Towards Scalable Parallel Simulation of the Structural Mechanics of Piezoelectric-Controlled Beams

Jeremy M. Rotter

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Calvin J. Ribbens, Chair
Layne T. Watson
Romesh C. Batra

June 18, 1999
Blacksburg, Virginia

Keywords: Parallel, Scalable, Distributed Memory, Finite Element, Piezoelectric

Towards Scalable Parallel Simulation of the Structural Mechanics of Piezoelectric-Controlled Beams

Jeremy M. Rotter

(ABSTRACT)

In this thesis we present a parallel implementation of an engineering code which simulates the deformations caused when forces are applied to a piezoelectric-controlled smart structure. The parallel simulation, whose domain decomposition relies on the finite element representation of the structure, is created with an emphasis on scalability of both memory requirements and run time. We take into consideration sequential performance enhancements, the structure of a banded symmetric positive definite linear system, and the overhead required to completely distribute the problem across the processors. The resulting code is scalable, with the exception of a banded Cholesky factorization, which does not fully utilize the parallel environment.

Acknowledgments

First, and foremost, I would like to extend my deepest gratitude to my advisor, Dr. Calvin Ribbens, for the enormous amount of time and effort he devoted to helping me complete this thesis. His leadership, advice, encouragement, and sense of humor kept me going through this difficult process. I consider his friendship to be one of the greatest rewards of this experience, and I hope some day I can influence someone's life as he has influenced mine.

I would also like to take this opportunity to thank my other committee members for their advice and guidance through this whole process. I thank Dr. Layne Watson for both his assistance in troubleshooting difficult problems and for his expertise in numerical analysis and parallel computing. I thank Dr. Romesh Batra for his support of my endeavors over the past 3 years, and for always having a minute (or more) to answer my engineering questions.

For writing and documenting the piezoelectric code, I thank Xiaoqing Liang. Also, I am thankful to Art Geng for the hours of help he gave me in debugging my code and mesh generator.

For their sympathy, support, and willingness to listen to hours of complaining, I would also like to thank Matthew Burnett, Jason Thweatt, and Kevin Motto. I am honored to call you colleagues.

Finally, I would like to thank my parents, Jack and Kathy, and my sister, Heather for their love and support over the past 6 years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As available computing resources become cheaper and more powerful, engineers in all fields find numerical simulation to be an attractive alternative to physical experimentation. If a problem can be mathematically modeled and simulated, a computer program can give engineers more freedom in modifying experimental parameters with more accurate results. Numerical simulations are often faster, cheaper, and safer than other experiments.

In order for numerical simulations to be useful, they must be accurate and efficient. Accuracy is clearly the higher priority, and as a result, simulations can often take from hours to weeks to run, even on high end workstations. Furthermore, in order to gain the desired accuracy in results, simulations may require enormous amounts of memory as well. Parallel computation can help solve both of those problems.

By porting a simulation code to a parallel machine, we take the same simulation that was running on a single processor, and instead divide it into parts to run on several processors—sometimes hundreds or thousands of processors. Using this divide-and-conquer approach, we can run the simulation faster, and with less memory per processor than it originally required. However, this process of "parallelizing" a sequential program can be quite difficult.

To simplify things, programmers can utilize parallel compilers and programming tools, like High Performance Fortran [18] and KAP [19], which are capable of finding some parallelism within codes and exploiting it. These tools are often helpful; ideally, they can quickly allow a code to run in parallel, even if the programmer has little knowledge of how the code

works. If the ultimate goal is a scalable, highly efficient parallel implementation of the code, however, automatic tools will rarely get the job done. These tools lack knowledge of the actual problem that is being simulated. They look superficially at the code for common parallelizable structures, which program complexity can often conceal.

Even if parallel compilers and programming tools could always implement a perfectly parallel version of the code they are given, potential improvements routinely lie in the original code. Often, engineers who write these simulations may not have strong backgrounds in the fields of numerical analysis, computer algorithms, parallel programming, or computer architecture. As a result, improvements to the sequential code may more readily (and efficiently) run in parallel. Algorithmic improvements will help the code run faster sequentially and in parallel, especially for expensive operations, like the solution of linear systems of equations or the computation of eigenvalues. There might also be inconsequential assumptions about the input data which could lead to great improvements in performance.

Many tasks are common to numerical simulations, but are difficult to code efficiently in parallel. For several of these, there exist parallel library routines, which a programmer can call from their code. These libraries are often freely accessible, and programmers can usually find optimized versions for the platform they are working on. An example of a parallel library is SCALAPACK [4], the parallel extension of the LAPACK [3] linear algebra library. SCALAPACK is designed to allow an easy transition from sequential to parallel execution for codes calling LAPACK routines.

With all these compilers, tools, and libraries to aid the parallel programmer, the process of implementing a parallel code might seem easy. Ideally, the programmer would be the same engineer who mathematically modeled the program and wrote the sequential code. Unfortunately, there aren't many people with a strong enough understanding of all the fields necessary to do that. In most real life cases, we settle for the next best thing: a team of collaborating experts in their respective fields. For instance, a good team to write a parallel simulation would be an engineer who is capable of writing (or has already written) a sequential simulation code, a numerical analyst who understands computer algorithms to solve linear algebra problems, and a computer scientist with a thorough understanding of parallel computing. There will inevitably be some overlap in knowledge on the team, which will only aid each party in understanding the others' opinions. Communication between all

members of the team is essential to the code's development. With close attention paid to each aspect of the parallel implementation, efficiency and scalability can be obtained.

In this thesis, we show that it is possible to make an engineering simulation code scalably parallel, but it takes a great deal of work, and many issues must be considered. We must consider a wide range of factors, from sequential optimization, to potential parallelism, to the details of parallel implementation, including problem and data decomposition and communication requirements.

The remainder of the thesis is organized as follows. In the remainder of this chapter, we present the problem, our computational environment, and some related work in this field. In Chapter 2 we describe the sequential simulation code, followed by our first parallel implementation in Chapter 3. In Chapter 4 we focus on issues concerning the largest remaining sequential bottleneck, namely the solution of a linear system. Chapter 5 describes an improvement designed to significantly reduce the per processor memory requirements of the simulation, thus improving scalability. In Chapter 6 we look at the parallel performance of our implementation, and in Chapter 7 we conclude with a summary of our results and look at future issues. Appendix A contains pseudocode for the sequential program, and Appendix B discusses MESH, our preprocessing program which discretizes the physical domain being simulated and generates input files for the parallel simulation code.

1.1 The PZT Problem

Certain materials, when under stress, produce an electric surface charge [22]. This is called the piezoelectric effect, and these materials are called piezoelectric materials. The converse piezoelectric effect is also true: when electrical charges are applied to certain substances, mechanical deformations occur. By utilizing these effects, materials engineers can use patches of piezoelectric materials to first detect, or "sense," the stresses in a material, and then apply, or "actuate," a charge to counteract those stresses. These piezoelectric materials are quite light, and as a result, do not have much influence on the deformation of the non-piezoelectric material. A material with piezoelectric patches monitoring and controlling it in this fashion is sometimes called a "smart" structure.

There are an enormous number of useful applications for smart structures. Already, piezoelectric materials are used in a wide range of applications, from the stabilization of satellites to airbag deployment in cars. The most commonly used piezoelectric material for these purposes is lead zircon titanite, hence piezoelectric materials are also referred to as PZT materials.

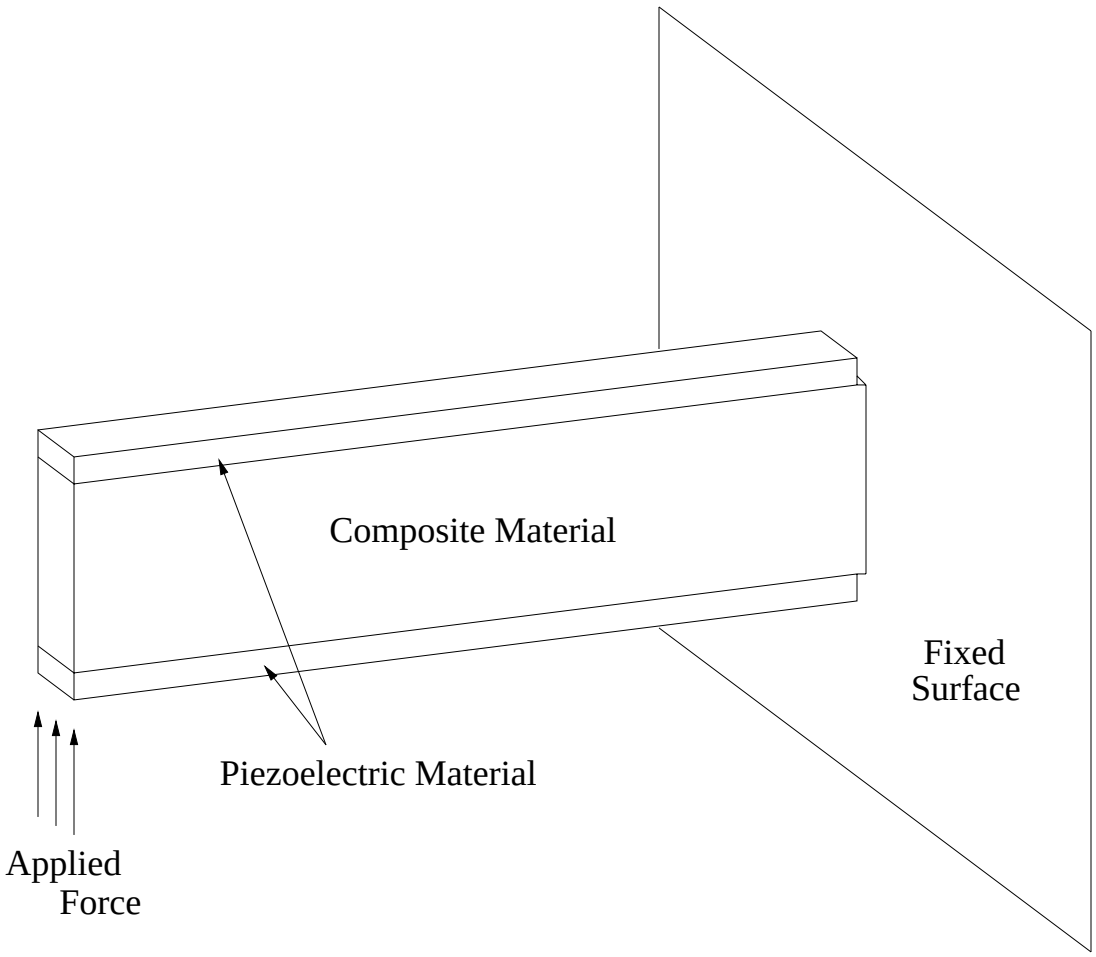

Figure 1.1: The smart structure used in our numerical simulation.

For the remainder of this thesis, we will be using a numerical simulation to study the effects of forces applied to a specific type of smart structure. The structure consists of a beam made of composite material, fixed at one end to an immovable surface, with PZT patches on both its top and bottom. We will apply a force on the free end of the beam, and compute the stresses on the beam throughout the simulation. In our problem, both PZT

patches are sensors, but with a slight modification to the code, one patch can be changed to an actuator, which can be used to reduce stresses in the smart structure. An illustration of the structure is shown in Figure 1.1.

In 1997, Liang [20] created a numerical simulation, called SMART, which uses the finite element method to analyze three-dimensional transient finite deformations of a nonlinear smart structure. For SMART to run, we must first discretize the structure into nodes and elements, where each three-dimensional brick element is delimited by eight nodal points. SMART then performs calculations on nodal and elemental values over discrete timesteps. As with other finite element codes, the accuracy of the output is determined by the number of elements used to simulate the problem, as well as the size of the timestep we use. For instance, we should expect better accuracy from a finite element representation, or mesh, of $100 \times 100 \times 100$ elements than we can from a mesh of $10 \times 10 \times 10$ elements. Furthermore, we should expect more accurate results from a timestep of 10^{-7} seconds than we should with a timestep of 10^{-5} seconds.

The purpose of SMART is to find both the elemental stresses and the nodal displacements in the smart structure, given the applied forces and charges to that structure. To find this information, we must find the forces on each node in the system, which can result from either external applied forces or from charges applied to the PZT material. SMART assumes that the behavior of the system is governed by the following two equations:

$$M\ddot{u}(t) = F_{ext}(t) - F_{int}(u(t), \Phi(t)),$$

$$P_{int}(u(t), \Phi(t)) - P_{ext}(t) = 0,$$

where the unknowns are the displacement $u(t)$ and the potential $\Phi(t)$ at each node. In the first equation, M represents the mass matrix, which is constant throughout the problem. The nodal displacement vector, u, will vary with time t. F_{ext} is the applied force to the system, which changes with time t, while F_{int} is the internal force of the system, which is determined at time t by both the displacement, u, of each node, and the potential, Φ at each node. In the second equation, our applied nodal charge vector, P_{ext}, which may vary with time t, and our internal nodal charge vector, P_{int}, which varies with both displacement $u(t)$ and electric potential $\Phi(t)$, must be equal.

The simulation uses these governing equations to find the nodal displacements and the nodal electric potentials at each timestep. We will describe the algorithm used by Liang and his implementation of the SMART program in Chapter 2.

1.2 The Computational Environment

All parallel code in this thesis has been developed on an Intel Paragon supercomputer. By Flynn's taxonomy[11], the Paragon is a multiple instruction-multiple data, or MIMD machine, meaning each processor can perform its own instructions, and has its own memory address space. Individual processors cannot see other processors' memories, hence all interprocessor communications must be made via message passing. In the Paragon, processors are connected in a two-dimensional rectangular mesh, meaning each processor has at most four direct connections to other processors. This distributed memory mesh-connected architecture makes it extremely scalable: it can contain from tens to thousands of processors.

Virginia Tech's Paragon has 100 compute nodes, each containing two 50 MHz Intel i860 RISC processors, one of which is used for computation, while the other is used for input and output. We treat each compute node as a single processor, since the Paragon itself handles distribution of tasks between the processors on a single compute node. Each node has 32 MB of memory, but with the operating system resident on each, we can expect to see somewhere between 25 and 30 MB available for our program's use. Nodes can use virtual memory, but access to a storage device like a hard disk will be quite detrimental to the performance of programs. Each processor also has a cache, which we must utilize for optimal performance.

Due to availability problems with the Paragon, all performance results presented in this thesis were computed on a Silicon Graphics Incorporated (SGI) Origin 2000. The Origin, a more modern supercomputer than the Paragon, features a distributed shared memory architecture, meaning the processors all physically have their own memory address space, but the machine can emulate a giant shared address space. This virtual shared memory architecture makes programming parallel codes easier, but often those codes are less efficient and almost always less scalable than codes designed for distributed memory architectures.

The SGI Origin 2000 we used belongs to Boston University and has 64 MIPS R10000 pro-

cessors, each running at 195 MHz. These processors are connected in a hypercube topology, which provides a faster interconnection network than that of the Paragon, but does scale as well as the Paragon's mesh topology. In general, shared memory machines, which often use the hypercube interconnection network, do not get much larger than 256 processors. Each pair of processors on the Origin shares 256 MB of memory, but the code presented in this thesis, which does not utilize the Origin's virtual shared memory, will see the system as a 64-processor distributed memory machine, with each processor owning 128 MB of memory.

For message passing in the code, we use Message Passing Interface, or MPI [13], which is freely available for a wide variety of parallel platforms, including both the Paragon and the Origin. MPI codes will run on any machine with MPI installed, including other supercomputers or networks of workstations. This portability makes our code useful not only on the machines we are using, but on larger, more powerful machines as well.

MPI provides a wide range of message passing calls, of which we use several. The most simple calls, `mpi_send` and `mpi_recv` allow us to send data between pairs of processors. The `mpi_bcast` call allows us to send data from a single processor to a group of processors. The `mpi_reduce` call allows the reduction, with an operation like `mpi_sum` or `mpi_max`, of a single array on several processors to a single processor. Other MPI calls, including `mpi_allreduce`, `mpi_scatter`, and `mpi_gather` will be useful to us in various stages of our parallelization, and we will describe them in later chapters.

1.3 Related Work

Current literature from the fields of computer science, engineering, and numerical analysis reveals that there are many ongoing research efforts related to the parallelization effort presented in this thesis. Due to the wide range of subjects involved, this survey will merely scrape the surface of the work that has been undertaken.

From a computer science perspective, the use of parallel numerical simulations can be broken up into two parts, the first being the parallel environment, which will include both the hardware and software necessary to run the simulation in parallel, and the second being the parallel implementation, which will include the algorithms used and their scalability

and/or performance.

Relating to the parallel environment, it may be useful to consider Abandah's [1] studies on improving communication times within an Origin 2000. Dongarra [9] presents a comparison of the message passing performance of several machines, including the Intel Paragon, and the Cray T3D, a predecessor to the Origin 2000. This aids in a comparison between the two platforms. Concerning the software side of the environment, VanderWiel [23] argues that a parallel compiler like High-Performance Fortran is easy to use, but parallel library codes like ours still perform the best.

Relating to the parallel implementation of the code, Davies [8] considers some methods for modeling parallel performance and scalability. Furthermore, we can consider other parallelization efforts on similar codes, like Crowley's [6] work on a molecular dynamics code, Ahmad's [2] efforts to simulate ocean circulation on a Paragon, or Cwik's [7] work in using the finite element method to model electromagnetic scattering. Even closer to our simulation is Prevost's [21] Dynaflow program, which offers finite element analysis of problems in solid, structural, and fluid mechanics. Furthermore, the code is designed to easily enable parallel solutions of these problems. In studying unstructured mesh algorithms, Jones et al. [17] are working on several applications, including the parallel modeling of piezoelectric crystal behavior. Their work includes mesh generation, smoothing, refinement, and partitioning, as well as parallel implementation of linear system solutions.

