
Donald J. Morton Jr
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/5819

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
An adaptive finite element methodology for the high-performance computer simulation of multiphase flow processes

Morton, Donald J., Jr., Ph.D.
The Louisiana State University and Agricultural and Mechanical Col., 1994
AN ADAPTIVE FINITE ELEMENT METHODOLOGY FOR THE
HIGH-PERFORMANCE COMPUTER SIMULATION OF
MULTIPHASE FLOW PROCESSES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Donald J. Morton, Jr.
B.S., College of Great Falls, 1988
August 1994
Acknowledgements

Above all others, I would like to express my sincere gratitude to my wife, Claire, for her patience and understanding as I devoted our first married years to this venture. She has been strong, and without her I never would have succeeded.

I will always be indebted to my dissertation advisor, Dr. John Tyler, for providing me with the opportunity to pursue this area of research which I have always found exciting. He had faith in me when I barely understood the concept of a differential equation, and has always been supportive. No one could ask for a more decent human being as a major professor.

Dr. Robert Muller, my “second” major professor, has been an inspiration since my first semester at LSU. Deep-thinking and inquisitive, he and his students provided me with an outlet to pursue interests in climatology, always making me feel at home with them. My only regret is that I spent too little time with their group.

Dr. Robert Dorroh, who has no part on my dissertation committee, nevertheless gave many hours of his time to Dr. Tyler and myself as we sought his advice on numerous issues. Providing us with the mathematical guidance to insure we didn’t stray too far from sound principles, he has been a great inspiration in this work.

I am deeply grateful to Dr. Ted Bourgoyne, for initially launching me into this area of research and inviting me on as a graduate research assistant.
His insight, and those of others in Petroleum Engineering, have made this all possible. It is quite difficult for a computer scientist to learn the language of another discipline, but they were all patient and helpful.

Finally, but certainly no less important, I am thankful for the times I have had to interact with my other committee members, Dr. Donald Kraft and Dr. Bush Jones. Dr. Jones has provided me with exceptional guidance and instruction in the areas of numerical methods and reconstructability analysis, and Dr. Kraft has always been an inspiration with his sharp wit and incredible energy.
Table of Contents

Acknowledgements ii
Abstract vi

Chapter 1. Introduction 1

2.1 Background 6
2.1.1 Nature of the Problem 6
2.1.2 A Finite Element Method Approach 7
2.1.3 Review of Recent Attempts 8
2.2 Solution Methodology 12
2.2.1 Development of Mathematical Model 12
2.2.2 Application of Galerkin's Method 14
2.2.3 Adaptive Methods 20
2.3 Simulation of Model Problems 38
2.3.1 Simulating Hypothetical Problems 38
2.3.2 Simulating Laboratory Experiments 39

Chapter 3. A New Three Dimensional Finite Element for Use in 1-Irregular Meshes 62
3.1 Introduction 62
3.2 Construction of Shape Functions 65
3.3 Continuity of Shape Functions Across Interelement Boundaries 73
3.4 Demonstration of Element Performance 78
3.5 Conclusion 82

Chapter 4. Analysis and Implementation of a Distributed Finite Element Methodology Based on Domain Decomposition 83
4.1 Introduction 83
4.2 Parallelization of a Finite Element Method Multiphase Flow Code 85
4.2.1 An Overview of the Parallel Methodology 85
4.2.2 The Substructuring Approach 91
4.2.3 Solving the Global Boundary Node System 99
4.3 Analysis of Relative Speedup Characteristics 112
4.3.1 A General Formula for Estimating Time Complexity 112
4.3.2 Analysis of Complexity – Test Cases 115
4.3.3 Analysis of Complexity – Summary 127
Chapter 5. Summary and Future Directions 138
5.1 Summary ......................................................... 138
5.2 Future R & D ....................................................... 138
   5.2.1 Dynamic Mesh Modification .................. 139
   5.2.2 Mesh Generation and Visualization ........ 140
   5.2.3 Testing of More Realistic Problems .......... 141
5.3 Conclusion ......................................................... 142

Bibliography 143

Appendix 148
A.1 Modified Quadrature for Discontinuous Integrand .......... 148
A.2 Shape Function Derivatives ..................................... 148
   A.2.1 Corner Nodes ............................................. 148
   A.2.2 Mid-edge Nodes .......................................... 150
   A.2.3 Mid-face Nodes .......................................... 152

Vita 154
Abstract

A methodology has been developed for the computer simulation of multiphase flow processes in porous media. The solutions to the nonlinear equations describing these processes are approximated by Galerkin's method on the spatial dimensions and the finite difference method on the temporal dimension. Due to the transient nature of discontinuities in the spatial domain, dynamic mesh refinement (and unrefinement) techniques, based on the maintenance of a 1-irregular mesh, are employed on a two dimensional mesh to produce fine resolution in regions of activity and coarse resolution elsewhere. Our unique approach is tested by comparing computed results with data from laboratory experiments. The groundwork for extending this approach to three dimensional problems is laid in the development of a new finite element for use in 1-irregular adaptive schemes. We describe the development of this element, prove its correctness, and demonstrate its utility in a test problem. Finally, a three dimensional static-mesh version of the approach is distributed over a cluster of workstations, utilizing PVM for message passing. The repeated solution of large systems of equations dominates the computations, and is the focus of the effort in parallelization. Substructuring techniques are employed, allowing for efficient coarse-grained computations due to the distribution of expensive matrix operations over multiprocessors. An analysis of the performance characteristics of this approach is given, followed by a description of tests on a real-world problem.
Chapter 1

Introduction

The accurate computer simulation of multiphase fluid flow processes continues to be a difficult endeavor. Such problems feature near-discontinuities in the solution which are not sufficiently resolved by standard domain discretization procedures without extensive grid refinement. Meanwhile, the problem domains are often large-scale, irregularly shaped, and exhibit heterogeneous characteristics. The theoretical equations which describe these processes are nonlinear, and we often need to solve for more than one variable.

A representative problem is that of oil-water flow through porous media, and in particular, the coning problem which is of interest to petroleum engineers [27, 36, 32]. A petroleum reservoir is composed of porous media saturated with water and oil. The oil, usually being less dense than water, occupies more of the upper regions of the reservoir while water occupies more of the lower regions, often driven by an acquifer. When production at a well begins, oil is initially removed, with the resulting pressure gradient drawing water upwards (see Figure 1.1); this water may eventually be produced along with the oil. This is an undesirable effect, and it may be minimized with an appropriate choice of production parameters. The ability to simulate this process provides petroleum engineers with an economical tool for experimenting
with the production parameters \textit{in vitro}, in order to maximize production of oil while minimizing that of water.

Methods which are developed to simulate these processes may be useful in a number of other disciplines in which flows are described by differential equations. Such processes as groundwater flow and hazardous waste migration are closely related to petroleum engineering problems, and it is obvious to see how improvements in the modelling of one problem may be utilized in obtaining the solution to another. However, other areas such as weather and climate modelling might also make beneficial use of these methods to efficiently resolve relatively small-scale features in large-scale problems. For example, the ability to accurately model fronts between air masses of different types, or to more rigorously handle long-term climatic forcing factors such as the Gulf Stream may be enhanced with these methods. Thus, work in this area has far reaching ramifications in our quest to better understand complex geophysical phenomena.
The purpose of this dissertation is to describe the development, analysis, and implementation of several facets of an ambitious, long-term effort to construct computer simulation tools for modeling a broad spectrum of geophysical processes. The work described here has served as an introduction to the problems inherent to modelling complex phenomena, while providing the foundation for future work.

In Chapter 2, we develop a two dimensional adaptive finite element method for the simulation of two phase flow through porous media. Accurate solutions are obtained for the theoretical equations by utilization of a dynamically adaptive mesh which employs fine discretization in regions of activity and coarse resolution elsewhere. Our mesh modification scheme is a hybrid approach, combining straightforward adaptive mesh techniques of Devloo, Oden, et. al. [15, 39] with a little known element more suited to this approach, developed by Gupta [26]. The methodology is tested by simulating a laboratory experiment designed to study the coning problem, and a comparison is made between the computed and actual solutions. To the best of our knowledge, a computer simulation of a moving front problem, with subsequent comparison of results and real-world data, has not appeared in the literature.

Chapter 3 looks at a key problem in extending this work to three dimensional problems. The adaptive mesh techniques of Chapter 2 may be applied to three dimensional meshes, but no element analogous to the Gupta element previously existed. Thus, we formulate a new finite element which extends the advantages of the Gupta element to three dimensions. A full description of the construction is provided, followed by analysis and testing of the element.
Finally, in Chapter 4 we investigate the distribution of data and workload for a static mesh environment on parallel architectures. A domain decomposition approach is developed and an analysis is conducted in order to gain insight into the performance of the method. The approach is then implemented on a cluster of workstations and tested by modelling the problem discussed in Chapter 2, and speedup characteristics are compared with theoretical values derived in the analysis stage.
Chapter 2

Development and Implementation of a Model for the Simulation of Multiphase Flow in Porous Media

In this chapter, we describe the development and implementation of a numerical model for the simulation of multi-phase flow in porous media. Although the methodology we employ is applicable to a broad class of problems, for simplicity, our work will focus on the modelling of two-phase oil/water flow.

After providing the reader with some background information on the nature of this type of problem, we will discuss our reasons for utilizing the finite element method, and present a review of some recent attempts at finite element modelling of two-phase flow through porous media. Following that, we will discuss the construction of our mathematical model and our subsequent solution techniques. Emphasis is placed on utilizing “clean” mathematical methods, avoiding controversial “tricks” which are frequently found in the literature. Included in our solution methodology is a new technique which allows us to efficiently model problems of this type by localizing most of the computations in regions of high activity.

Verification of our method is provided by conducting tests on a hypothetical problem and simulating a laboratory experiment. We have found that much of the literature in this area introduces techniques which are applied to
hypothetical and academic problems, which may not always reflect the real world. We believe that matching actual physical processes with our model in a mathematically “clean” way provides a greater degree of confidence in our methods.

The chapter ends with a discussion of problems encountered and future directions that may be pursued. Our work is simply an experimental approach, testing various methodologies before proceeding further with development. Our long-range goal is to construct robust and efficient tools for the modelling of broad classes of problems, and we briefly discuss further plans to meet these goals.

2.1 Background

2.1.1 Nature of the Problem

The type of problem we focus on in this chapter is the simulation of the displacement of one fluid by another. These processes are generally described by differential equations derived from principles of conservation of mass, momentum, or energy, and known physical laws [4]. The equations are often highly nonlinear and may be of a general form

\[
\nabla \cdot f_1(v) \nabla u - \nabla \cdot f_2(v) \nabla v = -c_1 \frac{\partial v}{\partial t} + f_3(v) \frac{\partial u}{\partial t} 
\]

\[
\nabla \cdot g_1(v) \nabla u = c_1 \frac{\partial v}{\partial t} + g_2(v) \frac{\partial u}{\partial t} 
\]

in which most of the coefficients are functions of the unknown variables, \( u \) and \( v \). In addition, in many problems there is a near-discontinuity in the fluid properties over the spatial domain as we move from a region occupied by one fluid to a region occupied by another. These factors make it difficult to numerically model the flow processes by traditional techniques, requiring the development of new, high-performance tools.
Several classes of methods exist for the solution of these problems, including finite difference methods (FDM) and finite element methods (FEM). Both methods have been used quite successfully in the solution of differential equations, and each have their pros and cons. However, for reasons outlined below, we believe that finite element methods provide the greatest potential for the development of robust and flexible tools for multiphase flow problems.

2.1.2 A Finite Element Method Approach

Perhaps the greatest advantage of utilizing the finite element method is the abstraction that such a technique provides. In short, the method consists of discretizing the problem domain into a set of elements, coupled by a set of nodes at element vertices and edges. Within each element we construct equations which relate the nodal solutions to elemental properties. Subsequently assembling all of the elemental equations into a global system of equations, applying appropriate boundary conditions, and then solving for values at each of the nodes constitutes the remainder of the finite element method. A key point here is that the operations are similar from element to element, regardless of element size, shape, or physical properties in the region of the domain covered by the element. Thus, irregular and heterogeneous domains may be discretized with a close fit so that boundaries are represented accurately, and regions of the problem domain which have different physical properties are accommodated without modification to the basic algorithm.

Abstraction at the element level also permits us to construct discretizations which place smaller - or alternatively, higher order - elements in regions of activity and larger elements in regions of relative inactivity. This type of construction can provide us with efficient solutions by concentrating most of
the computations where they are most needed and not wasting calculations in regions of inactivity. Most importantly, all of this is possible without losing the accuracy inherent in the finite element approach [21].

Thus, the use of finite element approaches will allow us to model extremely complex problems at least as efficiently and accurately as finite difference methods. In fact, Zienkiewicz [48] has shown that standard finite difference methods may be viewed as a special case of the general finite element method [48]. For simple problems, the finite difference method, with its ease of implementation, will be the logical approach, but for problems that we are interested in, finite element methods will prove to be more useful.

2.1.3 Review of Recent Attempts

This section reviews recent attempts at modelling multiphase flow in porous media with finite element methods. Although we believe that all of these publications make significant and worthwhile contributions, we do not feel that any one publication satisfactorily presents evidence of a robust and versatile approach. Some publications propose interesting techniques and present results from hypothetical models as evidence of their effectiveness. However, there is still some controversy surrounding some of these techniques which make us suspicious of their overall robustness. In addition, we see no evidence that actual physical processes have been properly modelled. Although some of these methods may prove to be effective if tested on real world problems, we must remain suspicious until this is done.

The first of these publications is authored by Gottardi and Mesini [23]. They present a finite element methodology for the simulation of two dimensional oil/water flow, along with a detailed test problem and the source code
of their program. Since their methodology is similar to ours, we had the opportunity to model their test problem, which simulated the injection of water into a corner of a square domain initially saturated with oil, while holding the opposite corner at a constant pressure and saturation. We were able to closely match the saturation and pressure contours which they displayed, but carrying the test further than they did revealed numerous problems with their approach.

Although Gottardi and Mesini use straightforward finite element techniques with sound numerical methods, they used coarsely discretized meshes, which leads to severe oscillatory solutions in all but the simplest problems [25]. Close examination of the source code revealed “tricks” which would reduce oscillations. For example, nodes with “wrong” solutions were “adjusted” and held constant (through essential boundary conditions) at the next time step to force them towards better values. We found, through testing of these methods, that more rigorous test problems yielded grossly inaccurate results.

The problems revealed by this paper encouraged us to view other techniques with a healthy degree of skepticism and to watch out for certain approaches. First, as much as possible, solution methods should consist of mathematically sound techniques. As will be discussed in Section 2.2.3, the use of finite element methods to solve transient problems will inherently produce oscillations in coarsely refined regions experiencing sharp transitions. It appears that the only dependable way to reduce these oscillations is to more finely discretize the problematic regions. However, it is popular in the finite element community to use other techniques which may suppress the oscillations but may not converge to the proper solution [25]. Although these methods have much potential, the literature shows scant evidence of actual physical problems
being modelled successfully with them. Thus, in our opinion, these approaches should not be adopted for use in a potentially robust and flexible model until they have been rigorously tested on real-world problems. With this in mind, we now briefly discuss other recent approaches to the finite element modelling of multiphase flow processes.

Chavent, et. al. [6, 7] develop a formulation of the diphasic flow equations and use rather complex solution techniques, producing sharper fluid interfaces than are obtainable through standard finite difference methods. Using a combination of upwinding techniques and slope limiting, they also are able to reduce spurious oscillations. Thus, their solutions exhibit desirable characteristics such as non-oscillatory behavior and sharp fronts in an apparently efficient approach. However, close examination of their results reveals that even with these specialized techniques, sharpness of the front is limited by the grid resolution, which means that problems with sharp fluid interfaces will not be modelled accurately without extensive refinement of the finite element mesh. The work of Chavent, et. al. represents a significant improvement over finite difference methods and introduces techniques which may be useful for achieving stable solutions efficiently. However, the upwinding methods are controversial [35, 25], and no evidence is presented which would indicate positive results when these techniques are used to model real world problems.

Katyal and Parker [30] present a novel technique for modelling the movement of fluid fronts through problem domains, apparently solving the theoretical equations in a “clean” form by utilizing fine resolution on a static mesh to obtain the needed accuracy and stability. An Adaptive Solution Domain
(ASD) finite element method is used to achieve efficiency by removing inactive regions of the mesh from the computations. In this approach, elements in the mesh are labelled “active” or “inactive” depending on the variability of the solution within an element. If all elements which share a given node are labelled “inactive”, then the node is removed from the computations, reducing the size of the global system of equations. Katyal and Parker demonstrate their techniques on simple, hypothetical problems, focusing on the greatly increased efficiency of this technique over methods which compute on the entire mesh. The ASD method appears to be a viable technique for many problems. However, the implications of totally removing portions of the problem domain from computations should be considered. Finally, as in papers discussed above, believable results are obtained, but there is no evidence that the methodology is useful on real world problems.

An approach much like the one adopted by ourselves (see Section 2.2), but with some potential advantages in terms of solution stability, is presented by Sukirman and Lewis [47]. The equations used by the authors are formulated and solved implicitly for the pressures of each fluid, then auxiliary functions are utilized to solve for the saturations of each fluid. The auxiliary functions, built on a solid mathematical foundation, will guarantee stable saturations provided that the pressures are stable. Our approach, which solves implicitly for the pressure and saturation of a single fluid, results in stable pressures, but often oscillatory saturations. Thus, utilizing this alternative approach may prove to be quite helpful. Sukirman and Lewis also make use of “clean” solution methods, providing several test problems to demonstrate their results. Although these tests are compared favorably with tests performed by other researchers,
they too are based on relatively simple problems. In addition, since no use is made of adaptive methods, this particular approach would require extensive mesh refinement to model realistic problems. On the other hand, this approach could be incorporated into our existing adaptive scheme with minimum effort.

The remainder of this chapter is devoted to describing and testing our own methodology for the solution of multiphase flow problems. Although the work reviewed above presents some clever and useful techniques, we believe our work offers a viable approach to the solution of theoretical equations which describe these flow processes. By removing controversial “tricks” from the problem solving, we are better able to determine where problems in the mathematical models or the computational techniques lie. It may be advantageous to incorporate some of the schemes described above in future work, but only after we have verified that the basic modelling and simulation routine is sound.

2.2 Solution Methodology

2.2.1 Development of Mathematical Model

The equations which describe the flow of two immiscible fluids through porous media (see [41, 1]) are derived from the mass conservation equations

\[ -\nabla \cdot \rho_w \vec{v}_w - q_w = \frac{\partial}{\partial t} (\phi \rho_w S_w) \]  
\[ -\nabla \cdot \rho_n \vec{v}_n - q_n = -\{(f) \rho_n S_n \} \]  

and the relations

\[ S_n + S_w = 1 \]  
\[ p_c = p_n - p_w. \]

In the above, subscripts \( w \) and \( n \) designate wetting and non-wetting fluids, respectively (e.g. in a water-oil system, water would be the wetting fluid
and oil would be the non-wetting fluid). For a fluid $\ell$, $\rho_\ell$ is the density, $\vec{u}_\ell$ is the mass velocity vector, $q_\ell$ is the sink/source term, $p_\ell$ is the pressure, and $S_\ell$ is the saturation, or fraction of pore volume occupied by the fluid. $p_c$ is the capillary pressure, treated as an empirically derived function of the fluid saturation, and $\phi$ is the porosity of the medium.

Darcy's Law is used to expand the mass velocity vectors to

$$\vec{v}_\ell = -K \frac{k_\ell}{\mu_\ell} \nabla [p_\ell - \rho_\ell g z],$$

where $K$ is the absolute permeability tensor of the porous media, $k_\ell$ is the relative permeability of fluid $\ell$, also an empirically derived function of saturation, and $\mu_\ell$ is the viscosity of fluid $\ell$. $g$ is the acceleration due to gravity, and $z$ is the depth in the porous media. The absolute permeability tensor is combined with the relative permeability and viscosity to make up a mobility tensor for fluid $\ell$, which is designated as $\tilde{M}_\ell$.

These equations are recast to form a coupled system of equations in the unknown variables $p_n$ and $S_n$,

$$\nabla \cdot \tilde{M}_w \nabla p_n - \nabla \cdot \tilde{M}_w \frac{dp_n}{dS_n} \nabla S_n - \nabla \cdot \tilde{M}_w \nabla (\rho_w g z) - \frac{q_w}{\rho_w} = -\phi \frac{\partial S_n}{\partial t} + c_w \phi (1 - S_n) \frac{\partial p_n}{\partial t}.$$  \hspace{1cm} (2.8)

$$\nabla \cdot \tilde{M}_n \nabla p_n - \nabla \cdot \tilde{M}_n \nabla (\rho_n g z) - \frac{q_n}{\rho_n} = \rho \phi \frac{\partial S_n}{\partial t} + c_n \phi S_n \frac{\partial p_n}{\partial t}.$$  \hspace{1cm} (2.9)

Note that the recasting of the equations has introduced the fluid compressibilities $c_w$ and $c_n$. Also, note that some parameters may be expressed as functions of $p_n$ or $S_n$, making these equations nonlinear.
2.2.2 Application of Galerkin’s Method

An approximate solution to (2.8) and (2.9) is obtained by discretizing the problem domain $\Omega$ into a set of finite elements and applying Galerkin’s Method (see [4, 29]). First, all terms in (2.8) and (2.9) are moved to the left-hand side, and the resulting equations are defined as the residual equations

\begin{align}
R_w &= 0 \\
R_n &= 0.
\end{align}

(2.10)  
(2.11)

Assuming that an appropriate set of shape functions $N_i$, where $i = 1, \ldots, n$, has been defined, the weighted residual statements

\begin{align}
\int_{\Omega_e} R_w N_i \, d\Omega_e &= 0 \\
\int_{\Omega_e} R_n N_i \, d\Omega_e &= 0
\end{align}

(2.12)  
(2.13)

are minimized over each element domain $\Omega_e$, and the contributions of each element are assembled into a global system matrix. Finally, the resulting system of equations is solved to obtain the approximate solution.

