundant, it would take twice as many operations to compute this DFT if the symmetry of the Fourier transformed data were not exploited. This problem was solved by developing a custom parallel 3-D Real DFT algorithm. The conjugate symmetry relation exploited by Real DFT algorithms is described in Section 4.2.3. A point to note is that off-the-shelf Real DFT algorithms exist [7], and it is possible that off-the-shelf RDFT algorithms

which work on parallel processors also exist. However, these alternatives were not pursued because of the second drawback.

The second drawback applies both to using an off-the-shelf complex FFT algorithm and to using an off-the-shelf Real DFT algorithm. The grid point charge dataset is zero padded in all three dimensions to prevent aliasing when the dataset is Fourier transformed. Therefore, the nonzero data points reside in one octant of the total data space, and occupy one eighth of the total data. Each step of the custom Real DFT algorithm only operates on the lines of the data which are nonzero at that step. An off-the-shelf algorithm would have no way of knowing the data structure and would have to Fourier transform lines of zeros while transforming nonzero data.

These were the two key reasons why a custom Real DFT algorithm was developed for the convolution instead of finding and using an off-the-shelf parallel complex FFT algorithm or an off-the-shelf parallel RDFT algorithm. Overall, the custom RDFT algorithm uses about 50% of the memory that would be required by an off-the-shelf RDFT algorithm, and 25% of the memory that would have been required by an off-the-shelf complex 3-D FFT algorithm. Additionally, the custom RDFT algorithm requires about 58.3% of the computations that would be done by an off-the-shelf RDFT algorithm, and about 29.2% of the computations that would have been done by an off-the-shelf complex 3-D FFT algorithm.

4.2.2 Implementation of the Custom Convolution Algorithm

The convolution is performed by computing the 3-D Real Discrete Fourier Transform of the kernel and the grid charge data, taking the point-wise product of the two data sets, and then computing the 3-D Inverse RDFT of this product. The 3-D RDFT is computed by first computing the 3-D FFT of the packed real data sets and then performing some post-processing steps on these data. This method on a serial processor is described in [7]. The same procedure is followed in the parallel case, except that special care needs to be taken to assure that the data points that interact in each stage of the 3-D FFT as well as in the post-processing stage lie on the same processor while that stage is being processed.

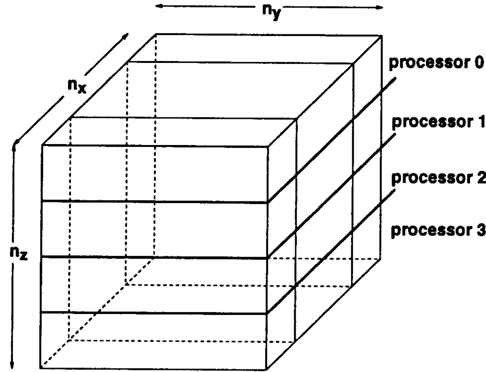


Figure 4-1: Definition of the cube dimensions

The three-dimensional grid charge data and kernel data are distributed across the processors along the z dimension. The FFT computations along the x and y dimensions which are local to a processor do not require interprocessor communication. The FFT along the z dimension has to be performed either by communicating the steps of the FFT from one processor to another, or by redistributing the data through a transpose operation, so that all the points along each z dimension line lie on a single processor. Since communicating the steps of the FFT across processors would be very expensive, a global data transpose is performed. The data is moved using a transpose algorithm which ensures that all the data points that interact during the FFT in the third dimension are on the same processor. The transpose algorithm also ensures that the data points which interact during the post-processing stages also lie on the same processor. Once the RDFT is performed on the kernel and grid point data sets, the two data sets are multiplied point-wise and the inverse operation is performed, to yield the convolved data. A high level description of the convolution algorithm which runs on each processor follows :

for $k = 1$ *to* $n_z p/2$ **do** /*Half the size because of zero padding*/

> **for** $[j = 1{:}n_y/2]$ fft1d$(1, j, k, x, n_x)$; /*FFT in the first dimension; Again half size because of zero padding*/
>
> **for** $[i = 1{:}n_x]$ fft1d$(i, 1, k, y, n_y)$; /*FFT in the second dimension*/

end for

global_transpose();

for $i = 1$ *to* $n_y/2$ **do**

for $j = 1$ *to* $n_xp/2$ **do**

fft1d($i, j, 1, z, n_z$); /*FFT in the third dimension*/

fft1d($wrapi, wrapj, 1, z, n_z$); /*FFT in the third dimension*/

rdft_process($i, j, wrapi, wrapj, n_z$); /*extract two real DFTs from two complex DFTs*/

multiplykernel(i, j, n_z); /*point-wise multiply with kernel*/

multiplykernel($wrapi, wrapj, n_z$); /*point-wise multiply with kernel*/

irdft_process($i, j, wrapi, wrapj, n_z$); /*pack two real DFTs into two complex DFTs*/

ifft1d($i, j, 1, z, n_z$); /*IFFT in the third dimension*/

ifft1d($wrapi, wrapj, 1, z, n_z$); /*IFFT in the third dimension*/

end for

end for

global_transpose();

for $k = 1$ *to* $n_zp/2$ **do**

for [$j = 1{:}n_x$] ifft1d($1, j, k, y, n_y$); /*Inverse FFT in the 2nd dim*/

for [$i = 1{:}n_y/2$] ifft1d($i, 1, k, x, n_x$); /*Inverse FFT in the 1st dim*/

end for

In the above description, n_x, n_y, n_z are the zero-padded sizes along the first, second and third dimensions respectively (see Figure 4-1); $n_xp = \frac{n_x}{nproc}$, $n_yp = \frac{n_y}{nproc}$ and $n_zp = \frac{n_z}{nproc}$; *wrapi* and *wrapj* are the x and y coordinates of the data lines which interact with data lines with x and y coordinates i and j for the Real DFT extraction and packing steps described in Section 4.2.3; fft1d() and ifft1d() are the 1-D forward FFT and inverse FFT respectively with the first three arguments giving the starting coordinates for the FFT and the next two arguments giving the direction

of the FFT and the length of the FFT; rdft_process() and irdft_process() are the post-processing functions which use the knowledge of the original data symmetry to extract the Real DFT values from the complex packed FFT values, and then to re-pack them; multiplykernel() performs a point-wise multiplication of the grid charges by the kernel in the Fourier domain and global_transpose() is a global operation which requires interprocessor communication, and redistributes the data across the processors so that the FFT in the z dimension and the RDFT processing can be done without additional interprocessor communication.