There also exist numerous other research efforts on generic finite element codes. Both Carey [5] and Johnsson [16] have researched the migration of general finite elements codes into a parallel environment. This research includes consideration of parallel domain decompositions. Johan et al. [15] went a step farther, considering the scalability of finite element simulations on distributed memory machines.

Chapter 2

The SMART Code

2.1 The Algorithm

The goal of our numerical simulation is, given the forces and charges applied to the smart structure (Figure 1.1), to find both the nodal displacements and the elemental stresses over time. Recall the governing equations:

$$M\ddot{u}(t) = F_{ext}(t) - F_{int}(u(t), \Phi(t)) \tag{2.1}$$

and

$$P_{int}(u(t), \Phi(t)) - P_{ext}(t) = 0, \tag{2.2}$$

where u and Φ are nodal displacement and potential, respectively. (We note that elemental stress is a quantity derived from the nodal forces.) In this section we briefly describe the numerical algorithm for solving Equations (2.1) and (2.2), as implemented in Liang's SMART code.

Given approximate solutions $u_n \cong u(t_n)$ and $\Phi_n \cong \Phi(t_n)$, the basic step of the algorithm seeks to advance these solutions in time by computing u_{n+1} and Φ_{n+1}. To find u_{n+1}, the central difference approximation

$$\ddot{u}_n(t) = \frac{1}{\Delta t^2}(u_{n+1} - 2u_n + u_{n-1}) + O(\Delta t^2) \tag{2.3}$$

is used. If Δt is small, the error term in (2.3) can be ignored, yielding a very close approxi-

mation to $\ddot{u}_n(t)$. Plugging this approximation into Equation (2.1) gives:

$$M(\frac{1}{\Delta t^2}(u_{n+1} - 2u_n + u_{n-1})) = F_{ext}(t_n) - F_{int}(u(t_n), \Phi(t_n)), \tag{2.4}$$

and therefore,

$$u_{n+1} = \Delta t^2 M^{-1}(F_{ext}(t_n) - F_{int}(u_n, \Phi_n)) + 2u_n - u_{n-1}. \tag{2.5}$$

This gives a way to compute the nodal displacement vector at time t_{n+1}, given the timestep, forces, potential, and displacements at time t_n.

With Equation (2.5) allowing us to find u_{n+1}, the nodal electric potential vector Φ_{n+1} must now be found by satisfying Equation (2.2) at time t_{n+1}. Newton's method is used to solve this equation for Φ_{n+1}. Specifically, at the ith step of Newton's Method, we must solve

$$J(u_{n+1}, \Phi_n)\Delta\Phi_{n+1}^{(i)} = -P_{int}(u_{n+1}, \Phi_{n+1}^{(i)}) + P_{ext}(t_{n+1}) - J(u_{n+1}, \Phi_n)\Delta\Phi_{n+1}^{BC} \tag{2.6}$$

for the correction term $\Delta\Phi_{n+1}^{(i)}$, where $i = 1, 2, 3, ...$ until convergence. Here, the Jacobian,

$$J(u_{n+1}, \Phi_n) = \frac{\delta P_{int}}{\delta \Phi}$$

is evaluated at (u_{n+1}, Φ_n), and the last term incorporates the boundary conditions. After we solve this linear system for $\Delta\Phi_{n+1}^{(i)}$, the nodal potential Φ_{n+1} can be updated by:

$$\Phi_{n+1}^{(i+1)} = \Phi_{n+1}^{(i)} + \Delta\Phi_{n+1}^{(i)}, \tag{2.7}$$

where the initial guess is $\Phi_{n+1}^{(0)} = \Phi_n$. Normally, SMART would repeat the solution of Equation (2.6) and the computation in Equation (2.7) until some convergence criterion for Φ_{n+1} was met, but in our particular smart structure, the polarization of our material is a linear function of the electric field; hence only one iteration of Newton's method is necessary.

The following three sections discuss in more detail the implementation of this algorithm in the SMART program. Appendix A contains pseudocode for SMART, which may aid in their reading.

2.2 Input and Preprocessing

The first decision to be made toward solving the problem is how to orient the smart structure in 3-space. We decide that the most narrow dimension of the structure, the width

of the beam, is in the y-direction. The longest dimension, the length of the beam, is in the x-direction, while the direction in which the force is being applied, the height of the beam, is in the z-direction. The axes are shown in Figure 2.1, along with the smart structure.

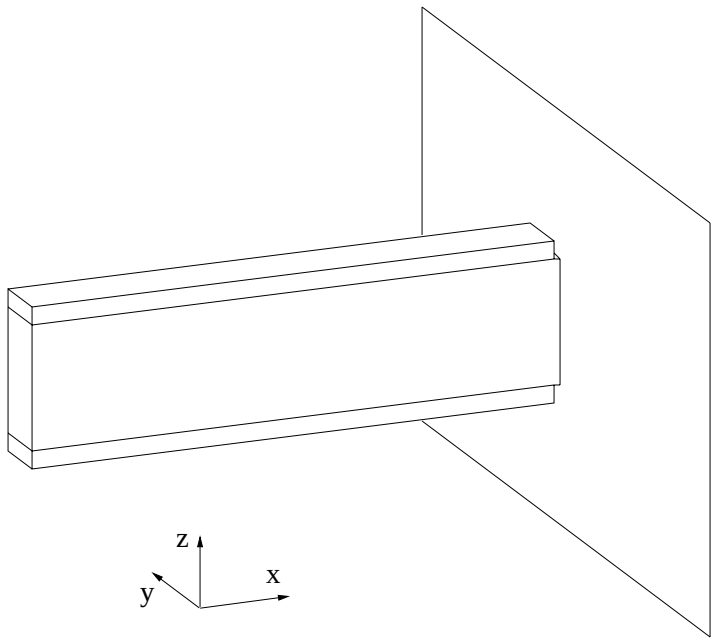

Figure 2.1: The orientation of our smart structure in 3-space.

Next, the structure is discretized with a finite element mesh. It should first be noted that we have no interest in deformations of the structure in the y-direction. Because of this, we will fix the displacements of the structure in y, and the mesh will only be one element thick in y. Our goal with the mesh is to have elements with nearly the same dimensions on all sides, so when we generate the mesh, we will choose the best y dimension we can toward achieving that goal. Furthermore, for the remainder of this thesis, when we refer to a mesh as $m \times n$, we are referring to the x and z dimensions, with the y-dimension assumed to be 1. Note also that an $m \times n$ mesh does not contain exactly mn elements due to the "notch" between the PZT patches and the surface to which the beam is attached.

We discretize the structure with a uniform mesh, meaning all elements are the same size. A 20×20 mesh is pictured in Figure 2.2, while a 50×50 mesh is pictured in Figure 2.3. The four element grey tones in these meshes represent the four types of material in the smart structure: there are three different layers of material making up the composite beam, and

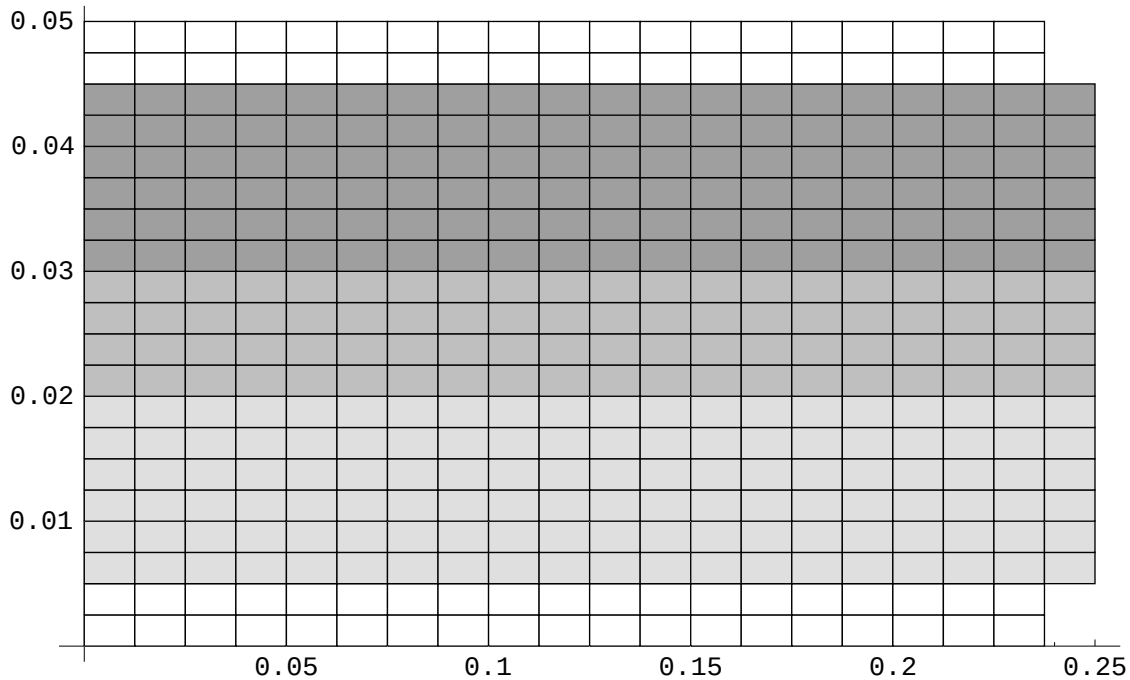

Figure 2.2: A 20×20 mesh representation of the smart structure; different shaded elements represent different material types.

there are layers of PZT material on both the top and bottom of the beam.

To accurately represent the interactions between different materials, we restrict elements in our problem to be of only one material type. As a result, with some meshes we cannot accurately represent the exact locations of the interfaces between material types, which are situated in our example at $z = 0.005$, $z = 0.01833$, $z = 0.03167$, and $z = 0.045$. Most meshes also will not exactly match the dimensions of the smart structure near the notch between the PZT material and the fixed surface. The notch width in the simulation is 0.001, but is clearly much wider in the two example meshes we have presented, since we assume the notch is at least one element long. With both the interface positions and the notch width, finer meshes will more closely represent the exact dimensions of the smart structure.

SMART reads its problem specification from eight input files. In these files, each element gets its own unique number, as do the nodes. Two input files store information about the size and layout of the structure. The node-to-element correspondence file, `ien.in`, contains one line per element, starting with the element's number, followed by the numbers of the

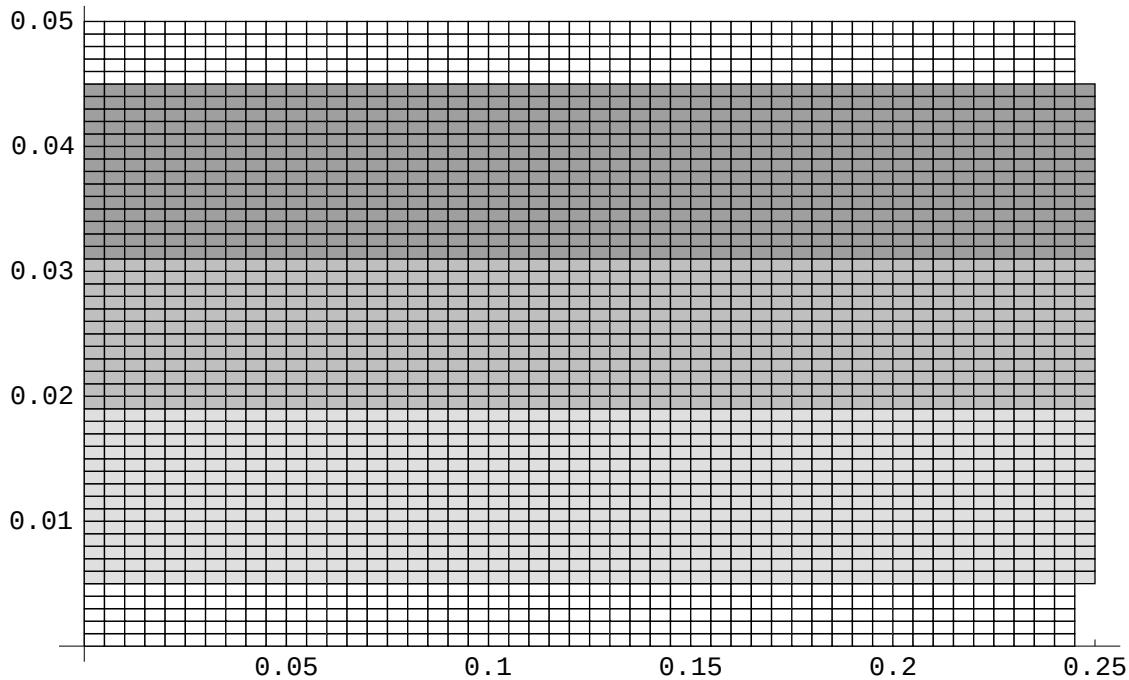

Figure 2.3: A 50×50 mesh representation of the smart structure; different shaded elements represent different material types.

eight nodes which define that element. This file and the `coord.in` file, which contains the x, y, and z coordinates for each node, tell SMART the size and shape of the structure to be simulated.

A third file, `materl.in`, stores a material type for each element in the mesh. For each layer of composite material in the beam, the program needs the orientation angle of that material's fibers. These angles are read in from the `mesh.in` file.

The applied forces to the structure, called the natural boundary conditions, are read in from the `naturbc.in` file. For the simulation studied in this thesis, we apply force only in the positive z-direction, on all the nodes farthest from the fixed surface (which would be the nodes on the left edge of Figures 2.2 and 2.3). The force we apply to the structure is not applied all at once. Instead, we increase it linearly with elapsed time, and then stabilize it at some time. The time at which the force stabilizes is given in the `fstable.in` file.

The essential boundary conditions allow us to state nodes whose displacements or poten-

tial will be fixed at 0 throughout the simulation. We do this in the `esenlbc.in` file. For our simulation, we will fix the y-displacements for all nodes, as well as the x and z displacements on the fixed edge. The notch between the PZT elements and the fixed edge is there to keep the PZT material from touching the fixed surface, so only the non-PZT elements are fixed to that surface. In `esenlbc.in`, we will also fix the electrical potentials for all nodes that are part of composite material elements (including those on the interface between composite material elements and PZT elements). We will refer to these as non-PZT nodes. Additionally, we fix the electrical potentials on the face of the PZT patch nearest to the fixed edge.

The remaining value specified to SMART through an input file is the termination criterion. When a force is applied to the structure, it will begin to move in the direction of that force. As the force waves travel through the structure, they will reflect off of the fixed side of the structure and return to the free side in the opposite direction. This causes the structure to actually move toward the force. Call this one "oscillation". The applied force will continue to push the structure, which will eventually move it back in the original direction. This would be the second oscillation. These oscillations will continue indefinitely—there is no damping applied to the structure. We specify the termination criteria in the `istage.in` file with an integer telling the program how many oscillations to simulate.

For ease in creating these eight input files to SMART, we have written MESH, a mesh generator. Given dimensions of a smart structure and a mesh size, MESH discretizes the smart structure into nodes and elements, creating all the appropriate input files. MESH also allows non-uniformly refined meshes to be generated, if relatively smaller elements are needed near the fixed surface, for example. A description of MESH is presented in Appendix B.

The sequential version of SMART does not have many dependencies on the order in which the input files are read. In fact, the only requirement is that the `mesh.in` file is read before `ien.in`, `materl.in`, and `coord.in`, since `mesh.in` contains the number of nodes, *nnp*, and elements, *nele*, in the mesh, which are necessary in order to read the other three files. Those three files are then read into the $8 \times nele$ array `ien`, the array `materl` of size *nele*, and the array `coord` of size $3 \times nnp$, respectively. We will refer to any array whose size depends exclusively on *nele* as an *elemental* array, while arrays whose size depends exclusively on *nnp* will be called *nodal* arrays. There are no arrays in SMART whose size depends on both *nele* and *nnp*.

There are two other categories of large arrays in the SMART program: *mechanical* arrays and *electrical* arrays. Both have sizes that are closely related to the number of nodes in the mesh. Mechanical arrays contain one value for each mechanical equation in the mesh. The number of mechanical equations is calculated in the `lm_mech` procedure, and is called, *neqmech*. Mechanical equations allow us to determine forces on nodes in each direction that is not restricted by the entries in the `esenlbc.in` file. In our meshes, *neqmech*, will be slightly less than twice the number of nodes, since most nodes have free displacement in the x and z directions. Electrical arrays contain one value for each electrical equation in the problem. The number of electrical equations, *neqelec*, is equal to the number of nodes which have free electrical potential, and is calculated in procedure `lm_elec`. In our mesh, there is an electrical equation for approximately one-fifth of the nodes, corresponding to the proportion of the elements that are PZT-elements.

After file reading is complete, the `lm_mech` procedure is called to number the mechanical equations and create the `idmech` and `lmmech` arrays. Given a node number n ($1 \leq n \leq nnp$), and a direction d ($1 \leq d \leq 3$), `idmech(d,n)` contains the mechanical equation number of that node in that direction. If there is no mechanical equation (i.e., there is an essential boundary condition for that node in that direction), `idmech(d,n)` contains 0. The `lmmech` array also stores mechanical equation numbers, but for elements instead of nodes. Given an element e ($1 \leq e \leq nele$), a node number n for that element ($1 \leq n \leq 8$), and a direction d ($1 \leq d \leq 3$), `lmmech(3(n-1)+d,e)` contains the corresponding mechanical equation number, or 0 if there is no corresponding equation.

