APPLICATION OF DISTRIBUTED ALGEBRAIC MULTIGRID TO NONLINEAR FINITE ELEMENT ANALYSIS OF GEOMECHANICS PROBLEMS

by

Charles L. Farris

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado

Date 11/21/96

Signed: ______________________
Charles L. Farris

Approved: Manavendra Misra.
Dr. Manavendra Misra
Thesis Advisor

Approved: ______________________
Dr. D. Vaughan Griffiths
Thesis Co-Advisor

Golden, Colorado

Date 11/21/96

____
Dr.
Professor and Head,
Department of Mathematical and Computer Sciences

ABSTRACT

The combination of the finite element method with the Monte Carlo method in the application of random variables to very large nonlinear geomechanics problems represents an extremely large computational undertaking. Improved performance of the finite element method would greatly decrease the computational intensity and allow the practical use of this combination in industry. Increased performance was sought through the selection of an appropriate linear solver and modification of that solver to take advantage of distributed processing. Since the primary performance limitation for this type of problem is disk swapping delays due to limited physical memory, the goal of the distributed processing was to eliminate the occurrence of swap during execution.

The algebraic multigrid algorithm (AMG) was selected over the preconditioned conjugate gradient (PCG) algorithm due to its better performance on problems involving poorly scaled meshes and their resulting ill-conditioned stiffness matrices. The AMG algorithm was then modified to handle two possible distributed data mappings and also exploit some parallelism during the distributed execution. The resulting algorithms achieved varying levels of performance, although on the whole every algorithm was able to achieve significant speedup.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank the members of my committee, Dr. Manavendra Misra, Dr. D.V. Griffiths, Dr. Bernard Bialecki, and Dr. Graham Mustoe for their work and support in this research. I would like to extend a special thanks to Dr. Misra and Dr. Griffiths for the help they gave me in merging mathematics and engineering in order to achieve advances in both fields. I would also like to thank KRJA Systems, Inc./Maptek for funding this research and allowing me to use their workstation LAN in order to generate my results.

Finally, I would like thank my wife Karen, who supported and encouraged my work throughout the last two years.

Chapter 1

INTRODUCTION

A distinct limitation of the use of computational methods, such as the finite element, boundary element, or finite difference methods, in geomechanics arises from the difficulty of obtaining accurate problem data. Values for material properties, boundary conditions, initial conditions, and problem geometry must be determined from field measurements and laboratory testing. Unfortunately, these sources of data are rarely complete and are often difficult and expensive to obtain, forcing the engineer to estimate and assume values for use in computational models.

Due to this difficulty, many design problems in geomechanics use empirical methods [1]. Empirical methods are relationships derived from physical observations rather than theoretical models. (1.1) illustrates an empirical relationship for pillar stability derived by Salamon and Munro [2] for use in South African mines. The factor of safety S for the pillar is calculated from the pillar height, width, and loading:

$$S = \frac{K h^{\alpha} \omega^{\beta}}{p} . \tag{1.1}$$

In this relationship, h, w, and p are the pillar height, width, and load, respectively. The constants K, α, and β are derived statistically from mine reports regarding the failure of pillars. If $S < S_c$ where S_c is the critical safety factor, the pillar is considered unstable and prone to failure. This particular empirical relationship has been so effective in the design of stable pillars that it is still in use today with little modification.

Empirical methods are extremely powerful in that they rely on limited problem information. Using the stability formula above as an example, the empirical method does not require the engineer to know the strength of rock, the degree of fracturing or faulting, or the location of the pillar relative to other nearby structures within the mine. This information is built into the relationship during its derivation. A useful property of many empirical relationships is the ability to estimate the probability of failure. In (1.1), the closer S is to S_c, the greater the probability of failure.

However, empirical methods do have a major disadvantage in that they suffer from vagueness in their results. The pillar stability formula can only indicate whether or not a pillar is likely to fail. It cannot indicate when or where the pillar might fail. This information is best obtained from a computational or analytical solution method. Closed-form solutions or numerical approximations of the solution allow the engineer to analyze the problem in much greater detail.

An ideal analysis method would combine the detailed analysis of the computational methods with the probabilistic power of the empirical methods. A possible method arises from the application of random variables to computational methods [3],[4],[5],[6]. This type of analysis has attracted considerable interest in the field of reliability engineering as an alternative to extensive physical model testing.

A particular application of this approach is in the derivation of the relationship between the factor of safety and the probability of failure. A factor of safety is an amount of over-design introduced into a design to decrease the probability of failure. For example, an engineer designing a hoist for lifting ore buckets estimates the cables lifting a full bucket must support 10 tons. To decrease the likelihood of a cable snapping, the engineer specifies the cables in the design must support 20 tons. This over-design should prevent failure under normal operation and also allow for unforeseen factors such as an overloaded bucket, dropped buckets, or inherent flaws in the cables.

By determining the relationship between the factor of safety and the probability of failure, the engineer can eliminate excessive over-design while maintaining a low probability of failure. Figure 1.1 illustrates a possible relationship derived using this method. An engineer could then use this relationship to reduce the factor of safety while maintaining a low probability of failure.

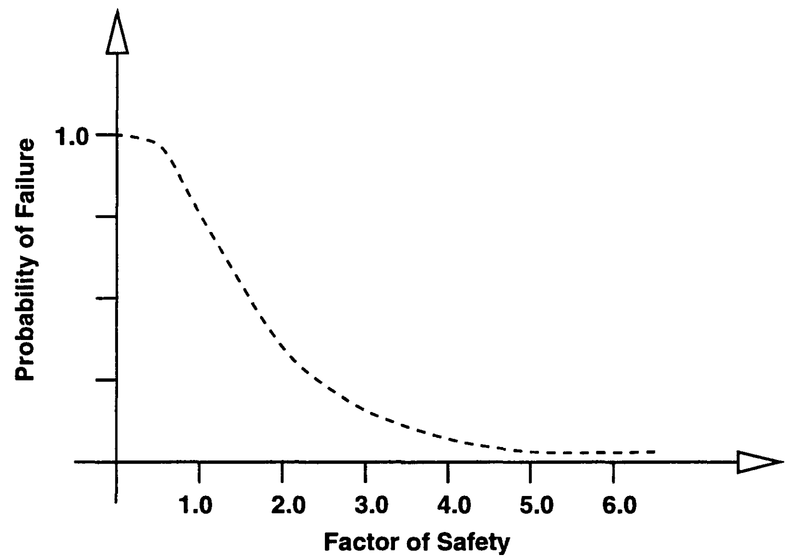

FIG. 1.1. Factor of Safety vs Probability of Failure

An example of a problem using random variables is the settlement of foundations on soil. Since soil is rarely a uniform material with consistent properties, random variables are used to describe the variability of the material. Figures 1.2 and 1.3 illustrate examples of uniform and nonuniform soils.

In the case of the nonuniform soil, the exact distribution of the soil properties might not be known. Accordingly, a random variable will be used to specify those properties. When the problem is solved, either analytically or numerically, the random

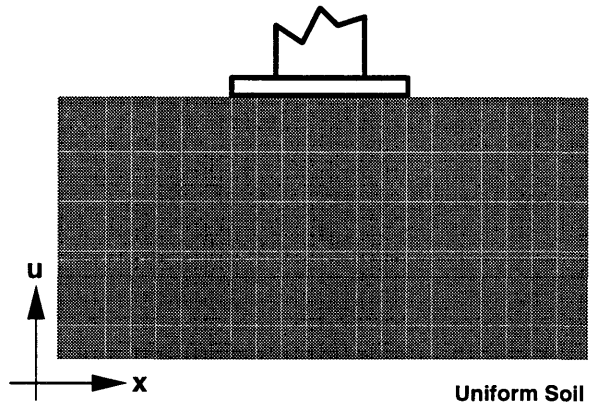

FIG. 1.2. Example Problem - Foundation Resting on a Uniform Layer of Soil

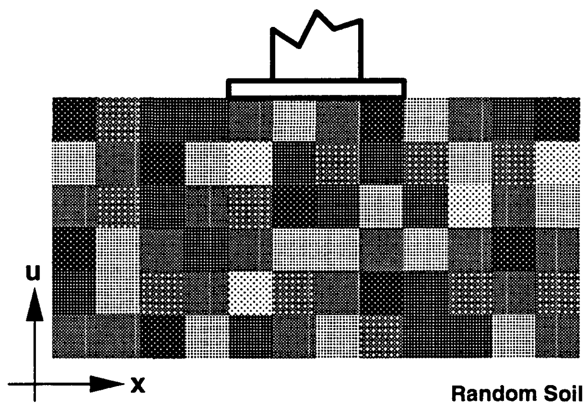

FIG. 1.3. Example Problem - Foundation Resting on a Nonuniform Layer of Soil

variable affects the results to reflect the most likely outcome of the analysis. Figure 1.4 shows the displacement of the soil based on uniform properties. The displacement is uniform and symmetrical. In comparison, Figure 1.5 shows the results based on the analysis using the random variable. The displacement in this case is not as regular as in the case of the uniform material. The way in which the problem is solved affects the method in which the random variables are applied. Two main techniques are used in the application of random variables. The first method is to derive the probabilistic results of the problem directly from the random variables, typically

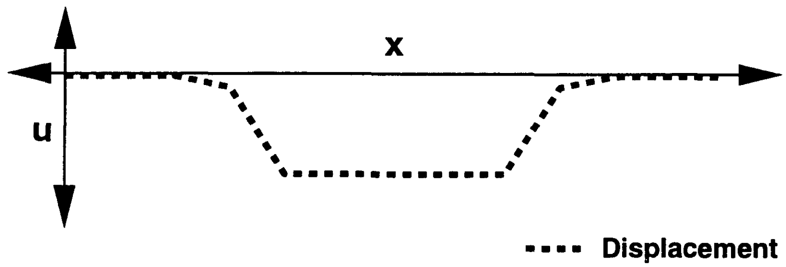

FIG. 1.4. Example Problem - Displacements for the Uniform Soil

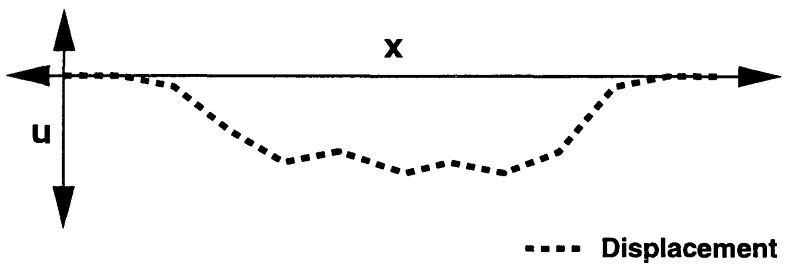

FIG. 1.5. Example Problem - Displacements for the Nonuniform Soil

using a truncated series expansion of the random process [5]. The other method is to use a Monte Carlo method to derive set of results suitable for statistical analysis. Each method has advantages and disadvantages. The first method is much simpler computationally, but is limited to problems involving low variance modeling [4],[5]. Some random processes are not described well using a truncated series expansion, particularly those involving only one or two terms in the expansion. The truncated series biases the random process being modeled, resulting in erroneous results in some cases. The Monte Carlo method is a brute force approach which is effective for a wide range of problems and random processes. Its primary drawback is its large computational expense. Algorithm 1 illustrates a typical Monte Carlo method.

`Algorithm 1:  Monte Carlo Analysis`

`set` $realization = 1$

```
while (realization < MAX_REALIZATION) do
    apply random variables
    solve the problem
    store results
    set realization = realization + 1
end
analyze results statistically
```

MAX_REALIZATION is the number of Monte Carlo realizations necessary to achieve the desired interval of confidence. If the problem being analyzed has a high degree of variability, the number of realizations will also be high [7]. The large number of realizations help ensure that the simulation is statistically valid.

In reliability analysis, random variables are usually used to describe aspects of the problem possessing spatial variability [8]. Figure 1.6 illustrates the difference between high and low spatial variability. Each shade of gray represents a different value of the material property. Common properties modeled in this fashion include material

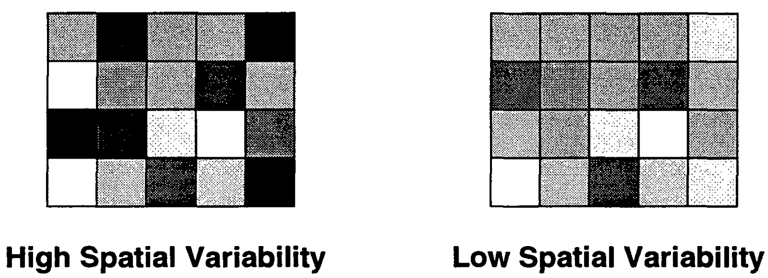

FIG. 1.6. Spatial Variability

strength, shear modulus, porosity, density, and conductivity [3]. In general, however, random variables can be used to describe any property possessing randomness or large uncertainties. In addition to spatial variability, a random variable can also possess spatial correlation. Whereas spatial variability describes the range of a property, the spatial correlation indicates the degree of randomness in distribution of that property.

Figure 1.7 illustrates the difference between high and low spatial correlation. A

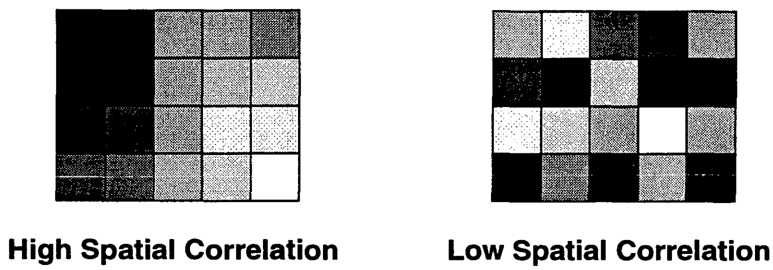

FIG. 1.7. Spatial Correlation

current area of research in the use of random variables is their combination with the finite element method, a common solution technique for engineering problems. In practice, both the series expansion and the Monte Carlo method are used in the application of random variables to the finite element method.

1.1 The Finite Element Method

The finite element method was originally developed in the 1960's to solve structural problems in aircraft design [9]. Since then, the method has been heavily researched and applied to a variety of different fields, including engineering, medicine, biology, and physics. The finite element is a very practical tool for solving partial differential equations in that it deals well with irregular problem domains and boundary conditions. However, this method can be computationally intensive, particularly for three dimensional problems.

The first step of the finite element method is the discretization of the problem domain. Fig. 1.8 illustrates an elasticity problem in which a cavity in a continuous material is subjected to external forces. The problem is then discretized into a set of simple geometrical elements and appropriate boundary conditions are applied to

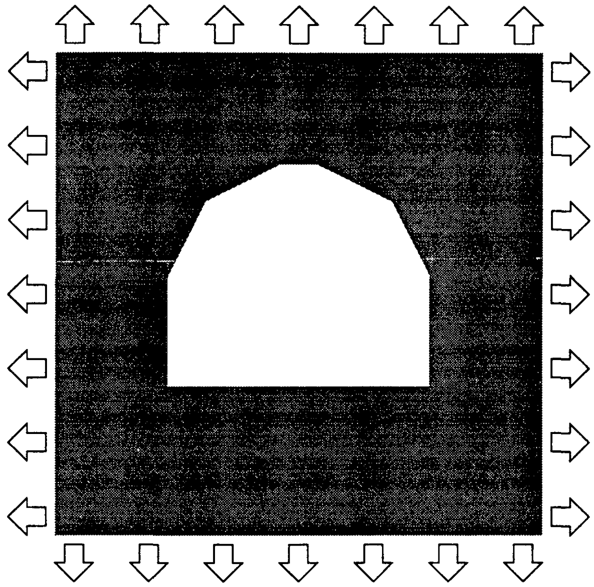

FIG. 1.8. Model of a Cavity in a Continuous Material with External Forces

the discretized problem. See Fig. 1.9. Each of the elements used to discretize the

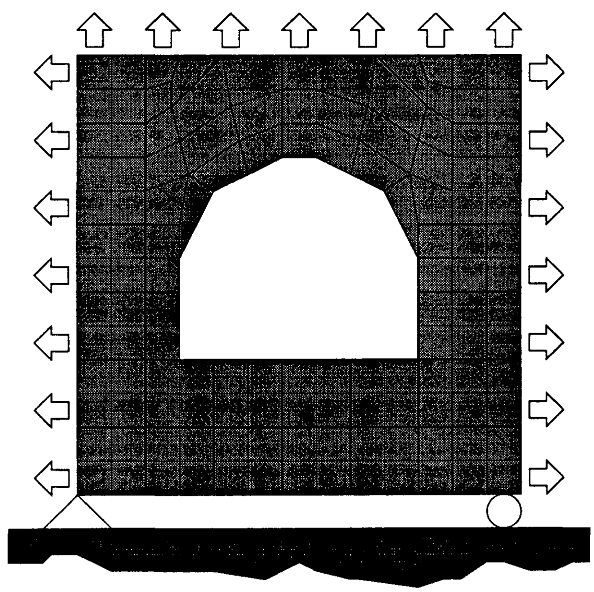

FIG. 1.9. Discretized Model with Appropriate Boundary Conditions

problem domain is a simple geometrical shape. These elements serve to model small finite pieces of the problem, hence the name finite element. The true power of the finite element method lies in the use of nodal interactions to represent the transmission of internal element forces between adjacent elements. Fig. 1.10 is a closeup of a single element from the cavity problem above. The arrows at each of the node represent the forces and displacements in the x and y directions for each node. Since the node is allowed to displace in both the x and y direction, this node is said to have 2 local degrees of freedom. For a 3D elasticity problem, nodes tend to have 3 local degrees of freedom. There are several different variations of the finite element method. In

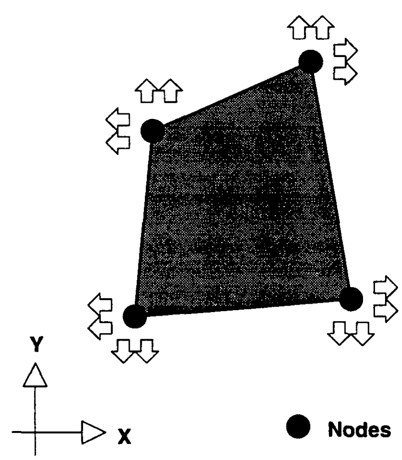

FIG. 1.10. Closeup of a Finite Element

this particular variation, displacements within the element away from the nodes are interpolated using a set of shape functions. Shape functions are simple functions, such as low order polynomials, that represent the solution of the governing equation across the element. These shape functions handle the interactions between the nodes within the elements as the solution at any point within the element can be approximated with the following relationship:

$$u(x,y) = \sum_{i=1}^{n} N_i(x,y)u_i \ . \tag{1.2}$$

In (1.2), the approximate solution u at the point (x, y) is sum of the shape function values $N_i(x, y)$ multiplied by the solution at nodes u_i. This approximation can be applied to the governing partial differential equations and the solution at the nodal points computed.

The end result of the finite element discretization process is a linear system representing the original set of partial differential equations and boundary conditions.

$$Au = b \ . \tag{1.3}$$

The values of the approximate solution at the nodes are represented in this system as the variables in the solution vector u. A is called the stiffness matrix and its coefficients represent the influence of each element on the nodes. Finally, b is the load vector and it contains the forces assumed acting on the individual nodes.

Boundary conditions are treated algebraically within the stiffness matrix. Dirichlet boundary conditions are the most common type of boundary condition found in elasticity. This type of boundary condition can either specify a nodal displacement $(u_i = u_0)$ or a constrained node $(u_i = 0)$. In the stiffness matrix, Dirichlet boundary conditions are accommodated through the use of the stiff spring method [10]. Assuming that u_i is set to some value u_0, the following changes would be made to the linear system:

$$A_{ii} \leftarrow A_{ii} + K, \tag{1.4}$$

$$b_i \leftarrow b_i + (A_{ii} + K)u_0, \tag{1.5}$$

where K is a real constant with a magnitude several orders greater than any of the other matrix coefficients. In the stiffness matrices arising from the research, $K = 1.0 \times 10^{20}$.

1.2 The Problem

The combination of finite element method with the Monte Carlo method represents an extremely large computational undertaking. In light of the fact that the computing resources available to most engineers and analysts are limited to workstations and PCs, the practical application of this type of analysis is currently restricted to those problems which do not yield to series expansion methods.

This research investigates ways of improving the performance of the Monte Carlo method using the finite element method for applications in the field of geomechanics. Funding for this research was obtained from a private company in Denver, Maptek/KRJA Systems, Inc. Maptek is currently developing a finite element module for *VULCAN*, their 3D mine design and geological mapping software package. The research was supported for two main reasons: 1) possible marketable concept in the Monte Carlo method and 2) improved performance of their own finite element software.

Referring back to the Monte Carlo algorithm described above, the computational bottleneck usually arises from the need to solve the large linear systems created by the finite element method in each Monte Carlo realization. Since the Monte Carlo method requires many realizations, even small improvements in this step of the process will yield substantial gains in the overall performance of the algorithm. Hence, this research is looking at ways of improving the performance of finite element method.

The performance of the finite element method must be improved in two areas: speed and robustness. The first area of improvement is necessary to make the Monte Carlo method practical for routine analysis in the engineering industry. The second area is linked to the first in that an extremely fast algorithm is virtually worthless if it is only applicable to a small class of problems.

1.2.1 The Model

The problem being studied in this research is the analysis of various geomechanical structures. This type of problem deals with the deformation of geological structures under loading. Practical examples include pillar strength analysis, slope stability, tunnel convergence, and ground subsidence. This problem requires a mathematical model for the behavior of rock and soil under loading. The characteristics of the problem that will be of interest are listed below:

Model:

2D/3D

Elastic-Plastic

A nonlinear material model that assumes the deformation of the material prior to the yield point is elastic and that the material will return to its original undeformed shape after the load is removed. After the yield point is exceeded, the material undergoes plastic deformation which results in permanent deformation of the material.

Vector Fields

A field refers to the set of unknown variables being modeled. In this case, the field variables are nodal displacements. Generally, 2D/3D structural analysis problems are vector field problems.

Large Irregular Domains

The problem geometry is not limited to simple geometric shapes and is assumed to have complicated boundaries.

Dirichlet Boundary Conditions

Dirichlet boundary conditions specify the field values at the boundary of the problem domain. In elasticity problems, this type of boundary condition indicates a fixed displacement along a boundary of the problem.

Material

Continuous

A continuous material is assumed to contain no cracks or fractures.

Heterogeneous

A heterogeneous material has material properties that vary from point to point within the material (i.e. the material has spatial variability).

Isotropic

The material properties are independent of direction.

1.2.2 The Finite Element Method in this Research

Just as the model describing the problem had several assumptions, the finite element method used to solve it also has several important assumptions. These are described below:

Solution Method:

Finite Element Method

Meshes

Unstructured Meshes

Unstructured meshes refers to the nodal numbering scheme. In an unstructured mesh, the physical position of the nodes do not reflect their numbering.

High Mesh Resolution

The mesh resolution refers to the number of elements used to cover the problem domain. Problems involving highly variable materials require high mesh resolutions in order to accurately model the material.

Elements

Quadrilateral/Hexahedron Elements

Quadrilateral elements are 4 sided elements. Hexahedron elements are general brick shapes with 8 vertices and 6 faces (see Fig. 1.11).

Quadratic Shape Functions

The functions used to approximate the solution across each element are second order polynomials.

Poor Element Scaling

The elements comprising the mesh are not uniform or regular. Uniformity implies that all of the elements are generally the same size and shape. Regularity, on the other hand, refers to the geometry of each element. Examples of regular elements include squares and cubes.