Note that in computing the 3-D FFT of the packed real data we save 75% of the number of FFTs that would have to be performed in along the x dimension and 50% of the FFTs that would have to be performed along the y dimension, because the algorithm takes into account the zero padding. All the FFTs along the z dimension have to be performed, because the FFTs in the other dimensions have filled in the padded zeros with nonzero data. Assuming that the FFTs are equally expensive along each dimension we get a total cost of $\frac{0.25+0.5+1.0}{3.0} = 58.33\%$ of the work which would have been required to perform the complex 3-D FFT by an off-the-shelf algorithm. This is the reduction in the required number of operations mentioned in Section 4.2.1 while describing the second drawback of using an off-the-shelf DFT algorithm. As an aside, actually the total work done for FFTs in each dimension is not equal; the work along dimension x versus that along dimension y differs by a factor of $\frac{\log(n_x)}{\log(n_y)}$. However, for the purposes of analysis we assume a symmetrical problem, in which $\log(n_x)$ and $\log(n_y)$ would be close to each other, if not equal. The data obtained after the complex 3-D FFT is used for extracting the Real DFTs as described in Section 4.2.3.

4.2.3 Real DFT Extraction and Packing

The RDFT algorithm works by exploiting the conjugate symmetry that the Discrete Fourier Transform of real data exhibits. The DFT of real data shows the symmetry

$$H(\vec{n}) = H(-\vec{n})^* \tag{4.1}$$

where $\vec{n}$ is a vector of indices for multidimensional data and $H(\vec{n})$ are the complex DFT values obtained by Fourier Transforming the real data set. Therefore, we only need to know the first half of these values to completely specify all the values in the Fourier domain. The first 50% of memory and computation savings, claimed in describing the first drawback of an off-the-shelf algorithm in Section 4.2.1, comes from not having to calculate or store the second half of these values.

All the real data was originally stored in the same storage space that is now carrying the first half of the Fourier Transformed data. This was done by packing the real data values as complex numbers, in which the first two real data values were paired as the first complex number and so forth. The RDFT algorithm works by first performing a complex 3-D FFT on this *complex packed* data set, and then extracting the first half of the DFT values of the original (unpacked) *real* data from this Fourier Transformed data set. This extraction step is called the rdft() function in the pseudo-code. The details and working of the RDFT algorithm are provided in [7]. In short, the RDFT extraction step takes two complex data points (i, j, k) and $(wrapi, wrapj, wrapk)$ after the 3-D FFT and replaces them with the DFT values of the real numbers (i, j, k) and $(wrapi, wrapj, wrapk)$ from the original data.

The important aspect of this extraction step, from the point of view of parallel performance, is that each pair of points (i, j, k) and $(wrapi, wrapj, wrapk)$ lie on the same processor when rdft() and irdft() are being called. The interaction coordinate $wrapi$ is defined as :

$$\begin{aligned} wrapi &= 0 && \text{if } i = 0 \\ &= n_x - i && \text{otherwise} \end{aligned} \tag{4.2}$$

and $wrapj$ and $wrapk$ are defined similarly. Also, it is necessary that each z dimension line be entirely on a single processor when fft1d() and ifft1d() in the z dimension are being called as described in Section 4.2.2. Given that the rdft() and irdft() calls are sandwiched between the fft1d() and ifft1d() calls for the z dimension, the algorithm has to ensure that the pairs of entire z dimension lines (i, j) and $(wrapi, wrapj)$ lie on

the same processor after the global_transpose() function has been called. The method for ensuring this data localization is described in Section 4.2.4.

There is one last piece of information about zero padding that we have that allows us to reduce the memory required by 50%. We know that half the data in the z dimension is zero padding. These zeros will get filled in when we perform the FFT in the z dimension as the third step in our complex 3-D FFT algorithm, before performing the rdft() post-processing. Instead of performing the FFT on *all* the z dimension lines before performing the rdft() function on all the lines, we could just take pairs of z dimension lines (i, j) and $(wrapi, wrapj)$ in two buffers. Then we could perform the fft1d() on both lines, rdft() on the pair, multiplykernel() with each line, then perform irdft() on the pair and finally ifft1d() on both lines. Only the first half of each of these buffers contains data that is necessary for the last two ifft1d() steps, since we are not concerned with the data that is now filling the zero padding regions. This means we can get by with two buffers having the length of the z dimension instead of filling in the z dimension zeros with intermediate data that is later ignored. Since we do not need to fill the z dimension zeros, we do not need to allocate them either; just knowing that they are zeros is sufficient. This memory reduction achieved in this custom RDFT algorithm is the same 50% memory reduction claimed in describing the second drawback to off-the-shelf FFT and RDFT algorithms, in Section 4.2.1.

4.2.4 Transpose Operation

The global transpose operation is the core of the parallelization of the algorithm and constitutes the major interprocessor communication step. To understand the transpose operation, consider a situation in which the problem is distributed across 4 processors, as shown in Figure 4-1. The one dimensional FFTs are first performed along the x and the y dimensions, without any interprocessor communication, as described in Section 4.2.2. The transpose operation has to now ensure that each z dimension line of data lies on a single processor after the transpose. It also has to ensure that the pairs of z dimension lines (i, j) and $(wrapi, wrapj)$ also lie on the same processor after the transpose.

The first objective is achieved if each x-plane of the input grid is subjected to a block transpose across processors as shown in Figure 4-2. We call this the primary block transpose. The data on processor i which belongs on processor i is denoted by (i, i) and the data on processor i which belongs on processor j is denoted by (i, j). Figure 4-3 shows the location of a z dimension line across the processor before a transpose and how it is placed on a single processor after the transpose operation. The thin arrows show the order in which the data is to be read from the processor in order to assemble the entire z dimension line. The 1-D FFT in the z dimension can now be performed without additional interprocessor communication. This simple transpose operation satisfies the first objective, but does not satisfy the second objective of ensuring that (i, j) and $(wrapi, wrapj)$ lie on the same processor.