The `lm_elec` procedure works in a fashion similar to `lm_mech`, filling the `idelec` and `lmelec` arrays, which are of sizes *nnp* and $8 \times nele$, respectively. There can be only one electrical equation per node, so these arrays are smaller and easier to access.

Next, SMART computes the global mass matrix, `globm`. The mass matrix is stored as a diagonal matrix, which makes its inversion easy. The `globm` array is of size *neqmech*, and will not change for the remainder of the program.

At this point in the code one further calculation is needed to prepare SMART for the simulation, beginning at time $t_0 = 0$. The central difference approximation (2.3) requires the nodal displacements at time $t_{-1} = t_0 - \Delta t$. These displacements are calculated by first finding the internal nodal force at each node at time t_0. Those forces are calculated using

the same procedure as in the main loop, which will be described in the next section. The computed forces are used to find the acceleration of each node at time t_0, which are in turn used to find the displacements at time t_{-1}. Once the displacements for time t_{-1} are calculated in this "priming" step, the main loop of SMART begins.

2.3 The Main Loop

For reasonably fine meshes, the SMART simulation requires a small timestep, and as a result, it requires tens of thousands of iterations to complete. Each iteration is a run through the main loop of the program, where the timestep is added to the current time, and the new approximations u_{n+1} and Φ_{n+1} are computed. This main processing portion of the program is described in this section.

At the beginning of the main loop, the nodal force vector, `globfn` (of size *neqmech*) is initialized to zero. SMART then enters a loop over the elements, from 1 to *nele*, to compute the internal force, $F_{int}(u_n, \Phi_n)$, where u_n is stored in the `displt` array, while Φ_n is stored in the `potent` array. For each element in the mesh, SMART computes the nodal force at all of that element's points. Those computed forces are then added into the `globfn` array, but only for those nodes with free displacement. We call this process of adding computed values into a global array "assembling" to that array. When all the elements have assembled their nodal values into `globfn`, it will contain the correct nodal force for each of the mechanical equations. An instance of this loop, which we will refer to as the nodal force loop, also occurs in the priming portion of code at the end of preprocessing. As was mentioned in Section 2.2, the elemental stresses, which we call σ-values, are computed within this loop as well.

We can now find the new displacements for the free nodes. With all the internal nodal forces computed and stored in `globfn`, SMART uses the `globm` mass matrix, the external nodal forces from the `naturbc.in` file, and the most recent displacements for the free nodes, `dnow` and `dold`, to solve Equation (2.5) for the new displacements, which are stored in `dnew`. All of these arrays are mechanical arrays (as opposed to nodal arrays), because we have not yet considered the fixed-displacement nodes. The `dnow` and `dold` arrays are then updated appropriately.

Next, the `dnew` array of computed displacements and the fixed displacements from the `esenlbc.in` file are combined into the global nodal displacement array, `displt`. The SMART code has been designed to allow the essential boundary conditions, especially the potentials, to change over time. The smart structure we are simulating does not utilize this capability of the code, so the same fixed displacements are copied into `displt` each timestep. This is not an expensive procedure, but it could be improved upon if the need arose. The nodal array `displt` now contains the correct values of u_{n+1}.

At this point, SMART begins to compute the nodal electrical potentials for time t_{n+1}. This process begins with the nodal potential loop, which assembles both the $neqelec \times neqelec$ global electrical Jacobian, $J(u_{n+1}, \Phi_n)$, and the right hand side of Equation (2.6). The nodal potential loop is a loop over only the PZT elements in the mesh. For each PZT element, it assembles to the Jacobian matrix, `globek`, and to the right hand side, `globfe`. The nodal potential loop does no work for non-PZT elements.

Upon exiting the nodal potential loop, SMART solves the linear system in Equation (2.6). For this purpose, the original SMART code used the IMSL routine `DLSLRG`, which solves a dense linear system of equations. Because IMSL was not available, we replaced `DLSLRG` with the equivalent LAPACK routine, `DGESV`. The solution is stored in the `dpoten` array. SMART then combines the computed potentials in `dpoten` with the fixed potentials from the `esenlbc.in` file, storing the combined data in the global nodal potential array `potent`. At this point in the program, both u_{n+1} and Φ_{n+1} are known.

Only two items remain in the main loop before we begin calculations for the next timestep. First, SMART includes a procedure to calculate a more appropriate timestep, given the current state of the simulation. This `timestep` procedure is called once every ten iterations of the main loop, and consists of a computationally intensive loop over the elements, in which a 24×24 eigenvalue problem is solved for each element. This was handled in the original code by the IMSL routine `DGVLRG`, and has been replaced by the LAPACK routine `DGEGV`. The final order of business before rerunning the main loop is to check the termination criteria. If the structure has reached the oscillation limit prescribed by the `istage.in` file, SMART leaves the main loop, outputs the final displacements and elemental stresses, and terminates.

2.4 Sequential Performance

Table 2.1: Running times for different code segments of the main loop of SMART; 100 iterations on a 60×60 mesh.

Segment of code	**Running Time**
Nodal Force Loop	194.6
Central Differencing	0.4
Apply Mech. Essential B.C. (`joint1`)	71.2
Prep. for Nodal Potential Loop	37.1
Nodal Potential Loop	188.0
Linear System Solve	616.7
Apply Elec. Essential B.C. (`joint2`)	46.5
Timestep Calculation	311.2
Other	0.3
Total	**1466.0**

In Table 2.1 it can be seen where the running time is spent while SMART is running the main loop for a 60×60 mesh. It should be noted that all timing data for this program has been compiled using 3 runs of each case. The data presented in this thesis for a given run of the code is actually that of the run with the median of the 3 total running times. The median total times were typically within 5% of the maximum and minimum running times.

Because the number of timesteps required to get an accurate answer is so great, SMART's total running time is approximately equal to the total time spent in the main loop. This immediately leads to the conclusion from Table 2.1 that the most expensive procedure in SMART is the linear system solve, followed by the timestep calculation, followed by the nodal force loop and the nodal potential loop. The `timestep` procedure is especially expensive, since it only runs once every ten iterations of the loop. Of the remaining code segments, the application of the essential boundary conditions proved to be surprisingly expensive. These procedures will be discussed in the next section.

2.5 Sequential Improvements

Before we discuss the potential for parallelism, we should first point out and improve upon some sequential inefficiencies in the original SMART code. A sequential code with improvements made to it can often be more efficient than a parallel version of the original sequential code. This is especially true when the improvements significantly change the time complexity for an algorithm in the code, as our first improvement does. Our second improvement, to the matrix-matrix multiplication procedure, is more subtle, but still improves the performance of the nodal potential loop. Our final two improvements each only speed up a single segment of the code, with the first focusing on the initialization of the `globek` array, while the second looks at loops within the nodal potential loop.

2.5.1 Application of Essential Boundary Conditions

As originally implemented, procedures `lm_mech` and `lm_elec` are very inefficient when there are a large number of essential boundary conditions. The structure we are simulating has more than nnp nodes whose displacements are fixed in at least one direction; the algorithms for handling those boundary conditions need improvement. Whenever the original program searches through the nodes to see where these initial conditions should be applied, it does so with a loop that looks as follows:

```
For each node
  For each initial condition in the list of initial conditions
    If this initial condition is for this node, Then apply it
  End For
End For
```

For a small number of initial conditions, this algorithm is $O(nnp)$, since the inner `For` loop only iterates a small number of times. When there is at least one initial condition for each node, as is the case for the displacements in our smart structure, this algorithm becomes $O(nnp^2)$. This is unacceptable for a large number of nodes, especially when this happens in the main loop of the program, which it does in the `joint1` procedure for mechanical values, and in both procedure `joint2` and within the nodal potential loop for electrical values.

To remedy this situation, we create indexing arrays to tell us, given a node, which boundary condition corresponds to that node. The first array, `index_ndspl`, of size $3 \times nnp$, is created for the mechanical equations in the `lm_mech` procedure. If we wish to see if a boundary condition corresponds with node n in the d direction, we can now look at `index_ndspl(d,n)`, and that array will either point us to the appropriate boundary condition in the list of fixed x-, y-, or z-displacements, or it will contain 0, indicating that node n has free displacement in that direction. The second array, `index_nelec`, of size nnp, is created for the electrical boundary conditions in the `lm_elec` procedure. Both arrays can be created by first initializing them in a loop over the nodes, and then filling them in a loop over the boundary conditions. These new indexing arrays allow us to look at their contents to find an initial condition, instead of searching the entire list of conditions each time. The new search looks like:

```
For each node
  Look at the indexing array data for that node
  If we see that the node has a boundary condition, apply it
End For
```

This search is $O(nnp)$, which saves an enormous amount of time in both preprocessing and the main loop. These savings are shown in Table 2.2.

Table 2.2: Sequential performance improvements due to an improved search for essential boundary conditions; same mesh as in Table 2.1.

Segment of code	$O(nnp^2)$ **Search**	**Improved Search**
Procedure `joint1`	71.2	0.2
Procedure `joint2`	46.5	0.1
Nodal Potential Loop	188.0	141.6

2.5.2 Matrix-Matrix Multiplication

The second improvement in the sequential version of SMART involves the matrix-matrix multiplication procedure, `mul`. The `mul` procedure computes a matrix product using the

code:

```
do i=1,m
   do j=1,k
      c(i,j)=0.0
      do l=1,n
         c(i,j)=c(i,j)+a(i,l)*b(l,j)
      enddo
   enddo
enddo
```

This algorithm multiplies matrices efficiently, but does not consider the additional cost of a cache miss, where the processor must retrieve a set of values from main memory. Despite the fact that the majority of matrix multiplications in SMART involve very small matrices, where cache misses are not an issue, we can still change this algorithm with hopes that the performance improvements in the larger matrix multiplications overshadow any additional cost incurred in the smaller ones. If we change the code to

```
do j=1,k
  do i=1,m
    c(i,j) = 0d0
  enddo
enddo

do j=1,k
  do l=1,n
    do i=1,m
         c(i,j)=c(i,j)+a(i,l)*b(l,j)
    enddo
  enddo
enddo
```

we are performing the exact same computation, but in a different order. Since FORTRAN stores matrices in column-major form, we can take advantage of that fact by making our

most rapid memory accesses in `mul` along columns of the matrices, instead of rows. This change will make the majority of memory accesses in `mul` one memory location away from the previous access, hence increasing the probability that the processor will find that data in its cache. Normally, this optimal memory access order for this operation could be achieved with a call to the BLAS3 [10] operation, `DGEMM`. Since SMART contains so many small matrix multiplications, however, the overhead in starting `DGEMM`, especially in handling its large number of parameters, dominates the procedure's run time and eliminates any benefit it might provide. The improvements from our modified `mul` code can be seen in Table 2.3.

Table 2.3: Performance increases in portions of the main loop due to an improvement in the `mul` procedure; same mesh as Table 2.1.

Segment of code	**Old `mul`**	**New `mul`**
Nodal Force Loop	194.6	187.5
Nodal Potential Loop	141.6	112.5

2.5.3 Initialization of `globek`

Our next significant sequential improvement to the code also involves avoiding cache-misses. The preparation for the nodal potential loop, which simply entails the initialization of the `globek` matrix, originally was performed in two embedded `do` loops:

```
do i = 1, neqelec
  do j = 1, neqelec
    globek(i,j) = 0d0
  enddo
enddo
```

Like the matrix multiplication, this block of code does not utilize the knowledge that FORTRAN stores matrices in column-major form. Interchanging the loops in this code causes SMART to initialize values in `globek` in the same order in which they are stored in memory. On a large matrix like this, this simple change produces an impressive performance improvement. This improvement can be seen in Table 2.4.

Table 2.4: Performance increases in portions of the main loop due to an improvement in the initialization procedure for `globek`; same mesh as Table 2.1.

Segment of code	**Original Code**	**Interchanged Loops**
Prep. for Nodal Potential Loop	37.1	7.8

2.5.4 Loop Invariant Computation

The final improvement to the sequential code effects the `jac3nl` procedure inside the nodal potential loop. In the original SMART code, `jac3nl`, which assembles the `globek` Jacobian matrix, contains many computations that look like the following:

```
      do i = 1, 8
         do j = 1, 8
            ekk(i,j) = ekk(i,j) + 2d0*(epsln1+vnu2*s1+vnu4*s2)*
1                                 wgauss(iint)*vj(iint,iele)*
2                                 wk3(i,j) / epsln0
         enddo
      enddo
```

These loops, which are performed once for each PZT element per iteration of the main loop, are doing an enormous amount of computation. The vast majority of the math, however, is loop invariant, and is being computed redundantly—64 times in this case. If we simply pull out the loop invariant arithmetic, and perform it before the loops, like the following:

```
      temp = 2d0*(epsln1+vnu2*s1+vnu4*s2)*wgauss(iint)*
1            vj(iint,iele)/epsln0
      do i = 1, 8
         do j = 1, 8
            ekk(i,j) = ekk(i,j) + wk3(i,j) * temp
         enddo
      enddo
```

Table 2.5: Performance increases due to moving some loop invariant computation out of loops.

Segment of code	**With Original Loops**	**With Improved Loops**
Nodal Potential Loop	112.5	92.6

Table 2.6: Overall performance increases due to improvements in the sequential code; same mesh as in Table 2.1.

Segment of code	**Original Sequential Code**	**Improved Sequential Code**
Nodal Force Loop	194.6	187.6
Prep. for Nodal Potential Loop	37.1	7.8
Nodal Potential Loop	188.0	92.6
Linear System Solve	616.7	617.8
Timestep Calculation	311.2	316.7
Other	118.4	0.9
Total	**1466.0**	**1223.4**

the code becomes much more efficient. Just in the example, since this occurs within a loop over the 8 integration points, we have saved 1008 additions, 2520 multiplications, and 504 divisions per PZT element. The removal of loop invariant computation from loops like this greatly reduces the amount of computation SMART performs in each iteration of the main loop. The performance increases due to this improvement can be seen in Table 2.5.

These sequential improvements offer us a significant improvement in SMART's performance, without even considering the parallel implementation of this program. The new sequential program performance is shown in Table 2.6.

Chapter 3

The Distribution of Elements

In this chapter, we describe our first effort in parallelizing the SMART program. This first parallel version allows SMART to run in parallel by giving each processor ownership of a distinct group of elements. That processor will be the only processor performing computation for the elements it owns, hence we can divide the computational cost of any loop over elements by the number of processors. This elemental distribution requires some overhead to maintain a correct set of all the nodal values on each processor, in the form of some intra-processor communication calls to MPI.

In Chapter 2, we saw that the linear system was the most time consuming portion of the main loop, yet the parallelization described in this chapter does not involve that portion of the code. Our reasoning for this is threefold. First, our description is somewhat "historical" in that it follows the order in which aspects of the code were parallelized. Secondly, distributing data and work associated with elements is important also—combined, the three loops we parallelize in this chapter cost more sequentially than the linear solve. Furthermore, scalable parallel performance would be impossible without distributing the elements. Finally, the SMART code allows us to compute stresses and displacements in a composite material without PZT layers. When there is no piezoelectric material present, SMART simply skips the nodal potential loop and the linear system solve for finding the potential. This makes parallelism in elemental computations such as the nodal force loop and the `timestep` procedure doubly important. The parallel implementation of SMART presented in this chapter speeds up both of those sections of the main loop. We will focus on the linear system solve

later, in Chapter 4.

3.1 Implementation

In Chapter 2, we saw that the main loop of SMART contains four portions of code which occupy most of its running time: the nodal force loop, the nodal potential loop, the linear system solve, and the timestep calculation. With the exception of the linear system, all of those portions of code are essentially loops over the elements. If the mesh is partitioned into p equally-sized groups of elements, and SMART is run on p processors, with each processor "owning" one of these groups, then SMART can be run in parallel with each processor performing loops over only the elements it owns. Since each processor will only have data pertaining to its "local" elements, they have to communicate with each other in order to maintain correct data from timestep to timestep. We can see the necessary communication by looking more closely at each loop.

The nodal force loop computes, in a loop over the elements, the internal nodal force at each node with free displacement. Within the loop, only nodal and elemental values corresponding with the current element are considered, and none are used after the loop is complete. In other words, for a given processor, the nodal force loop has no dependencies on elements not belonging to that processor, nor will it generate any elemental or nodal values needed by other processors, with one exception. The only data structure altered by the nodal force loop which will be needed later in the program is the nodal force array, `globfn`. This array is assembled by the loop, meaning we can find the correct nodal force for a particular equation by summing the corresponding local `globfn` values on each processor. If all processors are to have a copy of the correct nodal displacement values after the nodal force loop, they must use this process to combine their local copies of `globfn` into the correct global values of `globfn`.

The nodal potential loop is slightly different than the nodal force loop. First, it is a loop over just the PZT elements. This adds to our element partitioning scheme the restriction that PZT elements must now be evenly distributed among the groups of elements. Also, the nodal potential loop assembles two arrays instead of one: both the global electrical Jacobian, `globek`, and the right hand side of the linear system in Equation (2.6), `globfe`. Once again,

for all processors to possess the correct value of `globfe` or row of `globek` for a particular electrical equation, after this loop they can simply take the sum of local values for that equation on all the processors.

The `timestep` procedure serves one purpose. Given the current state of the simulation, it computes a more appropriate timestep, if necessary. For each element, the procedure computes a timestep, and then takes the minimum over all the elements as its final timestep. To run this in parallel, each processor computes a local minimum for its elements, and then a global minimum is selected as the chosen timestep.

