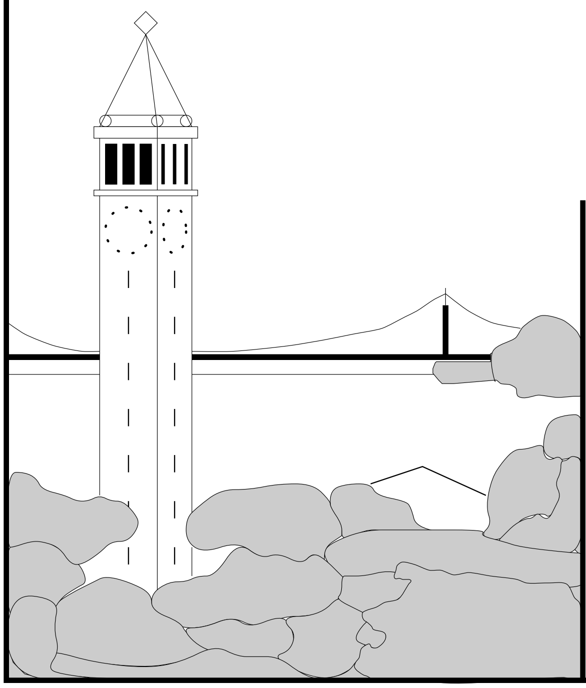
Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations

Mark Francis Adams



Report No. UCB/CSD-99-1033

January 1999

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations

by

Mark Francis Adams

B.A. (University of California, Berkeley) 1983

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Civil Engineering

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Robert L. Taylor, Co-chair
Professor James W. Demmel, Co-chair
Professor Gregory L. Fenves
Professor Katherine Yelick

1998

The dissertation of Mark Francis Adams is approved:

Co-chair Date

Co-chair Date

Date

Date

University of California, Berkeley

1998

Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations

Abstract

Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations

by

Mark Francis Adams
Doctor of Philosophy in Engineering - Civil Engineering

University of California, Berkeley

Professor Robert L. Taylor, Co-chair

Professor James W. Demmel, Co-chair

The finite element method has grown, in the past 40 years, to be a popular method for the simulation of physical systems in science and engineering. The finite element method is used in a wide array of industries. In fact just about any enterprise that makes a physical product can, and probably does, use finite element technology. The success of the finite element method is due in large part to its ability to allow the use of accurate formulation of partial differential equations (PDEs), on arbitrarily general physical domains with complex boundary conditions. Additionally, the rapid growth in the computational power available in todays computers - for an ever more affordable price - has made finite element technology more accessible to a wider variety of industries and academic disciplines.

As computational resources allow people to produce ever more accurate simulation of their systems - allowing for the more efficient design and safety testing of everything from automobiles to nuclear weapons to artificial knee joints - all aspects of the finite element simulation process are stressed. The largest bottleneck in the growth in the scale of finite element applications is the linear equation solver used in implicit time integration schemes. This is due to the fact that the direct solution methods - popular in the finite element community as they are efficient, easy to use, and relatively unaffected by the underlying PDE and discretization - do not scale well with increasing problem size.

The scale of problems that are now becoming feasible demand that iterative methods be used. The performance issues of iterative methods is very different from those of direct methods, as their performance is highly sensitive to the underlying PDE and dis-

cretization; the construction of robust iterative methods for finite element matrices is a hard problem which is currently a very active area of research. We discuss the iterative methods commonly used today, and show that they can all be characterized as methods that solve problems efficiently by projecting the solution to a series of subspaces. The goal of iterative method design, and indeed of finite element method design, is to select a series of subspaces that solves problems "optimally"; solvers try to minimize solution costs and finite element formulations try to optimize accuracy of the solution. The subspaces used in multigrid methods are highly effective in minimizing solution costs - particularly on large problems. Multigrid is known to be *the* most effective solution method for some discretized PDEs, however its effective use on unstructured finite element meshes is an open problem, and constitutes the theme of this study.

The main contribution of this dissertation is the algorithmic development and experimental analysis of robust and scalable techniques for the solution of the sparse, ill-conditioned matrices that arise from finite element simulation in 3D continuum mechanics. We show that our multigrid formulations are effective in the linear solution of systems with large jumps in material coefficients, for problems with realistic mesh configuration and geometries (including poorly proportioned elements), and for problems with poor "geometric" conditioning as is commonplace in structural engineering. We show that the these methods can be used effectively within nonlinear simulations via Newton's method. We solve problems with more than sixteen million degrees of freedom and parallel solver efficiency of about 60% on 512 processors of a Cray T3E. We also show that our methods can be adapted and extended to the indefinite matrices that arise in the simulation of problems with constraints, namely contact problems, formulated with Lagrange multiplier.

Professor Robert L. Taylor
Dissertation Committee Co-chair

Professor James W. Demmel
Dissertation Committee Co-chair

To my best friend Nighat

for constant support for many years before my graduate work and throughout the many years of this work

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to Professors R.L. Taylor and James Demmel for sharing their enthusiasm for, and expertise in, their respective fields - as well as for helping to make my experience as a graduate student a rewarding and enriching one.

I would like to express my gratitude to Professor Taylor for his support and constant enthusiasm for my work that has made my years in the graduate program for the Civil Engineering department a gratifying experience. I have learned much from Professor Taylor's extensive knowledge, gained in his 40 years as a researcher and a practitioner, in the field of Computational Mechanics.

I would like to thank Professor Demmel for his energetic support, that has been invaluable to my experience and education here at Berkeley. The author has benefited greatly from Professor Demmel's expertise, guidance and introduction to the fields of Scientific Computing, Computer Science, and Applied Mathematics.

I would like to express my gratitude to the following

- Professor Taylor for providing and maintaining FEAP, without which this work would have been grossly inferior [36].

- The PETSc team [10]: Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith for providing a great product, without which this work would have been grossly inferior. And I especially want to thank Barry Smith for his tireless support of PETSc, for his concise theory manual "Domain Decomposition" [78], and for informing me (in 1995 with Tony Chan) of the basic algorithm with which I work, and without which I would not have a life.

- George Karypis for providing a fast full features state-of-the-art mesh partitioner: METIS and ParMetis [52].

- Jonathan Shewchuk for providing fast robust geometric predicates [73].

- Argonne National Laboratory for the use of their IBM SP which was invaluable in the early development work for this dissertation.

- Steve Ashby for his support of this work and for providing me with the opportunity to work for a summer in the intellectually stimulating environment of CASC and LLNL.

- Juliana Hsu, for providing me with test problems from the ALE3D group at LLNL, and the entire ALE3D group, and the linear solvers group in CASC, for many interesting discussions during my summer at the lab.

- Lawrence Livermore National Laboratory for providing access to its computing systems and to the staff of Livermore Computing for their support services.

- Lawrence Berkeley National Laboratory for the use of their Cray T3E, and their very helpful support staff. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Energy Research of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

- Eric Kasper for providing me with the original FEAP source for test problem $P1$ in chapter 9.

This research was supported in part by the Department of Energy under DOE Contract No. W-7405-ENG-48 and DOE Grant Nos. DE-FG03-94ER25219, DE-FG03-94ER25206, and DE-FC03-98ER25351, the National Science Foundation under NSF Cooperative Agreement No. ACI-9619020, Infrastructure Grant No. CDA-9401156, and Grant No. ASC-9313958, the Defense Advanced Research Projects Agency under DARPA Contract No. F30602-95-C-0014 and Grant No. DE-FG03-94ER25206, gifts from the IBM Shared University Research Program, the California State MICRO program, Sun Microsystems, and Intel, and funds from the California State MICRO program. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Chapter 1

Dissertation summary

1.1 Introduction

The finite element method has proven to be a popular spatial discretization technique in the simulation of complex physical systems in many areas of science and engineering. The finite element method commonly provides for the spatial discretization of the domain for a partial differential equation (PDE); time discretizations or time integrators are also required. Time integrators fall into two main categories: implicit and explicit. Explicit schemes are preferable for problems in which the interval in the time discretization is short relative to the spatial discretization. Implicit methods are more compute intensive than explicit methods, however they have superior stability properties and are thus preferable when the time step required to capture the behavior of the system is large relative the spatial discretization. Implicit methods are expensive because they require that a *sparse* operator be linearized and its inverse (a dense operator) applied to a vector, rather than only the sparse operator itself - this is much more difficult to compute. The cost of applying the inverse of a finite element operator will dominate the total cost of the finite element method when an implicit time integrator is in use on large scale problems - the effective solution of this problem is the subject of this dissertation.

1.2 The finite element method

The finite element method finds the "optimal" solution of a partial differential equation in a user provided function space - that is it calculates *a* solution in the span of a provided set of functions whose error is orthogonal to this subspace in some inner product. This is often calculated with an *orthogonal projection* and is the basis of classical methods such as Galerkin and Rayleigh-Ritz methods. The key aspect of the finite element method, that has lead to its success, is the use of piecewise continuous (low order) polynomials first introduced by Courant in the 1940s [25] and independently by the engineering community in the 1950s [4]. Piecewise continuous polynomials are effective because they may be automatically constructed by "meshing" a complex domain into simple *elements* or polyhedra, and have compact support. Finite element *basis* functions can then be constructed, from this mesh, on the elements.

The finite element method is in part successful because it allows for a very hard problem to be effectively decomposed into separate and well defined disciplines. Broadly speaking these disciplines are

- Mesh generation
- Element formulation
- Non-linear and transient solution algorithms
- Linear equation solvers
- Visualization and post processing

This dissertation focuses on the effective construction of linear equation solvers for finite element method matrices in the high performance computational environments of today.

1.3 Motivation

The dominant costs of conducting a finite element analysis, once the software has been developed, is the construction of the model or mesh and the solution of a sparse set of algebraic equations. The solution of this system of equations is generally the dominant cost in an analysis as a single problem often requires the solution of hundreds or thousands of equation sets if dynamic (or transient) analysis is called for. Nonlinear formulations are generally solved with a series of linearized solutions, for instance by a Newton solution scheme, at each time step. Additionally, finite element analyses are usually used in design processes or parametric analyses, thus requiring that many finite element simulations be run with a single mesh or model.

When implicit time discretization methods are in use, the cost of the sparse linear system solve will dominate the cost of the finite element simulation. Furthermore the linear equation solver is one of the hardest parts of a finite element solution to implement efficiently as the problem size, and number of parallel processors increases - therefore scalable linear equation solver technology is of considerable importance to the continued growth in the use of the finite element method.

Direct solution methods based on Gaussian elimination are very popular as they are robust and effective for moderate sized problems. Accurate finite element simulations often require that a fine discretization be used, thus necessitating the solution of large systems of equations. Also, in some cases, it may simply be more economical to use large, fast, and ever cheaper computers to solve a large problem where in the past highly skilled engineers have had to painstakingly assemble smaller analyses and combine their results with intuition to assess the safety of a structure.

The difficulty with direct methods is that for typical finite element problems optimal direct solvers have a time complexity of about n^2 in 3D, where n is the number of degrees of freedom in the system. Direct methods are more applicable to 2D models as the number of degrees of freedom only increases quadratically with the scale of discretization, as opposed to cubicly for 3D models. Additionally the time complexity for 2D problems is about $n^{3/2}$ as 2D problems tend to have much less "fill" introduced during the factorization.

The rather small constants in the complexity of direct methods have allowed them to be superior to iterative methods for the size models that were affordable in the past. But the poor asymptotics of direct methods is now requiring that iterative methods be used as iterative methods have the potential of $O(n)$ time and space complexity.

Domain decomposition methods represent a framework for describing and analyzing optimal iterative methods for solving the sparse matrices from unstructured discretizations of PDEs [78]. Multilevel domain decomposition methods are theoretically optimal preconditioners. Of the family of multilevel domain decomposition methods, multigrid is the most powerful on structured meshes. Multigrid is also ideally suited for finite element problems on unstructured grids, as it uses many of the same mathematical techniques (e.g., orthogonal projections) as well as many of the same tools in its implementation (e.g., mesh generators). In fact, as we show, the core of multigrid is the recursive application of a variationally induced approximation of the "current" problem (grid) - this is also a description of the finite element method. Thus multigrid is, in a sense, the application of the finite element method to itself.

Many of the domains of interest to finite element method practitioners are inherently ill-conditioned problems with large ranges in scale of discretization, complex structures often requiring the use of less than ideally proportioned elements, sharp jumps in material properties or changes in physics. These difficulties are exacerbated by the use of state-of-the-art finite element formulations - e.g., mixed methods for nearly incompressible problems, plasticity formulation to simulate yielding of structural materials, large (finite) deformation elements, and the use of Lagrange multipliers in applying constraints. Many of these formulations result in a loss of positive definiteness in the equations and/or results in equations that are very poorly conditioned. These finite element formulations often severely degrade the performance of iterative methods, thus the serial performance as well as scalability of the solver is of utmost importance. This dissertation presents an effective methods for solving the sparse ill-conditioned systems of equations that arise from large scale (i.e., $10^6 - 10^8$ vertices), state of the art finite element simulations.

1.4 Goals

The exponential growth in the processing power of computers in the last twenty years is enabling scientists and engineers to conduct ever more accurate simulations of their systems. Along with the "pull" of the opportunity to conduct large scale simulations, applications are "pushing" for fast, accurate simulations to design efficient products and to bring them to market quickly because of increasing global competition in manufacturing. Also, testing products in the laboratory is at best a limited method of product testing and in the case of the U.S. government is no longer a politically viable means of insuring the safety of the nation's stockpile of nuclear weapons [6]. Thus, the time and expense of laboratory testing is proving to be increasingly less attractive than computational simulations. As the equation solver is the dominant cost in some large finite element simulations, scalable and robust equations solvers are of critical importance.

The primary requirements for the "next generation" of finite element method equation solvers are:

1. **Complete scalability**. Any method must be completely scalable and completely parallel. This means that the number of iterations required to reduce the error by a constant fraction must be independent of the mesh size; or that the number of

iterations grows polylogarithmically (i.e., polynomial in $\log(n)$) while the cost of each iteration is polylogarithmic in time. The algorithm must also be feasible to implement efficiently, i.e. it must be easily parallelized. All solvers in this class are *multilevel* solvers.

2. **Robustness**. In this context, robustness means that the method is effective on the problems that will be common in industrial practice - i.e., unstructured meshes with a wide range of scales of discretization, with multiple material properties, with widely varying material coefficients, and accurate material constitutions for all classes of industrial applications. We are concerned with solvers for 3D solid mechanics problems only, although the method that we discuss is applicable to problems in fluid mechanics and other areas of physics as well.

3. **Easy of use**. We would like to have a "black box" solver so as to simplify its use with existing finite element codes. This criterion is secondary, in that requiring more data from the user is certainly more palatable than not being able to solve the problem. Thus, we desire to have a minimal interface with the finite element implementation, and require only what is easily available in common finite element codes.

Multigrid is the best known method to date that satisfies the first criterion. Multigrid is an optimal solver of Poisson's equation [27] (discretized with finite element or finite difference methods) at least sequentially; the 3D FFT is competitive in parallel although to our knowledge it is not applicable to problems as general as those that we consider. The parallel time complexity of full multigrid is $O(\log(n)^2)$; although realizing the theoretical complexity on real machines is a challenge, multigrid can scale very well on the large computers of today (chapters 9,10) and in the foreseeable future (§8.3).

The *robustness* of multigrid is necessary for our purposes as large finite element simulations can require a wide variety of finite element formulations, in a variety of applications - many of which produce operators that are very demanding of iterative methods. The multigrid algorithm that we work with [44, 23] offers the well-known advantages of multigrid while maintaining a minimal - though not minimum - solver interface. The goal of having a "black box" solver is theoretically achievable with algebraic multigrid methods - although the only effective algebraic methods that we are aware of require geometric information [83, 19], as does ours. Our method builds on previous work [44, 23] and is an automated method for constructing the coarse grids for standard multigrid algorithms and forms the coarse grid *operators* algebraically. This combined geometric/algebraic approach yields a method that has a similar interface with the finite element implementation as effective algebraic methods; but our approach allows for a more extensive use of geometric information that is quite effective on many classes of problems.

1.5 Dissertation outline

We have organized this dissertation so that it can be read in a *multilevel* fashion; that is, one can read this chapter, the conclusion, and the preamble to each chapter to get a uniform introduction to our work. One can read the introduction in each chapter, in addition, to get a more detailed view of our work. The dissertation is organized as follows:

- Chapter 1 continues with a list of notations and concepts that we use in our work.
- Chapter 2 introduces pertinent background in iterative equation solvers for finite element matrices.
  - Finite element formulations are, in general, not necessary to understand the characteristics of the equation solver, although multigrid is in fact very much related to the finite element method - multigrid can be seen as "the recursive application of the finite element method". Thus a rudimentary introduction to the finite element method proves to be invaluable in understanding the behavior and construction of multigrid methods.
  - Basic iterative methods are introduced as we use some of them as "accelerators", and they lead to domain decomposition methods, one variety of which is multigrid.
  - One level domain decomposition methods are introduced as we use them within our solver, and they serve to introduce many of the basic concepts of projections which we use extensively.
- Chapter 3 discusses multi-level domain decomposition methods in general and multigrid in particular.

  - We introduce the basic issues of multi-level domain decomposition methods in the vein of describing what has come before multigrid, and the alternatives to multigrid.
  - We do not discuss the mathematical details of multigrid in great depth, however we introduce the current state-of-the-art in analyzing multigrid on unstructured grids. This description is not comprehensive but is intended to introduce some basic concepts and proves useful in understanding the performance behavior of multilevel methods.

- Chapter 4 describes the current competitive methods in the field of high performance linear equation solvers for finite element matrices.

- Chapter 5 introduces our multigrid method - we build on earlier work in 2D formulations, extending them to 3D and in parallel. This includes our development of a new maximal independent set algorithm for finite element meshes that, under the PRAM computational model (see 1.7) [41], has $O(1)$ time complexity, and is practical - this is an improvement over the previous best algorithm [50]. This chapter also includes a set of heuristics and methods for applying this multigrid algorithm effectively to complex finite element meshes and represents the most original direct contribution of the dissertation to linear equation solver algorithms.

- Chapter 6 presents a series of serial numerical studies aimed and elucidating some important characteristics of the behavior of iterative solvers in general, and multigrid in particular, on problems in solid mechanics. This chapter identifies and analyses the behavior of iterative solvers on problems with particular features, such as incompressibility, poorly proportioned elements, complex geometries, and large jumps in material coefficients.

- Chapter 7 discusses parallel computational aspects of the finite element method and unstructured multigrid solvers. We discuss the structure of our code and some of the algorithmic issues in optimizing performance of multilevel solvers on today's parallel computers.

- Chapter 8 develops a theoretical framework for modeling multigrid complexity, develops a simple complexity model for the computers in the near future, uses it to analyze multigrid solvers, and describes a more detailed computational model of unstructured multigrid solvers on distributed memory computers.

- Chapter 9 shows numerical results, for problems in 3D linear elasticity, of scalability experiments on an IBM PowerPC and Cray T3E, with up to 7,534,488 degrees of freedom on 512 processors.
- Chapter 10 extends our linear solver to material nonlinear and large deformation finite element analysis, and shows numerical results for problems of up to 16,553,759 degrees of freedom on 542 processors of a Cray T3E with about 60% parallel efficiency. We also develop methods for extending our solver to constrained problems with Lagrange multipliers that arise in finite element simulation with contact.
- Chapter 11 concludes with possible directions for future work.
- Appendix A lists our test problems and problem specifications.
- Appendix B lists the machines that we use for our numerical experiments.

1.6 Contributions

This dissertation develops a highly optimal linear equations solver for finite element matrices on unstructured grids. We extend an effective serial 2D multigrid algorithm to 3D with heuristics to maintain the geometry of a problem on automatically generated coarse grids to dramatically improve performance and robustness of out iterative solver. We develop a new parallel maximal independent set algorithm that has superior PRAM complexity for finite element graphs and is very practical as well. We develop algorithms to mitigate the parallel inefficiency the coarse grids of multigrid solvers on typical computers of today. We have developed a fully parallel finite element implementation built on an existing serial research finite element code, so as to fully test our code and algorithms. We show performance results for problems of up to 16,553,759 equations and up to 512 processors on a Cray T3E and an IBM PowerPC cluster in linear elasticity, large deformation elasticity, and plasticity. We also apply our solver, in serial, to contact problems formulated with Lagrange multipliers.

1.7 Notation

Finite element meshes provide a geometric description of individual elements; different element types require different mesh types. A brief taxonomy, by dimension of degrees of freedom, of 3D element types is as follows:

- **3D:** *continuum* elements meshes are made of polytopes and are fully 3D elements.
- **2D:** manifolds elements can be divided into two main classes of element types: plates and *shells* (with bending energy), and membranes (without bending energy).
- **1D:** elements can be divided into two main classes of element types: *beams* or rods (with bending energy), and trusses (without bending energy).

Finite element analysis can use any and all of these element types in a single analysis. This dissertation however only discusses continuum elements although multigrid methods are applicable to all finite element method formulations (with compactly supported basis functions) [83, 39]. Additionally 3D elements are usually either hexahedra or tetrahedra. Our examples use eight node hexahedral trilinear "brick" elements, but our methods can adapt to other element types.

We work closely with finite element meshes, in 3D continuum mechanics, so a few definitions prove useful.

- **Regular** meshes. The grid points are on a regular lattice.
- **Structured** meshes. The grid points are on a *logically* regular grid, though coordinates may be transformed.
- **Unstructured** meshes. The grid points are placed arbitrarily.
- **Block structured** meshes. Unstructured meshes of structured blocks.

The meshes of all finite element applications must satisfy certain conditions, namely that any vertex that is in the closure of an element domain must be one of the element's vertices, and no two elements may intersect each other. The meshes of primary interest are unstructured, as the primary strength of the finite element method is its ability to accommodate complex domains and boundary conditions.

dof: A *degree of freedom*, in a finite element model, is represented by one entry in a vector and produces one equation in a finite element matrix. The total number of degrees of freedom is denoted as n.

Vectors and functions: We use bold face $\mathbf{u}$ for functions, and plain text u for vectors. A vector u is a list of *weights* by which to scale a set of *basis* functions ϕ_i that add together to form an arbitrary function (in the span(ϕ_i) $i = 1, \ldots, m$) thus $\mathbf{u} = \sum_{i=1}^{m} u_i \phi_i$.

residual: The residual of system of equations $Ax = b$, with a given solution vector x is defined as $r = b - Ax$. Note, we use the (two) norm of the residual (i.e., $\sqrt{r^T r}$) throughout this dissertation, and will generally simple call it the residual.

Chapter 2

Mathematical preliminaries

In this chapter we present background material useful in understanding our work.

2.1 Introduction

We begin with a brief introduction to the finite element method, highlighting the components similar to those in multigrid. Understanding the mathematical structure of the finite element method is useful in describing the nature of the matrices that we solve, but more importantly it proves useful in understanding multigrid equation solvers. This is because the mathematical structure of the finite element method and multigrid are quite similar, even though finite elements and multigrid are solutions to two entirely different problems (i.e., multigrid is a method of implementing the application of the inverse of a finite element operator). In fact, as we show, the core of multigrid is the recursive application of a variationally induced approximation of the "current" problem (grid) - this is also a description of the finite element method.

Linear equation solver fundamentals are introduced as the significance of our work can only be appreciated in the context of the alternatives. Additionally multigrid uses many standard linear equation solvers in its construction. Modern domain decomposition theory is introduced (multigrid can be analyzed as a particular domain decomposition method), because modern domain decomposition analysis provides the strongest analytical methods for multigrid on unstructured meshes. More importantly, modern domain decomposition theory can provide invaluable insight into the nature of multigrid solvers and we thus provide a brief overview of this theory with the intent of touching on its more salient features.

2.2 The finite element method

The problem of simulating physical systems can, in general, be reduced to that of finding the solution $\mathbf{u}$, for a linearized PDE operator $\mathbf{L}$, and applied "forcing" function $\mathbf{Q}$ that has the general form

$$\mathbf{L}(\mathbf{u}, t) + \mathbf{Q} = 0 \quad in \quad \Omega \tag{2.1}$$

$$\mathbf{u} = \bar{\mathbf{u}} \quad on \quad \partial\Omega_d$$

$$\mathbf{B}(\mathbf{u}) = \bar{\mathbf{u}} \quad on \quad \partial\Omega_n$$

Were $\partial\Omega_d$ is the boundary of the domain for which the displacements, or primal variable, is specified (i.e., Dirichlet boundary conditions); $\partial\Omega_n$ denotes the portion of the boundary with natural (i.e., Neumann boundary conditions) for which the dual variable (e.g., force) is specified.

Complex physical phenomenon are modeled, with complex PDEs, which may include domains of entirely different physical behavior (as with fluid-structure interaction problems), or may have multiple coupled physical fields (as with thermal-mechanical systems). Thus the actual simulation may be a composition of multiple domains and these PDEs may be very complex. Additionally, accurate simulations are rarely linear, though often they are linearized for use in a nonlinear solution method (e.g., by Newton's method). Thus linear systems may be but one component of a fully nonlinear solution, but the core solution procedure is the solution of a problem that can be represented symbolically by equation (2.1).

The finite element method is a means of formulating, or (spatially) discretizing, partial differential equations, so they may be solved numerically [86, 24]. The finite element method provides a sound method of finding the "optimal" solution, within a given subspace of the space in which the true solution belongs. The finite element method commonly utilizes a Galerkin condition as its optimality criteria; a Galerkin condition can be stated as find $\tilde{\mathbf{u}} \in \mathcal{S} = \mathbf{span}(\phi_{\mathbf{i}}), i = 1, 2, ..., n$ such that $\mathbf{L}\tilde{\mathbf{u}} + \mathbf{Q} \perp \mathcal{S}$, or equivalently $\langle(\mathbf{L}\tilde{\mathbf{u}} + \mathbf{Q}), \mathbf{v}\rangle = 0 \;\; \forall \mathbf{v} \in \mathcal{S}$ where $\langle \mathbf{a}, \mathbf{b} \rangle = \int_\Omega a \cdot b$.

The success of the finite element method is due in part to its ability to apply the Galerkin condition to arbitrarily complex domains and boundary conditions. The strength of finite elements comes from mappings between arbitrary polyhedra and regular polyhedra (e.g., the unit square where the mathematics are tractable), as well as the availability of computational methods for automatically constructing good polyhedra on complex physical domains [74]. With mesh generation techniques, these physically intuitive meshes can be used to construct piecewise continuous polynomials for the set of basis functions $\mathcal{S}$.

Thus, the finite element method's success derives from its ability to construct accurate subspaces (via unstructured meshes), and from the ability to compute projection onto finite element spaces relatively inexpensively, as the linearized finite element operator is sparse in commonly used finite element discretizations. These projections are computed with a linear equation solver - the effective construction of these solvers, for large finite element problems, is the subject of this dissertation.

The process of applying the finite element method to the simulation of a physical system can thus be summarized as

1. Develop a theory to model the physics (i.e., the *strong* form of the PDE).
2. Formulate a *weak* form of the PDE that can be used in the finite element method.
3. Discretized the domain of interest with a finite element mesh.
4. Pick a set of basis functions for each element - in which to find the approximate solution.
5. Formulate a time integrator for transient PDEs.
6. Develop a nonlinear solution strategy for the discrete time form.
7. For each time step and for each iteration step in the nonlinear solution method, solve a system of linear equations for the parameters of the finite element basis functions.
8. For each time step, substitute the discrete answer into the basis functions to find the *answer* where required and often derivatives of the answer as well, and transform this data into a readable form.

2.2.1 Finite element example: Linear isotropic heat equation

For example, the strong form of the heat equation can be written as:

$$\sum_{i=1}^{D} \frac{\partial}{\partial x_i}\left(k\frac{\partial T}{\partial x_i}\right) + Q = \rho c \frac{\partial T}{\partial t} \quad in \;\; \Omega$$

with k the thermal conductivity, ρ the mass density and c the specific heat of the material; or simply

$$\nabla \cdot (k\nabla T) + Q = \rho c \frac{\partial T}{\partial t} \quad in \;\; \Omega \tag{2.2}$$

$$T = g_D \qquad on \;\; \partial\Omega_D$$

$$\frac{\partial T}{\partial n} = g_N \quad on \;\; \partial\Omega_N$$

$$\partial\Omega_D \cup \partial\Omega_N = \Omega \;\; and \;\; \partial\Omega_D \cap \partial\Omega_N = \emptyset$$

$$T = T_0 \;\; at \;\; t = 0$$

. For simplicity we only discuss the spatial discretization and look for a steady state solution, thus $\frac{\partial T}{\partial t} = 0$ in equation (2.2). Also assume homogeneous Dirichlet boundary conditions (i.e., $T = 0 \;\; on \;\; \partial\Omega$), and that the boundary is smooth enough so that the solution to 2.2 is in $L^2(\Omega)$ (i.e., $\int_\Omega T^2 < \infty$), then the homogeneous Dirichlet problem can be stated as the Poissons problem

$$L(T) + Q = \nabla \cdot (k\nabla T) + Q = 0 \quad in \;\; \Omega \tag{2.3}$$

$$T = 0 \quad on \;\; \partial\Omega$$

. If we multiply 2.3 by an arbitrary function $v \in L^2$ which satisfies the boundary conditions, and integrate over the domain, then 2.3 can be equivalently stated as

$$\int_\Omega v \left[\nabla \cdot (k\nabla T) + Q\right] \partial\Omega = 0 \;\; \forall v \in H_0^1(\Omega) \tag{2.4}$$

. Where $H_0^1(\Omega)$ is the set of functions that are zero on $\partial\Omega$ and whose first derivatives are in $L^2(\Omega)$. As u and v are in Sobolev spaces $H_0^1(\Omega)$, we can define an inner product

$$(u, v) = \int_\Omega u \cdot v$$

and also a bilinear form

$$a(v, u) = \int_\Omega \nabla v \cdot (k\nabla u)$$

so equation (2.4) can be transformed, with integration by parts, to the weak form of the differential equation:

$$a(v, u) - (v, Q) = 0 \;\; \forall v \in H_0^1(\Omega) \tag{2.5}$$

.

To formulate a numerical solution for a problem with the finite element method we first need a *test* space for $\mathbf{v}$

$$\mathcal{T} = span\,(\varphi_1, \varphi_2, ...\varphi_m)$$

and a *solution* space $\mathcal{S}$ for $\mathbf{u}$

$$\mathcal{S} = span\,(\phi_1, \phi_2, ...\phi_m)$$

We can express a vector $\tilde{\mathbf{u}} \in \mathcal{S}$ as $\sum_{j=1}^{n} \alpha_j \phi_j$, and $\tilde{\mathbf{v}} \in \mathcal{T}$ as $\sum_{i=1}^{m} \omega_i \varphi_i$. In practice $\mathcal{T}$ and $\mathcal{S}$ are often the same space, resulting in a Bubnov-Galerkin method, otherwise the method is known as a Petrov-Galerkin method.

A finite element approximation to T, in equations (2.5) and (2.4), can be constructed by replacing $\tilde{\mathbf{u}}$ and $\tilde{\mathbf{v}}$ in equation (2.5). If $\mathcal{T} = \mathcal{S}$, then we have n equation (one for each test function) in n unknowns of the form:

$$\omega_i \sum_{j=1}^{n} a\,(\phi_i, \phi_j) \cdot \alpha_j = \omega_i\,(\phi_i, Q) \tag{2.6}$$

.

Remember v is arbitrary, as long as it satisfies the boundary conditions and so ω_i is arbitrary and thus equation (2.6) may be written as

$$\sum_{j=1}^{n} a\,(\phi_i, \phi_j) \cdot \alpha_j = (\phi_i, Q) \tag{2.7}$$

.

This is a set of n linear equations in n unknowns $Ax = b$, where

$$A_{ij} = a\,(\phi_i, \phi_j) = \int_\Omega \nabla\phi_i \cdot k\nabla\phi_j, \quad b_i = \int_\Omega \phi_i \cdot Q$$

and $x = [\alpha_1, \alpha_2, ..., \alpha_n]^T$ is the discrete vector that we solve for. These inner products are implemented with numerical integration in which the operator $\mathbf{L}$, in Cartesian coordinates, is evaluated at carefully selected *Gauss integration points* - the weighed sum of which taken over each element provides an accurate answer for the polynomial *shape* functions used in finite elements analysis.

Most problems of interest are non-linear, thus A is a function of x. The weak form must then be *linearized* and a non-linear solution strategy formulated e.g., Newton's method [86].

What remains to be done is to solve this sparse set of algebraic equations, for the vector x, the answer $\tilde{\mathbf{u}} = \sum_{i=1}^{n} \alpha_i \phi_i$, can then be constructed and the solution or its derivative can be generated at any point in the domain. Our next step is to introduce the foundations of the method that we employ to solve these equations.

2.3 Iterative equation solver basics

The section introduces iterative equation solvers so as to motivate our research into multigrid. Iterative equation solvers rely on the application of a sparse operator. Direct solvers, based on Gaussian elimination, first factor an n by n matrix A into an upper and a lower triangular matrix (e.g., find L and U such that $A = LU$).

The factorization has a complexity of about $O\left(n^{3/2}\right)$ for typical 3D sparse finite element matrices, (although the exact cost is dependent on the precise structure of the matrix and the order of the equations). Finite element matrix factorizations have a space complexity (i.e., memory requirement) of about $O\left(n^{4/3}\right)$. Iterative methods have a space complexity of $O\left(n\right)$ but the time complexity is method dependent; the minimization of the time complexity of iterative equation solvers is the primary goal in the design of an iterative method.

This section gives a brief description of the classical iterative methods, beginning with the simplest (and least effective) and culminating with what we consider the most effective (multigrid). This introduction is useful not only for providing a context for multigrid but also proves useful in that multigrid actually uses many of the classical iterative methods as components. In fact, multigrid is really nothing more than an intelligent marshaling of iterative and direct methods in order to allow these methods, operating at differing scales of resolution, to "do what they do best" - this becomes clear in subsequent chapters.

2.3.1 Matrix splitting methods

Some of the oldest iterative methods can be described by considering a "matrix splitting", that is given a matrix A define a splitting, $A = M - K$. Substituting this expression into the system to be solved $Ax = b$, we get the equivalent expression $x = M^{-1}\left(b + Kx\right)$. This is not very useful in and of itself, however it does suggest an iterative method: set $k = 0$, and while $(|b - Ax_k| > tol)$ do, $k = k + 1$

$$x_k = M^{-1}\left(Kx_{k-1} + b\right) \tag{2.8}$$

. The idea is now to pick M and K such that the cost of applying the inverse of M is inexpensive relative to the reduction in the error in each iteration.

Jacobi's method is a simple iterative method that visits each equation i and sets $x_i^{(k)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j\neq i}^{n} A_{ij} \cdot x_j^{(k-1)}\right)$, where $(\cdot)_i^{(k)}$ is the i^{th} component of $(\cdot) \in \Re^n$ at iteration k. Jacobi's method is a matrix splitting method in which M in equation (2.8) (M_J) is the diagonal of A. Other commonly used matrix splitting methods are Gauss-Seidel and successive overrelaxation SOR. Gauss-Seidel is a simple and natural improvement (usually) of Jacobi in which the most recent data for x is used instead of only using the values of x form the previous iteration. For Gauss-Seidel M_{GS} is the diagonal and the lower triangular part of A. SOR follows the intuition that if a correction to an approximate solution to a problem is an improvement, then we can magnify this correction to get more improvement to the solution. Thus with a user provided parameter ω (typically $\omega > 1$), M_{SOR} is $\omega^{-1}\left(D - \omega\tilde{L}\right)$ where $\tilde{L}$ is the strictly lower triangular part of A, and K_{SOR} is $\left(\omega^{-1} - 1\right)D + \tilde{U}$.