The second-order derivative terms in the weighted residual statements ((2.12) and (2.13)) are integrated by parts to reduce the order of the derivatives to one, and recast as the elemental equations [23]

\begin{align}
[A] \{p_n\} + [CW] \{\dot{p}_n\} + \\
[B] \{\dot{S}_n\} + [C] \{\ddot{S}_n\} &= \\
&- \{Q_w\} - \{D\}
\end{align}

(2.14)

\begin{align}
[H] \{p_n\} + [CN] \{\dot{p}_n\} - [C] \{\ddot{S}_n\} &= \\
&- \{Q_n\} - \{U\}
\end{align}

(2.15)
where coefficients are

\[ A_{ij} = - \int_{\Omega_e} (\nabla N_i)^T \tilde{M}_w \nabla N_j \, d\Omega_e \]  

(2.16)

\[ CW_{ij} = \phi_{cw} \left[ \sum_{k \in \Omega_e} \left( S_n^t \int_{\Omega_e} N_i \nabla N_j \, d\Omega_e \right) - \int_{\Omega_e} N_i N_j \, d\Omega_e \right] \]  

(2.17)

\[ B_{ij} = \frac{d\rho_e}{dS_n} \int_{\Omega_e} (\nabla N_i)^T \tilde{M}_w \nabla N_j \, d\Omega_e \]  

(2.18)

\[ C_{ij} = \phi \int_{\Omega_e} N_i N_j \, d\Omega_e \]  

(2.19)

\[ H_{ij} = - \int_{\Omega_e} (\nabla N_i)^T \tilde{M}_m \nabla N_j \, d\Omega_e \]  

(2.20)

\[ CN_{ij} = - \phi_{cn} \sum_{k \in \Omega_e} S_n^t \int_{\Omega_e} N_i N_j N_k \, d\Omega_e \]  

(2.21)

\[ D_i = \sum_{j \in \Omega_e} \left\{ \int_{\Omega_e} (\nabla N_i)^T \tilde{M}_w \nabla N_j \, d\Omega_e \right\} \rho_{w_j}^t g_{z_j} \]  

(2.22)

\[ U_i = \sum_{j \in \Omega_e} \left\{ \int_{\Omega_e} (\nabla N_i)^T \tilde{M}_n \nabla N_j \, d\Omega_e \right\} \rho_{n_j}^t g_{z_j} \]  

(2.23)

\[ Q_{wi} = \int_{\Omega_e} \rho_{wi}^t N_i \, d\Omega_e \]  

(2.24)

\[ Q_{ni} = \int_{\Omega_e} \rho_{ni}^t N_i \, d\Omega_e \]  

(2.25)

In the above, summations are performed over the nodes which define an element. Superscripts of \( t \) denote values obtained from the most recent timestep.

Integration of (2.16) through (2.25) is performed numerically using Gauss-Legendre quadrature. Although the development of the integration scheme is beyond the scope of this dissertation, we provide a brief overview, refering the reader to [4, Chapters 8 and 13] for a thorough and readable discussion.

We first assume that for each type of element in the problem domain, there exists a parent element with shape functions defined over a local \((\xi, \eta, \zeta)\)
coordinate system which is mapped to the global \((x, y, z)\) coordinate system
(In two dimensions, we naturally are only concerned with \((\xi, \eta)\) and \((x, y)\) coordinates, and in one dimension with \(\xi\) and \(z\) coordinates) by use of a Jacobian matrix.

In two dimensions, an integration of the form

\[
\int_{\Omega_e} (\nabla N_i)^T \alpha \nabla N_j \, d\Omega_e
\]

is approximated numerically as

\[
\int_{\Omega_e} (\nabla N_i)^T \alpha \nabla N_j \, d\Omega_e \approx \sum_{k=1}^{n} \sum_{\ell=1}^{n} w_k w_\ell \left[ \left( \nabla N_i(\xi_k, \eta_\ell) \right)^T \alpha(\xi_k, \eta_\ell) \nabla N_j(\xi_k, \eta_\ell) \right] |J^{(e)}(\xi_k, \eta_\ell)|
\]

(2.27)

where \(|J^{(e)}(\xi_k, \eta_\ell)|\) is the determinant of the Jacobian matrix, \(J^{(e)}\), for element \(e\), evaluated at Gauss point \((\xi_k, \eta_\ell)\) in the parent element coordinate system. \(w_i\) is the weight associated with Gauss point \(i\), and \(n\) is the number of Gauss points used to integrate over a single dimension in the parent element. The Jacobian, representing the local mapping between parent shape functions and real shape functions is given in two dimensions as

\[
J^{(e)}(\xi, \eta)) = \begin{bmatrix}
\frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\
\frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta}
\end{bmatrix}
\]

(2.28)

where

\[
\frac{\partial x}{\partial \xi} = \sum_{m \in \Omega_e} x_m \frac{\partial N_m(\xi, \eta)}{\partial \xi}
\]

(2.29)

\[
\frac{\partial y}{\partial \xi} = \sum_{m \in \Omega_e} y_m \frac{\partial N_m(\xi, \eta)}{\partial \xi}
\]

(2.30)

\[
\frac{\partial x}{\partial \eta} = \sum_{m \in \Omega_e} x_m \frac{\partial N_m(\xi, \eta)}{\partial \eta}
\]

(2.31)

\[
\frac{\partial y}{\partial \eta} = \sum_{m \in \Omega_e} y_m \frac{\partial N_m(\xi, \eta)}{\partial \eta}
\]

(2.32)
and $x_m, y_m$ are coordinates for node $m$ in the real, or global coordinate system, and $N_m$ is the parent shape function defined in the local coordinate system, associated with node $m$. Other integrals are evaluated in a similar manner, for example

$$\int_{\Omega_e} N_i N_j \, d\Omega_e \approx \sum_{k=1}^{n} \sum_{l=1}^{n} w_k w_l \left[ N_i(\xi_k, \eta_l) N_j(\xi_k, \eta_l) \left| J^{(e)}(\xi_k, \eta_l) \right| \right]. \quad (2.33)$$

An important point to note is that nothing in these formulas restricts their use to a specific type of element. Given any element type with shape functions defined on an isoparametric parent element (which in two dimensions would have shape functions defined over $-1 \leq \xi \leq 1$ and $-1 \leq \eta \leq 1$) the integrations may be performed in a general manner, adding immensely to the abstraction the finite element method allows. Thus, for a given element type, we need only define (in the parent coordinate system) the shape functions and their derivatives, and the Gauss points and weights to be used in the Gauss-Legendre quadrature numerical integration of an element of this type.

Integration-by-parts of the second-order derivative terms in (2.12) and (2.13) also produces flux terms which may be used for the application of Neumann boundary conditions. In this model, the flux terms obtained from the first terms in (2.8) and (2.9) are utilized to specify flow rates and it is assumed that the other flux terms have negligible effect on the solution. However, more rigorous treatment of these terms might improve the accuracy of the solution and is being considered for future work.
The $p_n$ and $S_n$ in (2.14) and (2.15) are time derivatives, which may be approximated with a backwards difference scheme

\begin{align}
\dot{p}_n &\approx \frac{p_{n+\Delta t}^t - p_n^t}{\Delta t} \quad (2.34) \\
\dot{S}_n &\approx \frac{S_{n+\Delta t}^t - S_n^t}{\Delta t}. \quad (2.35)
\end{align}

Incorporating these approximations and re-arranging (2.14) and (2.15) gives us

\begin{align*}
\left([A] + \frac{1}{\Delta t} [CW]\right) \{p_{n+\Delta t}^t\} + \\
\left([B] + \frac{1}{\Delta t} [C]\right) \{S_{n+\Delta t}^t\} = - \{Q_w\} - \{D\} + \frac{1}{\Delta t} [CW] \{p_n^t\} + \\
\frac{1}{\Delta t} [C] \{S_n^t\}
\end{align*}

\begin{align}
\left([H] + \frac{1}{\Delta t} [CN]\right) \{p_{n+\Delta t}^t\} - \\
\frac{1}{\Delta t} [C] \{S_{n+\Delta t}^t\} = - \{Q_n\} - \{U\} + \frac{1}{\Delta t} [CN] \{p_n^t\} - \\
\frac{1}{\Delta t} [C] \{S_n^t\}. \quad (2.37)
\end{align}

Elemental equations (2.36) and (2.37) are assembled into the global system of equations

\[ [M] \{u\} = \{b\} \quad (2.38) \]

where

\[ \{u\} = \left[ p_{n_1}^{t+\Delta t}, S_{n_1}^{t+\Delta t}, \ldots, p_{n_h}^{t+\Delta t}, S_{n_h}^{t+\Delta t} \right]^T. \quad (2.39) \]

Due to the nonlinearity of this problem, these equations must be solved iteratively over each time step, re-evaluating the coefficients of $[M]$ and $\{b\}$. 
with each new solution \( \{u\} \). In this work, we utilize an adaptive time-stepping scheme in which the size of the timestep is increased when convergence occurs rapidly and decreased when an excessive number of iterations are required for convergence. A balance is required here, as each iteration requires an expensive solution to the system of equations. An excessively large timestep will result in too much time being spent on the solution of these equations, whereas an excessively small timestep will be wasteful.

To aid in better understanding the flow of events in the finite element computations, we present the following basic algorithm:

```
PROGRAM fem
  FOR t = t₀ TO tₙ DO
    {Dynamic adjustment of \( \Delta t \)}
    REPEAT
      FOR elmt = 1 TO num_elements DO
        {Evaluate nonlinear coefficients and construct elemental equations}
      ENDFOR
      {Add elemental equations to global system of equations}
    UNTIL converged
  END FOR
END PROGRAM
```

The above procedure constitutes the basic methodology for solving the equations of two-phase flow in porous media. In the simplest of problems,
accurate results may be anticipated. However, in more complex problems, and most notably in real-world problems, a high degree of spatial and temporal resolution will be needed to obtain accurate results. The degree of resolution required will be prohibitive in many instances, but numerous strategies exist for reducing the computational resources needed.

2.2.3 Adaptive Methods

Benefits of Adaptive Methods

Finite element methods, like finite difference methods, are prone to oscillations, or "wiggles", in the solution, particularly in problems dominated by convective or advective flow. Many effective "tricks" have been developed to dampen these wiggles, but their applicability to general, real-world problems is still debated [35, 25, 3, 42]. Further, all of these schemes are based on reducing the negative effects resulting from the numerical approximation of convection and/or advection terms, failing to address a little-mentioned, but fundamental problem which results from modelling sharp transitions with finite element methods [25, 20]. This problem may be illustrated by observing the finite element solution of a linear, one-dimensional, transient heat-diffusion equation

$$\frac{d^2u}{dx^2} = \frac{du}{dt}$$  \hspace{1cm} (2.40)

on a domain of length $L = 1$, partitioned into four elements of equal length (Figure 2.1(a)), $h = 1/3$, with initial conditions

$$u(x,0) = 0.0, \quad 0 < x \leq L$$  \hspace{1cm} (2.41)

$$u(0,0) = 1.0$$  \hspace{1cm} (2.42)

and boundary conditions

$$u(0,t) = 1.0$$  \hspace{1cm} (2.43)
The diffusion equation is transformed by a standard Galerkin finite element method to

$$[K] - [C] \left\{ \frac{\partial u}{\partial t} \right\} = 0 \quad (2.45)$$

Using a backwards-difference approximation for the time derivative, which is known to be unconditionally stable [4], with a step size of $\Delta t = 0.004$, we obtain a physically impossible solution, although in this simple example, convergence to the correct steady-state solution [25] eventually occurs (see Figure 2.2). If we further refine the problem domain, the wiggles disappear and, as expected, the solution becomes more accurate. In fact, if we refine only the region of transition, using an adaptive mesh (see Figure 2.1(b)), we obtain a nearly exact solution (see Figure 2.2).

Another method for reducing wiggles, at least in this simple problem, is to consolidate the entries of the mass matrix ([C] in 2.45) into the diagonal [25, 4]. However, there are some problems for which this actually generates wiggles. Thus, we cannot consider this to be a robust cure. As Gresho [25] and others [35] argue, the underlying problem is that the mesh is too coarsely refined to
Figure 2.2: Solutions of heat diffusion equation at $t = 0.012$. 
handle the transition, and the only dependable method for eliminating wiggles is to further refine the mesh in these regions.

The need to refine various regions within the domain motivates us to consider adaptive grid methods, wherein we refine only the regions of the grid with much activity, and settle for coarse refinement in regions of little transition or activity. Fortunately the finite element method, with its emphasis on element-level abstraction, enables us to easily compute solutions on such irregular meshes; this would not be a straightforward task with finite difference methods, and would result in a loss of accuracy [21]. Furthermore, since we wish to model problems in which sharp transitions move through the problem domain as time progresses, the use of dynamic adaptive grid methods allows us to refine and unrefine regions of the mesh as the simulation proceeds, localizing the computations where they are most needed.

A Dynamic Mesh Modification Scheme

Numerous dynamic mesh refinement methods appear in the literature, apparently blossoming in the mid 1980's [39, 18, 13, 15]. Of particular interest to us for its simplicity and applicability to moving-front problems is the method introduced by Devloo, et. al. [15], which serves as a foundation for our work. This method is based on the maintenance of a 1-irregular mesh composed of quadrilateral elements. A 1-irregular mesh in two dimensions is one in which each element side is adjacent to no more than two elements, or, stated in another way, the difference in the degree of refinement between two adjacent elements does not exceed one level. For example, the mesh of Figure 2.3(a) is 1-irregular, but the mesh of Figure 2.3(b) is not, since a side of the upper left element is adjacent to three elements.
The algorithm for maintaining the 1-irregular mesh is conceptually simple. In order for an element to be refined, it must be insured that doing so will not violate the 1-irregularity of the mesh. For example, in Figure 2.3(a), refinement of element A would result in a 2-irregular mesh. Therefore, in order to legally refine element A, it would be necessary to first refine any adjacent elements which would otherwise violate the 1-irregularity constraint. This will normally be a recursive procedure, since the refinement of elements adjacent to A might in turn be restricted by other elements. Thus, to refine element A, we first check elements on each side of A, and if they require refinement in order to maintain 1-irregularity, then they will call the refinement procedure, and so on. When the refinement procedure on the initial element has finished checking and refining any necessary adjacent elements, the initial element will be refined.

Unrefinement occurs by replacing a group of four elements (the group being initially created by a refinement of a single element) with a single element.
only if this will not violate the 1-irregularity of the mesh. Although this implies that some groups will not be successfully unrefined, it prevents the unrefinement of adjacent groups which may require continued refinement for greater solution accuracy. Thus, the refinement/unrefinement procedure plays it safe by sometimes refining more elements than solution accuracy may dictate, and only unrefining groups which are known to no longer require refinement.

Critical in any adaptive method is the maintenance of interelement continuity along the boundaries of transition elements. If we utilize standard element shape functions, then the solution along the transition edge of a larger element (see Figure 2.4) will be quadratic (since it is defined by three nodes - see Figure 2.5(a)), while the solution along the transition edges of the smaller elements will be piecewise linear (see Figure 2.5(b)), violating the requirement of interelement continuity of the shape functions and solution. A common approach for ensuring interelement continuity, and the one utilized by Devloo, et.
Figure 2.5: (a) Quadratic solution. (b) Piecewise linear solution. (c) Constrained linear solution.

al., is to constrain the solution at the middle node of the transition boundary to be a linear interpolation of the solution on the outer nodes (see Figure 2.5(c)).

Although the constraint method is a sound approach for insuring interelement continuity along transition boundaries, we believe there are disadvantages which can be improved. First, constraining the solution at the middle node of the transition will "smooth" the solution and, in the worst case, result in element "locking", a situation in which the solution at other nodes may lock as a result of too many constraints [8]. In addition, the special operations required for the constraint of a nodal value can be somewhat cumbersome, and doing so forces us to deviate from the idea of viewing the finite element procedure as an element-level abstraction, since the assembly of individual elemental equations will now be dependent on factors outside the scope of the element.
A Piecewise Linear Transition Element

A rarely-cited paper by Gupta [26] provides us with an attractive alternative for insuring interelement continuity along transition boundaries [31]. Gupta derived a generic element with piecewise linear shape functions, for use in the larger elements found along 1-irregular transition boundaries. This assured interelement continuity by forcing the solution along the three-node edge to be equivalent to the solution defined by the nodes of the two smaller adjacent elements (see Figure 2.4). Gupta also introduces modified integration rules for the new element which compensate for the fact that derivatives of the shape functions are discontinuous. Since the shape function derivatives are continuous within each of the element quadrants, the modified rules, which assume an approximate integration by Gaussian quadrature techniques, simply sum the integrations of the quadrants in the new element to serve as a valid integration over the whole element.

The Gupta element is defined in a manner which allows us to incorporate all possible element configurations into a single quadrilateral element type. All elements of this type are defined as having a node on each vertex, with “optional” nodes on the midpoint of each edge. Shape functions for the vertex nodes are constructed (in fact, they are the shape functions used for bilinear quadrilateral elements), followed by the piecewise-linear shape functions for nodes on the edge midpoints. The introduction of shape functions for the mid-edge nodes requires us to go back and modify the shape functions for the vertex nodes, in order to maintain compatibility between the two sets of shape functions. The modified shape functions for each of the vertex nodes are constructed by subtracting influences of the shape functions for the adjacent
mid-edge nodes. If a mid-edge node is not present, its shape function is set to zero, resulting in no modification to the shape functions for the vertices adjacent to it. Thus, a vertex shape function is only modified by adjacent mid-edge nodes. A depiction of the Gupta element and its node-numbering convention is provided in Figure 2.6.

The “original” shape functions for the vertex nodes are defined on a parent element in a standard isoparametric coordinate system (shape functions defined for $-1 \leq \xi, \eta \leq 1$) with the origin located at the centroid of the element and the vertical coordinate $\eta$ positive in the downward direction (we choose $\eta$ to be positive downward simply because our applications use depth as the vertical parameter - the shape functions are easily modified to accomodate $\eta$ being positive in the upward direction). The shape functions for the standard two-dimensional bilinear quadrilateral element are

$$\tilde{N}_1(\xi, \eta) = \frac{1}{4}(1 - \xi)(1 + \eta) \quad (2.46)$$
$$\tilde{N}_2(\xi, \eta) = \frac{1}{4}(1 + \xi)(1 + \eta) \quad (2.47)$$
\[ \tilde{N}_3(\xi, \eta) = \frac{1}{4}(1 + \xi)(1 - \eta) \]  
\[ \tilde{N}_4(\xi, \eta) = \frac{1}{4}(1 - \xi)(1 - \eta) \]  

Shape functions for the mid-edge nodes are

\[ \bar{N}_5(\xi, \eta) = \frac{1}{2}(1 - |\xi|)(1 + \eta) \]  
\[ \bar{N}_6(\xi, \eta) = \frac{1}{2}(1 + \xi)(1 - |\eta|) \]  
\[ \bar{N}_7(\xi, \eta) = \frac{1}{2}(1 - |\xi|)(1 - \eta) \]  
\[ \bar{N}_8(\xi, \eta) = \frac{1}{2}(1 - \xi)(1 - |\eta|) \]  

As described above, the shape functions \( \tilde{N}_1, \ldots, \tilde{N}_4 \) are modified so that they evaluate to 1 at their defined node and 0 at all other nodes. When a mid-edge node is not present, its shape function (as defined above, \( \bar{N}_5 \) through \( \bar{N}_8 \)) is simply set to zero and used as such in the following constraint equations. Since the shape functions corresponding to vertex nodes (\( \tilde{N}_1 \) through \( \tilde{N}_4 \)) evaluate to \( \frac{1}{2} \) at the midpoint of each edge connected to the node to which the shape function corresponds, we modify the shape functions for the vertex nodes by subtracting \( \frac{1}{2} \) of the shape functions associated with the edge nodes, giving us the modified shape functions

\[ N_1 = \tilde{N}_1 - \frac{1}{2}(\bar{N}_5 + \bar{N}_6) \]  
\[ N_2 = \tilde{N}_2 - \frac{1}{2}(\bar{N}_5 + \bar{N}_6) \]  
\[ N_3 = \tilde{N}_3 - \frac{1}{2}(\bar{N}_7 + \bar{N}_8) \]  
\[ N_4 = \tilde{N}_4 - \frac{1}{2}(\bar{N}_7 + \bar{N}_8) \]
Incorporation of Gupta Element in the Dynamic Mesh Modification Scheme

Since the Gupta element resolves the deficiencies of the adaptive procedure introduced by Devloo, et. al. [15], we have merged key concepts from Gupta and Devloo to construct an alternative scheme. In addition, as described in Chapter 3, we have developed a new three-dimensional finite element based on the Gupta element, which will allow us to extend our methodology to three-dimensional problems. At this point, we briefly describe the key elements of our adaptive refinement/unrefinement procedures. We emphasize that the procedure is much like that of Devloo, et. al. [15], our contribution being the incorporation of a Gupta element to eliminate the need for constraining the solution at transition nodes, and to provide greater abstraction facilities in the finite element computations.

The description of the adaptive mesh procedures begins with a discussion of conventions and abstract data structures. Algorithms for refinement and unrefinement are then presented, for illustrative purposes. The actual implementations are more complicated, as they must handle exceptions and special cases. For example, elements which lie on the boundary of the problem domain must consider the fact that at least one edge will not have adjacent elements. Also, it is important to control the degree of refinement, as each level of refinement markedly increases the number of elements - the procedures must ensure that refinement does not use up all available computer memory.

An element in an adaptive mesh will contain, in addition to other data, the following information:

- **Node/List**: List of global node numbers (absent nodes are designated with a zero).
Figure 2.7: Element nodes and connections.

- **Level**: Level of refinement - original mesh elements are at level zero.

- **Connections**: List of adjacent element numbers.

The Node List and Connections data structures hold entries in a standard order to insure that procedures may easily determine pertinent information concerning the element and its surroundings. Figure 2.7 depicts the local numbering scheme for nodes and connections within an element, while Figure 2.8 illustrates the global node (uncircled numbers) and element (circled numbers) numbering schemes on a mesh of seven elements. Table 2.1 shows the data structures for elements 2 and 3 of the mesh shown in Figure 2.8.

We also maintain a quad-tree data structure of element groups which enable us to unrefine groups of previously refined elements and return to the
Figure 2.8: Element nodes and connections (example).

Table 2.1: Element data structures associated with example.