In the main loop, when we distribute the elements, it turns out that only the four communications mentioned above are necessary. In the preprocessing stage of the program however, we must prepare the code to run in parallel, which will require several communications. These are discussed in the following section.

3.1.1 Preprocessing and Control Structures

In preparing to run SMART in parallel, the first thing we must do is enable the code to make MPI calls. We start MPI at the beginning of the code with a call to `mpi_init`, and close it at the end of the code with a call to `mpi_finalize`. Additionally, we need some information about the parallel environment created by MPI. After initializing MPI, we call `mpi_comm_rank` to give the process identification numbers, which we store in *myid*. Process IDs are unique integers, ranging from zero to the number of processes minus one. We also call `mpi_comm_size` to find how many processors are running the simulation. This number is stored in the variable *numprocs*. The number of processors equals the number of processes for all of our experiments.

With the information above, we can now distribute the elemental values. Our strategy is simple. Recall that the `mesh.in` file tells exactly how many elements are in the mesh. After reading that file, SMART then calls a procedure that tells each processor how many elements it is responsible for. Assuming we are running SMART with *nele* elements on p processors, this procedure will return a number of elements as close to $\frac{nele}{p}$ as it can. For example, on a three processor simulation on a mesh of 1000 elements, it will return 334 on processor 0, and 333 on the other two processors. Furthermore, when SMART reads in the

elemental input files `ien.in` and `materl.in`, it assumes that processor 0's elements are first, followed by those owned by processor 1, and so on. With this assumed distribution, we are relying on the elements and PZT elements being evenly distributed across the processors. It is the responsibility of the mesh generator to number the elements so that all elements and PZT elements are evenly distributed.

Before the elemental input files are read, SMART creates the `nelecount` array, which contains a count of the number of elements owned by each processor. For instance, `nelecount(1)` would contain the number of elements owned by the first processor, processor 0. The `nelecount` array is created by having each processor store its local number of elements, *nmyele*, in `nelecount(1)`. An `mpi_gather` operation is then performed, which creates the "full" array by storing everyone's values on processor 0 in rank order. Processor 0 then broadcasts a full copy of `nelecount` to all the other processor using `mpi_bcast`.

Now, the elemental input files can be read. If sufficient local memory is available, a straightforward approach is to have processor 0 handle all input and communicate all necessary information to the other processors. For `ien.in`, processor 0 reads the entire file into its local copy of `ien`, and then calls `mpi_scatter`, which distributes the elemental values according to the counts in `nelecount`. Processor 0 will still own a complete copy of `ien`, but it will only access the values for elements that it owns. The `materl.in` file is handled the same manner.

By handling the elemental input files in this fashion, we are letting the `mpi_scatter` operation renumber elements locally on processors. When given an array to distribute and a distribution scheme, like `nelecount`, `mpi_scatter` will distribute values to the first positions of each local array. Processor 0 will have element numbers 1 through `nelecount(1)` numbered locally as elements 1 through *nmyele*, while processor 2 will have elements `nelecount(1)` + 1 through `nelecount(1)` + `nelecount(2)` numbered locally as 1 through *nmyele*. While this local element numbering makes it easier to run the code in parallel, we still must maintain the original global element numbers for future use, namely for output. We store global element numbers for each local element on a processor in the `index_ele` array. A processor can easily find its first global element number by adding up the number of elements owned by processors with lesser rank in the `nelecount` array. That processor's local elements will then have global numbers equal to that sum plus their local numbers.

The remaining input files (e.g., `coord.in`, `esenlbc.in`, `naturbc.in`, etc.) are needed in their entirety by all of the remaining processors. Processor 0 simply reads each file in, just like in the sequential code, and then uses `mpi_bcast` to send a copy of all the input data to the other processors.

At this point, file reading is complete, but there exist some other elemental arrays in the program that have not been initialized yet. It takes a simple change to distribute these elemental arrays. On each processor, the original SMART program built these arrays in loops from 1 to *nele*. In the parallel SMART code, by changing those initialization loops from global elements to local elements (i.e., 1 to *nmyele*), SMART looks at only local elemental data while creating them, hence they will be built as local arrays. This is the case with both the `lm_mech` and `lm_elec` procedures, where SMART creates the `lmmech` and `lmelec` arrays, respectively.

3.1.2 Intraprocessor Communication and Output

Once the elemental arrays have been distributed across the processors, distributing the work in the main loop is a straightforward process. We first change all loops from 1 to *nele* (after the input files have been read) into loops from 1 to *nmyele*. Only five of these loops require communication between the processors.

The first loop, before the main loop of SMART begins, assembles the global mass array, `globm`. This array must be consistent between all processors for SMART to correctly compute displacements in the main loop. Because the array is assembled to on each processor, we can call `mpi_allreduce` with the `mpi_sum` operation to combine the local values, giving each processor the correct global values for `globm`. The second and third loops that require communication are the priming nodal force loop and the main nodal force loop. In both instances, the processors are assembling the global force array `globfn`, hence we can use `mpi_allreduce` with the `mpi_sum` operation, just like we did with `globm`.

The fourth loop, the nodal potential loop, could be handled in the same way as the nodal force loop, since it assembles to both `globek` and `globfe`, but we find that there is a more efficient way to handle this communication. An `mpi_allreduce` call would give each processor a complete copy of the `globek` matrix, which can be a very large data structure.

We find it is more efficient to use calls to `mpi_reduce`, which only brings an entire copy of the linear system to processor 0. Then processor 0 solves the linear system, and broadcasts the solution, using `mpi_bcast`, to the other processors, giving them the nodal potentials as if they had each individually solved the linear system.

The final place in the program where there is a loop over the elements requiring communication is in the `timestep` subroutine, which, as was mentioned earlier, only computes one value: the new timestep. For the reduction after this procedure, we can still use `mpi_allreduce`, but now the `mpi_min` function is the appropriate way to combine the data. After this call, each processor will have a copy of the minimum computed global timestep, which is exactly what the sequential code computed.

With these loops over the elements all taken care of, SMART runs correctly with the elemental values distributed across the processors. The only remaining concern is the output of values, namely the computed σ-values and nodal displacements. All output is handled by processor 0. Writing nodal arrays, like the displacements, is easy, since processor 0, like all the others, has a complete copy of the correct nodal values. Processor 0 simply writes its values to file and ignores the other processors, who all have the same nodal data in their arrays. Elemental output requires some additional communication, though. To output the elemental σ-values, we create a procedure called `write_sigma`, which outputs data in rank order of processors. The `write_sigma` procedure first outputs σ-values for the elements belonging to processor 0, in a format where each element's values are preceded by their global element number from `index_ele`. These global element numbers will be needed for post-processing purposes. The procedure then, in a loop over the processors, passes the σ-values from each processor to processor 0, along with that processor's local copy of `index_ele`, which processor 0 then uses to output those σ-values with the appropriate global element numbers.

3.1.3 Reducing Elemental Storage Requirements

In the previous section, for the parallel implementation of SMART, we assumed that "sufficient local memory" was available. When the amount of local memory is a concern, this implementation can be made more memory efficient. For the majority of execution, the

elemental values are completely distributed across the processors, but the `ien` and `materl` arrays are still held in their entirety by processor 0 while it distributes them. This requires storage space on processor 0, and therefore all processors, for $O(nele)$ values (our parallel implementation is SPMD: single program, multiple data). By changing the file input procedure so that no elemental array is ever contained entirely on any single processor, we reduce the total size needed to store the elemental values on each processor to $O(nmyele)$ values. This divides the total per processor memory requirements for elemental arrays by the number of processors, p.

Our assumed distribution of elements makes this process easier than it might have otherwise been. Given the value of *nele* that is input from the `mesh.in` file, and the value of *numprocs* MPI has given us, we know how the elements will be distributed. We will simply use a process similar to `write_sigma` to load the elemental data directly into a distributed environment. The trick to do this while avoiding multiple reads of the input file, and not requiring extra memory for the communication, is to not let processor 0 do the reading. Instead, the processor of highest rank reads the elemental values for each other processor into its local arrays, using the `nelecount` array to find the exact number of elements that each owns. This last processor then sends that information to each processor in turn, saving the last group of elemental values for itself. No extra space is required, because the processor uses its own arrays to store and distribute everyone else's array values before it reads its local values for those arrays. We implement this method of reading elemental arrays directly into the distributed environment in procedures `read_ien` and `read_materl`.

3.2 Performance

At a glance, the speedup numbers in Table 3.1 don't look very good. Essentially half of the main loop is running in parallel, so we can never expect a speedup better than 2. The performance of the three parallel loops, however, is quite promising. For this mesh, all three loops have near perfect speedup. We know the nodal potential loop must assemble to `globek`, and therefore has some dependence on the size of the linear system, but that does not show for a mesh of this size. With the dominating cost of the `globek` reduction and the linear system solution in this parallel code, no significant improvement can be expected

Table 3.1: Running times for blocks of code in the main loop of the distributed element version of SMART; run on a 60×60 mesh.

Segment of code	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Nodal Force Loop	189.8	94.9	47.5	23.9	11.9	6.0	3.0
AllReduce of `globfn`	0.1	0.3	0.6	0.8	1.1	1.3	1.8
Prep. for Nodal Potential Loop	11.9	10.2	10.2	10.2	10.5	10.5	11.4
Nodal Potential Loop	94.1	47.1	23.6	12.0	5.6	2.5	1.3
Reduce of `globek`	10.4	40.6	75.1	114.7	147.3	190.5	268.8
Reduce of `globfe`	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Linear System Solve	615.7	614.2	614.8	614.0	614.9	614.6	616.3
Timestep Calculation	317.0	158.7	79.6	40.0	20.2	10.1	5.2
Other	1.1	1.3	1.2	1.3	1.3	1.3	1.6
Total	**1240.2**	**967.4**	**852.8**	**817.0**	**812.9**	**836.9**	**909.5**
Speedup	**1.00**	**1.28**	**1.45**	**1.52**	**1.53**	**1.48**	**1.36**
Efficiency	**1.00**	**0.64**	**0.36**	**0.19**	**0.10**	**0.05**	**0.02**

without first considering these segments of the main loop.

Chapter 4

The Linear System Solve

4.1 Concerns

If our parallel version of SMART is to solve problems containing piezoelectric elements, significant attention must be paid to the linear system solve that is performed upon completion of the nodal potential loop. The solution of this system,

$$[\texttt{globek}]\{\texttt{dpoten}\} = \{\texttt{globfe}\} \tag{4.1}$$

causes both performance and space problems. The global Jacobian matrix, `globek`, is by far the largest data structure in the program. Furthermore, when the nodal potential loop completes, the matrix must be assembled from all the processors to the processor that is responsible for solving the linear system. This communication, as well as the actual solution of the linear system, become a significant part of the program's running time.

The space required to store the `globek` matrix is the most pressing issue when first solving problems containing piezoelectric elements. The `globek` matrix is the only multidimensional data structure where both dimensions depend linearly on the problem size. Even with a uniform mesh simulation, where the ratio of PZT elements to the total number of elements is less than it would be for a non-uniform mesh (with refinements in the PZT region), it only takes a relatively small mesh for `globek` to dominate the program's memory usage. Figure 4.1 shows that `globek` is too big for us to perform the simulation on a reasonably-sized mesh. On the Paragon, the largest uniform-mesh simulation we can run, while fitting

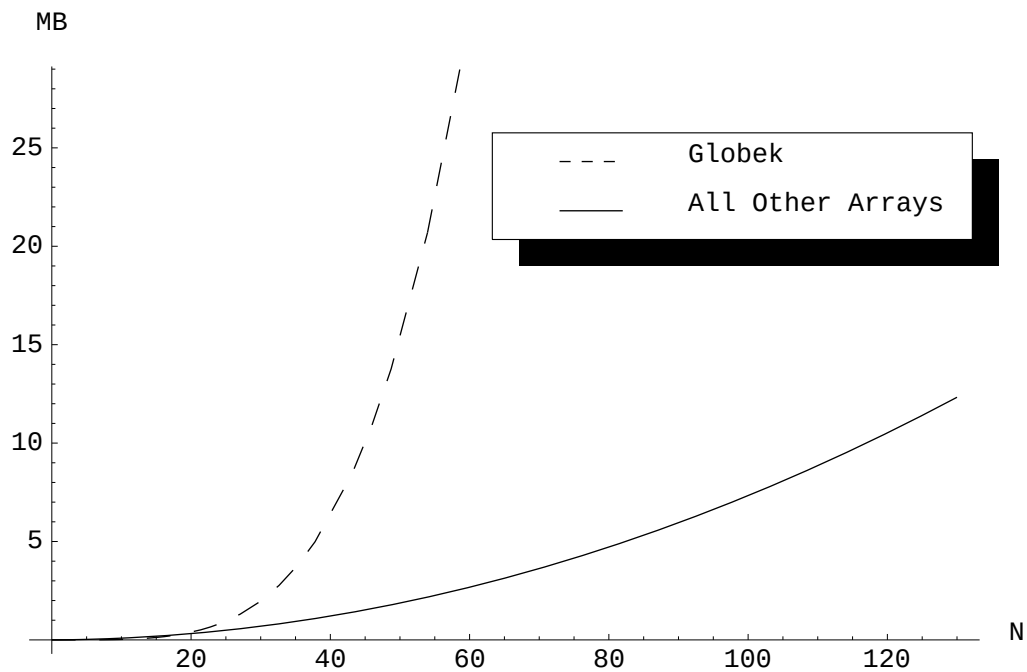

Figure 4.1: Memory required for the `globek` array vs. all the other arrays in the program combined; 64 processors running on an $N \times N$ uniform mesh. Only the elemental values are distributed.

under the 30MB per processor target, is only 60×60. In order to use the program for larger meshes, we must somehow reduce the size of this matrix. It turns out that we can.

The contents of the `globek` matrix are generated within the nodal force loop. Each time through the loop, each processor is looking at one of its PZT elements, and fills an 8×8 'electric stiffness' matrix, `ekk`, which it then assembles to the `globek` matrix. Each of the 64 values in `ekk` corresponds to a relationship between one of that element's nodes and another of its nodes. Figure 4.2 contains pseudocode for the procedure that each PZT element uses to assemble the `ekk` matrix to `globek`.

The assembly code is only assigning a value to a node's row in `globek` if that value corresponds to another node in the current PZT element. We can look at that from a different perspective and say that there will be at most one value assigned to a node's row in `globek` for each node with which it shares a PZT element (including the node itself). Since a node can be part of at most eight elements, we can reason that a node can share

```
For i = 1 to 8 (each of my nodes)
    If (my node i is a free potential node) Then
        ip = the electrical equation number for i
        For j = 1 to 8 (each of my nodes)
            If (my node j is a free potential node) Then
                iq = the electrical equation number that j corresponds to
                globek(ip,iq) = globek(ip,iq) + ekk(i,j)
            End If
        End For
    End If
End For
```

Figure 4.2: Pseudocode for the dense `globek` matrix assembly

PZT elements with at most 27 different nodes. Since `globek` is initialized to zero before we enter the nodal potential loop, there can be at most 27 values per row of the matrix. Hence, the `globek` matrix is sparse. Consider the 60×60 mesh used in Table 4.1. When we run SMART on that mesh, *neqelec* $= 1416$, so `globek` will be using $1416 \times 1416 = 2005056$ values (15 MB) to store at most $1416 \times 27 = 38232$ nonzero floating point values (299 KB). This is a small mesh, and we are already wasting almost 15 MB in storing `globek`. This wasted memory will grow quadratically as *neqelec* grows.

Table 4.1: Time for 100 iterations of the main loop with dense storage of `globek`; 60×60 mesh on 32 processors; only the elemental values are distributed.

Segment of code	**Run Time**
Nodal Force Loop	6.0
AllReduce of `globfn`	1.3
Prep. for Nodal Potential Loop	10.5
Nodal Potential Loop	2.5
Reduce of `globek`	190.5
Reduce of `globfe`	0.1
Linear System Solve	614.6
Timestep Calculation	10.1
Other	1.3

For every bit of memory we can save in storing the `globek` matrix, we will also save time in the reduction of `globek` after the nodal potential loop. As is clear from Table 4.1, on 32 processors, the communication of that matrix to one processor for the linear system solve is by far the most expensive communication in the program. In fact, the only piece of this parallel code that is more expensive than the communication is the linear system solve on that matrix. This, too, could be reduced if we could somehow utilize the sparsity of `globek`.

4.2 Improvements

When we look more closely at `globek`, two factors suggest possibilities for saving enormous amounts of time and space. First, `globek` turns out to be a symmetric positive definite matrix. Secondly, and even more importantly, we can look more closely at the mesh we are using, and see that `globek` is banded as well. In this section we describe two improvements in the efficiency with which the system (4.1) can be solved. We quantify these improvements by reporting performance on the same case as used in Table 4.1, i.e., 60×60 mesh on 32 processors.

4.2.1 Cholesky Factorization

Because `globek` is symmetric positive definite, we can make two improvements. First, as far as space is concerned, we now only need to store half of the matrix. This improvement does not help much, though, since we still must store our data in the lower triangle of the original square matrix. The payoff, however, is on the performance side, because we can now use a Cholesky factorization to solve the linear system. The Cholesky method has a time complexity of $O(n^3/6)$ compared to the $O(n^3/3)$ complexity of general Gaussian Elimination [12], which we have been using. Furthermore, we only need to worry about assembling to the lower triangle of the matrix, and that will save us some time in the nodal potential loop as well.

The implementation of the Cholesky factorization itself is easy; we simply replace the standard LAPACK calls to `DGESV` with the LAPACK call to `DPOTRF`, which computes the Cholesky factorization, and `DPOTRS`, which solves the system using that factorization.