2.3.2 Krylov subspace methods

Krylov subspace methods are an elegant means of designing iterative solvers that have proven to be quit valuable in practice. A Krylov subspace is defined for a given matrix A and vector b by $\mathcal{K}(A, b)_k = span\left(b, Ab, A^2b, ..., A^{k-1}b\right)$. One advantage of Krylov subspace methods is that the representation of the operator can be entirely abstracted from the iterative method; that is the iterative method need only be able to *apply* the operator and need not directly access the data that is used to represent the operator. This is a positive attribute for an iterative method to possess, but is often not of paramount importance as Krylov subspace methods invariably require preconditioning, which is often some type of an *incomplete factorization* which in turn requires that parts of the actual matrix be accessed. Regardless, Krylov subspace methods have proven to be very useful in the construction of iterative solvers; we introduce them here as we routinely use them in our solvers.

The first Krylov subspace methods were developed in the 1950s, though most of the research activity in these methods has taken place since the 1970s [48, 43]. Over the past 20 years Krylov subspace methods have been designed for may types of matrices; this is necessary as these methods can take advantage of *a priori* knowledge of the operator (i.e., symmetry, positive definiteness, semi-definiteness, etc.), to provide better methods for operators for which something is know about their spectra. A common property of finite element matrices, that Krylov subspace methods can take great advantage of, is symmetric positive definiteness (SPD). Conjugate gradients (CG), the first Krylov subspace method to be developed, is an effective method that requires that the operator be SPD. We describe and derive CG following the presentation in [43].

Consider the functional

$$\phi(v) = \frac{1}{2} v^T A v - v^T b \tag{2.9}$$

Notice that minimizing $\phi(v)$ yields $v = A^{-1}b = x$, that is the minimization of equation (2.9) is equivalent to solving $Ax = b$ for x, if A is SPD. This fact can be deduced by taking the Frechet derivative of equation (2.9)

$$\left.\frac{\partial}{\partial \epsilon}\Big(\phi(x + \epsilon\eta) - \phi(x)\Big)\right|_{\epsilon=0} = \frac{1}{2}\Big(\eta^T A x + x^T A \eta\Big) - \eta^T b \tag{2.10}$$

where η is an *arbitrary* perturbation vector and ϵ is a scalar. By setting 2.10 to zero we find that, if A is symmetric, $x = A^{-1}b$ is required to solve the resulting equation

$$\eta^T (Ax - b) = 0 \tag{2.11}$$

as η is arbitrary. Taking the second Frechet derivative of (2.11), simply give us A, thus if A is positive we are minimizing the functional.

This construction gives us a "heuristic" to iteratively solve for x in $Ax = b$ (actually this is not a heuristic but for the time being we will consider it as such). With this construction we can pick an "optimal" scalar α to improve current solution x_k with a vector p by finding the α that minimizes $\phi(x_k + \alpha p)$. In so doing we find that we should pick α as

$$\alpha = \frac{p^T r_k}{p^T A p} \tag{2.12}$$

where $r_k \equiv b - Ax_k$, if A is symmetric. This is all well and good but how can we pick p? A simple choice is let $p = r_k$, this is the method of *steepest descent* and it is not effective, and does not optimize the solution very well.

Since we are only applying the operator, and are only given the vector b, we also know that $x_{k+1} \in \mathcal{K}(A, b)_k$. For *general search directions* we would like to be able to pick p so the new current solution is optimal in some well defined sense. That is we would like our solution in $x_{k+1} \in \mathcal{K}(A,b)_k$ to minimize the residual in some norm; we can not minimize the error e_k as we do not know the solution, but we can calculate the residual $r_k = Ae_k = b - Av_k$.

An obvious choice is to use the two norm on the residual $\sqrt{r_{k+1}^T r_{k+1}}$, which, when A is symmetric, is equivalent to requiring that $r_{k+1} \perp \mathcal{K}(A,b)_k$, that is the Galerkin condition. With this choice we get *general minimum residuals* (GMRES) [72]. Conjugate gradients chooses p so that the k^{th} iterate x_k minimizes the residual in the A^{-1} norm - $\|x\|_{A^{-1}} = \sqrt{x^T A^{-1} x}$.

We need to deduce the conjugate gradient choice of p in each iteration. To do this express x_{k+1} as a linear combination of vectors $p_1, p_2, ..., p_{k+1}$ which span $\mathcal{K}(A,b)_k$. In matrix notation

$$x_{k+1} = P_{k+1}\bar{y} \tag{2.13}$$

where $\bar{y}$ is a vector of scalar *weights* of all of the *search* vectors in the columns of P_{k+1}. Split equation (2.13) into two parts

$$x_{k+1} = P_k y + \alpha p_{k+1}$$

use equation (2.9) to find x_{k+1}, that is

$$min_{y,\alpha}\phi\left(P_k y + \alpha p_{k+1}\right)$$

after some manipulation we get

$$min_{y,\alpha}\left(\phi\left(P_k y\right) + \alpha y^T P_k^T A p_{k+1} + \frac{\alpha^2}{2} p_{k+1}^T A p_{k+1} - \alpha p_{k+1}^T b\right) \tag{2.14}$$

The second term in equation (2.14), the "cross term" is problematic, as without it the minimization decouples into two "simple" minimizations as we show. The solution to minimizing the cross term is to make it zero, this is accomplished by requiring that all of our *search* vectors be A *conjugate*, that is $p_i^T A p_j = 0$ if $i \neq j$. Thus we have a specification for our search directions.

With the cross term eliminated equation (2.14) reduces to two minimizations: $min_y \phi\left(P_k y\right)$ and $min_\alpha \frac{\alpha^2}{2} p_{k+1}^T A p_{k+1} - \alpha p_{k+1}^T b$. The first part of this minimization $min_y \phi\left(P_k y\right)$ can be assumed to have already been done as we are applying the algorithm recursively; the base case of the recursion $P_k y = 0$ is clearly minimized. The second part of the minimization, after being "given" x_k from the first part, is similar to (2.12), $\alpha = p^T b / p^T A p$. So we have an expression for α in each iteration and a specification for p_{k+1}, namely

$$p_{k+1}^T A p_j = 0 \quad j = 1, ..., k$$

Amazingly if we apply a standard Gram-Schmidt technique to calculate p_{k+1} by A orthogonalizing Ax_k with $p_1, p_2, ..., p_k$, we find that we only need the first two terms of the recurrence (see [27] for a complete discussion of this material), and only need one matrix vector product per iteration.

$x_0 \leftarrow 0$, $r_0 \leftarrow b$, $p_1 \leftarrow b$, $k \leftarrow 1$ – solve $Ax = b$ for x
while $\|r_k\|_2 > tol$

$z \leftarrow Ap_k$
$\nu_k \leftarrow r_{k-1}^T r_{k-1} / p_k^T z$
$x_k \leftarrow x_{k-1} + \nu_k p_k$
$r_k \leftarrow r_{k-1} - \nu_k z$
$\mu_{k+1} \leftarrow r_k^T r_k / r_{k-1}^T r_{k-1}$
$p_{k+1} \leftarrow r_k + \mu_{k+1} p_k$
$k \leftarrow k + 1$

Figure 2.1: Conjugate Gradient Algorithm

Thus the beauty of CG is that this "orthogonality" can be maintained (in perfect arithmetic) with a short vector recurrence, unlike GMRES, which requires that all of the old search vectors be stored so that each new one may be orthogonalized against them [72].

We can state bounds for the convergence rate for CG as

$$\frac{\|r_{k+1}\|_{A^{-1}}}{\|r_k\|_{A^{-1}}} \leq \frac{2}{1 + \frac{2k}{\sqrt{\kappa - 1}}} \tag{2.15}$$

or

$$\|x - x_k\|_A \leq 2 \|x - x_0\|_A \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k \tag{2.16}$$

See [27, 61] for details of these derivations. The important point to see from this is that the convergence rate of CG depends on the condition number of A. CG provides us with an $O\left(n^{3/2}\right)$ solver for our model problem (2D Poisson); this is better than Jacobi and Gauss-Seidel and as good as SOR (without the need to pick a parameter) but it is not good enough for use in very large problems.

Krylov subspace methods require that they be *preconditioned* in order to be effective - the subject of this dissertation is such a preconditioner. Just about any *solver* can be used as a preconditioner for a Krylov subspace method. Many of the classical matrix splitting methods of §2.3.1, and their generalization in §2.4 are used as preconditioners for the Krylov subspace *smoothers* for our numerical experiments throughout this dissertation. Thus preconditioned Krylov subspace methods are central to the subject of this dissertation, but before we look at preconditioning we need to look at the convergence properties of CG.

In addition to CG for SPD matrices there are may other Krylov subspace methods for standard matrices classes (i.e., indefinite, semidefinite, symmetric and unsymmetric), see [27] and the references therein for a full discussion of these methods.

2.3.3 Preconditioned Krylov subspace methods

In the last section we found that the convergence rate of CG (and all other Krylov subspace methods) is dependent on the condition number of the matrix. A natural way to try to improve the convergence rate is to transform the system to an "easier" one, solve it, and then go back to the original system; that is instead of solving $Ax = b$ we want to find an M and solve $M^{-1}Ax = M^{-1}b$, or in symmetric form

$$M^{-1/2}AM^{-1/2}M^{1/2}x = M^{-1/2}b \tag{2.17}$$

One can substitute equation (2.17) into Figure 2.1; after some algebraic manipulations one can eliminate the $M^{-1/2}$ and $M^{1/2}$ terms, and get the algorithm in Figure 2.2.

The objective now resembles that of matrix splitting methods - find an M whose inverse is cheap to apply and is relatively effective at reducing the condition number of the operator $M^{-1}A$. We can look at the Krylov subspace method as an *accelerator* for an iterative solver. We have observed, in informal numerical experiments, that the use of a Krylov subspace method accelerator is almost always economically advantageous (in terms of total execution time for the solve); we therefore only consider this architecture, in our numerical experiments, as there are many other interesting parameters to investigate. So we are still left with a question of finding a good preconditioner for a Krylov subspace method.

$x_0 \leftarrow 0$, $r \leftarrow b$, $k \leftarrow 0$ – solve $Ax = b$ for x
while $\|r\|_2 > tol$
  $z \leftarrow M^{-1}r$
  $\beta_k \leftarrow r^T z$
  if $k = 0$
    $p \leftarrow z$
  else
    $p \leftarrow z + (\beta_k/\beta_{k-1})\, p$
  $z \leftarrow Ax$
  $\alpha \leftarrow \beta_k/p^T z$
  $x \leftarrow x + \alpha p$
  $r \leftarrow r - \alpha z$
  $k \leftarrow k + 1$

Figure 2.2: Preconditioned Conjugate Gradient Algorithm

2.3.4 Krylov subspace methods as projections

We can alternatively look at Krylov subspace methods as a projection of the problem onto a subspace. We often work with projections in the analysis of domain decomposition methods; but for now we can consider projections as an approximation $\mathbf{u}$ in a "low" dimensional space $\mathcal{K}$, to a function $\mathbf{u}^*$ in a "high" dimensional space $\mathcal{V}$. If

- $\mathcal{K}$ is a closed subspace of $\mathcal{V}$
- $\mathcal{V}$ is complete
- Our finite element operator is $\mathcal{V}$-elliptic [24]

then we can define a unique approximation that has an error that is orthogonal to the solution after we equip $\mathcal{L}$ with an inner product. Thus we can use the Galerkin condition from §2.2 and insist that the approximate solution satisfies $(\mathbf{u} - \mathbf{u}^*)^{\mathbf{T}} \cdot \mathbf{v} = \mathbf{0} \mid \forall \mathbf{v} \in \mathcal{L}$, or in the case of CG require that $\mathbf{u}$ minimizes $(\mathbf{u} - \mathbf{u}^*)^{\mathbf{T}} \cdot (\mathbf{Lu} - \mathbf{b})$.

In the case of symmetric systems it is natural to let $\mathcal{L} = \mathcal{K}$, and for the positive definite case this can be accomplished with little cost. We observe later that the cost of finding a projection is a linear solve of the order of the size of the subspace. The reason for the efficiency of CG is that the linear system is tridiagonal and hence very cheap to solve, see [27] for details.

2.4 One level domain decomposition

Domain decomposition methods are a popular approach to construct iterative solvers, especially in multiprocessor environments. Domain decomposition (of the *physical* domain) is a natural method to consider in a parallel computing environment because the finite element mesh invariably needs to be *partitioned.* Thus reasonable to use this structure in the solver, so as to exploit data locality.

Domain decomposition methods have been used by engineers for decades to build direct solvers and iterative solvers alike. Direct domain decomposition solvers are known as nested dissection node ordering or substructuring, iterative solvers are known as *iterative substructuring* or Schur complement methods.

Domain decomposition is important in the history of solvers for finite element matrices, but is also pertinent to this dissertation for two reasons. First, one of the simplest domain decomposition methods is a generalization of Jacobi's methods introduced in §2.3.1; namely use diagonal *blocks* of the matrix as the preconditioning matrix and use a direct solver on each of these blocks - we often block Jacobi solvers as a component in our solver. The second reason for interest in domain decomposition methods is that during the past ten years a powerful method has been developed in the domain decomposition community for the analysis of a wide range of iterative solvers [31], including multigrid methods, on unstructured meshes. See [78] for an introduction to these methods and the references therein for the literature in this area.

This chapter introduces the background and notation useful in the analysis of our, and most, iterative solvers for discretized PDEs; additionally the central concepts of restrictions and interpolation are introduced here as they are used extensively in the description of our methods.

2.4.1 Alternating Schwartz method

This section introduces the classical domain decomposition methods for solving PDEs, starting with Schwarz's method from the 19th century up to the modern numerical methods. This section is thus intended to provide a basis for the discussion of the analytical domain decomposition techniques by providing simple, concrete and intuitive examples of domain decomposition methods. The purpose of this section is to provide historical background of many of the components of our solve as well as introduce some of the basic concepts that we use throughout this dissertation. This discussion follows the presentation in [78].

The earliest domain decomposition method was introduced by Schwarz in 1870 to solve for the continuous solution of a PDE. Schwarz's method was *not* intended as a numerical method but as a way of solving elliptic PDEs composed of the union of simple domains for which explicit solutions were available. Thus this section will work with continuous functions and linear operators and *not* their discretized counterparts, vectors and matrices.

An example of the classical alternating Schwarz method proceeds as follows. Given a domain $\Omega = \Omega_1 \cup \Omega_2$, shown in Figure 2.3, on which we wish to solve the elliptic PDE

$$\mathbf{Lu} = \mathbf{f} \quad in \quad \Omega \tag{2.18}$$

$$\mathbf{u} = \mathbf{g} \quad on \quad \partial\Omega \tag{2.19}$$

For instance $\mathbf{L}$ could be the Laplacian operator ∇^2, and the boundary conditions could be of either Dirichlet or Neumann type, though here we consider only Dirichlet boundary conditions for simplicity.

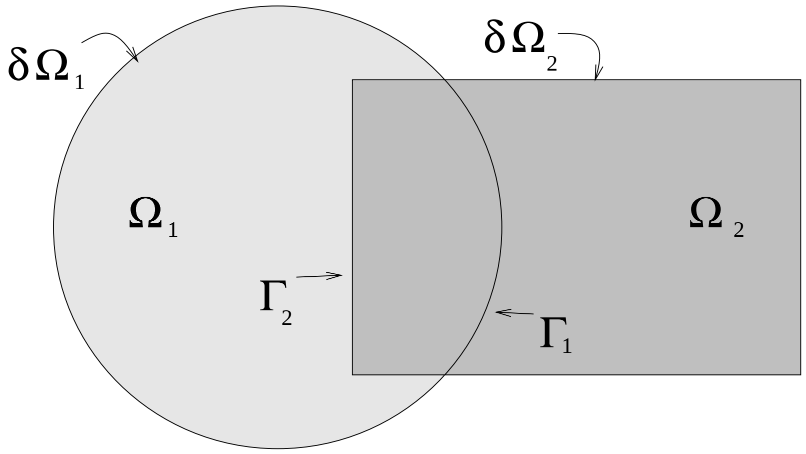


Figure 2.3: Schwarz's original figure

$\partial\Omega$ is the boundary of Ω and closure of Ω is denoted by $\bar{\Omega} = \Omega \cup \partial\Omega$. The artificial boundary Γ_i of subdomain Ω_i is defined as the part of $\partial\Omega_i$ within Ω. We often have use for the boundary $\partial\Omega_i \backslash \Gamma_i$, that is the "true" boundary (if any) of the subdomain i. We define $\mathbf{u}_i^{(k)}$ as the current solution at iteration k on domain $\bar{\Omega}_i$. Also $\mathbf{u}_i^{(k)}|_{\Gamma_j}$ is defined as the *restriction* of $\mathbf{u}_i^{(k)}$ to Γ_j, that is the values of $\mathbf{u}_i^{(k)}$ that are on Γ_j. Note that this type of restriction is simply a selection of certain values or embedding.

The classical alternating Schwarz method can be stated as, select an initial guess for $\mathbf{u}^0$, then iteratively for $k = 1, 2, ...$ solve the boundary value problem

$$\mathbf{L}\mathbf{u}_1^{(k)} = \mathbf{f} \quad in \quad \Omega_1$$

$$\mathbf{u}_1^{(k)} = \mathbf{g} \quad on \quad \partial\Omega_1 \backslash \Gamma_1$$
$$\mathbf{u}_1^{(k)} = \mathbf{u}_2^{(k-1)}|_{\Gamma_1} \quad on \quad \Gamma_1$$

for $\mathbf{u}_1^{(k)}$, then solve the following for $\mathbf{u}_2^{(k)}$

$$\mathbf{L}\mathbf{u}_2^{(k)} = \mathbf{f} \quad in \quad \Omega_2$$

$$\mathbf{u}_2^{(k)} = \mathbf{g} \quad on \quad \partial\Omega_2 \backslash \Gamma_2$$
$$\mathbf{u}_2^{(k)} = \mathbf{u}_1^{(k)}|_{\Gamma_2} \quad on \quad \Gamma_2 \tag{2.20}$$

and continue until the solution has converged.

In effect this method solves a small subdomain problem with boundary conditions augmented by the restriction of current solutions on other subdomains. Notice that this construction is similar to Gauss-Seidel iterations in §2.3.1 though instead of solving for just one equation at each substep in the outer iteration we solve a subdomain boundary value problem; also notice that if, in equation (2.20), we substitute $\mathbf{u}_2^{(k-1)}$ for $\mathbf{u}_2^{(k)}$ we get a Jacobi like iteration. Though this is a continuous method, and is not explicitly used in our work, it does provide intuition as to how domain decomposition methods work.

2.4.2 Multiplicative and additive Schwarz

The section describes a discrete version of the Schwarz method of the last section, and the additive variant. The next chapter extends these methods to include a *coarse grid correction* used in multilevel methods.

Figure 2.4 shows a mesh with two subdomains.

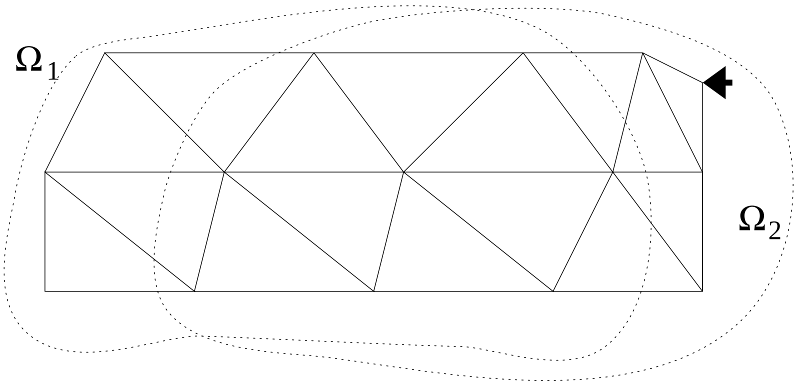


Figure 2.4: Two Subdomains with Matching Grids

For simplicity, assume one node has a Dirichlet boundary condition and the non-homogeneous term moved to the right hand side; also order the nodes with the interior nodes of the first subdomain first, then the first subdomain's boundary nodes, followed by the nodes common to both subdomains, as so on. The vector of unknowns look like this

$$u = \left(u_{\Omega_1 \backslash \bar{\Omega}_2} \quad u_{\Gamma_2} \quad u_{\Omega_1 \cap \Omega_2} \quad u_{\Gamma_1} \quad u_{\Omega_2 \backslash \bar{\Omega}_1} \right)$$

Figure 2.5 shows the resulting graph of the matrix, with this node ordering (now we only consider the closure of the domains and this drop the "bar" notation).

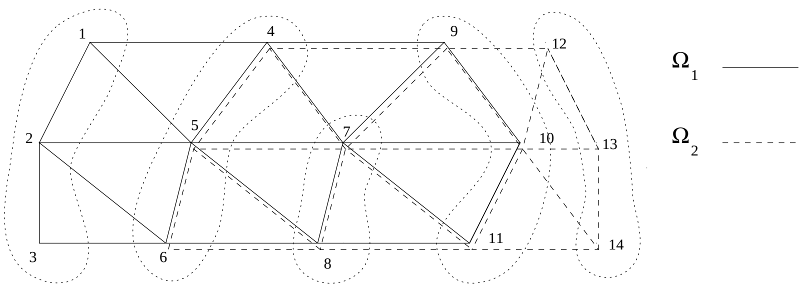


Figure 2.5: Matrix Graph of the Two Subdomain Problem

At this point it is useful to introduce some notation that we use extensively - the *restriction* operator. Figure 2.6 shows two overlapping *submatrices* A_{Ω_1} and A_{Ω_2}. We want to have an algebraic expression to define these submatrices, as in general they are not so simple (i.e., contiguous). We can construct A_{Ω_1} by $A_{\Omega_1} \leftarrow R_1 A R_1^T$, here R_1 is an $|A_{\Omega_1}| \times n$

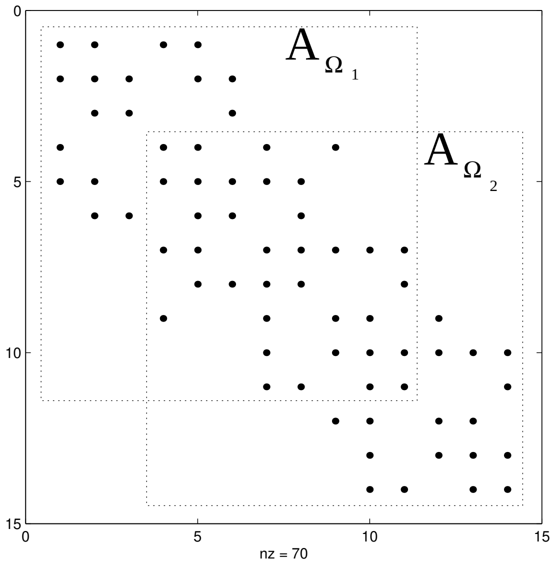


Figure 2.6: Matrix for the Two Subdomain Problem

boolean matrix with a very simple form $[I \quad 0]$. Thus if we had chosen a different ordering of the matrix A then R_1 would be a permuted identity matrix with zero columns inserted for the nodes that are not in Ω_1. These boolean or *embedding* operators are sufficient to describe these simple one level domain decomposition methods - *multilevel* methods require that non-boolean restriction operators be used but they retain this basic structure.

We can state some simple iterative solvers based on these decompositions. Figure 2.7 shows the multiplicative Schwarz algorithm, to solve $Ax^* = b$ for x^*.

$k \leftarrow 0$, $x_0 \leftarrow 0$
while $\|b - Ax_k\| > tol$
$\quad x_{k+1/2} \leftarrow x_k + R_1^T \left(R_1 A R_1^T\right)^{-1} R_1 \left(b - Ax_k\right)$
$\quad x_{k+1} \leftarrow x_{k+1/2} + R_2^T \left(R_2 A R_2^T\right)^{-1} R_2 \left(b - Ax_{k+1/2}\right)$
$\quad k \leftarrow k + 1$

Figure 2.7: Multiplicative Schwarz with Two Subdomains

The *additive* Schwarz algorithm is shown in figure 2.8.

$k \leftarrow 0$, $x_0 \leftarrow 0$

while $\|b - Ax_k\| > tol$

$\quad r_k \leftarrow b - Ax_k$

$\quad x_{k+1} \leftarrow x_k + R_1^T \left(R_1 A R_1^T\right)^{-1} R_1 r_k + R_2^T \left(R_2 A R_2^T\right)^{-1} R_2 r_k$

$\quad k \leftarrow k + 1$

Figure 2.8: Additive Schwarz with Two Subdomains

The additive form of the alternating Schwarz method is a bit cheaper as a new residual is not formed at each subdomain. The additive form is also more parallelizable as the solves on each subdomain are independent of each other, only the update of the solution vector need be synchronized. However multiplicative forms converge faster than additive methods.

Chapter 3

Multilevel domain decomposition

This chapter extends the discussion of the mathematical and algorithmic underpinnings of domain decomposition and multigrid methods from the previous chapter. The one level methods, discussed in the previous chapter, are not effective enough in and of themselves. The shortcomings of one level methods can be overcome by the use of multiple levels or multiple scales of resolution of the problem.

3.1 Introduction

All "optimal" methods have some multilevel component; the need for multiple levels can be understood in many ways. For one, take the linear discretization of the Poisson operator on a regular mesh and a typical row (for vertices with Neumann boundary condition) of the form $(\ldots -1 \ldots -1, 4, -1 \ldots -1)$; notice that the rows add up to zero. Now, the update, or correction, d for domain i with $A_{\Omega_i} = A_{\Omega_i} \leftarrow R_i A R_i^T$ in the Schwarz method with in the error e^n in Figure 2.7 can be written as

$$d = R_i^T A_{\Omega_i}^{-1} R_i \left(b - Ax^n\right) = R_i^T A_{\Omega_i}^{-1} R_i A \left(x^* - x^n\right) = R_i^T A_{\Omega_i}^{-1} R_i A e^n$$

If the error e^n is *constant* in Ω_i then the Schwarz updates are zero on interior or "floating" subdomains (those without any Dirichlet boundary condition) as $R_i A e^n = 0$. Therefore the subdomain corrections do not correct the constant part of the error. If the error (projected onto a subdomain) is dominated by the constant term then the block Jacobi method will not be effective.

Intuitively we can see that if the local part of the global error for a subdomain is "almost" constant then the global error must be smooth. These simple one level solver methods are called "smoothers" in multigrid terminology because they are effective at reducing non-smooth or high frequency error, thereby smoothing the error. This intuition is not valid for all operators however, more generally these simple methods damp the high *energy* components of the error and so it is the *low energy* error that we need to be concerned about (the energy of our solution error e being given by the bilinear form or $e^T Ae$). Thus, for the Poisson operator, the low energy functions are these "smooth" functions because the Poisson operator with constant coefficients is a "smooth" operator.

Another way to divine the need for a global component in a solver is from a simple information theoretic viewpoint. This can most easily be seen by allowing the subdomains to degenerate to single nodes, thus transforming the Schwarz method into Jacobi iterations. If a point load is applied at a corner of a 2D regular mesh then, as Jacobi only transfers information via a matrix vector product with $M^{-1}K$ (which has the same graph structure as A) and b, the *non-zero* structure of x can only advance to the neighbors of a non-zero node in each iteration. As the shortest path to the furthest node from a corner node in a 2D mesh is about $\sqrt{2n}$ long, we require a minimum of $\sqrt{2n}$ iterations to get a non-zero in all n degrees of freedom. The inverse of the Poisson operator is *dense* so all nodes have a non-zero value, in general, in their solution. Thus Jacobi can not (under any circumstances) converge, with high relative accuracy, in less than $\sqrt{2n}$ iterations for this particular right hand side. To remedy this situation we need to have some form of *global* communication in each iteration.

The solution to this inherent limitation on the convergence rate of an iterative solver, is to add a global correction. The global correction is a (perhaps approximate) projection to a smaller subspace. As we see in §3.4.1 these projects are implemented with a small(er) linear solve - we generally apply these methods recursively and only the top (coarsest) grid is solved exactly, thus all but the penultimate grids are corrected with an approximate projection. The subspace, from which we compute a correction, is a *coarse* grid space - the construction of this coarse grid space is the primary distinguishing characteristic of all scalable linear equation solvers.

3.2 A Simple two level method

To create a coarse grid (C) space the domain must first be rediscretized in some fashion, but with far fewer nodes. Using the discretized domain in figure 2.5 we can rediscretize. Figure 3.1 shows such a discretization after it has been remeshed.

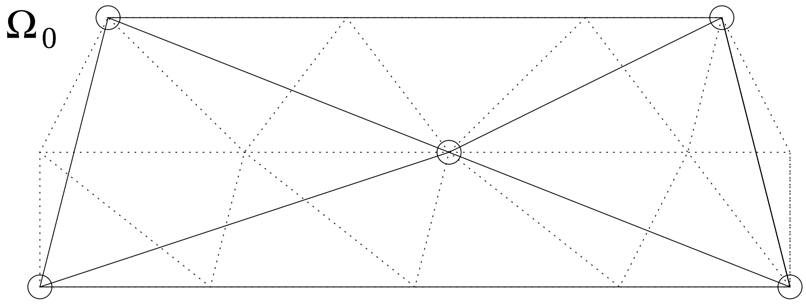


Figure 3.1: Matrix for the Two Level Method

With this coarse grid we can define a new linear operator A_C, treat this like a new overlapping subdomain, and apply either the additive or multiplicative method from §2.4.2. Thus, we *restrict* the current residual $(b - Ax_k)$ up to the coarse mesh, solve for the coarse grid correction, and interpolate the correction back to the fine grid. With the addition of a coarse grid we can write the multiplicative form of our two subdomain example in §2.4.2

$$\begin{aligned} x_{k+1/3} &\leftarrow x_k + R_1^T \left(R_1 A R_1^T\right)^{-1} R_1 \left(b - Ax_k\right) \\ x_{k+2/3} &\leftarrow x_{k+1/3} + R_2^T \left(R_2 A R_2^T\right)^{-1} R_2 \left(b - Ax_{k+1/3}\right) \\ x_{k+1} &\leftarrow x_{k+2/3} + R_C^T A_C^{-1} R_C \left(b - Ax_{k+2/3}\right) \end{aligned} \tag{3.1}$$

The restriction matrix for the coarse grid is no longer a simple boolean operator as we need to interpolate the nodal values on the fine mesh to multiple nodal values of the coarse mesh. Thus we need discrete *coarse grid* restriction operators; figure 3.2 show the *directed* graph (in bold) of this restriction operator. Much is known about calculating interpolation values as this is at the core of the finite element method i.e., mapping between domains. Thus given a *shape* function $\phi_I(x)$ (i.e., the finite element basis function) for coarse node I, we can calculate R_Ij, which is the *interpolate* of I at the fine node j, by $R_Ij = \phi_I(j.coord)$ with $j.coord$ being the coordinate of node j.

With this construction we again have the same options as with the one level Schwarz methods: we can use an additive form or a multiplicative form for the coarse

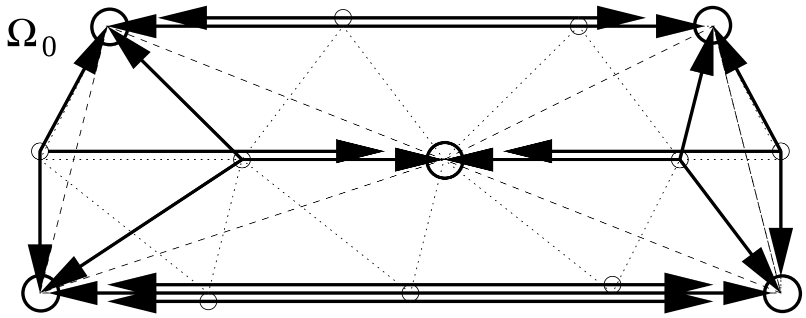


Figure 3.2: Graph for the Restriction Matrix for the Two Level Method

grid correction. We can therefore generate four basic forms of this two level method - the algorithm given in equation (3.1) is the *multiplicative-multiplicative* form i.e., multiplicative within a level and between levels. Additionally we can symmetrize equation (3.1) and if we use the additive method for the subdomain corrections we get the additive-multiplicative form

$$x_{k+1/3} \leftarrow x_k + R_1^T \left(R_1 A R_1^T\right)^{-1} R_1 \left(b - Ax_k\right) + R_2^T \left(R_2 A R_2^T\right)^{-1} R_2 \left(b - Ax_k\right)$$

$$x_{k+2/3} \leftarrow x_{k+1/3} + R_C^T \left(R_C A R_C^T\right)^{-1} R_C \left(b - Ax_{k+1/3}\right)$$

$$x_{k+1} \leftarrow x_{k+2/3} + R_1^T \left(R_1 A R_1^T\right)^{-1} R_1 \left(b - Ax_{k+2/3}\right) + R_2^T \left(R_2 A R_2^T\right)^{-1} R_2 \left(b - Ax_{k+2/3}\right) \tag{3.2}$$

This gives us the classic multigrid form (with a block Jacobi smoother) and is the basic approach that we utilize in our numerical experiments. We can now introduce the classic multigrid algorithm.

3.3 Multigrid

Multigrid has been accepted for the past 25 years as being the theoretically optimal solution method for some model problems, and a great deal of research has been focused on applying multigrid to many types of discretized PDEs [66]. In any given year there are many international conferences dedicated to multigrid, additionally multigrid is well represented in domain decomposition conferences and iterative method conferences worldwide.

This chapter is concerned with providing the basic multigrid background without the distraction of unstructured meshes and parallel computing. We discuss the classical multigrid form which is a natural extension from §3.2 - although unlike most classical presentations we assume that multigrid is used as a preconditioner. Using multigrid as a preconditioner merely means that we are not improving a solution of $Ax_k = b$ with x_k, but are finding an approximate solution to $Ax_{k+1} = r$. Additionally we enter the algorithm with a residual on the *fine* mesh, this changes the structure of *full* multigrid, and is explained shortly.

We number the grids from the bottom (fine) to the top (coarse), counter to the practice in the classic multigrid literature; this is because we start with the fine mesh and work our way up to the coarse mesh, stopping when we can solve the problem directly. Historically multigrid has been used primarily on structured meshes as one starts at the top mesh and continues to refine the mesh until we have an accurate answer. Note, we will use "top" to mean the coarse grid, even though it is at the *bottom* of figure 3.4.

The primary operators used by multigrid are

- **Smoother**. The smoother $S(A, r)$ is an iterative solver that is applied for only a few iterations, or even just one iteration. The smoother must be effective at reducing the error up to the frequency that can be resolved on the mesh and down to the frequency that can be resolved on the next coarsest mesh.

- **Restriction**. The restriction operator $R(r)$ must be able to map residuals to the next coarsest grid.

- **Interpolation or Prolongation**. The interpolation operator $P(x)$ must map values (solutions) from a grid to the next finer grid. The transpose of the restriction operator is commonly used (i.e., $P = R^T$).

- **Coarse Grid Operator**. The coarse grid operator A_{i+1} must represent *all* of the frequencies lower than those that can be effectively reduced by the smoother on this level. More precisely, the error in the coarse grid representation of the "low" eigenfunctions of the fine grid must be in the space spanned "high" eigenfunctions of the fine grid. We use a Galerkin or variational coarse grid operator as discussed above, that is $A_{i+1} = R_i A_i P_i$.