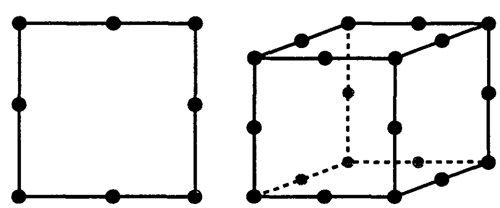

FIG. 1.11. 8-Node Quadrilateral Element and 20-Node Hexahedron Element

1.3 Computing Performance Limitations

As mentioned previously, the core of the finite element method consists of the solution of the linear system given in (1.3). However, the task of finding x is not a simple problem as the stiffness matrices arising from finite element models are often very large and possess irregular structures. The computational complexity depends on the algorithm used to solve the linear system and the computing hardware on which the algorithm is run.

The most common type of computing environment in use by engineers today is the workstation local area network (LAN). A LAN consists of several workstations linked together via cables, such as Ethernet or fiber optics. LANs allow the sharing of system resources, such as disk space, software libraries, printers, and CD-ROM drives as well as provide E-mail between users.

For the practical application of the Monte Carlo method in geomechanics, the method should run well on this type of computing environment. For large problems, the limiting factor to software performance on workstations is not the processor speed, but the amount of available memory. Modern workstations supplement their RAM with a disk-based swap file. As the RAM fills during software execution, the workstation swaps idle filled sections out to the disk. Since disk access times are far greater than memory access times, the CPU is forced to wait until the appropriate data become available. Depending on the amount of swapping, performance can degrade substantially, resulting in runtimes 30 to 50 times longer than runs without swapping.

1.3.1 Example of Performance Degradation due to Swap

In this example, a model of a mine pillar is run on an SGI Indigo 2 workstation with 64 Mb of RAM and a 2 Gb system disk. The pillar mesh contains 2000 elements which generated a $28,905 \times 28,905$ stiffness matrix. The combined memory requirements for the code and data structures is approximately 58 Mb.

Figure 1.12 illustrates the state of the system prior to the running of the finite element code. This figure is obtained from a snapshot of the `gr_osview` command in IRIX 5.3. This command provides a graphical system monitor which reflects the current usage of system resources. In this case, the monitor is configured to display the CPU utilization, CPU wait, and memory. The top two bars scroll from right to left and the height of the bars reflects the values of the CPU usage and CPU wait respectively. The bottom bar displays the amount of memory in use by coloring a corresponding area of the bar. In this figure, the current memory usage is approximately

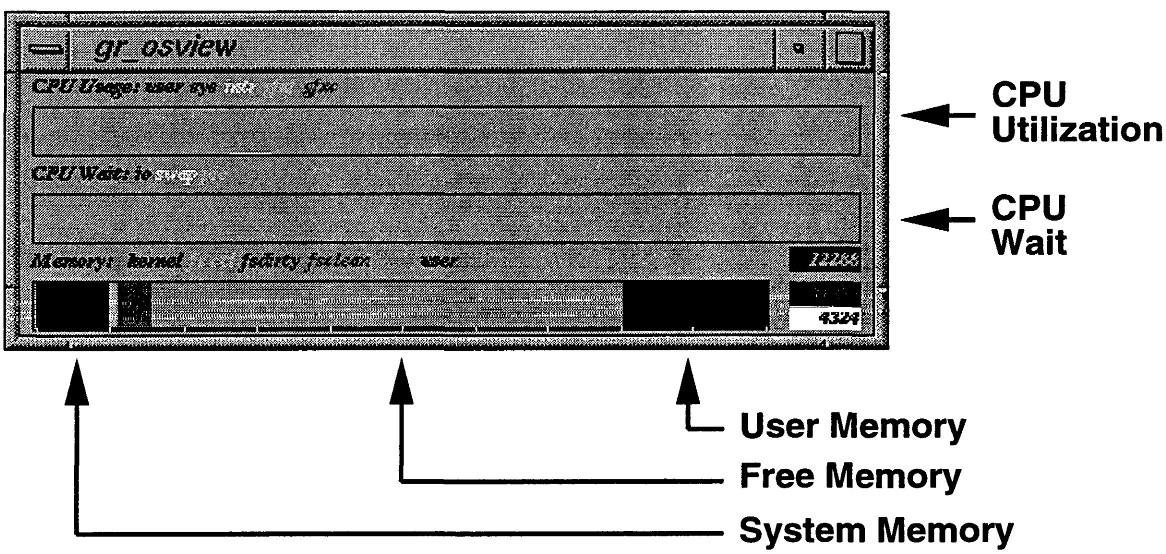

FIG. 1.12. System Status prior to Program Execution

35% of the total memory and is split between the UNIX kernel, file system, and user shells. At runtime, only about 42 Mb are free for use. Since this value is substantially less than the 58 Mb required by the program, this program will utilize the disk swap file. In addition, the data storage is fragmented, which implies that related data items are scattered throughout the memory. Fragmentation occurs in this example due to the sparse matrix storage technique used by the finite element code. Instead of storing data in contiguous memory blocks defined by variable declarations, this data structure uses `malloc` calls to define storage areas during program execution. While this method is very flexible and efficient, it does have the unfortunate tendency to scatter the data throughout the physical memory, resulting in severe fragmentation.

Figure 1.13 represents the state of the system just after the start of the finite element program. The program is in the process of loading the entire stiffness matrix and this is reflected by the large decrease in the amount of free memory. CPU utilization at this point is high as the program reconstructs the stiffness matrix from the data stored on the disk. Although the CPU does have some waits, they have little effect on the degree of CPU utilization.

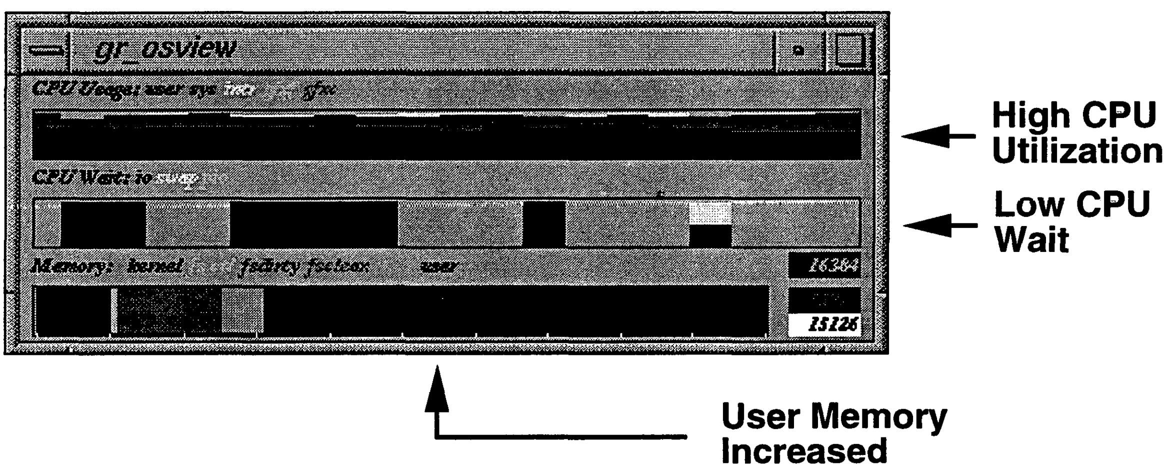

FIG. 1.13. System Status during the Loading of Stiffness Matrix Data

In Figure 1.14, the stiffness matrix has been fully loaded and the program is beginning to iterate to a solution. Core memory is nearly completely full with the UNIX and file system usage curtailed and a small amount of free memory set aside for temporary storage. Approximately 13 Mb has been swapped out to the hard disk. CPU utilization drops significantly as CPU waits are nearly continuous.

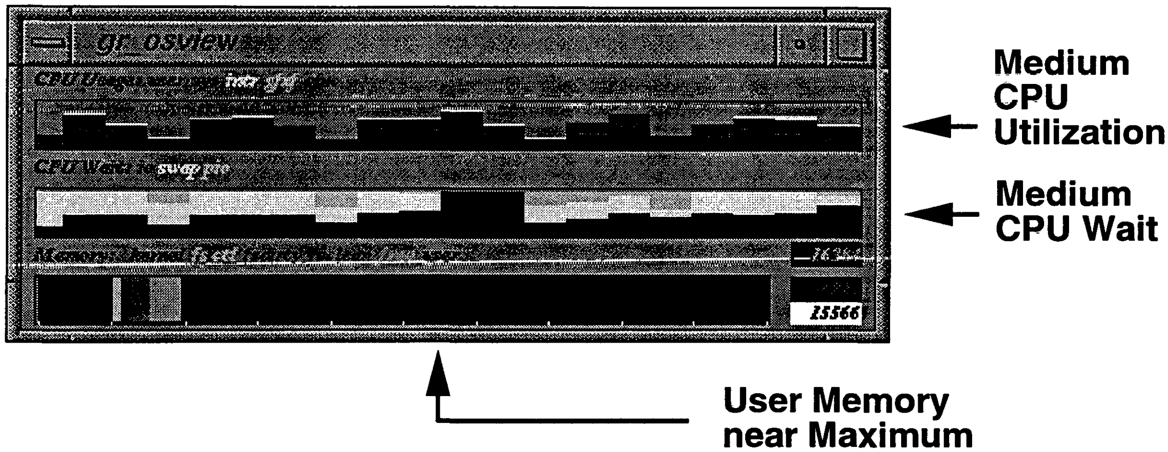

FIG. 1.14. System Status prior to start of Iterative Solver

Figure 1.15 represents the system in the middle of a single iteration. CPU utilization is extremely low as the processor is forced to wait for data to be swapped to and from the disk. On average, the processor is operating at about 10% of its peak speed which translates into runtimes about 10 to 15 times longer than expected.

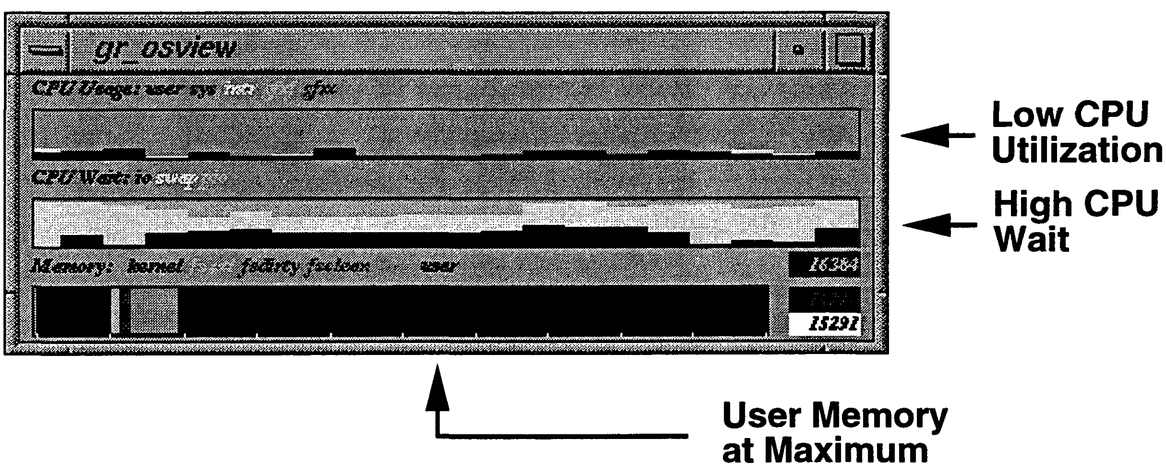

FIG. 1.15. System Status during Execution of the Iterative Solver

For some problems, a high level of swap activity can degrade the system performance to the point where the system is unable to maintain normal operation. This problem usually occurs with highly fragmented data structures. The computer is forced to dedicate so much of its resources to maintaining the large virtual memory and when conflicts arise over the use of those resources, the computer resolves them poorly. The end result of such conflicts varies depending on what processes are causing the conflict and on the amount of swap activity. Typical results include termination of the one or both of the processes, lockups, and system crashes.

1.4 The Research

Obviously, even the fastest solution algorithm will still run poorly if the computing resources are not sufficient. By the same token, a very slow algorithm will not produce spectacular results when run on a very fast system. The key is to combine a fast algorithm with the appropriate hardware to produce the best performance.

One aspect of the research investigates the use of distributed computing to overcome the performance degradation caused by disk swapping. This would greatly improve the execution speed for solving large linear systems on workstations with limited memories. The other aspect was to find a robust solver well suited for solving the linear systems arising from the finite element analysis of nonlinear geomechanics problems. In addition, the solver also had to be effective in a distributed computing environment.

Chapter 2 outlines the selection process for the appropriate linear solver, which in this case is the algebraic multigrid method (AMG). Chapter 3 outlines the AMG algorithm and its basic premise of operation. Chapter 4 deals with the parallelization for the AMG method onto a LAN and examines the benefits derived from the use of distributed computing for this problem. Finally, Chapter 5 contains the numerical results for several geomechanics problems and highlights the performance of the AMG method.

Chapter 2

SELECTION OF AN APPROPRIATE LINEAR SOLVER

In practical finite element analysis used in industry, (1.3) is typically solved using a direct method, such as *LU* factorization or Cholesky decomposition [11]. These methods are very robust, allowing the solution of a wide range of linear systems. In addition to their robustness, direct methods are very effective solvers for problems involving multiple load vectors. These methods typically decompose the matrix in upper and/or lower triangular pieces which are easily solved using forward and back substitution. For small to medium problems, the performance of direct methods is very good.

The primary drawback of direct methods when applied to large problems is their high memory requirements. During the factorization process, direct algorithms replace the zero entries in the matrix envelope with non-zero entries (see Fig. 2.1). Fill-in is very expensive in terms of memory as the number of zero entries in the matrix envelope is quite large. Direct solvers currently in use in most commercial finite element packages are highly optimized and make heavy use of hard disk storage to supplement small memories.

Due to these limitations, direct methods are not efficient solvers for very large problems [11]. The alternative to the direct methods is iterative solvers. These methods attempt to determine the solution of a linear system through a successive set of approximations. Iterative methods are attractive for the solution of large problems as they require far less memory that direct methods. For an iterative solver to be competitive with existing direct methods, it must run faster and require less storage

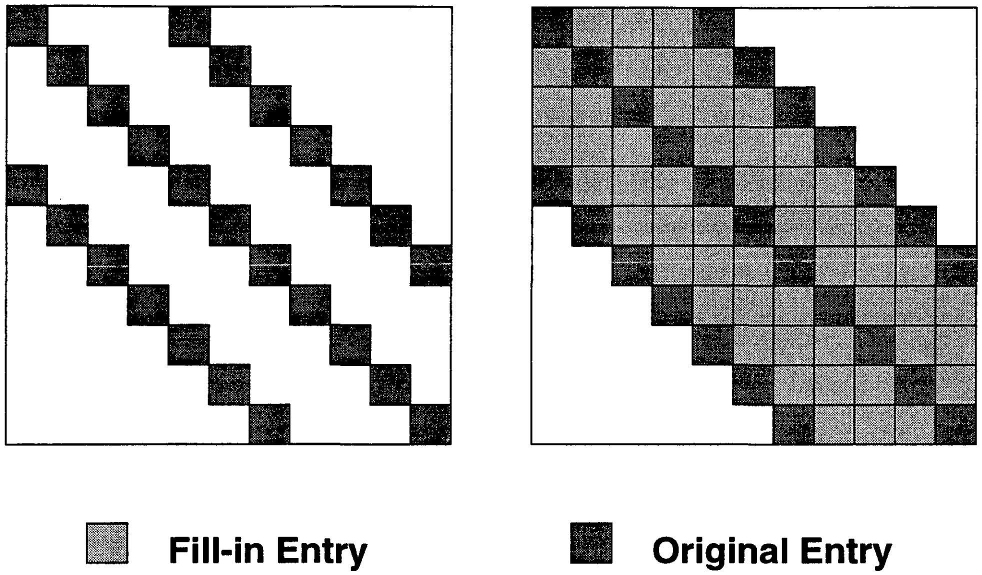

FIG. 2.1. Matrix Fill-in

while maintaining the same degree of robustness [11].

The three aspects most critical to the performance of iterative methods are the matrix structure, size, and conditioning. Since the stiffness matrix arises from the discretization of a partial differential equation (PDE) over some domain, those aspects will be dependent on the type of PDE being solved as well as the discretization method. An additional aspect of the solution process not directly related to the stiffness matrix is the handling of multiple load vectors b.

2.1 Matrix Structure

The properties of the matrix encompass several different facets of the matrix, including sparsity, bandedness, and definiteness. In general, the regularity of the matrix properties greatly influences the efficiency of its solution. A diagonal matrix with positive entries is an example of a very sparse symmetric banded positive definite matrix The solution of a linear system involving this matrix is trivial. On the other hand, a linear system containing a dense unsymmetric indefinite matrix is very difficult to solve.

Sparse matrices typically arise from the discretization of partial differential equations. In a finite element discretization, the variables in the linear system contain the values of the approximate solution at the nodes. Off-diagonal entries in the stiffness matrix describe the degree of connectivity between nodes. Since most nodes in a mesh are connected topologically to only a few adjacent nodes, this implies that many of the row entries in the stiffness matrix will be zero and the resulting matrix will be very sparse. For most large problems, the matrix density is usually much less than 1% of the total matrix size. The efficiency of iterative solvers is related to the sparsity of the matrix in that the number of nonzero entries within the matrix dictates the number of operations performed during each iteration of the solver.

Bandedness is also determined from the mesh and is directly related to the global numbering of the nodes. If the nodes are numbered optimally, the distance between nodal indices is minimal and the stiffness matrix assumes a regular banded structure. Once in this form, the matrix can be solved using several different solvers which exploit this structure to obtain very efficient solutions. However, optimal numbering is difficult to obtain for irregular geometries and unstructured meshes. In practice, most stiffness matrices have a very ragged banded structure which limits the effective application of most banded solvers.

A matrix is positive definite if

$$x^T A x > 0 \qquad x \epsilon \Re^n,\ x \neq \vec{0}\,. \tag{2.1}$$

where $\Re^n$ is the n-dimensional, real Euclidean space. Definiteness in a matrix is extremely useful in determining the eigen properties of the matrix. For those iterative solvers which rely heavily on the matrix's eigenvalues and eigenvectors, the definiteness of the matrix is very important to the rate of convergence of the solver. This is especially true of methods like steepest descent and conjugate gradient methods. Elliptic, self-adjoint partial differential equations generate symmetric positive-definite matrices.

2.2 Matrix Size

Matrix size is extremely important to the performance of a linear solver as it dictates how the matrix will be stored. In order to avoid swap performance degradation, the data structure used to store the sparse matrix should be as efficient as possible. If the matrix is symmetric, the simplest method of reducing storage is to store only half of the matrix and modify the matrix-manipulation routines to reflect this storage. In this research, the whole matrix is stored due in order to avoid the necessity of modifying the matrix manipulation routines.

The primary factors in the size of the stiffness matrix are the number of nodes and number of local degrees of freedom (DOF). For vector field problems, such as elasticity, the number of local DOF is generally the same as the dimensionality of the problem. For example, three dimensional problems typically have three local degrees of freedom. Thus, for a problem with n nodes and m local DOF, the size of the matrix is $nm \times nm$. Table 2.1 illustrates the sizes of some of the test meshes used in this research.

While the 2D and 3D problem have an equivalent number of elements, the resulting stiffness matrices vary greatly in size. The large jump in the computational complexity from 2D to 3D often forces engineers and analysts to use 2D approximation of 3D problems.

Table 2.1. Meshes and Stiffness Matrix Sizes

Problem	Elements	Matrix Size	Memory (Mb)
2D Foundation	960	6,034 × 6,034	9
3D Pillar (small)	1,000	14,883 × 14,883	32
3D Pillar (large)	2,000	28,905 × 28,905	56

The number of nodes is determined by the number and type of elements used in the mesh. In the Monte Carlo analysis, the mesh resolution (and hence the number of elements) is dictated by the degree of spatial variability in the material being modeled. Highly variable materials, such as soils, will require a high mesh resolution for statistical purposes and consequently a large stiffness matrix. Additionally, the order of the shape functions used in the element is very important to the number of nodes. A linear element (i.e. an element using linear shape functions) requires fewer nodes than a quadratic or cubic element. For example, a linear hexahedral element uses 8 nodes as compared to the 20 and 32 nodes used by the quadratic and cubic hexahedral elements.

Many practical problems in finite element analysis require 10^3 to 10^4 elements [11],[12],[13],[14]. As the stiffness matrix is about an order of magnitude larger, stiffness matrices in the range of 10^4 to 10^5 can be expected.

2.2.1 Sparse Matrix Storage

Since the primary performance limitation of computing on workstations is insufficient memory, an efficient sparse matrix storage technique must be used to avoid wasting core memory and causing swap performance degradation. A variety of data structures exist for the storage of sparse matrices. Most sparse storage techniques store only the nonzero matrix entries to save both memory and computational expense. Since the primary operation in most iterative solvers is a matrix-vector multiplication, the presence of zeros during computation results in wasted calculations.

Matrices with highly regular structures, such as banded symmetric matrices, use storage methods which exploit their structure. Many of the banded storage structures rely on this principle by only storing diagonals containing nonzero entries. However, if these types of methods are applied to matrices with irregular structures, the storage efficiency quickly degrades due to the storage of a large number of zeros entries.

The most common method for general sparse matrices is the row-column storage method. This method stores each nonzero matrix entry along with its row and column indices, although most implementations of this storage method define one of the indices implicitly. Depending on the type of operation, the entries in the structure can either be stored in an ordered format or in a random format. For matrix-vector multiplication, either format is appropriate as that product does not care in which order the matrix and vector entries are multiplied.

This research uses a very general implementation of the row-column algorithm. This implementation is not optimal in terms of speed and storage, but it is highly flexible and allows easy modification of the the matrix structure. The basic structure consists of an array of linked lists where each entry in the linked list is a block of matrix entries. The block size is dependent on the number of local DOF, which for the example problems yielded block sizes of 2×2 and 3×3. Due to the local DOF, the nonzero matrix entries are clustered in blocks of this size throughout the matrix, a property arising from the discretization process.

The data structure used in this research consists of an array of root cells containing the diagonal entries of the sparse matrix. Each row of the matrix is then represented as a linked list branching off the root cell corresponding to that row. Entries in the linked list are ordered in ascending column order with the first nonzero row entry adjacent to the root cell. Figure 2.2 illustrates a typical cell in the ith row. Cells are linked together by setting the link variable to the memory location

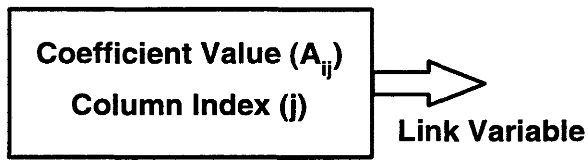

FIG. 2.2. Row-Column Cell

of the next cell in the list. Thus, the link variable points to the memory location of the next cell. This list is a singly linked list and can only be traversed from left to right. However, the addition of more link variables could allow the traversal in both directions along the row or even allow a column traversal. This data structure is row-oriented, which allows very fast row operations. However, accessing the columns within this structure is very slow due to the need to traverse each row looking for the appropriate column entry. This problem is somewhat alleviated, but not eliminated, by sorting the row by the column index. The block structure also reduces the row traversal time, but requires extra bookkeeping in order to keep track of the block indices and the indices of the entries within each block.