The difficulty comes after the new linear system solve is in place. We would like to avoid performing any computations that will result in `globek` values above the diagonal. The matrix assembly code in Figure 4.2 can be changed, since we only need to add values of `ekk` into `globek` if they are in the lower triangle; hence, the update of `globek(ip,iq)` in the inner loop can appear as:

```
If (iq ≤ ip) Then
     globek(ip,iq) = globek(ip,iq) + ekk(i,j)
End If
```

This may not seem like much of an improvement, but keep in mind that the `If` statement will be true only about half of the time. This means that, about $64npzt$ times, where $npzt$ is the number of PZT elements, we are replacing two array references and an addition with a comparison between two values that are most likely still sitting in registers. Unfortunately, due to some matrix-matrix multiplications immediately before this procedure is called, it is not possible to exploit the symmetry of `globek` any further, i.e., in computing `ekk`. A performance comparison of the dense Gaussian elimination procedure vs. the Cholesky factorization procedure is given in Table 4.2.

Table 4.2: Performance improvements by migrating to Cholesky factorization; same mesh as in Table 4.1 on page 35.

Segment of Code	**Gauss**	**Cholesky**
Prep. for Nodal Potential Loop	10.5	5.7
Linear System Solve	614.6	425.1

4.2.2 Banded Cholesky Solve

If we look back at the code in Figure 4.2, and this time consider the mesh we are dealing with, we can gain some more insight into the structure of `globek`. Recall that SMART chooses the electrical equation numbering scheme in the `lm_elec` procedure. The equation numbering is in the same order as the nodes are numbered, so the electrical equations are numbered most rapidly in z, then in y, and finally x. There is an electrical equation for only

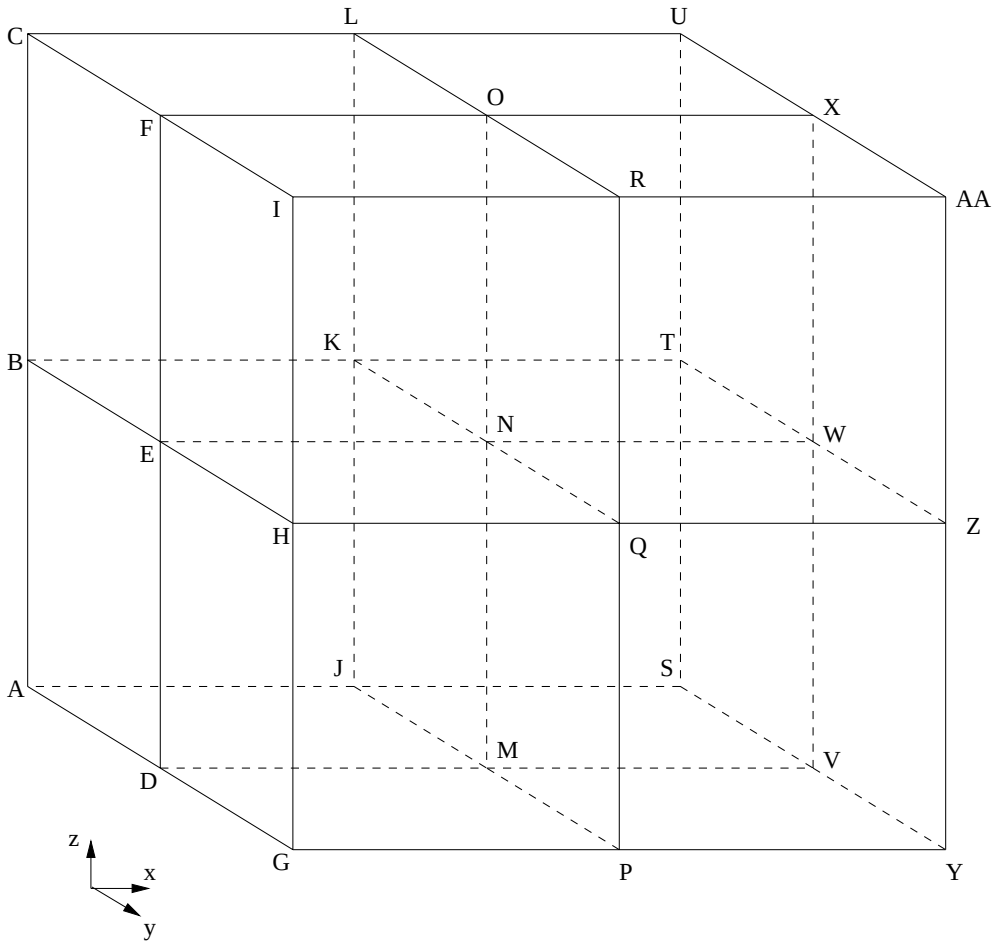

Figure 4.3: Nodes in a typical 3-D mesh

the nodes with free electric potential, so, given a node with free potential, the maximum difference between the equation number of that node and those of its neighbors (with free potential) is bounded by the maximum difference between that node number and the node numbers of its neighbors.

Consider Figure 4.3 and Table 4.3. Assume that some node N corresponds to electrical equation i, and hence row i of `globek`. The farthest a neighboring node's equation number can get from i is $(ny)(nz) + nz + 1$, and we know that `globek(i,i)` must contain a value[1]; hence `globek` is banded with a maximum bandwidth of $(ny)(nz) + nz + 1$.

[1]This can be verified by looking at the `globek` assembly code in Figure 4.2

Table 4.3: Distances in node numbers from node N in Figure 4.3; ny and nz are the number of nodes in the mesh in the y and z directions, respectively; nodes with numbers greater than that of N have been omitted because of symmetry.

Node	**Distance**	**Node**	**Distance**
A	$(ny)(nz) + nz + 1$	J	$nz + 1$
B	$(ny)(nz) + nz$	K	nz
C	$(ny)(nz) + nz - 1$	L	$nz - 1$
D	$(ny)(nz) + 1$	M	1
E	$(ny)(nz)$	N	0
F	$(ny)(nz) - 1$		

The situation is even better in the meshes we have been working with, since ny is always 2. Furthermore, the only elements that can possibly contain free potential nodes are PZT elements, so we can ignore substrate elements. In our problems, only the top tenth and bottom tenth of the beam (in the z-direction) is PZT material, meaning we only have to worry about one-fifth of the elements in the z-direction. This gives a bandwidth for our problem,

$$bandwidth = \frac{3}{5}nz + 1. \tag{4.2}$$

The LAPACK banded Cholesky factorization requires use of a specific banded matrix storage scheme. This scheme reduces the size of `globek` to $bw \times neqelec$ where bw is the bandwidth of the matrix (including the diagonal). The best way to describe this scheme is with an example. Consider storing a 6×6 symmetric banded matrix with 2 nonzero subdiagonals:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

Using the LAPACK band storage scheme, the matrix A is stored as:

$$ABD = \begin{bmatrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$$

where the elements of ABD denoted by '*' are locations which are not referenced by LAPACK.

We can convert to this banded storage scheme quite easily, by changing the `globek` assembly algorithm to the algorithm presented in Figure 4.4.

```
For i = 1 to 8 (each of my nodes)
    If (my node i is a free potential node) Then
        ip = the electrical equation number for i
        For j = 1 to 8 (each of my nodes)
            If (my node j is a free potential node) Then
                iq = the electrical equation number that j corresponds to
                If (iq ≤ ip) Then
                    k = ip - iq + 1
                    globek(k,iq) = globek(k,iq) + ekk(i,j)
                End If
            End If
        End For
    End If
End For
```

Figure 4.4: Pseudocode for the band Cholesky `globek` matrix assembly

Now, we can replace the LAPACK dense matrix calls to `DPOTRF` and `DPOTRS` with the banded matrix calls to `DPBTRF` and `DPBTRS`. The time complexity for the solution of banded $n \times n$ system using the Cholesky method is $O(nb^2)$, where b is the bandwidth of the system. As can be seen in Table 4.4, this gives us a significant performance increase. In addition, we have reduced the number of values stored in `globek` from $neqelec^2$ to $(\frac{3}{5}nz + 1)neqelec$. This savings will be discussed more in the next section.

Table 4.4: Performance improvements by storing `globek` in banded form; same mesh as in Table 4.1 on page 35.

Segment of Code	**Dense Cholesky**	**Band Cholesky**
Prep for Nodal Potential Loop	5.7	0.3
Reduce of `globek`	190.5	5.5
Linear System Solve	425.1	2.5

4.2.3 Reducing the Bandwidth

Switching to a banded matrix storage greatly improved all three sections of code influenced by the size of `globek`, but there is more that we can do. If we focus more on the bandwidth, two more improvements become apparent. The first deals with our node numbering scheme, and the second deals with the nature of the engineering problem that we are interested in.

The first change may be apparent from Equation (4.2). In our node numbering scheme, we number most rapidly in the z-direction, then in the y-direction, and then in the x-direction. It turns out that the bandwidth of `globek` will always depend on the two most rapidly changing dimensions of the mesh, with the quadratic term being a product of those two dimensions, and the linear term being the most rapidly changing dimension. If we change the order in which we number the nodes by swapping y and z dimensions, so that y is the most rapidly changing dimension, it does not change the quadratic term in the bandwidth; but the linear term is now ny, which, for our meshes will always be the smallest dimension. This gives us a new bandwidth equation,

$$bandwidth = \frac{2}{5}nz + 3 \tag{4.3}$$

which will lead to a slight improvement in space and time as our mesh grows.

Another, more significant improvement involves the setup of our problem. Recall that in our engineering problem the mesh has two distinct PZT patches. Since equations in the linear system only involve neighboring elements within the same PZT patch, distinct patches will allow us to decouple the system. There are two ways we can take advantage of this.

First, regardless of our method, we have to create some way to tell SMART what PZT elements are in which patch. We accomplish this by changing the format of the `materl.in`

file. Now, instead of reading in 0 as the material number for a PZT element, SMART reads in a negative number. Any element with a negative material number is a PZT element, and the magnitude of the number indicates the PZT patch number. If the material number is zero, SMART will run, but will not utilize this optimization.

The first way in which we exploit our "2-patch" problem is to change the order in which we number the electrical equations. Instead of numbering all of the equations by node order, we number them first by their PZT patch number, and then by node order. This method reduces the bandwidth of `globek` so that it no longer depends on the height (in the z-dimension) of both PZT layers, but instead on the height of just one, i.e., the bandwidth is cut in half.

The second benefit of exploiting the distinct PZT patches only applies when we have two or more processors. The idea is to decouple the linear system into two problems—one for each patch—which can then be solved in parallel. We introduce a pre-processing step (before SMART is even run) that generates a domain decomposition for the problem such that no processor owns elements in more than one PZT patch. This preprocessing step is described in Appendix B. For any node (PZT or not) that is not in a processor's PZT patch, that processor assumes the potential for that node is fixed at 0. Hence, processors will each calculate their own local value for *neqelec*.

Each processor must then join a group of processors based on the patch that they participate in. This can be implemented in MPI using communicators. A communicator can be thought of as a partitioning of the processors. Until now, all communications have been over the *mpi_comm_world* communicator, a single partition, containing all the processors. The new communicator, *mpi_comm_patch*, will contain two partitions, one with processors containing the first PZT patch, and one with processors containing the second PZT patch. Within each partition, processors get their own rank. This is convenient, because we can do things exactly the way we have been, only now on the *mpi_comm_patch* communicator. All processors will still reduce their arrays to a root node (processor 0), only now there is a processor 0 for each patch. Then each processor 0 will solve their linear system sequentially, and broadcast the results back to all the processors in their patch.

Getting the correct results to all the processors is a bit trickier (recall that each processor still needs a complete copy of all nodal arrays, such as `potent`). Each root node builds its

local version of the nodal array `potent` in the `joint2` procedure. The root nodes of each patch then do an `mpi_allreduce` on `potent` to get the correct global value for that array. To do this, we need another MPI Communicator, *mpi_comm_root*, which groups the root nodes together and leaves all other processors by themselves. Once the roots each have the correct values of `potent`, they can simply broadcast the values back to the rest of their respective *mpi_comm_patch* communicators. An example of processor ranks in different communicators for SMART running on 6 processors is given in Table 4.5.

Table 4.5: Processor ranks in different communicators for a 6 processor run: P_0, P_2, and P_4 own PZT patch 1; P_1, P_3, and P_5 own PZT patch 2.

Communicator	**Partitioning**	P_0	P_1	P_2	P_3	P_4	P_5
mpi_comm_world	$\{P_0, P_1, P_2, P_3, P_4, P_5\}$	0	1	2	3	4	5
mpi_comm_patch	$\{\{P_0, P_2, P_4\}, \{P_1, P_3, P_5\}\}$	0	0	1	1	2	2
mpi_comm_root	$\{\{P_0, P_1\}, \{P_2\}, \{P_3\}, \{P_4\}, \{P_5\}\}$	0	1	0	0	0	0

This patch decoupling optimization will reduce the size of *neqelec* to the maximum of *neqelec* over all the patches. In our case, since we have two equally sized patches, it divides *neqelec* in half. It also divides the bandwidth of `globek` in half, meaning

$$bandwidth = \frac{1}{5}nz + 3. \tag{4.4}$$

Our improved code therefore requires one-fourth the space to store `globek`. This smaller matrix gives us an impressive improvement in both the communication of `globek`, and the linear system solve. Table 4.6 provides profiling information after both of the improvements in this section were made. Most of the improvement seen in Table 4.6 is due to decoupling the linear systems and solving them in parallel; the bandwidth reduction due to changing the node numbering is relatively small for this small problem.

Table 4.6: Performance improvements by decoupling the linear system and reducing the bandwidth of `globek`; same mesh as in Table 4.1 on page 35.

Segment of Code	**Band Cholesky**	**Decoupled Band Cholesky**
Prep. for Nodal Potential Loop	0.3	0.1
Reduce of `globek`	5.5	4.4
Linear System Solve	2.5	0.6

Chapter 5

Distribution of Nodes

5.1 Motivation

By distributing the elemental arrays, we significantly reduced SMART's memory requirements. Since the nodal values referenced and written to by a processor depend directly on that processor's elements, we can now distribute the nodal values accordingly, which should offer us yet another major improvement in per processor memory usage. Due to several loops over nodal values, memory access issues, and a reduced amount of global communication, this improvement can give us a performance increase as well.

While nodal distribution would, if efficiently implemented, reduce the memory needed by the code, we should first verify that the savings is necessary, and will be significant enough to overshadow the cost of implementation. We can estimate that savings by approximating the total space needed by the program. To do so, we must consider all of the significant arrays used by the program. By significant, we mean arrays whose size vary with the size of the mesh. Recall that in the SMART code, there are four varieties of array:

- **Elemental** - Arrays over elemental values (*nele* is a dimension).
- **Nodal** - Arrays over nodal values (*nnp* is a dimension).
- **Mechanical** - Arrays over each mechanical equation (*neqmech* is a dimension).
- **Electrical** - Arrays over each electrical equation (*neqelec* is a dimension).

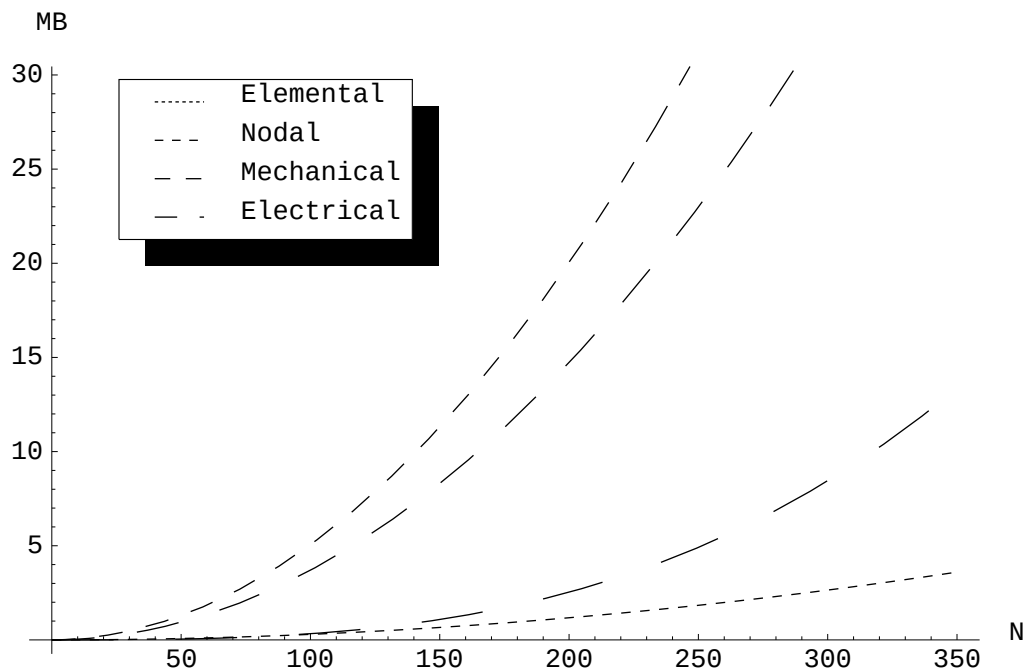

Figure 5.1: Memory required per processor for each of the four array types when only the elements are distributed; 64 processors running on an $N \times N$ uniform PZT mesh.