With these components we can state the classic multigrid *V-cycle* in Figure 3.3.

function $MGV(A_i, r_i)$

  if there is a coarser grid

    $x_i \leftarrow S(A_i, r_i)$

    $r_i \leftarrow r_i - Ax_i$

    $r_{i+1} \leftarrow R_{i+1}(r_i)$

    $x_{i+1} \leftarrow MGV(R_{i+1}A_iR_{i+1}^T, r_{i+1})$

    $x_i \leftarrow x_i + R_{i+1}^T(x_{i+1})$

    $r_i \leftarrow r_i - A_ix_i$

    $x_i \leftarrow x_i + S(A_i, r_i)$

  else

    $x_i \leftarrow A_i^{-1}r_i$

  return x_i

Figure 3.3: Multigrid *V-cycle* Algorithm

We represent this algorithm schematically in Figure 3.4.

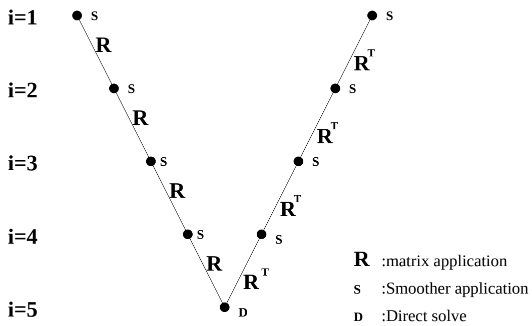


Figure 3.4: Multigrid *V-cycle*

The preconditioner, B in Figure §2.2, is $MGV(A_0, r)$. In practice it has been universally observed that a variant of the *V-cycle*, *full* multigrid or F-cycles, provides better solver performance. Figure 3.5 shows the full multigrid algorithm.

function $FMG(A_i, r_i)$
    if there is a coarse grid
        $x_i \leftarrow FMG(A_{i+1}, R_{i+1}(r_i))$
        $r_i \leftarrow r_i - A_i x_i$
    else
        $x_i \leftarrow 0$
    $x_i \leftarrow x_i + MGV(A_i, r_i)$
    if there is a finer mesh return $R_i^T(x_i)$
    else return x_i

Figure 3.5: *Full* Multigrid Algorithm

And we can schematically represent full multigrid in Figure 3.6.

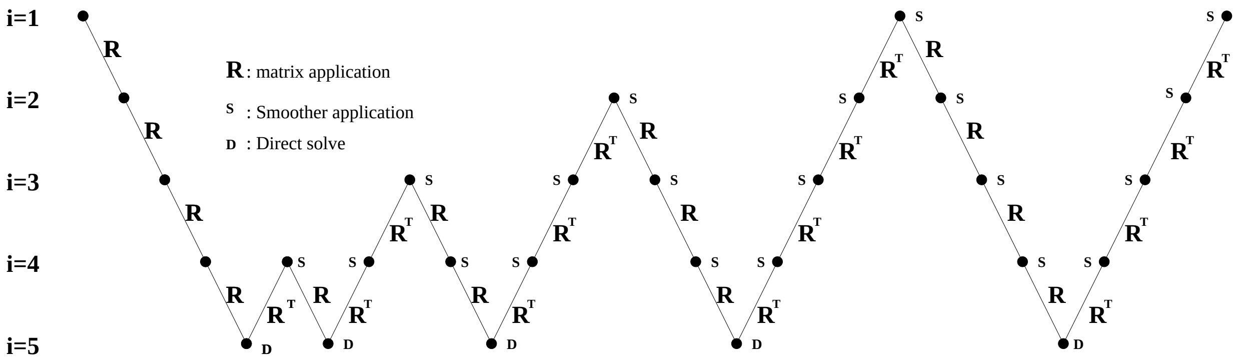


Figure 3.6: Multigrid *F-cycle*

3.3.1 Convergence of multigrid

The intuition behind the convergence behavior of multigrid starts with the notion that iterative methods, which "smooth" values of nearest neighbors on the grid, can effectively reduce the high frequency, or high energy, component of the residual. Multigrid organizes standard iterative methods to work at varying *scales of resolution* so as to allow their work to be most productive. To do this we need to be able to map residuals and corrections between grids in an effective manner; and we need a coarse grid operator that represents (in tandem with these inter grid transfer operators) the low frequency components of the error.

Informally we want the difference between the solution in the coarse grid space, mapped to the fine grid space, and the true answer on the fine grid to have little of the low energy components. Thus the effect of the coarse grid correction is to "demote" the low frequency error to high frequency error, which can be reduced cheaply. For our model problem (Poisson's equation in any dimension) a particular multigrid construction satisfies these requirements extremely well. The Poisson problem is one of the few problems for which hard bounds on the number of iterations required to achieve a specified tolerance is known. The next section sketches the proof of the convergence rate of multigrid on Poisson's equation on a unit square.

Convergence proof outline

We sketch the proof here as it useful in understanding the frequency domain decomposition nature of multigrid. This presentation follows that in [27]. As a model problem uses the regular 1D Poisson equation with Dirichlet boundary conditions. The optimal multigrid algorithm for Poisson's equation is carefully constructed to reduce the error by at least a factor of $\frac{1}{9}$ in each iteration.

Multigrid is optimal if we use $N_k = 2^k + 1$ nodes, with constant spacing between grid points, in each dimension (i.e., $\left(2^k+1\right)^D$ nodes and $\left(2^k-1\right)^D$ unknowns). The coarse grid "picks every other node" from the fine mesh and by the special choice of grid dimension the end nodes remain through all of the grids. In 2D and 3D one can perform the same procedure recursively - coarsening the edges then, starting from selected edge nodes, select surface nodes, and so on.

The restriction (and interpolation) operators are derived from standard linear interpolation. Thus the first row in the restriction matrix R is $R(1,:) = [\frac{1}{2} \quad 1 \quad \frac{1}{2} \quad 0 \quad 0 \quad \ldots \quad 0]$. The Galerkin form for the coarse grid operator (RAR^T) is the same Poisson matrix scaled by $\frac{1}{2}$ in magnitude and approximately $\frac{1}{2}$ in size. The smoother uses a *weighted* Jacobi method, similar to that introduced in §2.3.1, where $R_\omega = I - \omega A/2$ and $c_\omega = \omega b/2$; $\omega = 1$ gives the standard Jacobi operator. Let $e^k = x_k - x^*$ be the error in the k^{th} iteration, and note that with the eigendecomposition Z of A, $R_J = Z\left(I - \omega\Lambda/2\right)Z^T$ where Λ is a diagonal matrix of the eigenvalues of A. We have

$$\begin{aligned} e^k &= R_\omega e^{k-1} \\ &= R_\omega^k e^0 \\ &= \left(Z\left(I - \omega\Lambda/2\right)Z^T\right)^k e^0 \\ &= Z\left(I - \omega\Lambda/2\right)^k Z^T e^0 \end{aligned}$$

so

$$Z^T e^k = \left(I - \omega\Lambda/2\right)^k Z^T e^0 \quad or \quad \left(Z^T e^k\right)_j = \left(I - \omega\Lambda/2\right)_{jj}^k \left(Z^T e^0\right)_j$$

$\left(Z^T e^k\right)_j$ is called the j^{th} *frequency component* of the error e^k. The eigenvalues $\lambda_j\left(R_\omega\right) = 1 - \omega\lambda_j/2$ determine how fast each component of the error decreases in each iteration. Figure 3.7 plots $\lambda_j\left(R_\omega\right)$.

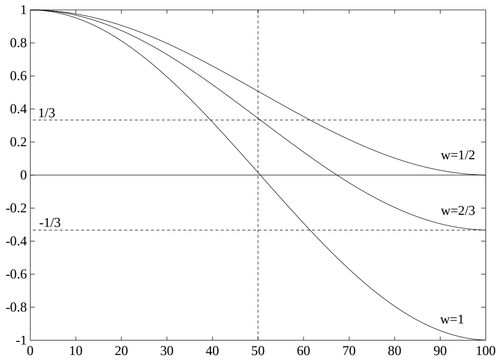


Figure 3.7: Graph of spectrum of (R_ω) for $\omega = 1/2, 2/3, 1$, $N = 99$

When $\omega = \frac{2}{3}$ and $j > \frac{N}{2}$ i.e., the upper half of the spectrum, $|\lambda_j| \leq \frac{1}{3}$. This means that upper half of the error components are multiplied by $\frac{1}{3}$ or less in each iteration.

With Figure 3.3 and by using the identity $x^* = R_{2/3}x^* + b/3$ in the two steps that apply the smoother, and the identity $b - Ax^* = 0$ in the line that takes the residual, we can construct an expression for the application of one $V - cycle$ to the error

$$e^{k+1} = R_{2/3}\{I - R^T \left(RAR^T\right)^{-1} RA\}R_{2/3}e^k \tag{3.3}$$

Note, this expression assumes, by induction, that the coarsest grid is solved exactly [18]. This expression can be nearly diagonalized with Z the eigenvector matrix of A. Equation (3.3) is block diagonal with a particular ordering of the eigenvector matrix. The eigenvalues of these blocks can be explicitly calculated (these blocks are 2 by 2 in the 1D case, 4 by 4 in the 2D case, and so on), see [27] and [78] for details. The eigenvalues of the matrix in equation (3.3) are bounded by $\frac{1}{9}$, and thus we get nearly one digit of accuracy per iteration; this rate is not only fast but it is completely independent of the size of the problem. This construction is *perfect* in that as we go down the $V - cycle$ we are reducing the error uniformly by $\frac{1}{3}$, and as the corrections percolate back up the $V - cycle$ they do not soil the lower part of the spectrum in any significant way, and they project their correction at least well enough so that the smoothing step reduces the entire spectrum of the error by at least $\frac{1}{9}$.

Multigrid is thus completely scalable in serial, provided the amount of work per iteration is $O(n)$ (i.e., $O(1)$ work for each unknown). As the work per grid is proportional to the number of unknowns and the number of unknowns per grid decreases with a geometric progression (i.e., $1, 1/2^D, 1/4^D, 1/8^D, ..., 1/2^{D \cdot L}$ for L levels in D dimensions) - the amount of work is bounded by twice the work done on the fine grid in 1D. Multigrid is therefore an $O(n)$ method sequentially.

3.4 Convergence analysis of domain decomposition

The convergence analysis for unstructured problems is less satisfying than that of the previous section. This is because the analysis for unstructured problems can not provide absolute bounds on the rate of convergence, but only bounds on the condition of the preconditioned system. With bounds on the condition number we can use the bounds from the Krylov subspace method (§2.3.2) to give bounds on the convergence rate.

Additionally the bounds that can be derived, though impressive feats of analysis, have loosely defined parameters (h and H) and other parameters that are difficult to explicitly calculate. Despite the shortcomings of this analytical framework it is quite valuable in providing necessary conditions for scalability, as well as for providing a theoretical framework to compare different algorithms in a rational way.

From "Domain Decomposition" [78], page 34

> In order to understand the convergence behavior of domain decomposition algorithms we need to introduce a mathematical framework. This is most easily done in the context of Sobolev spaces and Galerkin finite elements. Indeed, it turns out that the basic correction steps calculated in virtually all domain decomposition and multigrid/multilevel algorithms may be viewed as (approximate) orthogonal projections, in some suitable inner product, onto a subspace. This observation makes possible the complete analysis of many domain decomposition and multigrid methods ...

§3.4.1 introduces a variational framework and projections, with the example in §2.2.1, and show that the correction $R_1^T\left(R_1AR_1^T\right)^{-1}R_1r_k$ in equation (3.2) in the multiplicative Schwarz algorithm is an example of a projection. §3.4.2 will review or introduce the components required to describe, and analyze, all domain decomposition methods with the convergence theory described in §3.4.3.

This presentation follows that in [78].

3.4.1 Variational formulation

We recall the steady state Poisson's equation from §2.2.1 and consider only homogeneous Dirichlet boundary conditions. The strong form is

$$\nabla \cdot (k\nabla \mathbf{u}) + \mathbf{f} = \mathbf{0} \quad in \quad \Omega$$

$$\mathbf{u} = \mathbf{0} \quad on \quad \partial\Omega$$

and the weak form

$$\mathbf{a}\left(\mathbf{v},\mathbf{u}\right)=\mathbf{f}\left(\mathbf{v}\right)\ \ \forall\mathbf{v}\in H_0^1\left(\Omega\right)$$

with

$$a\left(\mathbf{v},\mathbf{u}\right)=\int_\Omega \nabla\mathbf{v}\cdot\left(\mathbf{k}\nabla\mathbf{u}\right)$$

$$\mathbf{f}\left(\mathbf{v}\right)=\int_{\mathbf{\Omega}}\mathbf{v}\cdot\mathbf{f}$$

Within this variational framework we can demonstrate that the corrections in the multiplicative Schwarz methods are projections of the error. Let $\mathbf{u}^n$ be the solution at the end of the n^{th} iteration, and let $\mathbf{u}^{n+1/2}$ be the solution at the end of the substep in iteration $n+1$. Then the one level (continuous) Schwarz algorithm can be written as

$$\mathbf{L}\left(\mathbf{u}^{n+1/2}-\mathbf{u}^n\right)=-\mathbf{L}\mathbf{u}^n+\mathbf{f}\ \ in\ \ \Omega_1$$

$$\mathbf{u}^{n+1/2}-\mathbf{u}^n=0\ \ on\ \ \partial\Omega_1$$

$$\mathbf{u}^{n+1/2}-\mathbf{u}^n=0\ \ on\ \ \Omega_2\backslash\Omega_1$$

and

$$\mathbf{L}\left(\mathbf{u}^{n+1}-\mathbf{u}^{n+1/2}\right)=-\mathbf{L}\mathbf{u}^{n+1/2}+\mathbf{f}\ \ in\ \ \Omega_2$$

$$\mathbf{u}^{n+1}-\mathbf{u}^{n+1/2}=0\ \ on\ \ \partial\Omega_2$$

$$\mathbf{u}^{n+1}-\mathbf{u}^{n+1/2}=0\ \ on\ \ \Omega_1\backslash\Omega_2$$

This can be expressed in weak form as

$$a\left(\mathbf{u}^{n+1/2}-\mathbf{u}^n,\mathbf{v}\right)=f\left(\mathbf{v}\right)-a\left(\mathbf{u}^n,\mathbf{v}\right)\qquad \mathbf{u}^{n+1/2}-\mathbf{u}^n\in H_0^1\left(\Omega_1\right),\forall\mathbf{v}\in H_0^1\left(\Omega_1\right)\quad(3.4)$$

and

$$a\left(\mathbf{u}^{n+1}-\mathbf{u}^{n+1/2},\mathbf{v}\right)=f\left(\mathbf{v}\right)-a\left(\mathbf{u}^{n+1/2},\mathbf{v}\right)\quad \mathbf{u}^{n+1}-\mathbf{u}^{n+1/2}\in H_0^1\left(\Omega_2\right),\forall\mathbf{v}\in H_0^1\left(\Omega_2\right)\quad(3.5)$$

Let $\mathbf{e}^n=\mathbf{u}^*-\mathbf{u}^n$ be the error in the n^{th} iterate $\mathbf{u}^n$, then

$$a\left(\mathbf{u}^{n+1/2}-\mathbf{u}^n,\mathbf{v}\right)=a\left(\mathbf{e}^n,\mathbf{v}\right)\qquad\forall\mathbf{v}\in H_0^1\left(\Omega_1\right)$$

and

$$a\left(\mathbf{u}^{n+1}-\mathbf{u}^{n+1/2},\mathbf{v}\right)=a\left(\mathbf{e}^{n+1/2},\mathbf{v}\right)\qquad\forall\mathbf{v}\in H_0^1\left(\Omega_2\right)\quad(3.6)$$

Define the projection $\mathbf{T}_i\mathbf{e}$, in the inner product $a(\cdot,\cdot)$, by

$$a(\mathbf{T}_i\mathbf{e},\mathbf{v}) = a(\mathbf{e},\mathbf{v}) \qquad \mathbf{T}_i\mathbf{e} \in H_0^1(\Omega_i), \forall\mathbf{v} \in H_0^1(\Omega_i) \tag{3.7}$$

At each half step the corrections, $\mathbf{u}^{n+1/2} - \mathbf{u}^n$ and $\mathbf{u}^{n+1} - \mathbf{u}^{n+1/2}$ calculated in equation (3.6) are projections of the error onto the subspaces $H_0^1(\Omega_1)$ or $H_0^1(\Omega_2)$.

Formally a **projection** of $\mathbf{e}$ onto a subspace $H_0^1(\Omega_1)$, in the inner product $a(\cdot,\cdot)$ is defined as

$$\mathbf{e}_1 = P\mathbf{e} = \arg \inf_{\mathbf{v}\in H_0^1(\Omega_1)} \|\mathbf{e}-\mathbf{v}\|_a \tag{3.8}$$

We can alternatively define $P\mathbf{e}$ by the following: Find $\mathbf{e}^1 \in H_0^1(\Omega_1)$ so that

$$a\left(\mathbf{e}^1,\mathbf{v}\right) = a(\mathbf{e},\mathbf{v}) \qquad \forall\mathbf{v} \in H_0^1(\Omega_1)$$

or equivalently

$$a\left(\mathbf{e}^1-\mathbf{e},\mathbf{v}\right) = 0 \qquad \forall\mathbf{v} \in H_0^1(\Omega_1)$$

It is now simple to show that (3.7) is the minimizer of (3.8), and thus our correction in the Schwarz methods are in fact projections of the error on to a subdomain. These projections have the attractive property that they find a correction, in a given subdomain, that is *closest* to the error in the A, or energy, norm. The next section shows that this indeed the case for the discrete forms of the Schwarz methods.

Matrix representation of projections

In §2.2 we constructed the finite dimensional subspace $\mathcal{S} = span(\phi_k) \subset H_0^1(\Omega), k = 1,2,...,n$, we now replace $H_0^1(\Omega_1)$ above by $\mathcal{S}$. After selecting domains $\Omega_i,\ \ i = 1,2,\ldots,p$, define the sets $\{\phi_k^{(i)}\}$ i.e., those functions in $\mathcal{S}$ with support in Ω_i. Express $\mathbf{u} \in \mathcal{S}$ as $\mathbf{u} = \sum_k u_k\phi_k$ and let A be the stiffness matrix and $A_{ij} = a(\phi_i,\phi_j)$. It is possible to derive an explicit representation for $\mathbf{T}_i\mathbf{u}$ in equation (3.7). Let $\mathbf{w} = \mathbf{T}_i\mathbf{u} = \sum_k w_k\phi_k^{(i)}$, then equation (3.7) can be expressed as

$$a\left(\sum_k w_k\phi_k^{(i)},\phi_l^{(i)}\right) = a\left(\sum_k u_k\phi_k,\phi_l^{(i)}\right) \qquad \forall\phi_l^{(i)}$$

$$\sum_k w_k a\left(\phi_k^{(i)}, \phi_l^{(i)}\right) = \sum_k u_k a\left(\phi_k, \phi_l^{(i)}\right) \qquad \forall \phi_l^{(i)}$$

Recall that the restriction matrix R_i maps the coefficients of u to coefficients of the local subdomain i, thus $a\left(\phi_k^{(i)}, \phi_l^{(i)}\right) = R_i A R_i^T$. We can now express a *discretized* form of our projection operator

$$A_{\Omega_i} w = (R_i A R_i^T) w = R_i A u$$

or

$$w = (R_i A R_i^T)^{-1} R_i A u$$

Hence the matrix representation of the operator $\mathbf{T_i}$ is given by

$$T_i = R_i^T (R_i A R_i^T)^{-1} R_i A$$

Note, this is the coarse grid correction term in equation (3.3), applied to the error.

3.4.2 Domain decomposition components

We have described some of the basic components of domain decomposition methods in functional form. This section list the components necessary to specify all domain decomposition methods, some of which have been introduced previously and some of which are introduced here.

Domain decomposition polynomials

Recall the correction step in the multiplicative Schwarz algorithm in figure 2.7 is given by

$$x_{k+1/2} \leftarrow x_k + R_1^T \left(R_1 A R_1^T\right)^{-1} R_1 (b - A x_k)$$

$$x_{k+1} \leftarrow x_{k+1/2} + R_2^T \left(R_2 A R_2^T\right)^{-1} R_2 \left(b - A x_{k+1/2}\right)$$

Define B_i by $R_i^T \left(R_i A R_i^T\right)^{-1} R_i$. The operator B_i restricts the residual to one subdomain, solves for a correction in this domain and then interpolates the correction back to the global space. We can rewrite the two domain multiplicative Schwarz correction as

$$x_{k+1} \leftarrow x_k + (B_1 + B_2 - B_2 A B_1)(b - A x_k) \tag{3.9}$$

We can interpret multiplicative Schwarz as a Richardson iterative procedure with the preconditioner B given by $B = B_1 + B_2 - B_2 A B_1$. B can be thought of as a polynomial in the residual r. Note that the utility of using a Krylov subspace method is now evident as CG requires very little additional cost and provides a solver with well defined optimization properties - precisely the optimization properties described in section §3.4.1 for the Schwarz methods.

The form of the multiplicative Schwarz method for the error

$$e^{k+1} \leftarrow e^k + (B_1 + B_2 - B_2AB_1)\, Ae^k$$

We use the projection operators from §3.4.1 and find an expression for the error reduction operator of the method $T = BA = T_1 + T_2 - T_2T_1 = I - (I - T_1)(I - T_2)$ with $B = B_1 + B_2 - B_2AB_1$. We can express the effect of a method on the error in the form of polynomials in T_i, with $i = 0, 1, 2, ..., p$. T_0 refers to the coarse grid in our convention. For the multiplicative multilevel method we have $T = BA = \mathcal{P}(T_0, T_1, ..., T_p) = I - (I - T_p) \cdots (I - T_0)$. The *art* of designing preconditioners is to design these polynomials such that they are well conditioned operators with respect to the cost of their application

Auxiliary bilinear forms

When an *approximate* solver is used in equation (3.4) it becomes

$$a_1\left(\mathbf{u}^{n+1/2} - \mathbf{u}^n, \mathbf{v}\right) = f(\mathbf{v}) - a(\mathbf{u}^n, \mathbf{v}) \qquad \mathbf{u}^{n+1/2} - \mathbf{u}^n \in \mathcal{S}, \forall \mathbf{v} \in \mathcal{S}_1$$

likewise equation (3.5) becomes

$$a_2\left(\mathbf{u}^{n+1/2} - \mathbf{u}^n, \mathbf{v}\right) = f(\mathbf{v}) - a(\mathbf{u}^n, \mathbf{v}) \qquad \mathbf{u}^{n+1/2} - \mathbf{u}^n \in \mathcal{S}, \forall \mathbf{v} \in \mathcal{S}_2$$

That is we do *not* require that we have the same bilinear form on the subdomains, be they coarse grids or subdomains, and can we can therefore use *auxiliary* bilinear forms.

Interpolation operators

The existence of subdomains requires interpolation operators to recover the subdomain corrections. To this end we define the set of operators $\mathbf{I_i} : \mathcal{S}_\mathbf{i} \rightarrow \mathcal{S}$.

Domain decomposition components

We now have all of the components necessary to define any particular domain decomposition method

- A set of subspaces $\mathcal{S}_i$.
- A set of interpolation operators I_i.
- A set of auxiliary bilinear forms $a_i(\cdot,\cdot)$.
- The polynomial $\mathcal{P}(T_0, T_1, ..., T_p)$ to prescribe the order of the application of the subdomain corrections.

The next section applies the formalism of this section to an abstract analysis applicable to *all* domain decomposition methods.

3.4.3 A convergence theory

The goal of this section is to sketch an abstract convergence theory for domain decomposition methods. These methods are composed of somewhat intuitive parameters that make the utility of multigrid evident as well as serve to provide an understanding of the convergence characteristics of domain decomposition methods on unstructured meshes.

To begin with we assume, for simplicity, that we have symmetric positive definite operators and work with *operators* and *functions* rather than matrices and vectors.

First, the abstract convergence theory makes extensive use of the following lemma.

LEMMA 1: *Define:* $\mathbf{T} = \sum_i \mathbf{T}_i$, Then

$$a(\mathbf{T}^{-1}\mathbf{u}, \mathbf{u}) = \min_{\mathbf{u}_i \in \mathbf{V}_i,\ \mathbf{u} = \sum_i \mathbf{I}_i \mathbf{u}_i} \sum_i a_i(\mathbf{u}_i, \mathbf{u}_i)$$

See [78] for a proof of this.

The abstract convergence theory for symmetric positive definite problems centers around three parameters which measure the interaction of the subspaces $\mathbf{V}_i$ and the bilinear forms $a_i(\cdot,\cdot)$ These parameters are presented in the form of assumptions.

Assumption 1: let C_0 be the *minimum constant* such that $\forall \mathbf{u} \in \mathbf{V}, \exists \mathbf{u} = \sum_i \mathbf{I}_i \mathbf{u}_i, \mathbf{u}_i \in \mathbf{V}_i$, with

$$\sum_i a_i(\mathbf{u}_i, \mathbf{u}_i) \leq C_0^2 a(\mathbf{u}, \mathbf{u})$$

If C_0 can be bounded independently of the grid parameters (size of elements and number of subdomains) then the $\mathbf{V}_i$ are said to provide a *stable splitting* of $\mathbf{V}$. This quantity $a(\mathbf{u}, \mathbf{u})$ is referred to as the *energy* of $\mathbf{u}$. A value of 1 is desirable for C_0 - thus we want the subdomain spaces to have minimal energy. This assumption together with Lemma 1 provides a lower bound (C_0^{-2}) on the spectrum of $\mathbf{T} = \sum_i \mathbf{T}_i$ - for the additive Schwarz operator. Note, multigrid is a stable splitting and the coarse grid functions have relatively low energy because the linear finite element shape functions (hat functions) are somewhat "smooth".

Assumption 2: *Define:* $0 \leq \mathcal{E}_{ij} \leq 1$ to be the minimal value that satisfies

$$|a(\mathbf{I}_i\mathbf{u}_i, \mathbf{I}_j\mathbf{u}_j)| \leq \mathcal{E}_{ij} a(\mathbf{I}_i\mathbf{u}_i, \mathbf{I}_i\mathbf{u}_i)^{1/2} a(\mathbf{I}_j\mathbf{u}_j, \mathbf{I}_j\mathbf{u}_j)^{1/2}$$

$$\forall \mathbf{u}_i \in \mathbf{V}_i, \mathbf{u}_j \in \mathbf{V}_j, \quad i, j = 1, \ldots, p$$

Define: $\rho(\mathcal{E})$ to be the spectral radius of $\mathcal{E}$. Note, we do not include the coarse grid space $\mathbf{V}_0$, in this definition. This parameter is in some sense the measure of orthogonality of the subspaces. When $\mathcal{E}_{ij} = 0$ the subspaces $\mathbf{V}_i$ and $\mathbf{V}_j$ are orthogonal; when $\rho(\mathcal{E}) = 1$ we have the usual Cauchy-Schwarz inequality, and for $0 < \rho(\mathcal{E}) < 1$ we have a strengthened Cauchy-Schwarz inequality. A value of 1 is desirable for $\rho(\mathcal{E})$.

Assumption 3: let $\omega \in [1, 2)$ be the *minimum constant* such that $\forall \mathbf{u}_i \in \mathbf{V}_i, \quad i = 0, \ldots, p$,

$$a(\mathbf{I}_i\mathbf{u}_i, \mathbf{I}_i\mathbf{u}_i) \leq \omega a_i(\mathbf{u}, \mathbf{u})$$

This parameter refers to quality of the subdomain solves ($\omega = 1$ corresponds to exact subdomain solves). This assumption also restrains us from simply *scaling* $a_i(\cdot, \cdot)$ to decrease C_0. Note, our coarse grid subdomains are recursive applications of multigrid, thus $\omega \neq 1$, except on the penultimate grid, though as we use a direct solver on the local subdomains $\omega = 1$.

For a linear operator $\mathbf{L}$, which is self adjoint with respect to $a(\cdot, \cdot)$, we use the Rayleigh quotient characterization of the extreme eigenvalues.

$$\lambda^A_{\min}(\mathbf{L}) = \min_{\mathbf{u} \neq 0} \frac{a(\mathbf{L}\mathbf{u}, \mathbf{u})}{a(\mathbf{u}, \mathbf{u})}, \qquad \lambda^A_{\max}(\mathbf{L}) = \max_{\mathbf{u} \neq 0} \frac{a(\mathbf{L}\mathbf{u}, \mathbf{u})}{a(\mathbf{u}, \mathbf{u})}$$

The condition number of $\mathbf{L}$ is thus given by $\lambda^A_{\max}(\mathbf{L})/\lambda^A_{\min}(\mathbf{L})$. As the bound on the number of iteration of the conjugate gradients method is proportional to the condition number of the preconditioned system BA, the abstract convergence bounds are derived by using Lemma 1 and the three assumptions to find expressions for this condition number $\lambda^A_{\max}(\mathbf{L})/\lambda^A_{\min}(\mathbf{L})$.

With this machinery we can state the bound on the condition number of the abstract additive Schwarz method, see Lemma 3 and Lemma 4 in [78] for the derivations

$$\mathcal{K}(\mathbf{BA}) \leq \omega[1 + \rho(\mathcal{E})]C_0^2$$

and for the abstract multiplicative Schwarz we have

$$\frac{\mathcal{K}(\mathbf{BA}) \leq [1+\omega^2\rho(\mathcal{E})^2]C_0^2}{2-\omega}$$

Lemma 1 in [78] shows that for the two level *overlapping* Schwarz methods (additive or multiplicative), with exact subdomain solves and subdomain overlap width of order H (the subdomain size), the condition number of the preconditioned system independent of H and h (the scale of discretization). Thus, this particular form of multigrid has optimal convergence characteristics within the context of the abstract convergence theory.

Chapter 4

High performance linear equation solvers for finite element matrices

The previous chapter introduced the ingredients used in multilevel iterative equation solvers. This chapter completes the context in which our work resides by discussing the particular methods that have been developed for the large unstructured sparse matrices that are of interest to the finite element community. Scalable solvers for unstructured finite element problems is an active area of research. This section provides a brief overview of current promising methods.

4.1 Introduction

We are interested in scalable technology i.e., we are interested in methods that have the potential to run in time $O(n)$ sequentially and polylogorithmically in parallel ($O(\log^k n)$ for a constant k). Thus, we look only at multilevel methods.

We segregate the major methods in three (somewhat arbitrary) categories: geometric multigrid, algebraic multigrid, and domain decomposition methods. We distinguish between algebraic (§4.2) and geometric (§4.3) multigrid methods by calling a method algebraic if it uses the matrix values in the construction of its restriction operators. We also discuss notable domain decomposition methods that are not multigrid methods (§4.4).

4.2 Algebraic multigrid

Again, we define algebraic methods as methods that use the *values* of the matrix entries in the construction of the multigrid restriction operators. Algebraic multigrid methods are an active area of research within the multigrid community as they provide an avenue toward "black box" scalable solvers for PDEs on unstructured meshes. There are three components to an algebraic (or indeed any) unstructured multigrid method: selection criterion for the coarse grid "points", construction of the restriction (and interpolation) operators or functions, and the method for construction of the coarse grid operators. The A_{i+1} coarse grid operators, in algebraic methods, are usually formed in the standard Galerkin or variational way ($A_{i+1} = R_i A_i R_i^T$). Thus, the coarse grid point selection and restriction function formulation are the only distinguishing aspects of an algebraic method.

Algebraic methods were first introduced by Ruge in 1986 [71], and tested on finite element matrices including thin body elasticity and incompressible materials. The results of these early algebraic methods were not very promising but they did set the stage for effective modern methods that we discuss in this section. Before we describe these algorithms we describe the general approach that many of these methods employ.

Many of the algebraic methods that we are aware of use some type of "heavy edge" (edges are off-diagonal stiffness matrix entries) heuristic to "agglomerate" vertices in the selection of the coarse grid point set [83, 32] (much like the method in [19] discussed in §4.3). These methods intend to keep tightly coupled vertices connected to each other via maximal matching algorithms [32] or "strongly coupled neighborhood of a node" [83]. Once these small (algebraic) regions have been defined interpolation functions, with compact support, are then constructed in some fashion.

4.2.1 A promising algebraic method

Vanek *et. al.* [83] constructed an algebraic algorithm that is particularly good at handling thin body elasticity and lightly supported structures e.g., a plate with Dirichlet boundary conditions on a small portion of the boundary. The algorithm proceeds as follows:

- Construct *strongly coupled neighborhood of nodes* - this approach is meant to mimic the behavior of semicoarsening [28] for anisotropic and stretched grids.

- Use these groups to construct a constant interpolation function, that is a function with a constant value at all of the vertices in the group and zero everywhere else. These functions are the translational rigid body modes for the group of vertices.

- These functions are not ideal as they have very high energy (i.e., $a(\mathbf{u}, \mathbf{u})$ is large), this results in a high bound on the convergence rate via the C_0 term of "assumption 1" in (§3.4.3). This is because these functions are "sharp" - a natural approach is to "smooth" these functions with a simple iterative scheme (i.e., *smoother* in multigrid terminology). Notice though that the support of these functions grows by one "layer" of vertices in each iteration, thus the overuse of this smoothing results in high computational complexity in applying them and more importantly a higher complexity of the coarse grid operators as they are constructed from the fine grid operator and these interpolation operator. Vanek *et. al.* thus apply only one step of the smoother (using a slightly altered operator) to produce the "simple" version of their method.

- For problems in elasticity, and fourth order problems, their results are dramatically improved with the use of geometric information in the form of what seems to amount to vertex coordinates. The vertex coordinates are used to orthogonalized their initial guess against user-provided polynomials or against the rigid body modes that can be inferred from these vertex coordinates. This approach of removing the rigid body modes is similar to many successful methods [19, 77, 63].

4.3 Geometric approach on unstructured meshes

To apply classic multigrid techniques to an unstructured grid one is faced with two main design decisions. Recall from §3.3 that for regular grids, the coarse grids are known *a priori*, hence the restriction/interpolation and coarse grid operators are known implicitly. For unstructured grids however the coarse grids must be explicitly constructed - after which standard finite element shape functions can be used to construct the restriction operator. The first design decision is whether the coarse grids are provided by the finite element code, or are constructed within the solver from the fine mesh.

The second decision, that often follows from the first, is whether to construct the coarse grid operators algebraically (Galerkin coarse grids) or let the finite element implementation generate the coarse grid operators. There are advantages and disadvantages in either approach. The Galerkin approach is harder to implement efficiently and requires more communication (as an "element" can not be redundantly calculated since there are no explicit elements). On the other hand the Galerkin approach is more robust because there is no need to take derivatives of the coarse grid shape functions; thus the coarse grids need not be as "good" e.g., zero volume elements are allowed since they do not have any fine grid vertices within them by construction. Also nonlinear elements can be more easily accommodated as some element formulations (e.g., large deformation plasticity) are not robust with poorly proportioned low order tetrahedra. Nonlinear materials can be accommodated with Full Approximation Storage (FAS) methods [13, 40] where the current solution and right hand side are restricted to the coarse mesh and not just the current residual - the coarse grid elements thus retain state variables in nonlinear materials.