Let us consider another block transpose operation, as shown in Figure 4-4. This second block transpose is a transpose across the secondary diagonal (diagonal from the top right to the bottom left) of the x-plane. We call this transpose the secondary block transpose. The secondary transpose operation also places each z dimension line in the plane onto a single processor as shown in Figure 4-5 so that the 1-D FFT in the z dimension can be performed without additional interprocessor communication. However, again this transpose across the secondary diagonal satisfies only the first objective of allowing the 1-D FFT but does not satisfy the second objective of ensuring that lines (i, j) and $(wrapi, wrapj)$ lie on the same processor.

The second objective of having both z-lines (i, j) and $(wrapi, wrapj)$ on a single processor can be achieved by using the primary block transpose operation for half of the x-planes and the secondary block transpose operation for the other half of the x planes. Note that we can achieve the first objective independently on any x-plane by using either the primary or the secondary transpose on that plane. Now for achieving the second objective, for every x plane i that we transpose using the primary block transpose, we transpose the x-plane $wrapi$ using the secondary block transpose. For example consider the plane in Figure 4-3 to be the ith x-plane and the plane in Figure 4-5 to be the $wrapi^{th}$ x-plane. The marked z-line in Figure 4-3 is the z-line (i, j) because it is the jth line in the ith x-plane and the marked z-line

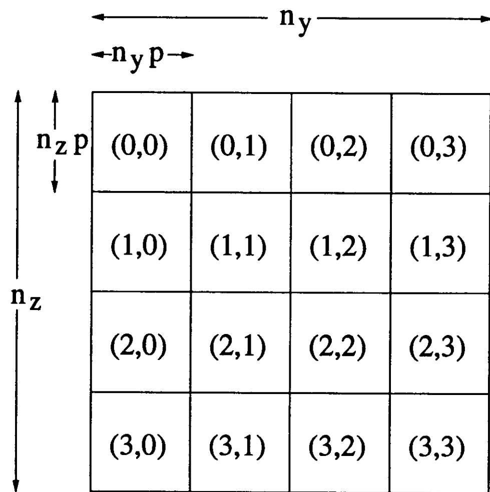


Figure 4-2: Primary block transpose operation

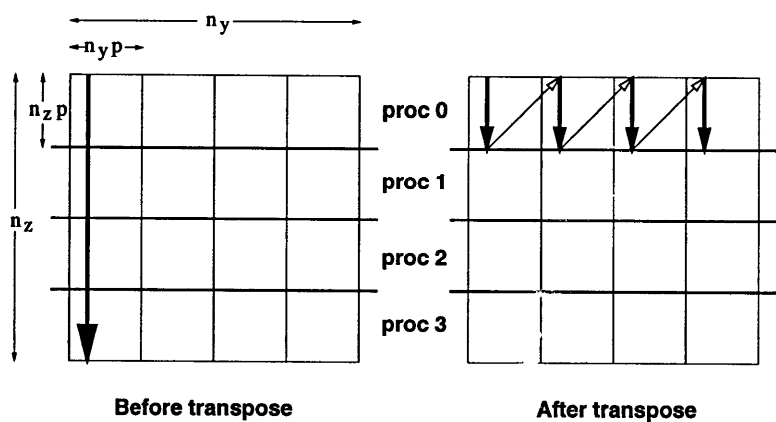


Figure 4-3: Movement of z-lines in the primary block transpose

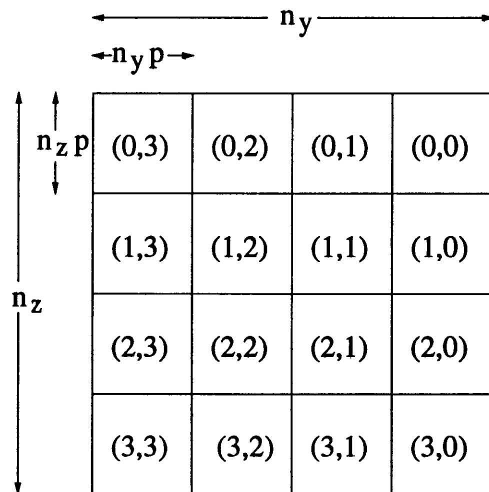


Figure 4-4: Secondary block transpose operation

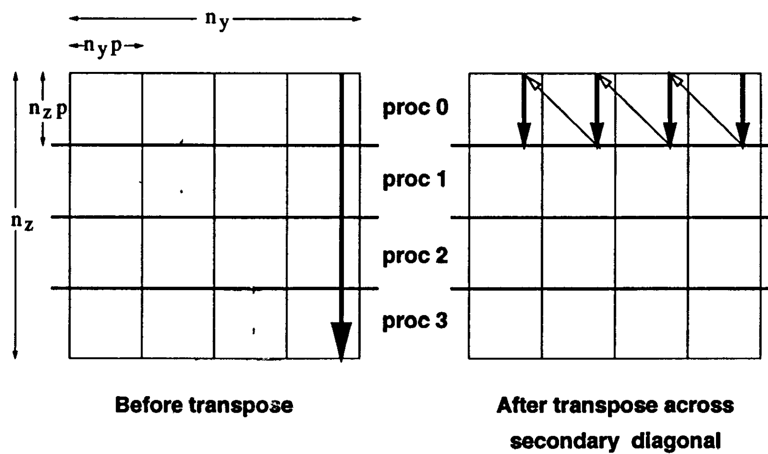


Figure 4-5: Movement of z-lines in the secondary block transpose

in Figure 4-5 is the z-line $(wrapi, wrapj)$, because it is on plane $wrapi$ and it is the $(n_y - j)^{th}$ line, i.e. the $wrapj^{th}$ line. Note that both marked lines lie on the same processor after their respective transposes. These two marked lines are the lines that interact for the RDFT extraction step. Similarly, each pair of lines that interact lie on the same processor if the primary and secondary transposes are used respectively on the ith and $wrapi^{th}$ x-planes. This can be simply done by applying the primary transpose on all x-planes for i=0:$\frac{n_x}{2}$-1 and the secondary transpose on all x-planes for i=$\frac{n_x}{2}$:n_x. Thus, the second objective of having each interacting complex number pair on the same processor is achieved.