2.3 Conditioning

The condition number of a matrix is a measure of its singularity. This number is used as an indicator for the accuracy of the results from the solution of a linear

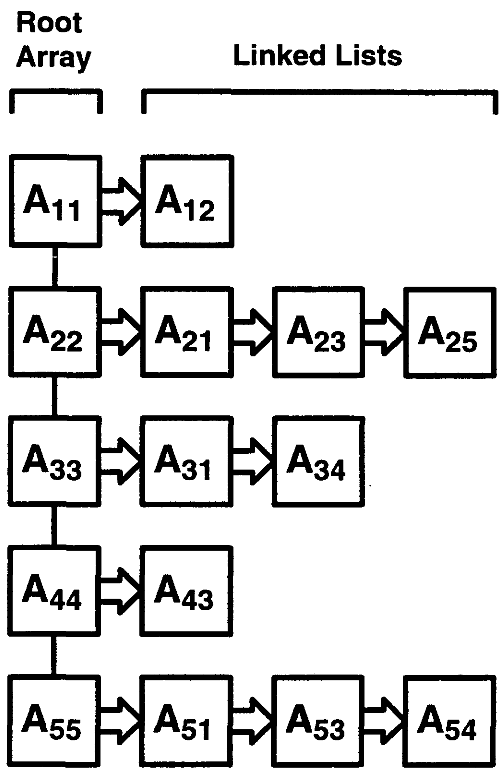

FIG. 2.3. Illustration of the Row-Column Data Structure

system or matrix inversion. The condition number is defined as

$$\kappa(A) = \begin{cases} \|A\|\|A^{-1}\| & \text{if } A \text{ nonsingular }, \\ \infty & \text{if } A \text{ singular }, \end{cases} \tag{2.2}$$

where $\|\cdot\|$ is a matrix norm. The condition number is rarely computed using this method due to the high computational cost involved with computing the inverse. Often, the condition number is estimated using a matrix factorization technique, such as the LU factorization.

If the matrix norm in (2.2) is $\|\cdot\|_2$ and the matrix is symmetric positive definite, then the condition number can be expressed as

$$\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}} , \tag{2.3}$$

where

$$\lambda_{max} = \max_j(\lambda_j) , \tag{2.4}$$

$$\lambda_{min} = \min_j(\lambda_j) . \tag{2.5}$$

λ_j, $j = 1, \ldots, n$ are the eigenvalues of the matrix A [15]. For symmetric positive definite matrices, these eigenvalues are both real and positive.

The conditioning of a matrix is particularly important during operations involving finite-precision arithmetic. Direct methods suffer the worst from ill-conditioning in that they accumulate numerical errors during the factorization of the matrix. Rounding errors caused by row and column updates propagate during the factorization process, resulting in very large numerical errors in the updates computed near the end of the algorithm. Iterative methods, on the other hand, do not accumulate numerical errors in quite the same way, resulting in better handling of ill-conditioned problems.

Ill-conditioning in finite element stiffness matrices can arise from several different sources, including variations in the mesh resolution, poor element aspect ratios, and the presence of discontinuities in material properties at element interfaces. The first two phenomena are common in the meshes that arise frequently during adaptive mesh refinement [16]. Adaptive refinement is currently used to improve the accuracy of the solution by modifying the position and size of the elements within the mesh. Figure 2.4 illustrates the result of mesh refinement. The presence of discontinuous material properties is frequently encountered in the modeling of heterogeneous materials. The

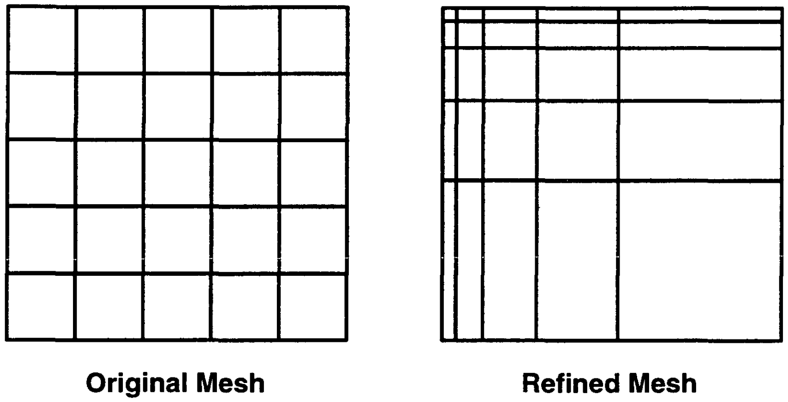

FIG. 2.4. Example of an Adaptively Refined Mesh

mechanism that causes ill-conditioning in each of the above methods is the same – poor scaling of the finite element shape functions. The example below uses a simple 1D elements to illustrate the problem.

2.3.1 Example of Ill-Conditioning in Finite Element Analysis

Fig. 2.5 describes a rod of length L being subjected to an axial distributed force of magnitude F. The rod has a uniform cross-section area (A) and stiffness (Young's modulus E). The governing differential equation from solid mechanics is

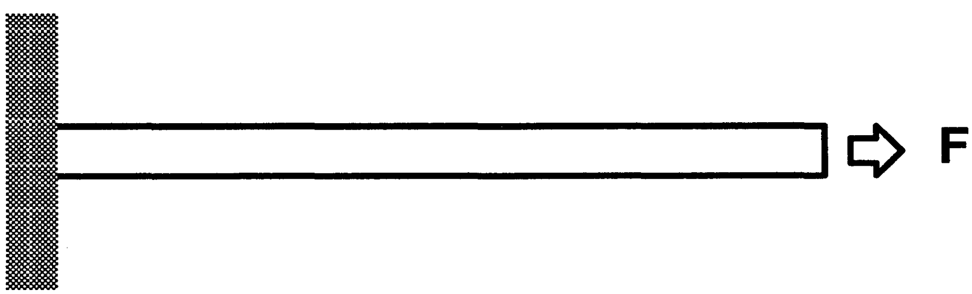

FIG. 2.5. Example Rod Problem - Axial Loading

$$EA\frac{d^2u}{dx^2} + F = 0 \ , \tag{2.6}$$

with

$$u(0) = 0 \ . \tag{2.7}$$

In this equation, the deformation of the rod u is related to the stiffness E, cross-sectional area A, and force F. Using 1D linear rod elements, the rod problem can be discretized into three finite elements. The linear rod element consists of two nodes and two linear shape functions N_1 and N_2. By approximating the continuous variable u using the nodal values, the governing differential equation can be reformulated into the following matrix-vector system:

$$EA\begin{bmatrix} \frac{1}{L_1} & -\frac{1}{L_1} & 0 & 0 \\ -\frac{1}{L_1} & \frac{1}{L_1}+\frac{1}{L_2} & -\frac{1}{L_2} & 0 \\ 0 & -\frac{1}{L_2} & \frac{1}{L_2}+\frac{1}{L_3} & -\frac{1}{L_3} \\ 0 & 0 & -\frac{1}{L_3} & \frac{1}{L_3} \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} \ . \tag{2.8}$$

The variables L_i, $i = 1, \ldots, 3$ are the element lengths and sum to L, u_i and F_i are the nodal displacements and nodal forces respectively. The stiffness matrix is is sparse, banded, and positive definite. The boundary condition at the wall is translated into a corresponding nodal displacement at Node 1.

$$u_1 = 0 \tag{2.9}$$

Deleting the row and column for u_1 to set the fixed end boundary condition and inserting the appropriate nodal forces, the above matrix-vector system becomes

$$EA \begin{bmatrix} \frac{1}{L_1} + \frac{1}{L_2} & -\frac{1}{L_2} & 0 \\ -\frac{1}{L_2} & \frac{1}{L_2} + \frac{1}{L_3} & -\frac{1}{L_3} \\ 0 & -\frac{1}{L_3} & \frac{1}{L_3} \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F_2 \\ F_3 \\ F_4 \end{bmatrix} . \tag{2.10}$$

Assuming the element lengths L_i, $i = 1, \ldots, 3$ are all equal (see Fig. 2.6), the problem reduces to

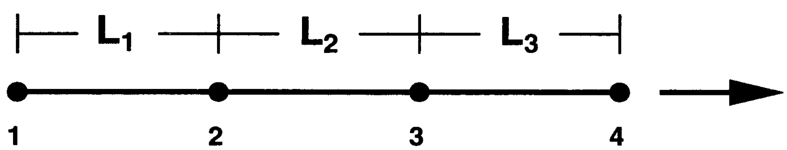

FIG. 2.6. Rod Problem with Equal Element Lengths

$$\frac{EA}{L_1} \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F_2 \\ F_3 \\ F_4 \end{bmatrix} . \tag{2.11}$$

Computing the eigenvalues for this matrix yields the following values for λ_{max} and λ_{min},

$$\begin{aligned} \lambda_{max} &= 3.2470 \\ \lambda_{min} &= 0.1981 \end{aligned} \tag{2.12}$$

which in turn generate a condition number of

$$\kappa = \frac{3.2470}{0.1981} = 16.391 . \tag{2.13}$$

Modifying the discretization slightly to reflect elements of differing lengths yields Fig. 2.7. For this example, the following element lengths are assumed

$$\begin{aligned} L &= L_1 + L_2 + L_3 \\ L_2 &= \frac{1}{10} L_1 \\ L_3 &= \frac{1}{100} L_1 \end{aligned} \tag{2.14}$$

Substituting the appropriate values yields this matrix-vector system

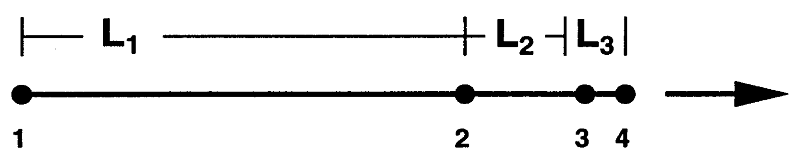

FIG. 2.7. Rod Problem with Unequal Element Lengths

$$\frac{EA}{L_1} \begin{bmatrix} 11 & -10 & 0 \\ -10 & 110 & -100 \\ 0 & -100 & 100 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F_2 \\ F_3 \\ F_4 \end{bmatrix} . \tag{2.15}$$

Computing the appropriate eigenvalues and condition number yields the following:

$$\begin{aligned} \lambda_{max} &= 205.395 \\ \lambda_{min} &= 0.3185 \end{aligned}$$

$$\kappa = \frac{205.3951}{0.3815} = 644.882 \ . \tag{2.16}$$

Note that the condition number has increased substantially with the change in the relative size of the elements. The effect of the change in the condition number between the two examples is not apparent until the problem is run using finite-precision arithmetic. The following values are used for the rod length, cross-sectional area, Young's modulus, and force.

$$\begin{aligned} L &= 3 \\ EA &= 100 \\ F_2 &= 0 \\ F_3 &= 0 \\ F_4 &= 1 \end{aligned}$$

This problem can also be solved analytically using the following relationship.

$$\delta L = \frac{F_4 L}{EA} \qquad (2.17)$$

Substituting the appropriate values for the constants and solving for δL yields the displacement of the end of the rod which also corresponds to u_3 in the finite element formulations.

$$\delta L = 0.03 \qquad (2.18)$$

The stiffness matrices and right hand sides for both finite element formulations were run through *MATLAB* using varying degrees of precision. The table below summarizes the results.

$\epsilon_{machine}$ is the smallest distance between real numbers represented under finite-precision arithmetic. As the number of bits comprising the mantissa of the floating point number decreases, so does the accuracy of the solution. The deviation in the

Table 2.2. Solutions under Finite Arithmetic

Mantissa (bits)	$\epsilon_{machine}$	u_3 (Example 1)	u_3 (Example 2)
53	2.22×10^{-16}	0.030000000000000	0.030000000000000
37	1.46×10^{-11}	0.029999999999515	0.029999999975871
21	9.54×10^{-7}	0.029999971389771	0.030000388622284
5	6.25×10^{-2}	0.028320312500000	0.008300781250000

results arises from rounding and truncation errors during floating point operations. Since both condition numbers are relatively low and do not represent ill-conditioned matrices, the effect of the conditioning is only visible under very low precision arithmetic. The first stiffness matrix is well-conditioned and manages to generate reasonable results even when subjected to extensive rounding errors. The second matrix, however, is not as well-conditioned and fails to produce a usable solution at the same level of precision.

Equivalently, the condition number can also be affected by leaving the element lengths uniform and varying the material properties in each element. This transforms (2.10) into

$$A\begin{bmatrix} \frac{E_1}{L_1}+\frac{E_2}{L_2} & -\frac{E_2}{L_2} & 0 \\ -\frac{E_2}{L_2} & \frac{E_2}{L_2}+\frac{E_3}{L_3} & -\frac{E_3}{L_3} \\ 0 & -\frac{E_3}{L_3} & \frac{E_3}{L_3} \end{bmatrix}\begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \\ F \end{bmatrix}. \tag{2.19}$$

Assuming $L_1 = L_2 = L_3 = 1.0$ and $E_2 = 10E_1$, $E_3 = 100E_1$, and $E_1A = 100$, (2.19) becomes

$$\frac{E_1 A}{L_1} \begin{bmatrix} 11 & -10 & 0 \\ -10 & 110 & -100 \\ 0 & -100 & 100 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ F \end{bmatrix} \tag{2.20}$$

which is the same as (2.15).

2.3.2 Conditioning and Iterative Solvers

In regards to the selection of an iterative linear solver, the conditioning of the matrix typically affects the convergence rate. Ill-conditioning will usually slow convergence, particularly for those methods relying on the eigen properties of the matrix. Conjugate gradient methods show decreased performance when applied to ill-conditioned problems and require the use of preconditioning to restore the convergence rate. The convergence rate of stationary methods is also affected by the degree of ill-conditioning in the system. However, the conditioning of the matrix is not as influential since the convergence rate is based on the eigen properties of a matrix derived from the original by splitting.

Ill-conditioning can be overcome through the use of preconditioning. A preconditioned linear system generally has the form

$$CAu = Cb \tag{2.21}$$

where the matrix C is the preconditioner. The matrix C is essentially an approximation of A^{-1} which has the effect of squeezing the eigenvalues of CA closer to unity. Preconditioners can often greatly reduce the number of iterations required by iterative solvers for convergence.

The optimal choice of a preconditioner tends to be a very difficult decision.

The simplest preconditioners, such as the diagonal preconditioner, are very simple to construct and require very little storage. However, these preconditioners achieve limited improvement in the convergence rates. The more complicated preconditioners, such as incomplete factorizations and matrix splittings, often greatly increase the rate of convergence. In general, the construction of these preconditioners is problem dependent and usually expensive in terms of memory and computation. The cost of constructing a preconditioner should always be weighed against the improvement in the convergence rate.

2.4 Multiple Load Vectors

Multiple load vectors do not directly affect the performance of the linear solver, but they do affect the overall runtime for the finite element method. For problems involving multiple load vectors, the linear solver should be very efficient at resolving the system for each new vector. Multiple load vectors can arise from several different sources, but in the case of this research, the source is material nonlinearity.

A common mathematical model of material behavior assumes elastic-perfectly plastic stress-strain behavior. Defining the material stiffness as

$$E = \frac{\sigma}{\epsilon}, \tag{2.22}$$

the stiffness E is constant prior to the yield point. Beyond the yield point, the stiffness E follows a linear relationship. Fig. 2.8 illustrates this type of material behavior.

Unfortunately, most materials do not behave in this fashion. The stress-strain relationship is rarely linear, particularly near the yield point. Fig. 2.9 illustrates typical material behavior.

Material nonlinearity can be modeled in two different ways. In the first approach,

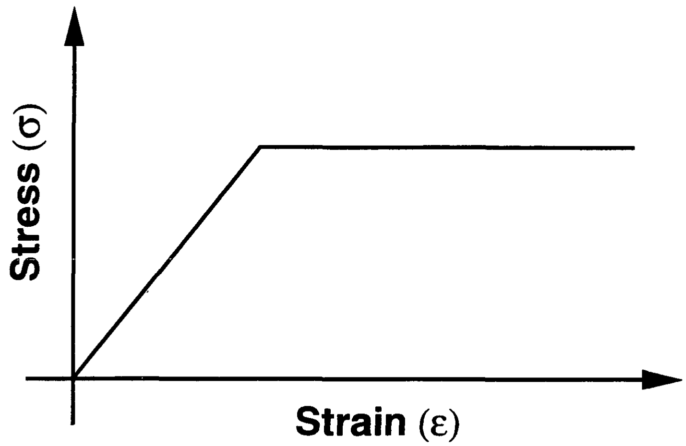

FIG. 2.8. Elastic-Perfectly Plastic Material Behavior

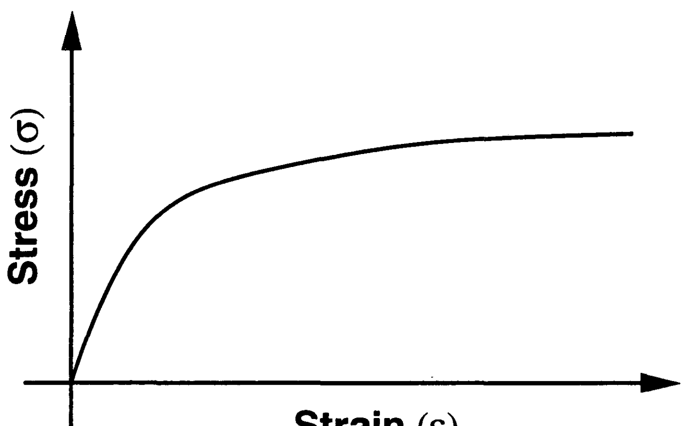

FIG. 2.9. Realistic Material Stress-Strain Behavior

the nonlinearity is handled by iteratively updating the load vector. This method is known as the constant stiffness method as the stiffness matrix is not modified during the iterations. In the second approach, the stiffness matrix is updated to reflect the change in the stiffness of the material as it approaches failure. This method is known as the variable stiffness method.

Both methods have advantages and disadvantages. The constant stiffness method avoids the necessity of reforming the stiffness matrix. However, this method also requires substantially more iterations near failure as the original stiffness matrix overestimates material stiffness. The variable stiffness method requires fewer iterations, but has the added cost of reforming the stiffness matrix. This research uses the constant stiffness method because of its simplicity.

In terms of the finite element analysis, the only difference to the core problem described in (1.3) is the presence of multiple right hand sides (RHS). Thus, (1.3) becomes

$$Au = \{b^{(0)}, b^{(1)}, \ldots, b^{(m)}\} \ . \tag{2.23}$$

$b^{(0)}$ is the original load vector. All subsequent load vectors $b^{(i)}$, $i \neq 0$, are of the form $b^{(i)} = b^{(i-1)} + f^{(i)}$ where $f^{(i)}$ is a vector containing the self-equilibrating forces computed from the stress distribution caused by the $b^{(i-1)}$ load vector. Generally, f_i results in a small perturbation of $b^{(i-1)}$ so that $b^{(i)} \approx b^{(i-1)}$.

2.5 Selection of an Appropriate Linear Solver

The combination of all the assumptions yield the following problem.

$$Au = \{b^{(1)}, b^{(2)}, \ldots, b^{(m)}\} \tag{2.24}$$

$A \rightarrow$ Stiffness Matrix

Large (10^4 to 10^5 rows)

Sparse (<1% Nonzero Entries)

Irregular Banding

Positive Definite

Ill-Conditioned

$$
\begin{aligned}
b^{(i)} \quad \rightarrow \quad & \text{Nodal Forces} \\
& b^{(i)} = b^{(i-1)} + f^{(i)} \\
& b^{(i)} \approx b^{(i-1)}
\end{aligned}
$$

The appropriate linear solver for this type of problem must be able to handle large sparse ill-conditioned positive definite matrices with multiple similar RHS. The two most competitive choices of the iterative type for this type of linear system are conjugate gradient and multigrid methods [11].

2.5.1 Conjugate Gradient Methods

Conjugate gradient (CG) methods are derived from the method of steepest descent. This type of method attempts to find iteratively the solution to $Au = b$ by performing a set of line searches using the residual to determine the appropriate search direction [17]. Unlike the steepest descent method, the conjugate gradient method uses orthogonal search directions which ensures convergence within n iterations (in infinite-precision arithmetic) where n is the size of the matrix. In practice, the convergence of the CG method occurs in fewer iterations than n, although ill-conditioned problems might require more than n iterations.

Convergence can be accelerated through the use of preconditioning. Depending on the type of preconditioner, the preconditioned conjugate gradient (PCG) method can substantially reduce the number of iterations required for convergence. The preconditioned conjugate gradient (PCG) method is fast becoming the standard iterative solution method for large linear systems arising from the discretization of partial differential equations.

CG and PCG methods do have limitations. First, these methods achieve their best performance on well-conditioned positive definite matrices. For very ill-conditioned matrices or matrices which are nearly indefinite, the convergence rates can be extremely slow. Unfortunately, these types of matrices often arise from practical problems, requiring effective preconditioners or modifications to the algorithms to handle indefiniteness. Second, these methods are usually not effective at handling multiple similar RHS. Depending on the difference between the previous RHS and the new RHS, the number of iterations required for convergence may not be very small. This disadvantage arises from the search direction nature of these methods. Finally, the most effective preconditioners require substantial memory and computation to construct. Depending on the amount of available system resources, the application of an effective preconditioner may be very difficult.

2.5.2 Multigrid Methods

Multigrid (MG) methods are a class of fast iterative numerical methods for solving the linear systems arising from partial differential equations. Multigrid methods work on the principle of accelerating convergence of stationary iterative methods through the use of a set of small systems called coarse levels derived from the original system. These coarse levels supplement the local solution information in the stationary solver with global solution information. MG methods are split into two classes: geometric and algebraic.

Geometric Multigrid

Geometric methods are currently the most popular methods due to their efficiency. Geometric MG creates its coarse levels through manipulation of resolution of the mesh describing the problem domain. A coarse level is generated by taking the mesh at the current level and reducing the resolution. The new coarse mesh is then assembled into its corresponding stiffness matrix. Figure 2.10 illustrates these coarse meshes and the corresponding coarse levels.

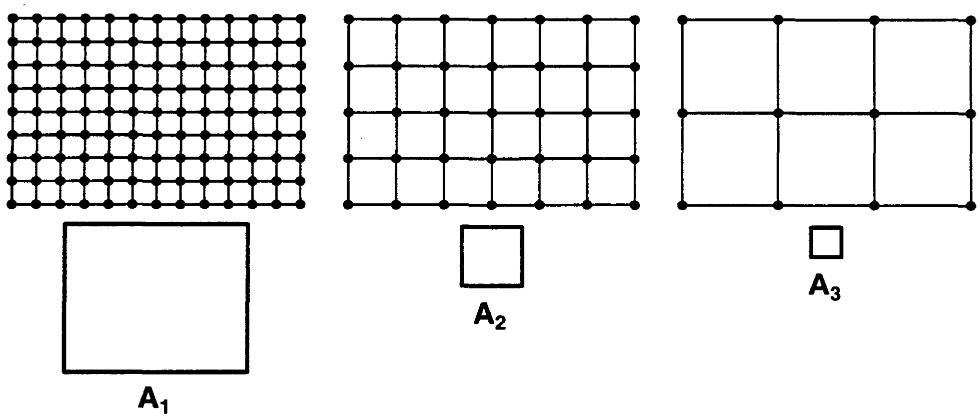

FIG. 2.10. Geometric Multigrid - Coarse Meshes and Corresponding Levels

Geometric MG often uses matrix-less techniques and generates the local solution information directly from the mesh instead of using a matrix derived from the mesh. This allows for the solution of extremely large problems on machines with limited memories. However, this approach is limited to structured meshes in which the nodal numbering is well-ordered.

The primary disadvantage of the geometric methods is their limited practical application. First, geometric methods do not deal well with unstructured meshes. The varying element sizes and unordered node numbers require extensive bookkeeping and often result in inefficient coarse levels and poor performance. This problem can be circumvented by closely coupling the multigrid method with the mesh generation and finite element analysis software [18], [19]. Second, geometric MG has troubles with discontinuous material properties which limit the degree of coarsening in the mesh. This problem is handled in the same way as the unstructured meshes.