<table>
<thead>
<tr>
<th></th>
<th>Element 2</th>
<th>Element 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node List</td>
<td>1 5 9 0 1 3 0 0</td>
<td>1 3 1 4 1 2 9 0 0 0 0</td>
</tr>
<tr>
<td>Level</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Connections</td>
<td>0 5 1 0 0 3 1 0</td>
<td>5 7 4 2 5 7 4 2</td>
</tr>
</tbody>
</table>
state which existed before refinement. An entry in the group data structure consists of four integers indicating an element number (which, in order to differentiate from groups, is always assumed positive) or the number of a lower child (which is negative). For example, in the series of refinements shown in Figure 2.9, after the first refinement, Group 1 will have entries of 1, 2, 3, 4, pointing to each of the elements that make up the group. After the second refinement, Group 1 will have entries of 1, 2, -2, 4, where -2 points to Group 2. Group 2 will have entries of 3, 5, 6, 7.
Since the process of refinement and unrefinement may result in a discontinuous sequence of node and element numbers (which would make the finite element computations quite difficult), we also maintain lists of free nodes, groups, and elements so that at any time during the mesh modification we will know whether a node, element, or group is being used or not. This will also provide us with a means of controlling the amount of refinement - when there are no more free nodes, refinement terminates.

The basic algorithm for element refinement is as follows

\[
\text{PROCEDURE refine(elmt)}
\]

\[
\{ \text{Insure that potential 1-irregularity violations are avoided.} \}
\]

\[
\text{FOR edge = 1 TO 4 DO}
\]

\[
\text{IF (level(elmt) - level(connections(edge))} \geq 1 \text{ THEN}
\]

\[
\text{refine(connections(edge))}
\]

\[
\text{ENDIF}
\]

\[
\text{ENDFOR}
\]

\[
\{ \text{Once we are here, we can refine the original element} \}
\]

\[
\{ \text{Retrieve a new group and add to the parent group of this element} \}
\]

\[
\{ \text{pos is the position this element currently holds in the parent group.} \}
\]

\[
\text{parent.group(pos) = - retrieve.new.group()}
\]

\[
\{ \text{Generate new nodes} \}
\]

\[
\text{FOR i = 5 TO 8 DO}
\]

\[
\text{IF (nodeList(i) == 0) THEN}
\]

\[
\text{nodeList(i) = retrieve.free.node()}
\]

\[
\{ \text{Add this node to adjacent element node_lists.} \}
\]

\[
\text{ENDIF}
\]

\[
\text{ENDFOR}
\]
{Generate new middle node}
node_list(9) = retrieve_free_node()

{Generate new elements - only three are needed since
the original element will constitute the fourth}
new_group(i) = elmt
FOR i = 2 TO 4 DO
    new_group(i) = retrieve_new_elmt()
ENDFOR

{Assign new nodes, new connections, etc. to
new elements}

{Modify connections of elements adjacent to the
original element}
END refine

To unrefine a group of elements, after insuring that 1-irregularity will be
maintained, we simply climb the Groups quad-tree data structure and reverse
the refinement procedure which previously occurred. The basic algorithm for
this is

PROCEDURE unrefine(group)
{Insure that all members of the group are elements
since we cannot unrefine a group which points to
other groups}
ok_to_unrefine = TRUE
FOR i = 1 TO 4 DO
    IF (group(i) < 0) THEN
        {Member is a group}
        ok_to_unrefine = FALSE
    ENDIF
ENDFOR

IF (ok_to_unrefine) THEN
\{Insure that 1-irregularity will be maintained\}
\text{FOR all elements adjacent to group DO}
\quad \text{IF ((adj_elmt.level - group.level) } \geq 1) \text{ THEN}
\quad \quad \text{ok_to_unrefine = FALSE}
\quad \text{ENDIF}
\text{ENDFOR}
\text{ENDIF}

\text{IF (ok_to_unrefine)}
\quad \text{\{Now, we may unrefine the group\}}
\text{FOR i = 2 TO 4 DO}
\quad \quad \text{return_free_elmt(group(i))}
\text{ENDFOR}

\text{\{Modify connections and nodes for this unrefined element\}}
\text{return_free_node(node_list(9)) \{This was center node\}}
\text{FOR i = 1 TO 4 DO}
\quad \quad \text{IF (connections(i) == connections(i+4)) THEN}
\quad \quad \quad \text{\{There is only one element adjacent to edge i\}}
\quad \quad \quad \text{return_free_node(node_list(i+1))}
\quad \quad \quad \text{\{Modify node list of the adjacent element\}}
\quad \text{ENDIF}
\text{ENDFOR}

\text{\{Find the parent group of this group and change the entry which points to this group back to the original element number\}}
\text{return_free_group(group)}
\text{ENDIF}
\text{END unrefine}
Upon completion of the unrefinement procedure, the sequence of node, element and group numbers may be discontinuous, since returning the numbers to the free lists can create a “hole” in the list of used items. Thus, to simplify the computations in the finite element code, it will be necessary to renumber the items. Although not currently implemented, this would also be an opportune time to take advantage of a bandwidth reduction algorithm, renumbering nodes and elements to attain as small a bandwidth as possible in the global system of equations in order to reduce memory and time requirements in their solution.

Having provided a basic overview of the refinement and unrefinement procedures, we conclude the discussion by outlining a typical cycle in the mesh modification procedures:

- Decide which elements/groups meet refinement/unrefinement criteria (e.g. solution gradient exceeds a specified threshold).

- Refine all elements which meet the refinement criteria. This may require successive refinements of a region.

- Unrefine all groups which meet unrefinement criteria and while retaining 1-irregularity of the mesh.

- Renumber nodes, elements, and groups to insure a continuous sequence of numbers.

This procedure is currently executed at the beginning of every time step. However, this is not necessary for many problems, and less frequent mesh modifications may save time while still producing acceptable results. Following mesh modification, the finite element solution procedure may be carried out on the
resultant mesh. A key issue in this implementation is that the mesh modification and finite element solution activities are carried out independently of each other. This independence allows us to make substantial modifications to either procedure without affecting the other.

2.3 Simulation of Model Problems

Initial testing of the model, which was performed on static meshes, yielded oscillatory and unrealistic solutions for most problems. However, with the incorporation of adaptive mesh techniques, whereby mesh refinement occurred only in localized regions, solutions improved substantially. We describe here two problems which were simulated with the newly constructed model.

2.3.1 Simulating Hypothetical Problems

In an attempt to verify that the model would produce "believable" results, a simple two-phase flow problem was constructed, simulating the production of fluids from one end of a 1\text{cm} \times 0.1\text{cm} \times 0.1\text{cm} strip, while the other end was held at constant pressure and water saturation. The expected outcome of such an experiment is that a fluid front will move from the injection end to the production end of the model as the simulation progresses. Physical parameters for the model are given in Table 2.2.

The outcome of the simulation is depicted graphically. Figure 2.10 shows the configuration of the finite element mesh at three different time steps while Figures 2.11 - 2.13 portray a surface profile of the oil saturation at the three time steps. The finite element mesh can be seen to undergo refinement in the vicinity of the fluid front, and undergoes a gradual unrefinement behind the front, which is the type of behavior we expect. However, minor oscillations are
Table 2.2: Properties of hypothetical injection-production simulation.

<table>
<thead>
<tr>
<th>Physical Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho_w = 0.904$ g/cm$^3$</td>
</tr>
<tr>
<td>$\rho_n = 0.756$ g/cm$^3$</td>
</tr>
<tr>
<td>$\mu_w = 1.02$ centipoise</td>
</tr>
<tr>
<td>$\mu_n = 1.02$ centipoise</td>
</tr>
<tr>
<td>$\phi = 1.0$</td>
</tr>
<tr>
<td>$K = 12.0$ Darcies (Isotropic)</td>
</tr>
<tr>
<td>$k_{rw} = S_n$</td>
</tr>
<tr>
<td>$k_{rw} = 1 - S_n$</td>
</tr>
<tr>
<td>$S_{nirr} = 0.30$ (Irreducible sats.)</td>
</tr>
<tr>
<td>$S_{wirr} = 0.30$</td>
</tr>
<tr>
<td>$g = 0.0$ cm/s$^2$</td>
</tr>
<tr>
<td>$q = 2.0 \times 10^{-3}$ cm$^3$/sec (Production rate)</td>
</tr>
<tr>
<td>$p_c = 0.0$ atm (Capillary pressure)</td>
</tr>
<tr>
<td>$p_b = 1.0$ atm (Bottom pressure)</td>
</tr>
</tbody>
</table>

present behind the front, and to some degree, these oscillations are present in almost all simulations.

2.3.2 Simulating Laboratory Experiments

Initial tests of the model's accuracy were performed by simulating the experiments performed on a laboratory model designed to study water coning near a horizontal well in a bottom-drive reservoir [27]. The physical model, packed with a porous media, had dimensions of $50\text{cm} \times 35\text{cm} \times 0.9\text{cm}$ (see figure 2.14). Water and oil analogs filled the pore volume, and a constant pressure was maintained at the bottom, simulating a bottom water-drive reservoir. The physical model tests commenced with the opening of a valve at the “well”, and a constant flow rate was maintained through the valve. As the oil analog was removed from the upper regions of the system, the water analog formed a cone. Plexiglass sides on the model permitted viewing of the water-oil interface, which was recorded by tracing its outline at different times (see Figures 2.16,
Figure 2.10: Dynamic mesh at $t = 1.0$, $t = 2.0$, and $t = 3.0$ seconds.

Figure 2.11: Saturation profile at $t = 1.0$ sec.
Figure 2.12: Saturation profile at $t = 2.0$ sec.

Figure 2.13: Saturation profile at $t = 3.0$ sec.
Figure 2.14: Physical model.
2.23 and 2.30). These traces were used to compare experimental results with computer results.

Due to the small size of the physical model and a desire to maintain a sharp water-oil interface, miscible fluids were used. The use of the immiscible fluid equations (eqs. (2.8) and (2.9)) has been justified by Lantz [34], under the assumptions that the fluids are incompressible ($c_\text{w} = c_\text{n} = 0$), relative permeabilities of fluid $\ell$ are equal to the saturation (or concentration in miscible flow) of fluid $\ell$ and $p_c$ (see eq. (2.6)) is replaced by

$$p_c = -\frac{D\phi\mu}{K} \ln \left( \frac{S_\text{n}}{1 - S_\text{n}} \right) + (\rho_\text{w} - \rho_\text{n})gz$$

(2.58)

where $D$ is the diffusion-dispersion tensor and $\mu$ is the viscosity of the two fluids in the mixing region. Fluid and media properties used are given in Table 2.3.

Three of Hebert's tests were simulated, all of them differing only in the rate of fluid withdrawal from the experimental apparatus. Each test began with the initial mesh shown in Figure 2.15, and was run past the point of water breakthrough at the production well. For each test run simulated, we provide graphics depicting the results at two different times during the simulation. In each simulation, the second set of results depicts the solution as water breakthrough was occurring. Each set of results includes a plot of the finite element mesh, whose region of greater refinement depicts the location of the front. In addition, for each set of results we mapped the solutions at the nodes to a uniform grid (by interpolating with the element shape functions) in order to utilize software which would provide contour plots and surface profiles of the oil saturation. These results are displayed in Figures 2.17 - 2.22 for test run # 1, 2.24 - 2.29 for test run # 2, and 2.31 - 2.36 for test run # 3.
Figure 2.15: Initial finite element mesh for all test runs.
Table 2.3: Properties of Hebert test runs.

<table>
<thead>
<tr>
<th>Physical Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho_w = 0.904 \text{ g/cm}^3 )</td>
</tr>
<tr>
<td>( \rho_n = 0.756 \text{ g/cm}^3 )</td>
</tr>
<tr>
<td>( \mu_w = 1.02 \text{ centipoise} )</td>
</tr>
<tr>
<td>( \mu_n = 1.02 \text{ centipoise} )</td>
</tr>
<tr>
<td>( \phi = 0.28 )</td>
</tr>
<tr>
<td>( K = 12.0 \text{ Darcies (Isotropic)} )</td>
</tr>
<tr>
<td>( D = 1.0 \times 10^{-8} \text{ cm}^2/\text{s (Isotropic)} )</td>
</tr>
<tr>
<td>( k_{r_n} = S_n )</td>
</tr>
<tr>
<td>( k_{r_w} = 1 - S_n )</td>
</tr>
<tr>
<td>( S_{niir} = 0.0 ) (Irreducible sats.)</td>
</tr>
<tr>
<td>( S_{wiir} = 0.0 )</td>
</tr>
<tr>
<td>( g = \frac{980.665 \text{ cm/s}^2}{1.0133 \times 10^6 \text{ dynes/(cm}^2\text{.atm)}} )</td>
</tr>
<tr>
<td>( z_0 = 8.89 \text{ cm (Depth of initial interface)} )</td>
</tr>
<tr>
<td>( q_1 = 8.838 \times 10^{-2} \text{ cm}^3/\text{sec (Run # 1)} )</td>
</tr>
<tr>
<td>( q_2 = 4.788 \times 10^{-2} \text{ cm}^3/\text{sec (Run # 2)} )</td>
</tr>
<tr>
<td>( q_3 = 2.793 \times 10^{-2} \text{ cm}^3/\text{sec (Run # 3)} )</td>
</tr>
<tr>
<td>( p_b = 1.1102 \text{ atm (Bottom pressure)} )</td>
</tr>
</tbody>
</table>

Analysis of the output data shows that the position of the computed fluid front agrees reasonably well with the cone front traces, particularly in test run #1, and the saturation profiles reveal only minor oscillations in the solution. In general, more accurate results are obtained in the problems experiencing larger pressure gradients. As the flowrate lessens, we begin to experience "noisy" solutions, as evidenced by the wider front in test run #3. We believe this provides convincing evidence that our model is successful at reproducing the physical processes dominated by large pressure gradients, but inadequately models problems in which the effects of gravity, diffusion, and capillary pressure play a larger role. The likely source of "corruption" in the problems with slower flow is an accumulation of errors over successive time-steps, in the absence of
over-powering pressure gradients. Work is currently underway to further analyze the smaller forces of gravity, diffusion, and capillary pressure to insure they are being modelled correctly. Also, since the effects of these smaller forces are more localized, it will be necessary to further refine the mesh in such regions to fully capture their behavior.

Further work in this area will be directed at replicating the post-breakthrough behavior of Hebert's model. Currently, we begin to experience numerous oscillations once the fluid front reaches the well. Considering the crude modelling of the well (currently represented by a single node), this is not a surprising outcome. In addition to increased refinement at the well, it may be necessary to more rigorously handle the flux terms discussed in Section 2.2, some of which we assumed were small enough to “throw away.” Finally, it may be necessary to further explore the small-scale processes which influence the behavior of a fluid front at a well.
Figure 2.16: Actual cone front traces for Hebert run #1 (time in minutes). Reprinted with permission of author [27].
Figure 2.17: Computed saturation contours at $t = 2.1$ min. for Hebert run # 1.

Figure 2.18: Computed saturation profile at $t = 2.1$ min. for Hebert run # 1.
Figure 2.19: Finite element mesh at $t = 2.1$ min. for Hebert run # 1.
Figure 2.20: Computed saturation contours at $t = 3.8$ min. for Hebert run # 1.

Figure 2.21: Computed saturation profile at $t = 3.8$ min. for Hebert run # 1.
Figure 2.22: Finite element mesh at $t = 3.8$ min. for Hebert run # 1.
Figure 2.23: Actual cone front traces for Hebert run # 2 (time in minutes). Reprinted with permission of author [27].
Figure 2.24: Computed saturation contours at $t = 5.1$ min. for Hebert run # 2.

Figure 2.25: Computed saturation profile at $t = 5.1$ min. for Hebert run # 2.
Figure 2.26: Finite element mesh at $t = 5.1$ min. for Hebert run # 2.
Figure 2.27: Computed saturation contours at $t = 7.6$ min. for Hebert run # 2.

Figure 2.28: Computed saturation profile at $t = 7.6$ min. for Hebert run # 2.
Figure 2.29: Finite element mesh at $t = 7.6$ min. for Hebert run # 2.
Figure 2.30: Actual cone front traces for Hebert run # 3 (time in minutes). Reprinted with permission of author [27].
Figure 2.31: Computed saturation contours at $t = 10.3$ min. for Hebert run # 3.

Figure 2.32: Computed saturation profile at $t = 10.3$ min. for Hebert run # 3.
Figure 2.33: Finite element mesh at $t = 10.3$ min. for Hebert run # 3.
Figure 2.34: Computed saturation contours at $t = 16.0$ min. for Hebert run # 3.

Figure 2.35: Computed saturation profile at $t = 16.0$ min. for Hebert run # 3.
Figure 2.36: Finite element mesh at $t = 16.0$ min. for Hebert run # 3.
Chapter 3

A New Three Dimensional Finite Element for Use in 1-Irregular Meshes

3.1 Introduction

Finite element methods (FEM) have become exceptionally useful for solving problems in which the unknown function varies sharply in localized regions [5]. FEM meshes are constructed so that smaller elements (h-refinement) or elements with higher order shape functions (p-refinement) are placed in regions of high activity while larger elements or elements with lower order shape functions are placed in regions where the unknown function varies little. Recently, adaptive refinement techniques have been developed, automating the process of refining the mesh where it is needed most, and in transient problems, allowing mesh refinement to occur in different regions as the solution evolves through the problem domain [39, 18]. Devloo, et. al. [15] provide a straight-forward algorithm for the adaptive solution of two dimensional problems, using h-refinement on 1-irregular meshes (an element in a 2D 1-irregular mesh will have either one or two elements adjacent to each side [13] - see Fig. 3.1).

The use of 1-irregular meshes generally results in a transition such as that displayed in Figure 3.2, where the three nodes along the bottom of element (e) define quadratic shape functions, whereas the two nodes along the top
Figure 3.1: 1-irregular 2-D adaptive mesh.

Figure 3.2: 1-irregular transition.
of each of elements (f) and (g) define linear shape functions, violating the requirement for interelement continuity. Many implementations, including that of Devloo, et. al. [15, 13], force interelement continuity by modifying the appropriate shape functions so that, in effect, the solution at the transition node (node 2 in figure 3.2) is constrained to be the average of the solution at the edge nodes (nodes 1 and 3). Although this has the computational advantage of reducing the number of nodes in the solution, it smoothes the solution along the transition boundaries, and removes degrees of freedom that may be needed for higher accuracy.

Gupta [26, 31] introduces a two dimensional transition element which does not rely on constraints to produce interelement continuity. Rather, he modifies the shape functions of the transition element to produce piecewise linear solutions along the interelement boundary, modifying the quadrature integration scheme to account for discontinuities in the shape function derivatives. An advantage of the Gupta element is the preservation of the piecewise linear solution on the interelement boundary, but at the expense of using more degrees of freedom than the constraint method would entail.

We have constructed finite element software for the modelling of two-dimensional multi-phase flow through porous media, using an adaptive refinement scheme based on that of Devloo, et. al. [15], and taking advantage of the Gupta element rather than the constraint method to assure interelement continuity (see Chapter 2). In our attempt to extend this work to three-dimensional domains, we have found no mention in the literature of a 3D transition element analogous to the 2D Gupta element. Thus, in this chapter we present shape
functions for a 3D transition element based on the 2D Gupta element, suitable for use in adaptive mesh schemes based on \( h \)-refinement of simple "brick" elements (see Fig. 3.3).

We first derive the shape functions for a generic "parent" element, then prove that the interelement continuity requirement will be satisfied when this element is used as a transition element in 1-irregular meshes. Finally, we will describe a simple test that we performed, illustrating the usefulness of this element in a locally refined mesh.

### 3.2 Construction of Shape Functions

We will construct a set of shape functions which interpolate correctly in the element interior and whose behavior on a boundary with refined elements is equivalent to the behavior of the combined shape functions on the interelement boundary of the four refined elements (see Fig. 3.4). These shape functions will be discontinuous out of necessity, but a modified quadrature scheme described by Gupta [26] will insure that integrations are performed in a piecewise manner.
over the element, with the end result being a valid integration over the entire element.

In the following, we define all shape functions on a parent element in a standard isoparametric coordinate system (shape functions defined for $-1 \leq \xi, \eta, \zeta \leq 1$) with the origin located at the centroid of the element and the vertical coordinate $\zeta$ positive in the downward direction (we choose $\zeta$ to be positive downward simply because our applications use depth as the vertical parameter - the shape functions are easily modified to accommodate $\zeta$ being positive in the upward direction).

The shape functions for the standard three-dimensional tri-linear brick element are given below [29]. The introduction of additional nodes will require us to modify these shape functions, so we denote them with a bar (e.g. $\bar{N}_i$). Shape functions in their final form will be denoted with no bar (e.g. $N_i$). Figure 3.6 illustrates $\bar{N}_7(\xi, \eta, -1)$ on the top face of the element.

Figure 3.4: Interelement boundary of 1-irregular transition.
Figure 3.5: 3-D tri-linear brick element.

\[
\begin{align*}
\bar{N}_1(\xi, \eta, \zeta) &= \frac{1}{8}(1 - \xi)(1 - \eta)(1 + \zeta) \\
\bar{N}_2(\xi, \eta, \zeta) &= \frac{1}{8}(1 + \xi)(1 - \eta)(1 + \zeta) \\
\bar{N}_3(\xi, \eta, \zeta) &= \frac{1}{8}(1 + \xi)(1 + \eta)(1 + \zeta) \\
\bar{N}_4(\xi, \eta, \zeta) &= \frac{1}{8}(1 - \xi)(1 + \eta)(1 + \zeta) \\
\bar{N}_5(\xi, \eta, \zeta) &= \frac{1}{8}(1 - \xi)(1 - \eta)(1 - \zeta) \\
\bar{N}_6(\xi, \eta, \zeta) &= \frac{1}{8}(1 + \xi)(1 - \eta)(1 - \zeta) \\
\bar{N}_7(\xi, \eta, \zeta) &= \frac{1}{8}(1 + \xi)(1 + \eta)(1 - \zeta) \\
\bar{N}_8(\xi, \eta, \zeta) &= \frac{1}{8}(1 - \xi)(1 + \eta)(1 - \zeta)
\end{align*}
\] (3.1)

Shape functions for brick elements with nodes added only at the midpoints of an edge are given below (see Fig. 3.7).