Two special cases need to be discussed for completeness. These are the cases in which i_0 is either 0 or $\frac{n_x}{2}$ or j_0 is either 0 or $\frac{n_y}{2}$. The problem that arises when i_0 is 0 or $\frac{n_x}{2}$ is that we need plane i_0 to be transposed using the primary transpose but $wrapi_0$ to be transposed using the secondary transpose, but $i_0 = wrapi_0$. To work around this problem, the two planes in which i_0 is either 0 or $\frac{n_x}{2}$ are considered separately as special cases, and for simplicity are computed on all processors. These single plane calculations could also have been parallelized, but the relatively low benefit to be gained from this did not warrant the extra complexity of code.

The second special case is when j_0 is either 0 or $\frac{n_y}{2}$, which means that $j_0 = wrapj_0$. Let us consider this exclusive of the first special case, so that $i \neq wrapi$. In this case the z-line (i, j_0) interacts with the line $(wrapi, j_0)$. Without loss of generality we can assume that $i < \frac{n_x}{2}$ and $wrapi > \frac{n_x}{2}$. Then if the plane i gets transposed with the primary transpose, we need to ensure that the z-lines $(wrapi, 0)$ and $(wrapi, \frac{n_y}{2})$ are transposed with the equivalent of the primary transpose, while ensuring that all the other z-lines $(wrapi, j)$ are transposed with the equivalent of the secondary transpose. This can be achieved in the framework of the secondary transpose, without any extra communication.

4.3 Interpolation of Grid Potentials onto Panels

As explained in Section 4.1 each processor, except the last, has an extra plane in which it can store grid charge or grid potential data. The extra plane in processor $(i-1)$ stores the data for the first plane of grid points in processor i. This ensures that no interprocessor communication is needed during the projection of panel charges to the grid or during the interpolation of grid potentials onto the panels.

Once the convolution is completed, the potential at the grid points is available. The first plane of grid point potentials in each processor i (except processor 0) is sent to processor $(i-1)$. Processor $(i-1)$ overwrites the extra plane it stores with the received potential data. The panel potentials in each processor are then computed locally from the grid potentials without any further interprocessor communication.

4.4 Precorrected Direct Interactions

The precorrected direct interaction between panels in a cube and panels in the cube's neighbor is computed directly using (2.7). If all the neighbors of a cube lie in the same processor, then no interprocessor communication is needed to compute precorrected direct interactions. However, if a cube's neighbor lies on a remote processor, information about the panels in the neighbor must be communicated to the cube's processor. This can be quite expensive as direct interactions are recomputed every time a matrix-vector product is needed. A faster approach is to eliminate most of this interprocessor communication by storing copies of panel charges for remote-processor neighbors, though this approach requires somewhat more memory. For the experiments we have conducted, the direct interactions of a cube are computed using the faster approach based on storing copies.

5

Computational Results

Computational results were obtained on an 8 node IBM SP2 parallel scalable system. Each SP2 node is a RISC System/6000 590 workstation with a IBM POWER2 Architecture. Four nodes have 512 MB of memory and the other four nodes have 256 MB of memory. The nodes are connected by a high performance switch with point to point communication. The unidirectional communication bandwidth at each node is about 40MB/second.

The computational results are divided into three sections. Section 5.1 covers parallel performance of Parallel FFTCAP on several problems as the number of processors is increased. This section as well as Section 5.3 assumes the use of adaptive grid depth selection and adaptive inter-grid spacing selection for enhanced parallel performance. Section 5.2 shows the effects of the adaptive inter-grid spacing selection, developed in Section 3.3, on the parallel performance of the algorithm. Section 5.3 gives some examples of large problems that have been solved using Parallel FFTCAP.

The time and memory costs for Parallel FFTCAP on a single processor are within 5% of the time and memory required by the FFTCAP algorithm. For examples in which enabling inter-grid spacing selection gives a performance improvement, Parallel FFTCAP on a single processor performs significantly better than FFTCAP.

5.1 Parallel Performance

The parallel performance of an algorithm is measured by comparing the computational resources required by the parallel algorithm on several machines, to the resources that would be required to run the same algorithm on a single machine. Parallel speedup is defined as the ratio of the length of time a parallel algorithm takes to run on one processor to the length of time it takes to run on several processors.

$$\text{Parallel Speedup} = \frac{T_1}{T_n} \tag{5.1}$$

where T_1 is the time required by the parallel algorithm on 1 processor and T_n is the time required by the parallel algorithm on n processors. Ideally for n processors the speedup is n, because this indicates that there is no parallel overhead.

In the precorrected-FFT based capacitance extraction algorithm, we can trade off time and memory by changing the grid depth as described in Section 2.2.5. In [5] the time-memory product for FFTCAP was shown to be more or less independent of whether the best grid for minimum memory usage or the best grid for minimum processor time usage was chosen. Therefore another figure of merit of the parallel performance of Parallel FFTCAP is the parallel scaling of the time-memory product, defined along the same lines as the parallel speedup. We also look at the parallel scaling of memory usage. Memory usage scaling is an important figure of merit, because memory is the limiting factor that determines the maximum size of a problem that can be solved using this algorithm.

Figure 5-2 shows the plots of parallel performance scaling of the Parallel FFTCAP in extracting the coupling capacitance matrix of three structures, using each of these three metrics. The three structures simulated were a cubic capacitor discretized to 60,000 panels, shown in Figure 5-1, a 15×15 woven bus discretized to 82,080 panels, shown in Figure 5-4(a) and a 10×10 bus crossing discretized to 84,280 panels, shown in Figure 5-5(a).