Algebraic Multigrid

Algebraic multigrid (AMG) methods achieved notable popularity in the mid 1980s, but have since fallen into disuse. Algebraic multigrid methods generate their coarse levels directly from the stiffness matrix using a variety of methods to determine the nodal connectivity from the matrix coefficients. The most notable drawback and perhaps the main reason for their declined use is the required storage of the stiffness matrix and coarse levels. In addition, AMG suffers from relatively slow convergence in comparison to geometric MG and CG methods. Finally, AMG requires a substantial amount of setup time in the generation of the coarse levels, which can represent as much as 80% of the total algorithm runtime [20].

The redeeming factor inherent to the AMG methods and which serves to offset its drawbacks is its robustness. Unlike the geometric MG methods, AMG works well with unstructured meshes and discontinuous coefficients [21]. Since AMG simply accelerates stationary iterative methods, the AMG is well-suited to multiple similar RHS where $b^{(i)} \approx b^{(i-1)}$. Also, ill-conditioning does not affect AMG as greatly as it does the conjugate gradient methods due to the difference between their operating principles. Finally, AMG works only with the stiffness matrix and does not require any information from the finite element mesh. This property is attractive to users of commercial software where detailed mesh information is difficult or impossible to export to external software and allows AMG to be used with existing commercial software packages without extensive modification.

The next chapter describes in more detail the algebraic multigrid algorithm used in this research.

Chapter 3

ALGEBRAIC MULTIGRID

The beginnings of the algebraic multigrid methods can be traced back to the 1960s to the work of Russian mathematicians [22]. The original algorithms used relaxation techniques to solve the finite difference problems arising from the discretization of elliptical partial differential equations. In the original work, the coarse grids were generated without obtaining information from the discretized system. In the 1980's, work on problems involving strongly discontinuous coefficients led to changes in the algorithms that tightly coupled the coarse grid generation with the original problem. The modern version of the algorithm was presented by Brandt, McCormick, and Rüge [23] in 1984. However, research interest shifted to geometric multigrid methods and most recent work in AMG has been in the area of improving convergence through better methods of generating coarse levels.

Until recently, AMG suffered from poor convergence rates. This problem arises from the method generating the coarse levels in the problem. The original premise behind the coarsening process outlined in [24] was to emulate the behavior of the geometric multigrid coarsening process. Unfortunately, this type of approach is difficult to implement optimally when the only source of information regarding the problem geometry is the resulting stiffness matrix. Recently, P. Vaněk J. Mandel, and M. Brezina at the University of Colorado at Denver have taken a different approach to the generation of coarse levels through a method called smoothed aggregation. This method was selected for this research due to its improved convergence rate [25],[26].

The AMG method works through the acceleration of stationary iterative solvers such as Jacobi, Gauss-Seidel, and SOR. A property inherent to iterative methods of this type is smoothing of the residual. During each iteration, the stationary iterative method reduces the high frequency components of the vector. Frequency can be considered as the gradient between adjacent vector components. Figure 3.1 illustrates the difference between high and low frequency vector components. In the figure, the

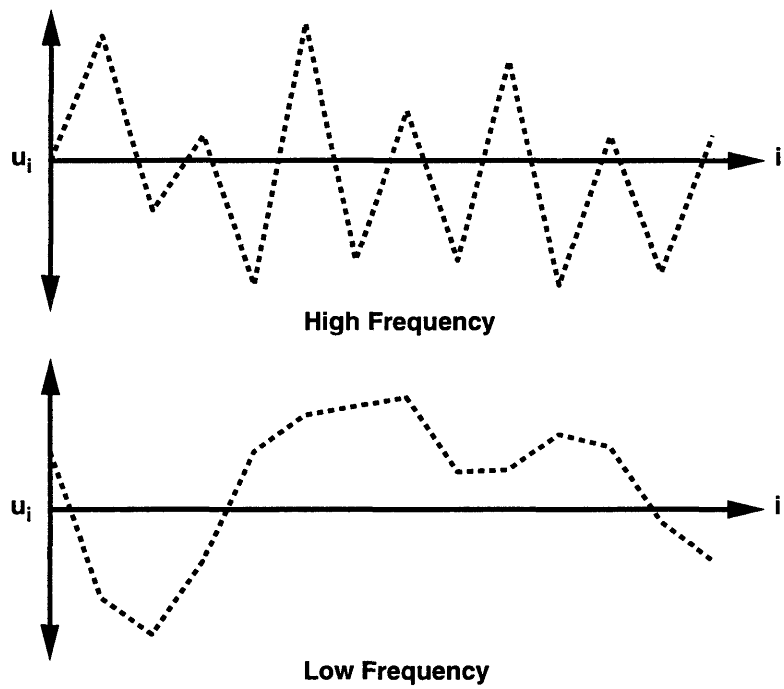

FIG. 3.1. Frequency in the Vector Components of $\vec{u}$

vector $\vec{u}$ has been displayed as a line graph with the x-axis the index i of the vector component and u_i the ith component of the vector $\vec{u}$.

The smoothing property results in the rather slow convergence of these methods. During the first few iterations, the approximate solution is quite different from the actual solution, resulting in high frequency error in the residual. The stationary method quickly reduces these components, leaving behind low frequency error in the residual. Figure 3.2 illustrates the residual of a small linear system prior to iteration using point Gauss-Seidel. Note that the residual vector has a large amount of high

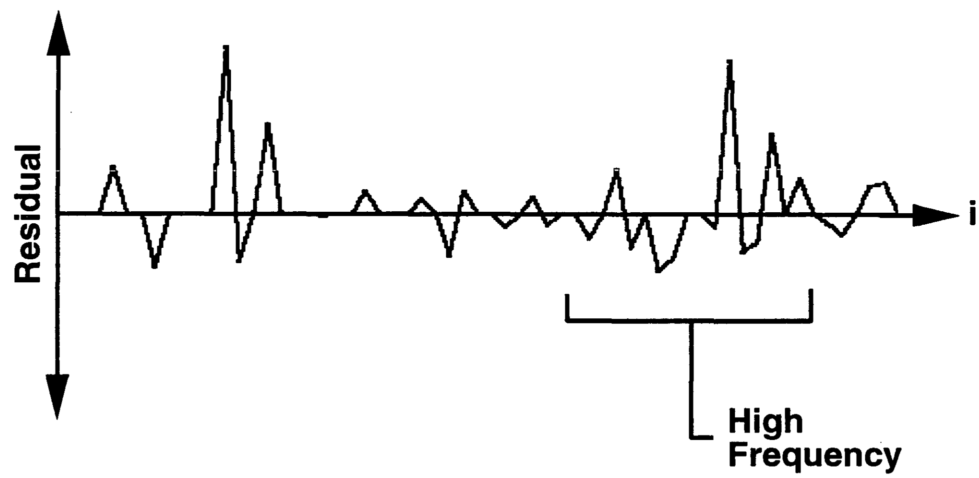

FIG. 3.2. Residual prior to 1 Iteration of Point Gauss-Seidel

frequency error, particularly in the large spikes on the left and in the large region on the right. Next, a single iteration of point Gauss-Seidel is applied to the linear system. Figure 3.3 illustrates the residual vector after the iteration. Most of the residual components have reduced in magnitude. In addition, the high frequency regions have been eliminated, leaving only low frequency error.

Since the stationary solvers are best suited to reducing high frequency error, their convergence rates degrade quickly after a few iterations as most of the high frequency error has been removed. The AMG method accelerates convergence at this point by reducing the low frequency components of the residual using the coarse level correction.

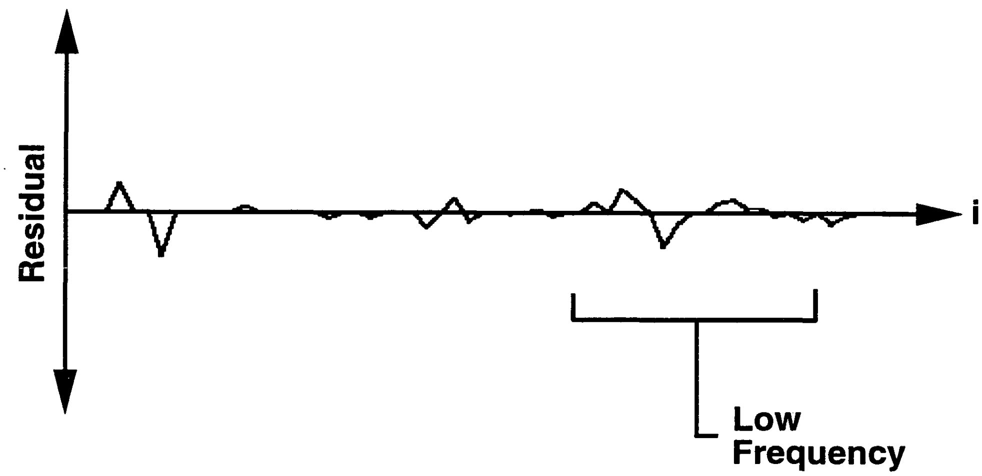

FIG. 3.3. Residual after 1 Iteration of Point Gauss-Seidel

3.1 Basic Algorithm

Given the linear system of the form

$$Au = b\,, \tag{3.1}$$

AMG first constructs the set of coarse levels along with a set of interpolation operators used to transfer solution information from level to level. This process is described in detail in Section 3.2.

The coarse levels are related hierarchically via the relationship described in (3.2). A_1 is the original matrix and corresponds to the finest level, $l = 1$. A_L is the coarsest level at $l = L$. Each coarse level matrix is smaller than the preceding level, with $n_l > n_{l+1}$ where n_l is the size of the matrix at level l.

$$A_{l+1} = P_l^T A_l P_l \tag{3.2}$$

P_l is the interpolation operator derived from A_l during the setup phase. P_l is an

$n_l \times n_{l+1}$ transformation matrix which either restricts or interpolates depending on its application. When applied as P^T, the operator restricts a vector from length n_l to n_{l+1}. Inversely, the operator interpolates an n_{l+1} length vector to an n_l length vector when applied as P.

The primary type of multigrid algorithm used in this research is the V-cycle. The type of cycle depends on how the coarse levels are used to generate the coarse level correction. Other common cycles include W- and F-cycles, shown in Fig. 3.4. Due to the algebraic handling of the boundary conditions in the stiffness matrix, the use of the W- and F-cycles is not recommended as they can cause instabilities within the coarse level correction.

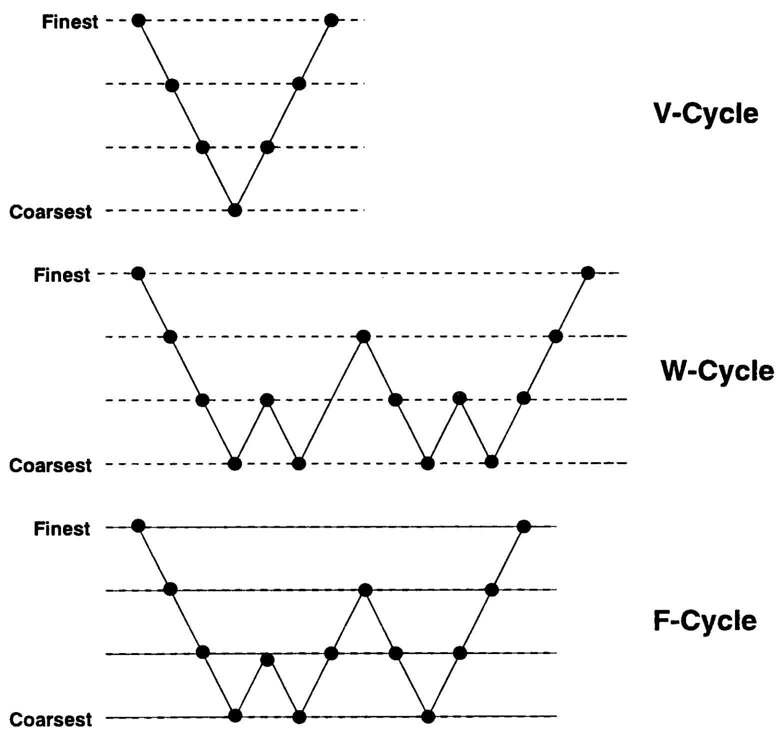

FIG. 3.4. Multigrid Cycles

The AMG V-cycle is incorporated into the operation of the stationary iterative solver. Stationary iterative solvers can be written in the form:

$$u^{(k+1)} = Mu^{(k)} + Nb \, , \tag{3.3}$$

where M and N are derived from the matrix A and have different forms depending on the type of stationary iterative solver being used [27]. The standard algorithm for computing the solution to $Au = b$ using a stationary method of the type described in (3.3) is illustrated below.

```
Algorithm 1:  Stationary Iterative Method

set u to an initial guess
while ||b − Au|| > TOL do
    set u ← Mu + Nb
end
output solution u
```

Computation of the coarse level correction is a recursive algorithm. Algorithm 2 outlines the coarse level correction procedure.

```
Algorithm 2:  Coarse Level Correction

coarse_level_correction(A_l, u_l, b_l)
    apply pre-smoother S_pre(A_l, u_l, b_l)
    set b_{l+1} = P_l^T (b_l − A_l u_l)
    if l + 1 = L then
        solve A_{l+1} u_{l+1} = b_{l+1} using a direct method
    else
        call coarse_level_correction(A_{l+1}, u_{l+1} = 0, b_{l+1})
    end
    set u_l ← u_l + P_l u_{l+1}
    apply post-smoother S_post(A_l, u_l, b_l)
```

```
    end
end
```

The smoother $S(A_l, u_l, b_l)$ generally consists of a combination of iterations by stationary iterative solvers of the form $u \leftarrow Mu + Nb$. Examples of smoothers include Gauss-Seidel iteration, SOR iteration, damped Jacobi iteration, and semi-iterative solvers. The choice of solver used in the smoother depends on the type of problem being solved.

The AMG algorithm can be derived from Algorithm 1 by applying the appropriate hierarchical notation for the coarse levels and the appropriate call to the coarse level correction algorithm. Algorithm 3 outlines the AMG algorithm.

```
Algorithm 3:  AMG

set A_1 ← A and b_1 ← b
derive coarse levels and interpolation operators from A_1
set u_1 to an initial guess
while ||b_1 − A_1 u_1|| > TOL do
    call coarse_level_correction(A_1,u_1,b_1)
end
set u ← u_1
output solution u
```

The `coarse_level_correction` algorithm replaces the application the stationary iterative solver in Algorithm 1.

3.2 Generation of the Coarse Levels

Coarse levels can be derived from the stiffness matrix in several different ways. As mentioned previously, this research uses the smoothed aggregation approach due to its attractive computational complexity [25]. The concept behind this approach is to create a disjoint covering of the set of nodes, define a tentative piecewise constant interpolation operator from the covering, and then smooth the tentative operator to obtain the final interpolation operator.

For small to medium problems, the smoothed aggregation process usually only generates a single coarse level, yielding the following coarse levels and interpolation operators: A_1, P_1, and A_2. The coarsest level is typically $\approx 10^1 \times 10^1$. For larger problems, the coarsening process generates several coarse levels, yielding A_1, P_1, A_2, P_2, ..., A_L and P_{L-1}. Again, the coarsest level is a small matrix, typically $\approx 10^1 \times 10^1$ Thus, the coarse levels are quite small in comparison to the finest level A_1.

3.2.1 Creation of the Disjoint Covering

The purpose behind this step is to take the set of nodes and decompose it into mutually disjoint subsets. These subsets consist of aggregates of nodes which are considered to be strongly coupled. The strength of the connectivity between nodes is determined algebraically from the relative magnitude of the coefficients in the stiffness matrix. Algorithm 4 describes the creation of the disjoint covering of the stiffness matrix A_l at the l coarse level where A_l is an $n_l \times n_l$ matrix. A detailed explanation of the steps follows the presentation of the algorithm below.

`Algorithm 4:  Creation of the Disjoint Covering`

`set` $R = 1, 2, \ldots, n_l$
`set` $j = 0$
`for` $i = 1 \rightarrow n_l$
    `set` $N_i^l(\epsilon) = \{j : \|A_{ij}\| \geq \epsilon\sqrt{\|A_{ii}\|\|A_{jj}\|}\} \cup \{i\}$
`end`
`for` $i = 1 \rightarrow n_l$

```
    if $N_i^l(\epsilon) \subset R$ then
        set $j = j + 1$
        set $C_j^l \leftarrow N_i^l(\epsilon)$
        set $R \leftarrow R \backslash C_j^l$
    end
end
copy $\tilde{C}_k^l = C_k^l,\ k = 1, 2, \ldots, j$
while $\exists i \epsilon R$ and $k$ st $N_i^l(\epsilon) \cap \tilde{C}_k^l \neq \emptyset$ do
    set $C_k^l \leftarrow C_k^l \cup \{i\}$
end
while $\exists i \epsilon R$ do
    set $j = j + 1$
    set $C_j^l = R \cap N_i^l(\epsilon)$
end
set $n_{l+1} = j$
```

The first step is to establish the set of nodes suitable for mapping to the coarse level. This set is labeled R and is initially set to all of the nodes on level l. The next step is to create the tightly coupled neighborhoods using the relationship:

$$N_i^l(\epsilon) = \{j : \|A_{ij}\| \geq \epsilon\sqrt{\|A_{ii}\|\|A_{jj}\|}\} \cup \{i\} \ . \tag{3.4}$$

Since the problem being solved is a vector field problem, the matrix A is divided into blocks

$$A_{kl} = A(df(k), df(l)) \ , \tag{3.5}$$

where $df(k)$ denotes the list of degrees of freedom associated with node k. The connectivity between nodes is gauged by comparing the norm of the off-diagonal blocks with the norms of the corresponding diagonal blocks. The norm $\|\cdot\|$ used in this research was the matrix infinity norm. ϵ is a coarsening parameter that preserves the rate of coarsening on the coarser levels. Since the coarse levels consist of element aggregates, the magnitude of the coefficients on the coarse levels is much greater than on the finest level. The ϵ used in this for this research was computed using the relationship

$$\epsilon = \left(\frac{1}{2}\right)^{l+2} . \tag{3.6}$$

This ϵ was determined via experimentation.

With the tightly coupled neighborhoods now defined, the initial covering C_k^l is selected from the neighborhoods N_l^i. As each C_k^l is defined, the nodes within it are removed from R. Once all the possible neighborhoods have been mapped into the disjoint covering, the next step is to attempt to map the remaining nodes in R to one of the covering sets C_k^l that the node is strongly connected to. This strong connection is defined as the intersection between that node's neighborhood N_i^l and the existing covering sets C_k^l. A nonempty intersection denotes strong coupling and the node is then mapped into that covering set and removed from R.

Any remaining nodes in R after the previous two steps are then combined into aggregates of subsets of the neighborhood N_i^l. Each node left in R defines a covering as the intersection of its neighborhood and R.

This creates n_{l+1} mutually disjoint subsets C_k^l, $k = 1, 2, \ldots, n_{l+1}$, from the set of neighborhoods N_i^l, $i = 1, 2, \ldots, n_l$, containing strongly coupled nodes. These subsets form a disjoint covering of the free nodes on level l. The covering subsets are equivalent geometrically to a set of macro-elements (See Fig. 3.5) consisting of aggregated elements.

3.2.2 Creation of the Interpolation Operator

Using the disjoint covering C_k^l, $k = 1, 2, \ldots, n_{l+1}$, the tentative interpolation operator $\tilde{P}_l$ is defined from the nodes in the covering. In the case of scalar field problems, the identity matrix (I) and zero matrix (0) in (3.7) are replaced with their

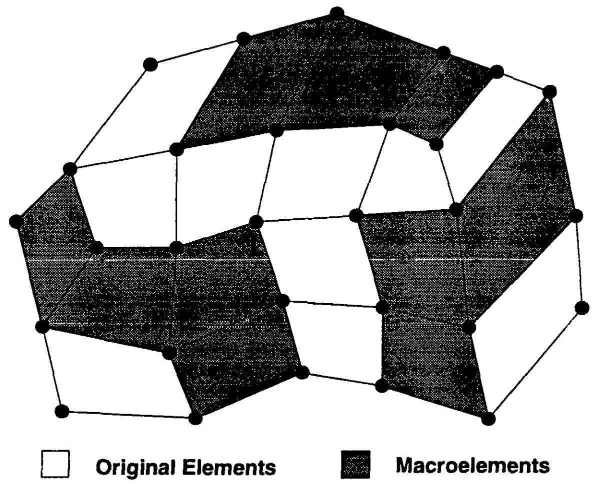

FIG. 3.5. Example of 2D Macro-Elements

scalar equivalents.

$$(\tilde{P}_l)_{ij} = \begin{cases} I & \text{if } i \in C_j^l \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

(3.8) defines the final interpolation operator P_l derived from the Jacobi smoothing of the tentative operator. The parameter ω in (3.8) was assumed to be $\frac{4}{3}$.

$$P_l = (I - \omega D^{-1} A_l)\tilde{P}_l \tag{3.8}$$

The smoothing is applied to the tentative interpolation operator as a method of reducing the energy of the coarse basis functions. Since the effectiveness of this coarsening process relies on the assumption that the energy of the coarse basis functions is small and bounded, this step serves to reinforce that assumption.

Details on the convergence properties of AMG using this type of interpolation operator are described in greater detail in [25] and [26].

3.2.3 Poorly Scaled Meshes

During the testing of the AMG algorithm on ill-conditioned matrices, the coarse level generation algorithm often produced extremely poor levels. During iteration, these levels produced large errors in the coarse level corrections, resulting in very slow convergence or even divergence. Further investigation revealed that the difficulty was arising from the wide range of entry magnitudes within the matrices. The meshes used to generate the matrices for these example problems contained elements with very large aspect ratios (+10). The aspect ratio of the element is the ratio of its dimensions. Regular elements, such as squares and cubes, have aspects ratios of 1 as all of their dimensions are equal. These large aspect ratios manifested themselves by biasing the magnitudes of the stiffness matrix coefficients. Since the coarse level generation depends solely on the relative magnitudes of the coefficients to determine if nodes are tightly coupled, the biased coefficients created an artificial decoupling of the nodes.

The solution to this problem is to apply a diagonal preconditioner to the linear system to eliminate the scaling difficulties prior to the construction of the coarse levels. This modification are illustrated below in Algorithm 5, which represents the final AMG algorithm used in the research.

Algorithm 5: AMG

set $A_1 \leftarrow D^{-1}A$
set $b_1 \leftarrow D^{-1}b$
derive coarse levels and interpolation operators from A_1
set u_1 to an initial guess
while $\|b_1 - A_1u_1\| > TOL$ do
    call coarse_level_correction(A_1,u_1,b_1)
end
set $u \leftarrow u_1$

`output solution` u

3.3 Stationary Multiplicative Iterative Solvers

At the core of the AMG smoothers $S(A_l, u_l, b_l)$ are stationary multiplicative solvers of the form

$$u^{(k+1)} = M_l u^{(k)} + N_l b\ , \tag{3.9}$$

where M_l and N_l are derived from A_l via matrix splitting. Table 3.1 illustrates the structure of the M and N matrices for the Jacobi, Gauss-Seidel, and SOR iterative methods. D is the diagonal of A and $-L$ and $-U$ are the upper and lower triangular parts of A, respectively [28]. The ω parameter in the SOR matrices is the over-

Table 3.1. M and N Matrices for Various Stationary Iterative Solvers

Solver	M Matrix	N Matrix
Jacobi	$D^{-1}(L+U)$	D^{-1}
Gauss-Seidel	$(D-L)^{-1}U$	$(D-L)^{-1}$
SOR	$(D-\omega L)^{-1}(\omega U + (1-\omega)D)$	$\omega(D-\omega L)^{-1}$

relaxation parameter and is used to accelerate convergence.