Figure 3.8 illustrates \( \bar{N}_{15}(\xi, \eta, -1) \) on the top of the element - note that \( \bar{N}_{15} \) is equal to one at node 15 and zero at all other nodes. Again, these shape
Figure 3.6: Shape function $\tilde{N}_7(\xi, \eta, -1)$ for corner node.

functions will need to be modified to account for other nodes added to the element.

\[
\tilde{N}_9(\xi, \eta, \zeta) = \frac{1}{4}(1 - |\xi|)(1 - \eta)(1 + \zeta) \quad (3.9)
\]

\[
\tilde{N}_{10}(\xi, \eta, \zeta) = \frac{1}{4}(1 - |\xi|)(1 + \eta)(1 + \zeta) \quad (3.10)
\]

\[
\tilde{N}_{11}(\xi, \eta, \zeta) = \frac{1}{4}(1 - |\xi|)(1 + \eta)(1 - \zeta) \quad (3.11)
\]

\[
\tilde{N}_{12}(\xi, \eta, \zeta) = \frac{1}{4}(1 - \xi)(1 - |\eta|)(1 + \zeta) \quad (3.12)
\]

\[
\tilde{N}_{13}(\xi, \eta, \zeta) = \frac{1}{4}(1 + \xi)(1 - |\eta|)(1 + \zeta) \quad (3.13)
\]

\[
\tilde{N}_{14}(\xi, \eta, \zeta) = \frac{1}{4}(1 - \xi)(1 - |\eta|)(1 + \zeta) \quad (3.14)
\]

\[
\tilde{N}_{15}(\xi, \eta, \zeta) = \frac{1}{4}(1 + \xi)(1 - |\eta|)(1 - \zeta) \quad (3.15)
\]

\[
\tilde{N}_{16}(\xi, \eta, \zeta) = \frac{1}{4}(1 - \xi)(1 - |\eta|)(1 - |\zeta|) \quad (3.16)
\]

\[
\tilde{N}_{17}(\xi, \eta, \zeta) = \frac{1}{4}(1 + \xi)(1 - \eta)(1 - |\zeta|) \quad (3.17)
\]

\[
\tilde{N}_{18}(\xi, \eta, \zeta) = \frac{1}{4}(1 + \xi)(1 - \eta)(1 - |\zeta|) \quad (3.18)
\]
Shape functions for elements with nodes at the center of the faces are introduced below. Figure 3.10 illustrates $\tilde{N}_{25}(\xi, \eta, -1)$ on the top of the element. Note that each of the shape functions vanishes at the edges and will not need to be modified to account for the nodes introduced above.

\[
\begin{align*}
N_{21}(\xi, \eta, \zeta) &= \frac{1}{4}(1 - |\xi|)(1 - \eta)(1 - |\zeta|) \\
N_{22}(\xi, \eta, \zeta) &= \frac{1}{4}(1 - |\xi|)(1 + \eta)(1 - |\zeta|) \\
N_{23}(\xi, \eta, \zeta) &= \frac{1}{4}(1 - |\xi|)(1 + \eta)(1 - |\zeta|) \\
N_{24}(\xi, \eta, \zeta) &= \frac{1}{4}(1 + |\xi|)(1 - |\eta|)(1 - |\zeta|) \\
N_{25}(\xi, \eta, \zeta) &= \frac{1}{4}(1 - |\xi|)(1 - |\eta|)(1 - |\zeta|) \\
N_{26}(\xi, \eta, \zeta) &= \frac{1}{4}(1 - |\xi|)(1 - |\eta|)(1 + \zeta)
\end{align*}
\]
Figure 3.8: Shape function $\bar{N}_{15}(\xi, \eta, -1)$ for mid-edge node.

Figure 3.9: 3-D brick element face nodes.
We now begin to modify the shape functions $\tilde{N}_1, \ldots, \tilde{N}_{20}$ so that they evaluate to 1 at their defined node and 0 at all other nodes. The following derivations assume that all variations of this element have nodes 1 through 8 (the corner nodes) present. This is a logical assumption under the context of 1-irregular mesh refinements. However, nodes 9 through 26 may or may not be present in various combinations. When a node is not present, its original shape function (as defined above, either $\tilde{N}_9$ through $\tilde{N}_{20}$ or $N_{21}$ through $N_{26}$) is simply set to zero and used as such in the following constraint equations.

The shape functions corresponding to corner nodes ($\tilde{N}_1$ through $\tilde{N}_8$) evaluate to $\frac{1}{2}$ at the midpoint of each edge connected to the node to which the shape function corresponds. Thus, we modify the shape functions for the corner nodes by subtracting $\frac{1}{2}$ of the shape functions associated with the edge nodes. The resultant shape function evaluates to $-\frac{1}{4}$ at the face nodes, so we modify this function by adding $\frac{1}{4}$ of the shape functions associated with the
Figure 3.11: Modified shape function $N_7(\xi, \eta, -1)$ for corner node.

By inspection, we determine that the shape functions for the edge mid-points evaluate to zero at the endpoints of the edge and will not have to be
modified to account for the corner nodes. However, these functions evaluate to $\frac{1}{2}$ at each of the face nodes adjacent to them, so we modify them by subtracting $\frac{1}{2}$ of the shape functions associated with the face nodes. Figure 3.12 illustrates $N_{15}(\xi, \eta, -1)$ at the top of the element.

\begin{align*}
N_9 &= \bar{N}_9 - \frac{1}{2}(N_{21} + N_{26}) \\
N_{10} &= \bar{N}_{10} - \frac{1}{2}(N_{22} + N_{25}) \\
N_{11} &= \bar{N}_{11} - \frac{1}{2}(N_{22} + N_{25}) \\
N_{12} &= \bar{N}_{12} - \frac{1}{2}(N_{21} + N_{25}) \\
N_{13} &= \bar{N}_{13} - \frac{1}{2}(N_{23} + N_{26}) \\
N_{14} &= \bar{N}_{14} - \frac{1}{2}(N_{24} + N_{26}) \\
N_{15} &= \bar{N}_{15} - \frac{1}{2}(N_{24} + N_{25}) \\
N_{16} &= \bar{N}_{16} - \frac{1}{2}(N_{23} + N_{25}) \\
N_{17} &= \bar{N}_{17} - \frac{1}{2}(N_{21} + N_{23}) \\
N_{18} &= \bar{N}_{18} - \frac{1}{2}(N_{21} + N_{24}) \\
N_{19} &= \bar{N}_{19} - \frac{1}{2}(N_{22} + N_{24}) \\
N_{20} &= \bar{N}_{20} - \frac{1}{2}(N_{22} + N_{23})
\end{align*}

(3.35) \quad \text{(3.36)} \quad \text{(3.37)} \quad \text{(3.38)} \quad \text{(3.39)} \quad \text{(3.40)} \quad \text{(3.41)} \quad \text{(3.42)} \quad \text{(3.43)} \quad \text{(3.44)} \quad \text{(3.45)} \quad \text{(3.46)}

Gauss points for Gauss-Legendre integration and derivatives of the shape functions may be found in the appendix.

### 3.3 Continuity of Shape Functions Across Interelement Boundaries

To show that continuity exists across element boundaries, it is sufficient to show that the shape functions associated with the nodes on the boundary
are equivalent in both elements. We will consider the shape functions at one such node. Since the shape functions at the other nodes are derived in a similar manner, equivalence at one node will let us conclude that equivalence exists at the other nodes.

Consider the case shown in Figure 3.13. We will show that the shape function at node 1 of element $e$ is equivalent to the shape function at node 5 of element $f$. Since the shape functions are defined on generic parent elements, they are defined on different coordinate systems. Note, for example, that node 2 of element $e$ has local coordinates $(1, -1, 1)$ while the corresponding node 12 of element $f$ has local coordinates $(0, -1, -1)$.

The transformation between the coordinate systems of elements $e$ and $f$ are

$$\xi_f = -\frac{1}{2}(1 - \xi_e), \quad \eta_f = -\frac{1}{2}(1 - \eta_e), \quad \zeta_f = -\frac{1}{2}(1 - \zeta_e)$$ (3.47)
or

\[ \xi_e = 1 + 2\xi_f, \quad \eta_e = 1 + 2\eta_f, \quad \zeta_e = 1 + 2\zeta_f \quad (3.48) \]

These transformations will be used later to establish the equivalence of the shape functions along the element boundary.

First, we look at the behavior of \( N_1 \) in element \( e \) along the interelement boundary:

\[ N_1(e)(\xi_e, \eta_e, \zeta_e) = \frac{1}{8}(1 - \xi_e)(1 - \eta_e)(1 + \zeta_e) \quad (3.49) \]

Since we are interested in behavior only at the interelement boundary, we set \( \zeta_e = 1 \), giving

\[ N_1(e) = \frac{1}{4}(1 - \xi_e)(1 - \eta_e) \quad (3.50) \]

Next, we evaluate the shape function \( N_6^{(f)} \) on the top surface of element \( f \) by looking at behavior in each of four regions (See Figure 3.14). We will show that in Region \( i \), \( N_6^{(f)} \) is equivalent to \( N_1(e) \), and in Regions \( ii, iii, \) and \( iv \), the
function vanishes. In all cases, since we are interested only in behavior at the interelement boundary, we set $\zeta_f = -1$. This lets us reduce $N_s^{(f)}$ as follows:

\[
N_s^{(f)} = \tilde{N}_s - \frac{1}{2}(\tilde{N}_{12} + \tilde{N}_{16} + \tilde{N}_{17}) + \frac{1}{4}(N_{21} + N_{23} + N_{20})
\]

\[
= \frac{1}{8}(1 - \xi_f)(1 - \eta_f)(1 - \zeta_f) - \\
\frac{1}{2}\left[\frac{1}{4}(1 - |\xi_f|)(1 - |\eta_f|)(1 - |\zeta_f|) + \right. \\
\frac{1}{4}(1 - \xi_f)(1 - |\eta_f|)(1 - \zeta_f) + \\
\frac{1}{4}(1 - \xi_f)(1 - \eta_f)(1 - |\zeta_f|) + \\
\frac{1}{4}(1 - |\xi_f|)(1 - \eta_f)(1 - \zeta_f) + \\
\frac{1}{2}(1 - |\xi_f|)(1 - |\eta_f|)(1 - |\zeta_f|) + \\
\frac{1}{2}(1 - |\xi_f|)(1 - |\eta_f|)(1 - |\zeta_f|)
\] + \\
\frac{1}{4}(1 - \xi_f)(1 - \eta_f)(1 - \zeta_f) - \\
\frac{1}{4}(1 - |\xi_f|)(1 - |\eta_f|)(1 - |\zeta_f|)
\]

\[
= \frac{1}{4}(1 - \xi_f)(1 - \eta_f) - \frac{1}{4}(1 - |\xi_f|)(1 - |\eta_f|)
\]

(3.51)
In Region \( i \), \( \xi_f \leq 0, \eta_f \leq 0 \). Equation (3.51) becomes

\[
N_6^{(f)} = \frac{1}{4}(1 + \xi_f)(1 + \eta_f) - \frac{1}{4}(1 - \xi_f)(1 + \eta_f) - \frac{1}{4}(1 + \xi_f)(1 - \eta_f) + \frac{1}{4}(1 - \xi_f)(1 - \eta_f)
\]

which reduces to

\[
N_6^{(f)} = \xi_f \eta_f
\]

Applying the coordinate transformation from element \( f \) to element \( e \) (See Equation (3.47)), we get

\[
N_6^{(f)} = \frac{1}{4}(1 - \xi_e)(1 - \eta_e)
\]

which of course is equivalent to (3.50). Thus, in Region \( i \),

\[
N_6^{(f)} \equiv N_1^{(e)}.
\]

In Region \( ii \), \( \xi_f > 0, \eta_f < 0 \). Equation (3.51) becomes

\[
N_6^{(f)} = \frac{1}{4}(1 - \xi_f)(1 + \eta_f) - \frac{1}{4}(1 - \xi_f)(1 + \eta_f) - \frac{1}{4}(1 - \xi_f)(1 - \eta_f) + \frac{1}{4}(1 - \xi_f)(1 - \eta_f)
\]

\[
= 0.
\]

In Region \( iii \), \( \xi_f > 0, \eta_f > 0 \). Equation (3.51) becomes

\[
N_6^{(f)} = \frac{1}{4}(1 - \xi_f)(1 - \eta_f) - \frac{1}{4}(1 - \xi_f)(1 - \eta_f) - \frac{1}{4}(1 - \xi_f)(1 - \eta_f) + \frac{1}{4}(1 - \xi_f)(1 - \eta_f)
\]

\[
= 0.
\]

Finally, in Region \( iv \), \( \xi_f < 0, \eta_f > 0 \). Equation (3.51) becomes

\[
N_6^{(f)} = \frac{1}{4}(1 + \xi_f)(1 - \eta_f) - \frac{1}{4}(1 - \xi_f)(1 - \eta_f) - \frac{1}{4}(1 + \xi_f)(1 - \eta_f) + \frac{1}{4}(1 - \xi_f)(1 - \eta_f)
\]

\[
= 0.
\]
Thus, we may conclude that

$$N_5^{(f)} \equiv N_1^{(e)}$$  \hspace{1cm} (3.59)

over the entire interelement boundary.

### 3.4 Demonstration of Element Performance

To demonstrate the usefulness of the transition elements, we solve Laplace's equation ($\nabla^2 u = 0$) on a cube domain of dimension ($\pi \times \pi \times \pi$) with boundary conditions

$$u(0, y, z) = u(\pi, y, z) = u(x, 0, z) = u(x, \pi, z) = u(x, y, \pi) = 0 \quad (3.60)$$
$$u(x, y, 0) = 100 \sin(2x) \sin(2y) \quad (3.61)$$

having an analytic solution of

$$u(x, y, z) = \frac{100 \sin(2x) \sin(2y) \sinh(2\sqrt{2}(\pi - z))}{\sinh(2\sqrt{2}\pi)} \quad (3.62)$$

which has an amplitude of 100 on the top face (see Figure 3.15) rapidly dampening to approximately 1 at a depth of $\frac{\pi}{2}$ (see Figure 3.16), and to 0 on the bottom face.

Since most of the activity is near the top of the problem domain, we compare the solutions on a uniform mesh of 729 nodes (Figure 3.17) with the solutions on a locally-refined mesh possessing 277 nodes (Figure 3.18). Results of these tests indicate very close agreement of the solutions on the uniform mesh and the locally-refined mesh, as illustrated in Figures 3.19 and 3.20. Thus, we are able to achieve almost the same degree of accuracy in the region of greatest activity, using less than 40% of the nodes from a uniform mesh.
Figure 3.15: \( u(x, y, 0) \).

Figure 3.16: \( u(x, y, \frac{\pi}{2}) \).
Figure 3.17: Uniform mesh of 729 nodes.

Figure 3.18: Locally-refined mesh of 277 nodes.
Figure 3.19: Solutions for $u(x, \frac{\pi}{4}, 0)$.

Figure 3.20: Solutions for $u(x, \frac{\pi}{4}, \frac{\pi}{4})$. 
3.5 Conclusion

We have introduced a new finite element suitable for use in 3D adaptive mesh refinement strategies. The use of this element facilitates the implementation of adaptive refinement algorithms based on 1-irregular meshes, while retaining the nodes and the piecewise solution in the regions of transition. Subsequent work will be focused at the extension of our 2D adaptive FEM software to 3D domains utilizing this new element, followed by applications to moving-front fluid flow problems in which adaptive refinement methods are necessary to track fluid interfaces through the problem domain. The immediate application areas will be in petroleum engineering and groundwater research, but a broad spectrum of disciplines might make beneficial use of this work.
Chapter 4

Analysis and Implementation of a Distributed Finite Element Methodology Based on Domain Decomposition

4.1 Introduction

The computer simulation of complex physical phenomena is often computationally prohibitive, requiring many hours and megabytes of resources. Meanwhile, countless workstations stand idle during evening and weekend hours, representing a tremendous waste of computational power. The distribution of data and workload over these idle workstations provides an opportunity to solve complex problems on available computing equipment. If we consider a simulation which requires forty-eight hours of computing time on a single machine, parallelize the code efficiently, and distribute the computations over four workstations, then it is possible for us to complete the simulation overnight. In addition, simulations which are too large to be handled by a single computer's memory may be distributed to allow different workstations to work on smaller portions of the original problem.

The purpose of this chapter, then, is to investigate and implement methods which will allow us to parallelize our computations for the finite element analysis of multiphase flow through porous media. Although the development platform is a cluster of RS/6000 workstations, there are plans for porting the
code to large-scale supercomputers, such as the Cray T3D. Thus, there is a need for compatibility of the code on the various platforms it may be implemented on. To insure such portability, we feel it is best to work with a distributed memory paradigm, in which each processor has direct access to its local memory and can only address other processors’ local memory by passing messages over a network. This decision allows us to consider a broad range of computers from PC’s to supercomputers as potential platforms.

Numerous tools now exist which make distributed computing accessible at low cost on a range of architectures from desktop PC’s to state-of-the-art supercomputers. Of particular interest to us for its wide range of applicability is Parallel Virtual Machine (PVM), available at no cost from Oak Ridge National Laboratory [38, 17, 24, 2]. Designed for use on a heterogeneous network of Unix based computers, PVM consists of a library of message-passing routines which may be called from C or Fortran programs. PVM allows the user to construct a virtual machine by specifying nodes on a network (which may range in size from a LAN to the worldwide Internet) which will participate in the parallel session. On each node in the virtual machine, a PVM daemon is spawned, its purpose being to control the message passing for the distributed program executed on that machine. The programming model is Multiple Program, Multiple Data (MPMD), in which independent programs execute concurrently, coordinating actions only by exchanging messages over the network. On some architectures, particularly supercomputers such as the Cray T3D [12], the programming model is restricted to Single Program, Multiple Data (SPMD). In this scenario, which is the one utilized by ourselves, identical copies of a program execute independently on different processors, operating on different
sets of data, and periodically cooperating to solve “global” problems. All of the work described in this chapter has been implemented with PVM on the RS/6000 cluster at Louisiana State University’s System Network Computer Center and is currently being ported to the Cray T3D at the University of Alaska’s Arctic Region Supercomputing Center.

This chapter will progress by discussing the domain decomposition (sometimes referred to as substructuring) method as it applies to the distribution of a general finite element code. It will be seen that this method facilitates the coarse-grained distribution of data and workload by coupling subdomains through a set of nodes lying on interprocessor boundaries. In many cases, existing code need only undergo slight modification, although additional code is needed to maintain mappings between local and global data structures. The pertinent components of the method are discussed and analyzed in terms of relative performance characteristics. The method is then discussed in a unified framework and again, analyzed in terms of relative performance under different conditions. Finally, results of the implementation of this method are discussed, comparing results with those from the previously developed serial code, and analyzing speedup characteristics.

4.2 Parallelization of a Finite Element Method Multi-phase Flow Code

4.2.1 An Overview of the Parallel Methodology

In this section we describe the methods used to construct a distributed implementation of a static-mesh version of the finite element code described in Chapter 2. Although we will later present ideas for the crude implementation of a dynamic mesh modification scheme in a distributed computing environment,
we believe this is an area of future research and will assume in this chapter that the finite element mesh remains fixed throughout the computations.

Before detailing the procedures involved in the distribution of the work, we briefly outline the overall methodology, so that the reader may be able to look at the specifics from a broader perspective. The foundation for this approach comes from Doltsinis and Nöltting [16] and similar strategies have been used successfully in numerous finite difference and finite element models [40, 37]. Figure 4.1 provides a graphical representation of the approach, depicting
Figure 4.2: Example problem domains partitioned into two element sets coupled by a set of common nodes.

the tasks carried out by two processes operating on two distinct element sets in the problem domain, which are coupled by a set of common nodes (see Figure 4.2 for example.). The dashed horizontal lines in Figure 4.1 depict points during the computation at which interprocess communication will be necessary in order to solve "global" problems. It is evident from this illustration that for most of the time, the processes perform the same set of tasks, but on their own data. Only at the beginning, when it is necessary for processes to partition the global data set, and at two points during each iteration, is it necessary for the processes to communicate.

It is also evident that this procedure is essentially the same as the general finite element procedure described in Chapter 2, with the addition of routines needed for the coupling and synchronization of the processes. After the processes initially cooperate to partition the data, they enter the time-stepping
and linearization loops. During each iteration, each process constructs elemental equations for each of its elements, adding them to a system of equations local to the process. After all elements in a process have been assembled into the local system of equations, boundary conditions are applied. Up to this point, there has been no change in the finite element procedure described in Chapter 2, other than the fact that elements are being assembled in parallel, rather than one at a time. Once the local systems of equations have been constructed and adjusted for boundary conditions, each process performs a series of algebraic manipulations to form a Schur complement, a system of equations for the unknowns which lie on the boundaries with other processes. At this point, the processes cooperate to assemble each of the individual Schur complements into a global system of equations for all of the unknowns which lie on interprocess boundaries. These equations are solved globally (we consider two methods later in this section), then the solutions for values at the interprocess boundary nodes are returned to the appropriate processes. Next, each process again works in parallel, using the solution at the local interprocess boundary nodes to solve for the unknowns at all of the process' interior nodes. Once the interior unknowns have been solved for, each process checks its own convergence status, then works cooperatively with the other processes to determine the global convergence status. Having completed the global convergence check, each process either begins another iteration, or starts a new time step. It should be emphasized that a large number of processes may execute concurrently to significantly speed up computations, the only significant bottleneck being the global solution for the unknowns at the interprocess boundary nodes. In addition, if we consider the fact that this methodology results in normally expensive
matrix operations being performed on smaller matrices, then we begin to see great potential in this approach.

As we discuss the details of the substructuring method, we will formulate expressions for the relative time complexity of each stage. A unique approach for estimating the complexities is introduced, allowing us to incorporate computational and communication costs in a single formula. Standard approaches in the literature [45, 33] typically describe parallel algorithm complexities as a function of problem size and number of processors, neglecting the often dominant overhead resulting from communication costs. In our approach, the complexity estimate includes such limiting factors as communication costs and bottlenecks due to imperfect load balancing. This provides us with expressions which may be used to analyze the advantages of increasing the number of processors for a given problem size. Although at this early point it will be difficult to make much use of these expressions, we feel they are best derived here as we introduce the operations involved in the methodology. Section 4.3 will tie these formulas together in a unified framework, allowing us to analyze speedup characteristics as a function of partitioning schemes, number of unknowns per node, and the approximate cost of sending a message from one process to another. The unit of measure used for estimating the relative complexities will be called a t\textit{flop} (this is our own convention, adopted in order to provide a common measure for computational and communication costs). We will define a t\textit{flop} to be equivalent to the time required to execute a floating point operation in double precision arithmetic. In this context, a floating point operation is as defined by Golub and Van Loan [22] to be a \textit{single} arithmetic
operation. For example, the multiply and add instruction

\[ y = a + b \cdot z \]

constitutes two tflops.