The cubic capacitor and the 10×10 bus crossing examples show good parallel time scaling and a fair memory scaling. Note that the memory scaling cannot be

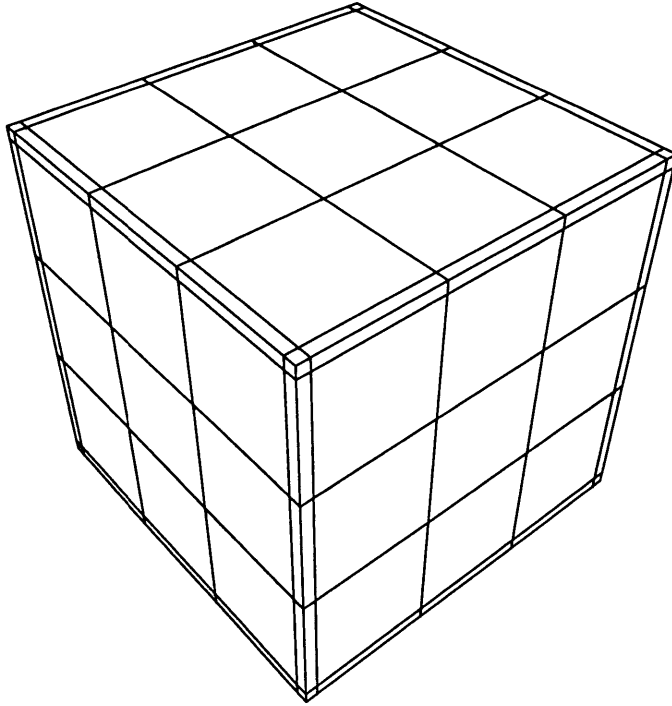
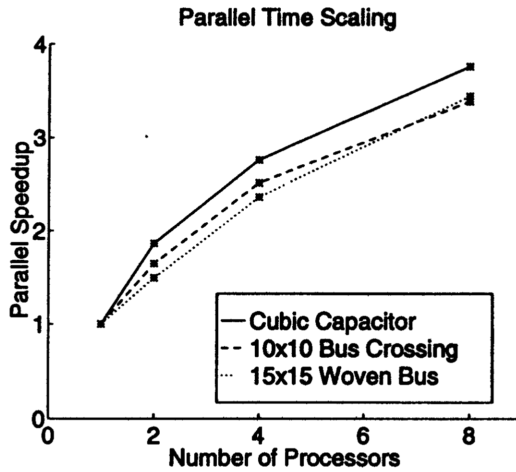


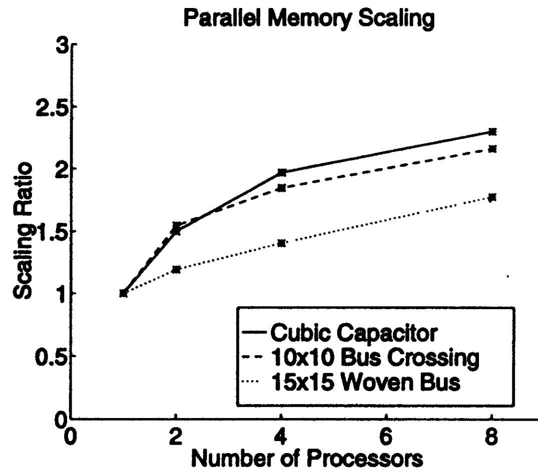
Figure 5-1: Cubic capacitor discretized to 60,000 panels

perfect because there are several parts of the data that need to be stored on all the processors for the algorithm to work efficiently. The overall scaling of the algorithm can be measured by the time-memory product, which captures the total costs of the algorithm, and puts aside differences that can be achieved by trading off time and memory. The time-memory product shows similar scaling in both the cubic capacitor and the 10×10 bus crossing case.

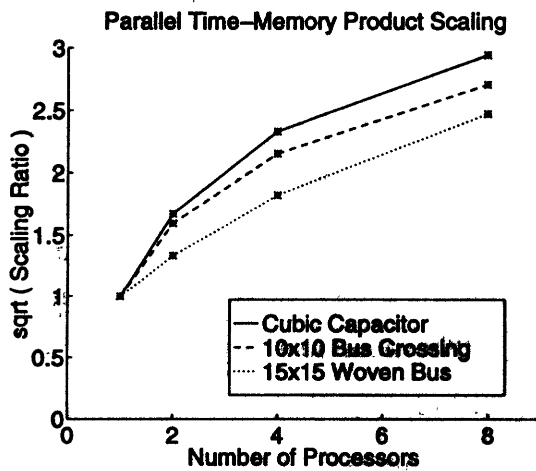
The 15×15 woven bus shows a time scaling very close to that of the cubic capacitor but a relatively poor memory scaling. The poor memory scaling is due a choice made by the adaptive grid scaling algorithm, to reduce the time-memory product cost. The memory scaling could have been improved as shown in Figure 5-4(c), but only by paying a significant premium in the number of computations as shown in Figure 5-4(a). Section 5.2 presents a discussion of the tradeoffs associated with grid scaling. The overall parallel scaling of this example, shown by the time-memory product, is still comparable to that of the other two examples shown.



(a)



(b)



(c)

$$\text{Scaling Ratio} = \frac{\text{cost on 1 processor}}{\text{max cost among n processors}}$$

Where cost is time, memory, or time-memory product.

Figure 5-2: Parallel scaling of costs

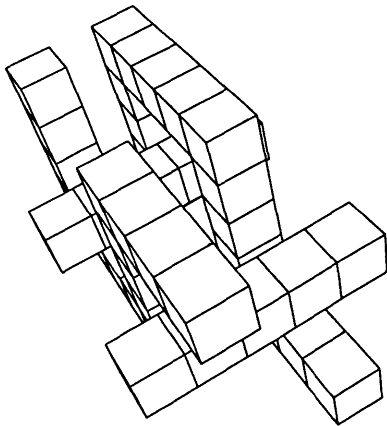
5.2 Effects of Grid Scaling

The selection of inter-grid spacing, described in Section 3.3, can have a significant impact on the parallel performance of the Parallel FFTCAP algorithm. Figure 5-3 shows the performance of the Parallel FFTCAP algorithm on a 2×2 woven bus discretized to 39,600 panels. The dashed line in the plot shows the performance of the algorithm with the adaptive grid scaling procedure disabled, and the solid line shows the performance with adaptive grid scaling enabled. An increase in speed of about a factor of two can be seen due to this grid scaling. In addition to this, an increase in the parallel efficiency in memory usage is also seen, because the panels get distributed much more equitably across processors after scaling the grid.