An additional advantage to the Galerkin approach with nonlinear problems is that regions of localized softening may not appear on the coarser grids if a new finite element problem is formed. However with the Galerkin approach a region of localized nonlinearity will contribute to the coarse grid stiffness matrix. Thus Galerkin coarse grids provide, in a sense, a higher order of approximation as they sample the function (on the fine grid) at more points. Our method is novel in that it constructs a *geometric* coarse grid automatically, thus relieving the finite element user for this burden; we use Galerkin coarse grids because of their desirable properties and as they can be constructed automatically within the solver.

4.3.1 Promising geometric approaches

The work in applying geometric forms of multigrid to unstructured problems in continuum mechanics has come primarily from the engineering community. Many of these methods require an explicit finite element mesh for the coarse grids, supplied by the user. Most of this work uses methods that require that coarse meshes (indeed the entire problem) be provided by the user; e.g., see [30] for second order PDEs, [39] for fourth order PDEs, and for nonlinear problems [40, 51, 69] (note [69] uses Galerkin coarse grids). These methods have the same structure as classical multigrid methods (§4.3), as does our method, and good results can be achieved for the problems that can be addressed with these methods (i.e., problems where the coarse meshes can be effectively constructed). These methods have practical difficulties on large problems with complex boundaries and material interfaces, as the (small) size of the coarse grids, required for efficiency, can be so small that the geometry of the original problem can not be well represented. Thus some type of *approximate* mesh must be constructed, for these coarse problems, but a *mesh generator* is not in general equipped to produce such meshes. These methods are effective however when the geometry/scale of the problem is such that all of the coarse meshes can be effectively constructed (and the user is willing to do so).

Another method that falls into *our* definition of geometric multigrid but could also be called an algebraic method is that of using rigid body modes (Bulgakov *et. al.*[19]), in the construction of the restriction operators which are used for the standard Galerkin coarse grid operators. This method is notable in that it has an algebraic architecture so that the user must only provide the fine mesh. This method proceeds by partitioning vertices in the finer mesh into small sets of connected vertices - these sets then produce a coarse grid space with constant interpolation. The displacement and rotational rigid body modes of these aggregates is then used to provide interpolation values from the fine (aggregated) vertices to the coarse vertex. This method is similar to that described in [83] with the addition of rotational modes, discussed in §4.2.

A theoretical drawback of this approach is that the (discrete) interpolation functions have disjoint support, the actual interpolation functions have high energy, and the constant C_0 in "assumption 1" of §3.4.3 is very large. The numerical results however, show promising results.

4.4 Domain decomposition

As noted previously, domain decomposition methods have a rich history in the structural engineering community [70] as a way of organizing the direct solution of large complex structures. These methods, also known as nested dissection vertex ordering factorizations, can benefit from the reuse of factorizations from duplicate substructures. These direct methods explicitly form a Schur complement - its factorization being the primary cost of these methods. Multilevel domain decomposition methods use a two level scheme such as Keyes [55] for fluid problems, and rely on a powerful overlapping domain decomposition smoother to ameliorate the effects of using a single (very) coarse grid. In the past 15 years iterative solutions of the Schur complement have been investigated extensively (see [63] and the references therein). Here we mention only one method which has been well developed and tested on structures problems.

4.4.1 A domain decomposition method

The finite element tearing and interconnect method (FETI) developed by Farhat *et. al.*[35, 34] is an iterative substructuring method that "tears" the monolithic problem into a series of subdomains. These subdomains are "interconnected" via Lagrange multipliers - thus the Schur complement is only solving for the Lagrange multipliers (these become the primary variables in the solve).

As the subdomains are given Neumann boundary conditions on the artificial (i.e., Γ_i in §2.4.1) the subdomain are singular in general. Thus, a pseudo-inverse [27] is used for the subdomain solves. The resulting Schur complement equations need to be augmented by applying a constraint, again applied with Lagrange multipliers, that the applied load minus the Lagrange multiplier (on each floating subdomain) is orthogonal to the rigid body modes of the subdomains. Thus the Schur complement equations must be augmented with Lagrange multipliers to enforce the rigid body constraint. This final set of equations is solved with a *projected* conjugate gradient algorithm on the "primary" Lagrange multipliers where the rigid body component of the residual is projected out in each iteration.

Preconditioning is required for the original Schur complement only and can be implemented using any of the standard iterative substructuring methods [78]. Thus the FETI method is a non-overlapping domain decomposition method in the dual space of the problem. The drawback of FETI is that the coarse grid is not in the same *form* as the fine grid and hence FETI can not be applied *recursively* - this fact prevents one from developing an optimal complexity multilevel FETI algorithm by simply solving the coarser problems by another application of FETI.

Chapter 5

Our method

This chapter, and the numerical results in chapter 6, describe and discuss the core issues of our algorithms in terms of convergence rates, serial performance, and some simple PRAM parallel complexity. The second half of this dissertation (chapter 7 to the end) discusses parallel performance issues in more depth and verifies our claim of scalability with performance results with our largest test problems.

This chapter discusses our main contributions to serial multigrid methods for 3D finite element problems on unstructured grids. We have developed heuristics to optimize the quality of the vertex sets that are promoted to the coarse grid at all level in multigrid, and techniques to modify the Delaunay tessellation on these sets, to optimize the solution time for solid mechanics problems on unstructured finite element meshes with large jumps in material coefficients, complex geometries, thin body domains, and poorly shaped elements. This chapter also discusses our new practical and highly optimal (in PRAM) parallel maximal independent set algorithm.

5.1 Introduction

This chapter discussed the technical details of the multigrid method that we use and is the core of this dissertation. As we have seen in §3.3 multigrid is a very powerful solution method. While multigrid is used extensively for structured meshes, the use of multigrid on unstructured meshes is less wide spread. This is due to the fact that the construction of unstructured meshes is a challenging endeavor. Thus, the effective construction of coarse grids for unstructured problems is not a well developed subject, though an active area of research, and is the primary goal of this dissertation.

Our work is based on a method developed by Guillard [44], and independently by Chan and Smith [23]. This method relies on

- *Maximal independent sets* to automatically select vertices to be "promoted" to the next coarse grid,
- Automatic mesh generation techniques to construct the coarse grids, using these vertices,
- Coarse grid spaces constructed with finite element shape functions,
- Galerkin coarse grid operators.

Our approach has the advantage that it is relatively independent of the finite element implementation, i.e. the solver interface with the finite element code is relatively small. The finite element code need *only* provide nodal coordinates as well as the stiffness matrix itself. We also use element connectivity information and material type indices (to identify material interfaces). This method can thus be interfaced with an existing finite element code with relatively little difficulty, as the finite element code need only provide the finite element mesh to the solver.

Solvers should be highly modular to be useful to the finite element community. This is of primary importance as the finite element codes are large and complex systems that have, in many cases, been developed over decades. The method that we use satisfies our need for a highly modular and "optimal" finite element solver.

Our method was previously developed in serial for 2D linear elasticity applications. We have extended the method to parallel computers for applications in 3D continuum mechanics. The method has on four basic components:

1. **Coarsening**: Multigrid's coarse grids should capture the low frequency eigenvectors effectively. *Maximal independent sets* (MISs) are often an effective, and popular, heuristic for capturing the low energy modes of unstructured finite element problems. We describe a new algorithm for the parallel construction of an MIS in §5.2 [2], as well as a set of heuristics to improve the quality of the MIS in §5.3[1]. Note that there is no fundamental reason to "promote" vertices on the fine mesh to be in the coarse mesh, i.e. multigrid does not require "node-nested" coarse grids, see [23] for

numerical experiments with dual coarse grids; also, we take advantage of this in one of our methods.

2. **Mesh**: After the vertices have been created we need a mesh; the mesh is used to construct the finite element function space. Thus we need a mesh generator; our mesh generation is discussed in §5.4.

3. **Restriction/Interpolation Operators**: After the coarse meshes are created one can use standard finite element shape functions for the restriction and interpolation operators. This is described in §5.5.

4. **Coarse Grid Operators**: Coarse grid operators can be constructed in one of two ways as discussed in §4.3. §5.6 describes the construction of Galerkin coarse grid operators on parallel computers.

Also, general issues of smoothers (or preconditioners for our smoothers) are discussed in §5.7 .

5.2 A parallel maximal independent set algorithm

An *independent set* is a set of vertices $I \subseteq V$ in a graph $G = (V, E)$, in which no two members of I are adjacent (i.e. $\forall v, w \in I, (v, w) \notin E$); a *maximal independent set* (MIS) is an independent set for which no proper superset is also an independent set. The parallel construction of an MIS is useful in many computing applications, such as graph coloring and coarse grid creation for multigrid algorithms on unstructured finite element meshes. In addition to requiring an MIS (which is not unique), many of these applications want an MIS that maximizes a particular application dependent quality metric. Finding the optimal solution in many of these applications is an NP-complete problem i.e., they can not be solved in polynomial time or can be solved by a nondeterministic(N) machine in polynomial(P) time [49]; for this reason greedy algorithms in combination with heuristics are commonly used for both the serial and parallel construction of MISs. Many of the graphs of interest arise from physical models, such as finite element simulations. These graphs are sparse and their vertices are connected to only their *nearest neighbors*. The vertices of such graphs have a bound Δ on their maximum degree. We discuss our method of attaining $O(1)$ PRAM (see 1.7) [41] complexity bounds for computing an MIS on such graphs, namely finite element models in three dimensional solid mechanics. Our algorithm is notable in that it does not rely on global random vertex ordering (see [50, 60] and the references therein) to achieve correctness in a distributed memory computing environment but explicitly uses knowledge of the graph partitioning to provide for the correct construction of an MIS in an efficient manner.

The complexity model of our algorithm also has the attractive attribute that it requires far fewer processors than vertices, in fact we restrict the number of processors used in order to attain optimal complexity. Our PRAM model uses $P = O(n)$ processors ($n = |V|$) to compute an MIS, but we restrict P to be at most a fixed fraction of n to attain the optimal theoretical complexity. The upper bound on the number of processors is however far more than the number of processors that are generally used in practice on common distributed memory computers of today; so given the common use of relatively *fat* processor nodes in modern computers, our theoretical model allows for the use of many more processors than one would typically use in practice. Thus, in addition to obtaining optimal PRAM complexity bounds, our complexity model reflects the way that modern machines are actually used. Our numerical experiments confirm our $O(1)$ complexity claim.

We do not include the complexity of the graph partitionings in our complexity model, though our method explicitly depends on these partitions. We feel justified in this as it is reasonable to assume that the MIS program is embedded in a larger application that requires partitions that are usually much better than the partitions that we require.

5.2.1 An asynchronous distributed memory algorithm.

Consider a graph $G = (V, E)$ with vertex set V, and edge set E, an edge being an unordered pair of distinct vertices. Our application of interest is a graph which arises from a finite element analysis, where elements can be replaced by the edges required to make a clique of all vertices in each element, see Figure 5.1. Finite element methods, and indeed most discretization methods for PDEs, produce graphs in which vertices only share an edge with their nearest physical neighbors, thus the degree of each vertex $v \in V$ can be bounded by some modest constant Δ. We restrict ourselves to such graphs in our complexity analysis. Furthermore to attain our complexity bounds we must also assume that vertices are "partitioned well" (which is defined later) across the machine.

We introduce our algorithm by first describing the basic random greedy MIS algo-

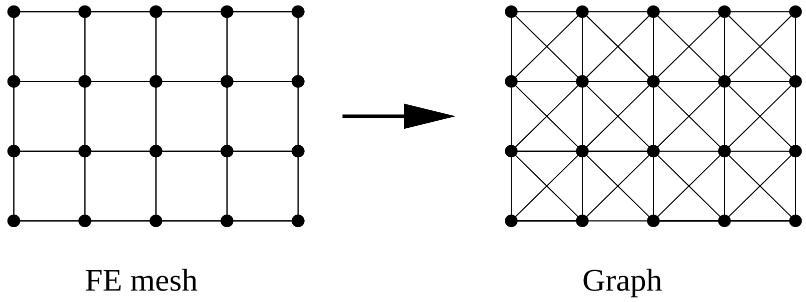


Figure 5.1: Finite element quadrilateral mesh and its corresponding graph

rithms described in [60]. We utilize an *object oriented* notation from common programming languages, as well as set notation, in describing our algorithms; this is done to simplify the notation and we hope it does not distract the uninitiated reader. We endow vertices v with a mutable data member $state$, $state \in \{selected, deleted, undone\}$. All vertices begin in the *undone* state, and end in either the *selected* or *deleted* state; the MIS is defined as the set of *selected* vertices. Each vertex v is also given a list of adjacencies $adjac$.

$Definition$ 1: The adjacency list for vertex v is defined by
$v.adjac = \{v1 \mid (v, v1) \in E\}$

We also assume that $v.state$ has been initialized to the *undone* state for all v and $v.adjac$ is as defined in $Definition$ 1 in all of our algorithm descriptions. With this notation in place we show the basic MIS algorithm (BMA) in Figure 5.2.

forall $v \in V$
  if $v.state = undone$ **then**
    $v.state \leftarrow selected$
    forall $v1 \in v.adjac$
      $v1.state \leftarrow deleted$
$I \leftarrow \{v \in V \mid v.state = selected\}$

Figure 5.2: Basic MIS algorithm (BMA) for the serial construction of an MIS

For parallel processing we partition the vertices onto processors and define the vertex set V_p owned by processor p of P processors. Thus $V = V_1 \cup V_2 \cup \ldots \cup V_P$ is a disjoint union, and for notational convenience we give each vertex an immutable data member *proc* after the partitioning is calculated to indicate which processor is responsible for it. Define the *edge separator* set E^S to be the set of edges (v, w) such that $v.proc \neq w.proc$. Define the *vertex separator* set $V^S = \{v \mid (v, w) \in E^S\}$ (G is undirected, thus (v, w) and (w, v) are equivalent). Define the *processor vertex separator* set, for processor p, by $V_p^S = \{v \mid (v, w) \in E^S \text{ and } (v.proc = p \text{ or } w.proc = p)\}$. Further define a processors *boundary* vertex set by $V_p^B = V_p \cap V^S$, and a processors *local* vertex set by $V_p^L = V_p - V_p^B$. Our algorithm provides for correctness and efficiency in a distributed memory computing environment by first assuming a given ordering or numbering of processors so that we can use inequality operators with these processor numbers. As will be evident later, if one vertex is placed on each processor (an activity of theoretical interest only), then our method will degenerate to one of the well known random types of algorithms [60].

We define a function $mpivs(vertex\_set)$ (an acronym for "maximum processor in vertex set"), which operates on a list of vertices:

Definition 2:

$$mpivs(vertex\_set) = \left\{ \begin{array}{ll} max\{v.proc \mid v \in vertex\_set, v.state \neq deleted\} & if \quad vertex\_set \neq \emptyset \\ -\infty & if \quad vertex\_set = \emptyset \end{array} \right\}$$

Given these definitions and operators, our algorithm works by implementing two rules within the BMA running on processor p, as shown below.

- *Rule* 1: Processor p can *select* a vertex v only if $v.proc = p$.
- *Rule* 2: Processor p can *select* a vertex v only if $p \geq mpivs(v.adjac)$.

Note that *Rule* 1 is a *static* rule, because *v.proc* is immutable, and can be enforced simply by iterating over V_p on each processor p when looking for vertices to select. In contrast, *Rule* 2 is *dynamic* because the result of $mpivs(v.adjac)$ will in general change (actually monotonically decrease) as the algorithm progresses and vertices in $v.adjac$ are deleted.

5.2.2 Shared memory algorithm

Our Shared Memory MIS Algorithm (SMMA) in Figure 5.3, can be written as a simple modification to BMA.

We have modified the vertex set that the algorithm running on processor p uses (to look for vertices to select), so as to implement *Rule* 1. We have embedded the basic

while $\{v \in V_p \mid v.state = undone\} \neq \emptyset$

  forall $v \in V_p$ - - implementation of *Rule* 1

    if $v.state = undone$ **then**

      if $p \geq mpivs(v.adjac)$ **then** - - implementation of *Rule* 2

        $v.state \leftarrow selected$

        forall $v1 \in v.adjac$

          $v1.state \leftarrow deleted$

$I \leftarrow \{v \in V \mid v.state = selected\}$

Figure 5.3: Shared memory MIS algorithm (SMMA) for MIS, running on processor p

algorithm in an iterative loop and added a test to decide if processor p can select a vertex, for the implementation of *Rule* 2. Note, the last line of Figure 5.3 may delete vertices that have already been deleted, but this is inconsequential.

There is a great deal of flexibility in the order which vertices are chosen in each iteration of the algorithm. Herein lies a simple opportunity to apply a heuristic, as the first vertex chosen is always *selectable* and the probability is high that vertices which are chosen early is also selectable. Thus if an application can identify vertices that are "important" then those vertices can be ordered first and so that a less important vertex can not delete a more important vertex. For example, in the automatic construction of coarse grids for multigrid equation solvers on unstructured meshes one would like to give priority to the boundary vertices [1]. This is an example of a *static* heuristic, that is a *ranking* which can be calculated initially and does not change as the algorithm progresses. *Dynamic* heuristics are more difficult to implement efficiently in parallel. An example is the saturation degree ordering (SDO) used in graph coloring algorithms [14]: SDO colors the vertex with a maximum number of different colored adjacencies; the degree of an uncolored vertex increases as the algorithm progresses and its neighbors are colored. We know of no MIS application, that does not have a quality metric to maximize - thus it is of practical importance that an MIS algorithm can accommodate the use of heuristics effectively. Our method can still implement the "**forall**" loops, with a serial heuristic, i.e. we can iterate over the vertices in V_p in any order that we like. To incorporate static heuristics globally (i.e. a ranking of vertices), one needs to augment our rules and modify SMMA, see [1] for details, but in doing so we lose our complexity bounds, in fact if one assigns a random rank to all vertices this algorithm would degenerate to the random algorithms described in [60, 50].

To demonstrate correctness of SMMA we proceed as follows: show termination; show that the computed set I is maximal; and show that independence of $I = \{v \in V \mid v.state = selected\}$ is an invariant of the algorithm.

- Termination is simple to prove and we do so in §5.2.4.
- To show that I is maximal we can simply note that if $v.state = deleted$ for $v \in V$, v must have a selected vertex $v1 \in v.adjac$ as the only mechanism to delete a vertex is to have a selected neighbor do so. All deleted vertices thus have a selected neighbor and they can not be added to I and maintain independence, hence I is maximal.
- To show that I is always independent first note that I is initially independent - as I is initially the empty set. Thus it suffices to show that when v is added to I, in line 7 of Figure 5.3, no $v1 \in v.adjac$ is selected. Alternatively we can show that $v.state \neq deleted$ in line 7, since if v can not be deleted then no $v1 \in v.adjac$ can be selected. To show that $v.state \neq deleted$ in line 7 we need to test three cases for the processor of a vertex $v1$ that could delete v:
  - Case 1) $v1.proc < p$: v would have blocked $v1.proc$ from selecting $v1$, because $mpivs(v1.adjac) \geq v.proc = p > v1.proc$, so the test on line 6 would not have been satisfied for $v1$ on processor $v1.proc$.
  - Case 2) $v1.proc = p$: v would have been deleted, and not passed the test on line 5, as this processor selected $v1$ and by definition there is only one thread of control on each processor.
  - Case 3) $v1.proc > p$: as $mpivs(v.adjac) \geq v1.proc > p$ thus $p \not\geq mpivs(v.adjac)$ the test on line 6 would not have succeeded, line 7 would not be executed on processor p.

Further we should show that a vertex v with $v.state = selected$ can not be deleted, and $v.state = deleted$ can not be selected. For a v to have been selected by p it must have been selectable by p (i.e. $\{v1 \in v.adjac \mid v1.proc > p,\ v1.state \neq deleted\} = \emptyset$). However for another processor $p1$ to delete v, $p1$ must select $v1$ ($p1 = v1.proc$), this is not possible since if neither v nor $v1$ are deleted then only one processor can satisfy line 6 in Figure

5.3. This consistency argument is developed further in §5.2.4. Thus, we have shown that $I = \{v \in V \mid v.state = selected\}$ is an independent set and, if SMMA terminates, I is maximal as well.

5.2.3 Distributed memory algorithm

For a distributed memory version of this algorithm we use a message passing paradigm and define some high level message passing operators. Define $send(proc, X, Action)$ and $receive(X, Action)$ - $send(proc, X, Action)$ sends the object X and procedure $Action$ to processor $proc$, $receive(X, Action)$ receives this message on processor $proc$. Figure 5.4 shows a distributed memory implementation of our MIS algorithm running on processor p. We have assumed that the graph has been partitioned to processors 1 to P, thus defining V_p, V_p^S, V_p^L, V_p^B, and $v.proc$ for all $v \in V$.

while $\{v \in V_p \mid v.state = undone\} \neq \emptyset$
    forall $v \in V_p^B$ - - implementation of *Rule* 1
        if $v.state = undone$ **then**
            if $p \geq mpivs(v.adjac)$ **then**   - - implementation of *Rule* 2
                $Select(v)$
                $proc\_set \leftarrow \{proc \mid v \in V_{proc}^S\} - p$
                forall $proc \in proc\_set$ $send(proc, v, Select)$
    while $receive(v, Action)$
        if $v.state = undone$ **then** $Action(v)$
    forall $v \in V_p^L$ - - implementation of *Rule* 1
        if $v.state = undone$ **then**
            if $p \geq mpivs(v.adjac)$ **then**   - - implementation of *Rule* 2
                $Select(v)$
                if $v1 \in V^B$ **then**
                    $proc\_set \leftarrow \{proc \mid v1 \in V_{proc}^S\} - p$
                    forall $proc \in proc\_set$ $send(proc, v1, Delete)$
$I \leftarrow \{v \in V \mid v.state = selected\}$

Figure 5.4: Asynchronous distributed memory MIS algorithm (ADMMA) on processor p

procedure $Select(v)$

$v.state \leftarrow selected$

forall $v1 \in v.adjac$

$Delete(v1)$

procedure $Delete(v1)$

$v1.state \leftarrow deleted$

Figure 5.5: ADMMA "Action" procedures running on processor p

A subtle distinction must now be made in our description of the distributed memory version of the algorithm in Figure 5.4 and 5.5: vertices (e.g. v and $v1$) operate on local copies of the objects and not to a single shared object. So, for example, an assignment to $v.state$ refers to assignment to the local copy v on processor p. Each processor has a copy of the set of vertices $V_p^E = V_p \cup V_p^S$, i.e. the local vertices V_p and one layer of "ghost" vertices. Thus all expressions refer to the objects (vertices) in processor p's local memory.

Note that the value of $mpivs(v.adjac)$ monotonically decreases as $v1.state$ ($v1 \in v.adjac$) are deleted, thus as all tests to select a vertex, are of the form $p \geq mpivs(v.adjac)$ some processors have to wait for other processors to do their work (i.e. select and delete vertices). In our distributed memory algorithm in Figure 5.4 the communication time is added to the time that a processor may have to wait for work to be done by another processor; this does not effect the correctness of the algorithm but it may effect the resulting MIS. Thus ADMMA is not a *deterministic* MIS algorithm; although the synchronous version - that we use for our numerical results - is deterministic for any given partitioning.

The correctness for ADMMA can be shown in a number of ways, but first we define a *weaving monotonic path* (WMP) as a path of length t in which each consecutive pair of vertices ($(v_i,v_j) \in E$) satisfies $v_i.proc < v_j.proc$, see Figure 5.6.

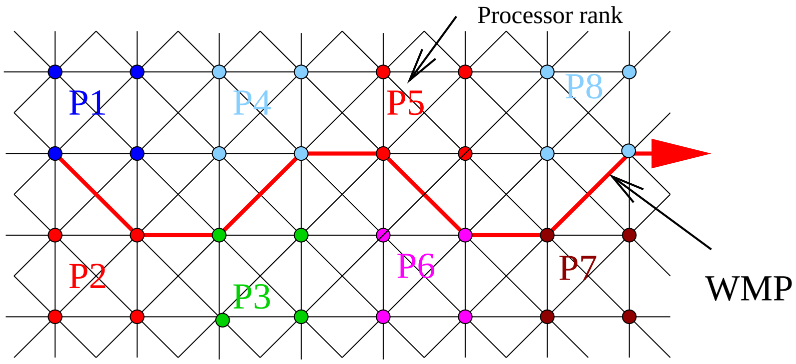


Figure 5.6: Weaving monotonic path (WMP) in a 2D FE mesh

One can show that the semantics of ADMMA run on a partitioning of a graph is equivalent to a random algorithm with a particular set of "random" numbers. Alternatively, we can use the correctness argument from the shared memory algorithm and show consistency in a distributed memory environment. To do this first make a small isomorphic transformation to ADMMA in Figure 5.4:

- Remove $v.state \leftarrow selected$ in Figure 5.5, and replace it with a memoization of v so as to avoid "selecting" v again.
- Remove the "Select" message from Figure 5.4 and modify the *Select* procedure in Figure 5.5 to send the appropriate "Delete" messages to processors that "touch" $v1 \in v.adjac$.
- Redefine I to be $I = \{v \in V \mid v.state \neq deleted\}$ at the end of the algorithm.
- Change the termination test to: **while** $(\exists v1 \in v.adjac \mid v \in V_p, v.state \neq deleted, v1.state \neq deleted)$, or simply **while** ($I$ is not independent).

This does not change the semantics of the algorithm but removes the *selected* state from the algorithm and makes it mathematically simpler (although less concrete of a description). Now only *Delete* messages need to be communicated and $v.state \leftarrow deleted$ is the only change of state in the algorithm. Define the directed graph $G^{WMP} = (V^S, E^{WMP})$, $E^{WMP} = \{(v, w) \in E \mid v.proc < w.proc\}$; in general G^{WMP} is a forest of acyclic graphs. Further define $G^{WMP}_p = (V^S_p, E^{WMP}_p)$, $E^{WMP}_p = \{(v, w) \in E \mid w.state \neq deleted, v.proc = p\}$. G^{WMP}_p is the current local view of G^{WMP} with the edges removed for which the "source" vertices have been deleted. *Rule* 2 can now be restated: processor p can only select a vertex that is not the end of an edge in E_p^{WMP}. Processors deletes "down stream" edges in E^{WMP} and send messages so as other processors can delete their "up stream" copies of these edges, thus G_p^{WMP} is pruned as p deletes vertices and receives delete messages. Informally, consistency for ADMMA can be inferred as the only information flow (explicit delete messages between processors) moves down acyclic graphs in G^{WMP}; as the test, $(\forall v1 \in v.adjac \mid v1.proc \leq p$ or $v1.state = deleted)$ for processor p to select a vertex v, requires that *all* edges (in E^{WMP}) *to* v are "deleted". Thus the order of the reception of these delete messages is inconsequential and there is no opportunity for race conditions or ambiguity in the results of the MIS. More formally we can show that these semantics insure that $I = \{v \in V \mid v.state \neq deleted\}$ is maximal and independent:

- I is independent as no two vertices (in I) can remain dependent forever. To show this we note that the only way for a processor p to *not* be able to select a vertex v is for v to have a neighbor v_1 on a higher processor. If v_1 is deleted then p is free to "select" v. Vertex v_1 on processor p_1 can in tern be selected unless it has a neighbor v_2 on a higher processor. Eventually the end of this WMP is reached and processor p_t processes v_t and thus releases p_{t-1} to select v_{t-1} and on down the line. Therefore no pair of undone vertices remains, and I is eventually be independent.

- I is maximal as the only way for a vertex to be deleted is to have a selected neighbor. To show that no vertex v that is "selected" can ever be deleted, as in our shared memory algorithm, we need to show that three types of processors p_1 with vertex v_1 can not delete v.

  - For $p = p_1$: we have the correctness of the serial semantics of BMA to ensure correctness, i.e. v_1 would be deleted and p would not attempt to select it.

  - For $p > p_1$: p_1 will not pass the $mpivs(v_1.adjac)$ test as in the shared memory case.

  - For $p < p_1$: p does not pass the $mpivs(v_1.adjac)$ and will not "select" v in the first place.

Thus I is maximal and independent.

5.2.4 Complexity of the asynchronous maximal independent set algorithm

In this section we derive the complexity bound of our algorithm under the PRAM computational model. To understand the costs of our algorithm we need to bound the cost of each outer iteration, as well as, bound the total number of outer iterations. To do this we first make some restrictions on the graphs that we work with and the partitions that we use. We assume that our graphs come from physical models, that is vertices are only connected by an edge to its nearest neighbors so the maximum degree Δ of any vertex is bounded. We also assume that our partitions satisfy a certain criterion (for regular meshes we can illustrate this criterion with regular rectangular partitions and a *minimum* logical dimension that depends only on the mesh type). We can bound the cost of each outer iteration by requiring that the sizes of the partitions are independent of the total number of vertices n. Further we assume that the asynchronous version of the algorithm is made synchronous by including a barrier at the end of the "receive" **while** loop, in Figure 5.4, at which point all messages are received and then processed in the next **forall** loop. This synchronization is required to avoid more than one leg of a WMP from being processed in each outer iteration. We need to show that the work done in each iteration on processor p is of order N_p ($N_p = |V_p|$). This is achieved if we use $O(n)$ processors and can bound the *load balance* (i.e. $max\{N_p\}/min\{N_p\}$) of the partitioning.

LEMMA 3.1. *With the synchronous version of ADMMA, the running time of the PRAM version of one outer iteration in Figure 5.4 is* $O(1) = O(n/P)$, *if* $max\{N_p\}/min\{N_p\} = O(1)$

Proof. We need to bound the number of processors that *touch* a vertex v i.e. $|v|_{proc} \equiv \left|proc \mid v \in V^S_{proc}\right|$ In all cases $|v|_{proc}$ is clearly bounded by Δ. Thus, $max\ |v|_{proc} * N_p$ is $O(1)$ and is an upper bound (and a very pessimistic bound) on the number of messages sent in one iteration of our algorithm. Under the PRAM computational model we can assume that messages are sent between processors in constant time and thus our communication costs in each iteration is $O(1)$. The computation done in each iteration is again proportional to N_p and bounded by $\Delta * N_p$, the number of vertices times the maximum degree. This is also a very pessimistic bound that can be gleaned by simply following all the execution paths in the algorithm and successively multiplying by the bounds on all of the loops (Δ

and N_p). The running time for each outer iteration is therefore $O(1) = O(n * \Delta/P)$.

$\diamond$

Notice for regular partitions $|v|_{proc}$ is bounded by 4 in 2D, and 8 in 3D, and that for *optimal* partitions of large meshes $|v|_{proc}$ is about 3 and 4 for 2D and 3D respectively. The number of *outer* iterations, in Figure 5.4, is a bit trickier to bound. To do this we look at the mechanism by which a vertex *fails* to be selected.

LEMMA 3.2. *The running time in the PRAM computational model, of ADMMA, is bounded by the maximum length weaving monotonic path in G.*

Proof. To show that the number of outer iterations is proportional to the maximum length WMP in G, we need to look at the mechanism by which a vertex can *fail* to be selected in an iteration of our algorithm and thus potentially require an additional iteration. For a processor p_1 to fail to *select* a vertex v_1, v_1 must have an *undone* neighbor v_2 on a higher processor p_2. For vertex v_2 to not be selectable, v_2 in turn must have an *undone* neighbor v_3 on a higher processor p_3 and so on until v_t is the top vertex in the WMP. The vertex v_t at the end of a WMP is be processed in the first iteration as there is nothing to stop $v_t.proc$ from selecting or deleting v_t. Thus, in the next iteration, the top vertex v_t of the WMP has been either selected or deleted; if v_t was selected then v_{t-1} has been deleted and the *Undone* WMP (UWMP), a path in G^{WMP}, is at most of length $t-2$ after one iteration; and if v_t was deleted (the worst case) then the UWMP could be of at most length $t-1$. After t outer iterations the maximum length UWMP is of length zero, thus all vertices are selected or deleted. Therefore, the number of outer iterations is bounded by the longest WMP in the graph.

$\diamond$

COROLLARY 3.1. *ADMMA will terminate.*

Proof. Clearly the maximum length of a WMP is bounded by the number of processors P. By LEMMA 3.2 ADMMA will terminate in a maximum of P outer iterations.

$\diamond$

To attain our desired complexity bounds, we want to show that a WMP can not grow longer than a constant. To understand the behavior of this algorithm we begin with a few observation about regular meshes. Begin by looking at a regular partitioning of a 2D finite element quadrilateral mesh. Figure 5.6 shows a 2D mesh and a partitioning with regular blocks of four $(2*2)$ and a particular processor order. This is just small enough to allow for a WMP to traverse the mesh indefinitely, but clearly a nine $(3*3)$ vertex partitions would break this WMP and only allow it *walk* around partition intersections. Note that the $(2*2)$ case would require just the right sequence of events to happen on all processors for this WMP to actually govern the run time of the algorithm. On a regular 3D finite element mesh of hexahedra the WMP can *coil* around a line between four processors and the required partition size, using the same arguments as in the 2D case, would be five vertices on each side (or one more than the number of processors that share a processor interface line).

For irregular meshes one has to look at the mesh partitioning mechanism employed. Partitions on irregular meshes in scientific and engineering applications generally attempt to reduce the number of edges cut (i.e. $\left|E^S\right|$) and balance the number of vertices on each partition (i.e. $|V_p| * p/n \approx 1$). We assume that such a partitioner is in use and make a few general observations. First the partitions of such a mesh will tend to produce partitions in the shape of a hexagon in 2D for a large mesh with relatively large partitions. This is because the partitioner is trying to reduce the *surface* to *volume* ratio of each partition. These partitions are not likely to have skinny regions where a WMP could *jump* through the partition, and thus the WMP is relegated to following the lines of partition intersections. We do not present statistical or theoretical arguments as to the minimum partition size N that must be employed to bound the growth of a WMP for a given partitioning method; though clearly some constant N exists that, for a give finite element mesh type and a given reasonable partitioning method, will bound the maximum WMP length by a constant. This constant is roughly the number of partitions that come *close* to each other at some point, an optimal partitioning of a large D dimensional mesh will produce partitioning in which $D+1$ partitions meet at any given *point*. Thus, when a high quality mesh partitioner is in use, we would expect to see the algorithm terminate in at most four iterations on adequately well partitioned and sized three dimensional finite element meshes.

5.2.5 Numerical results

We present numerical experiments on an IBM SP with 80, 120 Mhz, Power2 (P2SC) processors at Argonne National Laboratory. An extended version of the Finite Element Analysis Program (FEAP)[36], is used to generate out test problems and produce our graphics. We use ParMetis [53] to calculate our partitions, and PETSc [10] for our parallel programming and development environment. Our code is implemented in C++, FEAP is implemented in FORTRAN, PETSc and ParMetis are implemented in C. We want to show that our complexity analysis is indicative of the actual behavior of the algorithm with real (imperfect) mesh partitioners. Our experiments confirm our PRAM complexity model is indicative of the performance one can expect with practical partitions on graphs of finite element problems. Due to a lack of processors we are not able to investigate the asymptotics of our algorithm throughly.

Our experiments are used to demonstrate that we do indeed see the behavior that our theory predicts. Additionally we use numerical experiments to quantify lower bound on the number of vertices per processor that our algorithm requires before growth in the number of outer iterations is observed. We use a parameterized mesh from solid mechanics for our test problem. This mesh is made of eight vertex hexahedral trilinear “brick” elements and is almost regular; the maximum degree Δ of any vertex is 26 in the associated graph. Figure 5.7 shows one mesh (13,882 vertices). The other meshes that we test are of the same physical model but with different scales of discretization (this problem is referred to as $P1$ in later chapters.