3.3.1 Smoothing

Within the AMG algorithm, the smoother serves two slightly different roles. The first is at the finest level where the smoother serves two main purposes: reduce the residual and clean up the solution after the addition of the coarse level correction. On the coarser levels, the primary goal of the smoother is to smooth the residual vectors used to compute the coarse grid correction. Selection of the appropriate solvers is dictated by those goals.

The smoother used in this research is an extension of the smoother used by Vaněk, Mandel, and Brezina in their AMG work [25]. For the pre-smoothing, their smoother consisted of one forward iteration of point Gauss-Seidel followed by an iteration of backward point SOR. The post-smoothing consisted of a forward iteration of point SOR followed by an iteration of backward point Gauss-Seidel.

The primary modification was to change the point solvers to block solvers. Since the problems being solved involve vector fields, the freedoms at each node are represented by two or more linear equations. These equations are tightly coupled with large off-diagonal coefficients within the block caused by their derivation from the governing equations. In contrast, the coefficients connecting these blocks are much smaller. In point solvers, each equation is relaxed successively regardless of its coupling with other equations in the block. This can result in large spurious perturbations in the other equations within the block during relaxation. The end result is the introduction of a large amount of error in the solution. Block solvers, on the other hand, relax all of the equations within the block simultaneously. This prevents the spurious perturbations.

The second modification was to modify the number of iterations performed by the smoother at each level. At each level l, the number of iterations for the iterative solvers within the smoother is $L - l + 1$. Thus, the smoother performs more iterations at the finest level and fewer at the coarser level. The reasoning behind this modification was to have the smoother perform the most work at the levels where it would produce the greatest effect. In practice, this assumption generated better convergence and faster runtimes.

The resulting smoother $S(A_l, u_l, b_l)$ is the same for both pre- and post- smoothing and consists of $L-l+1$ iterations of forward block SOR followed by $L-l+1$ iterations of forward block Gauss-Seidel. The value of 1.85 for the over-relaxation parameter ω was determined via experimentation.

Algorithm 6: Smoother $S(A_l, u_l, b_l)$

set $\omega = 1.85$
repeat $L - l + 1$ times
    set $u_l \leftarrow (D_l - \omega L_l)^{-1}(\omega U_l + (1-\omega)D_l)u_l + \omega(D_l - \omega L_l)^{-1}b_l$
end
repeat $L - l + 1$ times
    set $u_l \leftarrow (D_l - L_l)^{-1}U_l u_l + (D_l - L_l)^{-1}b_l$
end

The next chapter details the modifications of the AMG algorithm given in Algorithm 5 to incorporate distributed processing.

Chapter 4

DISTRIBUTED PROCESSING OF AMG

4.1 Processing Paradigm

High performance computing used to be the exclusive domain of supercomputers that exploited vector processing to achieve high computational power. However, these machines suffered from a lack of portable software, particularly in the area of compilers and development tools, as well as extremely large price tags. As a result, applications developed on these computers were not portable to new or different machines and required extensive re-engineering. This along with a large decrease in the sales of supercomputers led to the demise of many supercomputer companies. The current trend is to use existing workstation processors to build parallel computers. This approach leads to a much wider range of available software and requires less new technology to produce. A good example of this type of parallel computer is the SGI Power Challenge which uses existing RISC processors.

In addition, the decline in the use of large supercomputers also led to an increase in the use of distributed computing. A cluster of workstations are connected to create a local area network (LAN) (see Fig. 4.1). Message passing software allows the LAN to function as a loosely-coupled parallel computer with each workstation operating as a distributed memory processor.

Distributed processing is of particular interest to companies with existing LANs who wish to extend the useful life of their current systems. Often, the workstations in the LAN are run as separate computers during the day by engineers and technicians

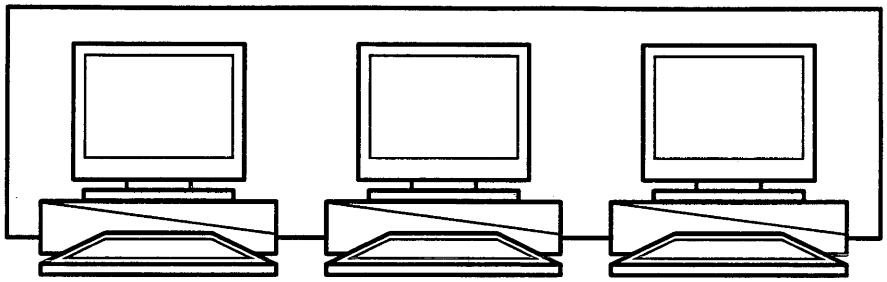

FIG. 4.1. Typical Workstation LAN

working on small individual problems. During evenings and weekends, the LAN is used as a large virtual computer for large computational problems. Economically, the distributed processing paradigm is very attractive. Addition of more processors is as simple as adding more workstations to the network. These workstations are usable for other tasks besides acting as processors for the virtual machine, an ability which greatly adds to their value.

In addition, many companies have purchased multiprocessor workstations as servers which expand the degree of parallelism within the LAN (see Fig. 4.2). These servers improve the efficiency of the network by reducing the need for individual workstations to handle most of networking tasks.

Distributed processing can be applied in two different ways. First, distributed processing can be used to run very large problems through the sharing of system resources, such as core memory and disk space. This approach usually does not offer significant gains in performance as no parallelism is exploited. However, this type of application does allow for the solution of problems too large to fit onto a single system. Second, a problem can use distributed processing to achieve better runtimes through the exploitation of parallelism within the algorithm. Due to the high communication costs associated with distributed processing, this use is less common than the first.

This research is looking primarily at the distributed processing paradigm based

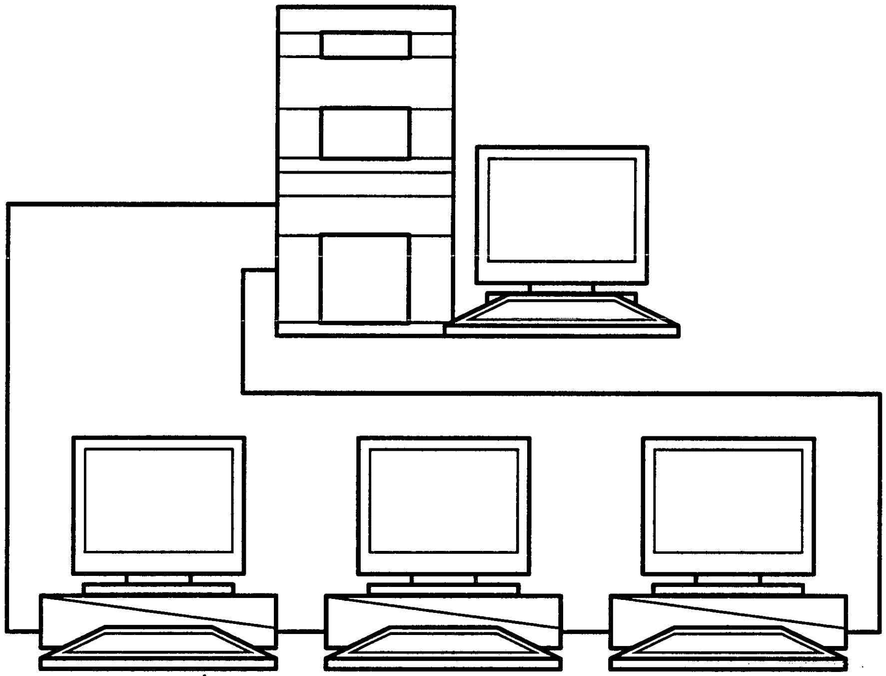

FIG. 4.2. Workstation LAN with Multiprocessor Server

on information regarding the type of systems used by engineers in the geomechanics industry. This information was obtained from the Maptek system sales over the past several years. These data are primarily concerned with practical applications in the field of mining engineering and represent the typical available hardware at a mine site. In addition, future projections based on client requests include the use of multiprocessor servers at some sites, which results in configurations similar to that shown in Fig. 4.2. However, this research is examining only distributed computing on networks of single processor workstations. Table 4.1 summarizes the hardware assumed for this research. The primary performance limitation, as mentioned in Chapter 1, is the limited memory available on a single workstation which causes the workstation to perform large amounts of disk swapping for problems too large to fit into the core memory. Hence, the goal of distributed processing for this research is not to decrease the runtime by reducing the amount of operations done sequentially, but to decrease the runtime by eliminating the disk swapping.

Table 4.1. Assumed Hardware Configuration of the LAN

Workstation LAN
3-5 Workstations
64 Mb RAM (per workstation)
2 Gb Hard Disk (per workstation)
10 Mbit/s Ethernet Network

Due to the large amount of processing power inherent to each processor in a distributed environment, the distributed computing paradigm is extremely flexible in the types of problems it can handle. The primary drawback to the use of distributed computing is the slow communication time between workstations. In a real parallel computer, the processors are connected via high-speed dedicated links. On a distributed computer, however, the processors are linked via a network connection, such as Ethernet. The data transmission rate is much slower for this type of connection. Also, message passing between workstations often has to compete with normal network traffic, such as Email, ftp telnet, X Windows, network file system (NFS), etc. However, developments in networking technology are improving the bandwidth available to LANs. Ethernet technology is nearly a decade old and its bandwidth of 10 Mbit/s is quite low as compared with new technology, such as FDDI, HiPPI, and ATM [29]. The use of fiber optics and parallel cables greatly increases the throughput and results in bandwidths from 100 Mbit/s to 1 Gb-it/s [29]. The end result of this technology is much faster message passing between distributed processors, which reduces execution delays caused by message waits.

4.2 Data Mapping

The goal of distributed computing is to arrange the data structures necessary for the AMG algorithm within the core memory of a set of workstations. A variety of different mapping configurations are possible.

For the AMG algorithm outlined in Chapter 4, the largest data structures are those used to store the coarse level matrices A_l, $l = 1, \ldots, L$ and the interpolation operator matrices P_l, $l = 1, \ldots, L-1$. Fortunately, $A_1 >> A_2 >> \ldots >> A_L$ which implies that the storage requirements for the coarse levels are much less than the storage requirements for the finest level. A trivial mapping of the problem to the workstations is to map one level for each workstation and obtain the storage of matrices as illustrated in Fig. 4.3. Message passing between the workstations is then

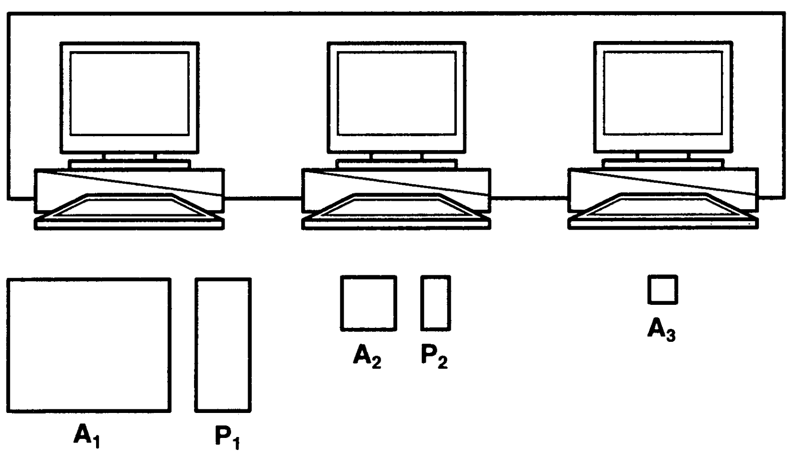

FIG. 4.3. Tentative Distributed Levels Data Mapping

limited to the restricted residual b_{l+1} and the coarse level correction u_{l+1}.

A limitation of this mapping arises from the storage of A_1. If the storage requirements of A_1 and P_1 exceed the core memory available on a single workstation. (a likely occurrence for large problems), disk swapping will occur on that machine.

This results in a computational bottleneck and little increase in the performance. An alternative is to move the interpolation operators P_l over one workstation. This mapping is illustrated in Fig. 4.4 and can prevent the occurrence of swapping depending on the size of A_1. For this reason, this data mapping was selected for use in this research. In addition, the number of available workstations might be less than the

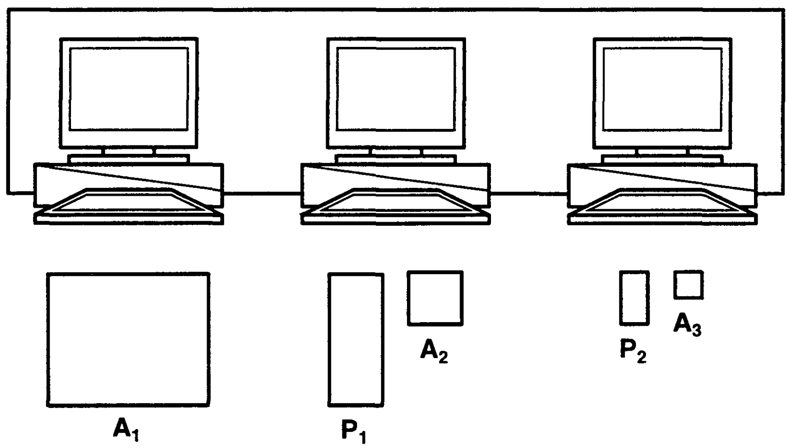

FIG. 4.4. Final Distributed Levels Data Mapping

total number of levels. In this case, the only solution is to lump several of the smaller levels into a single workstation. This yields the data mapping in Figure 4.5.

The primary advantage of these mappings is their simplicity. With each level contained on its own workstation, communication is limited to sending and receiving vectors between each level, which is always less than or equal to the number of workstations ($N \leq L$). The disadvantage of this type of data mapping is its limited scalability. If the problem size grows and the number of levels remains constant, the addition of more workstations will not increase the amount of effective system resources. Once A_1 exceeds the core memory on its host workstation, this data mapping will be unable to prevent disk swapping.

For problems with extremely large A_1 matrices, another option for in-core storage

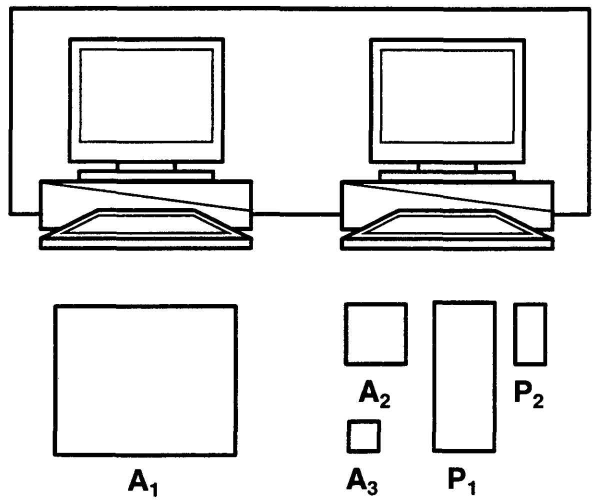

FIG. 4.5. Distributed Levels Data Mapping for Limited Workstations

is to split the matrix into small chunks and distribute the chunks to other workstations. The size of the chunks is defined primarily by the size of the core memories on the workstations onto which the chunks are being mapped. For example, a 180 Mb matrix can be split into 3 chunks of 60 Mb for mapping onto three workstations with 64 Mb each. Fig. 4.6 illustrates the resulting mapping accomplished by splitting A_1 into smaller chunks. In addition to the previous mapping, this mapping was also selected for use in the research.

Unlike the distributed levels data mapping, this mapping is much more scalable and is not limited by the number of levels. Additional workstations can be added to reduce the memory requirements on each workstation to the point where the entire problem can be stored within core memory.

The disadvantage of this mapping is the extra communication cost. Since the A_1 matrix is now distributed over several different workstations, any operations involving the A_1 matrix will involve several inter-workstation communications. In addition, the splitting of the matrix should be done in such a way as to facilitate whatever

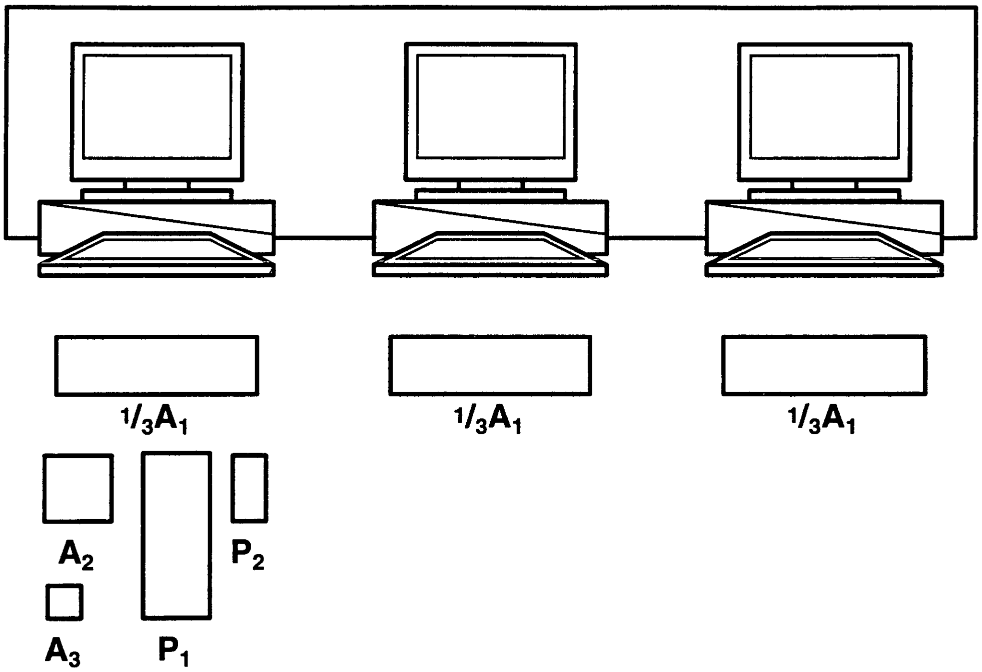

FIG. 4.6. Distributed Data Mapping for Very Large A_1 Matrices

operations are going to be performed during the algorithm. Examination of the AMG algorithm reveals that the two main operations are matrix-vector multiplication and back-substitution. Since these operations are primarily row-oriented, the optimum splitting of the matrix is into chunks of rows. In this research, the number of rows in the matrix was divided equally among the N workstations, such that each workstation receives n/N rows of a $n \times n$ matrix.

4.3 Algorithm Structure

Depending on the data structure selected, the AMG algorithm must be modified based on how the distributed data structures are accessed. The amount of modification is dependent on the distributed data mapping selected for use with the AMG algorithm.

4.3.1 Distributed Levels Algorithm

Using the data mapping shown in Fig. 4.4, the modified AMG algorithm is illustrated in Table 4.2.

Table 4.2. AMG on a Distributed System

Workstation 1	Workstation 2	Workstation 3
apply $S(A_1, u_1, b_1)$ set $r_1 = b_1 - A_1 u_1$ send r_1 to Workstation 2	IDLE	
	receive r_1 $b_2 = P_1^T r_1$ $u_2 = 0$ apply $S(A_2, u_2, b_2)$ $r_2 = b_2 - A_2 u_2$ send r_2 to Workstation 3	IDLE
IDLE	IDLE	receive r_2 $b_3 = P_2^T r_2$ solve $A_3 u_3 = b_3$ $\hat{u}_2 = P_2 u_3$ send $\hat{u}_2$ to Workstation 2
	receive $\hat{u}_2$ $u_2 \leftarrow u_2 + \hat{u}_2$ apply $S(A_2, u_2, b_2)$ $\hat{u}_1 = P_1 u_2$ send $\hat{u}_1$ to Workstation 1	IDLE
receive $\hat{u}_1$ $u_1 \leftarrow u_1 + \hat{u}_1$ apply $S(A_1, u_1, b_1)$	IDLE	

In this particular version of the distributed AMG algorithm, the only modification of the AMG algorithm is to transfer the processing of each level from workstation to workstation. No operations are performed in parallel and the only gains in performance are those arising from the avoidance of swap delays. During the computation of the coarse level correction, most of the workstations are idle. In examining the above algorithm, one notices the opportunity to perform a smoothing operation in parallel during the computation of the coarse level correction. Table 4.3 illustrates the modified distributed algorithm. This would utilize the workstations on the finer levels

Table 4.3. AMG on a Distributed System with Tentative Parallel Smoothing Step

Workstation 1	Workstation 2	Workstation 3
apply $S(A_1, u_1, b_1)$ $r_1 = b_1 - A_1 u_1$ send r_1 to Workstation 2	IDLE	
	receive r_1 $b_2 = P_1^T r_1$ set $u_2 = 0$ apply $S(A_2, u_2, b_2)$ set $r_2 = b_2 - A_2 u_2$ send r_2 to Workstation 3	IDLE
apply $S(A_1, u_1, b_1)$	apply $S(A_2, u_2, b_2)$	receive r_2 $b_3 = P_2^T r_2$ solve $A_3 u_3 = b_3$ $\hat{u}_2 = P_2 u_3$ send $\hat{u}_2$ to Workstation 2
	receive $\hat{u}_2$ $u_2 \leftarrow u_2 + \hat{u}_2$ apply $S(A_2, u_2, b_2)$ $\hat{u}_1 = P_1 u_2$ send $\hat{u}_1$ to Workstation 1	IDLE
receive $\hat{u}_1$ $u_1 \leftarrow u_1 + \hat{u}_1$ apply $S(A_1, u_1, b_1)$	IDLE	

during the computation of the coarse level correction and apply additional sweeps of the smoother.

However, a major difficulty arises during the application of the coarse level correction $\hat{u}_l$ to the solution u_l after the additional sweep of the smoother S. The coarse level correction applies to a very specific set of vector components and application of the coarse level correction to the solution u_l after modification in the smoother can result in destructive interference which results in little to no acceleration. In some cases, this interference can significantly slow the algorithm.

To avoid this interference, both the solution u_l and the coarse level correction

$\hat{u}_l$ must undergo the same matrix operations prior to their addition. Due to the multiplicative nature of the stationary solvers used within the smoother S, the application of an additional smoother sweep in parallel with the computation of the coarse level correction is virtually impossible. However, the possibility does exist for the application of a single iteration of a stationary iterative solver. Knowing that

$$u_l^{(k+1)} = M_l u_l^{(k)} + N_l b_l \tag{4.1}$$

and

$$u_l^{(k)} \leftarrow u_l^{(k)} + \hat{u}_l \ , \tag{4.2}$$

the order in which the operations are performed can be modified such that

$$\begin{aligned} u_l^{(k+1)} &= M_l(u_l^{(k)} + \hat{u}_l) + N_l b_l \\ &= M_l u_l^{(k)} + M_l \hat{u}_l + N_l b_l \ . \end{aligned} \tag{4.3}$$

In this form, the application of the M_l matrix is performed on the solution and coarse level correction prior to summation. Using this alternate order of operations, the AMG algorithm can be modified so that a single iteration of a stationary iterative solver is applied in parallel to the solution u_l during the computation of the coarse level correction. Then, the coarse level correction is multiplied by M_l prior to its addition to the solution u_l. Since the operations performed on u_l and $\hat{u}_l$ are consistent, no interference occurs and the integrity of the solution is maintained.

Using this information, the AMG algorithm in Table 4.3 is modified so the application of the smoother $S(A_l, u_l, b_l)$ is replaced with a single iteration of block SOR. In addition, $\hat{u}_l$ is multiplied by M_l prior to its addition to the solution u_l where M_l is the M matrix for block SOR. The resulting algorithm is described in Table 4.4.