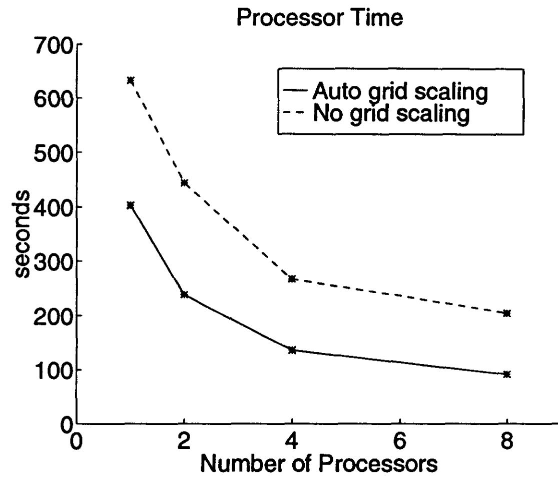
It is very important that this grid rescaling be done in an adaptive manner and not performed as a blanket rescaling for all problems. In some cases it can be more efficient to give up on load balancing the panels and trade off the factor of two decrease in size of the FFT, so that the grid does not become too sparse through rescaling. The tradeoffs associated with selecting the grid depth are described in Section 2.2.5. The adaptive algorithm predicts whether rescaling the grid would be efficient.

The dashed lines in Figure 5-4 show the parallel performance of the algorithm on a 15×15 woven bus problem on which grid rescaling has been forced, while the solid line shows the performance on the same problem with adaptive grid scaling. In this case the adaptive algorithm has opted not to scale the grid. Figure 5-4(b) shows that a significant cost in processor time would be incurred if the grid scaling were forced. Figure 5-4(c) shows that the parallel efficiency of memory distribution would be increased by performing the grid scaling, but significant memory gains would not be achieved. In the time-memory product, however, the adaptive algorithm which chooses not to scale the grid, clearly wins out. This is because the adaptive algorithm tries to choose the grid depth and inter-grid spacing which will use the smallest time-memory product for solving the problem.

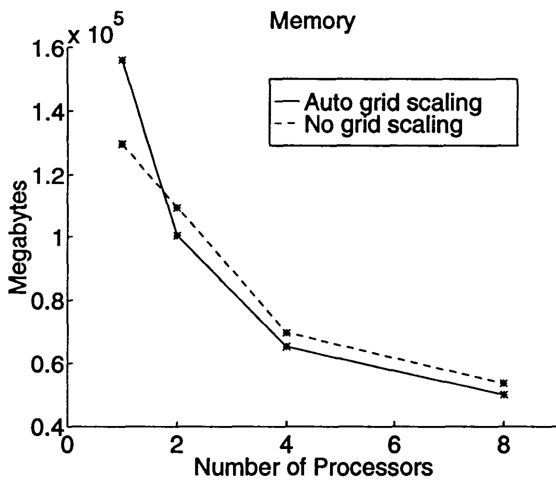
The third example of the grid scaling shows a situation in which the adaptive



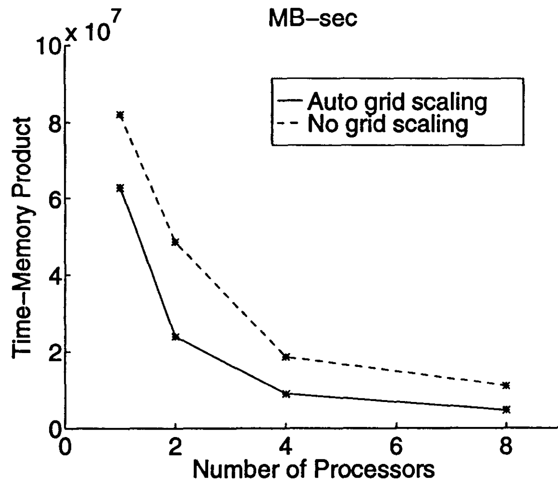
2×2 Woven Bus - 39,600 Panels
(a)



(b)

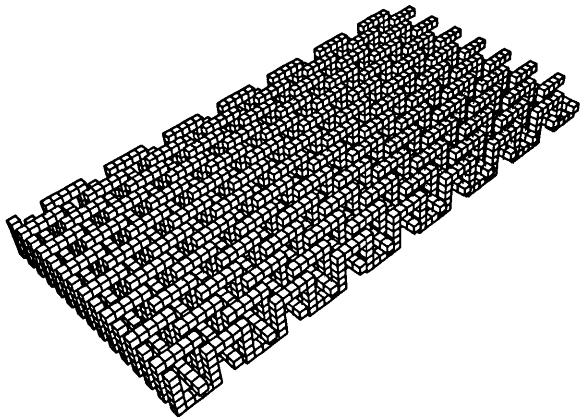


(c)

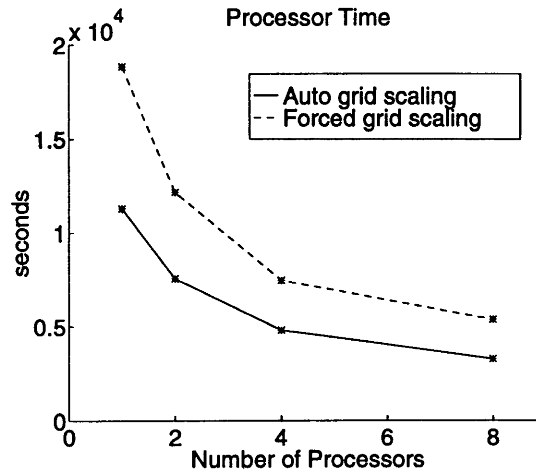


(d)