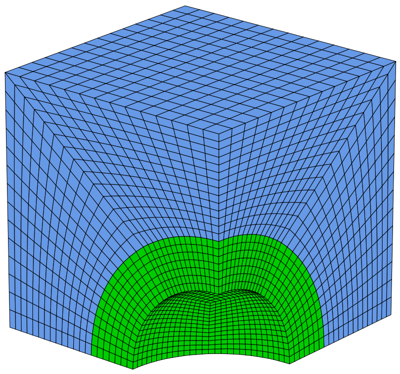


Figure 5.7: 13,882 vertex 3D FE mesh

We add synchronization to ADMMA on each processor by receiving *all* messages from neighboring processors in each iteration, to more conveniently measure the maximum length WMP that actually governs the number of outer iterations. Table 5.2.5 shows the results of the number of iterations required to calculate the MIS. Each case was run 10 times, as we do not believe that ParMetis is deterministic, but all 10 iteration counts were identical, thus it seems that this did not effect any of our results. A perfect partitioning of a large D-dimensional mesh with a large number of vertices per processor results in $D+1$ processors intersecting at a "point", and D partitions sharing a "line". If these meshes are optimal we can expect that the length of these lines (of partition boundaries) are of approximately uniform length. The length of these lines required to halt the growth of WMPs is $D+1$ vertices on an edge, as discussed in §5.2.4. If the approximate average size of each partition is that of a cube with this required edge length, then we would need about 64 vertices per partition to keep the length of a WMP from growing past 4. This assumes that we have a perfect mesh, which we do not, but none the less this analysis gives an approximate lower bound on the number of vertices that we need per processor to maintain our constant maximum WMP length.

	Processors									
Vertices	8	16	24	32	40	48	56	64	72	80
427	3	3	3	3	4	4	4	4	6	6
1,270	2	4	3	3	4	4	4	3	4	3
2,821	3	3	4	3	3	3	4	3	4	4
5,296	2	2	3	3	3	3	4	3	3	3
8,911	3	3	4	3	4	3	4	3	3	3
13,882	3	3	3	3	3	3	3	3	3	3

Table 5.1: Average number of iterations

Figure 5.8 shows a graphic representation of this data for all partitions. The growth in iteration count for constant graph size is reminiscent of the polylogarithmic complexity of *flat* or vertex based random MIS algorithms [50]. Although ParMetis does not specify the ordering of processors, it is not likely to be very random. These results show that the largest number of vertices per processor that "broke" the estimate of our algorithms complexity bound is about 7 (6.5 average) and the smallest number of vertices per processor that stayed at our bound of 4 iterations was also about 7 (7.3 average). To demonstrate our claim of $O(1)$ PRAM complexity we only require that there exists a bound N on the number of vertices per processor that is required to keep a WMP form growing beyond the region around a point where processors intersect. These experiments do not show any indication that that such a N does not exist. Additionally these experiments show that our bounds are quite pessimistic for the number of processors that we were able to use.

This data suggests that we are far away from the asymptotics of this algorithm, that is, we need many more processors to have enough of the longest WMPs so that one consistently governs the number of outer iterations.

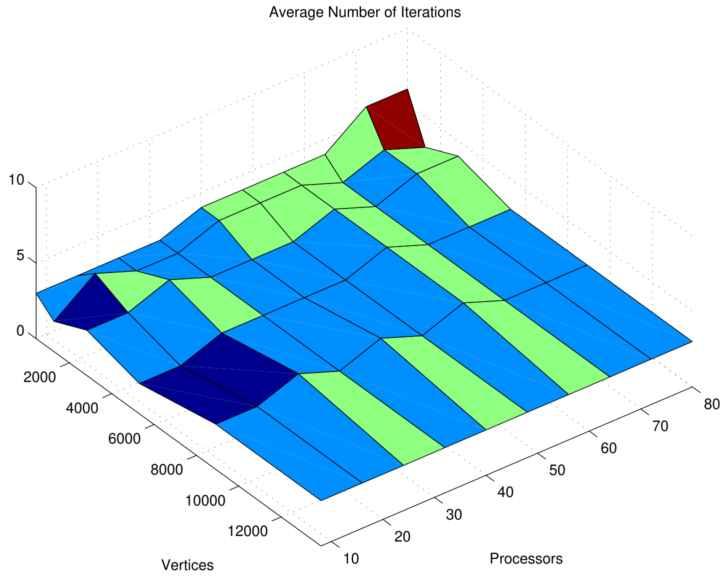


Figure 5.8: Average iterations vs. number of processors and number of vertices

5.3 Maximal independent set heuristics

This section discusses heuristics useful in optimizing the quality of multigrid restriction operators. These methods use coordinate data available in all finite element simulations, and we also employ some element data: element type (e.g. tetrahedra, quadrilateral shell, etc.) and element connectivity. We use this data to categorize topological elements of the finite element mesh, and use this information to modify the graph used in the MIS algorithm to improve solver performance. We also show how these heuristics can be applied globally on parallel platforms, as well as a simple method to get the coarse grids to more effectively "cover" the fine grids.

5.3.1 Automatic coarse grid creation with unstructured meshes

This section introduces the components that we use for the automatic construction of coarse grids on unstructured meshes, but first we state what we want our coarse grids to be able to do. The goal of the coarse grids in multigrid is to approximate the low frequency error in the current grid. Each successive grid's finite element function space should (with a drastically reduced vertex set) approximate, as best as it can, the highest frequency (or eigenfunctions) of the current grid. That is with say 10% of the vertices it is natural to expect that one could only represent the lowest 10% of the fine grid spectra well.

The coarse grid functions should approximate the *highest* part of this *lower* part of the spectrum as well as possible. It is not possible to satisfy this criterion directly (on unstructured grids), but a natural heuristic is to represent the geometry as well as possible with a much smaller set of vertices. One promising approach is to use computational geometry techniques to characterize features and maintain them on the coarser grids [81]. One popular method is to use a *maximal independent set* as a heuristic to evenly coarsen the vertex set, as discussed in the last section. If vertices are added to the MIS randomly then the MIS is expected to be a good representation of the fine grid in the sense of *evenly* coarsening the grid points and maintain the feature characteristics of the mesh. An MIS is not unique in general, and an arbitrary MIS is not likely to perform well as we show below, thus we use heuristics to improve performance.

We motivate our approach by first looking at the *structured* multigrid algorithm. We can characterize the behavior of multigrid on structured meshes, as shown in Figure 5.9, as: select every other vertex (starting from the boundary), in each dimension, for use in the coarse grid.

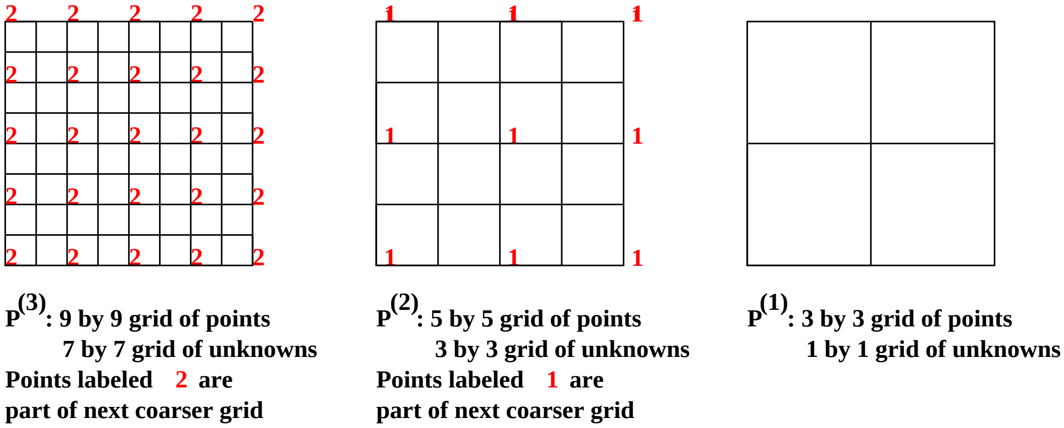


Figure 5.9: Multigrid coarse vertex set selection on structured meshes

To apply multigrid to unstructured meshes it is natural to try to imitate the behavior of the structured algorithm in hopes of imitating its success. Consider that, in addition to evenly coarsening the vertex set, the coarse grids in Figure 5.9 also emphasize the boundaries. One description of multigrid meshes on regular grids is: place each vertex v in a *topological category* of dimension d, for instance, corners ($d = 0$), edges ($d = 1$), surfaces ($d = 2$), and interiors ($d = 3$). Note, we overload *edges* here to mean a topological feature and *not* a graph edge - the type of "edge" should be obvious from the context in the following discussion. Given these categories we have collections of features for each category (e.g. a set of 3D connected surface vertices bounded by edge vertices would be one face in the set of faces in the problem). Notice that the regular mesh in Figure 5.9 produces an MIS *within* each feature. This section discusses algorithms to implement these observations and provide numerical experiments on model problems in linear elasticity. Note, Guillard and Chan and Smith also use simple 2D heuristics to preserve boundaries and emphasis corners [44, 23].

Maximal independent set algorithms

Recall the basic greedy MIS algorithm in Figure 5.10, which we discussed in §5.2.

There is a great deal of flexibility in the order that vertices are chosen in each iteration of the algorithm. Herein lies a simple opportunity to apply a heuristic, as the first

forall $v \in V$
  if $v.state = undone$ **then**
    $v.state \leftarrow selected$
    forall $v1 \in v.adjac$
      $v1.state \leftarrow deleted$
$I \leftarrow \{v \in V \mid v.state = selected\}$

Figure 5.10: Basic MIS algorithm for the serial construction of an MIS

vertex chosen is always *selectable* and the probability is high that vertices which are chosen early are also selectable. Thus if an application can identify vertices that are "important" then those vertices can be ordered first and so that a less important vertex can not delete a more important vertex. We can now decide that corners are more important than edges and edges are more important than surfaces and so on, and order the vertices with all corners first, then edges, etc. With this heuristic in place and the basic MIS algorithm in Figure 5.10 we can guarantees that the number of edge vertices on the coarse grid (in each edge segment) satisfies $\left|V_{edge}^{coarse}\right| \geq \frac{\left|V_{edge}^{fine}\right|-2}{3}$ for 2D meshes; whereas a valid MIS could remove *all* edge and corner vertices from the graph, which would be disastrous, as is shown in §5.3.7.

Parallel maximal independent set algorithms

The order in which each processor traverses the local vertex list can be governed by our heuristics although the global application of a heuristic requires an alteration in the MIS algorithm in §5.2. Our partition based MIS algorithm requires that vertices v are give an immutable data member $v.topo$; in the MIS algorithm, processor p can select a vertex v only if $\{\forall v1 \in v.adjac \mid v1.state \neq deleted\}$:

$$v1.topo < v.topo \ \ or \ \ (v1.topo = v.topo \ \ and \ \ v1.proc < v.proc)$$

This replaces the test expression in line 6 of Figure 5.3.

5.3.2 Topological classification of vertices in finite element meshes

Our methods are motivated by the intuition that the coarse grids of multigrid methods must represent the boundary of the domain well in order to approximate the function space of the fine mesh well, which is necessary for multigrid methods to be effective

[44, 23]. Intuitively this can be done by emphasizing the vertices that "define" the *domain*. Note, we define a domain in a slightly non-standard way as to mean a contiguous region of the real finite element domain with a particular material property, thus for our discussion the boundary of the PDE proper are augmented with boundaries between different material types. If the domain is convex then a convex hull is useful in reasoning about how to best classify the vertices. Vertices that are on the convex hull should be given more emphasis the those that are not (i.e. interior vertices). Vertices that are required to define the convex hull are likewise more important than vertices that simply lien the convex hull. Domains of interest are by no means necessarily convex, but the idea of emphasizing vertices by their contribution to defining the boundary of a domain is useful in the exposition of our methods.

The first type of classification of vertices is to find the *exterior* vertices - if continuum elements are used then this classification is trivial. For non-continuum elements like plates, shells and beams, heuristics such as minimum degree could be used to find an approximation to the "exterior" vertices, or a combination of mesh partitioners and convex hull algorithms could be used. For the rest of this section we assume that continuum elements are used and so a boundary of the domains, represented by a list of *facets* or 2D polygons, can be defined. The exterior vertices give us our first vertex classification from the last section: *interior* vertices are vertices that are not exterior vertices. Exterior vertices require further classification, but first we need a method to automatically identify *faces* in our finite element problems.

5.3.3 A simple face identification algorithm

To describe our algorithm we assume that a list of facets $facet\_list$ has been created of the boundaries of the finite element mesh. Assume that each facet $f \in facet\_list$ has calculate its unit normal vector $f.norm$. Assume that each facet f has a list of facets $f.adjac$ that are adjacent to it. With these data structures, and a list with with $AddTail$ and $RemoveHead$ functions with the obvious meaning, we can calculate a $face\_ID$ for each facet with the algorithm shown in Figure 5.11.

This algorithm simply repeats a breadth first search, of trees rooted at an arbitrary undone facet, which is pruned by the requirement that a minimum angle (arccos TOL) be maintained by all facets in the tree relative to the root. This heuristic is a simple way to

forall $(f \in facet\_list)$ $f.face\_ID \leftarrow 0$

$Current\_ID \leftarrow 0$

forall $f \in facet\_list$

  if $f.face\_ID = 0$

    $list \leftarrow \{f\}$

    $norm \leftarrow f.norm$

    $Current\_ID \leftarrow Current\_ID + 1$

    while $list \neq \emptyset$

      $f \leftarrow list.RemoveHead$

      $f.face\_ID \leftarrow Current\_ID$

      forall $f1 \in f.adjac$ TOL, $-1 < TOL \leq 1$, is a user selected tolerance

        if $norm^T \cdot f1.norm > TOL$ and $f1.face\_ID = 0$

          $list.AddTail(f1)$

Figure 5.11: Face identification algorithm

identify *faces* (or manifolds that are somewhat "flat") of the boundaries in the mesh.

These faces are useful for two reasons:

- Topological categories for vertices, used in the heuristics of §5.3.3, can be inferred from these faces:
  - A vertex attached to only one face is in the interior of a *surface*.
  - A vertex attached to two different faces is in the interior of an *edge*.
  - A vertex attached to more than two different faces is a *corner*.
- Vertices not associated with the same faces should not interact with each other in the MIS algorithm.

This second criterion is discussed in the next section.

5.3.4 Modified maximal independent set algorithm

We now have all of the pieces that we need to describe the core of our method. First we classify vertices and ensure that a vertex of *lower* rank does not suppress a vertex of *higher* rank - this was done with a slight modifications to standard MIS algorithms in

§5.3.1. Second we want to maintain the integrity of the “faces” in the original problem as best we can. The motivation for this second criterion can be seen in Figure 5.12.

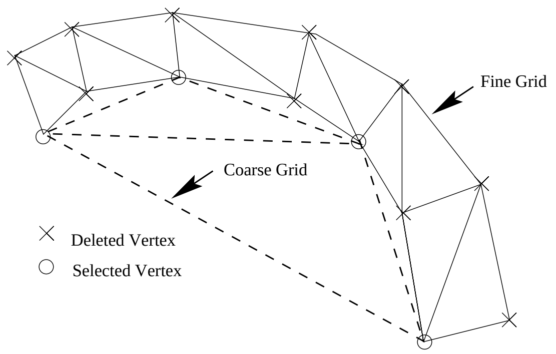


Figure 5.12: Poor MIS for multigrid of a “shell”

If the finite element mesh has a thin region then the MIS as described in §5.3.1 can easily fail to maintain a cover of the vertices in the fine mesh. This comes from the ability of the vertices on one face to decimate the vertices on an opposing face as shown in Figure 5.12. This phenomenon could be mitigated by randomizing the order that the vertices are added to the MIS, at least within a vertex type. But randomization is not good enough as these skinny regions tend to lower the convergence rate of iterative solvers, so we need to do something better.

A simple fix for this problem is to modify the graph to which we apply the MIS algorithm - we want to maintain the same vertices in the graph, but will reduce the edge set. To avoid the problems illustrated in Figure 5.12, we can look at our method of classifying vertices again:

- A vertex attached to only one face is in the interior of a *surface*.
- A vertex attached to two different faces is in the interior of an *edge*.
- A vertex attached to more than two different faces is a *corner*.

Now we claim that by *removing all edges between vertices that are not attached to a common face*, we force the MIS to be a more “logical” and economical (in terms of solver performance) representative of the fine mesh, as shown in Figure 5.13. For instance we do not want corners to delete a edge vertex with which it does not share an exterior facet. Another example is that we do not want edge vertices to delete edges with which it does not share a face (this is the most effective heuristic in this example). And finally we augment our heuristic and not allow corners to be deleted at all - this could be problematic on some meshes that have many initial "corners", as defined by our algorithm, and a reclassification of the remeshed vertices on coarse meshes is advisable.

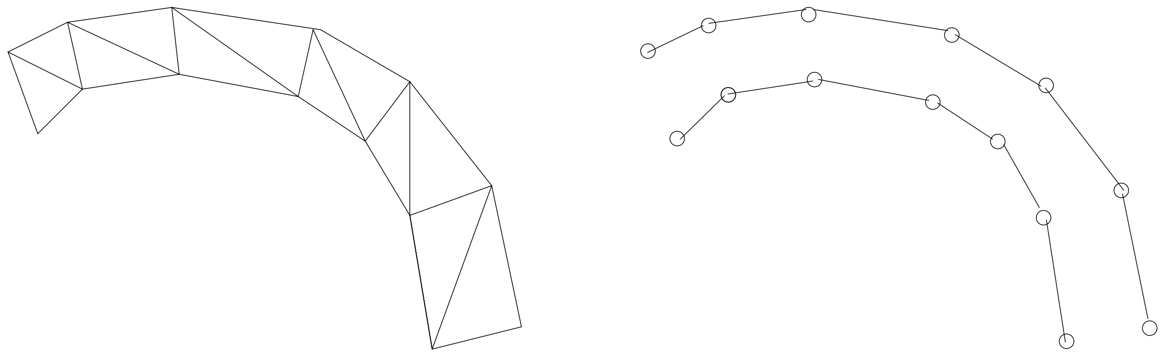


Figure 5.13: Original and fully modified graph

We are now free to run our MIS algorithm on this modified graph, Figure 5.14 shows and example of a possible MIS and remeshing.

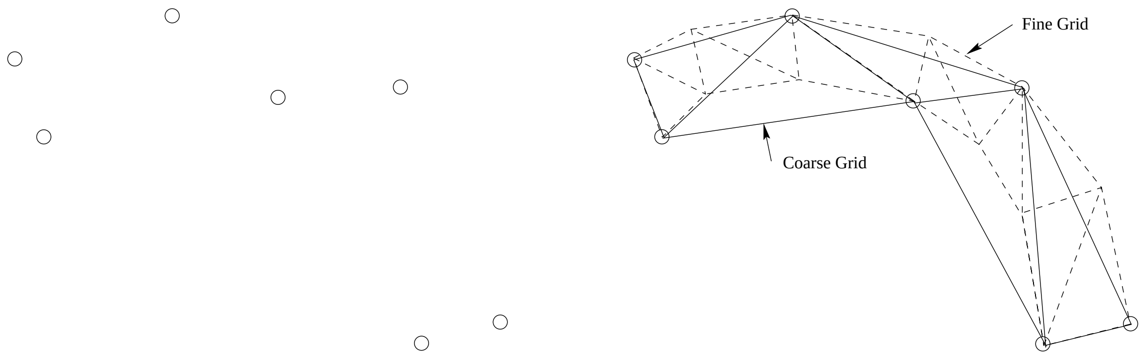


Figure 5.14: MIS and coarse mesh

5.3.5 Vertex ordering in MIS algorithm on modified finite element graphs

An additional degree of freedom, in this algorithm as described thus far, is the order of the vertices within each category. Thus far we have implicitly ordered the vertices by topological category; the ordering within each category can also be specified. Two simple heuristics can be used to order the vertices: random order, or a "natural" order. Meshes may be initially ordered in either a block regular order (i.e. an assemblage of logically regular blocks), but this depends on the mesh generation method used. Initial vertex orders can also be ordered in a cache optimizing order [82] like Cuthill-McKee. Both of these ordering types are what we call *natural* orders, and we assume that the "initial" order of our mesh is of this type (if not then we can make it so). The MISs produced from natural orderings tend to be rather dense, random ordering on the other hand tend to be more sparse. That is the MISs with natural orderings tend to be larger than those produced with random orders. Note that for a uniform 3D hexahedral mesh, the asymptotics of the size of the MIS is bounded from above by $1/2^3$, and from below by $1/3^3$ as the largest MIS will pick every second vertex and the smallest MIS will select every third vertex; natural and random orderings are simple heuristics to approach these bounds.

Small MISs are preferable as this means that there is less work to be done on the coarser mesh, also fewer levels are required before the coarsest grid is small enough to solve directly, but care has to taken to not degrade the convergence rate of the solver by compromising the quality if the coarse grid representation. In particular, as the boundaries are important to the coarse grid representation it may be advisable to use natural ordering for the exterior vertices and a random ordering for the interior vertices - we use this approach in many of our numerical experiments.

Meshing of the vertex set on the coarse grid

The vertex set for the coarse grid remains to be meshed - this is necessary in order to apply finite element shape functions to calculate the restriction operator. We use a standard Delaunay meshing algorithm to give us these meshes. This is done by putting the mesh inside of a bounding box, thus adding dummy vertices to the coarse grid set, and then meshing this to produce a mesh that covers all fine grid vertices. The tetrahedron attached to the bounding box vertices are removed and the fine grid vertices within these deleted tetrahedron are added to a list of "lost" vertices (*lost_list*).

With this body we continue to remove tetrahedra from the mesh that connect vertices that were not "near" each other on the fine mesh (recall the vertex set are still nested), and that do not have any vertices that lie "uniquely" within the tetrahedron. Define a vertex v to lie uniquely in a tetrahedron if v lies completely within the tetrahedra and not on its surface, or there is no adjacent tetrahedra to which v can be added. More precisely if a vertex's shape function values are all larger than some small tolerance ϵ (we use only linear shape functions), or there is not an adjacent tetrahedra that can "accept" the vertex, then that tetrahedra is deemed necessary and not removed. We also use a more aggressive phase in which we use a more negative, though still small, tolerance, to try to remove more tetrahedra - but the "orphaned" vertices are added to the *lost_list*. The resolution of the vertices in the *lost_list* is discussed in §5.3.6.

5.3.6 Coarse grid cover of fine grid

The final optimization that we would like to employ is to improve the cover of the coarse mesh on the fine vertex set. With these coarse grids constructed the interpolation operators are calculated by evaluating standard finite element element shape functions of the element to which the fine grid vertex is *associated*. Each fine grid vertex is associated with an element on the coarse grid - the element that covers the fine grid vertex. In general however some fine grid vertices (the *lost_list* from the previous section) will fail to be covered by the coarse grid as shown in Figure 5.14. This problem can be solved in one of two ways: find a nearby element and use it (thus extrapolate), or move the vertices on the coarse grid so as to cover all fine grid vertices. We can use the extrapolation of an element that does not cover a fine mesh point, the extrapolation values will simply not all be between zero and one. Intuition tells us however that interpolation is better than extrapolation. Alternatively one can move the coarse grid vertex positions to cover the fine vertices in *lost_list*.

The optimal coarse grid vertex positions (or an approximation to them) could perhaps be constructed with the use of interpolation theory to provide *cost functions*, and linear or nonlinear programming. We have instead opted for a simple, greedy algorithm that iteratively traverses the exterior vertices of the coarse mesh and applies a simple algorithm to *try* to cover the uncovered vertices that are near it. First we define a list ext_neigh_c, for each coarse grid vertex c, that contains all of the vertices attached to all of the exterior facets to which c is attached. We define a list $lost\_list_c$ for each coarse grid vertex c, and put the vertices v in *lost_list* into the list of the coarse grid vertex to which v is closest; $lost\_list_c$ is then expanded to include the vertices in $lost\_list_i$ for each vertex $i \in ext\_neigh_c$. Given a maximum number of outer iterations M, a maximum distance tol_{max} that a vertex can be move, and a larger tolerance tol_{delete} to prune the list of potential coarse grid vertices that can help to cover a fine grid vertex, the algorithm is as follows

- **do** M **times**: **forall** c on the exterior of the coarse grid
  - Calculate a vector ϕ: the weighted average of the outward normals of the facets connected to the coarse grid vertex ($c.facet\_list$), weighted by the facet area.
  - The normal of each facet that does not have a positive inner product (with this ϕ vector) is added to ϕ until $\{f \in c.facet\_list \mid f.norm^T \cdot \phi < 0\} = \emptyset$, then ϕ is normalized to unit length.
  - $\omega \leftarrow \infty$
  - **forall** $v \in lost\_list_c$: **forall** $f = (a, b, c) \in c.facet\_list$
    * Solve for α in

$$\begin{vmatrix} a.x & a.y & a.z & 1 \\ b.x & b.y & b.z & 1 \\ c.x + \alpha \cdot \phi.x & c.y + \alpha \cdot \phi.y & c.z + \alpha \cdot \phi.z & 1 \\ v.x & v.y & v.z & 1 \end{vmatrix} = 0.0$$

    * if $\alpha > tol_{delete}$ remove v from $lost\_list_c$
    * else if $\alpha > tol_{max}$: $\omega \leftarrow tol_{max}$
    * else if $\alpha > 0$ and $\alpha < \omega$: $\omega \leftarrow \alpha$
  - if $\omega < \infty$
    * $c.cood \leftarrow c.cood + \omega \cdot \phi$
    * Recalculate the shape functions for all of fine grid vertices that are within an element connected to c, $c.elems$.
    * For all $e \in c.elems$, For all $v \in lost\_list_c$: calculate the shape function for v in element e
      · If all shape values are greater than $-\epsilon$ for some small number ϵ, then
      · Add v to e and remove v from $lost\_list_i$ for all $i \in ext\_neigh_c$

Figure 5.15 shows an illustration of what our algorithm might do on our running example.

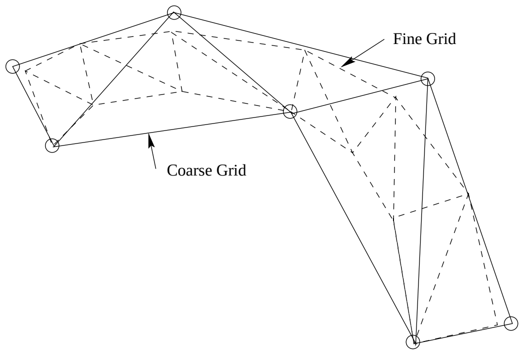


Figure 5.15: Modified coarse grid

Figure 5.16 shows an example of our methods applied to a problem in 3D linear elasticity. The fine (input) mesh is shown with three coarse grids used in the solution.

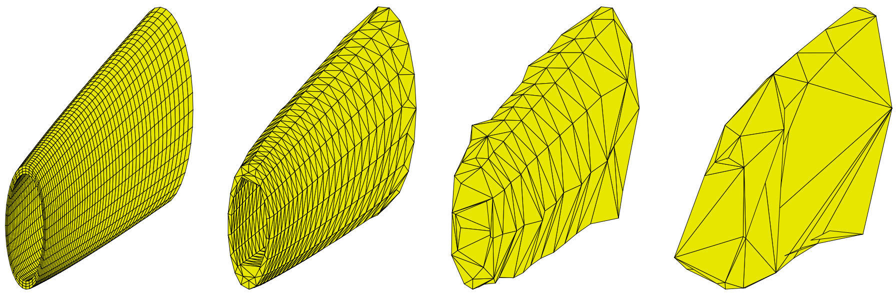


Figure 5.16: Fine (input) grid and coarse grids for problem in 3D elasticity

5.3.7 Numerical results

To demonstrate the effectiveness of the methods that we have discussed we use three test problems in linear elasticity, shown in Figure 5.17. The problems are chosen to exercise the primary problem features that we have tried to accommodate: material coefficient discontinuities, thin "shell" types of features, and curved surfaces. The first problem is of a hard sphere (Youngs modulus E = 1, Poisson ratio $\nu = 0.30$) encased in a soft rubber-like material (E = 10^{-4}, $\nu = 0.49$). The second problems is a steel beam-column connection made of thin, poorly proportioned, elements. The third problem is of a tube fixed at one end and loaded at the other end like a cantilever - a thin slit of the rubber-like material of the first problem runs down the length of one side of the tube. The "sphere" problem has 39,732 equations, the "beam" has 34,460 equations, and the "tube" has 57,600 equations. All problems use eight vertex trilinear "brick" elements; the hard material is a standard displacement element [86] and the soft material is a mixed formulation [76].

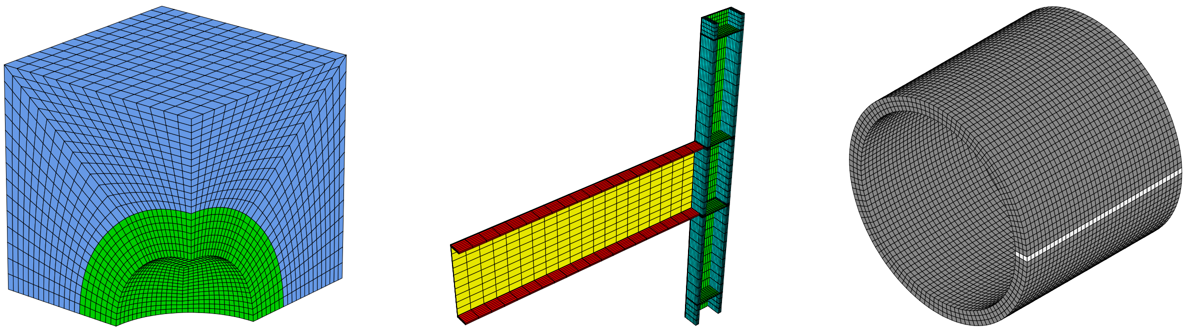


Figure 5.17: Test problems from linear elasticity: Sphere (39,732 dof), beam-column (34,460 dof), tube (57,600 dof)

Each problem is solved with a conjugate gradient (CG) solver, to a relative tolerance of 10^{-6}. Full multigrid is used for the preconditioner, the smoother is CG preconditioned by block Jacobi, the number of blocks was reduced by a factor of eight for each successive level. All problems used three coarse grids, so that the top grid (solved directly) was a few hundred equations in size. We present numerical experiments on an IBM SP (120 MHz Power2 - P2SC). The Finite Element Analysis Program (FEAP)[36], is used to generate out test problems and produce our graphics. We use ParMetis [53] to calculate our partitions, and PETSC [10] for our parallel programming development environment. Our code is implemented in C++, FEAP is implemented in FORTRAN, and PETSc and

ParMetis are implemented in C.

To demonstrate the effectiveness of our methods, we run these test problems with: Pure MIS (no optimizations); Pure MIS with the heuristics of exterior vertices ordered first, and interior ordered last and randomly; our modified MIS without the vertex cover heuristics; and finally all of our optimizations. The solution times and the iteration counts are shown in Figure 5.2.

To demonstrate the effectiveness of our methods, we run these test problems with:

- Pure randomized greedy MIS (no optimizations) [2].
- Pure MIS with the heuristic of exterior vertices ordered first, and interior ordered last and randomly §5.3.5.
- Our modified MIS §5.3.4, *without* the vertex cover heuristics §5.3.6
- All of our optimizations §5.3.5, §5.3.4, and §5.3.6.

The solution times and the iteration counts are shown in Table 5.2.

Problem Names	Sphere	Beam-column	Tube
Number of equations	39,732	34,460	57,600
Condition $K(A)$ of matrix	$7.2 \cdot 10^6$	$1.0 \cdot 10^8$	$1.8 \cdot 10^5$
Number of pre (and post) smoother applications	2	3	3
Number of blocks in Jacobi smoother (fine grid)	240	32	256
Pure MIS (no optimizations)	116(58)	201(50)	296(69)
Pure MIS w/ exterior ordered 1^{st}, interior last and random	75.1(36)	225(54)	286(70)
Modified MIS w/o vertex cover heuristics	56.5(27)	91.0(25)	104(14)
Modified MIS w vertex cover heuristics (all optimizations)	56.5(27)	91.0(25)	52.8(9)
Matrix vector product time on the fine grid (sec)	0.0727	0.057	0.0907

Table 5.2: Solve time in seconds (number of iterations)

These experiments show that our methods provide significant improvement over a random MIS, especially on complex domains. Moving coarse vertices to cover fine ones, did not help on meshes that do not have curved surfaces, as is expected, and provide some improvement to meshes with curved surfaces. The vertex orderings, within each category when applicable, affects these results a small amount, particularly on the "Pure MIS" data; we see about a 10-15% variance with different randomization schemes. Thus one should consider that the standard deviation rather high for this data - but we have consistently observed the dramatic benefit of the modified MIS, that is reflected in this data. Thus we feel that our modified MIS heuristics are effective - especially on domains with complex geometry.

5.4 Mesh generation

Mesh generators have been developed using three basic methods: Advancing front [21, 9, 59, 45], Delaunay [11], and Octree [20]. We use a simple Delaunay method with Watson's insertion algorithm [38]. This method proceeds by trivially meshing a "dummy" envelope of vertices then inserting each vertex into the existing mesh. The algorithm invariant is that the mesh remain a valid Delaunay mesh after each vertex insertion. The method is not highly optimal nor parallelizable but it is simple to implement, moreover we do not *need* a parallel Delaunay, the the design of which is an open problem [12].

Additionally Delaunay is only well defined on vertex sets with no nontrivial coplanar or cospherical point sets - most algorithms work on these non-*general position* point sets although exact arithmetic is required for robustness. Delaunay methods in 3D require the evaluation of a 5 by 5 determinant - exactly - although only the sign (negative, zero, or positive) is required.

We use a very efficient method of adaptive precision arithmetic that is specialized for the numerical predicates used in Delaunay tessellations [73]. This method achieves its efficiency by first unrolling the loops in the determinant evaluation. After the result has been calculated and intermediate results have been judiciously saved, a backward error analysis is performed and if the results of this analysis suggest that the *sign* of the answer can not be trusted then this processes is repeated back through the intermediate levels until the values of the intermediate results can be trusted and exact arithmetic is used for the rest of the calculation. Note, if a set of input points is in *general positions* (and plain double precision arithmetic is robust) then the only additional cost in this method is the one backward error analysis (this is about a 10% time penalty). If the points are cospherical then the determinant is zero and the calculation must be done completely in exact arithmetic, otherwise this method allows for as much of the earlier terms to be reused.

5.5 Finite element shape functions

After the coarse meshes are created one can simply use standard finite element shape functions for the restriction and interpolation operators. Linear (and bilinear or trilinear) finite element shape functions are affine mappings of a point in space with the vertices of a polytope or element. That is, given a point in space we would like to express its coordinate as a weighted sum of the coordinates of the element vertices. This in effect requires that the element vertices be used to construct a *shape function* which has the very intuitive meaning of the *assumed* shape of the element, given the displacements of its vertices. These shape functions are in general a partition of unity so that any constant function can be represented exactly. Note, displacements are the vertex variables for displacement based finite element methods, however the vertex variable may, in general, be any physical quantity - e.g. coordinates, velocities, temperature, pressure, flux, or electro magnetic field.

One needs to find the element that contains each fine grid vertex (j). The finite element shape function (for the element) is evaluated, at the location of the fine grid vertex, for each vertex (I) of the coarse grid element containing j to compute the R_{Ij} entry in the restriction operator. Note that in general one needs to search the coarse grid elements to find the place for each fine grid vertex, this can be done in $O(\log(n))$ time with an optimal searching algorithm, but since our coarse vertices are created from the fine ones it is possible for each vertex to find its "parent" in constant time.