Table 4.4. AMG on a Distributed System with Parallel Smoothing Step

Workstation 1	Workstation 2	Workstation 3
apply $S(A_1, u_1, b_1)$ $r_1 = b_1 - A_1 u_1$ send r_1 to Workstation 2	IDLE	
	receive r_1 $b_2 = P_1^T r_1$ $u_2 = 0$ apply $S(A_2, u_2, b_2)$ $r_2 = b_2 - A_2 u_2$ send r_2 to Workstation 3	IDLE
$u_1 \leftarrow M_1 u_1 + N_1 b_1$	$u_2 \leftarrow M_2 u_2 + N_2 b_2$	receive r_2 $b_3 = P_2^T r_2$ solve $A_3 u_3 = b_3$ $\hat{u}_2 = P_2 u_3$ send $\hat{u}_2$ to Workstation 2
	receive $\hat{u}_2$ $u_2 \leftarrow u_2 + M_2 \hat{u}_2$ apply $S(A_2, u_2, b_2)$ $\hat{u}_1 = P_1 u_2$ send $\hat{u}_1$ to Workstation 1	IDLE
receive $\hat{u}_1$ $u_1 \leftarrow u_1 + M_1 \hat{u}_1$ apply $S(A_1, u_1, b_1)$	IDLE	

A wide variety of methods have been researched in an effort to perform the smoothing and the computation of the coarse level correction in parallel [30],[31],[32],[33],[34],[35]. However, the AMG algorithm is inherently sequential when used with a multiplicative smoother such as Gauss-Seidel or SOR. While these parallel modifications were generally successful, the convergence rate often dropped significantly, resulting in little or no improvement in the algorithm runtime from the parallelism alone. In the AMG algorithm above, however, the only modification to the operations is a change in their order of application. As a result, this algorithm is able to exploit a limited amount of parallelism without sacrificing its rate of convergence. In fact, this algorithm should have slightly better convergence in comparison with its non-parallel relative in that this version applies an additional iteration of block SOR. The corresponding modification to the non-parallel version to match would be an increase in the number of SOR iterations in the smoother $S(A_l, u_l, b_l)$ during the post-smoothing. Instead of $L - l + 1$ iterations of block SOR, the smoother would have to perform $L - l + 2$ iterations.

4.3.2 Distributed A_1 Algorithm

Using the second data mapping illustrated in Fig. 4.6, the AMG algorithm must be modified in order to access the distributed A_1 matrix.

Table 4.5. AMG using a Distributed A_1 Matrix

Workstation 1	Workstation 2	Workstation 3
distributed $S(A_1, u_1, b_1)$	distributed $S(A_1, u_1, b_1)$	distributed $S(A_1, u_1, b_1)$
distributed compute $A_1 u_1$	distributed compute $A_1 u_1$	distributed compute $A_1 u_1$
$r_1 = b_1 - A_1 u_1$		
$b_2 = P_1^T r_1$		
$u_2 = 0$		
apply $S(A_2, u_2, b_2)$		
$r_2 = b_2 - A_2 u_2$	IDLE	IDLE
$b_3 = P_2^T r_2$		
solve $A_3 u_3 = b_3$		
$u_2 \leftarrow u_2 + P_2 u_3$		
apply $S(A_2, u_2, b_2)$		
$u_1 \leftarrow u_1 + P_1 u_2$		
distributed $S(A_1, u_1, b_1)$	distributed $S(A_1, u_1, b_1)$	distributed $S(A_1, u_1, b_1)$

In this case, the only operations requiring the accessing of the distributed A_1 matrix are the smoother $S(A_1, u_1, b_1)$ and the computation of $A_1 u_1$. Examination of the matrix-vector operations for these steps yields the following:

$$y = (\omega U_1 + (1 - \omega) D_1) u \tag{4.4}$$

$$(D_1 - \omega L_1) u = y \tag{4.5}$$

$$z = A_1 u \,. \tag{4.6}$$

Of the three operations, the first and last are matrix-vector multiplications and the middle a lower-triangular back-substitution.

Distributed Matrix-Vector Multiplication

Distributed matrix-vector multiplication is the simplest of the distributed matrix-vector operations. Consider the product $y = Au$ where A is an $n \times n$ matrix. The matrix A is distributed to N workstations where each workstation contains n/N rows. Due to the type and degree of parallelism inherent in this operation, the distributed matrix-vector multiplication fits well into the master-slave paradigm. The master workstation, Workstation #1, which contains the first n/N rows, takes the u vector and sends it to the rest of the workstations. Each workstation then multiplies the u vector by its portion of the A matrix. These pieces are then returned to Workstation #1 which sums the returning products to form Au. Figure 4.7 illustrates this distributed operation.

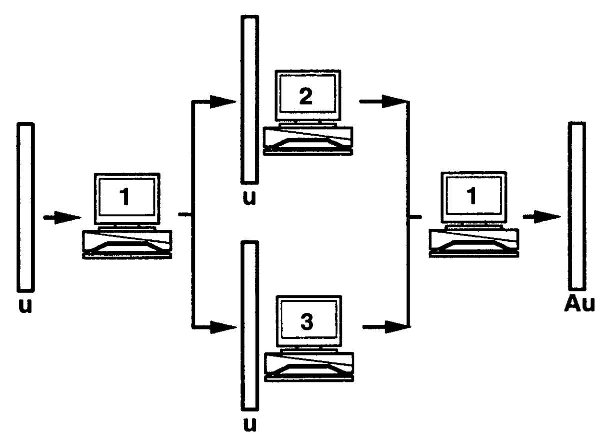

FIG. 4.7. Distributed Matrix-Vector Multiplication

This operation requires $2(N-1)$ inter-workstation communications due to the need to distribute the u vector and return the Au products. However, the high communication cost for this particular operation is offset by the parallel computation of the Au products.

Distributed Forward/Backward Substitution

Forward and backward substitution on a distributed system can be approached in one of two ways. In the first method, variables are solved by substituting the previously calculated variables into the current row equation and solving for the only unknown. This method is easily modified to use distributed processing as the algorithm switches workstations as as all the rows on a particular workstation are solved. This method is very attractive as it requires very little inter-workstation communication.

The second method approaches back substitution is a slightly different manner. As a variable is solved for, its value is multiplied by the appropriate matrix coefficients and subtracted from the RHS. This version of the back-substition algorithm is very popular in parallel computing as the RHS updates can be performed in parallel. The disadvantage of this particular algorithm is the high communication cost associated with distributing the values of the solved variables to every workstation. For a matrix of rank n, the second method requires $N(n^2-n)/2$ inter-workstation communications. In addition, each communication only consists of a single double precision variable. Small messages incur a large overhead relative to their size and therefore are not very efficient. Combined with the high number of communications necessary in this algorithm, the overall efficiency is extremely low and not practical on a 10 Mbit/s workstation LAN.

The first method, however, requires only N communications. Once Workstation #1 is finished with its portion of the matrix, it sends the b vector and the calculated portion of the u vector to Workstation #2. At Workstation #N, the computation of the u vector is completed and the u vector is returned to Workstation #1. Figure 4.8 illustrates this operation. This version of the back-substitution algorithm is much

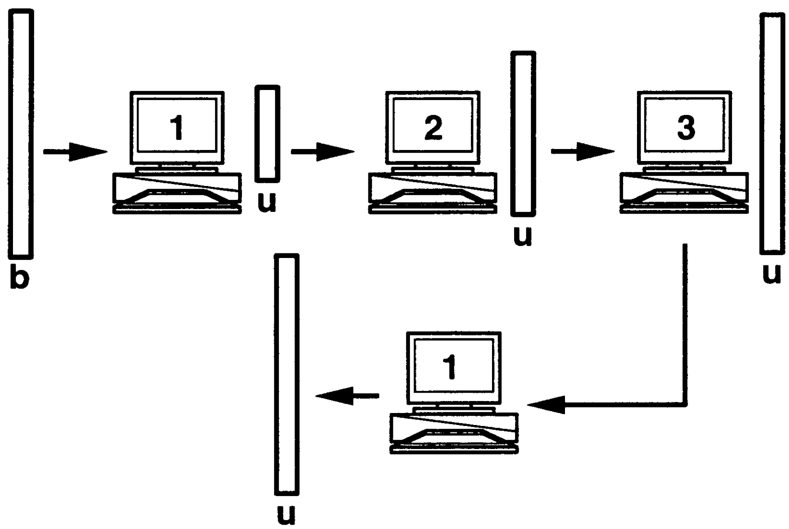

FIG. 4.8. Distributed Back Substitution

more efficient than the parallel version in that its requires far less communication and each communication transfers a large amount of data relative to the message passing overhead. As a result, this version of the back-substitution algorithm was selected for use in this research.

4.4 Comparison of the Distributed AMG Algorithms

In the analysis of parallel and distributed algorithms, performance is measured through the computation of the algorithm's speedup. Speedup is defined as the runtime of the sequential algorithm, $T(1)$, over the runtime of the parallel/distributed algorithm on N processors, $T(N)$:

$$S = \frac{T(1)}{T(N)} . \tag{4.7}$$

For parallel or distributed processing to be effective, the speedup should be greater than 1.0. In most practical applications, the speedup should be much larger than one in order to justify the increased cost of parallel/distributed hardware.

Distributed algorithms can experience speedup due to the exploitation of parallelism in the algorithm, although in most cases the massive communication overhead overwhelms the parallel performance gains. In the case of the distributed AMG algorithms, the amount of parallelism is quite limited, precluding the possibility of large performance gains via parallelism.

A variety of theoretical models exists for predicting the performance of parallel/distributed algorithms on various types of parallel architectures. These models allow designers and programmers to compare the performance of parallel algorithms before they implement the algorithms on the target architecture. Nearly all theoretical models center on two basic aspects of a parallel/distributed algorithm: computation and communication. While these models are often quite sophisticated and are extremely useful for predicting performance, they do have limitations.

First, most theoretical models make assumptions that often do not accurately reflect reality. For example, a common assumption is that each processor has unlimited memory. The drawback of this assumption is that model is unable to predict performance degradation due to swapping. While these assumptions are essential in reducing the complexity of the model, they can lead to inaccurate results.

Second, these models are not well suited to modeling performance on workstation LANs. A primary assumption of most theoretical models is that inter-processor communication is strictly limited to that generated by the parallel algorithm. However, On a workstation LAN inter-workstation communication must compete with network traffic not related to the algorithm being run. Performance degradation created by such interference is not predicted in most theoretical models.

These limitations hamper the application of these models to distributed algorithms running on workstation LANs. For the distributed AMG algorithms, the real performance gain arose from the use of distributed processing to avoid computational performance degradation due to swapping. As this degradation is a function of the physical memory of each workstation, most theoretical models will not predict improved performance through the use of distributed processing. Consequently, the predicted performance will be worse than the sequential algorithm performance due to the communication required by the distributed algorithms.

A very simple model is used below to predict the performance of the distributed AMG algorithms. This model assumes that the amount of time required to complete the distributed AMG algorithm is equal to the sequential runtime plus the total time required for inter-workstation communications. Obviously this model will not predict speedups greater than 1.0. However, this model will allow comparison of distributed performance between the distributed levels and distributed A_1 algorithm. In addition, the LAN architecture is assumed to be Ethernet and that no communication is performed in parallel.

Assume that a single iteration of the non-distributed AMG algorithm takes some unit of time t and the number of iterations until convergence is m. The total runtime for the algorithm is mt. In the distributed levels AMG algorithm, the cost of the inter-workstation communication must be added to the runtime. In this case, the number of workstations (N) dictates the number of inter-workstation communications as some levels may be lumped together on a single workstation. In the case of lumping, the communication time is very fast as the communication is handled internally between processes within the workstation and not across the LAN. Assume the time for a single inter-workstation communication of a vector of length n is t_0. During the execution, $N-1$ inter-workstation communications are performed as the residual r_l descends through the levels and $N-1$ communications as the coarse level correction $\hat{u}_l$ ascends. The total runtime is

$$\text{Runtime} = m(t + 2(N-1)t_0)\ . \tag{4.8}$$

The speedup for this distributed algorithm would be

$$\begin{aligned} S &= \frac{mt}{m(t+2(N-1)t_0)} \\ &= \frac{t}{t+2(N-1)t_0}\ . \end{aligned} \tag{4.9}$$

(4.9) illustrates the detrimental effect of distributed processing on the speedup. Since the denominator is always larger than the numerator, the speedup S for this algorithm will always be less than 1.0 for $N > 1$. Since the number of workstations N is bounded by the number of levels, the most significant penalty to the performance of this algorithm is the time required for an inter-workstation communication (t_0). In a heavily loaded network environment such as a university or large corporation, the time t_0 can be quite significant if the vector length n is large.

Analysis of the distributed A_1 algorithm follows along the same lines as the distributed levels algorithm above. In this case, the communication costs are much higher due to the particular data mapping used in this algorithm. The communications required by applying the distributed smoother and computing $A_1 u_1$ are summarized below. The runtime for the distributed levels AMG algorithms becomes

Table 4.6. Inter-workstation Communications for Distributed Levels Algorithm

Operation	Number of Operations	Communication Cost
Pre-Smoother $S(A_1, u_1, b_1)$	L	$2(N-1)+N$
Compute $A_1 u_1$	1	$2(N-1)$
Post-Smoother $S(A_1, u_1, b_1)$	L	$2(N-1)+N$
	Total Cost	$2[(2L+1)(N-1)+LN]$

$$\text{Runtime} = m(t + 2[(2L+1)(N-1) + LN]t_0)\ . \tag{4.10}$$

In this case, the communication costs are much greater than those in the distributed levels algorithm and the corresponding speedup for this algorithm will be less:

$$\begin{aligned} S &= \frac{mt}{m(t + 2[(2L+1)(N-1) + LN]t_0)} \\ &= \frac{t}{t + 2[(2L+1)(N-1) + LN]t_0)}\ . \end{aligned} \tag{4.11}$$

This speedup estimate does not take into account the effect of parallel processing in the matrix-vector multiplication computation and, accordingly, this estimate represents the lower bound on the speedup produced through the use of this distributed algorithm. The degree to which the parallelism improves the speedup is dependent on the size of the matrix A_1. A very large matrix will benefit more than a small A_1 matrix as the communication costs become comparable to the processing times.

For this particular algorithm, the number of levels L can be considered to be fixed. The number of workstations N is free to increase up to $N = n$. An optimal value of N occurs at the point where the parallelism in the distributed operations offsets the inter-workstation communication by the greatest amount. This value of N is dependent on the size of the matrix n and the amount of available memory in each workstation.

The graphs below illustrate the sensitivity of the distributed AMG algorithms with respect to the inter-workstation communication time t_0. The graphs are normalized so the maximum speedup is assumed to be 1. For each graph, the communication time t_0 is assumed to be proportional to the processing time t. The number of levels L is assumed to be 10 and each workstation is assumed to have unlimited memory.

In the first graph, the inter-workstation communication time is very small in comparison to the processing time t, such that $t_0 = \frac{1}{10{,}000}t$. This example simulates running the algorithm on a very fast LAN such that the communication costs are negligible. In this example, the effect of using a large number of workstations for the distributed processing is minimal for both algorithms. In the next graph, t_0 was increased by a factor of 10. This simulates a LAN where inter-workstation communications are large enough to be significant. From the graph, it is obvious that the increased communication cost has little affect on the distributed levels algorithm. However, the distributed A_1 algorithm shows a marked decrease in performance. Considering the amount of communication inherent to that algorithm, this result is not unexpected. In the last graph, the communication cost t_0 is increased again by a factor of 10. This example simulates running the algorithm on a slow LAN. At this point, the distributed levels algorithm begins to exhibit performance degradation as the number of processors increases. The distributed A_1 algorithm shows a severe drop in performance even when being run on a small number of workstations. Obviously the distributed A_1 AMG algorithm is the most sensitive to the changes in t_0. When

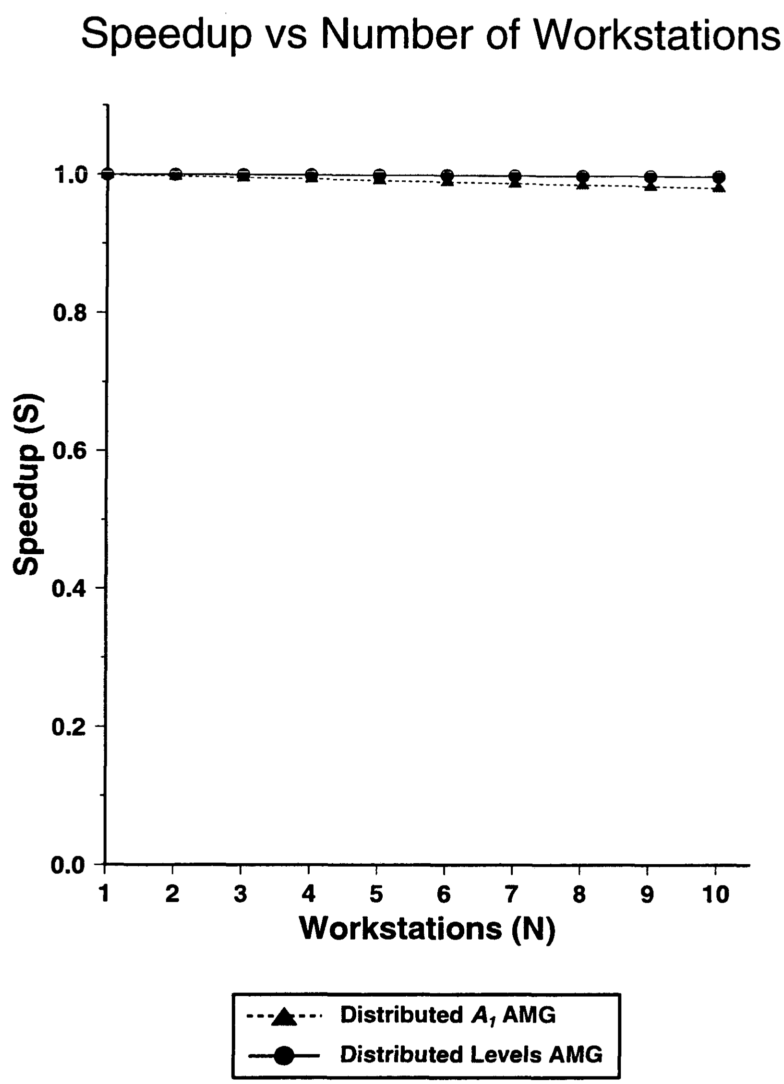

FIG. 4.9. Theoretical Speedup Curves for $t_0 = \frac{1}{10{,}000}t$

using a large number of workstations on a network with heavy traffic, the speedup gained by avoiding swap delays is minimal at best for this algorithm.

In comparison, the distributed levels AMG algorithm appears to offer the better performance than the distributed A_1 algorithm. However, the distributed levels

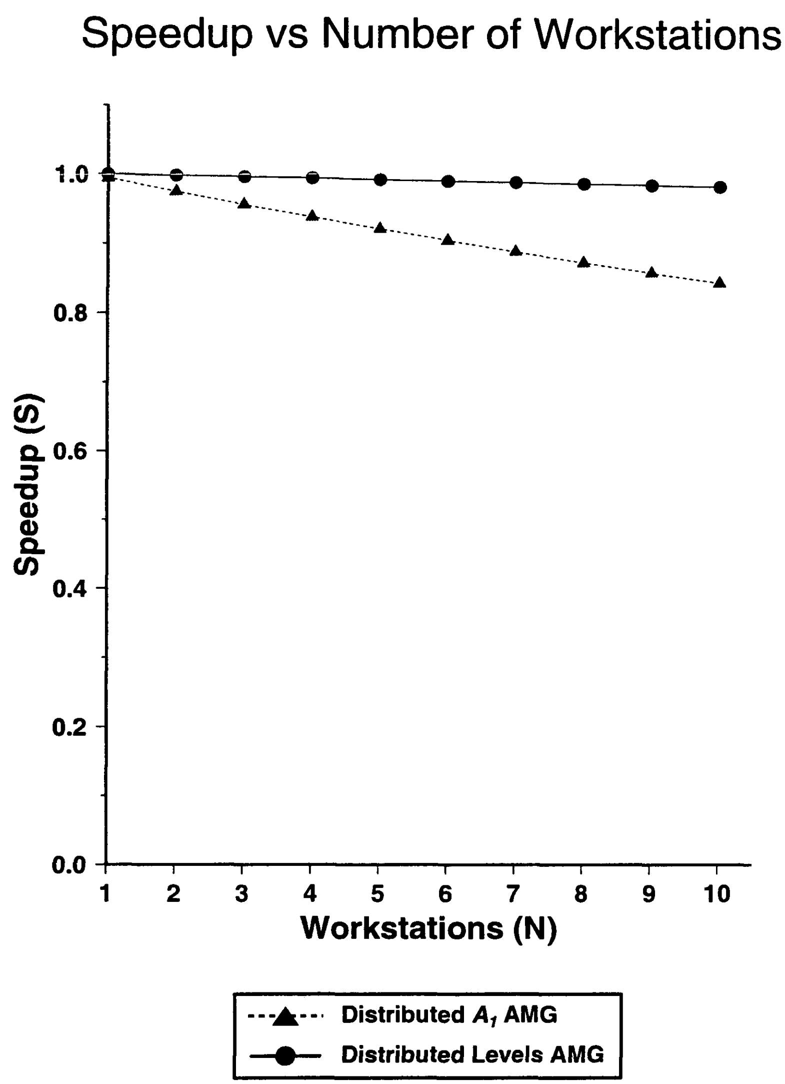

FIG. 4.10. Theoretical Speedup Curves for $t_0 = \frac{1}{1{,}000}t$

algorithm is much less flexible due to its more restrictive mapping. Accordingly, the distributed levels AMG algorithm is less general than the distributed A_1 AMG algorithm. In those cases where the available hardware is suited to the distributed levels algorithm, the best choice for performance is the distributed levels algorithm.

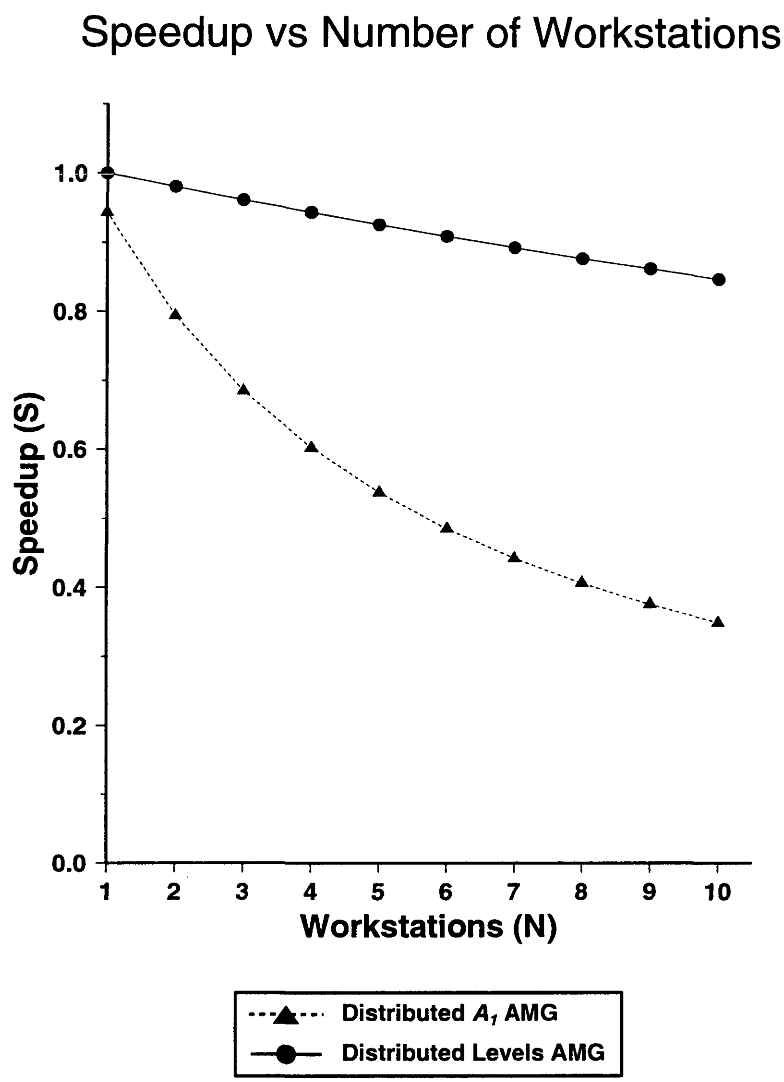

FIG. 4.11. Theoretical Speedup Curves for $t_0 = \frac{1}{100}t$

From the predicted theoretical performance, the use of distributed processing in the AMG algorithms should be avoided as it yields worse performance than the sequential algorithm. In all cases, the communication costs drove the speedups below one, particularly in those cases where the inter-workstation cost t_0 approached the processing time t. In the next chapter, however, the actual speedups for the distributed algorithms show a very large increase in performance which is not predicted in the model. This gain arises from fitting the entire problem into phyiscal RAM and avoiding any swap delays.