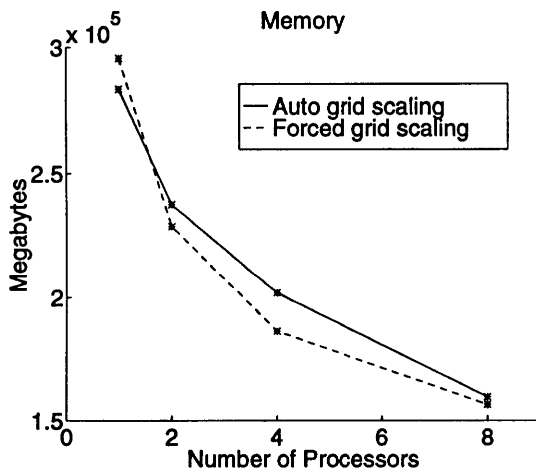
Figure 5-3: Performance gains through grid scaling



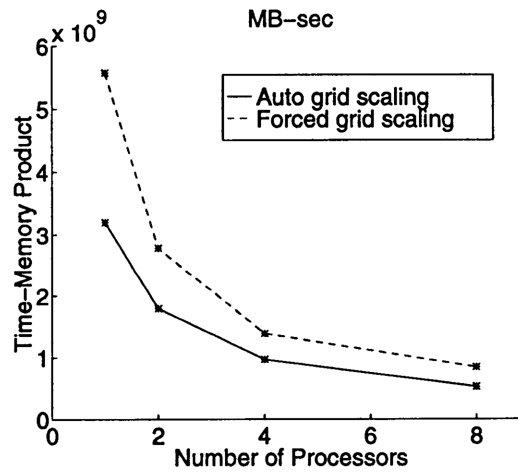
15×15 Woven Bus - 82,080 Panels
(a)



(b)

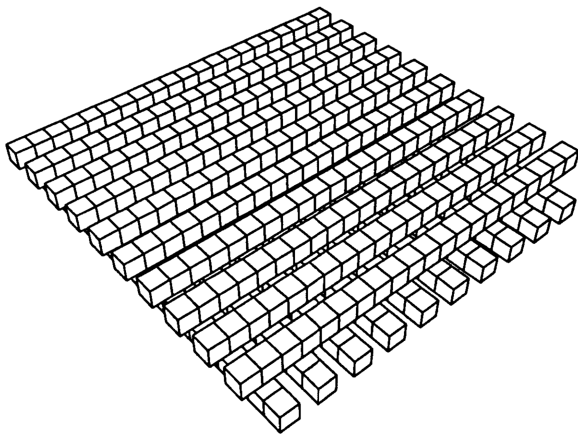


(c)

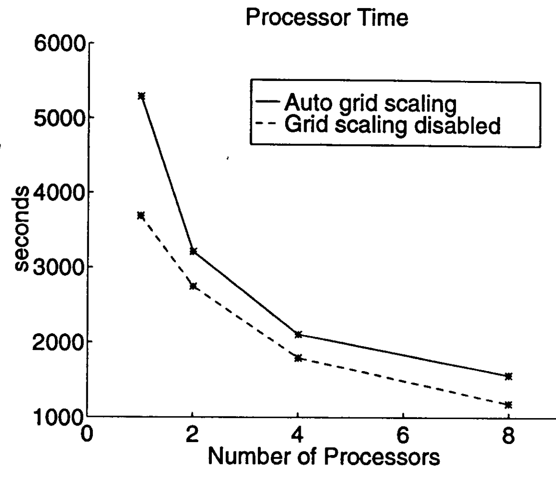


(d)

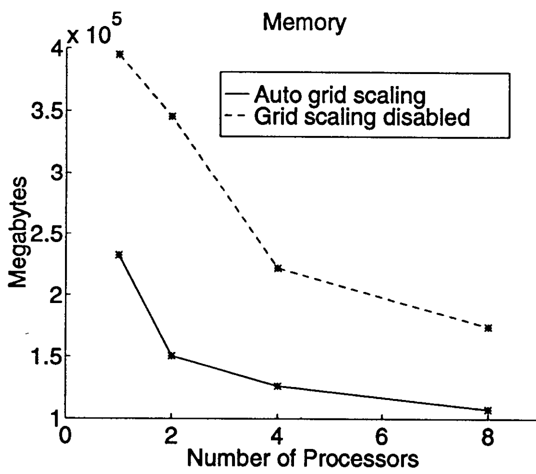
Figure 5-4: Adaptive grid scaling performs better than forced grid scaling



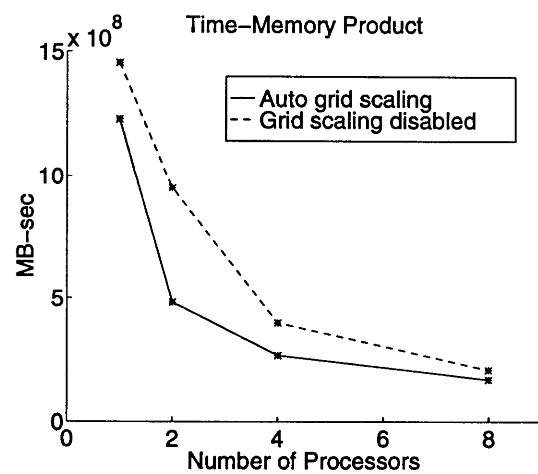
10×10 Bus Crossing - 84,280 Panels
(a)



(b)



(c)



(d)

Figure 5-5: Adaptive grid scaling trades off speed for memory performance

algorithm fails to choose the better of the two scaling choices based on processor time usage, but does significantly better on memory usage. Figure 5-5 shows a plot of the parallel performance of Parallel FFTCAP on the 10×10 bus crossing structure discretized to 84,280 panels. The dashed line shows the time required when adaptive grid scaling is disabled and the solid line shows the time required when the adaptive grid scaling is enabled. Note that the overall cost of the option chosen is lower, based on the time-memory product.

The adaptive algorithm chooses whether to scale the inter-grid spacing for a given depth grid based on the magnitude of the scaling factor. If the scaling factor is large, then the grid is not scaled, because rescaling the inter-grid spacing by a large factor is likely to increase the local interaction costs significantly as described in Section 2.2.5. However, if the grid scaling factor is small then the grid is scaled. After the decision about whether to scale the grid for the given grid depth is made, the algorithm computes the expected local interaction and FFT cost in the form of an expected total time-memory product for that grid depth. The adaptive grid selection algorithm then chooses a grid depth, with its associated inter-grid spacing based on which has the smallest expected cost. A high level description of this entire algorithm is given in Section 3.3.