The communication costs, in order to incorporate them neatly into the analysis, will be expressed in tflops also. We will utilize a parameter, \( c_{dp} \), to represent the cost (in tflops) of passing a double precision number from one process to another. This factor should incorporate all of the overhead inherent to message passing, and will include the cost of message reception by the destination process. As an example, a distributed system which takes one hundred times longer to send a double precision number between processes than it takes to perform a floating point operation will have the communication cost parameter set as

\[ c_{dp} = 100 \text{ tflops}. \]

A simple two-process example illustrates our conventions for complexity estimation:

\begin{verbatim}
PO

FOR i=1 TO 1000 DO
  z(i) = y(i)+ a*x(i)
ENDFOR

{send z to P1}

{receive u}

FOR i=1 TO 1000 DO
  z(i) = y(i)+ u(i)
ENDFOR

P1

FOR i=1 TO 1000 DO
  z(i) = y(i) + x(i)
ENDFOR

{send z to P0}

{receive u}

FOR i=1 TO 1000 DO
  z(i) = y(i) + u(i)
ENDFOR
\end{verbatim}
In this example, P0 and P1 calculate entries for the vector $z$, but P0 does so with an addition and multiplication for each entry of $z$, while P1 only performs an addition for each entry. Thus, it takes P0 2000 tflops to fill $z$ while P1 requires only 1000 tflops. It is important that we always use the largest value as an indication of complexity, since the process with the most work will cause other processes to wait for it at some point. Thus, we consider the complexity of the algorithm at this point to be 2000 tflops. Each process then sends $z$ to the other process and waits for the arrival of the other process' message, which it will store in $u$. Since each process sends and receives 1000 numbers (which we assume are double precision), the communication cost of this step is $1000 \cdot c_{dp}$. Finally, each process calculates new entries for $z$ using a single floating point operation for each entry, for a total of 1000 tflops each. The cost of the complete algorithm then, is

$$3000 + 1000 \cdot c_{dp} \text{ tflops}.$$ 

We emphasize that our methods of estimating computational complexity are crude at best. The actual cost of passing messages over a network is complicated by such unpredictable factors as contention for a limited bandwidth bus. Even an estimate of the time required to perform arithmetic operations is difficult to obtain. Factors such as location of data in memory can profoundly affect the speed with which floating point operations are performed. Thus, these methods are meant only as a gross estimate for assessing the potential benefits of various distribution schemes.

### 4.2.2 The Substructuring Approach

We now describe the details of the substructuring method as a series of six steps. Step 1 is performed at the beginning of the program to partition
and send the mesh data to the appropriate processes, then construct mappings which will enable processes to cooperate in the computations which are solved globally. Although the current implementation is on a static mesh, the use of dynamic meshes would require that this step be performed everytime the mesh is modified. Steps 2 – 6 constitute the basic finite element assembly and solution procedures, and are executed during every iteration of the linearization loop.

**Step 1 – Data Setup**

In this step, we take the global mesh data and partition it for solution on a distributed set of processes. The problem domain is partitioned in a manner which insures that each element belongs to exactly one process, and interprocess boundaries follow the edges of elements. Thus, each process will contain a unique set of elements and all of the nodes which belong to its elements. Nodes lying on the interprocess boundaries will have identical copies in each process which has an element containing them. Note that this partitioning strategy will work for any finite element mesh, containing any mixture of element types. In the current implementation, the user specifies which elements belong to which process and which nodes lie on interprocess boundaries. Thus, it is currently the user’s responsibility to establish a partitioning to achieve a good load balance. Much of this can be automated on simple meshes such as those we have been using.

Once the data has been partitioned and sent to the appropriate processes, each process constructs a local element and node numbering scheme, insuring that nodes which lie in the interior of the process are numbered before the nodes lying on interprocess boundaries. The reasoning for this will be clear
when we discuss the following steps. Thus, the nodes are conveniently classified as interior nodes or boundary nodes. This step concludes with the construction of data structures which map the local and element numbers to the original global values, in addition to other mappings which are required to perform the global solution phase discussed in Step 4.

**Step 2 – Assembly of Local Systems of Equations**

In embarrassingly parallel fashion, each process begins a linearization iteration by constructing equations for each of its elements and assembling them into a local system of equations

\[
\begin{bmatrix} A_{ii} & A_{ib} \\ A_{bi} & A_{bb} \end{bmatrix} \begin{bmatrix} u_i \\ u_b \end{bmatrix} = \begin{bmatrix} f_i \\ f_b \end{bmatrix},
\]

where

\[
A_{ii} \in \mathbb{R}^{n_i \times n_i}, \quad A_{ib} \in \mathbb{R}^{n_i \times n_b}, \quad u_i, f_i \in \mathbb{R}^{n_i \times 1},
\]

\[
A_{bi} \in \mathbb{R}^{n_b \times n_i}, \quad A_{bb} \in \mathbb{R}^{n_b \times n_b}, \quad u_b, f_b \in \mathbb{R}^{n_b \times 1},
\]

\(n_i\) is the number of unknowns at the interior nodes of the process, and \(n_b\) is the number of unknowns on the interprocess boundary nodes. If necessary, the equations are then modified to account for boundary conditions applied to any nodes in the process.

An estimate of the time complexity of this step within a process is derived as a function of \(n\), the number of nodes in the process, and \(d_f\), the number of unknowns per node. Although numerous operations are performed in the assembly of the local system of equations, each element is processed exactly once, and the number of operations for each element is bounded by a constant, which includes such factors as the maximum number of nodes an element can have, and the accuracy we wish to achieve in the numerical integration of the weighted residual terms. Thus, the number of operations will be directly
proportional to the number of unknowns in the process and we describe the time complexity for this step as

$$T_{assemble}(n, df) = n \cdot df.$$  \hspace{2cm} (4.2)

We are admittedly treating this formula rather loosely, but we will see that the time complexities of the following stages are much greater than this, and that the operations in this step play a minor role in the total time required for the entire method.

**Step 3 - Formation of Schur Complement**

With this step, we transform the system of equations (Eq. (4.1)) to a form which will express the solution for the unknowns at the interior nodes as a function of the solution at the interprocess boundary nodes. We express (4.1) as

$$A_{ii}u_i + A_{ib}u_b = f_i \hspace{2cm} (4.3)$$

$$A_{bi}u_i + A_{bb}u_b = f_b. \hspace{2cm} (4.4)$$

Solving (4.3) for $u_i$ gives us

$$u_i = A_{ii}^{-1}(f_i - A_{ib}u_b), \hspace{2cm} (4.5)$$

which we substitute into (4.4) and rearrange to obtain

$$\left( A_{bb} - A_{bi}A_{ii}^{-1}A_{ib} \right)u_b = f_b - A_{bi}A_{ii}^{-1}f_i. \hspace{2cm} (4.6)$$

Equation (4.6) may be expressed as

$$A^*u_b = b^*, \hspace{2cm} (4.7)$$
where \( A^* \) is often referred to as the Schur complement or Gauss transform [37].

Upon solution of (4.7) (which all processes must cooperate in - see Step 4), \( u_b \) may be substituted into (4.5) to obtain the solution for all of the interior unknowns.

An estimate of the process time complexity as a function of \( n_i \), the number of interior nodes, \( n_b \), the number of interprocess boundary nodes, and \( d_j \), degrees of freedom per node, is easily obtained by analyzing the matrix operations required for the construction of the Schur complement (and the right hand side), which is performed with the following sequence of matrix operations:

1. Compute \( A_{ii}^{-1} \). Using standard matrix inversion routines, this is known to require \( 2n^3 \) operations for an \( n \times n \) matrix [44]. Thus, we expect this procedure to require \( 2(d_j n_i)^3 \) tflops.

2. \( \text{templhs} \leftarrow A_{ii}^{-1} A_{ib} \). Multiplication of a \( d_j n_i \times d_j n_i \) and \( d_j n_i \times d_j n_b \) matrix will require \( 2d_j^2 n_i^2 n_b \) tflops.

3. \( A^* \leftarrow A_{ii}^{-1} \cdot \text{templhs} \). Multiplication of a \( d_j n_b \times d_j n_i \) and \( d_j n_i \times d_j n_b \) matrix will require \( 2d_j^2 n_i n_b^2 \) tflops.

4. \( A^* \leftarrow A_{ib} - A^* \). This \( d_j n_b \times d_j n_b \) matrix subtraction requires \( 2(d_j n_b)^2 \) tflops.

5. \( \text{temprhs} \leftarrow A_{ii}^{-1} f_i \). Multiplication of a \( d_j n_i \times d_j n_i \) matrix and a \( d_j n_i \times 1 \) vector will require \( 2(d_j n_i)^2 \) tflops.

6. \( f^* \leftarrow A_{ib} \cdot \text{temprhs} \). Multiplication of a \( d_j n_b \times d_j n_i \) matrix and a \( d_j n_i \times 1 \) vector will require \( 2d_j^2 n_i n_b \) tflops.

7. \( f^* \leftarrow f_b - f^* \). This \( d_j n_b \times 1 \) vector subtraction requires \( 2d_j n_b \) tflops.
Totalling the complexities gives us an estimated time complexity of

\[
T_{\text{schur}}(n_i, n_b, d_f) = 2\left( (d_f n_i)^3 + d_f^2 n_i^2 n_b + d_f n_i n_b^2 + d_f n_i n_b + d_f^2 n_i n_b + d_f n_b \right)
\]

within a process.

\textit{Step 4 – Assembly and Solution of Global Boundary Node System}

Up to now, all operations (except for the initial partitioning) have been performed with perfect parallelism. At this point, however, it is necessary to assemble each process’ Schur complement and right hand side into a global boundary node system, solve for all the unknowns on interprocess boundaries, then return solutions to the appropriate processes. In this implementation, we choose to assemble the global boundary node system in a single process. Solution of this system can then occur either serially in the same process, or the system matrix can be distributed and solved in parallel, with the solution being returned to the original process. In Section 4.2.3 we analyze both methods in detail. No matter which method is used, once a solution has been calculated, the process which assembled the system will return the solution to the appropriate processes.

At this point, we analyze the cost of sending each local boundary node system to the single process, assembling into the global boundary node system, and returning the solution to the processes. We leave analysis of the actual solution for Section 4.2.3.

To estimate the cost of sending each local boundary node system to a single process and assembling it, we consider it from the point of view of the
single process doing the assembling. This process acts as a synchronization point at which all processes must wait for a boundary node solution to be returned. In addition, this single process can only receive and assemble serially, so the time complexity is simply the sum of the cost of receiving each process' boundary node system and adding to the global boundary node system. The cost of receiving all local boundary node systems is expressed as a function of \( P \), the number of processes, \( d_f \), \( c_{dp} \), the cost of sending a double precision number, and \( n_b \), the number of interprocess boundary nodes in each process as

\[
T_1 = \sum_{i=0}^{P-1} \left( c_{dp} \left( (d_f n_b) + d_f n_b \right) \right). \tag{4.9}
\]

Once received, each entry of the local system is added to the global system in time

\[
T_2 = \sum_{i=0}^{P-1} \left( (d_f n_b) + d_f n_b \right). \tag{4.10}
\]

Combining (4.9) and (4.10) gives us

\[
T_{recuscur}(P, d_f, c_{dp}, n_b) = \sum_{i=0}^{P-1} \left( (d_f n_b) + c_{dp} + d_f n_b \right). \tag{4.11}
\]

The cost of transmitting the global boundary node solutions is estimated by considering the point of view of the single process which does the sending. Again, this is a serial operation, so we simply sum the costs of sending each process its local boundary node solution, or

\[
T_{retsol}(P, d_f, c_{dp}, n_b) = \sum_{i=0}^{P-1} (c_{dp} d_f n_b). \tag{4.12}
\]

The total cost for the solution of the global boundary node system, including communication costs, is then,

\[
T_{soln}(P, N_b, d_f, c_{dp}, n_b) = T_{recuscur} + T_{globalsol} + T_{retsol}, \tag{4.13}
\]
where \( N_b \) is the total number of global boundary nodes and \( T_{global} \) is to be defined in Section 4.2.3.

**Step 5 - Solution of Unknowns at Interior Nodes**

Once the solutions at the interprocess boundary nodes have been returned to the processes, each process again works independently to solve for its interior node unknowns, using Equation (4.5). Two matrix operations are sufficient, since we can utilize \( \text{templhs} \) and \( \text{temprhs} \) which were calculated in Step 3:

1. \( u_i \leftarrow \text{templhs} \cdot u_b \). Multiplication of a \( df_n_i \times df_n_b \) matrix and \( df_n_b \times 1 \) vector will require \( 2df_n_i df_n_b \) tflops.

2. \( u_i \leftarrow \text{temprhs} - u_i \). This \( df_n_i \times 1 \) vector subtraction requires \( 2df_n_i \) tflops.

Totalling the complexities gives us an estimated time complexity of

\[
T_{int}(n_i, n_b, df) = 2df_n_i (df_n_b + 1).
\]  

**Step 6 - Local and Global Convergence Checking**

Convergence checking is the last step performed in a linearization iteration and requires interprocess communication to establish a global convergence status. First, each process checks its local convergence status in time

\[
T_{localconvg}(n, df) = df_n_i
\]  

where \( n \) is the total number of nodes in the process. Next, we utilize a balanced binary tree communication scheme (known as fan-in / fan-out) to consolidate the local convergence status of each process into a global convergence status. This implementation stores the binary tree as a heap, insuring a depth of
Using a fan-in, each process with children (in the binary tree) receives an integer flag from each child denoting convergence status in the lower subtree. These flags are combined with that of the current process in a logical AND operation and sent to the parent process, if it exists. When the root process receives and combines the flags, it holds the global convergence status and reverses the process, fanning-out the global convergence status to all processes. Clearly, this is a simple operation with a complexity dependent on the number of processes, $P$, and the communication cost $c_{dp}$. We derive a complexity of

$$T_{glbconvg}(P, c_{dp}) = 2c_{dp} \left[ \log_2 P \right],$$

noting that as in Step 1, this is a minor operation relative to the expensive matrix operations and does not warrant rigorous treatment.

This completes the description of the substructuring method details. We note that this technique requires little modification of existing finite element code, although a great deal of additional code will be required to implement the partitioning and to set up interprocess communication. If a direct solver was being utilized in the original finite element code, we can expect identical results (with a few possible exceptions in very ill-conditioned systems), since the substructuring methodology simply rearranges the matrix operations needed for the solution of the system in order to facilitate coarse grained parallelism. In effect, nothing in the general solution procedure has changed.

### 4.2.3 Solving the Global Boundary Node System

We now consider the solution of the global boundary node system derived in Step 4. Two methods have been implemented and tested - a single-process solver and a distributed solver. The nature of our equations is such that
we cannot assume any structure such as symmetry. In addition, eigenvalues for the equations are large, and attempts at utilizing iterative methods have been unsuccessful. Thus, our current work focuses on the direct solution of nonsymmetric matrices. In the following discussion, we assume that an $M \times M$ system of equations

$$Au = f$$

(4.17)

has been assembled in a single processor, as described in Step 4.

**Single-Process Solver**

Little needs to be said about a single-process solver. If we assume the use of a standard Gaussian elimination or LU decomposition routine with pivoting, we expect a time complexity of

$$T_{SGE}(M) = \frac{2}{3} M^3 + M^2 + M^2$$

(4.18)

to perform the factorization with pivoting, and backsubstitution [22]. Naturally, this is an attractive option because no interprocess communication is required, and it is simple to implement. The data is already in place and will not need to be distributed. If the global boundary node system is relatively small, this will be more than adequate and little will be gained by parallelization of this step.

**Distributed Solver**

For large global boundary node systems, we may want to consider parallelizing the solution. This will require the distribution of the assembled system of equations to the processes, extensive communication as the processes cooperate to solve the system, and a consolidation of the distributed solutions back to a single process.
The background for the method we have constructed is provided in Ortega [40]. This method is an LU decomposition in which the $M \times M$ matrix, $A$, is factored in a series of $M$ iterations where, at the $i^{th}$ iteration, the entries below the diagonal in the $i^{th}$ column of $A$ are used in the calculation of a multiplier vector, which is then used in the reduction of all entries in $A$ below and to the right of the diagonal entry, $A_{i,i}$. After the $i^{th}$ iteration, rows $(1,\ldots,i+1)$ and columns $(1,\ldots,i+1)$ will undergo no more modifications (see Figure 4.3).

After the matrix is factored, a forward and backward substitution are normally employed to obtain solutions for possibly more than one right hand side.

In the distributed implementation, the columns of $A$ are distributed to the processes in a wrapped interleaved fashion (see Figure 4.4) in order to load balance the processes. In addition, we store an identical copy of the right hand side, $f$ in each process and redundantly reduce each copy during the factorization stage. The merits of this approach have not been quantified, but the idea of performing redundant work on local data in order to avoid communication
Figure 4.4: Column wrapped interleaved storage of $A$. 
costs is utilized in many applications [28]. This allows us to bypass the forward reduction phase typical of LU decomposition methods, eliminating the associated communication costs. An algorithm for our implementation is now given, followed by a discussion of the key steps and a time complexity analysis.

**ALGORITHM LU.SPMD**

{Factorization phase}

1. FOR each column, \( col \), in global system DO
   IF \( my_{\text{process}} \) contains \( col \) THEN
   \[ j \leftarrow \text{local column number of } col \]
   {Find pivot row and entry, adjusting pivot vector, \( ipvt \)}
2. \[ \text{FOR } i = col + 1 \text{ TO num.global.cols DO} \]
   \[ l(i) = \frac{a(ipvt(i),j)}{a(ipvt(col),j)} \]
   ENDFOR
3. \[ \text{END IF} \]
4. {Fan-out \( l \) and pivot info to other processes}
   ENDFOR

{Reduce rows and columns past this diagonal entry}
5. FOR each column, \( j \), which follows \( col \) DO
6. \[ \text{FOR } i = col + 1 \text{ TO m DO} \]
   \[ a(ipvt(i),j) = a(ipvt(i),j)-l(i)\times a(ipvt( col ),j) \]
   ENDFOR
   ENDFOR

{Forward reduce load vector}
7. \[ \text{FOR } i = col + 1 \text{ TO m DO} \]
   \[ f(ipvt(i)) = f(ipvt(i))-l(i)\times f(ipvt( col )) \]
   ENDFOR

{Back substitution}
8. \[ \text{FOR } i = col \text{ DOWNTO 1 DO} \]
   \[ \text{partialJLp = 0} \]
   \[ k \leftarrow \text{lcl idx of 1st col. to right of glb col } i \]
9. \[ \text{FOR } j = k \text{ TO num.local.cols DO} \]
\[
\text{partial_ip} = \text{partial_ip} + a(i\text{pv}(i), j) \times u(j)
\]

ENDFOR

10. \{All processes fan-in and sum their partial inner products to the process which holds } u(i) \{\}

\[
\text{IF my-process holds u(i) THEN}
\]
\[
j \leftarrow \text{local column number of i}
\]
\[
u(j) = (f(i\text{pv}(i)) - \text{partial_ip}) / a(i\text{pv}(i), j)
\]

ENDIF

ENDFOR

END

For the sake of simplicity, in the following analysis we will assume without loss of generality, that the columns of the global boundary node system partition evenly over the set of processes, or \( M \mod P = 0 \).

The main loop (Line 1) of the solver routine is executed once for each of the \( M \) columns in the complete matrix. At a given iteration, \( i \), the process which possesses the “active” column (column \( i \) of the global matrix) finds the pivot element (Line 2) by searching all rows of the active column below the diagonal entry for the largest magnitude. This operation is performed in \( M - i \) TFLOPS. Next, the process with the active column forms a multiplier vector (Line 3) which will be sent to all processes so that they can reduce all appropriate rows and columns of their portion of the global matrix. Again, this operation will be performed in \( M - i \) TFLOPS.

Up to now, all but the process with the active column have remained idle. This ends when the multiplier vector is distributed to the other processes by fanning-out over a balanced binary tree of depth \( \lceil \log_2 P \rceil \) (Line 4). All but the leaf processes, upon receiving the vector from their parent, will send it to each of two children (if they exist). Thus, the two sends performed
at each level of the binary tree, on a vector of length $M - i$, will occur in $2\lceil \log_2 P \rceil (c_{dp}(M - i))$ tflops.

Once the multiplier vector has been received, the processes work in parallel to reduce their rows and columns below and to the right of the global diagonal entry, $A_{col, col}$, where $col$ is the current active column (Line 5). By inspection, we determine that the number of columns, $col$, at iteration $i$ in a given process, $P_j$ (where $0 \leq j \leq P - 1$), which are to the right of the global diagonal entry may be expressed as

$$col = \begin{cases} \frac{M - i \text{ div } P}{P} & \text{for } i \text{ mod } P \leq j < P \\ \frac{M}{P} - (i \text{ div } P - 1) & \text{for } 0 \leq j < i \text{ mod } P. \end{cases}$$  (4.19)

Since a process which reduces more columns than others will limit the overall speedup, we use (4.19) as an estimator. The reduction of entries in a single column (Line 6) will occur in $2(M - i)$ tflops, giving us an estimated complexity of

$$2(M - i) \left( \frac{M}{P} - (i \text{ div } P - 1) \right)$$

for the reductions performed at iteration $i$. In addition, since we are reducing the right hand side at the same time (Line 7), we require another $2(M - i)$ tflops.