5.6 Galerkin construction of coarse grid operators

The Galerkin construction of the coarse grids requires that three sparse matrices be multiplied together. In indicial notation (i.e., with the sum_i symbol omitted) we have

$$A_{IJ}^{coarse} \leftarrow R_{Ii} A_{ij}^{fine} R_{Jj}$$

This construction takes about 5 or 6 times as many floating point operations (flops), for hexahedral meshes, as a matrix vector product on the finer grid, although achieving optimal performance is difficult as there are no sparse libraries that currently support a sparse matrix triple product. We have implemented a distributed memory sparse matrix triple product that operates on PETSc [10] matrices. We are only able to get about 18-41% of the floating point performance that we get in the multigrid iterations on one processor, but this matrix triple product a small though not insignificant part of the solve, and it scales reasonably well. The reasons for this lack of performance is three fold:

- We have four sparse objects in the matrix triple product and only one in the matrix vector product - this requires much more indirect addressing.
- Communication in the matrix triple product requires that a *matrix* be communicated instead of a vector (in the matrix vector product) - matrices are much "heavier" objects than vectors (i.e. they are 2D sparse arrays, whereas vectors are 1D dense arrays)
- We implemented this, and as the performance of the matrix triple product is not the object of this research, and it is not a bottleneck in the code, we have devoted a limited (although not insignificant) amount of development effort to the task.

That said, we can briefly discuss our algorithm and our reasons for the decisions that we made. We first assume that the matrices are ordered in block rows (with compressed sparse block row storage format), that is each processor "own" a contiguous logical block of the matrix - this is standard practice and is the format used by PETSc.

For example a matrix A (for an almost regular block of hexahedral elements) with 14,880 dof that has 1,064,988 non-zeros, its restriction matrix R has 1,938 rows (and 14,880 columns) with 47,633 non-zeros and the coarse matrix is 1,938 by 1,938 and has 211,982 non-zeros. Thus the restriction operator has about $\frac{1}{20}$ as many non zeros as the larger stiffness matrix, and about $\frac{1}{4}$ as many non-zeros as the smaller stiffness matrix. From this example it is clear that the fine grid stiffness matrix is much larger than all of the other matrices combined - thus we wish to begin with the fine grid matrix in optimizing for cache performance. This implies the first two loops of our iteration, in Figure 5.18, runs sequentially through the fine grid matrix. This decision now dictates that we access matrix entries in R that are stores on another processor, and also that we accumulate off-processor data to the product matrix A^{coarse}. Therefore some parts of R are replicated and communication is required to accumulate the results - but this is more attractive than communicating and duplicating parts of the fine grid matrix.

Figure 5.18 does not show many optimizations that are done in practice but is intended to show the details of the computation and the structure of these matrices. Note

$--$ $pRowStart$ and $pRowEnd$ are the first and last rows on processor p

for $i = pRowStart : pRowEnd$

  forall $j \in i.columns$ $--$ $i.columns$ is the set of adjacencies for vertex i

    for $II = 1 : 4$ for each vertex in $.element$

      $I = i.element.vertex[II].index$

        $--$ $shape(c)$ is the scalar value, at coordinate c, of the shape function

      $shape\_I = i.element.vertex[II].shape(i.coord)$

      for $JJ = 1 : 4$

        $J = j.element.vertex[JJ].index$

        $shape\_J = j.element.vertex[JJ].shape(j.coord)$

        $--$ Note that $A_{IJ} \in \Re^{ndf \times ndf}$

        $A_{IJ}^{coarse} \leftarrow A_{IJ}^{coarse} + shape\_I \cdot A_{ij} \cdot shape\_J$

Figure 5.18: Matrix triple product algorithm running on processor p

that this matrix triple product is, in general, only done once for each fine grid matrix and so there is no opportunity to amortize the communication costs of the fine grid matrix in order to save costs in the communication of the resultant coarse grid matrix.

5.7 Smoothers

Smoothers are "simple" solvers in and of themselves, §2.3.1, §2.3.2, and §2.4. Matrix splitting methods such as Jacobi, Gauss-Seidel and SOR are used frequently although they are not effective enough for illconditioned systems. Jacobi and its generalization *block Jacobi* however is very useful as a *smoother* in our solver as we shall see. Another popular preconditioner is a so-called *incomplete factorization* which calculates a factorization of A that limits the amount of *fill* that occurs [80]. This fill is responsible for the non-$O(n)$ complexity in factoring and solving finite element matrices.

We use Krylov subspace (§2.3.2) smoothers (§3.3) preconditioned with one level domain decomposition methods (§2.4). We have found that the best smoother preconditioner, available in PETSc are diagonal, block Jacobi, and overlapping Schwarz - with block Jacobi being our primary method for harder problems

As we are using (or at least assuming) unstructured grids, the construction of the subdomains defining the block of the block Jacobi smoother is by no means evident. Fortunately a good solution exists - it is natural to think that we would want "well" shaped subdomains in a domain decomposition smoother, so as to capture the lowest energy functions as is possible with a fixed number of vertices per block - thus improving the constant of "assumption 2" in §3.4.3. We can use our standard mesh partitioners to give us good partitions - thus we construct our subdomains with METIS [52].

Chapter 6

Multigrid characteristics on linear problems in solid mechanics

This chapter presents numerical studies that are aimed at providing insight into some of the fundamental challenges in applying iterative methods to the solution of complex problems in solid mechanics and some characteristics of iterative solvers in general, and multigrid in particular. Recall that the motivation for using iterative methods is to solve large scale problems; we address issues of scale in the following chapters, but first we study the convergence behavior of our solver on some basic classes of problems in linear elasticity.

6.1 Introduction

This section addresses the issues of incompressibility (§6.5), poor aspect ratio elements (§6.6), poor "geometric" conditioning (§6.3), and large jumps in material coefficients (§6.3,§6.4), via a suite of numerical experiments. But first we shall verify that multigrid works in §6.2 - that is, for simple problem in linear elasticity the convergence rate is invariant to the scale (or "fineness") of discretization. We conclude that multigrid is a very promising solver of the challenging finite element problems that are found in many areas of science and engineering.

6.2 Multigrid works

Our model problem for this section is a long thin cantilever beam (i.e. all displacements fully restrained at one end of a long skinny rectangular prism). We use a regular mesh, with perfect cubic elements, for a $1 \times 1 \times 32$ cantilever discretized with N elements through the thickness. A load applied at the end and and use a linear elastic displacement based element with Poisson ratio of 0.3, shown in Figure 6.1. The load is "off axis", that is the load vectors (all parallel) have significant components in all three directions (-1.0 in all directions, with the axis at the support and the beam being in the positive 1 direction) - this is significant as we are using Krylov subspace methods and the convergence rate can depend on the nature of the applied load. For example if the applied load were an eigenvector of the stiffness matrix then *any* (unpreconditioned) Krylov subspace method would converge in one iteration.

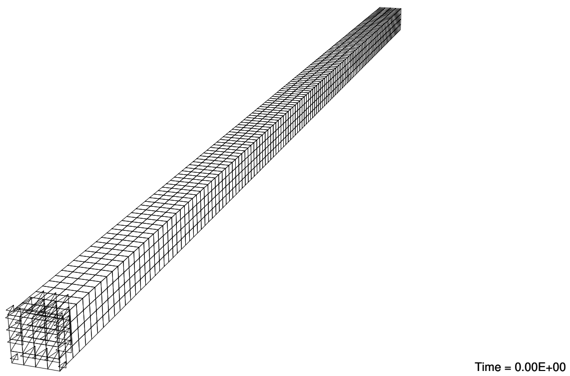


Figure 6.1: Cantilever with uniform mesh and end load, $4 \times 4 \times 128$ element mesh, $N = 4$

This problem is meant to demonstrate that multigrid, applied to a model problem, does indeed converge at a rate independent of the scale of discretization. Actually we show some *superlinear* convergence; this is a common phenomenon, i.e. better meshes lead to faster convergence with multigrid. Table 6.1 shows the number of iterations, and the condition number of each matrix. The condition number is estimated from below by running CG *without* preconditioning and calculating the extreme eigenvalues of the *projected* tridiagonal matrix that CG uses to calculate its (approximate) answer (§3.4.1). This unpreconditioned CG solve was run with a relative residual tolerance of 10^{-6}. The drastic increase in the number of iterations shows that these matrices are indeed getting harder for CG to solve.

Our solver is CG with a full multigrid preconditioner, using a diagonally preconditioned CG smoother. We use two iterations in the pre and post smoothers. We declare convergence when norm of the initial residual has been reduced by a factor of 10^{-6}.

N	Levels	Iterations	dof	Condition	Unpreconditioned iterations
2	2	14	1,728	$2.9 \cdot 10^7$	478
4	3	12	9,600	$1.2 \cdot 10^8$	1052
8	4	10	62,208	$4.3 \cdot 10^8$	2200

Table 6.1: Multiple discretizations of a cantilever

Figure 6.2 shows the convergence history of these problems.

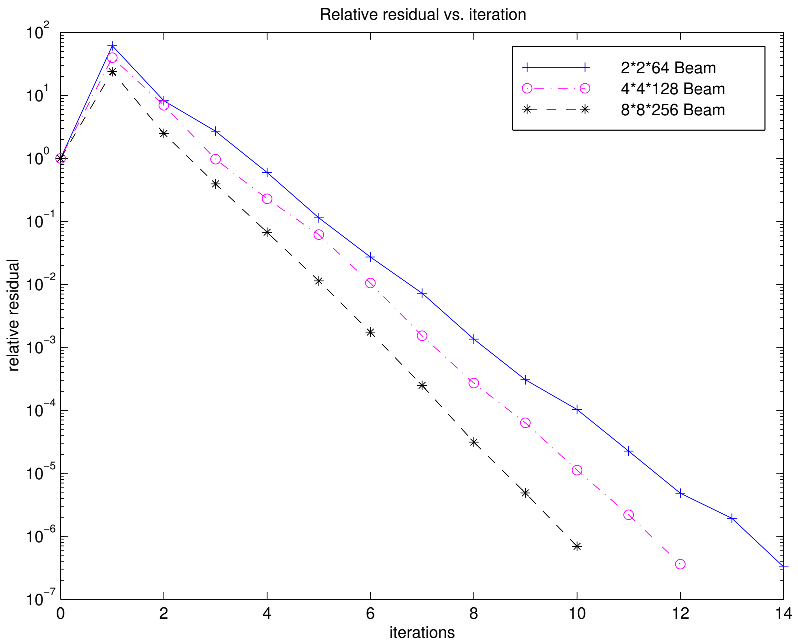


Figure 6.2: Residual convergence for multiple discretizations of a cantilever

Note another phenomenon that is common of iterative solvers, the residual increases (jumps) during the first few iterations. This jump can sometimes be down; a general rule of thumb is that the more poorly conditioned the problem the larger this initial jump is.

This section shows that multigrid has a hope of being an effective solver for finite element method matrices. We can see that on a trivial problem, multigrid's rate of convergence is indeed nearly independent of the condition number of the matrix as reflected in finer discretizations - indeed, it improves with size. The next section also uses this model problem and shows that multigrid is invariant to changes in material coefficients - *if* the material interfaces are captured on the coarse grids.

6.3 Large jumps in material coefficients - soft section cantilever beam

In this section we take the cantilever beam of the last section and add two rows of a soft material in the middle (thus the problem is singular if the soft material has no stiffness). The $N = 8$ (62,208 dof) problem is used from the last section - the "soft" material has a Poisson ratio of 0.3 and we parameterize its elastic modulus to range from 1.0 (that of the rest of the mesh) down to the point that the matrix is singular to working precision (and CG breaks down). We have encouraged a very regular structure of the coarse meshes by using the *natural* vertex order in the maximal independent set algorithm - additionally we have placed this "soft" layer judiciously so as to produce the "best" coarse grids (without changing the mesh's geometric configuration). Thus this is very much an artificial example and it is simply meant to show that indeed "multigrid *can* work perfectly" for 3D elasticity, with large jumps in material coefficients, if the problem is perfectly partitioned Table 6.2 shows the results in terms of iteration count for these problems. Figure 6.3 shows the

$\log_{10} E_{soft}$	0	-2	-4	-6	-8
Iterations	11	11	12	13	14
Condition	$4.3 \cdot 10^8$	$5.6 \cdot 10^8$	$1.5 \cdot 10^{10}$	-	-
Unpreconditioned iterations	2193	2717	6738	-	-

Table 6.2: Cantilever with soft section

convergence history of these problems.

The important point to notice here is that the *slope* of the convergence is the same for all of the problems. Also notice the "dot" plots in Figure 6.3 of the **true** residual (i.e. explicit calculation of $b - Ax_k$) - the last two problems do not converge to the specified tolerance because of the illconditioning caused by these extreme coefficients.

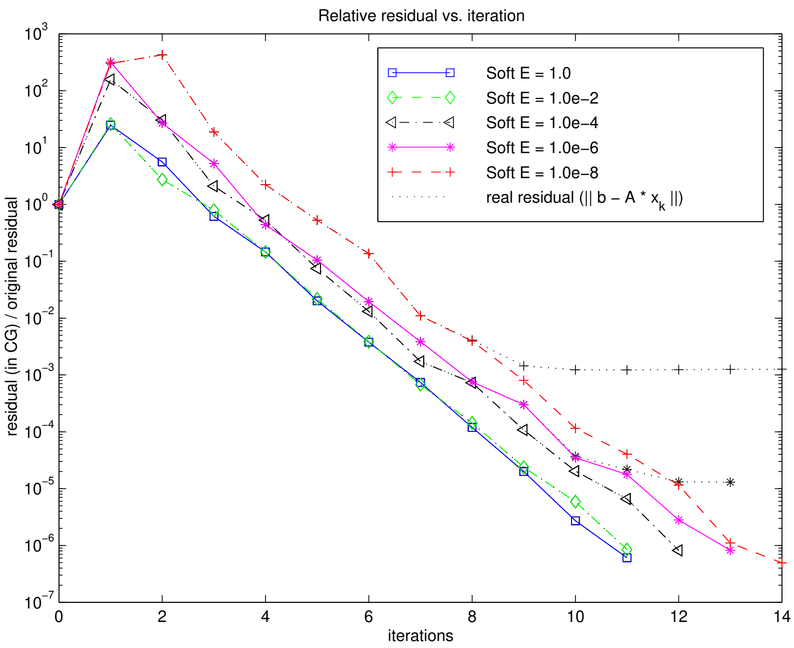


Figure 6.3: Residual convergence for a cantilever with soft a section

6.4 Large jumps in material coefficients - curved material interface

This section demonstrates the effect of *not* capturing the geometry perfectly in a mixed coefficient problem. Figure 5.7 shows a finite element model of a hard sphere included in a "soft" somewhat incompressible material (Poisson ratio of 0.49) [76]. The difficulty with this problem is that the coarse grids *can not* capture the geometry of the fine grids (unless all vertices on the curved surface were retained in the coarse grid). This results in fine grid points on the curved surface being interpolated by points in the interior of the soft material, thus points of very different character (stiffness in this case) are "polluting" each other.

We use block Jacobi preconditioner for the CG smoother as this is most effective for this problem §5.7. Our problem has 24,800 dof and we use 150 blocks in the block Jacobi preconditioner for the conjugate gradient smoother in multigrid.

Table 6.3 shows the convergence (again with a residual tolerance of 10^{-6}). Figure 6.4 shows the convergence history of these problems.

$\log_{10} E_{soft}$	0	-2	-4	-6	-8	-10	-12
Iterations	17	19	24	45	55	55	55
Condition	$6.4 \cdot 10^4$	$3.5 \cdot 10^4$	$3.5 \cdot 10^6$	$3.5 \cdot 10^8$	$3.1 \cdot 10^{10}$	NA	NA
Unpreconditioned iterations	926	930	8607	9000+	9000+	NA	NA

Table 6.3: Iterations for included sphere with soft cover

Thus, unlike the previous example where we were able to capture the material interface perfectly on the coarse grids, the convergence rate does degrade as the elastic modulus of the soft material is reduced.

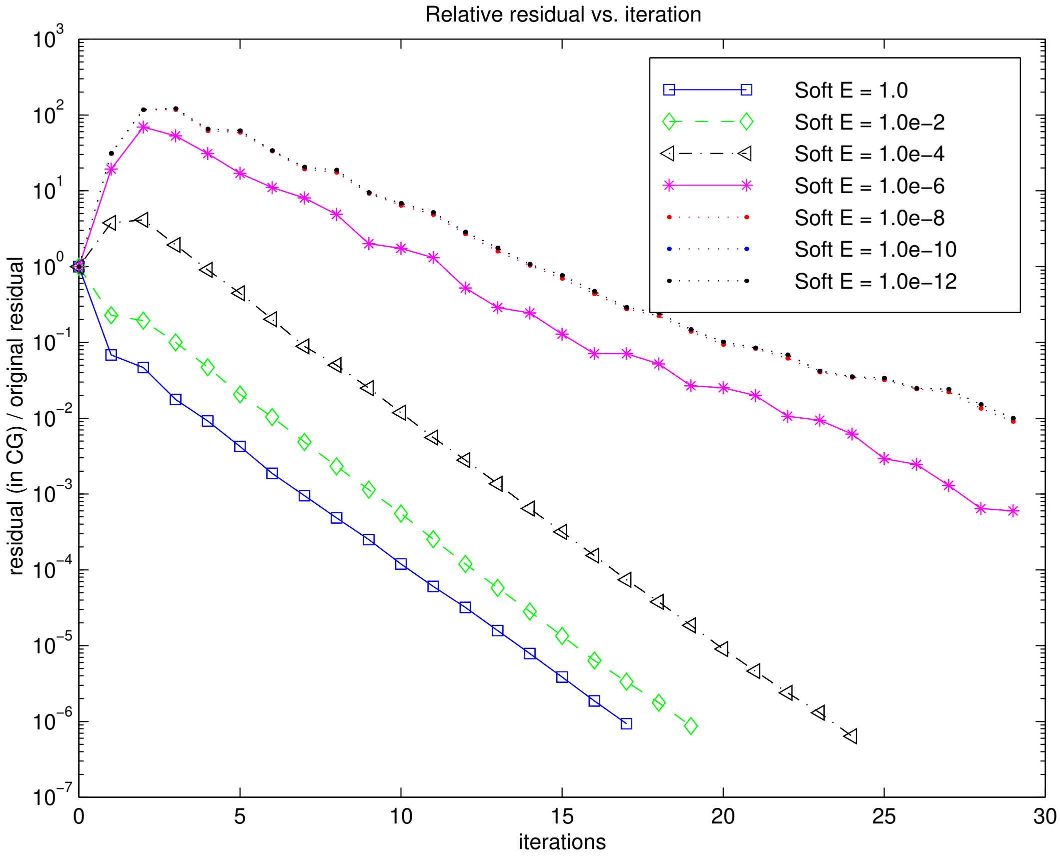


Figure 6.4: Residual convergence for included sphere with soft cover

6.5 Incompressible materials

This section investigates the effect of incompressible materials on the convergence rate of our solver. We use the 14,800 dof version of the same model as the last section. We parameterize the Poisson ratio ν_{soft}. The ratio of the elastic modulus in the soft material, to that of the hard material, is held at 10^{-4}, Three different smoothers are used, conjugate gradients preconditioned with: Jacobi, block Jacobi, and overlapping Schwarz (one layer of overlap). Table 6.4 shows the number of iterations for each case as well as the condition number of a smaller version of this problems (see below for more details). Note, we restart

ν_{soft}	0.3	0.4	0.45	0.49	0.499
Jacobi	17	19	22	74	188
Block Jacobi	11	13	15	33	126
Overlapping additive Schwarz	12	12	16	19	35
Condition ($3,420$ dof version)	$1.52 \cdot 10^6$	$1.64 \cdot 10^6$	$1.70 \cdot 10^6$	$1.75 \cdot 10^6$	$1.76 \cdot 10^6$
Unpreconditioned iterations	5131	5686	6855	8643	9146

Table 6.4: Iterations for included sphere with common preconditioners in CG smoother

CG every 30 iterations for the data in Table 6.4. Figure 6.5 shows the solve time and convergence history of these problems with each smoother.

We can observe many common characteristics of iterative solvers in this data. First, the condition number of the matrix is not effective at predicting the convergence behavior of these problems as the convergence rate deteriorates drastically whereas the condition number is hardly effected by the higher Poisson ratio. Also we can see that the more expensive preconditioners become economical an the harder problems only

The condition number is calculated by calculating the entire spectrum with LAPACK's "dsyev" [3]. Figure 6.6 shows a plot of the spectrum for $\nu_{soft} = 0.3$ to 0.49999, on a $3,420$ dof version of this problem. From this we can see that the Poisson ratio effects only the eigenvalues in the middle of the spectrum (up to the point where the volumetric stiffness of the "soft" material is rivaling and surpasses that of the "stiff" material. Thus for all but the $\nu = 0.49999$ the condition number of these problems are virtually identical.

The second observation to be made from this data is that the more powerful preconditioners are only cost effective on the harder problems with the the overlapping Schwarz preconditioner being the most powerful. Note, we are not able to achieve the same

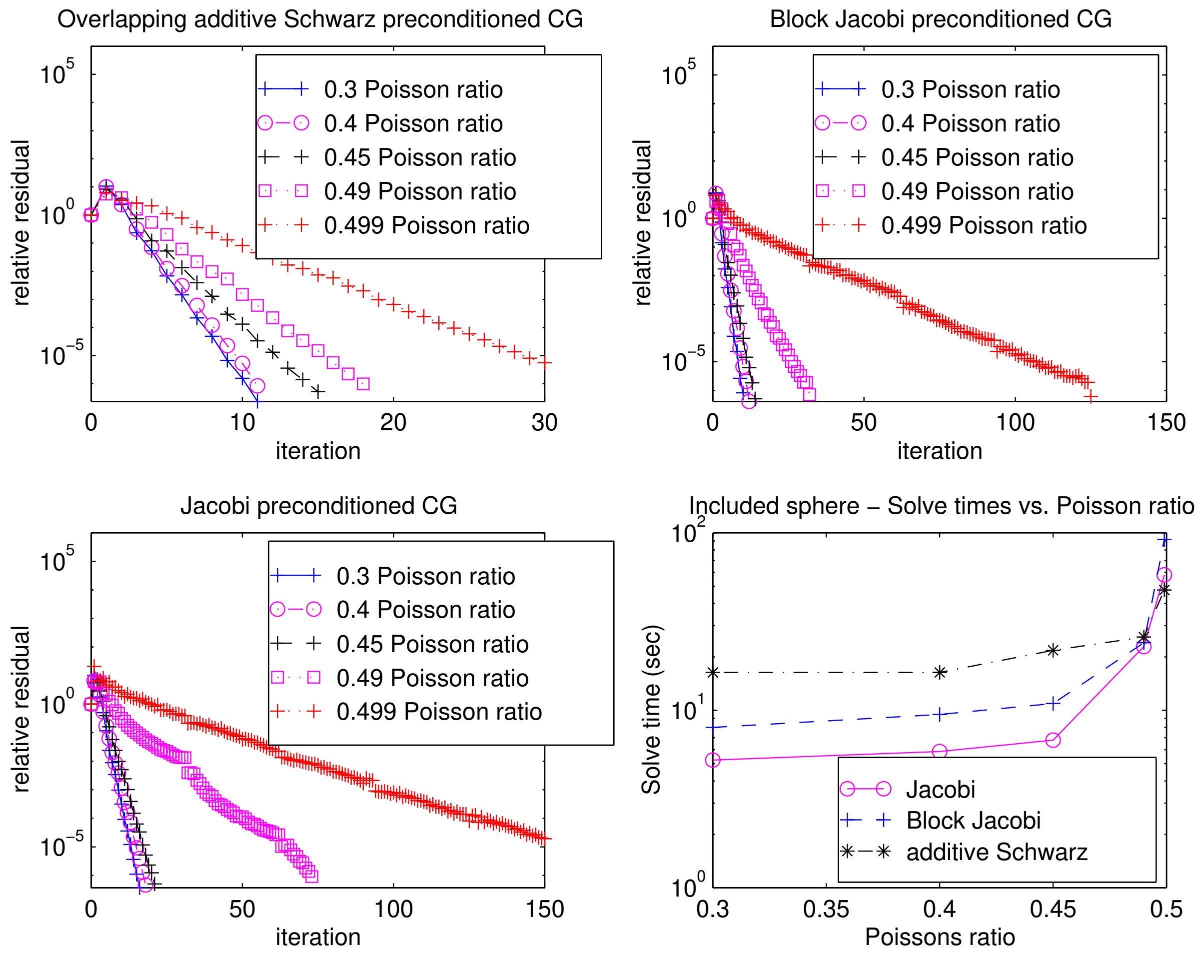


Figure 6.5: Residual convergence for included sphere with incompressible cover material (Note, different scale for overlapping additive Schwarz)

relative performance of overlapping Schwarz in parallel, as it requires communication and as we do not have global coarse grids and hence can not reproduce our serial semantics in parallel. This leads to the conclusion that as the *physics* of our problems get more challenging we need more powerful smoothers, for optimal performance.

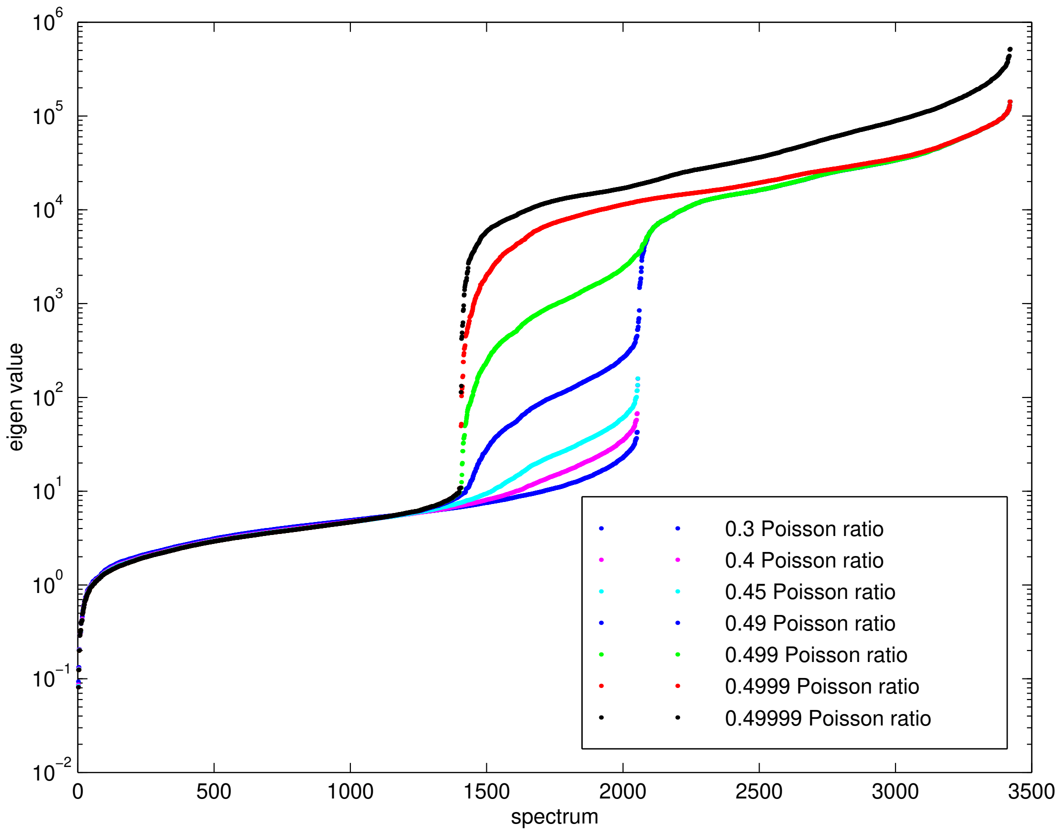


Figure 6.6: Spectrum of 3,800 dof included sphere with incompressible cover material

6.6 Poorly proportioned elements

This section investigates problems with poorly proportioned elements. It is well known that the use of elements with large aspect ratios severely degrades the performance of iterative solvers [80, 83]. The reasons for this are not well understood; though the condition number of the *elements* does increase the overall condition of the matrix does not rise dramatically. For these tests we use a model of a truncated hollow cone, shown in Figure 6.7 with the deformed shape and the first principal stress. These elements have aspect ratios in of about $12 : 5 - 9 : 1$. This problem has $21,600$ dof and a condition number of $3.6 \cdot 10^7$ - the total solve required 27.4 seconds (19 iterations) on one node of a Cray T3E.

A matrix vector product takes about 0.0283 seconds, thus the total solve time (coarse grid matrix vector products and the actual solve) takes the time required to do ≈ 970 matrix vector products. Figure 6.8 shows the convergence history of this problem. These results show that this method has the potential to be effective on thin body elasticity problems with poor aspect ratio elements. Additionally, in our experience, these types of problems benefit greatly from our heuristics described in §5.3.

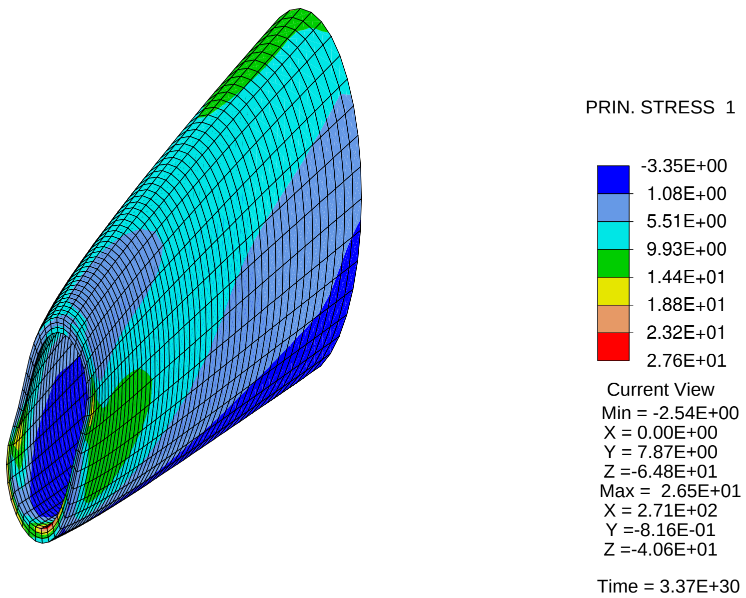


Figure 6.7: Cantilevered hollow cone, first principal stress and deformed shape

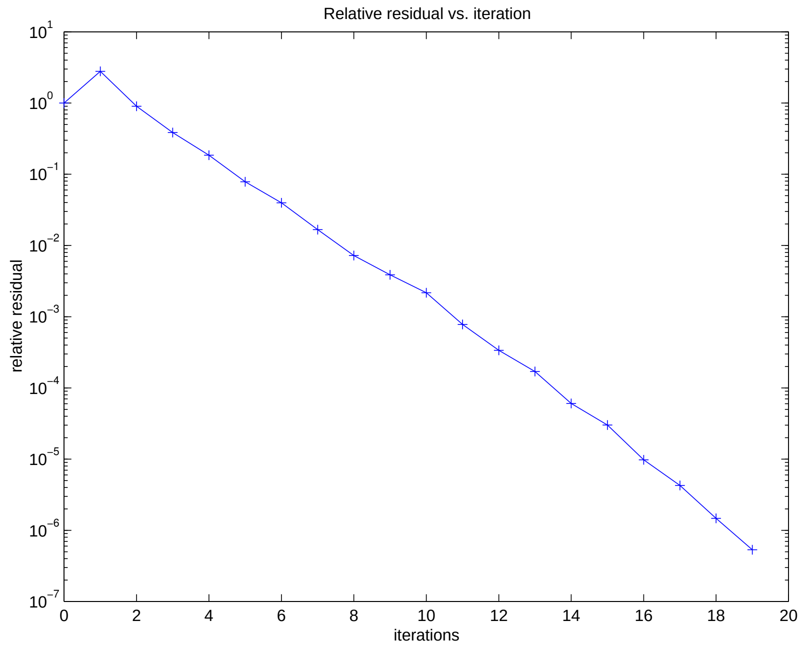


Figure 6.8: Residual convergence for Cone problem

6.7 Conclusion

This section has demonstrated many of the characteristics of multigrid solvers for computational mechanics problems on model problems designed to isolate some of the more common components of problems in solid mechanics. We have shown that multigrid has the potential to be an effective solver for the challenging problems in industry and academia. We have identified some of the characteristics of solid mechanics problems that make them hard for iterative solvers, e.g. incompressibility, thin elements, large jumps in material coefficients, and curved surfaces. The next chapter discusses the performance and parallelism issues of multigrid on large scale finite element problems.

Chapter 7

Parallel architecture and algorithmic issues

This chapter discusses general parallelism issues of finite element codes, architectural details of our parallel finite element implementation, our parallel finite element programming model, and parallel algorithmic issues of large scale parallel finite element implementations.

7.1 Introduction

We proceed by discussing parallelism issues in large finite element codes and "algebraic" multigrid methods in §7.2. §7.3 describes our parallel finite element design and implementation. We discuss our programming model for the complete parallelization of an existing large finite element code with minimal modification to, and no parallel constructs in, the existing finite element code. We conclude with algorithmic issues associated with optimizing performance of multilevel solvers on common parallel computers in §7.4, by developing algorithms for grid agglomeration, useful in mitigating the inherent parallel inefficiencies of the coarsest grids in multigrid solvers.

7.2 Parallel finite element code structure

This section discuses parallelism issues of finite element codes and "algebraic" multigrid methods. This section proceeds as follows: §7.2.1 discusses the effect of the time integrator on the structure of finite element codes; §7.2.2 discuses parallelism in finite element codes; §7.2.3 discuses graph partitioning; §7.2.4 describes parallel computer architectures of today; §7.2.5 discusses the need for multiple levels of partitioning on high performance computers; §7.2.6 touches on overall solver structure and complexity.

7.2.1 Finite element code structure

Here we review, and the next section will extend, the basic steps of the finite element method as listed in §2.2. The first four of these steps do not have a significant impact on the code construction, but the fifth step "Formulate a time integrator for the PDE" does have a major impact on the code construction and problem characteristics.

One of the distinctions among finite element implementations is whether a code is an "implicit code" or an "explicit code". This is a bit surprising as the time integrator may not be the most distinguishing aspect of a finite element package. One would think that a "fluid or solid or multiphysics" distinction would be more important. One might also think that "linear or nonlinear", or "transient or static" would be a more important way to categorize a code - yet the "implicit or explicit" descriptor seems to play a far greater role in the construction of finite element implementations than one would initially expect.

An implicit method must apply the inverse of the linearization (the sparse stiffness matrix) of the residual to the residual - this inverse is a dense operator. An explicit method applies the inverse of the mass matrix to a vector - the mass matrix can be approximated with a "lumped", or "block diagonal," mass matrix effectively, and thus its inverse is *not* dense. Herein lies the source of the difference in the structure of implicit and explicit codes.