A more robust and accurate approach is to actually calculate the expected local interaction and FFT cost for both the scaled and unscaled inter-grid spacing within a given grid depth, and then choose the one with the lower expected cost. This optimization goes hand in hand with other grid optimizations to the single processor code (FFTCAP) that were discovered to be possible in the course of this research. The implementation of these optimizations was beyond the scope of this research project.

5.3 Solving Large Problems

The two major objectives behind implementing a parallel program are:

1. To solve problems faster than they can be solved on a single processor.

2. To solve larger problems than possible on a single processor.

Section 5.1 measured the extent to which the first objective was achieved, and additionally discussed memory scaling issues that tied into the second objective. This section focuses on the performance of the algorithm on problems which were too large to solve on a single SP2 node with 512 megabytes of memory. It also shows the largest problem that has been solved on a single SP2 node. Table 5.1 shows the time and memory required on each processor to solve these large problems.

Structure	Number of Panels	Number of Processors	Time in sec	Memory in MB	Time-Memory Product
18×18 woven bus	209,664	4	12,474	421,550	5.3498×10^9
18×18 woven bus	209,664	8	8,583	335,553	3.0483×10^9
32×32 bus crossing	150,912	4	16,532	415,361	6.9671×10^9
32×32 bus crossing	150,912	8	11,477	395,853	4.6334×10^9
20×20 woven bus	258,560	8	24,489	410,637	1.2511×10^{10}
32×32 bus crossing	268,288	4	38,359	399,157	1.5311×10^{10}
32×32 bus crossing	268,288	8	55,608	350,402	2.9485×10^{10}
cubic capacitor	144,150	1	959	491,145	4.7101×10^8
cubic capacitor	415,014	8	1,395	502,617	7.0115×10^8

Table 5.1: Large problems solved using Parallel FFTCAP

One set of results that stands out from the table are the timing results for the 32×32 bus crossing example. This structure looks very similar to the 10×10 bus crossing in Figure 5-5, except that it has 32 conductors in each direction instead of 10, and is composed of 268,288 panels. The results seem anomalous because 8 processors take longer to solve this problem than 4 processors do. This is because there is only 256 MB of memory on processors 5 to 8, compared with 512 MB on processors 1 to 4. Since the problem requires more than 256 MB per processor when running on 8 processors, part of the problem is swapped to virtual memory on disk. This swapping slows the algorithm down significantly. However, since the algorithm requires less than 512 MB per processor when running on 4 processors, the algorithm runs without swapping, and thus runs faster on 4 processors, leading to the seemingly anomalous result.

The largest problem solved on a single SP2 node with 512 MB of memory is the cubic capacitor discretized to 144,150 panels. It was solved in 959 seconds, which is fast compared to the other problems. The cubic capacitor discretized to 415,014 panels has a larger number of panels than any of the other problems, but it also has been solved faster than the other problems. The reason is that the potential equations to compute capacitance need to be solved for only one conductor for each cubic capacitor example, compared to 64 solves that need to be executed in the case of the 32×32 bus crossing, or 36 solves that need to be performed in the case of the 18×18 woven bus. The cubic capacitor with 415,014 panels is the largest problem that has been solved on our parallel system.

6

Conclusion

In this thesis, a parallel algorithm for capacitance extraction in complicated three dimensional structures was presented. Parallel FFTCAP, an implementation of this algorithm, was also developed and analyzed on an eight processor IBM SP2.

The capacitance extraction problem was mathematically formulated in Chapter 2, and the precorrected-FFT based capacitance extraction algorithm developed in [6] to solve this problem was described. In Chapter 3, the issues involved in decomposing the problem and the algorithm across processors were discussed. Two methods for load balancing the algorithm across processors were also presented in this chapter. Chapter 4 described the implementation of the algorithm, focusing on the issues involved in parallelizing FFTCAP.

Chapter 5 presented the results of computational experiments performed using Parallel FFTCAP. The parallel performance of the algorithm was analyzed on three cost metrics, time, memory and time-memory product. The algorithm was shown to have reasonable parallel scaling in all three metrics. The adaptive inter-grid spacing selection algorithm was shown to provide up to a factor of two cost improvement of the algorithm, in time, memory or in time-memory product. The single processor performance improvements achieved in Parallel FFTCAP through the inter-grid spacing selection can be translated directly into equivalent improve-

ments in the performance of the single processor FFTCAP algorithm for capacitance extraction. Finally, the limits of the IBM SP2 running Parallel FFTCAP were tested with large capacitance extraction problems. The largest problem solved on the 8 processor IBM SP2 was the cubic capacitor with 415,014 panels.

Bibliography

[1] N. R. Aluru, V. B. Nadkarni, and Jacob K. White. A parallel precorrected FFT based capacitance extraction program for signal integrity analysis. In 33rd *ACM/IEEE Design Automation Conference*, pages 363–366, Las Vegas, Nevada, June 1996.

[2] A. Brandt. Multilevel computations of integral transforms and particle interactions with oscillatory kernals. *Computer Physics Communications*, 65:24–38, 1991.

[3] L. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* M.I.T. Press, Cambridge, Massachusetts, 1988.

[4] K. Nabors, S. Kim, and J. White. Fast capacitance extraction of general three-dimensional structure. *IEEE Trans. on Microwave Theory and Techniques*, 40(7):1496–1507, July 1992.

[5] J. Phillips. *Rapid Solution of Potential Integral Equations in Complicated 3-Dimensional Geometries.* PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1997.

[6] J. Phillips and J. White. A precorrected-FFT method for capacitance extraction of complicated 3-D structures. In *Proceedings of the Int. Conf. on Computer-Aided Design*, November 1994.

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*, chapter 12. Cambridge University Press, second edition, 1992.

[8] A. E. Ruehli and P. A. Brennan. Efficient capacitance calculations for three-dimensional multiconductor systems. *IEEE Transactions on Microwave Theory and Techniques*, 21(2):76–82, February 1973.

[9] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SISSC*, 7(3):856–869, July 1986.