Summing the total space required by these four types of arrays (in bytes) we come up with the equation:

$$\text{Total Memory} = 1972nele + 260nnp + 96neqmech + (16b + 24)neqelec, \tag{5.1}$$

where b is 1 plus the half-bandwidth of the matrix `globek`. Next, we must consider the fact that the elemental arrays have already been distributed. In fact, since those values have been distributed perfectly, we can simply alter Equation (5.1) by dividing the elemental term by the number of processors, p.

$$\text{Total Memory} = \frac{1972nele}{p} + 260nnp + 96neqmech + (16b + 24)neqelec.$$

With this new equation, we still cannot tell by inspection which term is dominant. When we graph the memory requirements for square meshes in Figure 5.1, though, it becomes apparent where the memory bottleneck is. The nodal arrays are taking the most space, followed by the mechanical values.

We should be concerned about the growth rate of the electrical values, though, since they may be growing faster than the mechanical values due to the b term corresponding to the band storage of `globek`. In our engineering problem, recall from Equation (4.4) that

$$b \approx \frac{nz}{5},$$

where nz is the number of elements in the z direction. We can also come up with approximates for $nele$, nnp, $neqmech$, and $neqelec$ as a function of nx and nz:

$$nele \approx (nx)(nz)$$

$$nnp \approx 2(nx)(nz)$$

$$neqmech \approx 4(nx)(nz)$$

$$neqelec \approx \frac{(nx)(nz)}{5}.$$

Using these approximations, our total per processor space requirement is:

$$\begin{aligned}
\text{Total Memory} &\approx \frac{1972(nx)(nz)}{p} + 520(nx)(nz) + 384(nx)(nz) + (\frac{16nz}{5} + 24)\frac{(nx)(nz)}{5} \\
&\approx \frac{1972(nx)(nz)}{p} + 520(nx)(nz) + 384(nx)(nz) + \frac{3}{5}(nx)(nz)^2 + 5(nx)(nz) \\
&\approx \left[909 + \frac{1972}{p}\right](nx)(nz) + \frac{3}{5}(nx)(nz)^2.
\end{aligned}$$

The above estimate implies that the space required by the electrical arrays grows quadratically with respect to the number of elements in the z-direction, while the rest of the arrays grow linearly with respect to the number of elements in the z-direction. As the z-dimension of the mesh grows, the size of the electrical arrays will grow faster than the other types of arrays. The point at which they pass the other types however, is when the y-dimension is greater than about 1500. This is much larger than the meshes we can currently run SMART on, so we will ignore the growth rate for the remainder of this discussion.

In the meshes SMART can currently run on, it can be seen in Figure 5.1 that the nodal values are using the most memory. It is also important to note that the mechanical arrays

use the second most. Because of their close relationship to the nodal values—there are two mechanical values for almost every node—the mechanical values will be distributed automatically if we distribute the nodes. Unfortunately, if we continue to solve the linear system sequentially, the electrical arrays must stay intact on at least one processor, hence their distribution would save us no memory.

Assuming we distribute both the nodal and mechanical arrays, we can use Equation (5.1) to approximate the new memory usage. To keep our approximation realistic, let's assume that both array types are not distributed perfectly. Instead, we will say that processors require twice as many nodes as they would in a perfect distribution. This assumption should more than account for the redundancies in nodal storage, as well as the additional data structures required for implementation. With this pessimistic assumption, our estimated memory requirements for nodal distribution are:

$$\text{Total Memory} = \frac{1972nele}{p} + \frac{2}{p}260nnp + \frac{2}{p}96neqmech + (16b + 24)neqelec.$$

In terms of the mesh parameters nx and nz we get:

$$\begin{aligned}
\text{Total Memory} &\approx \frac{1972(nx)(nz)}{p} + \frac{2}{p}520(nx)(nz) + \frac{2}{p}384(nx)(nz) + (\frac{16nz}{5} + 24)\frac{(ny)(nz)}{5} \\
&\approx \frac{3780(nx)(nz)}{p} + \frac{3}{5} + (nx)(nz)^2 + 5(ny)(nz) \\
&\approx (\frac{3780}{p} + 5)(ny)(nz) + \frac{3}{5}(nx)(nz)^2.
\end{aligned}$$

From this pessimistic estimate, the largest uniform mesh we could run the new simulation on (using 64 processors and 30MB per processor) would be 340×340, which is significantly larger than the 200x200 mesh we could use before. Clearly the potential savings from nodal distribution make it worth our while to implement.

From a performance standpoint, the distribution of nodes can also be beneficial. There are several loops in the code over *neqmech* which would now be over local values of *neqmech*. The same holds for any loop over the essential boundary conditions in x, y, and z, as well as for the natural boundary conditions. These loops are spread about the program, and will give us small improvements in any part of the code that contains them. We should also expect an impressive performance enhancement in what was the `mpi_allreduce` call we make after the nodal force loop. Each processor was originally making that call to share its

copy of the `globfn` array (of size *neqmech*) with the world. Now, though, they will only be sending and receiving values of `globfn` for the nodes they share with other elements.

Table 5.1: Profiling information before nodal distribution for the code running on a 60×60 mesh where $nele = 3588$, $nnp = 7418$, $neqmech = 14640$, $neqelec = 1416$, and $b = 19$. All totals are from 100 iterations of the main loop.

Segment of code	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Nodal Force Loop	189.7	94.6	47.6	24.3	11.1	4.8	1.7
AllReduce of `globfn`	0.1	0.4	0.6	1.5	1.7	1.9	3.2
Nodal Potential Loop	92.7	46.3	23.6	12.6	6.0	3.0	3.1
Reduce of `globek`	0.2	0.1	0.2	0.7	0.5	0.7	1.9
Reduce of `globfe`	0.0	0.0	0.0	0.1	0.1	0.1	0.1
Linear System Solve	1.2	0.6	0.6	0.6	0.6	0.6	0.6
Timestep Calculation	316.1	158.2	79.3	40.4	21.2	10.6	5.6
Other	1.1	1.0	0.9	2.1	1.0	1.2	1.4
Total	**601.1**	**301.2**	**152.8**	**82.3**	**42.2**	**22.9**	**17.6**
Speedup	**1.00**	**2.00**	**3.93**	**7.30**	**14.24**	**26.25**	**34.15**
Efficiency	**1.00**	**1.00**	**0.98**	**0.91**	**0.89**	**0.82**	**0.53**

Table 5.1 gives profiling information for a run of the code before distribution of the nodes. We can use this data to see which segments of code are affected by the distribution of nodal and mechanical arrays. Note that the data in Table 5.1 reflects the Chapter 4 improvements to the linear solve, and hence, the parallel speedup is much better than in Table 3.1 on page 32.

5.2 Implementation

We can break the distribution of nodes across the processors down into three parts. First, in the preprocessing stage of the program, we must use the data we get from the input files to devise a node storage scheme such that processors are only getting the nodes that are part of their elements. This will involve creating the data structures to make the next two parts feasible, and furthermore, efficient. Secondly, once the data is properly distributed,

we must verify that the main loop has the proper communications so that it may function properly. Finally, assuming the data is distributed properly, and the main loop is generating the correct values, we must be able to output nodal values in an organized manner.

5.2.1 Input and Preprocessing

In order to perform the input and preprocessing efficiently, we must think carefully about the order in which we read our input files. We would like to read in each file only once. Since we need the elemental values to determine where the nodes should be read, it makes sense to read them first. We therefore start with the `parallel.in` file, which tells us which processors own which elements. From there, we can read in the remainder of the elemental files, `materl.in` and `ien.in`.

At this point, each processor knows, from their copy of `ien`, which nodes they need. Recall that `ien(i,j)` holds the global node number of the jth corner of element i, where $1 \leq i \leq nnp$ and $1 \leq j \leq 8$. Each processor then creates a sorted list of the global node numbers that they require. We accomplish this by running the quicksort algorithm on each of the 8 columns of `ien`, and then merging those sorted lists. To merge, we call our `merge` procedure, which assumes both lists are sorted, and outputs a sorted list containing the union of those two lists with all duplicates removed. This is an efficient algorithm, which gives each processor a sorted list of the global node numbers that it needs. We then store this list in the `index_nod` array. We will use this array as a local-to-global node numbering mapping. For instance, if `index_nod(3)` = 78 on a given processor, then that processor's local node 3 corresponds to global node 78.

Now we want to determine which nodes are shared between processors. To store this information, we create a new data structure (implemented as a FORTRAN common block) containing several new variables. The definition of these new variables is presented in Table 5.2. Each processor takes turns broadcasting its copy of `index_nod` to all the other processors. The receiving processors then search through the list, comparing the sender's global node numbers to its own. Since both lists are sorted, this is a very efficient procedure.

If there are nodes shared with the sender, the receiving processor increments its *neighbors* variable, adds the sender to `neighborrank`, and continues through the list, incrementing

Table 5.2: Declarations and descriptions of data structures necessary for nodal distribution.

Variable	**Contents**
`MAX_NODE`	Maximum number of nodes on any processor
`MAX_NEIGHBORS`	Maximum number of processors sharing any single node
`MAX_SHARED`	Maximum number of nodes any single processor has to share
`nmynnp`	Number of nodes owned locally
`neighbors`	Number of processors with whom local nodes are shared
`index_nod(MAX_NODE)`	Mapping of local node numbers to global node numbers
`neighborrank(MAX_NEIGHBORS)`	Rank (in *mpi_comm_world*) of each neighbor
`num_nod_shared(MAX_NEIGHBORS)`	Number of nodes shared with each neighbor
`nod_shared(MAX_NEIGHBORS,MAX_SHARED)`	Local node numbers of the nodes shared with each neighbor

`num_nod_shared` for each match it finds, and adding those matches to *nod_shared*. This ensures that each processor's copy of *nod_shared* is sorted by global (and therefore local) node number. The processors take turns sending, in order of processor rank. Once the sender sends, it prepares for its next receive. This guarantees that the `neighborrank` array will be sorted in ascending order on all processors.

The sole purpose of the data structures we are creating here is to provide fast and efficient communication of distributed nodal values. It is imperative that any two processors who share a set of nodes know how to send and receive values corresponding to those nodes between each other, even if those nodes are known by different local node numbers on each processor. By keeping all of the lists in the *nod_shared* array sorted by global (and local) node number, we ensure that all communication of nodal values can be sorted by local node number. For example, let processors P_1 and P_2 share global nodes 3, 31, 44, and 76. If P_1 locally numbers the nodes 1, 4, 6, and 12, respectively, it can always communicate nodal values for those nodes to P_2 in that order, and P_2 will be expecting that order. In essence, we are defining all communications of nodal values to be ordered by global node number.

Our next step will be to input the nodal values from the files `coord.in`, `naturbc.in`, and `esenlbc.in`. All the processors know which nodes they need values for, but unlike the elements, some values are needed for more than one processor. This will make it difficult for a single processor to handle all the input for the program, as we did with the elemental values. For these input files, each processor will read its own nodal values. This simultaneous 'free-for-all' file reading is by no means the most efficient way to perform this task. A more efficient method, however, in which a single processor handles all the reading and communication of values, would be much more difficult to implement in a scalable manner. Since execution time in the preprocessing stage is almost meaningless compared to thousands of main loop iterations, we use the easier to implement, but less efficient algorithm.

Each processor first proceeds through the `coord.in` file, storing only the nodal coordinates that it needs. Next, the processors each read in the `naturbc.in` file, building their own local copy of the natural boundary conditions. Finally, they do the same with the essential boundary conditions for displacement in the `esenlbc.in` file.

The essential boundary conditions for potential must be handled differently, though. Since the root processor for each patch communicator will be responsible for solving that

patch's linear system, it must have enough memory allocated to store the entire linear system. This implies that every processor must allocate the space to store a complete copy of their patch's linear system. Using the `materl.in` file, each processor can determine which PZT patch it will be working with, so SMART can therefore create the *mpi_comm_patch* communicator, which will allow all processors in a patch to share the essential boundary conditions they read in. With the communicator in place, processors can read in the essential boundary conditions for potential just as they read the conditions for displacement.

From the data in `esenlbc.in`, processors each generate a list of the nodes they own that have free potential. Then, in procedure `lm_elec`, the processors in each patch take turns broadcasting the global node numbers of their local free nodes to the rest of the *mpi_comm_patch* communicator. The receiving processors use the `merge` procedure to build a complete list of the free potential nodes. When all of the processors have broadcast, each one will have the entire list of the global node numbers corresponding to each electrical equation in their patch, ordered by global node number. They can then fill their local copies of the `lmelec` array and the `idelec` array with their elements' equation numbers in that list.

The changes we have made to this point will provide us with all of the information necessary to distribute the nodal arrays. If we do nothing else, local mechanical arrays will be created for each processor, just as we want them to be, but we have given the processors no means of communicating their mechanical values. To perform this task, we need data structures similar to those we created for communicating the distributed nodal values. Hence, we create a `num_mech_shared` array and a `mech_shared` array (analogous to `num_nod_shared` and `nod_shared` described above). Since mechanical equations correspond directly to nodes, we already know how many processors we share mechanical equations with (`neighbors`) as well as their ranks (`neighborrank`). Filling `num_mech_shared` and `mech_shared` turns out to be surprisingly easy. For each processor a processor shares nodes with, it simply goes through each of those shared nodes in order and looks at the essential boundary conditions it has for the displacements of those nodes. For each free displacement it has for a node, it adds its local mechanical equation number to `mech_shared`, sorting them in the order x, y, and then z (if necessary). By using the definitions of `nod_shared`, which we know are consistent between neighbors, we have guaranteed that `mech_shared` is consistent between neighbors as well.

5.2.2 Main Loop

We now have the data structures in place to handle the communication of nodal and mechanical values between processors. Our next step is to devise efficient procedures to do so. The procedures we create will sum all shared remote nodal or mechanical values with the computed local values. Before the nodes were distributed, we accomplished this with a call to `mpi_allreduce` with the `mpi_sum` operation. For our distributed node program, we will create the `share_nodes` and `share_mech` procedures to handle communication of the nodal and mechanical values, respectively. It turns out that the only communications we need in the main loop are with `share_mech`, but we implement `share_nodes` anyway, in case it is needed in future improvements. Since both procedures are essentially identical, we will only look at `share_mech`.

Procedure `share_mech` is implemented with a loop over the processors. Each processor, in turn, plays the role of "sender." If a processor is the sender, it proceeds through its `neighborrank` array, using `mpi_send` to send each neighbor the appropriate mechanical values. If a processor is not the sender, it uses `neighborrank` to see if it shares nodes with the sender, and if so it uses `mpi_recv` to receive the appropriate number of mechanical values from the sender. The `mpi_send` operation is non-blocking, so as soon as a sender finishes sending, it looks for its next receive. Since the senders are in ascending order, as are the local lists of neighbors to receive from, we are guaranteed never to hit a state of deadlock where two processors are both waiting to receive data that the other has sent. A deadlock like this would be more difficult to avoid in a more elaborate communication scheme.

All processors maintain a running sum of values they have received to add to their mechanical arrays. These remote values cannot be added to the local mechanical arrays until after all shared values are sent. Once the remote values have been added into the local values, however, all local values on the processor are correct, and it has completed the communication. We will see in Section 5.3 that this new communication procedure is significantly faster than an `mpi_allreduce` on all of the mechanical values.

With our new `share_mech` procedure implemented, we simply replace all the `mpi_allreduce` calls for mechanical arrays with `share_mech` calls. We do this first for the communication of the global mass matrix, `globm`, after we assemble to it before the main loop. We also make the substitution after both instances of the nodal force loop, when we reduce `globfn`.

Since all processors cannot see all of the displacements, we cannot determine a stopping point in the same manner as previously. Originally all processors could see global node 1, and when its displacement first became negative, SMART would exit the main loop and begin output. Now, we give the owner of global node 1 that responsibility, and it broadcasts *istage* to the rest of the processors after each timestep. This is the only additional communication in the main loop of the distributed node program. We should note that no changes are necessary in the communication of electrical arrays.

5.2.3 Output

In the distributed node environment, one processor can no longer output all of the correct nodal values just by looking at its own local memory. Processors cannot all output all their local values either, since shared nodes will be stored on multiple processors. For output purposes, we need to establish unique ownership of each global node, so only one processor will output information for each node. Then the processors can take turns outputting the values they own, and we can be guaranteed output for one and only one of each global node. We define a node as being owned by a processor if that processor is the lowest ranked processor storing values for that node. We keep this information in the array `iown`, of size *MAX_NODE*. Given a local node i on a processor, that processor owns and is responsible for the output of that node only if `iown(i)` is not 0. Node ownership is determined by processor rank—the lowest numbered processor that needs a node is that node's owner. The values of `iown` are filled by each processor for all local nodes, using this criterion, from the local information in `neighborrank` and `nod_shared`. Processors output nodal data to file one at a time in rank order. Each processor takes its turn sending its nodal output to processor 0, which in turn, adds it to the output file. All nodal value output lines for local node i are preceded by the global node number, `index_nod(i)`. This allows us to order them by global node number with the UNIX `sort` command before we use those values for any post processing operation, e.g., plotting.