Chapter 5

NUMERICAL RESULTS

Several finite element analysis problems were taken from [10] for use as test problems for the AMG algorithms. Due to the equivalence in ill-conditioning between poorly-scaled meshes and the modeling of random materials, these examples use poorly scaled meshes to produce the ill-conditioned stiffness matrices. This approach was selected as it required the least amount of modifications to the existing finite element algorithms in [10].

The example programs are all written in FORTRAN 77 and modified to call C subroutines which handle the sparse matrix data structures and AMG algorithms. For example, the FORTRAN code for the 2D foundation analysis problem contains the following lines of code which assembles the global stiffness matrix from the element stiffness matrices. In the notation of [10], the stiffness matrix A is stored in the 1D array KB.

```
   CALL MSMULT(BTDB,IBTDB,QUOT,IDOF,IDOF)
20 CALL MATADD(KM,IKM,BTDB,IBTDB,IDOF,IDOF)
10 CALL FORMKB(KB,IKB1,KM,IKM,G,IW,IDOF)
```

The corresponding call to the C subroutine for assembling the global stiffness matrix is `csparv` which replaces `FORMKB`.

```
   CALL MSMULT(BTDB,IBTDB,QUOT,IDOF,IDOF)
20 CALL MATADD(KM,IKM,BTDB,IBTDB,IDOF,IDOF)
10 call csparv(KM,IKM,G,IDOF)
```

Similar modifications replace the Cholesky factorization and back-substitution calls with calls to the AMG and preconditioned conjugate gradient (PCG) solver subroutines.

Since [10]'s finite element code removes constrained freedoms from the global stiffness matrix, the corresponding number of local DOF no longer remains constant. Since the generation of coarse levels in the AMG algorithm depends on a constant local DOF, constrained freedoms are treated as fixed nodes with a displacement of 0.

5.1 Distributed Processing Setup

The numerical results were run on a network of SGI Indy R5000 workstations. Each workstation had 64 Mb of physical RAM and a 1 Gb hard disk. The workstations were networked using 10 Mbit/s Ethernet. The message passing software used to implement the distributed processing communication was Parallel Virtual Machine (PVM) 3.3. Mapping of the matrices to the workstations was accomplished by saving each matrix to disk after its construction and then using NFS to cross-mount the disks. The matrix was then loaded onto the appropriate workstation. This method was found to be much faster than transmission of the matrix data via the Ethernet connections.

5.2 2D Foundation Analysis

This example problem is concerned with the failure analysis of a flexible strip footing resting on a uniform layer of undrained clay (see Figure 5.1). The footing is assumed to bear a uniform stress which is increased incrementally until failure. The failure criteria used in this particular example is the Von Mises failure criterion The FORTRAN code used for this example is Program 6.0 from [10].

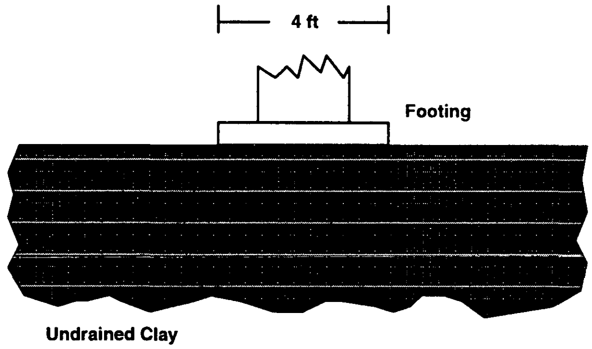

FIG. 5.1. 2D Foundation Analysis Problem

The footing in this case is also assumed to be flexible, which means the footing is free to deform under loading (see Figure 5.2). In contrast, a rigid footing is assumed

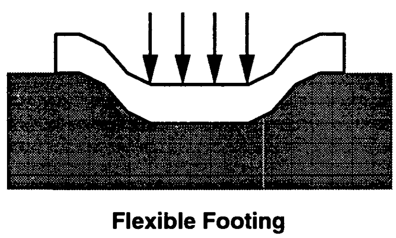

FIG. 5.2. Flexible Footing under Loading

to displace as a single unit. This example uses the flexible footing as it does not require any modification of the stiffness matrix or algorithm.

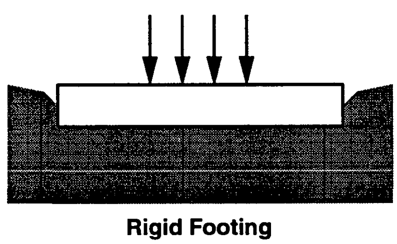

FIG. 5.3. Rigid Footing under Loading

The goal of this example was to compare the performance of the AMG algorithm with the PCG algorithm using the diagonal preconditioner. Two meshes for the same problem domain were created, one with uniform elements and one with elements of varying thickness. From the analysis presented in Chapter 2, the global stiffness matrix generated from the non-uniform mesh should be much more ill-conditioned compared to the stiffness matrix from the uniform mesh.

Using the problem domain in [10], the problem was meshed using 960 8-node quadrilateral elements (48 $\times$ 20). In the first mesh, the elements were all uniform squares while in the second mesh, the thickness of the elements varied with depth. Elements at the bottom of the mesh are 20 times thicker than the elements at the top. Figures 5.4 and 5.5 illustrate the two meshes.

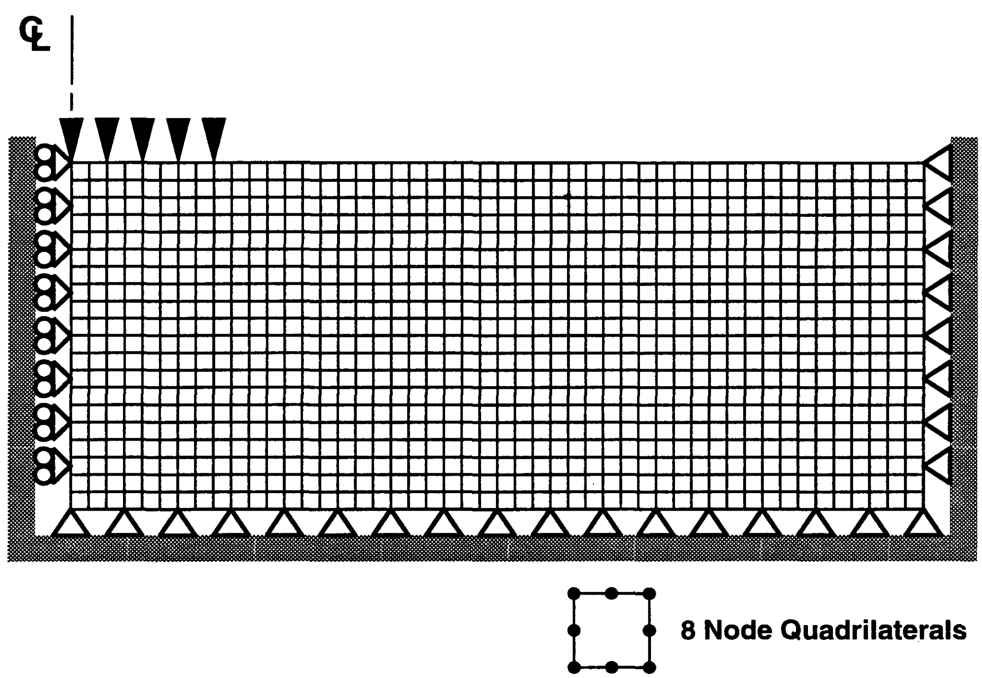

FIG. 5.4. Uniform Mesh

Each mesh was run with both the PCG and AMG algorithms. Attempts were made to construct an incomplete Cholesky preconditioner for use with the PCG

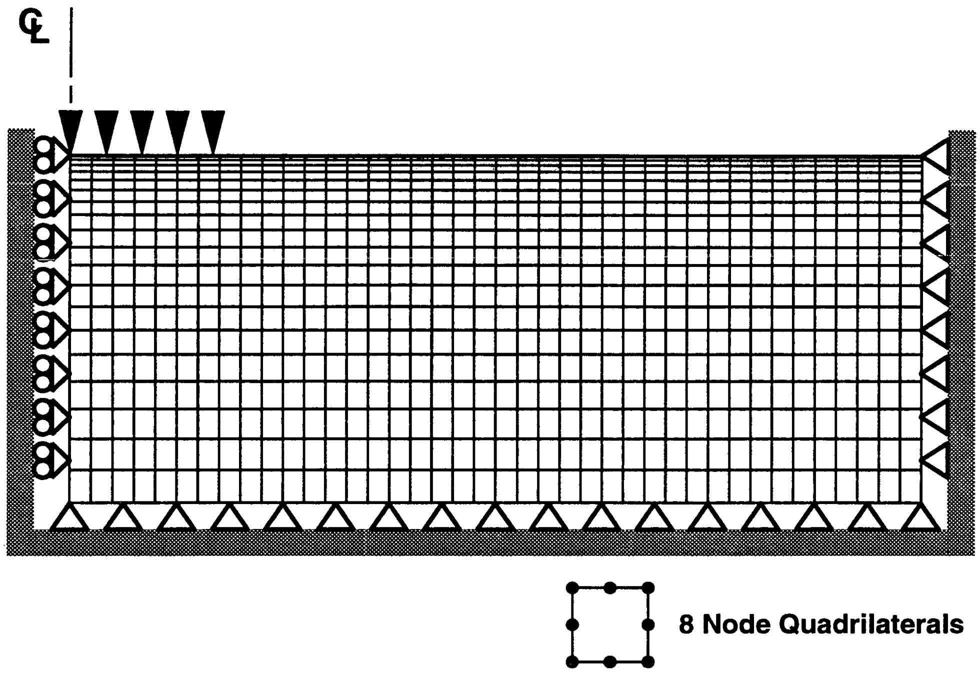

FIG. 5.5. Non-uniform Mesh

algorithm, but the generation process failed on the non-uniform mesh. As a result, PCG was run only with the diagonal preconditioner. Table 5.1 summarizes the results of the runs. Runtimes for the sequential AMG algorithm include the setup of the coarse levels and interpolation operators.

Table 5.1. 2D Foundation Analysis Runtimes

Mesh	PCG Runtime (minutes)	AMG Runtime (minutes)
Uniform Elements	152	190
Non-uniform Elements	356	311

On the uniform mesh, the PCG outperformed the AMG algorithm. This performance is expected as the stiffness matrix resulting from the uniform mesh is relatively well-conditioned and the AMG algorithm was not designed for problems of this type. On the non-uniform mesh, however, both PCG and AMG slowed substantially due to the ill-conditioning present in the stiffness matrix. In this case, AMG was able to outperform PCG with the diagonal preconditioner.

5.2.1 Quality of the Results

As this is a common soil mechanics problem, the analytical solution to this problem is well known. Failure should occur at

$$q_{ultimate} = (2+\pi)c_u \ . \tag{5.1}$$

In this expression, the stress at failure $q_{ultimate}$ is proportional to the undrained cohesion c_u. Given that $c_u = 100$ in this problem, the failure load is

$$q_{ultimate} = (2+3.1416)(100.0) = 514.16 \ . \tag{5.2}$$

Both the PCG and AMG algorithms iterated until the l_2 norm (see (5.3)) of the residual was less than or equal to 0.0001.

$$||\vec{r}|| = ||\vec{b} - A\vec{u}|| = \left(\sum_{i=1}^{n}(b_i - (Au)_i)\right)^{\frac{1}{2}} \tag{5.3}$$

Table 5.2 illustrates the displacements at each stress. Note that both the AMG and PCG runs yield a failure stress of 515.00, which is much closer to the predicted failure load of 514.16 than the result of 520.00 cited in [10]. However, this difference in the accuracy arises from the use of 32 elements in the mesh shown in [10] as compared to the 960 elements in the meshes used for the AMG and PCG algorithms. The results

Table 5.2. Numerical Results for 2D Foundation Analysis - Uniform Mesh

Stress q	Displacement (PCG) (mm)	Displacement (AMG) (mm)
200.00	-0.00661	-0.00661
300.00	-0.01160	-0.01159
350.00	-0.01670	-0.01668
400.00	-0.02351	-0.02348
450.00	-0.03350	-0.03345
480.00	-0.04242	-0.04232
500.00	-0.05087	-0.05070
510.00	-0.05672	-0.05647
515.00	-0.06319	-0.06239

for the non-uniform mesh were very similar to the results shown above.

5.3 3D Pillar Analysis

The primary example problem consists of a 3D elasto-plastic analysis of a cubic pillar. The pillar is assumed to be fixed at the base and undergoes a set of incremental deformations until failure (See Figure 5.6). The Mohr-Coulomb failure criteria is used with this particular example and FORTRAN code for this example is Program 6.4 from [10]. The problem was run assuming the material had an angle of internal friction (ϕ) of 30^o and no cohesion ($c = 0$).

Similar to the 2D example described above, the problem domain was divided due to symmetry and meshed using 1000 20-node hexahedron elements ($10 \times 10 \times 10$). The first mesh contained uniform cubic elements. The second mesh varied the thickness of the elements with depth such that the elements at the bottom of the mesh were 10 times thicker than the elements at the top. Figures 5.7 and 5.8 illustrate these 3D meshes.

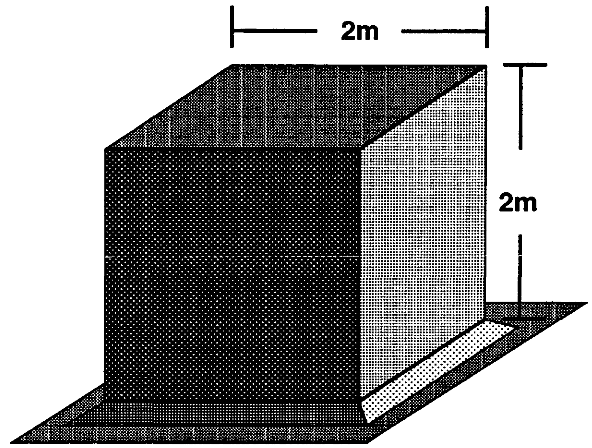

FIG. 5.6. 3D Pillar Analysis

The coarse generation algorithm generated 4 levels $L = 4$ from the original stiffness matrix. Table 5.3 summarizes the coarse levels sizes and storage requirements. Note that the non-uniform mesh did not coarsen as well as the uniform mesh. This

Table 5.3. Coarse Levels - Matrix Sizes

Uniform Mesh			Non-uniform Mesh		
Matrix	Size	Storage	Matrix	Size	Storage
A_1	14,883 × 14,883	32.2 Mb	A_1	14,883 × 14,883	32.2 Mb
A_2	498× 498	2.1 Mb	A_2	990 × 990	5.7 Mb
A_3	42 × 42	26.0 Kb	A_3	69 × 69	70.3 Kb
A_4	6 × 6	0.5 Kb	A_4	9 × 9	1.2 Kb
P_1	14,883 × 498	6.3 Mb	P_1	14,883 × 990	9.3 Mb
P_2	498 × 42	0.3 Mb	P_2	990 × 69	0.8 Mb
P_3	42 × 6	5.3 Kb	P_3	69 × 9	11 Kb

difficulty arises from the poor scaling of the elements within the mesh and is alleviated somewhat by the use of the diagonal preconditioner prior to generation of the coarse levels. Figures 5.9, 5.10, 5.11, and 5.12 illustrate the coarse levels for the uniform mesh. Both the A_3 and A_4 matrices are completely populated, although their small

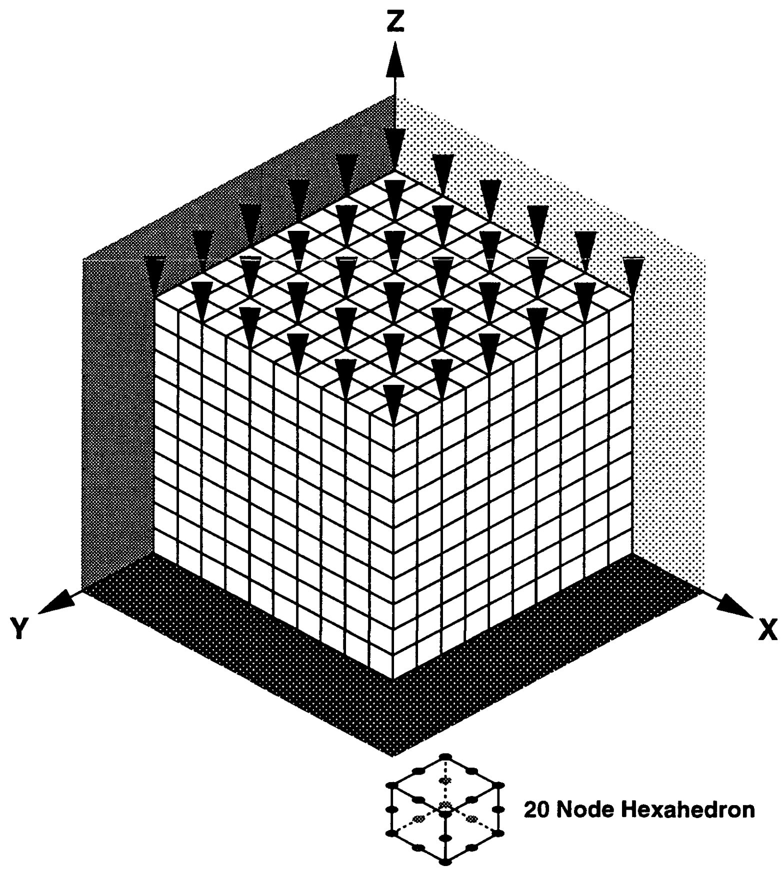

FIG. 5.7. Uniform Mesh

sizes allows for efficient handling.

In the first set of test runs, the meshes were run through both PCG and AMG on a single workstation to generate a baseline runtime for computing speedup and also for comparison purposes. In addition, AMG was also run under a low memory condition to force disk swapping to generate runtimes under swap conditions. 16 Mb of physical RAM was removed from the workstation, reducing the memory from 64 Mb to 48 Mb. The runtimes for the sequential runs are summarized in Table 5.4. As in the 2D example, the AMG runtimes include the time required to setup of the coarse levels and interpolation operators.

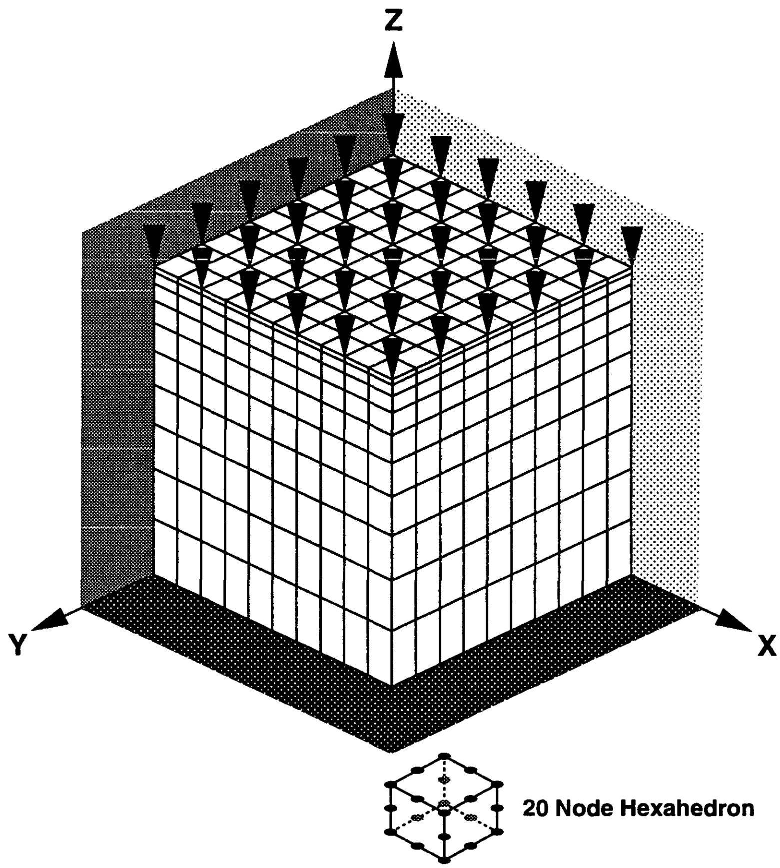

FIG. 5.8. Non-uniform Mesh

Table 5.4. 3D Pillar Analysis Runtimes - Single Workstation

Mesh	PCG Runtime (minutes)	AMG Runtime (minutes)	AMG Runtime (swap) (minutes)
Uniform Elements	32	47	1,366
Non-uniform Elements	125	98	3,038

As with the 2D foundation example, the runtimes indicate that PCG is better suited for problems involving uniform meshes and AMG is better suited to meshes involving poorly scaled elements.