After $M$ iterations of the outer loop (Line 1), the factorization phase is completed. The sum of the estimated complexities at each iteration, $i$, enables us to derive an expression for the complexity of the entire factorization phase as a function of $M$, the dimension of the global boundary node system, $P$, the number of processes, and $c_{dp}$, the cost of passing a double precision number between processes:

$$T_{LU,F}(M, P, c_{dp}) = \sum_{i=1}^{M} ((M - i) + (M - i) + 2\lceil \log_2 P \rceil (c_{dp}(M - i)) +$$
\[ 2(M - i) \left( \frac{M}{P} - (i \div P - 1) \right) + 2(M - i) \]
\[ = 2 \sum_{i=1}^{M} \left\{ (M - i) \left[ 2 + c_{dp} \log_{2} P \right] + \left( \frac{M}{P} - (i \div P - 1) \right) \right\} \] (4.20)

Having factored the global distributed coefficient matrix and forward reduced the right hand side, the solution \( u \) may be obtained by back substitution with the upper triangular portion of the LU decomposition. In a serial implementation, the solution, \( u_i \), would be obtained as follows:

\[
\begin{align*}
    u_n &= \frac{f_n}{A_{n,n}} \\
    u_i &= \frac{f_i - \sum_{k=i}^{n-1} A_{i,k}u_k}{A_{i,i}}, \quad i = n - 1, n - 2, \ldots, 1.
\end{align*}
\] (4.21) (4.22)

In an environment which distributes the columns, for each iteration \( i \) of the backsubstitution loop (Line 8) we calculate the "partial" inner products of Eq. (4.22) (Line 9) for the terms owned by a process (performing this in parallel, of course), then accumulate the sums by a fan-in to the process which contains column \( i \) of the global matrix (Line 10). Finally, the process containing column \( i \) calculates the solution for \( u_i \) (Line 11).

By inspection (admittedly, a tedious trial and error task in this case!), we find that at iteration \( i \), in process \( P_j \), where \( 0 \leq j \leq P - 1 \), the calculation of the partial inner product (Line 9) will require two operations at each of the local columns to the right of global column \( i \), the number of such columns being either \( (i - 1) \div P + 1 \) or \( (i - 1) \div P \), depending on the iteration and the process. Since the processes containing more columns will limit the speedup, we use the larger value, giving us an estimated complexity of \( 2 ((i - 1) \div P + 1) \) for the calculation of the inner products.
The fan-in of the partial inner products to the process containing global column $i$ (Line 10) will occur over a balanced binary tree of depth $[\log_2 P]$. At all but the leaf processes, up to two partial inner products will be received and added to the local inner product, in time

$$2(c_{dp} + 1)[\log_2 P].$$

(4.23)

Finally, $u_i$ is calculated with two operations in the root process (Line 11).

After $M$ iterations of the outer loop (Line 8), we obtain a solution which is distributed over the processes. The sum of the estimated complexities over these iterations is

$$T_{LU-B}(M, P, c_{dp}) = \sum_{i=1}^{M} \left(2\left((i - 1) \div P + 1\right) + 2(c_{dp} + 1)[\log_2 P] + 2\right).$$

(4.24)

We can now express the complexity estimate of the distributed LU factorization and solution as

$$T_{LU}(M, P, c_{dp}) = T_{LU-F}(M, P, c_{dp}) + T_{LU-B}(M, P, c_{dp}).$$

(4.25)

Of course, Eq. (4.25) assumes that the global boundary node system has already been distributed to the processes, and it neglects the consolidation of the solution. This is now considered, obtaining a formulation for the entire solution complexity, under the assumption that the global boundary node system is initially stored in a single process (which is the assumption we have been utilizing) and that the solution will be returned to a single process. As before, we will assume without loss of generality, that the columns of the global matrix distribute evenly across the processes, or $M \mod P = 0$. 


Distribution of the global boundary node system must occur serially. For each process the single process sends the appropriate $\frac{M^2}{P}$ coefficients of the matrix and the $M$ entries of the right hand side, each process requiring

$$c_{dp} \left( \frac{M^2}{P} + M \right) \text{tflops.}$$

Thus, the total cost of distribution is

$$P \left( c_{dp} \left[ \frac{M^2}{P} + M \right] \right) = c_{dp}(M^2 + MP).$$  \hfill (4.26)

Consolidation of the solution will occur in a similar manner. The single process receives each process’ $\frac{M}{P}$ solutions serially, at a total cost of

$$c_{dp}M.$$  \hfill (4.27)

Now, we have all the pieces necessary to formulate an estimate of the distributed solver complexity, expressing it by combining (4.25), (4.26), and (4.27) to form

$$T_{DSOL}(M, P, c_{dp}) = c_{dp}(M^2 + MP) + T_{LU}(M, P, c_{dp}) + c_{dp}M$$

$$= c_{dp}M(M + P + 1) + T_{LU}(M, P, c_{dp}).$$  \hfill (4.28)

Before moving to the next section, we will briefly analyze the theoretical performance of the distributed solver on two matrices of dimension 1024 (2^{10}) and 131072 (2^{17}). For each matrix, we test two different communication cost parameters, $c_{dp}$. The first of these parameters is chosen to represent “perfect” communication on the RS/6000 cluster at LSU. The RS/6000-370 has a rating of 25.9 MFLOPS (double precision), and the ethernet communication transfer rate on the cluster of workstations is 100 megabits/second [19]. If we assume that the network will always pass messages at the peak rate (at 64 bits per
double precision number, this rate is equivalent to approximately $1.6 \times 10^6$

double precision numbers per second), then we arrive at a communication cost of

$$c_{dp} = \frac{25.9 \times 10^6 \text{flops}}{1.6 \times 10^6 \text{d.p. numbers per sec}} \approx 16 \text{ tflops}.$$  

Given the system specifications, this is clearly an unrealistic value as it ignores network contention and message handling time within the processors. However, we use it for illustration, and compare results against a somewhat arbitrarily chosen value of

$$c_{dp} = 100.0 \text{ tflops}.$$

Tables 4.1 and 4.2 list the results of this analysis and Figures 4.5 and 4.6 display plots of theoretical speedup for each system of equations on

$$P = 2^i, \quad 0 \leq i \leq 8$$

processors. Speedup, defined as

$$S_p = \frac{T_1}{T_p} \quad (4.29)$$
Table 4.2: Theoretical performance of distributed solver for $M = 131072$.

<table>
<thead>
<tr>
<th>Procs</th>
<th>$c_{dp} = 16$ tflsps</th>
<th>Speedup</th>
<th>$c_{dp} = 100$ tflsps</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.50e15</td>
<td>1.0e0</td>
<td>1.50e15</td>
<td>1.0e0</td>
</tr>
<tr>
<td>2</td>
<td>7.51e14</td>
<td>2.0e0</td>
<td>7.52e14</td>
<td>2.0e0</td>
</tr>
<tr>
<td>4</td>
<td>3.76e14</td>
<td>4.0e0</td>
<td>3.79e14</td>
<td>4.0e0</td>
</tr>
<tr>
<td>8</td>
<td>1.88e14</td>
<td>8.0e0</td>
<td>1.93e14</td>
<td>7.8e0</td>
</tr>
<tr>
<td>16</td>
<td>9.50e13</td>
<td>1.6e1</td>
<td>1.01e14</td>
<td>1.5e1</td>
</tr>
<tr>
<td>32</td>
<td>4.83e13</td>
<td>3.1e1</td>
<td>5.56e13</td>
<td>2.7e1</td>
</tr>
<tr>
<td>64</td>
<td>2.52e13</td>
<td>6.0e1</td>
<td>3.38e13</td>
<td>4.4e1</td>
</tr>
<tr>
<td>128</td>
<td>1.37e13</td>
<td>1.1e2</td>
<td>2.38e13</td>
<td>6.3e1</td>
</tr>
<tr>
<td>256</td>
<td>8.12e12</td>
<td>1.8e2</td>
<td>1.97e13</td>
<td>7.6e1</td>
</tr>
</tbody>
</table>

Figure 4.5: Theoretical speedup vs. number of processors for $M = 1024$. 
where \( p \) is the number of processors and \( T_i \) is the time required to perform the computations with \( i \) processors, is ideally equal to the number of processors solving the problem, implying that each of the \( P \) processors is concurrently computing \( 1/P \) of the problem. In reality, this is difficult to achieve, and we must often settle for less. However, the speedup value is a useful metric for gauging the efficiency with which we are utilizing the processors.

In Figure 4.5, we see that speedups deviate rapidly from the theoretical maximum, particularly when the communication costs are high. Figure 4.6 shows us better performance, although when the communication costs are high we notice a marked deterioration in the speedups. These results are expected, reinforcing the well known communication facts in the parallel computing community. In general, we attempt to utilize parallel processors in a way that will maximize the amount of computation performed in the processors relative to the interprocess communication.

Figure 4.6: Theoretical speedup vs. number of processors for \( M = 131072 \).
The theoretical performance analysis of the distributed solver shows us that for large problems, we can expect efficient computations with impressive speedups. For small problems, however, it may be more efficient to solve the global boundary node system in a single processor and avoid the overhead involved in distributing such a system.

### 4.3 Analysis of Relative Speedup Characteristics

#### 4.3.1 A General Formula for Estimating Time Complexity

Having described the substructuring method and performed a theoretical complexity analysis of the pieces, we now put everything together and analyze the relative performance of the method on two sample problems under various degrees of partitioning. First, to summarize, and for convenience, we display the pertinent formulas derived in the previous section.

\[
T_{\text{assem}}(n, d_f) = n \cdot d_f. \tag{4.30}
\]

\[
T_{\text{schur}}(n_i, n_b, d_f) = 2 \left( (d_f n_i)^3 + d_f^2 n_i^2 n_b + d_f^2 n_i n_b^2 + (d_f n_b)^2 + (d_f n_i)^2 + d_f^2 n_i n_b + d_f n_b \right) \tag{4.31}
\]

\[
T_{\text{solvbd}}(P, N_b, d_f, c_{dp}, n_b) = T_{\text{recuschur}} + T_{\text{global sol}} + T_{\text{resol}} \tag{4.32}
\]

\[
T_{\text{int}}(n_i, n_b, d_f) = 2 d_f n_i (d_f n_b + 1) \tag{4.33}
\]

\[
T_{\text{ldconvg}}(n, d_f) = d_f n \tag{4.34}
\]

\[
T_{\text{gilconvg}}(P, c_{dp}) = 2 c_{dp} \log_2 P \tag{4.35}
\]

The formula for the solution of the global boundary node system \(T_{\text{solvbd}}\), we recall, considers the costs of sending each local boundary node system to a
single process

\[ T_{\text{recuschur}}(P, d_f, c_{dp}, n_b) = \sum_{i=0}^{P-1} \{(d_f n_{bi}) (d_f n_{bi} (c_{dp} + 1)) + 2\}, \]  

(4.36)

solving the system, then returning the solution to the appropriate processes

\[ T_{\text{reetsol}}(P, d_f, c_{dp}, n_b) = \sum_{i=0}^{P-1} (c_{dp} d_f n_{bi}). \]  

(4.37)

To solve the global boundary node system, we either do so in a single process, where the relative complexity formula is

\[ T_{\text{globsol}} = T_{\text{SGE}}(M) = \frac{2}{3} M^3 + M^2 + M^2, \]  

(4.38)

or distribute the system and solve in parallel, in which case the complexity formula is

\[ T_{\text{globsol}} = T_{\text{DSOL}}(M, P, c_{dp}) = c_{dp} M(M + P + 1) + T_{\text{LU}}(M, P, c_{dp}), \]  

(4.39)

where

\[ T_{\text{LU}}(M, P, c_{dp}) = T_{\text{LU,F}}(M, P, c_{dp}) + T_{\text{LU,B}}(M, P, c_{dp}), \]  

(4.40)

and

\[ T_{\text{LU,F}}(M, P, c_{dp}) = \sum_{i=1}^{M} ((M - i) + (M - i) + 2\lfloor \log_2 P \rfloor (c_{dp}(M - i)) + \]
\[ 2(M - i) \left(\frac{M}{P} - (\text{idiv}P - 1)\right) + 2(M - i) \right\}, \]  

(4.41)

\[ T_{\text{LU,B}}(M, P, c_{dp}) = \sum_{i=1}^{M} (2((i - 1)\text{div}P + 1) + 2(c_{dp} + 1)\lfloor \log_2 P \rfloor + 2) \]
\[ = \sum_{i=1}^{M} ((i - 1)\text{div}P + (c_{dp} + 1)\lfloor \log_2 P \rfloor + 2). \]  

(4.42)
In the above formulas, the following notations apply — \( P \) is the number of processes, \( n_i \) and \( n_b \) are the number of interior and interprocess boundary nodes, respectively, in a given element, \( n = n_i + n_b \) is the total number of nodes in a given process, \( N_b \) is the total number of interprocess boundary nodes, \( d_f \) is the number of degrees of freedom, or unknowns, per node, \( c_{dp} \) is the communication cost parameter, \( M \) is the dimension of a system of equations, and \( \bar{n}_b \) is a vector of length \( P \) where the \( i^{th} \) entry, \( 0 \leq i \leq P - 1 \), is the number of interprocess boundary nodes in process \( i \).

To construct a general formula for the relative complexity of a single iteration in the substructuring finite element method, we must consider the fact that processes with a heavier workload will determine the amount of time that a set of parallel tasks requires for computations. Thus, between each synchronization point (e.g. after the global convergence check and before the solution of the global boundary node system at the next iteration) we base the complexity on the slowest process. The formula describing the complexity of an iteration will be

\[
T_{iter} = \max_{p \in P} \left( T_{assembl}(n^p, d_f) + T_{Schur}(n^p_i, n^p_b, d_f) + T_{solbnd}(P, N_b, d_f, c_{dp}, \bar{n}_b) + \right.
\]

\[
\max_{p \in P} \left( T_{int}(n^p_i, n^p_b, d_f) + T_{ldconvg}(n^p, d_f) + T_{gibconvg}(P, c_{dp}) \right), \tag{4.43}
\]

where a \( p \) superscript denotes the value of a variable for process \( p \).

Equation (4.43) is a general formula which considers most of the computational and communication requirements of the methods described above. As one might imagine, most of the work is performed in a small part of the overall procedure. Experience has shown us that for all but the smallest of problems, \( T_{Schur} \) and/or \( T_{solbnd} \) dominate the complexity by at least two orders of magnitude. This should come as no surprise, since both account for the expensive
\( \mathcal{O}(N^3) \) matrix operations. Thus, in the following analyses we only consider the effects of these two terms. We note that these analyses originally considered all terms, and we verified that the neglected terms were insignificant. For our purposes then, the relative complexity will be estimated by using

\[
T_{\text{iter}} = \max_{p \in P} (T_{\text{Schur}}(n_i^p, n_b^p, d_f)) + T_{\text{solvnd}}(P, N_b, d_f, c_{dp}, n_b) \tag{4.44}
\]

### 4.3.2 Analysis of Complexity — Test Cases

**Test Case I**

We first investigate the complexity of this method on a two dimensional square domain of 256 \( \times \) 256 elements (66049 nodes – assuming two unknowns at each node gives us a total of 132098 unknowns), partitioned evenly into \( P = 4^i, 0 \leq i \leq 4 \) processes. For \( i \leq 1 \), all partitions will have the same number of interior and interprocess boundary nodes. However, for \( i > 1 \), we must consider three types of partitions and choose the partition type having the largest complexity to make our estimates. Refering to Figure 4.7, we define a Type I partition to be a partition on a corner of the problem domain, having two edges which lie on interprocess boundaries. A Type II partition lies on an edge of the problem domain (excluding corners) and has three edges lying on interprocess boundaries, and a Type III partition is located in the interior of the problem domain, having all four edges on interprocess boundaries.

Tables 4.3 and 4.4 list the results of the computations with \( c_{dp} = 16.0 \) and \( c_{dp} = 100.0 \), respectively, and \( d_f = 2 \). For each partitioning, we list \( N_b \), the total number of interprocess boundary nodes, the type (and number of such types) of partitions, \( n_i \) and \( n_b \), \( T_{\text{Schur}} \), the estimated complexity (in tflops) for constructing the Schur complement, \( T_{\text{solvnd}} \), the estimated complexity for solving the global boundary node system in a single process, and \( T_{\text{dist}} \), the
Table 4.3: Estimated complexity for $256 \times 256$ element problem with $c_{dp} = 16.0$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$N_b$</th>
<th>Type (Num)</th>
<th>$n_i$</th>
<th>$n_b$</th>
<th>$T_{Schar}$</th>
<th>$T_{soln}^{1}$</th>
<th>$T_{distr}^{1}$</th>
<th>$T_{par}^{1}$</th>
<th>$T_{distr}^{2}$</th>
<th>$T_{par}^{2}$</th>
<th>$S_{raw}^{1}$</th>
<th>$S_{distr}^{1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>66049</td>
<td>0</td>
<td>4.6e15</td>
<td>-</td>
<td>-</td>
<td>4.6e15</td>
<td>-</td>
<td>1.0e0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>513</td>
<td></td>
<td>16384</td>
<td>257</td>
<td>7.1e13</td>
<td>7.4e8</td>
<td>2.5e8</td>
<td>7.1e13</td>
<td>7.1e13</td>
<td>6.5e1</td>
<td>6.5e1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1533</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I (4)</td>
<td>4096</td>
<td>129</td>
<td>1.1e12</td>
<td>1.9e10</td>
<td>2.0e9</td>
<td>1.1e12</td>
<td>1.1e12</td>
<td>4.2e3</td>
<td>4.2e3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (8)</td>
<td>4032</td>
<td>193</td>
<td>1.1e12</td>
<td>1.9e10</td>
<td>2.0e9</td>
<td>1.1e12</td>
<td>1.1e12</td>
<td>4.2e3</td>
<td>4.2e3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>III (4)</td>
<td>3969</td>
<td>256</td>
<td>1.1e12</td>
<td>1.9e10</td>
<td>2.0e9</td>
<td>1.1e12</td>
<td>1.1e12</td>
<td>4.2e3</td>
<td>4.2e3</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>3549</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I (4)</td>
<td>1024</td>
<td>65</td>
<td>1.8e10</td>
<td>2.4e11</td>
<td>9.6e9</td>
<td>2.6e11</td>
<td>2.8e10</td>
<td>1.8e4</td>
<td>1.6e5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (24)</td>
<td>992</td>
<td>97</td>
<td>1.7e10</td>
<td>2.4e11</td>
<td>9.6e9</td>
<td>2.6e11</td>
<td>2.8e10</td>
<td>1.8e4</td>
<td>1.6e5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>III (36)</td>
<td>961</td>
<td>128</td>
<td>1.6e10</td>
<td>2.4e11</td>
<td>9.6e9</td>
<td>2.6e11</td>
<td>2.8e10</td>
<td>1.8e4</td>
<td>1.6e5</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>7485</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I (4)</td>
<td>256</td>
<td>33</td>
<td>3.1e8</td>
<td>2.2e12</td>
<td>4.2e10</td>
<td>2.2e12</td>
<td>4.2e10</td>
<td>2.1e3</td>
<td>1.1e5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (56)</td>
<td>240</td>
<td>49</td>
<td>2.8e8</td>
<td>2.2e12</td>
<td>4.2e10</td>
<td>2.2e12</td>
<td>4.2e10</td>
<td>2.1e3</td>
<td>1.1e5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>III (196)</td>
<td>225</td>
<td>64</td>
<td>2.5e8</td>
<td>2.2e12</td>
<td>4.2e10</td>
<td>2.2e12</td>
<td>4.2e10</td>
<td>2.1e3</td>
<td>1.1e5</td>
<td></td>
</tr>
</tbody>
</table>
Table 4.4: Estimated complexity for 256 × 256 element problem with $c_{dp} = 100.0$.

<table>
<thead>
<tr>
<th>P</th>
<th>$N_b$</th>
<th>Type (Num)</th>
<th>$n_t$</th>
<th>$n_b$</th>
<th>$T_{Schur}$</th>
<th>$T_{solbnd, 1}$</th>
<th>$T_{solbnd, 1}$</th>
<th>$T_{par, 1}$</th>
<th>$T_{par, 1}$</th>
<th>$S_{raw}^{1}$</th>
<th>$S_{distr}^{1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>—</td>
<td>66049</td>
<td>0</td>
<td>4.6e15</td>
<td>0.0e0</td>
<td>—</td>
<td>4.6e15</td>
<td>—</td>
<td>1.0e0</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>513</td>
<td>—</td>
<td>16384</td>
<td>257</td>
<td>7.1e13</td>
<td>8.3e8</td>
<td>6.1e8</td>
<td>7.1e13</td>
<td>7.1e13</td>
<td>6.5e1</td>
<td>6.5e1</td>
</tr>
<tr>
<td>16</td>
<td>1533</td>
<td>I (4)</td>
<td>4096</td>
<td>129</td>
<td>1.1e12</td>
<td>1.9e10</td>
<td>6.2e9</td>
<td>1.1e12</td>
<td>1.1e12</td>
<td>4.2e3</td>
<td>4.2e3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (8)</td>
<td>4032</td>
<td>193</td>
<td>1.1e12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>III (4)</td>
<td>3969</td>
<td>256</td>
<td>1.1e12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>3549</td>
<td>I (4)</td>
<td>1024</td>
<td>65</td>
<td>1.8e10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (24)</td>
<td>992</td>
<td>97</td>
<td>1.7e10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>III (36)</td>
<td>961</td>
<td>128</td>
<td>1.6e10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>7485</td>
<td>I (4)</td>
<td>256</td>
<td>33</td>
<td>3.1e8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (56)</td>
<td>240</td>
<td>49</td>
<td>2.8e8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>III (196)</td>
<td>225</td>
<td>64</td>
<td>2.5e8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.7: 256 × 256 element problem domain partitioned into 16 subdomains.

The estimated complexity with the distributed solution approach. The estimated complexity for a complete iteration is the sum of the maximum value of \( T_{\text{Schur}} \) and the solver. This is given as \( T^1_{\text{par}} \) for the serial solver and \( T^1_{\text{distr}} \) for the distributed solver. Finally, the “raw” speedups, \( S^1_{\text{raw}} \) and \( S^1_{\text{distr}} \) are calculated as the ratio of the estimated complexity for the case of a single process to the estimated complexity for the problem running on the given number of processors.