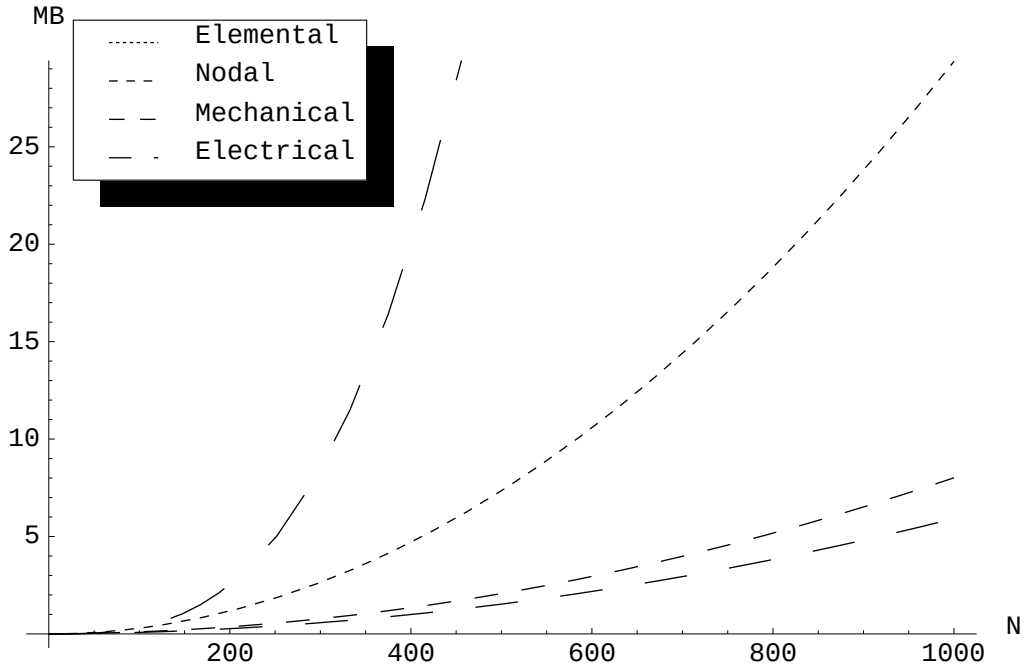

Figure 5.2: Memory required per processor for each of the four array types when elemental, nodal, and mechanical arrays are distributed; 64 processors running on an $N \times N$ uniform mesh.

5.3 Performance

Figure 5.2 shows the memory requirements for the new version of SMART (compare with Figure 5.1 on page 46). The distribution of nodal and mechanical values enables another refinement step in the mesh size that SMART can handle, increasing the maximum uniform mesh size from 200×200 to 400×400, given a fixed 30MB per processor. Additionally, nodal arrays are no longer the memory bottleneck. In fact, both electrical and elemental arrays now require more memory.

The running times and scalability of the new code are discussed in Chapter 6. From a performance standpoint, almost the entire main loop has been accelerated. With the reduction of `globek` and the linear system solve being the only exceptions, every significant portion of the program is running more efficiently. This can be seen in Table 6.1.

Chapter 6

Parallel Performance

In this chapter we present two performance metrics for analyzing our parallel implementation of SMART. Both metrics compare parallel run times to sequential run times, but the first fixes the global problem size, illustrating speedup, while the second fixes the local problem size, with a focus on scalability.

6.1 Fixed Size Speedup

The first parallel performance metric we use is the fixed size speedup of the SMART simulation. As implied by its name, for this evaluation we fix the size of the problem the code is simulating, and run it on different numbers of processors, comparing the parallel performance with the sequential performance of the code to compute its parallel speedup and efficiency on that fixed problem.

This evaluation is best used as a diagnostic tool, to locate bottlenecks in parallel performance (as it did in Table 3.1 on page 32, revealing the enormous relative cost of the linear system solve on 64 processors). It will also give an indication of how well the program is running in parallel, even though it is unrealistic to think that global problem sizes will be fixed as we run the simulation on more and more processors. One danger of the fixed size speedup data lies in the computation of the speedup numbers. The true speedup for a parallel implementation of a simulation code is the performance gain allowed by the parallel code

over the best sequential code. Often, when compared to just a one processor run of itself, the current parallel implementation will show better speedup because of the overhead involved in the parallel implementation of that code. Ideally, a table like this will show both the best sequential code available, for speedup calculations, as well as the uniprocessor version of the parallel code, which will indicate the parallel overhead in the new code.

Table 6.1: Profiling information after nodal distribution; 200×200 mesh.

Segment of code	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Nodal Force Loop	2093.4	1047.6	524.6	262.8	131.5	67.9	39.8
Comm. of `globfn`	0.0	0.6	0.3	0.3	0.4	0.6	1.8
Nodal Potential Loop	1038.6	519.4	261.0	130.9	65.4	34.2	21.6
Reduce of `globek`	3.9	2.7	6.9	15.4	21.8	29.7	44.1
Reduce of `globfe`	0.1	0.2	0.2	0.3	0.4	0.5	1.8
Linear System Solve	41.0	20.3	18.9	20.7	18.2	18.4	18.4
Timestep Calculation	3510.9	1757.2	880.5	441.1	220.4	114.3	62.5
Other	13.6	9.1	4.7	3.0	2.9	2.6	3.5
Total	**6701.5**	**3357.1**	**1697.1**	**874.5**	**461.0**	**268.2**	**193.5**
Speedup	**1.00**	**2.00**	**3.95**	**7.66**	**14.54**	**24.99**	**34.63**
Efficiency	**1.00**	**1.00**	**0.99**	**0.96**	**0.91**	**0.78**	**0.54**

Table 6.1 shows the fixed size speedup of the code segments in the main loop of SMART. Note that the data for the single processor case is simply this code running on one processor. In the sequential case, the "communication of `globek`" is more expensive than the communication on 2 processors, which is odd, since the single processor case should not require any communication. The `mpi_allreduce` call we are using must also copy the `globek` to another array. Because the decoupled `globek` cannot be handled in parallel on the single processor case, this copy requires twice the time it needs in the two processor case. Furthermore, intraprocessor communication is trivial during a 2-processor run on the Origin, since the two processors share the same local memory. For these reasons, it is understandable why the two processor case is running faster.

On this large problem, up to 32 processors, the nodal force loop, nodal potential loop, and the timestep calculation are speeding up almost perfectly. On 64 processors, however,

the parallel efficiency of all three suffers. This poor efficiency can be attributed to a load imbalance between the processors. Recall that the parallel domain decomposition divides up half-columns of the mesh between processors, so some processors will have 4 half-columns, while others will only have 3. With 100 elements per half-column, and 40 of those being PZT elements, the work imbalance on those three loops begins to show in the final running times.

For this mesh size, the linear system is small enough to not seriously effect the overall speedup until the 32 processor case. The linear system solve combined with the communication of the `globek` matrix accounts for over a third of the 64 processor total running time, compared with less than a hundredth of the total running time for the 1 and 2 processor cases.

6.2 Fixed Local Problem Size

The second set of parallel performance data looks at run times in the main loop for a fixed local problem size. We define "fixing the local problem size" to mean fixing the number of elements per processor. Fixing this quantity will allow us to evaluate the scalability of the program. The less a single processor is influenced by the global problem size (which grows linearly with the number of processors), the more scalable the code is, and the better it will run on large numbers of processors. A perfectly scalable parallel code is influenced solely by the local problem size. For example, for 100 timesteps of the main loop, a perfectly scalable version of the SMART code would take just as long simulating a 100 element mesh on 2 processors as it would simulating a 10000 element mesh on 200 processors.

SMART contains several data structures that still grow with the global problem size, namely the electrical arrays, and most notably the `globek` array. For this reason, the scalability of the code can still be improved on. This fact is reflected in Table 6.2.

The table reveals a great deal about the scalability of our parallel implementation of SMART. The running times for the nodal force loop and the timestep calculation appear to depend solely on the number of elements per processor, and not on the problem size. The running time for the nodal potential loop does, however depend on the size of the electrical

Table 6.2: Parallel performance of SMART running on different meshes, fixed at 1600 elements per processor.

Processors:	**1**	**2**	**4**	**8**	**16**	**32**	**64**
nele:	**1600**	**3200**	**6400**	**12800**	**25600**	**51200**	**102400**
Nodal Force Loop	84.4	84.6	85.1	85.5	86.0	85.5	85.5
Comm. of `globfn`	0.0	0.1	0.1	0.2	6.6	8.4	41.3
Nodal Potential Loop	41.5	42.0	42.5	43.9	56.3	95.2	240.3
Reduce of `globek`	0.1	0.1	0.4	2.1	11.3	36.0	289.8
Reduce of `globfe`	0.0	0.0	0.0	0.1	0.7	0.5	1.5
Linear System Solve	0.3	0.3	1.0	2.2	8.7	18.1	163.9
Timestep Calculation	140.3	146.4	142.6	143.4	147.2	173.1	156.6
Other	0.4	0.5	1.5	0.9	2.0	3.1	14.2
Total	**267.0**	**274.0**	**272.3**	**278.2**	**318.8**	**419.9**	**993.1**

arrays, *neqelec*, which is growing with the mesh. This dependence is primarily in the local assembly procedure for the `globek` matrix discussed in Chapter 4. In fact, every part of the main loop with any relation to `globek`, is growing with the problem size. The "Other" section is included in this group, because it includes the initialization of `globek`, which accounts for almost all of the time in that section.

Chapter 7

Conclusions and Future Work

7.1 Summary and Conclusions

In this thesis we describe the results of a project whose goal was to develop an efficient scalable implementation of SMART: a large engineering code that simulates a smart structure reacting to an applied force. Our target parallel environment is characterized by distributed memory and explicit message passing. Our codes have been developed and tested on an Intel Paragon and an SGI Origin 2000. In the preceding chapters we described the numerical algorithm and its implementation in the original SMART code of Liang, summarized its sequential performance, which is dominated by the solution of a linear system and three loops over elements, and described some sequential improvements. We then described three major optimizations to the code. Our first parallel version gains efficiency by distributing the data and work associated with elements across multiple processes. Our second series of optimizations involves substantially reducing the cost of the linear solves, by exploiting the bandedness and symmetric positive definiteness of the matrix, and by renumbering nodes and elements in such a way that the bandwidth of the matrix is reduced and so that the matrix decouples into two independent problems which may be solved in parallel. Finally, our third parallel version of SMART distributes all nodal data and work associated with the nodes across the processors. This final version of the code has a much smaller per-processor memory requirement than either of the previous versions.

Table 7.1: Run times for each code version; all times are for 1000 main loop iterations on a 60×60 mesh.

Code Version	**Run Time**
Original Code	1466.0
Improved Sequential Code	1223.4
Distributed Elements (32 procs)	836.9
Improved Linear System (32 procs)	22.9
Distributed Nodes (32 procs)	20.7

The parallel performance of the distributed nodes SMART code is promising. With a high parallel efficiency, on up to 16 processors of our 200×200 test case on the Origin, the parallel code will provide near perfect speedup for meshes containing a high ration of non-PZT to PZT elements. Furthermore, the nodal force loop, the timestep computation, and even the nodal potential loop scale reasonably well on the Origin, depending almost completely on the local problem size. The goal of SMART being scalable on larger machines, however, will not be possible until the electrical arrays are distributed in some manner—the global assembly and solution of the linear system must better utilize parallelism in order to reduce both its running time and memory requirements. Until such an improvement is made, the linear system will remain a bottleneck in terms of both performance and space requirements.

In addition to the particular parallel code implemented for this research, we have also illustrated an important fact about large-scale parallel simulations: the development of an efficient scalable parallel engineering simulation is a process that requires knowledge of, careful thinking about, and effort on many different fronts. Starting with the characteristics of the problem, we had to understand that PZT elements and non-PZT elements require different amounts of computation, and hence balancing the number of those element types across the processors is crucial to achieving peak efficiency. An understanding of the numerical algorithm was required, not just so we could see what operations the code performs, but so we could have some insight into why those operations are being performed. This understanding allows us to find better ways to implement algorithms, like our sequential improvement on the application of essential boundary conditions. Clearly, knowledge of linear algebra was required. We exploited the symmetric positive definiteness and the bandedness of the

linear system, which a parallel programming tool would likely not have detected. We also, however, used our knowledge of the problem geometry to take that a step further, not only reducing the bandwidth of the linear system by changing the equation numbering order, but also by recognizing that linear systems for distinct patches decouple, and solving those decoupled systems in parallel. The data and problem distribution was extremely important to understand as well. Seeing how nodal and elemental values were being used in the program, and recognizing their dependencies on each other, allowed us to determine that the code was better suited for elemental distribution, and gave us ideas about how to go about distributing those values. The implementation of efficient communication between processors was also required. By sacrificing a small amount of preprocessing, we implemented a nodal communication scheme that was optimal for both the sending and receiving processors, while maintaining a small message size. We utilized our knowledge of single processor computer architectures, especially with respect to caching. The initialization of the dense linear system ran 4 times faster, if we simply switched the loop indices. We also needed familiarity with the input and output files for SMART. These files are the link between our simulation code, and pre-existing pre- and post-processing tools. These tools can be useful for creating a mesh, imposing a domain decomposition, or for visualizing output. Often, post-processing tools require input files as well to generate data. If the output of the program is not consistent with the input files (e.g. if local element numbers are output instead of global numbers) these tools will not work.

7.2 Future Work

There is much potential for future work on the SMART simulation. First and foremost, an attempt should be made to implement the linear solve in parallel. This parallel implementation can be either with a direct method or with an iterative method. Most parallel direct methods for computing banded Cholesky decompositions, like SCALAPACK, tend to only be efficient on narrow band matrices. Recently, Gupta et al. [14] have implemented an algorithm which is more efficient on matrices with wider bands. The `globek` matrix would most likely factor more efficiently with this algorithm than with the narrow band algorithms. A direct method of this type still would not take advantage of the sparsity within the band of `globek`. A parallel iterative method could take advantage of that sparsity, but would

require the choice of a preconditioner. If SMART is to truly achieve scalability, the linear system must be solved efficiently in parallel. Assuming that is accomplished, the next logical step would be to evaluate the scalability of SMART, on both bigger problems and bigger machines.

The SMART code is designed to handle much more complex meshes than we use in our simulation. SMART can handle three dimensional meshes, with piezoelectric patches and composite material arranged any way imaginable. A parallel implementation of SMART for this general case would be significantly more difficult for several reasons. First, finding a parallel domain decomposition where all the elements, as well as the PZT elements are distributed evenly, is not difficult; but finding such a decomposition where the number of nodes per processor is balanced is a difficult problem. Furthermore, to allow the parallel solution of decoupled linear systems, each processor is restricted to one PZT patch. This must be taken into account as well. Even with the responsibility of the domain decomposition left up to other, pre-processing programs, a three dimensional mesh will still present some problems. For the linear systems, we can still assume they are banded, but the bandwidth in a three dimensional mesh will be much greater. SMART is flexible enough to run on a more general problem geometry, but will not run as efficiently as it does on our smart structure.

With the distributed memory implementation of SMART already ported to the SGI Origin, it would also be interesting to develop a virtual shared memory code for comparison. In such a code, the nodal arrays would not have to be distributed—every processor could have access to the global copies of all nodal arrays. It would still be helpful, and perhaps easier, to utilize parallelism in the linear system solve. The performance of this shared memory code versus a scalable distributed memory code would provide a good indication of how well distributed shared memory and message passing are implemented on the Origin.

Appendix A

The SMART Program - Sequential Pseudocode

The following is a high-level pseudocode description of Liang's SMART code. Variable names, where useful, are given in parentheses, while subroutine names are given within brackets at the end of lines.

```
Variable Declarations

Set Calculation Constants / Initialize variables

Read in data from the input files

Number the nodes that can move freely in each direction [lm_mech]
Number the nodes whose charge can change [lm_elec]

For each element

   Store coordinate information for element
   Calculate dN/dX (shpxyz) and Jacobian (vj) at quadrature points [shape]

End For

For each element

   Calculate the element mass
   Assemble mass to the global mass matrix (globm) [mass]
```

```
13  End For

14  For each node

15     Copy initial displacements to the global displacement vector (dnow)
16     Copy initial velocities to the global velocity vector (vnow)

17  End For

18  Set the essential B.C. displacement arrays (x,y,zdspl)
19  Set the essential B.C. potential array (potlbc)
20  Set the natural B.C. force arrays (x,y,zload)

21  Initialize RHS of mechanical equations (globfn) to 0

22  For each element (the preliminary nodal force loop)

23     Calculate the element's strain [strain]
24     Calculate the element's deformation gradient and its determinant [strain]

25     If the element is a PZT element

26        Calculate the element's electric field [w_field]
27        Calculate the element's first Piola-Kirchhoff stress tensor [law3NL]

28     Else

29        Calculate the element's first Piola-Kirchhoff stress tensor [law4L]

30     End If

31     Calculate the internal nodal force for the element [nforce]
32     Assemble it to the global force vector (globfn) [nforce]

33  End For

34  Add the natural B.C. forces to the global force vector [load_mech]

35  For each mechanical equation (1 to neqmech)

36     Solve for acceleration (anow = globfn / globm)
37     Calculate the displacements for time -dt (dold)

38  End For
```

```
For each node [joint1]

   Copy computed displacements (dnow) to global displacement vector (displt)
   Write essential B.C. (x,y,zdspl) to global displacement vector (displt)

End For

Start the main loop (repeat for prescribed number of stages)

   Initialize RHS of mechanical equations (globfn) to 0

   For each element (the nodal force loop)

      Calculate the element's strain [strain]
      Calculate the element's deformation gradient and its
         determinant [strain]

      If the element is a PZT element

         Calculate the element's electric field [w_field]
         Calculate the element's first Piola-Kirchhoff stress
            tensor [law3NL]

      Else

         Calculate the element's first Piola-Kirchhoff stress tensor [law4L]

      End If

      Calculate the internal nodal force for the element [nforce]
      Assemble it to the global force vector (globfn) [nforce]

   End For


   Add the natural B.C. forces to the global force vector [load_mech]

   For each mechanical equation (1 to neqmech)

      Compute new displacements (dnew) using central difference formula

   End For

   Set the essential B.C. displacement arrays (x,y,zdspl)
   Set the essential B.C. potential array (potlbc)
```

```
63     Calculate the essential potential increment (pbcinc) for the new time
64     Set the natural B.C. force arrays (x,y,zload)

65     For each node [joint1]

66        Copy computed displacements (dnow) to the global displacement
             vector (displt)
67        Write essential B.C. (x,y,zdspl) to the global displacement
             vector (displt)

68     End For


69     If there are PZT elements in the mesh

70        For each element (nodal potential loop)

71           If the element is a PZT element
72              Calculate the element's strain [strain]
73              Calculate the element's deformation gradient and its
                   determinant [strain]
74              Calculate the element's electric field [w_field]
75              Calculate the Jacobian of the discretized Maxwell
                   equation [jac3NL]
76              Assemble the Jacobian into global electrical Jacobian
                   matrix (globek) [jac3NL]
77              Calculate the element's first Piola-Kirchhoff stress
                   tensor [law3NL]
78              Calculate the element's electric force [echarge]
79              Assemble it into the global electric force (globfe) [echarge]
80              Calculate the nodal charge contributed by
                   essential B.C. [e_BC1a]
81              Assemble that charge to global electric RHS (globfe) [e_BC1a]
82           End If

83        End For

84        Solve the potential increment for the reduced Maxwell equation
85        (a general system of linear equations)

86     End If


87     For each node [joint2]
```

```
      Copy the computed nodal potential to the global potential
         vector (potent)
      Write essential B.C. (potlbc) to the global potential vector

   End For


   If this iteration number is divisible by 10
      For each element [timestep]

         Assemble an element mass matrix (elmass)
         (It suffices to say that this requires a lot of computation)

         Solve a 24x24 EVP: [elstif]{U} = ev [elmass]{U}
         Use this result to find a timestep
         If the timestep is smaller than all the others so far, use it

      End For
   End If

Repeat the main loop until the prescribed number of stages have completed

Output sigma values

Program complete
```

Appendix B

Mesh Generation and Domain Decomposition

SMART requires two main preprocessing steps. First, the smart structure must be discretized into a finite element mesh. This is performed by MESH, our mesh generator, which takes information about the structure as input, and outputs the input files for SMART. The second step is parallel domain decomposition, where a second program, DECOMP, is required to divide the task among the processors involved in the computation. The input required for this stage is simply a number of processors, and the input files generated by MESH. Both of these preprocessing steps are discussed in this appendix.