In addition, some swap delays did occur during the generation of the coarse

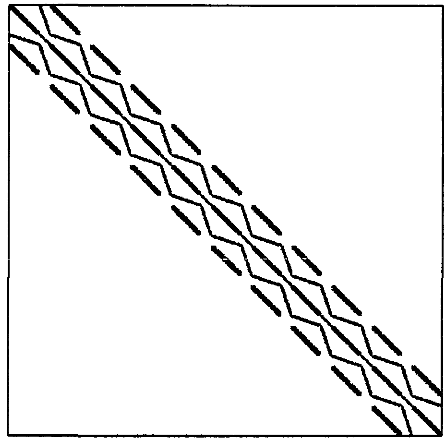

FIG. 5.9. A_1 Matrix - 14,883 $\times$ 14,883

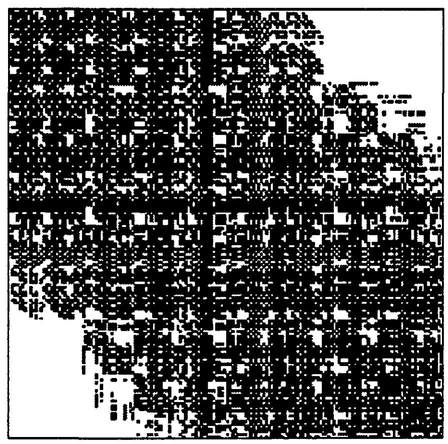

FIG. 5.10. A_2 Matrix - 498 $\times$ 498

levels for the non-uniform mesh on the single workstation run of the AMG algorithm. The generation algorithm creates several temporary sparse matrices during the computation of the coarse levels and the combination of the coarse levels and the temporary matrices forced the workstation to swap. This problem did not occur during the generation of the coarse levels for the uniform mesh due to the smaller storage requirements for that mesh (40 Mb vs 48 Mb). This problem was avoided in the distributed algorithms by moving completed levels to their respective work-

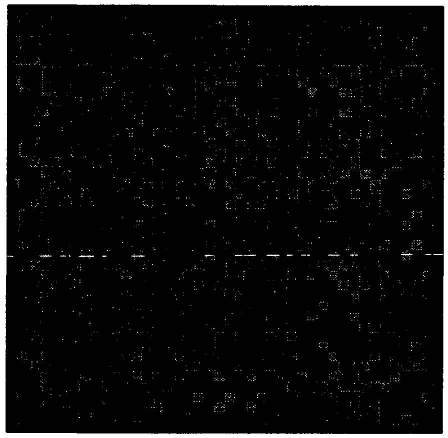

FIG. 5.11. A_3 Matrix - 42×42

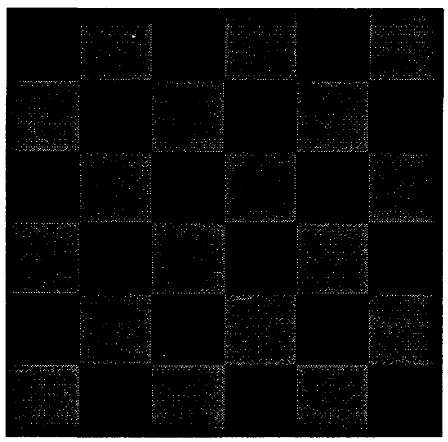

FIG. 5.12. A_4 Matrix - 6×6

stations during the generation process. This reduces the storage requirements on the workstation performing the generation of the coarse levels and prevents swap delays.

In this problem as well as the 2D foundation problem mentioned previously, AMG had better performance on non-uniform meshes than PCG. This performance increase offsets the additional memory requirements of the AMG algorithm. For large problems being run on a single workstation, the choice between the algorithms becomes more difficult. The higher memory requirements of the AMG algorithm will most likely offset the better solution performance due to larger swap delays. Due to the severe performance penalties imposed by swapping, both algorithms are likely to perform identically.

5.3.1 Quality of the Results

As with the 2D foundation problem, the 3D pillar analysis is a simple solid mechanics problem and has a well-known analytical solution. From the the analytical solution, the principal stresses achieve the following maximum values:

$$\sigma_x = 20 \, \frac{kN}{m^2} \, , \tag{5.4}$$

$$\sigma_y = 20 \, \frac{kN}{m^2} \, , \tag{5.5}$$

$$\sigma_z = 60 \, \frac{kN}{m^2} \qquad \text{(Axial Stress)} \, . \tag{5.6}$$

Table 5.5 summarizes the results for the PCG and AMG runs along with the results in [10]. In general, the results obtained from the iterative solution methods matched up well with the analytical results. However, the iterative methods did contain some error, particularly near the end of the analysis. This would indicate that the stopping tolerance of 0.0001 was probably too large.

5.3.2 Distributed Algorithm Performance - No Swapping

In general, the results of the runs agree favorably with the predicted performance on a machine where swapping does not occur. Table 5.6 summarizes the speedups for the uniform and non-uniform meshes.

For the uniform mesh, the results are very consistent. The speedup has dropped

Table 5.5. Numerical Results for 3D Pillar Analysis - Uniform Mesh

Displacement	Stress (PCG) ($\frac{kN}{m^2}$)	Stress (AMG) ($\frac{kN}{m^2}$)
0.050	-25.01	-25.00
0.100	-30.01	-30.00
0.150	-35.01	-35.00
0.200	-40.01	-40.00
0.250	-45.01	-45.00
0.300	-50.01	-50.00
0.350	-55.01	-55.00
0.400	-59.87	-59.97
0.450	-60.10	-59.96
0.500	-60.33	-59.96

Table 5.6. Distributed Algorithms Speedups - No Swapping

Uniform Mesh				
Algorithm	Workstations			
	2	3	4	5
Distributed Levels	0.64	0.65	0.66	N/A
Distributed Levels (parallel)	0.75	0.75	0.75	N/A
Distributed A_1	0.52	0.50	0.45	0.39
Non-uniform Mesh				
Algorithm	Workstations			
	2	3	4	5
Distributed Levels	1.06	1.05	1.05	N/A
Distributed Levels (parallel)	1.24	1.25	1.23	N/A
Distributed A_1	0.83	0.86	0.65	0.63

significantly due to the addition of communication. The distributed levels algorithm had larger speedups than the distributed A_1 level as expected. The distributed levels algorithm exploiting parallelism was able to boost performance slightly, but the extent of the parallelism was not significant enough to overcome communication.

In the case of the non-uniform mesh, the performance of the distributed algorithms appears to have increased dramatically. However, this increased performance can be traced to the presence of a slight amount of swapping during the construction of the coarse levels and interpolation operators. Notice that the use of the distributed processing is beneficial for problems involving even small amounts of swap delays as a way of maintaining performance.

Obviously, the distributed AMG algorithms are not very useful in those problems where enough physical memory is available to prevent swapping. The communication necessary for the distributed algorithms quickly degrades performance. Once the problem begins to exceed the physical memory, the distributed algorithms should be used to overcome the delays caused by disk swapping.

5.3.3 Distributed Algorithm Performance

The runtime for the AMG algorithm with swapping was approximately 30 times slower than normal AMG runtime. Using these runtimes as benchmarks, the distributed algorithms were run on the same two meshes. Table 5.7 and Table 5.8 summarize the results for the uniform and non-uniform meshes respectively.

Table 5.7. 3D Pillar Analysis Runtimes - Distributed Algorithms/Uniform Mesh

Algorithm	Workstations			
	2	3	4	5
	Runtime (minutes)	Runtime (minutes)	Runtime (minutes)	Runtime (minutes)
Distributed Levels	73	72	71	N/A
Distributed Levels (parallel)	62	63	62	N/A
Distributed A_1	91	94	104	120

Table 5.8. 3D Pillar Analysis Runtimes - Distributed Algorithms/Non-uniform Mesh

Algorithm	Workstations			
	2	3	4	5
	Runtime (minutes)	Runtime (minutes)	Runtime (minutes)	Runtime (minutes)
Distributed Levels	92	93	93	N/A
Distributed Levels (parallel)	79	78	80	N/A
Distributed A_1	118	133	150	155

The runtimes for the distributed levels algorithms are slightly better than the runtime for the single workstation run. This deviation arises from the swap delays mentioned previously during the generation of the coarse levels for the non-uniform mesh.

From both sets of results, the best performing algorithm was the distributed levels AMG algorithm with the parallel smoothing step. Considering this algorithm completes 1 extra iteration of SOR for every AMG iteration at the finest level with no extra computational cost, the convergence rate of this algorithm should be slightly higher and the corresponding runtime faster. In comparison, the distributed A_1 algorithm did not perform as well as the distributed levels algorithms, but this is to be expected due to the higher amount of communication inherent in this version of the AMG algorithm. Table 5.9 summarizes the speedups for the different distributed algorithms on the two meshes.

Even the distributed A_1 AMG algorithm at its worst performance provided at least a tenfold improvement over the AMG swap runtime. Figure 5.13 and 5.14 graphically illustrates the results in Table 5.9.

From Table 5.9, the actual speedups obtained during the runs were much greater than the speedups predicted in the previous chapter. Performance was increased by

Table 5.9. Distributed Algorithms Speedups - Swapping

Uniform Mesh				
Algorithm	Workstations			
	2	3	4	5
Distributed Levels	18.71	18.97	19.24	N/A
Distributed Levels (parallel)	22.03	21.68	22.03	N/A
Distributed A_1	15.01	14.53	13.13	11.38
Non-uniform Mesh				
Algorithm	Workstations			
	2	3	4	5
Distributed Levels	33.02	32.67	32.67	N/A
Distributed Levels (parallel)	38.46	38.95	37.96	N/A
Distributed A_1	25.75	22.84	20.25	19.60

over 2000% through the use of distributed processing. This increase is not predicted within the theoretical model as the model does not take into account slowed computation due to swap delays. This deviation illustrates an obvious need for a better theoretical model. Overall, the use of distributed processing was extremely useful in maintaining the AMG algorithm's performance for increasing problem size.

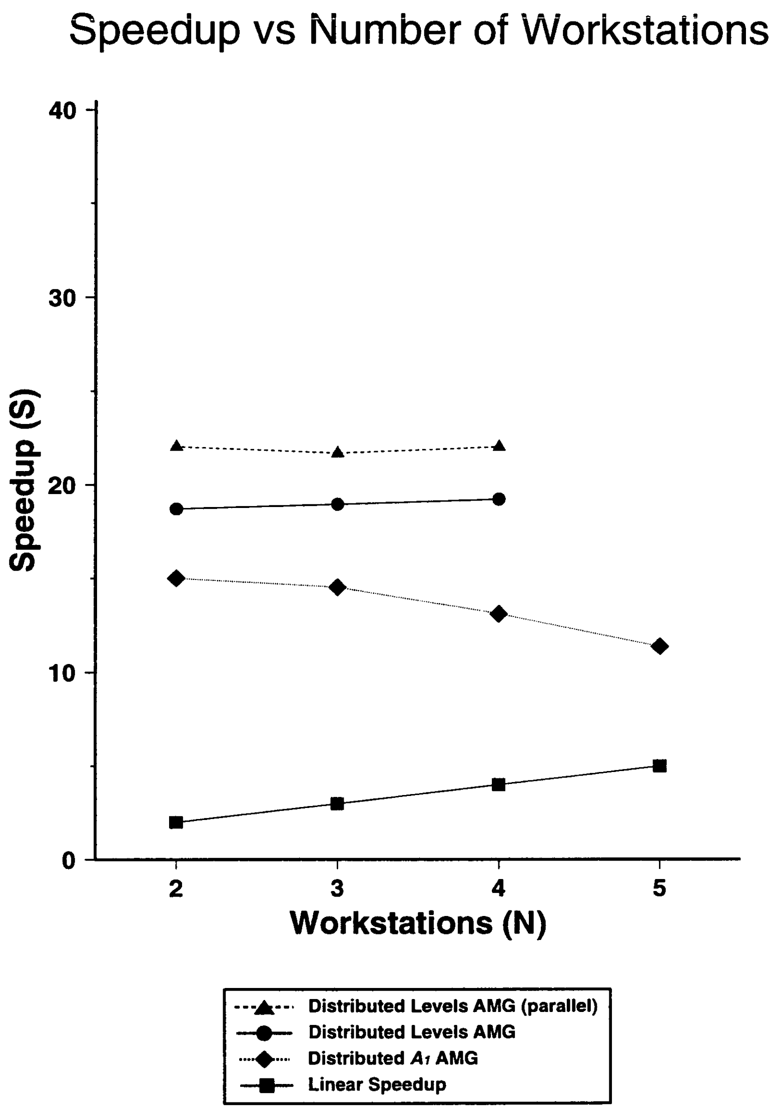

FIG. 5.13. Speedup Curves - Uniform Mesh

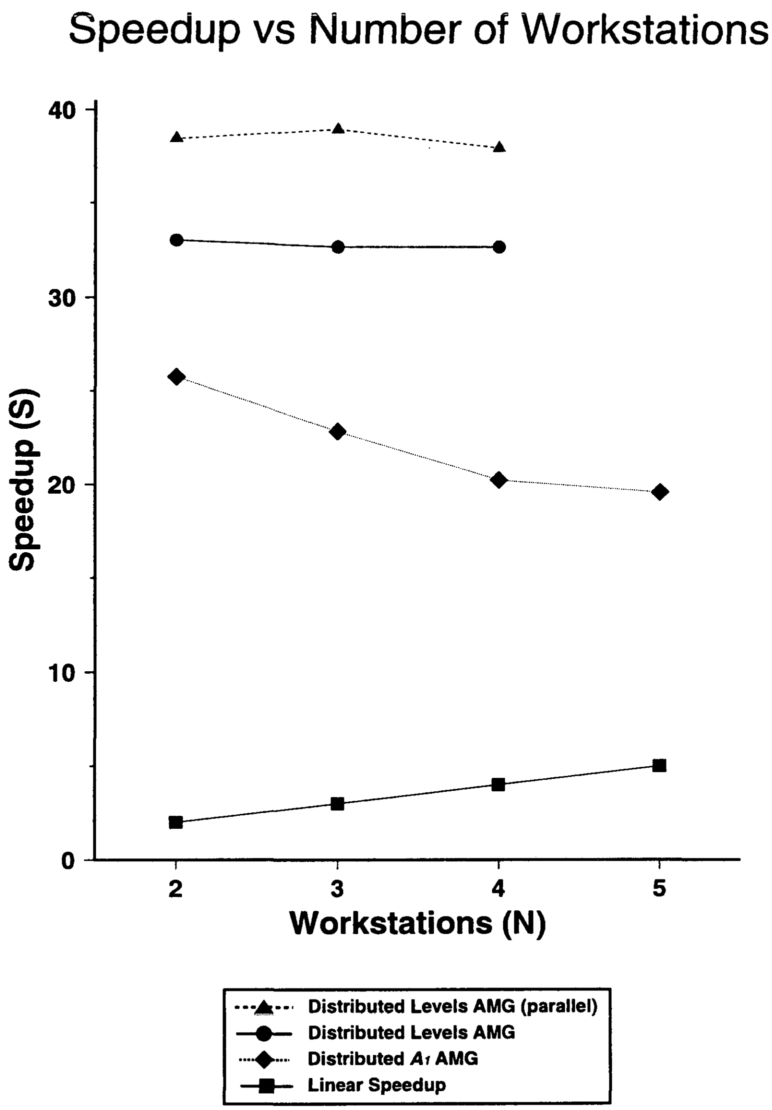

FIG. 5.14. Speedup Curves - Non-uniform Mesh

Chapter 6

CONCLUSIONS

The overall goal of this research was to improve the performance of the finite element method for use with the Monte Carlo method. Since the computational bottleneck in the finite element method was the solution of very large linear systems with multiple similar RHS, the first step was to find a suitable linear solver. Iterative solvers were chosen over direct methods due to their low memory requirements. Of the iterative solvers, the preconditioned conjugate gradient and algebraic multigrid algorithms were selected. From the numerical results, the AMG algorithm was selected due to its better performance on nonuniform meshes. Considering very few realistic problems involve regular problem geometries with uniform elements, AMG is the more practical solver.

The second step was to use distributed processing to overcome the performance degradation caused by swapping. First, suitable data mappings were selected to ensure that the entire problem fit into core memory. Of the possible mappings, two were selected. In the first mapping, each level was mapped to a single workstation and in the second, the A_1 matrix was split into chunks of rows and mapped onto the available workstations. Second, the AMG algorithm was modified in order to access the distributed data structures. These modifications yielded two distributed algorithms, but with some extra modifications, a fourth version of algorithm was also developed that exploited a limited amount of parallelism. Finally, the original and three distributed AMG algorithms were run on several example problems. In general, the distributed levels algorithms performed better than the distributed A_1 algorithm due to their smaller communication costs. Of the two distributed levels algorithm, the one employing the limited parallelism ran substantially faster than the one without parallelism.

The end result of this research is a pair of distributed algebraic multigrid algorithms suitable for use on the very large linear systems arising from the nonlinear finite element analysis of geomechanical problems. While these algorithms do not yield significant speedups on small problems, they are able to overcome the severe performance penalties imposed by limited memories and thus increase the feasibility of using the finite element method as part of the Monte Carlo method for very large problems.

Several opportunities exist for future research in this area. First and foremost is investigation into the use of multiprocessor workstations within the LAN. One possibility is to modify the distributed AMG algorithm to take advantage of a multiprocessor server. The increased processing power of this type of server presents opportunities to improve communication scheduling, message routing, and most importantly, load balancing within the LAN. Another possibility is to consider the use of multiprocessor workstations to exploit parallelism during the processing of distributed tasks.

Another opportunity is to investigate the use of different networking technology. Since the primary performance limitation in the distributed AMG algorithms in the inter-workstation communication time, improved networks would help to improve the algorithm performance. This work ties in closely with the multiprocessor server direction mentioned previously.

Finally, the AMG algorithm could also benefit from better smoothers. The modified block smoothers used in this research are very general as most of the work in this research went into distributing the algorithm. However, better smoothers with higher rates of convergence are another future direction for this research. Clearly, the work in this thesis provides a strong foundation and supports a variety of possibilities for future research.

GLOSSARY

Anisotropic Material

A material whose properties exhibit different behavior in different directions.

Cohesion (c)

A measure of the integrity of a continuous medium.

Elastic Deformation

A material undergoes elastic deformation under a load if the material returns to its original undeformed state when the load is removed.

Factor of Safety

A factor of safety provides a measure of the margin of safety in a given design.

Heterogeneous Material

A material whose properties vary from point to point within the material.

Homogeneous Material

A material whose properties are identical from point to point within the material.

Isotropic Material

A material whose properties exhibit the same behavior in all directions.

Local Area Network (LAN)

A group of computers linked directly together via cables. LANs are extremely useful in that they allow the sharing of system resources, such as disk space, memory, and peripherals.

Plastic Deformation

A material undergoes plastic deformation under a load when the material does not return to its original undeformed state when the load is removed.

Poisson's Ratio (ν)

A measure of the change in the dimensions of a material as the result of a deformation in one direction.

Random Variable

Any variable that assumes each of its possible value x_i with a definite probability (not necessarily the same probability for all values). In general, any variable which may have a probability function is a random variable.

Scalar Field

A scalar field is composed of a set of variables with one local degree of freedom per variable. Examples of scalar field problems include pressure in a fluid, electro-magnetic field strength, and heat distribution in a material.

Young's Modulus (E)

A measure of a material's resistance to tensile/compressive deformation.

REFERENCES

[1] R.S. Sinha, editor. *Underground Structures: Design and Implementation.* Number 59A in Developments in Geotechnical Engineering. Elsevier, 1989.

[2] M.D.G. Salamon and A.H. Munro. A study of the strength of coal pillars. *Journal of the South African Institute of Mining and Metallurgy*, pages 55–67, September 1967.

[3] P. Liu and K. Liu. Selection of random field mesh in finite element reliability analysis. *Journal of Engineering Mechanics*, 119:667–680, April 1993.

[4] G.A. Fenton. *Simulation and Analysis of Random Fields.* PhD dissertation, Princeton University, Department of Civil Engineering and Operations Research, January 1990.

[5] P.D. Spanos and R.G. Ghanem. Stochastic finite element expansion for random media. *Journal of Engineering Mechanics*, 115:1035–53, May 1989.

[6] M.A. Mahtab and P. Grasso. *Geomechanics Principles in the Design of Tunnels and Caverns in Rock.* Number 72 in Developments in Geotechnical Engineering. Elsevier, 1992.

[7] I. Miller, J.E. Freund, and R.A. Johnson. *Probability and Statistics for Engineers.* Prentice Hall, Inc., Englewood, Fourth edition, 1990.

[8] T.L. Saaty and J.M. Alexander. *Thinking with Models: Mathematical Models in Physical, Biological, and Social Sciences*, volume 2 of *Modern Applied Mathematics and Computer Science.* Pergammon Press, 1981.

[9] S.S. Rao. *The Finite Element Method in Engineering.* Pergammon Press, Second edition, 1989.

[10] I.M. Smith and D.V. Griffiths. *Programming the Finite Element Method.* John Wiley & Sons, New York-Chichester-Brisbane-Toronto-Singapore, Second edition, 1988.

[11] M. Papadrakis, editor. *Solving Large-scale Problems in Mechanics.* John Wiley and Sons, 1993.

[12] A. Larabi and F. De Smedt. Solving three-dimensional hexahedral finite element groundwater models by preconditioned conjugate gradient methods. *Water Resources Research*, 30:509–521, February 1994.

[13] C. Farhat. A multigrid-like semi-iterative algorithm for the massively parallel solution of large scale finite element systems. In *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, pages 171–180, Philadelphia, PA, 1989. SIAM.

[14] M.W.F. Grabinsky and J.H. Curran. Efficient mesh generation procedures for finite element analysis of underground structures. *Internal Journal of Rock Mechanics and Mining Sciences*, 30:591–600, 1993.

[15] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method.* Cambridge University Press, Cambridge, MA, 1990.

[16] R.E. Bank and L.R. Scott. On the conditioning of finite element equations with highly refined meshes. *SIAM Journal on Numerical Analysis*, 26:1883–1394, December 1989.

[17] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, August 1994.

[18] J.E. Dendy Jr. Two multigrid methods for three-dimensional problems with discontinuous and anisotropic coefficients. *SIAM Journal on Scientific and Statistical Computing*, 8(5):673–685, September 1987.

[19] R.E. Alcouffe, A. Brandt, J.E. Dendy Jr., and J.W. Painter. The multi-grid method for the diffusion equation with strongly discontinuous coefficient. *SIAM Journal on Scientific and Statistical Computing*, 2(4):430–454, December 1981.

[20] H. Regler and U. Rüde. Layout optimization with algebraic multigrid methods. Technical report, Universität München, 1993.

[21] A. Brandt. Algebraic multigrid theory: The symmetric case. *Applied Mathematics and Computation*, 19:23–56, July 1986.

[22] L.U. Zaslavsky. An adaptive algebraic multigrid for reactor criticality calculations. *SIAM Journal on Scientific Computing*, 16:840–847, July 1995.

[23] A. Brandt, S.F. McCormick, and J. Ruge. Algebraic Multigrid (AMG) for sparse matrix equations. In D.J. Evans, editor, *Sparsity and Its Applications*, Cambridge, MA, 1984. Cambridge University Press.

[24] J. Ruge and K. Stüben. Efficient solution of finite difference and finite element equations. In D.J. Paddon and H. Holstein, editors, *Multigrid methods for integral and differential equations*, pages 169–212. Clarendon Press, Oxford, 1985.

[25] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid on unstructured meshes. Technical Report 34, University of Colorado at Denver, December 1994.

[26] J. Mandel P. Vaněk and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order problems. Technical Report 36, Center for Computational Mathematics, University of Colorado at Denver, 1994.

[27] W. Hackbush. *Iterative Solution of Large Sparse Systems of Equations*, volume 95 of *Applied Mathematical Series.* Springer-Verlag, 1994.

[28] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, 1994.

[29] A. Geist, A. Beguelin, J. Dongarra, W. Jaing, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine.* The MIT Press, Cambridge, MA, 1994.

[30] C.P. Thompson, W.R. Cowell, and G.K. Leaf. On the parallelization of an adaptive multigrid algorithm for a class of flow problems. *Parallel Computing*, 18:449–466, 1992.

[31] A. Greenbaum. A multigrid method for multiprocessors. *Applied Mathematics and Computation*, 19:75–88, July 1986.

[32] C. Lin, W. Proskurowski, and J. Gaudiot. A parallel multigrid method for data-driven multiprocessor systems. In *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, pages 299–318, Philadelphia, PA, 1989. SIAM.

[33] H. Hoppe and H. Mühlenbein. Parallel adaptive full-multigrid methods on message-based multiprocessors. *Parallel Computing*, 3:269–287, 1986.

[34] A. Stewart and G.J. Shaw. A parallel multigrid FAS scheme for transputer networks. *Parallel Computing*, 16:335–342, 1990.

[35] M. Alef. Concepts of efficient multigrid implementation on SUPRENUM-like architectures. *Parallel Computing*, 17:1–16, 1991.