A first glance at the raw speedups for this problem leads one to believe that a superlinear algorithm has been implemented. Indeed, we see remarkable speedups when we consider the performance of the test runs relative to each other. However, it turns out that most of this improvement in speed comes not from utilizing parallel processors, but from partitioning the matrix operations into smaller, less expensive ones. In other words, much of this speedup could be attained if the substructuring method was employed on a single processor.
Tables 4.5 and 4.6 we compute $\sum T_{Schur}$, the sum of $T_{Schur}$ in each partition, as if the calculations were occurring serially in a single process. Next, we calculate $T_{solbnd}^{ser}$, the estimated complexity of a serial solver on the global boundary node system, excluding any communication costs (i.e. $c_{dp} = 0.0$). $T_{ser}$, the sum of $\sum T_{Schur}$ and $T_{solbnd}^{ser}$ give us an estimate for the complexity of this method when calculated on a single processor. We repeat the entries for $T_{par}^{1}$ and $T_{par}^{distr}$ from the previous tables, then calculate the “adjusted” speedups, $S_{adj}^{1}$ and $S_{adj}^{distr}$ as the ratio of $T_{ser}$ to the estimated complexity for the parallel computations. This provides us with a more realistic measure of the efficiency. Adjusted speedups for the two cases are plotted in Figures 4.8 and 4.9.

Table 4.5: Calculation of adjusted speedups for 256 $\times$ 256 element problem with $c_{dp} = 16.0$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$N_b$</th>
<th>$\sum T_{Schur}$</th>
<th>$T_{solbnd}^{ser}$</th>
<th>$T_{ser}$</th>
<th>$T_{par}^{1}$</th>
<th>$T_{par}^{distr}$</th>
<th>$S_{adj}^{1}$</th>
<th>$S_{adj}^{distr}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4.6e15</td>
<td>0.0e0</td>
<td>4.6e15</td>
<td>4.6e15</td>
<td>–</td>
<td>1.0e0</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>513</td>
<td>2.8e14</td>
<td>7.2e8</td>
<td>2.8e14</td>
<td>7.1e13</td>
<td>7.1e13</td>
<td>3.9e0</td>
<td>3.9e0</td>
</tr>
<tr>
<td>16</td>
<td>1533</td>
<td>1.8e13</td>
<td>1.9e10</td>
<td>1.8e13</td>
<td>1.1e12</td>
<td>1.1e12</td>
<td>1.6e1</td>
<td>1.6e1</td>
</tr>
<tr>
<td>64</td>
<td>3549</td>
<td>1.0e12</td>
<td>2.4e11</td>
<td>1.2e12</td>
<td>2.6e11</td>
<td>2.8e10</td>
<td>4.6e0</td>
<td>4.3e1</td>
</tr>
<tr>
<td>256</td>
<td>7485</td>
<td>6.6e10</td>
<td>2.2e12</td>
<td>2.2e12</td>
<td>2.2e12</td>
<td>4.2e10</td>
<td>1.0e0</td>
<td>5.2e1</td>
</tr>
</tbody>
</table>

Table 4.6: Calculation of adjusted speedups for 256 $\times$ 256 element problem with $c_{dp} = 100.0$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$N_b$</th>
<th>$\sum T_{Schur}$</th>
<th>$T_{solbnd}^{ser}$</th>
<th>$T_{ser}$</th>
<th>$T_{par}^{1}$</th>
<th>$T_{par}^{distr}$</th>
<th>$S_{adj}^{1}$</th>
<th>$S_{adj}^{distr}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4.6e15</td>
<td>0.0e0</td>
<td>4.6e15</td>
<td>4.6e15</td>
<td>–</td>
<td>1.0e0</td>
<td>1.0e0</td>
</tr>
<tr>
<td>4</td>
<td>513</td>
<td>2.8e14</td>
<td>7.2e8</td>
<td>2.8e14</td>
<td>7.1e13</td>
<td>7.1e13</td>
<td>3.9e0</td>
<td>3.9e0</td>
</tr>
<tr>
<td>16</td>
<td>1533</td>
<td>1.8e13</td>
<td>1.9e10</td>
<td>1.8e13</td>
<td>1.1e12</td>
<td>1.1e12</td>
<td>1.6e1</td>
<td>1.6e1</td>
</tr>
<tr>
<td>64</td>
<td>3549</td>
<td>1.0e12</td>
<td>2.4e11</td>
<td>1.2e12</td>
<td>2.6e11</td>
<td>5.8e10</td>
<td>4.6e0</td>
<td>2.1e1</td>
</tr>
<tr>
<td>256</td>
<td>7485</td>
<td>6.6e10</td>
<td>2.2e12</td>
<td>2.2e12</td>
<td>2.2e12</td>
<td>2.1e12</td>
<td>1.0e0</td>
<td>1.0e0</td>
</tr>
</tbody>
</table>
Figure 4.8: Speedup estimates for the $256 \times 256$ element problem with $c_{dp} = 16.0$.

Figure 4.9: Speedup estimates for the $256 \times 256$ element problem with $c_{dp} = 100.0$. 
We see that for both values of $c_{dp}$, the speedups begin to deteriorate with more than sixteen processors. Note that when we use a small number of processors, the number of interior nodes is large in each process, creating a $T_{Schur}$ which dominates the complexity, and as the partitioning of the domain increases, the number of interprocess boundary nodes grows, and the solution of the global boundary node system dominates the complexity. Notice also that the speedups begin to depart from the theoretical maximum (the line with a slope of 1) just as the computations for solving the global boundary node system begin to dominate. This is no coincidence, for even with high communication costs, the construction of the Schur complement is embarrassingly parallel. As one would expect, the bottleneck occurs where a large amount of work must be performed cooperatively among the processors.

**Test Case II**

To lead into the performance test of the next section, we perform the estimate of complexity on a representative three dimensional static mesh (Figure 4.10) utilized for modelling the Hebert problem (modelled in two dimensions on a dynamic mesh in Section 2.3.2). We observe that the twelve major columns which make up the three dimensional mesh allow for several domain partitioning schemes which assign equal numbers of nodes and elements to each process. For example, we may utilize a single process that holds all twelve columns, two processes that hold six columns each, three processes holding four columns each, four processes holding three columns each, or six processes holding two columns each. Again, it is necessary to define partition types, since some will have the same total number of nodes, but not necessarily the same proportion.
Figure 4.10: Three dimensional static mesh for Hebert problem.
of interior nodes to interprocess boundary nodes. Thus, we define a Type I partition to be a partition on either end of the mesh, sharing only one boundary with another process. A Type II partition is located between end partitions, sharing two boundaries with other partitions.

The analyses performed for Test Case I are repeated for this problem. Tables 4.7 and 4.8 list the results of the computations with $c_{dp} = 16.0$ and $c_{dp} = 100.0$, respectively, and $d_f = 2$.

As before, the “raw” speedups observed are real, but most of this is a result of the operations on smaller matrices that the substructuring method provides. The adjusted speedups are provided in Tables 4.9 and 4.10 and plotted in Figures 4.11 and 4.12. As expected, we see deteriorating performance as we increase the number of processes working on the problem, and as communication costs increase. It is evident that a problem of this size will
Table 4.7: Estimated complexity for Hebert problem with $c_{dp} = 16.0$.

<table>
<thead>
<tr>
<th>P</th>
<th>$N_b$</th>
<th>Type (Num)</th>
<th>$n_i$</th>
<th>$n_b$</th>
<th>$T_{schur}$</th>
<th>$T_{s_b}$</th>
<th>$T_{s_b}^{dist}$</th>
<th>$T_{par}$</th>
<th>$T_{par}^{dist}$</th>
<th>$S_{raw}$</th>
<th>$S_{raw}^{dist}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>1112</td>
<td>0</td>
<td>2.2e10</td>
<td>0.0e0</td>
<td>-</td>
<td>2.2e10</td>
<td>-</td>
<td>2.2e10</td>
<td>2.2e10</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>-</td>
<td>540</td>
<td>32</td>
<td>2.7e9</td>
<td>3.2e5</td>
<td>3.8e5</td>
<td>2.7e9</td>
<td>2.7e9</td>
<td>8.1e0</td>
<td>8.1e0</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>I (2)</td>
<td>360</td>
<td>32</td>
<td>8.2e8</td>
<td>1.8e6</td>
<td>1.5e6</td>
<td>8.2e8</td>
<td>8.2e8</td>
<td>2.7e1</td>
<td>2.7e1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (1)</td>
<td>328</td>
<td>64</td>
<td>7.0e8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>96</td>
<td>I (2)</td>
<td>270</td>
<td>32</td>
<td>3.6e8</td>
<td>5.4e6</td>
<td>3.8e6</td>
<td>3.6e8</td>
<td>3.6e8</td>
<td>6.1e1</td>
<td>6.1e1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (2)</td>
<td>234</td>
<td>64</td>
<td>2.8e8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>160</td>
<td>I (2)</td>
<td>180</td>
<td>32</td>
<td>1.4e8</td>
<td>2.3e7</td>
<td>1.0e7</td>
<td>1.6e8</td>
<td>1.5e8</td>
<td>1.4e2</td>
<td>1.5e2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (4)</td>
<td>148</td>
<td>64</td>
<td>8.4e7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4.8: Estimated complexity for Hebert problem with $c_{dp} = 100.0$.

<table>
<thead>
<tr>
<th>P</th>
<th>$N_b$</th>
<th>Type (Num)</th>
<th>$n_i$</th>
<th>$n_b$</th>
<th>$T_{Schur}$</th>
<th>$T_{\text{sol,}i}$</th>
<th>$T_{\text{dist,}i}$</th>
<th>$T_{\text{par}}$</th>
<th>$T_{\text{dist,}par}$</th>
<th>$S^{\text{1}}_{\text{raw}}$</th>
<th>$S^{\text{1}}_{\text{dist}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>1112</td>
<td>0</td>
<td>2.2e10</td>
<td>0.0e0</td>
<td>-</td>
<td>2.2e10</td>
<td>-</td>
<td>1.0e0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>-</td>
<td>540</td>
<td>32</td>
<td>2.7e9</td>
<td>1.0e6</td>
<td>1.8e6</td>
<td>2.7e9</td>
<td>8.1e0</td>
<td>8.1e0</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>I (2)</td>
<td>360</td>
<td>32</td>
<td>8.2e8</td>
<td>3.9e6</td>
<td>6.4e6</td>
<td>8.2e8</td>
<td>8.2e8</td>
<td>2.7e1</td>
<td>2.7e1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (1)</td>
<td>328</td>
<td>64</td>
<td>7.0e8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>96</td>
<td>I (2)</td>
<td>270</td>
<td>32</td>
<td>3.6e8</td>
<td>8.9e5</td>
<td>1.7e7</td>
<td>3.6e8</td>
<td>3.8e8</td>
<td>6.1e1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (2)</td>
<td>234</td>
<td>64</td>
<td>2.8e8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>160</td>
<td>I (2)</td>
<td>180</td>
<td>32</td>
<td>1.4e8</td>
<td>2.9e7</td>
<td>4.2e7</td>
<td>1.7e8</td>
<td>1.8e8</td>
<td>1.3e2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>II (4)</td>
<td>148</td>
<td>64</td>
<td>8.4e7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 4.9: Calculation of adjusted speedups for Hebert problem with $c_{dp} = 16.0$.

<table>
<thead>
<tr>
<th>P</th>
<th>$N_b$</th>
<th>$\sum T_{Schur}$</th>
<th>$T_{ser, solbood}$</th>
<th>$T_{ser}$</th>
<th>$T_{par}^{1}$</th>
<th>$T_{distr}^{par}$</th>
<th>$S_{adj}$</th>
<th>$S_{distr}^{1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2.2e10</td>
<td>0.0e0</td>
<td>2.2e10</td>
<td>2.2e10</td>
<td>-</td>
<td>1.0e0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>5.4e9</td>
<td>1.8e5</td>
<td>5.4e9</td>
<td>2.7e9</td>
<td>2.7e9</td>
<td>2.0e0</td>
<td>2.0e0</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>2.3e9</td>
<td>1.4e6</td>
<td>2.3e9</td>
<td>8.2e8</td>
<td>8.2e8</td>
<td>2.8e0</td>
<td>2.8e0</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
<td>1.3e9</td>
<td>4.8e6</td>
<td>1.3e9</td>
<td>3.6e8</td>
<td>3.6e8</td>
<td>3.6e0</td>
<td>3.6e0</td>
</tr>
<tr>
<td>6</td>
<td>160</td>
<td>6.2e8</td>
<td>2.2e7</td>
<td>6.4e8</td>
<td>1.6e8</td>
<td>1.5e8</td>
<td>4.0e0</td>
<td>4.3e0</td>
</tr>
</tbody>
</table>

Table 4.10: Calculation of adjusted speedups for Hebert problem with $c_{dp} = 100.0$.

<table>
<thead>
<tr>
<th>P</th>
<th>$N_b$</th>
<th>$\sum T_{Schur}$</th>
<th>$T_{ser, solbood}$</th>
<th>$T_{ser}$</th>
<th>$T_{par}^{1}$</th>
<th>$T_{distr}^{par}$</th>
<th>$S_{adj}$</th>
<th>$S_{distr}^{1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2.2e10</td>
<td>0.0e0</td>
<td>2.2e10</td>
<td>2.2e10</td>
<td>-</td>
<td>1.0e0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>5.4e9</td>
<td>1.8e5</td>
<td>5.4e9</td>
<td>2.7e9</td>
<td>2.7e9</td>
<td>2.0e0</td>
<td>2.0e0</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>2.3e9</td>
<td>1.4e6</td>
<td>2.3e9</td>
<td>8.2e8</td>
<td>8.2e8</td>
<td>2.8e0</td>
<td>2.8e0</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
<td>1.3e9</td>
<td>4.8e6</td>
<td>1.3e9</td>
<td>3.6e8</td>
<td>3.6e8</td>
<td>3.6e0</td>
<td>3.6e0</td>
</tr>
<tr>
<td>6</td>
<td>160</td>
<td>6.2e8</td>
<td>2.2e7</td>
<td>6.4e8</td>
<td>1.7e8</td>
<td>1.8e8</td>
<td>3.8e0</td>
<td>3.6e0</td>
</tr>
</tbody>
</table>

Figure 4.12: Theoretical speedup estimates for the Hebert problem with $c_{dp} = 100.0$. 
gain little by attempting to utilize more processors, due to the increasingly fine granularity of the partitioning.

4.3.3 Analysis of Complexity – Summary

The results of these complexity estimates show us that there is great potential in this methodology, but at the same time we must be aware of its limitations. The most important parameter for determining whether a partitioning scheme will provide efficient computations appears to be the granularity of the tasks. Particularly important is the granularity in the distributed solver. In all cases of the complexity estimates, it has been the performance of the distributed solver which ultimately resulted in deteriorated speedups. Partitioning schemes which minimize the number of interprocess boundary nodes should exhibit the best performance, since in this case, most of the work will be done in parallel and only a small system of equations will need to be solved globally. In many cases, with such a small system it may be more efficient (in terms of execution time and programming effort) to compute the solution in a single process.

The “raw” speedups exhibited in the examples, although not strictly superlinear, are real. The distributed substructuring method serves both as a convenient approach for parallelization, while at the same time offering the computational advantages of working with smaller matrices. Thus, the superlinear behavior we first observed is not an unrealistic outcome for an actual problem, as we shall see in the next section.

Finally, we note that this method is scalable in the sense that we should be able to handle larger problems by distributing them over more processors. We have seen with the complexity estimates that as the problem size increases,
it is possible to use a larger number of processes with greater efficiency. However, we should avoid the temptation of utilizing too many processors when they are not justified by the size of the problem. The analyses have shown that in some cases, adding more processors will actually create so much overhead that execution times increase.

4.4 Testing the Approach

Testing of the substructuring method was performed on a three dimensional version of the Hebert problem (see Section 2.3.2) utilizing the static mesh of Figure 4.10. To verify correct solutions on three dimensional meshes and in parallel implementations, we performed the following actions

1. Ran the problem in two dimensions on the existing serial code, using the two dimensional static mesh shown in Figure 4.13. Using the parameters from Hebert’s first test run, we obtained solutions which we could use as benchmarks for the testing of the parallel and three dimensional solutions. Results for $t = 2.0$ and $t = 3.7$ minutes are depicted graphically in Figures 4.14 – 4.17.

2. Repeated the problem with the distributed version of the program, obtaining exactly the same results as in the previous step.

3. Ran the problem on the three dimensional mesh using the serial code. Results compared very closely with those of the two dimensional tests. The three dimensional test required a stricter convergence criteria than the two dimensional tests, creating a slightly different time-stepping sequence (which is adaptive). This prohibited us from comparing answers
Table 4.11: Timer results for distributed Hebert problem.

<table>
<thead>
<tr>
<th>P</th>
<th>Ti (sec)</th>
<th>T_{distr} (sec)</th>
<th>T_{distr} St S_{distr}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>236</td>
<td>237 236 236</td>
<td>238 236 237 236</td>
</tr>
<tr>
<td></td>
<td></td>
<td>236</td>
<td>236 238 237 236</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>34 34 34</td>
<td>35 35 35 34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>34 35 34 34</td>
<td>34 35 34 35 34 34</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>12 12 12</td>
<td>12 14 14 14 14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 12 12 12</td>
<td>12 14 14 14 14 14</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>6 6 6 6</td>
<td>8 9 9 9 10 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 6 6 6 6 6</td>
<td>6 6 6 6 6 6 6 6 6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4 4 4 4</td>
<td>9 12 13 11 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 4 4 4 4 4</td>
<td>4 4 4 4 4 4 4 4 4 4</td>
</tr>
</tbody>
</table>

exactly, but it was judged that behavior in the two and three dimensional cases was identical, with breakthrough occurring at the same time. This is to be expected, since the three dimensional problem only exhibits movement in two dimensions.

4. Repeated the three dimensional problem with the distributed version of the program, obtaining exactly the same results as in the previous step. Timing results are presented below.

In all tests, we printed the wall clock time at key points in the computation, enabling us to easily determine the amount of time required for a single linearization iteration. For each test run, we recorded the time required (in seconds) for the first iteration in each of the first four time steps, using the average of this value to represent the timing. In Table 4.11 we display the collected data and computations for “raw” speedup, where \(T_{i}\) and \(T_{\text{distr}}\) are the times required for each of the four iterations using a serial solver and distributed solver, respectively. The averages are denoted as \(\bar{T}_{i}\) and \(\bar{T}_{\text{distr}}\), and the raw speedups are denoted as \(S_{i}\) and \(S_{\text{distr}}\). The raw speedups of the actual and theoretical (see Tables 4.7 and 4.8) analyses are plotted together in Figures 4.18 and 4.19 for comparison.
Figure 4.13: Two dimensional static mesh used for Hebert problem.
Figure 4.14: Computed saturation contours at \( t = 2.0 \) min. for Hebert run # 1.

Figure 4.15: Computed saturation profile at \( t = 2.0 \) min. for Hebert run # 1.
Figure 4.16: Computed saturation contours at $t = 3.1$ min. for Hebert run # 1.

Figure 4.17: Computed saturation profile at $t = 3.7$ min. for Hebert run # 1.
Figure 4.18: Comparison of actual and theoretical raw speedups for Hebert problem with serial solver.

Figure 4.19: Comparison of actual and theoretical raw speedups for Hebert problem with distributed solver.
In Section 4.3.2, we stated that partitioning the problem domain and employing the substructuring method on a single processor would provide us with superlinear improvements in execution time, due to the smaller matrices which would be used. Thus, a true measure of the benefits of parallelism would come from comparing the execution times of a parallel implementation and an implementation which utilizes an identical partitioning and substructuring scheme on a single processor. Since we have not implemented any code which would perform a general partitioning and substructuring method on a single processor, we have no means for determining the “real”, or adjusted speedup of the actual test problems. Instead, we compare the theoretical “raw” speedups of Section 4.3.2 with the actual raw speedups calculated from the timing results of the test problems, and this allows us to arrive at some conclusions. As is often the case, our actual results do not meet the theoretical expectations, as in all cases the actual speedups were less than those predicted. The most probable reason for this is a gross under-estimation of the communication cost parameter, $c_{dp}$. If we look at the complexity estimates for this problem (Table 4.8), we see that for the case of $P = 2$, formation of the Schur complement should dominate the time complexity, giving us perfect speedup since the task is embarrassingly parallel. The fact that we are not achieving this speedup implies that the Schur complement construction is not dominating the complexity. Coupled with the fact that the distributed solver - which is communication intensive - performs poorly in relation to the serial solver, we surmise that the communication costs are much higher than expected. It was stated earlier that our estimates of complexity were crude, and this is where it seems to hold true. Future revisions of the complexity formulas should incorporate a more
realistic indicator of communication costs, including such factors as network contention and message initiation time. Crandall and Quinn [11, 10] discuss communication costs over networks in detail, providing equations which describe the relationships between these factors. In addition, Clement and Quinn [9] present an interesting approach which predicts the performance of a parallel algorithm at compile time, based on these factors. Both of these approaches may prove to be valuable when incorporated into our work.

We conclude this section by comparing the performance of the highly optimized serial code with the parallel code. For the same problem, the serial code performed a single iteration of the linearization loop in approximately sixteen seconds, a time which the parallel code can beat only if it uses three or more processors. There are two major factors which most likely contribute to this disparity. First, performing an inversion on an $N \times N$ matrix, as is required in the substructuring method, requires $2N^3$ operations, whereas the solution to an $N \times N$ set of equations only requires $\frac{2}{3}N^3$ operations [44]. This alone accounts for a degradation in performance by a factor of three. Also, the serial code employs a canned solver (ESSL) which takes advantage of the sparsity in the system matrix. This produces approximately a factor of ten improvement over standard, direct solution methods. Thus, the disparity in performance between the serial and parallel codes is not unexpected. For larger problems, we expect to see substantial improvements when utilizing parallel code simply because the problems will be too big for a single machine to handle efficiently. Finally, for smaller problems, such as the one we have tested here, the matrix inversion process can probably be optimized, and this is a potential area for future work.
4.5 Conclusion

In this chapter, we have described, analyzed, and implemented a method for the parallel solution of finite element problems. Attractive features of this method are its flexibility and ease of implementation. Although we have limited our implementation to simple elements and meshes, the approach is well-suited to any mesh. In addition, the approach is not limited to finite element methods, even allowing for the coupling of different solution methods. For example, it would be useful to incorporate a finite element solution for localized regions of a hydrocarbon reservoir, while employing finite difference methods for other regions. This can be accomplished in a conceptually simple manner by partitioning the problem domain into finite difference and finite element sections, then coupling the regions through the global boundary node system. Of course, this can all be accomplished in parallel as described above.