The source of the effect of the time integrator on finite element codes can to some degree be attributed to the fact that direct solvers have dominated much of the finite element community in the past. Direct solvers for the stiffness matrix have time complexity $O(n^2)$ for 3D finite element problems with n degrees of freedom (dof), whereas the application of the elements to form the residual and the application of the inverse of the mass matrix have time complexity $O(n)$. The result of this fact is that the performance of implicit codes is dominated by the solver whereas the performance of explicit codes is dominated by the element application - optimizing performance for these two endeavors requires very different techniques. Note that although iterative methods are about $O(n)$ in time and space complexity, and are often more easily parallelized than direct methods, the constants of iterative methods (for most interesting finite element applications) are still much larger than that of the element code - thus the solver will remain the bottleneck with iterative solvers also.

The optimization of an element formulation requires many different "small" numerical operations (e.g., tensor operations in element material constitution, inversions of small Jacobians, sometimes small nonlinear solves, and many other operations), all of which require intimate knowledge of the finite element formulation. Element optimization has traditionally been done entirely by engineers; whereas direct solver performance is primarily in the realm of a numerical linear algebraist, as sparse direct solvers are used in many disciplines. Thus the different nature of optimizing implicit and explicit codes is one source of the influence of the time integrator on the structure of finite element codes. This dissertation is concerned with solvers (step seven in §2.2), and thus we assume that an implicit time integrator is in use, or an explicit method that uses preconditioning to "soften" the operator [65].

7.2.2 Finite element parallelism

The parallel implementation of the finite element method requires that we augment the basic steps in §7.2.1 and §2.2. At some point the domain requires partitioning, as inter-processor communication costs are inherently a bottleneck if they are not minimized. Traditionally sparse matrix implementations use a block row (or block column) partitioning of the *matrix* storage - this implies that a row or nodal partitioning is sufficient to parallelize the matrix storage, as finite element matrices are structurally symmetric. A high degree of scalability requires partitioning all of the work and storage for each vertex and each element. Any non-parallel constructs, or significant parts of the code that do not scale well, will inflict severe performance penalties as the scale of the problems increase.

The partitioning, or parallelization, needs to be brought into the system as early as possible to allow for the maximum amount of scalability and must itself be parallelized for the scale of problems that are of interest to this dissertation. Additionally the back end of the system (the last step in §2.2) should remain parallel for as long as is possible. We have elected to draw the line for parallelism of the finite element system after the mesh generation but before the partitioning (although if a parallel mesh generator were available it would be simple to insert); and we go back to the serial code after the solve but before the visualization.

7.2.3 Parallelism and graph partitioning

Mesh partitioning is central to parallel computing with unstructured meshes and we thus discuss some of the basic issues and algorithms involved. Mesh partitioning is the assignment of each vertex and element to one processor p of the P processors, thereby defining a set of *local* vertices an elements for each processor. Note, we partition elements and vertices as most of the costs of a finite implementation can be effectively Because of the size of our our finite element problems and the (increasing) ratio of processor speeds to communication speeds, we need to have each *processor* be responsible for many more than one vertex. The physical nature of finite element graphs (i.e., vertices communicate only with physically close vertices), allows graph partitioning to be very effective at reducing the amount of data required from other processors, which is generally several orders of magnitude more expensive to use than local data. Thus, as we want good scalability (at least at the algorithmic level), a nontrivial nodal partitioning (and the related element partitioning) must be computed.

Partitioners attempt to minimize communication cost by minimizing the number of ghost vertices on each processor (ghost vertices are non-local vertices that are connected to at least one local vertex), and the amount of load imbalance i.e., the maximum amount of work for any one processor divided by the average amount of work per processor. In general one wants to weigh the cost of load imbalance and the cost of a ghost, along with the number of neighbor partitions, to define an optimization problem. Most partitioning methods try to minimize edge cuts (as a heuristic to minimizing ghosts), with the constraint that the number of vertices per partition are equal within some tolerance. In attempting to optimize the solution time of an iterative solver it is natural to optimize the performance of the matrix vector product, as this is where a majority of the computation takes place. Also, essentially all of the communication takes place in the matrix vector product, thus its optimization is essential for parallel efficiency. Thus we can, and do, simplify the partitioning problem to that of optimizing the matrix vector product. This optimization is relatively simple, as opposed to partitioning for sparse matrix factorizations. Not all vertices in a finite element problem are associated with the same amount of computation (especially in domains where different physics exist), so it is desirable to not simply place equal number of vertices in each partition but to put equal amounts of work (i.e. floating point operations and data) in each partition.

Partitioners that are of interest today fall into two groups: geometric partitioners [42] and multilevel partitioners [53, 46]. Geometric partitioners have low complexity (as low as $O(1)$ in PRAM, see §1.7) and use vertex coordinates only to *separate* the graph into two roughly equal pieces - applied recursively they can generate partitions with 2^k partitions. These partitioners can be shown to produce partitions with (expected) *separator* sizes within a constant factor of optimal i.e., the size of the set of vertices that separates a graph, with bounded aspect ratio elements, into two unconnected pieces is $O(n^{\frac{D-1}{D}})$, and within a constant of optimal load balance [42]. These methods do not however look at the edges in the graph but instead rely on the "physical" nature of finite element graphs to minimize edge cuts by minimizing the number of vertices "near" a cut.

Multilevel partitioners rely on the notion of restricting the *fine* graph to a much smaller *coarse* graph, by using maximal independent set or maximal matching algorithms. This process is applied recursively until the graph is small enough that a high quality partitioner such as spectral bisection [47] or k-way partitioners can be applied. This partitioning of the coarse problem is then "interpolated" back to the finer graph - a local "smoothing" procedure (e.g., Kernighan-Lin [54]) is then used, at each level, to locally improve the partitioning. These methods have polylogarithmic in n complexity though they have the advantage that they can produce more refined partitions and can more easily accommodate vertex and edge weights in the graph. We use a multilevel partitioner as a good public domain implementation exists - ParMetis [52].

Coarse grid partitioning

Chapter 5 described our method of "promoting" vertices to coarse grids; this chapter has discussed partitioning the fine grid - we have yet to define the partitioning of the coarse grids. We have a natural partitioning for the coarse grids, as our coarse grid vertices are derived from fine grid vertices. This natural partition is, however, not adequate for many classes of problems.

One must explicitly repartition the coarse grids - even if the fine grid is perfectly balanced the coarse grids are in general unbalanced. One source of imbalance is *intentional*, nonuniform coarsening in the domain [64]. Another source of imbalance is nonuniform subdomain topologies (e.g., an area with “line” topology like a cable, and areas with “thin” topology) coarsen at different rates from each other and from “thick body” subdomains (§5.3). On our *uniform* problems, the load imbalance increases only mildly as we go up the grid hierarchy; nonuniform problems however develop severe load imbalance and thus we repartition the coarse grids (note, ParMetis provides services to improve existing partitions with minimal vertex movement).

7.2.4 Parallel computer architecture

Common computer architectures of today have a hierarchy of memory storage components. The top of the memory hierarchy are the registers - data must be in registers to be used. Below the registers are often two or three levels of *cache*, next is main (local) memory, and *other* processor elements (PE) main memories, then disk, and so on. Moving data between levels of the memory hierarchy is generally far more expensive than performing floating point operations on data in registers, thus communication is often the bottleneck in numerical codes. Figure 7.1 shows a schematic of the common computer architectures of today.

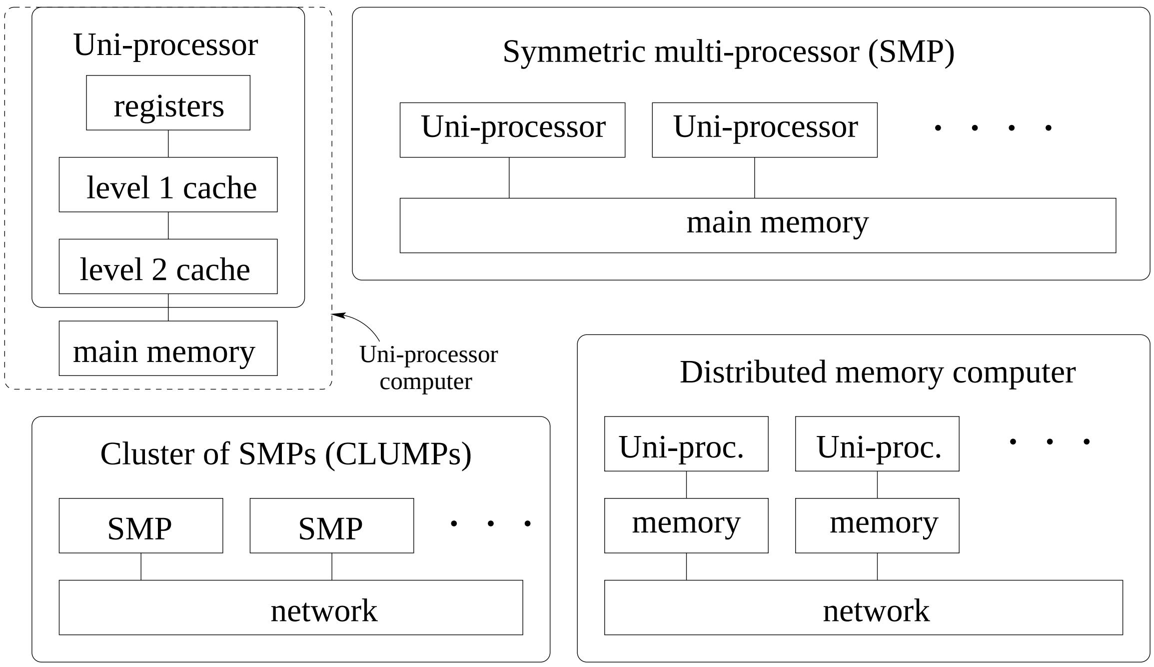


Figure 7.1: Common computer architectures

Some sort of partitioning is required for *all* of these levels, as we see in the next section. We explicitly partition once for "flat" machines, and twice for clusters of symmetric multi-processors (SMPs), which we call "CLUMPs".

7.2.5 Multiple levels of partitioning

A reason for multiple levels of partitioning is minimize movement of data between levels of the memory hierarchy on todays computers. Our problems (multiple degrees of freedom per vertex) are naturally partitioned for registers, though perhaps not optimally. Local matrix vector products require matrix ordering to optimize cache performance (utilizing a graph partitioner is one method [82]).

In the case of CLUMPs, we partition our domain to the SMPs first, as communication is much faster (at least in theory) within an SMP, we then partition to each processor within the SMP. We use a *flat* MPI programming model (one thread per processor); the message passing layer can be optimized with a multiprotocol MPI [62], to communicate more quickly within an SMP than between SMPs, although the SMP cluster that we use (Blue-TR at LLNL) does not yet have a multiprotocol MPI implementation thus multiple levels of partitioning is not currently of benefit. Alternatively a multithreaded matrix vector product could be used, to take advantage of the shared memory, and thus use a non-flat programming model for some parts of the code. The purpose of the second partitioning phase, on each SMP, is to minimize the communication *within* the SMP; even if a multithreaded matrix vector product were in use, this partitioning would serve to minimize cache misses on the left hand side vector.

7.2.6 Solver complexity issues

For our overall non-linear solver, we use a Newton-Krylov-Schwarz solution method [55] - that is, our multilevel Schwarz algorithm (discussed in the last chapter) is used as a preconditioner for a Krylov subspace method (discussed in the chapter 2) which in turn is used as an approximate linear solver in an outer Newton iteration (see any numerical analysis text for a description of Newton's method [56]). The complexity of our system is governed primarily by the application of our multigrid preconditioner.

Global communication (such as dot products, or other "small" global reduction operations) are in general required in each of the *control steps* (e.g., time step selection, Newton convergence checking, and solver convergence checking), and within the solver itself if a Krylov subspace smoother is used - these reductions are the most *non-scalable* constructs in our algorithms (i.e., the $\log(P)$ or $\log(n)$ terms in the PRAM complexity). The constants for the $\log(n)$ terms are small, but will increasingly become important as the scale of problems increases.

7.3 A parallel finite element architecture

Our code - ParFeap - is composed of three basic components:

- **Athena** is a parallel finite element code (without a parallel solver) that uses ParMetis [53] to partition the finite element graph, then constructs a fully valid "serial" finite element problem, and finally runs a serial research finite element implementation - Finite Element Analysis Program (FEAP) [36] - to provide a well specified finite element problem on each processor.

- **Epimetheus** is an "algebraic" multigrid solver infrastructure that provides a solver to Athena, a driver for Prometheus and an interface to PETSc [10] (§7.3.2) and numerical primitives not provided by PETSc (i.e., the matrix triple product and a Uzawa solver §10.6).

- **Prometheus** is our restriction operator constructor - the core of the research work for this dissertation.

Athena, Epimetheus, and Prometheus are implemented with about 30,000 lines of C++ code, PETSc and ParMetis are implemented in C, and FEAP is implemented in FORTRAN.

7.3.1 Athena

We have developed a highly parallel finite element code, built on an existing serial research "legacy" finite element implementation. Testing iterative solvers convergence rates requires that challenging test problems be used - test problems that test a wide range of finite element techniques. We needed a full featured finite element code, but full featured finite element codes are inherently large, complex, and not easily parallelized. Thus by necessity we have developed a domain decomposition based parallel finite element programming model, in which a *complete* finite element problem is built on each processor. This abstraction allows for a very simple though expressive interface, and required only very simple modifications to FEAP.

Athena uses ParMetis [53] to calculate a mesh partitioning, and then constructs a fully valid (though not necessarily well posed) finite element problem for each processor. In addition to the "local" vertices for processor p, prescribed by the vertex partitioning V_p^L (§5.2.1), duplicate vertices are required (i.e., "ghost" vertices that are required for some local computation but are not in V_p^L). The partitioning also requires duplicate elements, as each partition must have a copy of all elements that touch any local vertex if one wishes to calculate the stiffness matrix without any communication (although with redundant computation). The displacements or Dirichlet boundary conditions must be applied redundantly (i.e., on ghosts), whereas the loads (Neumann boundary conditions) must be applied uniquely to maintain the semantics of the problem (as residuals or forces are added into a global vector) - if elements are *not* redundantly applied (otherwise the residuals for the ghost vertices are ignored). Materials are specified by index, bound to the elements in the partition, and specified at run time with a "material file" described below.

A *slightly* modified serial finite element code runs on each processor. Though the serial code is modified it does not have *any* parallel constructs or knowledge of the global problem - this is useful for debugging and the continued independent development of FEAP. Parallelism is introduced by providing the finite element code with a matrix assembly routine, solve routine, and a *global* dot product (additional support functions are provided for expressiveness and performance but are not strictly necessary). The global dot product is *hardwired* for a vector of the type x or b in an equation like $Ax = b$ (where A is the stiffness matrix for the entire local finite element problem), and thus allows for a very simple *parallelization* of the serial code, that with the addition of a "solver" is adequate, for all or most of the global operations in all finite element codes. This simple interface could also be adequate for a simple explicit method, where the solver simply needs to invert a diagonal mass matrix and do a component by component vector-vector product - and then *communicate* the solution, on local vertices, to neighbor processors. Any method that requires other global operations (e.g., a solver with solver specific initialization routines) can be added as needed - thus this interface provides the *kernel* for a full featured parallel finite element implementation. The advantage of this method is that the serial finite element code (e.g., FEAP) is *completely* parallel - and has a very small interface with the parallel code (e.g., Athena). With the addition of a solver that allocates the parallel stiffness matrix and global vectors and solves a system of the form $Ax = b$, many finite element codes (among those used in industry and academia) are *ready-to-run*.

7.3.2 Epimetheus

Prometheus provides Epimetheus with restriction operators, and Athena uses Epimetheus to solve the equations. Epimetheus uses METIS [52] to determine the block Jacobi subdomains and PETSc for the parallel numerical library and programming development environment. Future work may include adding an interface to the parallel unstructured multilevel grids that we use to construct our coarse grids as these are generally useful for anyone building a parallel "algebraic" multigrid code, or in fact any parallel algorithm on unstructured multilevel grids, similar to how Kelp [8] and Titanium [85] provide parallel multilevel *structured* grid primitives.

7.3.3 Prometheus

FEAP provides Prometheus with the local finite element problem (that was originally constructed by Athena) i.e., coordinates, element connectivities, material identifiers, and the boundaries conditions. Prometheus constructs the global restriction operators for each grid, and is the core of the algorithmic contribution of this dissertation discussed at length in chapter 5.

7.3.4 Athena/Epimetheus/Prometheus construction details

This section discusses some low level details of our code construction; this is not intended to be a manual or specification but is intended to provide enough details so that the interested reader can get some idea of the interface and architecture. Note, for the interested reader, the FEAP manual can be found in [36].

To run our system one must first input a problem into a serial version of FEAP, using FEAP's simple mesh generation, then output a mesh file (a FEAP input file) in a "fixed" format so as to allow it to be read efficiently in parallel - this file is the input to Athena. If one had a scalable mesh generator to provide the mesh at run time, then one would only need disjoint vertex and element partitionings and provide each processor with its vertices and elements (there are no constraints on the properties of these partitions other than being disjoint). Alternatively one could transform a pre-generated mesh into our (flat FEAP) format easily as it is very simple.

One can now run ParFeap with three input files: the large *flat* FEAP input file and two small files, one with the material parameters and the other with the *FEAP-solution-script*; FEAP's command language is used to invoke Athena routines from this *FEAP-solution-script*, as well as invoke FEAP commands (see Figure 7.2). We will not discuss the "material file" further, but the *FEAP-solution-script* is of interest as it uses FEAP's command language and is an important component of the user interface with ParFeap (again, documentation details may be found in [36]). Athena can now be run with *any* number of processors, including one processor, with no further preparation. This flexibility drives this design, as it is paramount for solver development (e.g., debugging, experimenting with new methods, and collecting performance data).

Within the *FEAP-solution-script*, displacements can be written to a file after time step is complete. The serial version of FEAP is then invoked, FEAP's "read" command is used to open the file (for example "read,disp" reads in the displacements in a file named "disp"), and the results can then be visualized using the serial version of FEAP. If a more sophisticated visualization tool were available, it could be inserted here. Figure 7.2 shows an example *FEAP-solution-script* that initializes Prometheus (mgin), makes four coarser multigrid levels (mg++), finalizes the data structures (mgfn) - then does a simple linear pseudo time stepping problem and writes out the displacements at each step.

Figure 7.3 shows a graphic representation of the overall system architecture.

```
batch          ! begin batch block
mgin           ! initialize Prometheus
loop,,4
    mg++       ! make a new level
next
mgfn           ! finalize the multigrid setup - prepare for a solve
dt,,0.01       ! set the "time" step
prop           ! use a "prop"ortional load
loop,time,10   ! loop for 10 time steps
    time       ! increment the time
    tang,,1    ! form and solve the tangent (stiffness) matrix
    mgds       ! write a PETSc displacement file
next
mgds,end       ! collect displacements and write to FEAP file
end            ! end batch block
inte           ! begin interactive mode
```

Figure 7.2: FEAP command language example in the *FEAP-solution-script*

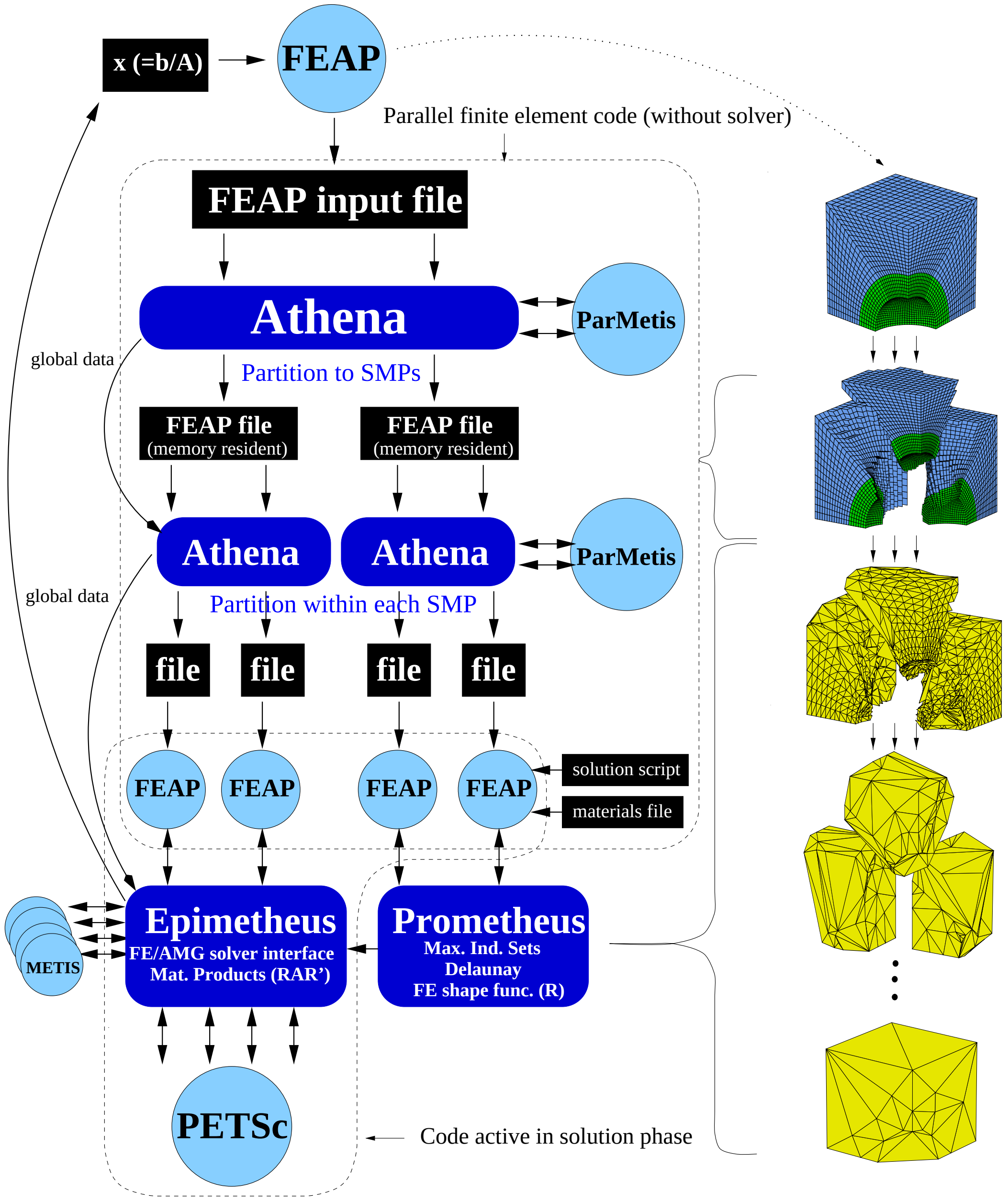


Figure 7.3: Code Architecture

7.4 Processor subdomain agglomeration

Subdomain agglomeration refers to reducing the number of active processors and coalescing (agglomerating) the subdomains on each processor to a new reduced set of active processors. Subdomain agglomeration is valuable for two entirely different reasons, first for performance and second for algorithmic considerations. As mentioned in §5.2 we partition many vertices (i.e., $O(10^4)$) to each processor - this allows all of the processors to do useful work in much of the multigrid algorithm - for the "large" problems of today (e.g., 10^6 - 10^8 vertices). The difficulty is that, as the number of vertices per processor dwindles the ability to do work efficiently decreases. At some point in the grid hierarchy it is *efficient* to let some processors remain idle and agglomerate the work to fewer processors - that is, the time spent on a grid will *decrease* if fewer processors are used. Note, agglomeration, and more significantly repartitioning, can slow the restriction and interpolation operators on the previous grid - to *some* degree - though we are not able to quantify this well and thus must rely on the *total* run time to justify, and optimize, agglomeration. Also, agglomeration requires data movement in the setup phase, though as agglomeration takes place when there are very few vertices per processor the cost is not significant.

A second reason for the use of processor agglomeration comes from the multigrid algorithm that we use, both in terms of mathematics and practical implementation issues. Mathematically, when any multigrid algorithm uses a block Jacobi preconditioner in the smoother you no longer have the "same solver" in parallel, as on one processor, since one can no longer maintain the same block sizes on the coarser grids (assuming that a serial solver is used on the subdomains).

Agglomeration should take place when the value of the global Mflop rate (e.g., Figure 7.4 (left)), of the *operator* that one wishes to optimize on your machine, using the current number of *active* processors, falls below that of a smaller integer number of processors. We need a methods to pick the number of processors to use given the structure of the stiffness matrix at each level of multigrid. This section will discuss three approaches to determine, at run time, the number of processors to use for the coarsest grids. First §7.4.1 will discuss a simple method and introduce some of the concepts and tools involved in subdomain agglomeration algorithms; §7.4.2 defines the problem more concretely as an integer programming problem. A sophisticated, though complex and intractable, approach will be discussed in §7.4.3, and the method that we use in our numerical results (§9.6) will be discussed in §7.4.4.

7.4.1 Simple subdomain agglomeration method

We can quantify the point at which this agglomeration *should* take place for a particular problem by measuring the performance of the operation (or mix of operations) one is optimizing for on each level. A simple method is to first measure the performance of the operator on a range of matrix sizes and number of processors. For example, if we approximate our operator (multigrid on one given level) by a matrix-vector product, then we can use Figure 7.4.

A simple subdomain agglomeration method is to decide on the "optimal" number of equations to have on each processor B and, given the number of equations on a grid n_i, use $P_i = \lceil \frac{n_i}{B} \rceil$ processors if $P_i \leq P$. From this data we *could* conclude that the optimal number of equations per processor is *about* 300 on the Cray T3E, as this is about where we have peak floating point operation (flop), or megaflop (Mflop), rates.

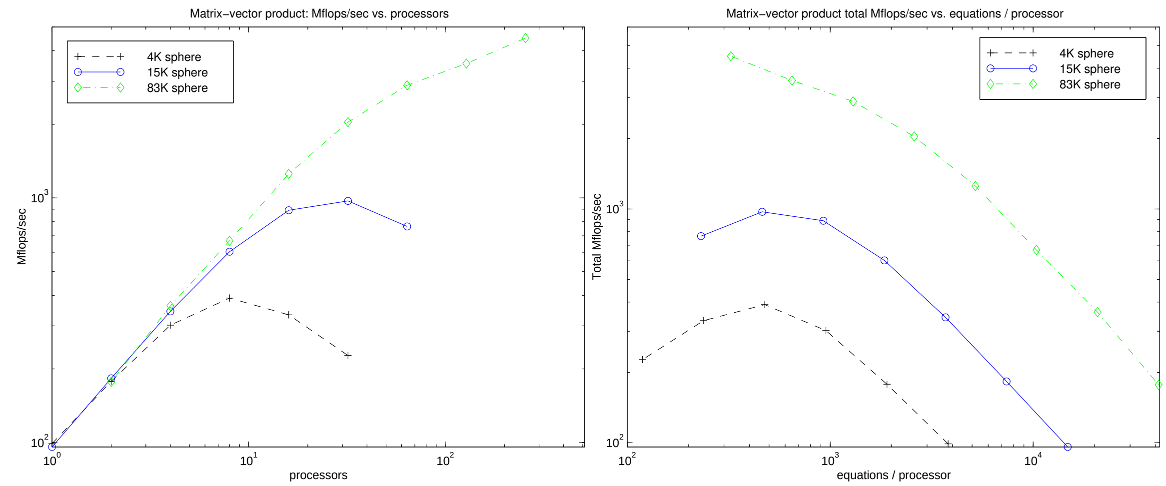


Figure 7.4: Matrix vector product: per processor, and total, Mflop/sec on a Cray T3E

Experience has shown that this *very* simple method is not adequate (at least on some of the platforms that we use), as the optimal number of equations per processor is a function of P. Also, as our coarse grids tend to be denser than the finest grid, the number of non-zeros in the matrix should also be considered, though we do not currently consider the number of non-zeros.

7.4.2 Subdomain agglomeration as an integer programming problem

The optimal number of processors is a constrained optimization problem. First we define our constraints: obviously we need an integer number of processors, but we are not free to select any number of processors in our system as our implementation is not completely flexible in the number of processors used. For simplicity of the implementation, we require that the number of active processors is a factor of the number of processors on the previous (finer) grid. For example, if we have six levels and 36 processors, then the series of processor group sizes {36 36 36 18 12 1} is not valid, as 12 is not a factor of 18; while the set { 36 36 36 18 9 1} is valid. Note, this constraint is not onerous as the number of equations drops by a significant fraction (e.g., 4 to 8) on the coarser grids, though it does demand that we avoid using a number of processors with large primes in their factors.

We want to pick the number of processors to minimize the time to do the work on a particular level. We neglect the effect of agglomeration on the restriction and interpolation operations as they are not large on machines with good networks, are difficult to measure accurately without effecting the performance, and there inclusion would significantly complicate the optimization problem (which, as we will see, is already too complex for us to do directly). Most of the flops, in a level of multigrid, are in the matrix-vector products and the block Jacobi preconditioner. Block Jacobi preconditioners require no communication, hence motivate the use of as many processors as there are blocks. Matrix-vector products have some nearest neighbor communication and, do not speedup perfectly forever as seen in Figure 7.4. Additionally, the dot products require global communication, but perform very few flops per dof. Hence the dot products push for fewer processors than other operators, especially on machines with slow communication.

To select the number of processors P to use on a coarse grid G with an integer programming technique, one needs an explicit and accurate function $T(G,P)$ for the execution time of a particular grid on a particular machine with any number of processors. Integer programming could then be used to fine the minimum, with respect to P, of this function T to decide on the number of processors to partition the current grid on to. The next section will discuss issues in construction a function T via a computational model.

7.4.3 Potential use of a computational model

We can pick the optimal number of processors by modeling the time complexity of a level of multigrid as a function of the number of processors P, and the number of vertices n as follows. First we define complexity as the *maximum* time T_x that any processor spends in operation x. Note, this is a different definition of T than we use in chapter 8 (i.e., T = maximum time $\times$ number of processors), because here we have already "paid" for the processors and gain no benefit from using fewer processors, thus this as a somewhat unusual cost model. Define fp_1: the minimum flop rate on any processor for BLAS1[1] operators, α: the maximum message latency, β: the maximum inverse bandwidth between any two processors for a double precision floating point number, and ndf: the number of degrees of freedom per vertex. In other words the time to send a message with n words is $\alpha + n \cdot \beta$. A simple model of dot products is

$$T_{dot}(P, n) \leq \frac{2 \cdot ndf \cdot n}{P \cdot fp_1} + (\alpha + \beta)\log(P)$$

We will assume perfect load balance and ignore network contention, thus each processor has exactly $\frac{n}{P}$ vertices and α and β are not functions of P. Note, as all the cost components are the maximum costs on any one processors, and we are looking for the maximum *total* time on any one processor, we use $T_{dot}(P, n) \leq \cdots$

To model the cost of the matrix vector product, we can use a simple model (see §8.5.4 for a more detailed model); assume all vertices have 27 neighbors (i.e. neglect the boundary and assume that the problem is uniform), and fp_2 is the minimum flop rate of matrix vector products on any one processor, Assume there are $6 \cdot (\frac{n}{P})^{2/3}$ ghosts vertices (see §8.5.4), then the matrix-vector product complexity could be modeled as

$$T_{matrix-vector}(P, n) \leq \frac{2 \cdot 27 \cdot ndf^2 \cdot n}{P \cdot fp_2} + 6 \cdot ndf \cdot \beta \cdot \frac{n^{\frac{2}{3}}}{P} + 22 \cdot \alpha$$

This assumes the maximum number of neighbor processors is 22, as this is a common maximum that we observe in our numerical experiments.

[1]**BLAS1, BLAS2, BLAS3**: Basic linear algebra subroutines (BLASs) refer to a set of subroutine definitions that define an interface for highly tuned machine specific implementations of standard linear algebra operators. BLASs operations fall into three categories, defined by the algebraic objects in the operation. A BLAS1 subroutine operates on vectors and scalars only. BLAS2 operate on at least one vector and one matrix, and BLAS3 operate on matrices only. These categories are useful as they accurately define the performance of the operations within them, with BLAS3 operations being the fastest and the BLAS1 being the slowest.

We can model the cost of a level of multigrid with diagonal preconditioning by two dot products and one matrix-vector product i.e., $T_{MG}(P,n) \leq 2 \cdot T_{dot}(P) + T_{matrix-vector}(P)$ ignoring the lower order terms e.g., smoother preconditioner, AXPYs. We can minimize the cost by solving $\frac{d}{dP}T_{MG}(P,n) = 0$ for P (assuming that our $T_{MG}(P,n)$ expression is and equality), with

$$\frac{d}{dP}T_{MG}(P,n) = \frac{2(\alpha+\beta)}{P} - 4 \cdot ndf \cdot \frac{n}{P^2 \cdot fp_1} - \frac{27 \cdot ndf^2 \cdot n}{P^2 \cdot fp_2} - \frac{4 \cdot ndf \cdot \beta \cdot n^{2/3}}{P^{5/3}} \tag{7.1}$$

This simple model is however not accurate, as the load imbalance has not been considered, and the α and β terms need to include the costs of packing, and unpacking, vector data in the matrix-vector product (note, we define α and β to include this in §8.5.4), but this reduces the generality of these terms and requires that they be derived from numerical experiments on typical problems. Also, more significantly, the matrix-vector product expression implies that using more processors always lowers the run time - this is not an accurate assumption, as β is a function of P, as Figure 7.4 demonstrates for small problems (i.e. typical coarsest grids). We discuss modeling via operation counting in more detail in the next chapter, as well as reasons for the discrepancy between these models (see §8.5.5 for matrix-vector product models in more detail).

7.4.4 Subdomain agglomeration method

We currently work with *empirical* models, based on curve fitting, that use measurements from actual solves. This method has the disadvantage that model parameters have to be calculated for each machine that the code is to be run on, and may need to be recalculated for different problem types, but it has the advantage that it is resonably accurate and simple to construct.

We can start by looking at in Figure 7.5 - the *form* of the function f that we want. The concave shape of the curve, for f, in Figure 7.5 is derived from the intuition that as more processors are in use the $\log(P)$ term in the dot products, and any other source of parallel inefficiency, will push for the use of fewer processors. For convenience we transpose this function to get the convex function $n = g(P)$.

Now it is natural to *begin* to approximate this function with a quadratic polynomial $n = AP^2 + BP$, or alternatively

$$\frac{n}{P} = A \cdot P + B$$

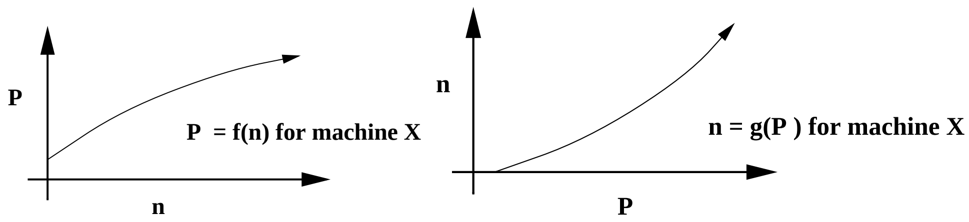


Figure 7.5: Cartoon of cost function, and its transpose

where $\frac{n}{P}$ is the problem size per processor. or the function that we actually want

$$P_{float} = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \tag{7.2}$$

A and B are machine and problem dependent parameters that are determined by experimental observation on actual multigrid solves of the problem at hand, or from experience with other problems on the target machine. We have selected A and B by starting with starting with an initial guess and using a large problem to "search" for the optimal solve time by perturbing A, measuring the total performance (the only quantity that we can effectively measure), and "search" for an "optimal" A (we assume that solve time is a convex function of A); we repeat this process for B, and go back to A, and so on until we find the minimal solve time for this one problem. This will give us good results on similar problems as the one that we test on.