B.1 Mesh Generation

We have written the MESH program to generate the discretized smart structure simulated in this thesis. MESH requires several parameters, including the x-, y-, and z-dimensions, which denote the size of the structure, as well as the maximum number of elements in the x- and z-directions (with the number of y-elements fixed at 1). MESH works for both PZT and non-PZT meshes, so if there are PZT layers on both sides of the mesh, the height of those layers, as well as the width of the notch on the fixed end can be specified. A non-PZT mesh would contain no PZT elements.

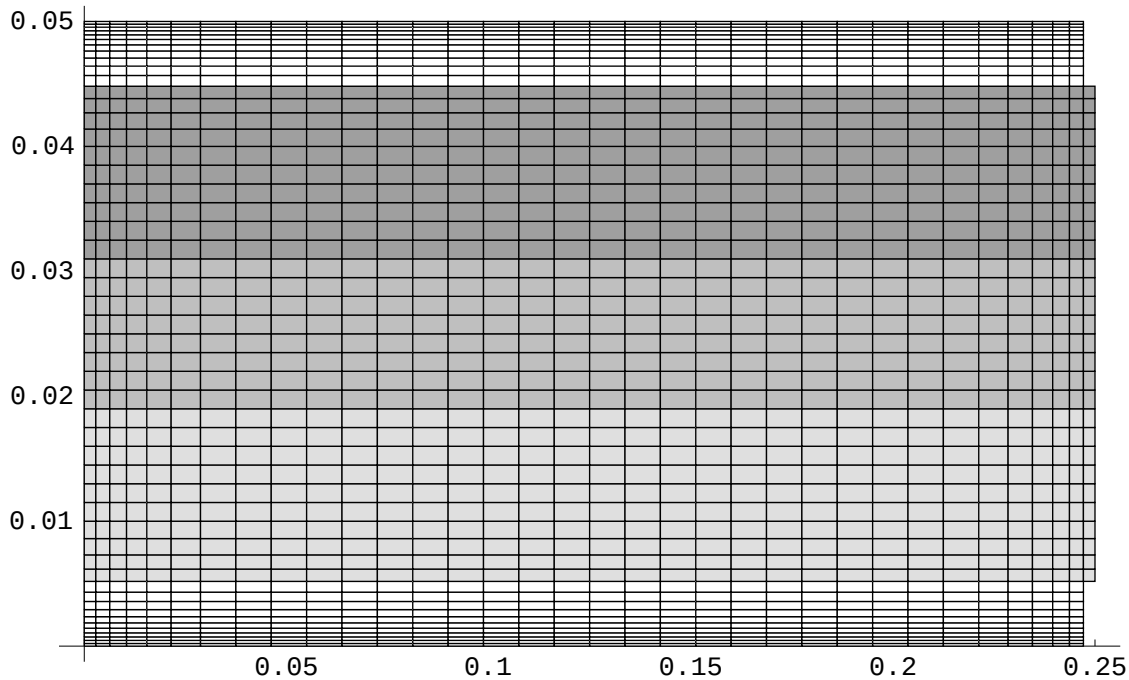

Figure B.1: A sample geometrically refined mesh.

One feature of the mesh generator which we have not utilized in this thesis is the ability to generate nonuniform meshes. MESH can geometrically refine its meshes, near the edges, in both the x- and z-directions. This geometric refinement is classified by two parameters, α and β. The α parameter tells the mesh generator where to start refining the mesh in a given direction, assuming the axis in that dimension ranges from 0 to 1. For example, if $\alpha_z = 0.1$, then eight-tenths of the material will have uniform element lengths in z, while the tenths of the beam nearest the top and bottom edges will both contain geometrically refined elements. The β parameter indicates the geometric constant of refinement. If $\beta_z = 0.98$, then each element in a geometric region has a z-length of $0.98k$, where k is the z-length of the neighboring element closest to the center of the beam. A geometric mesh, with $\alpha_x = 0.15$, $\beta_x = 0.83$, $\alpha_z = 0.2$, and $\beta_z = 0.87$ is shown in Figure B.1.

The transition from uniform to geometric element widths in a mesh is rarely perfect, meaning it is difficult to generate a geometric mesh where *every* nonuniform-element has length β times that of its largest neighbor. This is because the user also has the flexibility to determine the exact length of the material. As a result, the "transition" elements between the

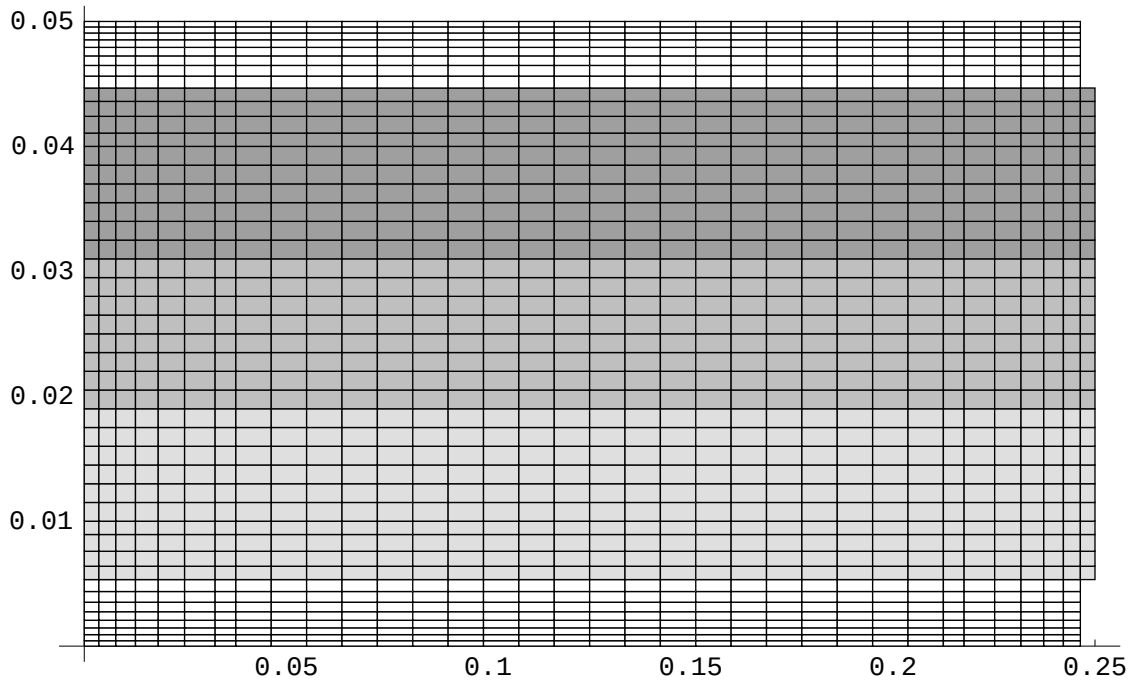

Figure B.2: A sample geometrically refined mesh with small transition elements.

uniform and non-uniform portions of the mesh may be smaller than the elements surrounding them. The mesh generator warns the user when this occurs. An example of such a mesh can be seen in Figure B.2. In the figure the transition elements in rows 12 and 33 are smaller than their surrounding elements in order to allow exact geometric refinement near the edges. The same is true for the transition elements in columns 7 and 28.

Additionally, MESH applies force parabolically to the nodes on the free edge (farthest from the fixed surface in the x-direction). A force in the positive z-direction is applied to every node on that free edge of the structure, with the maximum force specified by the user. The mesh generator assumes the forces on the nodes at the top and bottom edges are zero, and then considers a parabola "over" that face of the structure with zeroes at the edges and the user-specified maximum in the center. Point forces for the `naturbc.in` file are then computed by symbolically integrating that parabola over the face of each element on that face of the smart structure.

B.2 Parallel Domain Decomposition

The parallel domain decomposition for our mesh is important to the scalability of the program. First, to guarantee load balancing in the nodal force loop and the timestep calculation, the decomposition must give each processor approximately the same number of elements. Second, to assure load balancing in the nodal potential loop, the PZT elements should be evenly distributed as well. Furthermore, for parallelism in the linear system solve, no processor should own elements from more than one PZT patch. Finally, the procedures `share_mech` and `share_node` must be efficient. Ideally this means keeping the `nod_shared` data structure small, hence the number of nodes shared between any two processors should be kept small, as well as the number of processors with whom any given processor shares nodes.

Our program DECOMP handles this decomposition. Requiring as input the number of processors, and then by reading the input files for the program (specifically `ien.in` and `materl.in`), DECOMP generates a new file, `parallel.in`, and rearranges the elemental input files for easier parallel input by SMART. Our decomposition scheme assumes that, if the simulation is running in parallel, there must be an even number of processors. This assumption allows the creation of two mesh halves, each containing a PZT patch, which are then partitioned evenly, based on the total number of processors being used. The parallel domain decomposition over 6 processors for a 30×30 mesh is shown in Figure B.3.

The `parallel.in` file created by DECOMP contains two values per line: a global element number and a processor rank. The `read_parallel` procedure in SMART will read this file in, under the assumption that it is sorted by processor rank. Like the other elemental input files, `parallel.in` is read in by the highest rank processor, which first reads for processor 0, then for processor 1, and so on, determining the number of elements on each processor. These numbers enable SMART to read the other input files, which must also be rearranged in preprocessing so that their elements are in the same order as those in the `parallel.in` file. With all the elemental input files sorted by processor and by global element number on that processor, `nelecount` can be filled during the reading of `parallel.in`, and hence the procedures for reading the other elemental input files can still rely on the assumed distribution mentioned in Chapter 3.

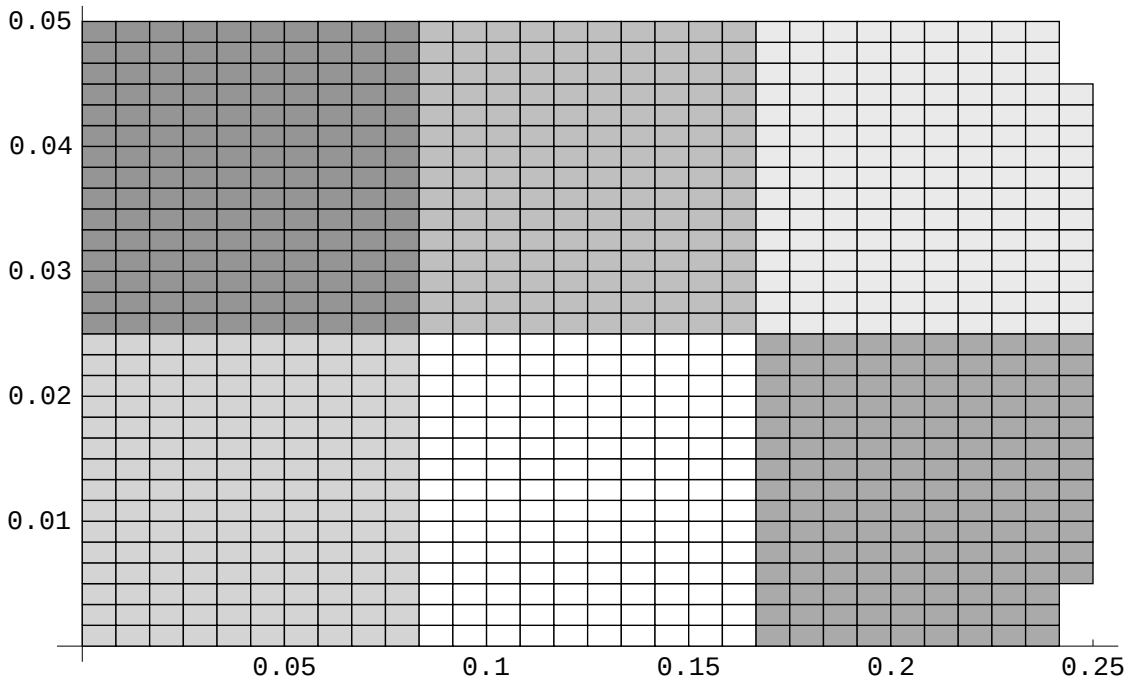

Figure B.3: Parallel domain decomposition of a 30×30 mesh on 6 processors.

Bibliography

[1] G. Abandah. *Reducing Communication Cost in Scalable Shared Memory Systems*. PhD thesis, University of Michigan, Ann Arbor, MI, 1998.

[2] I. Ahmad, K. Leung, and S. Hsu. Ocean Circulation on the Intel Paragon: Modeling and Implementation. *The Journal of Supercomputing*, 10(4):349–369, 1997.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1995.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *SCALAPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1997.

[5] G. Carey. Parallelism in Finite Element Modelling. *Communications in Numerical Methods in Engineering*, 2:281–287, 1986.

[6] M. Crowley, T. Darden, T. Cheatham, and D. Deerfield. Adventures in Improving the Scaling and Accuracy of a Parallel Molecular Dynamics Program. *The Journal of Supercomputing*, 11(3):255–278, November 1997.

[7] T. Cwik, D. Katz, C. Zuffada, and V. Jamnejad. The Application of Scalable Distributed Memory Computers to the Finite Element Modeling of Electromagnetic Scattering. *International Journal for Numerical Methods in Engineering*, 41:759–776, February 1998.

[8] N. Davies. *The Performance and Scalability of Parallel Systems*. PhD thesis, University of Bristol, Bristol, United Kingdom, 1994.

[9] J. Dongarra and T. Dunigan. Message-Passing Performance of Various Computers. *Concurrency: Practice and Experience*, 9(10):915–926, 1997.

[10] J. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[11] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions*, 21(9):948–960, September 1972.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1983.

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

[14] A. Gupta, F. Gustavson, M. Joshi, and S. Toledo. The Design, Implementation, and Evaluation of a Symmetric Banded Linear Solver for Distributed-Memory Parallel Computers. *ACM Transactions on Mathematical Software*, 24(1):74–101, March 1998.

[15] Z. Johan, K. K. Mathur, S. L. Johnsson, and T. J. R. Hughes. Scalability of finite element applications on distributed-memory parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 119(1-2):61–72, 1994.

[16] S. Johnsson and K. Mathur. Data Structures and Algorithms for the Finite Element Method on a Data Parallel Supercomputer. *International Journal for Numerical Methods in Engineering*, 29:881–908, 1990.

[17] M. Jones and P. Plassmann. Computational Results for Parallel Unstructured Mesh Computations. *Computing Systems in Engineering*, 5:297–309, 1994.

[18] C. Koelbel, D. Loveman, R. Schreiber, G. Stelle Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[19] Kuck and Inc. Associates. *http://www.kai.com*.

[20] X. Liang. *Dynamic Response of Linear/Nonlinear Laminated Structures Containing Piezoelectric Laminas*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, March 1997.

[21] J. Prevost. *Dynaflow User Manual*. Princeton University, Princeton, NJ, 1999.

[22] J. P. Shields. *Basic Piezoelectricity*. H. W. Sams, Indianapolis, IN, 1966.

[23] S. VanderWiel, D. Nathanson, and D. Lilja. A Comparative Analysis of Parallel Programming Language Complexity and Performance. *Concurrency: Practice and Experience*, 10(10):807–820, August 1998.

Vita

Jeremy Michael Rotter was born on July 25, 1975 in New Brunswick, New Jersey to Jacob and Kathleen Rotter. He graduated from Rutgers Preparatory School in Somerset, New Jersey in 1993. He got a Bachelor of Science degree in computer science and mathematics from Virginia Polytechnic Institute and State University in 1993. He received a Master of Science degree in Computer Science and Applications from Virginia Polytechnic Institute and State University in 1999.