Since most of the potential applications for this approach are large-scale, it would be wise to focus future improvements on the solution of the global boundary node system. This appears to be the limiting factor as we increase the number of processors being utilized. Thus, a more efficient distributed solver would enable us to make better use of a large number of processors. Possible approaches to this problem include methods such as QR factorization, which requires twice as many operations as standard Gaussian elimination, but fewer communications [28].

Finally, this approach has currently been implemented only on static meshes. Since the substructuring method will work with any mesh, an efficient approach to dynamically modify a distributed mesh would provide the benefits of both parallelism and adaptive methods. Naturally, one key issue is that
of load balancing meshes which are constantly changing. In simple problems such as the Hebert problem, it may be possible to maintain an acceptable level of load balancing without moving partition boundaries. This would make implementation of an adaptive method rather simple.

The most straightforward approach to implementing dynamic meshes would be to keep an identical copy of the global mesh in each process. Each process would redundantly modify the entire mesh (or perhaps only a subset of the mesh which we could insure contained all of the modifications needed by the local process), then make the appropriate modifications to its own data based on the newly modified mesh. This approach avoids the complicated data structures and communications routines which might be needed to coordinate mesh modifications occurring near interprocess boundaries.

A simpler approach for incorporating adaptive methods would be to let a single master process store a copy of the global mesh. Periodically, all processes would consolidate their solutions and send them to the master process, which would modify the mesh based on the current solutions, then would send the appropriate mesh data back to the processes. The simplicity of this approach lies in the fact that all of the routines needed for this already exist! The current implementation periodically consolidates solutions so that they may be directed to I/O devices. In addition, we already have routines for partitioning an existing mesh among the processes and setting up the pertinent data structures - this action occurs at the beginning of the program when the initial mesh is distributed. Thus, to utilize this approach, we simply repeat the same actions every time the mesh is modified.
Chapter 5

Summary and Future Directions

5.1 Summary

This dissertation has described the development, analysis, and implementa­
tion of several key components needed for the high performance computer
modelling of geophysical processes. Beginning in Chapter 2, we described our
methodology for the accurate solution of multiphase flow problems by utiliza­
tion of adaptive finite element methods to produce fine mesh resolution where
most needed. This approach was tested by comparing the results of com­
puter simulations with laboratory experiments. The work of Chapter 2 may
be extended to three dimensions, and in Chapter 3 we discussed our formula­
tion of a new finite element which would facilitate the construction of adaptive
mesh techniques in three dimensions. Finally, parallelization of a finite element
methodology using static meshes was discussed in Chapter 4, with an analysis
of theoretical performance, and a test of the implementation on a cluster of
workstations.

5.2 Future R & D

Although a complete, robust model is still a long-term future goal, many
problems have been investigated and solved. A framework has been constructed
on which future work may be based by extension and improvement of existing
techniques. We have found that the general methodology is sound and warrants

138
further development. Numerous opportunities exist for future research and development, and the remainder of this dissertation is devoted to the discussion of several key extensions and improvements which would aid in transforming this work from a predominantly experimental approach to a production environment suitable for distribution to the scientific community.

5.2.1 Dynamic Mesh Modification

The extension of this work will not be complete until a dynamic mesh modification scheme has been implemented for three dimensional meshes. The methodology would be similar to that of the two dimensional case discussed in Chapter 2. However, since the two dimensional code was "exploratory" in nature, efficiency was not a prime consideration. For the problems we have been modelling, the storage and cpu time required by the mesh modification procedures are minimal. If we are successful in implementing iterative solution techniques for large systems of equations, we will be able to solve much larger problems in a shorter period of time. In this case, the mesh modification procedures may begin to dominate the computations in terms of cpu time. For this reason, it would be useful to study the existing approach closely, looking for more efficient data structures and algorithms.

Related to the mesh modification procedures is the implementation of a node numbering scheme which would minimize the bandwidth of the global system matrices, allowing us to incorporate solvers which use much less memory and generate solutions in a shorter period of time. Since element and node renumbering is necessary after any modification, this would be a logical point for incorporation. A possible approach at bandwidth reduction is discussed in [46].
Improved mesh modification routines should also attempt to address the issues of parallelism, particularly the problems of load balancing. "Quick and dirty" approaches were discussed in Chapter 4, but a robust methodology will need to consider the fact that load balances may change drastically as regions of refinement move through the problem domain. This remains an open area of research, but several approaches are discussed in [16, 14].

5.2.2 Mesh Generation and Visualization

Three steps are necessary for a complete finite element analysis of a physical problem - preprocessing, computation, and postprocessing. The work discussed in this dissertation has focused almost exclusively on the computation step. Although this step will typically dominate the requirements for memory and cpu time, and should be optimized as much as possible, the software is not adequate for practical applications without the ability to generate finite element meshes. Up to now, only simple meshes have been utilized, but the construction of these has often been a long, tedious, and error prone process. In three dimensional geophysical problems, it is necessary to model problems with irregular boundaries and heterogeneous properties, making the process of manual mesh generation prohibitive. Certainly, there will be no interest in this software if users must construct such meshes by hand! Envisioned is a user-friendly, interactive approach, enabling a finite element modeller to construct arbitrary meshes unique to the problem. The approach should have the facility for specifying various physical properties and boundary conditions throughout the problem domain. This may be a difficult task, as the software would need to derive many of the relationships needed for dynamic mesh modification, but construction of such software is an essential step towards modelling anything
but academic problems. Numerous commercial packages have such mesh generation facilities, based on CAD/CAM technology and it may be possible to locate and modify the source code of public domain software to accomplish some of these goals.

Postprocessing of model results is also an essential step, but probably requires little effort. Numerous software packages exist for the display of two and three dimensional data, typically requiring as input a uniform grid of data points, including coordinates and solution values - graphics depicting the saturation solutions in Chapters 2 and 4 were generated with such software. The problem of generating this data lies in mapping a uniform grid of points to the irregular finite element mesh. For each point in the uniform grid, it is necessary to determine which element of the finite element mesh it lies in. Once determined, it is a trivial task to calculate the solution at this point, utilizing the interpolating shape functions of the element. The problem of determining which element a given point lies in is a computational geometry problem, discussed in [43].

5.2.3 Testing of More Realistic Problems

Although the Hebert problem (see Chapter 2) has provided us with a well-controlled experiment for testing of the model, results obtained here may not be indicative of results for more realistic problems. The pressure gradients exhibited in Hebert's experiment are extremely small, just barely above the natural pressure gradient due to gravity which exists in a "no-flow" scenario. This results in otherwise minor oscillations corrupting the solution. Although the source of all the oscillations has probably not been determined, we suspect several factors. First, as discussed in Chapter 2, several of the flux terms
generated in the application of Galerkin’s method to the equations of flow were neglected under the assumption that their effects were minimal - in problems with such small pressure gradients, this may not be the case. Thus, it will be necessary to more rigorously handle the “negligible” terms in the equations in order to more accurately model these small-scale problems. A more productive approach would be to begin modelling and comparing results with large-scale problems, for which this methodology was ultimately designed. Numerous “benchmarks” exist in the literature, and matching some of these results will provide even more credibility for our approach.

5.3 Conclusion

The scientific community is sorely in need of useable, efficient, and affordable tools for the simulation of complex geophysical phenomena. It is no longer feasible for the practicing scientist to generate modern simulation tools on state-of-the-art computing equipment with a few lines of simple Fortran statements. Rather, the hardware technology available today requires a much greater depth of knowledge in the computational aspects of deriving model results. Thus, in order for some of these large-scale problems to be solved, it is imperative that a multidisciplinary approach be taken, fusing the contributions of computer scientists, mathematicians, and scientists from relevant disciplines to achieve solutions. The work which led to this dissertation represents an initial attempt at pursuing such research.
Bibliography


Appendix

A.1 Modified Quadrature for Discontinuous Integrand

\[ \int_{-1}^{1} F(\xi) d\xi \approx \sum_{i=1}^{4} w_i F(\xi_i) \]  

(A.1)

When \( F(\xi) \) is a cubic polynomial (or lower order) in \( \xi \), and is discontinuous at \( \xi = 0 \), Equ. A.1, using the gauss points and weights from Table A.1, will integrate exactly. Refer to [26] for more detailed explanation.

A.2 Shape Function Derivatives

A.2.1 Corner Nodes

\[ \frac{\partial \hat{N}_1}{\partial \xi} = \frac{1}{8} (-1 + \eta)(1 + \zeta) \]  

(A.2)

\[ \frac{\partial \hat{N}_1}{\partial \eta} = \frac{1}{8} (-1 + \xi)(1 + \zeta) \]  

(A.3)

\[ \frac{\partial \hat{N}_2}{\partial \xi} = \frac{1}{8} (1 - \eta)(1 + \zeta) \]  

(A.4)

\[ \frac{\partial \hat{N}_2}{\partial \eta} = -\frac{1}{8} (1 + \xi)(1 + \zeta) \]  

(A.5)

\[ \frac{\partial \hat{N}_2}{\partial \xi} = -\frac{1}{8} (1 - \xi)(1 + \zeta) \]  

(A.6)

Table A.1: Gauss points and weights for modified quadrature rules.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \xi_i )</th>
<th>( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \frac{1}{2}(\frac{1}{\sqrt{3}} - 1) )</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>( -\frac{1}{2}(\frac{1}{\sqrt{3}} + 1) )</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>( -\xi_2 )</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>( -\xi_1 )</td>
<td>0.5</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
\frac{\partial \tilde{N}_2}{\partial \zeta} &= \frac{1}{8}(1 + \xi)(1 - \eta) \\
\frac{\partial \tilde{N}_2}{\partial \xi} &= \frac{1}{8}(1 + \eta)(1 + \xi) \\
\frac{\partial \tilde{N}_3}{\partial \eta} &= \frac{1}{8}(1 + \xi)(1 + \zeta) \\
\frac{\partial \tilde{N}_3}{\partial \zeta} &= \frac{1}{8}(1 + \eta)(1 + \xi) \\
\frac{\partial \tilde{N}_4}{\partial \xi} &= \frac{1}{8}(1 - \xi)(1 + \eta) \\
\frac{\partial \tilde{N}_4}{\partial \eta} &= \frac{1}{8}(1 - \zeta)(1 + \xi) \\
\frac{\partial \tilde{N}_5}{\partial \xi} &= \frac{1}{8}(1 - \eta)(-1 + \zeta) \\
\frac{\partial \tilde{N}_5}{\partial \eta} &= \frac{1}{8}(1 - \zeta)(1 - \eta) \\
\frac{\partial \tilde{N}_6}{\partial \xi} &= \frac{1}{8}(1 - \eta)(1 - \zeta) \\
\frac{\partial \tilde{N}_6}{\partial \eta} &= \frac{1}{8}(1 + \xi)(1 - \eta) \\
\frac{\partial \tilde{N}_7}{\partial \xi} &= \frac{1}{8}(1 + \eta)(1 - \zeta) \\
\frac{\partial \tilde{N}_7}{\partial \eta} &= \frac{1}{8}(1 + \zeta)(1 - \eta) \\
\frac{\partial \tilde{N}_8}{\partial \xi} &= \frac{1}{8}(1 + \xi)(1 + \eta) \\
\frac{\partial \tilde{N}_8}{\partial \eta} &= \frac{1}{8}(1 - \xi)(1 - \zeta)
\end{align*}
\]
\[ \frac{\partial \tilde{N}_8}{\partial \zeta} = \frac{1}{8}(-1 + \xi)(1 + \eta) \] (A.25)

**A.2.2 Mid-edge Nodes**

\[ \xi_o = \begin{cases} 
1 & \text{for } \zeta > 0 \\
-1 & \text{for } \zeta < 0 
\end{cases} \] (A.26)

\[ \eta_o = \begin{cases} 
1 & \text{for } \eta > 0 \\
-1 & \text{for } \eta < 0 
\end{cases} \] (A.27)

\[ \zeta_o = \begin{cases} 
1 & \text{for } \zeta > 0 \\
-1 & \text{for } \zeta < 0 
\end{cases} \] (A.28)

\[ \frac{\partial \tilde{N}_9}{\partial \xi} = \frac{1}{4}\xi_o(-1 + \eta)(1 + \zeta) \] (A.29)

\[ \frac{\partial \tilde{N}_9}{\partial \eta} = \frac{1}{4}(-1 + |\xi|)(1 + \zeta) \] (A.30)

\[ \frac{\partial \tilde{N}_9}{\partial \zeta} = \frac{1}{4}(1 - |\xi|)(1 - \eta) \] (A.31)

\[ \frac{\partial \tilde{N}_{10}}{\partial \xi} = -\frac{1}{4}\xi_o(1 + \eta)(1 + \zeta) \] (A.32)

\[ \frac{\partial \tilde{N}_{10}}{\partial \eta} = -\frac{1}{4}(1 - |\xi|)(1 + \zeta) \] (A.33)

\[ \frac{\partial \tilde{N}_{10}}{\partial \zeta} = \frac{1}{4}(1 - |\xi|)(1 + \eta) \] (A.34)

\[ \frac{\partial \tilde{N}_{11}}{\partial \xi} = \frac{1}{4}\xi_o(1 + \eta)(-1 + \zeta) \] (A.35)

\[ \frac{\partial \tilde{N}_{11}}{\partial \eta} = \frac{1}{4}(1 - |\xi|)(1 - \zeta) \] (A.36)

\[ \frac{\partial \tilde{N}_{12}}{\partial \zeta} = \frac{1}{4}\xi_o(1 - \eta)(-1 + \zeta) \] (A.37)

\[ \frac{\partial \tilde{N}_{12}}{\partial \eta} = \frac{1}{4}(1 - |\xi|)(-1 + \zeta) \] (A.38)
\[
\begin{align*}
\frac{\partial N_{12}}{\partial \zeta} &= \frac{1}{4}(-1 + |\xi|)(1 - \eta) \quad \text{(A.40)} \\
\frac{\partial N_{13}}{\partial \xi} &= \frac{1}{4}(-1 + |\eta|)(1 + \zeta) \quad \text{(A.41)} \\
\frac{\partial N_{13}}{\partial \eta} &= \frac{1}{4}\eta(1 - \xi)(1 + \zeta) \quad \text{(A.42)} \\
\frac{\partial N_{14}}{\partial \xi} &= \frac{1}{4}(1 - |\xi|)(1 + \zeta) \quad \text{(A.43)} \\
\frac{\partial N_{14}}{\partial \eta} &= \frac{1}{4}(1 + |\eta|)(1 + \xi) \quad \text{(A.44)} \\
\frac{\partial N_{14}}{\partial \xi} &= \frac{1}{4}(1 + |\xi|)(1 - |\eta|) \quad \text{(A.45)} \\
\frac{\partial N_{15}}{\partial \xi} &= \frac{1}{4}(1 - |\xi|)(1 - |\eta|) \quad \text{(A.46)} \\
\frac{\partial N_{15}}{\partial \eta} &= \frac{1}{4}(1 - \xi)(1 - \zeta) \quad \text{(A.47)} \\
\frac{\partial N_{16}}{\partial \eta} &= \frac{1}{4}(1 + \xi)(-1 + |\eta|) \quad \text{(A.48)} \\
\frac{\partial N_{16}}{\partial \xi} &= \frac{1}{4}(-1 + |\eta|)(1 - \zeta) \quad \text{(A.49)} \\
\frac{\partial N_{17}}{\partial \xi} &= \frac{1}{4}(1 - \eta)(-1 + |\xi|) \quad \text{(A.50)} \\
\frac{\partial N_{17}}{\partial \eta} &= \frac{1}{4}(1 - \xi)(-1 + |\xi|) \quad \text{(A.51)} \\
\frac{\partial N_{18}}{\partial \xi} &= \frac{1}{4}(1 - \eta)(1 - |\zeta|) \quad \text{(A.52)} \\
\frac{\partial N_{18}}{\partial \eta} &= \frac{1}{4}(1 + \xi)(-1 + |\zeta|) \quad \text{(A.53)} \\
\frac{\partial N_{19}}{\partial \xi} &= \frac{1}{4}(1 - \zeta)(-1 + |\xi|) \quad \text{(A.54)} \\
\frac{\partial N_{19}}{\partial \eta} &= \frac{1}{4}\zeta(1 - \xi)(1 - \eta) \quad \text{(A.55)} \\
\frac{\partial N_{20}}{\partial \xi} &= \frac{1}{4}(1 - \eta)(1 - |\zeta|) \quad \text{(A.56)} \\
\frac{\partial N_{20}}{\partial \eta} &= \frac{1}{4}(1 + \xi)(-1 + |\zeta|) \quad \text{(A.57)}
\end{align*}
\]
\[
\frac{\partial \tilde{N}_{18}}{\partial \zeta} = -\frac{1}{4}\zeta_{o}(1 + \xi)(1 - \eta) \quad \text{(A.58)}
\]
\[
\frac{\partial \tilde{N}_{19}}{\partial \xi} = \frac{1}{4}(1 + \eta)(1 - |\zeta|) \quad \text{(A.59)}
\]
\[
\frac{\partial \tilde{N}_{19}}{\partial \eta} = \frac{1}{4}(1 + \xi)(1 - |\zeta|) \quad \text{(A.60)}
\]
\[
\frac{\partial \tilde{N}_{19}}{\partial \bar{\zeta}} = -\frac{1}{4}\zeta_{o}(1 + \xi)(1 + \eta) \quad \text{(A.61)}
\]
\[
\frac{\partial \tilde{N}_{20}}{\partial \xi} = \frac{1}{4}(1 + \eta)(-1 + |\zeta|) \quad \text{(A.62)}
\]
\[
\frac{\partial \tilde{N}_{20}}{\partial \eta} = \frac{1}{4}(1 - \xi)(1 - |\zeta|) \quad \text{(A.63)}
\]
\[
\frac{\partial \tilde{N}_{20}}{\partial \bar{\zeta}} = -\frac{1}{4}\zeta_{o}(1 - \xi)(1 + \eta) \quad \text{(A.64)}
\]

### A.2.3 Mid-face Nodes

\[
\frac{\partial N_{21}}{\partial \xi} = \frac{1}{2}\xi\zeta_{o}(1 - \eta)(-1 + |\zeta|) \quad \text{(A.65)}
\]
\[
\frac{\partial N_{21}}{\partial \eta} = \frac{1}{2}(1 - |\xi|)(-1 + |\zeta|) \quad \text{(A.66)}
\]
\[
\frac{\partial N_{21}}{\partial \bar{\zeta}} = -\frac{1}{2}\zeta_{o}(1 - |\xi|)(1 - \eta) \quad \text{(A.67)}
\]
\[
\frac{\partial N_{22}}{\partial \xi} = \frac{1}{2}\xi\zeta_{o}(1 + \eta)(1 + \xi)(1 - |\zeta|) \quad \text{(A.68)}
\]
\[
\frac{\partial N_{22}}{\partial \eta} = \frac{1}{2}(1 - |\xi|)(1 - |\zeta|) \quad \text{(A.69)}
\]
\[
\frac{\partial N_{22}}{\partial \bar{\zeta}} = -\frac{1}{2}\zeta_{o}(1 - |\xi|)(1 + \eta) \quad \text{(A.70)}
\]
\[
\frac{\partial N_{23}}{\partial \xi} = \frac{1}{2}(1 - |\eta|)(-1 + |\zeta|) \quad \text{(A.71)}
\]
\[
\frac{\partial N_{23}}{\partial \eta} = -\frac{1}{2}\zeta_{o}(1 - \xi)(1 - |\eta|) \quad \text{(A.72)}
\]
\[
\frac{\partial N_{23}}{\partial \bar{\zeta}} = \frac{1}{2}(1 - |\eta|)(1 - |\zeta|) \quad \text{(A.73)}
\]
\[
\frac{\partial N_{24}}{\partial \xi} = -\frac{1}{2}\eta\zeta_{o}(1 + \xi)(1 - |\zeta|) \quad \text{(A.74)}
\]
\[
\frac{\partial N_{24}}{\partial \eta} = -\frac{1}{2}\eta_{o}(1 + \xi)(1 - |\zeta|) \quad \text{(A.75)}
\]
\[ \frac{\partial N_{24}}{\partial \zeta} = -\frac{1}{2} \zeta \phi(1 + \xi)(1 - |\eta|) \]  
(A.76)

\[ \frac{\partial N_{25}}{\partial \xi} = \frac{1}{2} \xi \phi(-1 + |\eta|)(1 - \zeta) \]  
(A.77)

\[ \frac{\partial N_{25}}{\partial \eta} = -\frac{1}{2} \eta \phi(1 - |\xi|)(1 - \zeta) \]  
(A.78)

\[ \frac{\partial N_{26}}{\partial \zeta} = \frac{1}{2} (1 - |\xi|)(-1 + |\eta|) \]  
(A.79)

\[ \frac{\partial N_{26}}{\partial \xi} = \frac{1}{2} \xi \phi(-1 + |\eta|)(1 + \zeta) \]  
(A.80)

\[ \frac{\partial N_{26}}{\partial \eta} = -\frac{1}{2} \eta \phi(1 - |\xi|)(1 + \zeta) \]  
(A.81)

\[ \frac{\partial N_{28}}{\partial \zeta} = \frac{1}{2} (1 - |\xi|)(1 - |\eta|) \]  
(A.82)
Don Morton was born in Fargo, North Dakota in 1959. Graduating from high school in 1976, he pursued educational and vocational interests until his enlistment in the U.S. Air Force in 1980. In 1988 he received a B.S. degree in Computer Science from the College of Great Falls, Montana, and left his military career to pursue graduate studies in Computer Science at Louisiana State University beginning in 1989. His research interests are in the computer modelling and simulation of complex physical processes, utilizing state-of-the-art hardware and software technology to aid in mankind's quest for a more thorough understanding of the universe in which we live.
Candidate: Donald J. Morton, Jr.

Major Field: Computer Science

Title of Dissertation: An Adaptive Finite Element Methodology for the High-Performance Computer Simulation of Multiphase Flow Processes

Approved:

John M. Tolle
Major Professor and Chairman

George F. Vermeulen
Dean of the Graduate School

EXAMINING COMMITTEE:

Robert A. Mullin

Alan T. Bourgeois

J. D. Jones

Donald R. Flett

Burl M. Paskley

Date of Examination:

April 7, 1994