In a production setting one could automate this process for selecting the coefficients (e.g., A and B) for a polynomial (e.g., equation 7.2), by running parametric experiments (parametric in A and B) with a large representative problem; one could then simply select the A and B used in the experiment with the fastest solve time, or use curve fitting to construct a function that can be minimized. Note, this process would be repeated for each new machine or machine configuration and for each problem class.

The use of many more processors would likely require that a higher order polynomial or a more complex function be used. Note, for more accuracy one should also use the number of non-zeros in the matrix in addition to the number of equations n, as this is a more direct measure of the matrix vector cost, and does *not* remain constant on all grids. Equation (7.2) provides us with reasonable results, as we run homogeneous problems (i.e., trilinear elements with three dof per node) with an (approximately) even distribution of non-zeros.

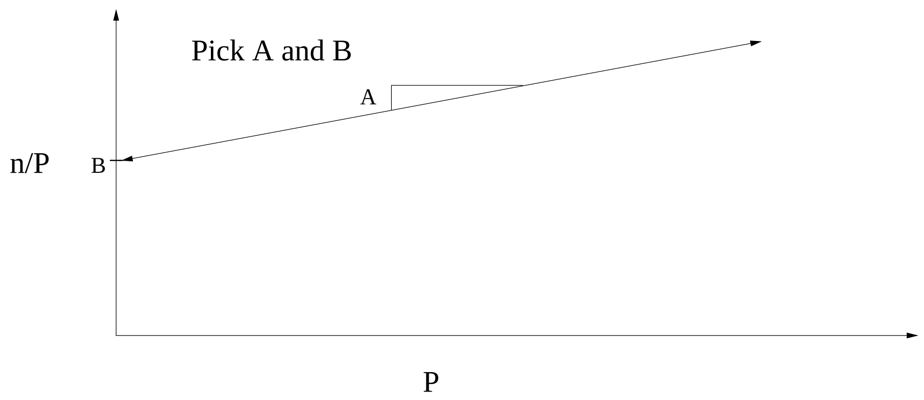


Figure 7.6: Cartoon of the “fitted” function

After the number of vertices n on a grid has been determined at run time, A, B, and equation (7.2) are used to compute P_{float}. After computing P_{float} we find the first feasible integer in each direction, above and below P_{float}. For example with 35 processors and $P_{float} = 6.2$ in Figure 7.6, we now find the two feasible number of processors e.g., $P_1 = 5$ and $P_2 = 7$ in Figure 7.4.4. We pick P to be the closer of P_1 and P_2 to P_{float} as the

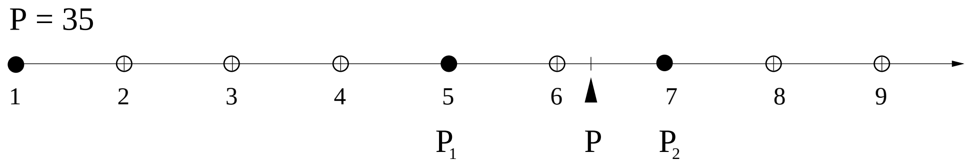


Figure 7.7: Search for best feasible integer number of processors on 35 processors, if $p = 6.2$

result of this method.

Chapter 8

Parallel performance and modeling

This chapter discusses performance modeling for multigrid solvers in general and our solver in particular.

8.1 Introduction

Accurate time complexity models based on operation counts are very challenging on todays parallel machines; even for regular problems (i.e., parallel dense linear algebra) accurate computational models can be intractable because of the many contributors to performance and their complex interactions [79]. Previous multigrid modeling work, for regular 2D grids on a 1,024 processor NCUBE (run with approximately 10^{-2} times as many non-zeros per processor as is typical today), can be found in [84]. This chapter discusses complexity issues and models of 3D unstructured multigrid solvers.

The goal of this chapter is to develop the basis for a useful model to predict the run time, and memory usage, of multigrid solvers with unstructured finite element meshes on typical computers of today. An accurate complexity model of this type is beyond the scope of this dissertation, however, we will discuss qualitative models useful in understanding the overall complexity of multigrid solvers, begin to develop quantitative models of some of the major components of multigrid solvers, and present a framework for a multigrid complexity model. Additionally this chapter will discuss our performance measures, which will be used in our numerical experiments in chapters 9 and 10. Thus, the primary goal of this chapter is to introduce the overall complexity structure of multigrid on unstructured meshes and begin to develop a complexity model via a decomposition of multigrid and analysis of some of its components.

Our complexity models will first decompose multigrid into components whose interaction, with each other, can be ignored. We assume that all of our models are static in that we are able to count for all of the work to be done in a solve, at least to the level of detail that we articulate in our decomposition. At some point our decomposition takes the form of a functional decomposition (e.g. time to apply an operator = number of flops × flops per second). At the component level one may resort to a *bottom-up* model that is based on low level (and general) machine specifications such as clock rate, cache size, network specifications, etc, or one can use a *top-down* approach of measuring the code component on a particular machine with various inputs and curve fit the function to approximate the measured data [15]. See Table 8.1 for the pros and cons of bottom-up and top-down approaches.

Pros and cons	
top-down	bottom-up
Needs running implementation on a give machine	can predict performance on new machines, or without implementations
can predict performance only for different parameter values on a given machine	may badly underestimate time depending on whether all operations are counted
takes all interactions into account	may be very hard to count all operations or model their interactions
may or may not account for irregular operations	irregular operations may require approximating work per processor

Table 8.1: Top-down vs. bottom up performance modeling

This chapter proceeds as follows: §8.2 discusses the motivation for complexity modeling, the notation and the complexity models that we use. §8.3 presents a simple computational model based on PRAM [41] that is used to speculate on the scale of future multigrid solves and is intended to introduce multigrid complexity by providing an overall view of the complexity of multigrid solvers. §8.4 discusses the high level structure of our cost model, and introduces our primary metric (efficiency). §8.5 discusses the scope of our model, the primary cost components of the multigrid solution phase, models for some of the primary solver components, and quantitative measures of the complexity of unstructured multigrid solvers. §8.6 will conclude with a sketch of a path for continuing to develop our complexity model.

8.2 Motivation, computational models, and notation

There are several reasons for developing performance models and performing complexity analysis of a multigrid code. First, multigrid solvers have many parameters (e.g., number of processors, subdomains, smoothing steps, etc.) and many algorithmic choices (e.g., additive or multiplicative methods) - so one would like to make these choices on a rational basis since experience shows that performance can be a sensitive function of these parameters. A second use of performance modeling is to identify bottlenecks in the code that are an artifact of a code bug, a system problem (e.g using non-optimal system parameters or system bugs), platform specific behavior (e.g., a poor contention management implementation in the communication system might require more synchronous communication patterns), or some other "fixable" problem. A third reason for performance models is to project the behavior of your algorithm on new or future machines so to make informed purchasing decisions and so that system manufacturers can make informed decisions about the their system design (assuming that the manufacturer is interested optimizing the performance of their products for your application). Additionally performance models can be used to chose the optimal number of processors to employ at any given level as discussed in chapter 7, and to estimate resource needs on shared machines.

8.2.1 Notation and computational models

We need to decide which operations to count in our performance model. We use two complexity models of the underlying machine: PRAM and LogP [26].

For some algorithms we use the RAM (random access memory) and PRAM (parallel RAM) complexity models, as a basis for high level discussions and occasionally enrich them. The RAM model makes the simplification that all memory can be accessed in constant time (1) and primitive operations (i.e., $+, -, \div, \times$ also take the same constant time. The PRAM model builds on the RAM model and includes parallel constructs - it assumes that sending and receiving data, from and to any process, can also be done in constant time per data *word*. PRAM models come in many types: all combinations of *concurrent(C)/exclusive(E)* and *reads(R)/writes(W)*, concurrent reads and exclusive writes

(CREW) being the most intuitively realistic for a general computer program, queued write and others. We do not distinguish between these PRAM models in our work as we work primarily with a distributed memory programming model, however if it is not otherwise stated we assume the CREW PRAM model. The advantage of the PRAM model is its simplicity; this simplicity is also its limitation as communication costs can be very complex.

The second model that we use, LogP [26], models the communication system more realistically. We use the LogP model in the more detailed discussions of multigrid complexity - LogP models, the overhead in communication (overhead o), network (inverse) bandwidth and capacity (gap g), and network latency (latency L). P stands for the number of processors.

The overhead o is the time that the processor uses to process an MPI message plus the time that our solver (PETSc actually) spends collecting, packing, unpacking and placing data to and from a message. Gap g is the period during which messages (of a given size) can be processed by the network without stalling. Latency L is the time that it takes between the time that message is sent (by the application, via an MPI call) and the time that the application receives the message on another processor. We will use a simplified version of LogP and let $\alpha = O + L$ and $\beta = g$. Often empirical data is available in the form of measured time for an operation t, the number of processors involved in communication p, and an average number data words n in the communication with each processor, thus we look for functions of the form $t = f(p, n) = p \cdot (\alpha + n \cdot \beta)$.

Notation

Multigrid requires many parameters; Figure 8.1 labels the components and notation, of a multigrid solve, that we will use throughout this chapter.

$L \equiv$ number of levels in multigrid

$i \equiv$ grid number ($i = 0$ for the finest grid and thus $i = L - 1$ for the coarsest grid)

$P_i \equiv$ number of processors active on grid i

$n_i \equiv$ number of total degrees of freedom on grid i

$D \equiv$ dimension of problem (e.g., 3)

$ndf \equiv$ degrees of freedom per vertex (e.g., 3)

$s \equiv$ number of pre/post smoothing steps (e.g., 2)

$k \equiv$ number of iterations in global linear solution

$b_i \equiv$ number of blocks in block Jacobi preconditioner on grid i

$A_i \equiv$ matrix on level i

$R_i \equiv$ Restriction operator from grid i to grid $i + 1$

$P_i = R_i^T \equiv$ Interpolation operator from grid $i + 1$ to grid i

$Mvec_i \equiv$ Matrix vector multiply on grid i ($A_i \cdot x \rightarrow b$)

$VecDot_i \equiv$ vector dot product on grid i ($y_i^T \cdot x_i$)

$VecNorm_i \equiv$ vector norm on grid i ($\sqrt{x_i^T \cdot x_i}$)

$Axpy_i \equiv \alpha x_i + y_i \rightarrow y_i$

$Restrict_i \equiv R_i \cdot r_i \rightarrow r_{i+1}$

$Interpolate_i \equiv {R_i}^T \cdot x_{i+1} \rightarrow x_i$

$TriProd_i \equiv R_i \cdot A_i \cdot R_i^T \rightarrow A_{i+1}$

$F_i \equiv$ Full factorization of matrix A_i

$S_i \equiv$ Forward elimination and back substitution with the factored A_i

$F_i^j \equiv$ Factorization of the matrix for a subdomain j of A_i

$S_i^j \equiv$ Forward elimination and back substitution with the factored subdomain matrix j of A_i

$N_{neigh} \equiv$ maximum number of neighbors of any processor domain.

$F(x) \equiv$ Flops in one application of operator x.

$C(x) \equiv$ Communication time in one application of operator x.

$T(x) \equiv$ Time to do one application of operator x.

Figure 8.1: Multigrid computational components labels and parameters

8.3 PRAM computational model and analysis

Before we articulate a more detailed complexity model of multigrid solvers, we develop a simple model with an optimistic machine that we hope can be built in the future. We use PRAM-like complexity models (PRAM with some simple enhancements). PRAM is far too blunt a tool to model the performance of these codes, but it does provide a means of seeing the *big picture*. We also exercise this model by speculating on the scalability of the multigrid algorithm that we use, by defining a "base" case with no significant parallel inefficiency (in our model), and stipulating that we are only willing to solve problems with at least 50% *parallel efficiency* (see §8.4.1 for the definition of parallel efficiency); we show that problems of the order of trillions of unknowns could be solved with our current multigrid algorithm. This is meant to bring some perspective to the scale of problems that could be addressed with our multigrid solver.

We will model only the multigrid preconditioner and ignore the accelerator, and assume that we are using diagonal preconditioning for our smoothers. We ignore the AXPYs and other BLAS1 operators that require no communication, and the factorization cost on the coarsest grid L as this is a small constant shared by *all* solves. We also neglect the cost of the coarse grid construction and any subdomain setup, Thus we need only model the matrix-vector products, the dot products, and the forward elimination and back substitution on the coarsest grid.

Assume that we have $64K$ equations per processor on the finest grid - this is a bit larger than our common size but is reasonable on a well configured machine, with about 256 Mb of main memory per processor and a production quality implementation of our solver and PETSc (chapter 11). Assume a reduction of a factor of 8 on each successive grid, and that we have $n_{L-1} = 4K$ equations on the coarsest grid. Let us also assume that we are only willing to tolerate 128 equations on a processor before we "agglomerate" subdomains (§7.4) - and that this number is independent of the number of processors. Note, all of the coarsest grids have 128 equations per processor in this discussion. In this model, the number of levels $L = \log_8(\frac{n_0}{4K}) = \log_8 n_0 - 3$, n_0 being the global number of equations on the fine grid (assume n_0 is a power of eight).

First, define $S(\frac{n_i}{P_i})$ to be the cost, in time, of one level (not the coarsest level) in a multigrid "V" cycle, with $\frac{n_i}{P_i}$ equations on each processor - assume that this is a linear function in $\frac{n_i}{P_i}$, except for the parallel inefficiency of the matrix vector products and the time in the global reductions (dot products) which have a $\log(P)$ in their PRAM complexity. A good communication network is essential in containing the costs of these reductions (see §9.5) - many machines in the past have networks that could handle reductions effectively (CM5, Intel ASCI Red at Sandia National Laboratory [7]). We temporarily neglect the $\log(P)$ terms (two in each iteration of the Krylov subspace method smoother) because they have small constants, the range of performance on machines of today varies enormously, and these terms are difficult to capture in this model - though we accommodate them below. This assumption that the cost of $S(\frac{n_i}{P_i})$ is linear in $\frac{n_i}{P_i}$, assumes that the matrix-vector products scale linearly, this is not realistic as was discussed in §7.4, so we constrain our problems to have have at least $1K$ equations per processor, or penalize grids with fewer than $1K$ equations per processor. Thus, to account for the fact that the coarse grids run slower than the finer grid (i.e., speedup is not perfect *forever*), we add a factor of 4 to the costs of the coarsest grids (this reflects our empirical measurements of communication efficiency, or lack thereof).

Define $D(n_{L-1})$ to be the cost of the coarse grid solve (without the factorization), with n_{L-1} equations and as many processors as we like on the coarsest grid (e.g., $P_{L-1} = 32 = \frac{4K}{128}$). We estimate the solve cost of the coarsest grid $D(n_{L-1})$, in terms of $S(\frac{n_{L-1}}{P_{L-1}}) = S(\frac{n_{L-1}}{32}) = S(128)$, as the coarse grid would use 32 processors if it were treated as a non-coarsest grid. First, we estimate that $S(128)$ is 6 times larger than a matrix vector product with the stiffness matrix on the coarsest grid; this is the number of matrix vector products in $S()$ if we use two pre and post smoother applications and neglect the other terms in the $S()$ complexity (see Figure 8.8 for an inventory of multigrid component applications). Next, we estimate that the flop count in $D(n_{L-1})$ is 5 times larger than that in a matrix vector product on the coarsest grid (this is an approximation using the band width of the stiffness matrix for a cube with $4K$ equations). Thus, $D(n_{L-1}) = \frac{5 \cdot 4}{6} S(128) = \frac{10}{3} S(128)$. Note, an alternative to one small set of processors working on the coarse grid solve is to have n_{L-1} processors factor the coarse grid, and solve for one row of the explicit inverse of the coarse grid stiffness matrix - each solve (i.e., $D(n_{L-1})$) consists of a broadcast of the right hand side vector, a dot product of size n_{L-1} (on n_{L-1} processors), and a special interpolation operator [22].

Now we can state a cost estimate of full multigrid in equation (8.1)

$$T(n_0) = S(64K) + 2 \cdot S(8K) + 3 \cdot S(1K) + \sum_{i=3}^{L-2} (i+1) \cdot 4 \cdot S(128) + L \cdot \frac{10}{3} \cdot S(128) \quad (8.1)$$

The factors (2, 3, $i+1$, and $L-1$) come from the number time that full multigrid is applied on each level (see Figure 3.6 and Figure 8.8).

Table 8.3 shows some grid statistics with this model; from this we can see that, for example, with only two levels of partitioning (i.e., three distinct processor groups) and $L =$ six levels, we can solve a 128 million degree of freedom problems with $2K$ processors.

Model Configuration (*example* with $L = 6$ and $P = 2K$)			
Grid number (L total)	# active processors	# equations per processor	Total equations
L-1 (5)	32 (32)	128 (128)	4K (4K)
L-2 (4)	256 (256)	128 (128)	32K (32K)
L-3 (NA)	2K (NA)	128 (NA)	256K (NA)
.	.	.	.
4 (3)	$32 \cdot 8^{L-4}$ (2K)	128 (128)	$4K \cdot 8^{L-4}$ (256K)
3 (2)	$32 \cdot 8^{L-4}$ (2K)	1K (1K)	$4K \cdot 8^{L-3}$ (2M)
2 (1)	$32 \cdot 8^{L-4}$ (2K)	8K (8K)	$4K \cdot 8^{L-2}$ (16M)
1 (0)	$32 \cdot 8^{L-4}$ (2K)	64K (64K)	$4K \cdot 8^{L-1}$ (128M)

Table 8.2: Future complexity configuration ($K = 2^{10} \approx 10^3$; $M = 2^{20} \approx 10^6$)

As a thought experiment, let us define a "base" case for which there is no significant parallel inefficiency, and see how large of a problem can be solved, in this model, with 50% parallel efficiency described below. Let us define a "base" case to be, the largest problem in which no processors are idle, except on the coarsest grid, and no grid has a matrix-vector product with less than $1K$ equations per processor. Thus, the base case has $2M$ equations as all processors must be active and have $1K$ equations on the penultimate grid, the penultimate grid must have $8 \cdot 4K = 32K$ equations as $n_{L-1} = 4K$; therefore $P_0 = P_1 = P_{L-2=2} = 32$ and $n_0 = 2M = 32 \cdot 64K$ and $L = 4$. Thus the $2M$ degree of freedom problem with 32 processors is the largest problem with no significant parallel inefficiency. To get a rough idea of the size of problems that can be solved economically, with this performance model, we define an acceptable level of efficiency of 50% relative to our base case; i.e., *we decide* that we are willing to allow for twice the compute time for our largest problem, thus running at one half of the efficiency of the base case.

How many extra grids can we use, with this model and efficiency "pain" tolerance? First let us make a gesture to the $\log(P)$ and assume that 25% (or 50% of the extra time) of our compute costs are going into the communication cost of the dot products - so we only allow for a 50% longer compute time in the model. Note, this is a constant (though large) approximation to the $\log(P)$ terms in the dot products. We have 7 extra levels ($L = 11$) to give a problem size of $n = 4K \cdot 8^{10} \approx 4 \cdot 10^{12}$, or about 4 trillion equations on about seventy million processors (a petaflop machine).

One should note that this model does not attempt change our basic solver configuration (i.e., full multigrid, and Krylov smoothers). Depending on the machine and problem, additive formulations and/or "V" cycle multigrid, may be more economical as, without Krylov subspace method smoothers, "V" cycle multigrid has PRAM complexity of $\log(n)$ whereas full multigrid has PRAM complexity of $\log(n)^2$.

This model is meant to give a big picture view of multigrid complexity issues, to begin to augment the PRAM multigrid models so as to reflect the machines of today and the near future, and to put some approximate constants in the complexity of full multigrid, with Krylov subspace smoothers and accelerators. Thus, this section has introduced the broad outlines of multigrid complexity - we are now ready to build a multigrid complexity theory from the ground up (or from LogP up).

8.4 Costs and benefits

Before we set out to model multigrid solvers we will define what we are trying to accomplish and how we measure success. Our ultimate goal is to solve *all* sparse finite element problems, on unstructured grids - cheaply. The problems that we are concerned with are accurate simulations of complex physical phenomenon in solid mechanics via finite element methods on unstructured meshes; in particular we are concerned with the linear solve or preconditioning of matrices from such methods. The first metric that we introduce is the *benefit* that we wish to enjoy - "*all* sparse finite element problems" is not a feasible goal, nor easily quantifiable, and so we use the number of *degrees of freedom* (dof) as our measure of benefit for which we have *costs*. We implicitly include "*all* sparse finite element problems" by applying our solver to challenging test problems.

Costs can be defined in many ways - we use the run time of sample problems and apply other costs as constraints, (e.g., memory costs are included by assuming that it is free but limited to the size of main memory on most of todays machines). We have attempted to build test problems that are indicative of the "real world" problems that some people want to simulate and thus provide an approximate prediction of the costs of our solver on some demanding finite element simulations. Thus *robustness* is implicitly included in our analysis by the nature and difficulty of our test problems, and by using methods and parameters that are indicative of the most *efficient* way for us to solve these test problems. *Cost* is thus defined as the product of the maximum time required, by any one processor, to solve the problem and the maximum number of processors used.

With costs and benefits defined, we can describe the overall computational structure of solvers that is useful in providing broad categories in which all of our costs reside. There are three basic cost phases of a linear solve.

1. Setup cost per distinct mesh. The setup cost of a configuration or graph e.g., constructing coarse grids and restriction matrices, allocating memory for the data and setting up communicators and buffers for matrix vector products, etc.

2. Setup cost per matrix for a mesh that has been setup. Preprocessing each matrix for a solve e.g., the LU factorization for a direct method, matrix triple products and submatrix factorizations for multigrid.

3. Cost for each right hand side associated with a given matrix e.g., backward substitution for a direct method and the actual iterations in an iterative solver.

All thing being equal, we would in general like to move costs "up" this list, so they may be done less frequently, and optimize the implementations "down" the list as the number of applications of these cost components increases as we go move down the list. The setup cost for each mesh (phase 1) is of the same order as that of one solve on our linear test case in chapter 9 (though this is very machine and problem dependent), and it scales about as well as the solver (see Figure 9.9). Finite element applications require many solves on each configuration and thus this cost can be amortized by each application of the solver, thus we do not focus on the cost for each *configuration*. Figure 8.2 show the high level cost features that we measure: the red boxes, or leaves of the cost tree, are where actual work is done and are the high level code segments that are measured in §9.7 to give an overview of the "end-to-end" costs of a finite element simulation.

Now, we want to solve interesting problems cheaply - but how do we know when (or by how much) we have succeeded? To judge success, we measure *efficiency* - the percentage of "optimal" performance; this gives us a metric that tells us the fraction of "perfection" that we have achieved for the code or any subcomponent.

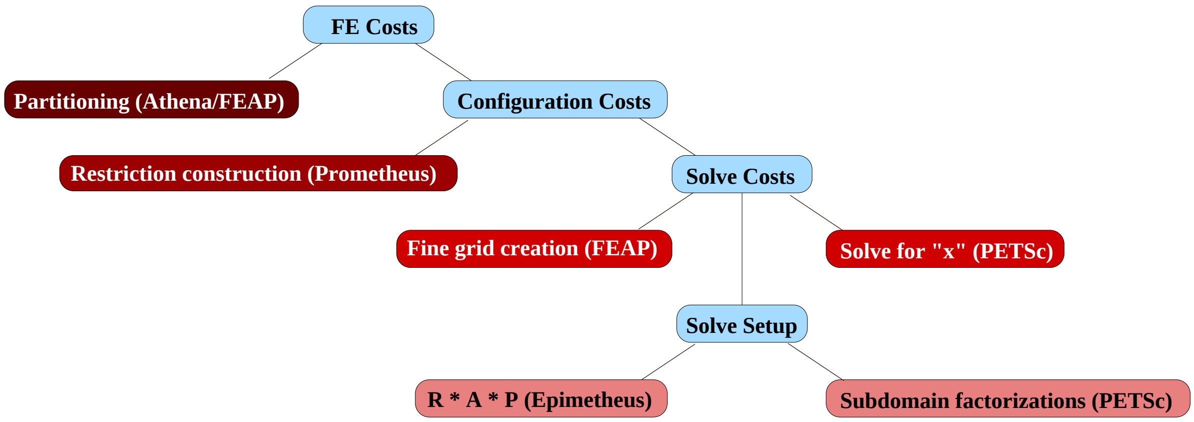


Figure 8.2: Finite element cost structure (code segment from 7.3)

8.4.1 Efficiency measures

We define *parallel efficiency* e, or *scaled efficiency*, as the time to solve a problem of size n_1 on one processor divided by the time to solve a refined discretization of the problem, with n_P equations on P processors, such that $P \cdot n_1 = n_P$. Thus, parallel efficiency $e = T(1)/T(P)$, where $T(P)$ is the time to solve the problem discretization for P processors.

We use time efficiency because it directly addresses the total cost of solving a problem when the machine is being used efficiently. Unscaled "Speed-up" plots (cost of solving a fixed sized problem vs. number of processors), are a useful diagnostic tool to judge the robustness of an implementation but are not useful in directly estimating the cost of using a particular machine. Thus our model of our environment is that we have a large number of processors (with finite memory per processor), many patient users with large implicit finite element problems, perfect job scheduling, and we select the number of processors to use for each job so as to maximize job throughput on the entire machine.

We separate the serial efficiency and the parallel efficiency - the total efficiency being the product of the two. We define the following efficiency measures, or sources of inefficiency:

- **serial efficiency** s: the fraction of peak megaflop rate (Mflop/sec) of the serial implementation. Peak is defined as the Mflop rate of the fastest flop rate on any floating point dominated application. The fastest dense matrix-matrix multiply is used for the peak flop rate if available, and the Linpack [29] "toward perfect parallelism" benchmark is used otherwise.

- **work efficiency** w: the fraction of flops in the parallel algorithm that are not redundant; i.e., the number of flops in the serial algorithm divided by the number of flops in the parallel algorithm - on the same problem discretization.

- **scale efficiency** z: this is the scalability of the algorithm with respect to flops per unknown in the RAM complexity model (i.e., the flops done per unknown as the problem size increases); z is similar to in that it relates to flop inefficiency, though distinct from w. Work efficiency w is related to the number of processors used, scale efficiency z is related to the size of the problem. Note, "scaled efficiency" plots that we use extensively will in general measure *all* of the parallel efficiency measures that are present in the system.

- **load balance** l: the ratio of the average to the maximum amount of work (flops) that a processor does for an operation. This is easily measured (and defined) as we do not use any non-uniform algorithmic constructs (i.e., we do not use task level parallelism).

- **communication efficiency** c: the highest percentage of time that a processor is *not* waiting, processing, packing data, or any other form of work associated with *interprocess* communication.

All of the algorithm components that we use, with the exception of the fine grid matrix creation (FEAP's element state determination), have perfect work efficiency i.e., $w = 1$. In our numerical experiments during the fine matrix creation, each processor calculates *all* of the elements that any vertex on a processor touches, and the stiffness matrix entries that belong on another processors are discarded; this leads to about 80% work efficiency on our test problems and common solver configuration (i.e., trilinear hexahedra and about $25,000$ dof per processor). This redundant evaluation of elements runs faster than exclusive element evaluation, on many machines, as there is no communication required.

Full multigrid is perfectly scalable in the sense that the amount of work (flops) required for each degree of freedom approaches a constant as the number of degrees of freedom goes to infinity, thus z approaches a constant - in theory. In fact, our one processor problems are large enough that the number of flops per equation *is* a constant (or as close to a constant as we can effectively measure), as the problem is scaled up (see Figure 9.6). Thus z efficiencies are close to 1.0 and we do not discuss them further. Therefore we concentrate on the inefficiencies: c,l,s.

Efficiency plots

Efficiency has the property that we can measure or model the sources of inefficiency and simply multiply them to get a total efficiency. Figure 8.3 shows a cartoon of a typical efficiency plot, and it is instructive to point out some of its more salient features.

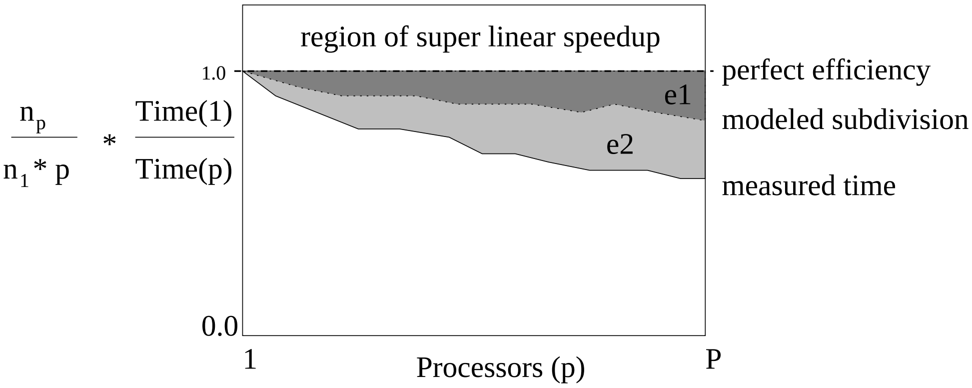


Figure 8.3: Efficiency Plot Structure

If we identify two sources of inefficiency $e1$ and $e2$, and we can model the effect of only one effectively (e.g., $e1$ = load balance, could be estimated with the number of flops on each processors, or some other measurable quantity), then we can plot $e1$ and the total "measured quantity" time in Figure 8.3 - and the ratio is $e2$. Also other curves could be modeled by various means as is described in this section, to provide approximate data on the particular sources of inefficiency. Note, this characterization implies that inefficiencies are decoupled, in the sense that they *could* be isolated, *and* have the same value regardless of the presence of the other - this is not in general true but is often a close approximation in our application. Note, these efficiency plots require that we prescribe the size of each problem n_p for each number of processors p - this is not in general possible so we pick the number of processors for each problem discretization and add a factor $(\frac{n_p}{n_1 \cdot p})$ to the efficiency plots to adjust for this error in our experimental structure.

We also occasionally plot total Mflop/sec against processors used (again with scaled problem discretizations). Total Mflop/sec plots are useful as the data is easily measured for the solve or some part of the code with good load balance and they provide a useful diagnostic tool to see how efficiently the machine is being used in the floating point intensive parts of the code. Mflop/sec data is not useful to compare with non-floating point intensive parts of the code like the partitioning and the coarse grid set up phases; nor is it useful in understanding costs, as they do not include convergence rate - a core component of the cost of iterative methods.

8.5 Computational model of multigrid

This section introduces a more detailed performance model of multigrid. First, this model does not take into account the convergence rate of the solver in question, and thus we only discuss the time to solve a problem with a given number of iterations; configuration costs are not generally included, so the only costs that we will be concerned with will be the matrix setup and the solve costs. Therefore we do not present a *complete* performance model that could be used to, for example, provide the number of subdomains to use in the block Jacobi preconditioner on a given number of processor, on a given machine, and so on. We concentrate on analyzing the run time of components for a particular solver configuration on a particular type of problem, to provide concrete (and more readable) expressions. To avoid a glut of variables we fix the number of iterations at a value typical of our test problems. Note, we show, in chapter 9 that the number of iterations, that our solver requires to achieve a fixed relative reduction in the residual, is not effected by the scale of the problem - thus the assumption of a fixed number of iterations for any one problem is valid.

We further simplify the analysis by making assumptions about the class of problems and the solver configuration, in our model. In particular we assume that we are working with uniformly "thick body" problems i.e., problems in which coarsening can take place in all three dimensions at all levels, throughout the domain. We assume that we are using the symmetrize multiplicative Schwarz method, equation (3.2). Our model could easily be modified to accommodate additive methods but we do not conduct numerical experiments with additive methods as we have not spent time optimizing them (i.e., we have not introduced the task level parallelism that makes additive methods attractive).

We assume that the top (coarsest) grid has about 500 degrees of freedom. This is a rather small problem for the top grid as can be inferred by the fact that the time spent in the top (coarsest) grid is minute compared to the time of the penultimate grid (§9.6). There are two reasons for this less than optimal choice of parameters (i.e., using too many grids)

- memory: we are often memory bound and as we have a fully symmetric programming model using a smaller top grid reduces the *maximum* memory used on any one processor.

- program development: we want to scale to much larger problems, so we want to "harden" the algorithms so as to ease the eventual transition to larger problems. A tiny coarse grid may introduce robustness problems in problem with more complex geometries, so we wish to try to reveal these issues in preparation for larger machines.

We currently use PETSc's serial sparse solver for the top grid; future work will include incorporating a parallel direct solver [58] for the coarse grid problem.

For simplicity we restrict ourselves to one set of solver parameters - typical of our numerical experiments. We assume two ($s = 2$) iterations of a CG smoother, preconditioned with block Jacobi with $n_b \approx 150$ equations per block, thus the number of blocks is $b_i = \lceil \frac{n_i}{150} \rceil$. We let the number of levels $L = \lceil \log_8(n_0) \rceil - 3$ (with $n_0 > 512$), this assumes that the coarsest grid has about $8^3 = 512$ equations. The problem size is $n_0 = ndf \cdot |V|$ equations (ndf degrees of freedom per vertex), and $n_i \approx \frac{n_0}{8^i}$. These will eventually be used in the multigrid inventory of Figure 8.8 to give us the total component count of a multigrid solve as a function of problem size n_0, and the number of iterations k. We denote the cost (in time) of an operator x by $T(x)$, the floating point cost by $F(x)$, and the communication costs by $C(x)$. We do not include load balance in $F(x)$, nor do we explicitly include nonuniform communication or network contention costs between individual processors in $C(x)$.

This section proceeds as follows, first §8.5.1 rewrites the "full" multigrid algorithm in more detail than we have previously. §8.5.2 discusses models for the sizes of coarse grids and "ghost" vertices, used primarily in the communication cost models. The costs of the components are modeled, in terms of floating point operations and flop rates in §8.5.3; communication costs are modeled in §8.5.4. §8.5.5 integrates the communication and floating point costs; matrix vector products are modeled in more detail as they are one of the largest costs in most multigrid solves.

8.5.1 Multigrid component labeling

This section enumerates the complexity structure of full multigrid. First let us show the "full" multigrid cycle that we use, as a preconditioner for a Krylov subspace method, in all of our numerical experiments in Figures 8.4.

$x = MultiGrid\_Fcycle(A_i, b)$

if $A_i.IsCoarsest()$

$x \leftarrow A_i^{-1} \cdot b$ – direct solve

else

$\acute{r} \leftarrow R_i \cdot b$ – restrict the residual b to the coarsest grid

$\acute{d} \leftarrow MultiGrid\_Fcycle(A_{i+1}, \acute{r})$ – recursion to get first correction from coarsest grid

$x \leftarrow P_i \cdot \acute{d}$ – interpolate correction back to this grid i

$r \leftarrow b - A_i \cdot x$ – form new residual

$\hat{x} \leftarrow MultiGrid\_Vcycle(A_i, r)$ – approximate solve

$x \leftarrow x + \hat{x}$ – add correction

$x = MultiGrid\_Vcycle(A_i, b)$

$if\ A_i.IsCoarsest()$

$x \leftarrow A_i^{-1} \cdot b$ – direct solve

$else$

$x \leftarrow Smooth(A_i, b)$ – smooth error

$r \leftarrow b - A_i \cdot x$ – form new residual

$\acute{r} \leftarrow R_i \cdot r$ – restrict the residual r to the coarsest grid

$\acute{d} \leftarrow MultiGrid\_Vcycle(A_{i+1}, \acute{r})$ – get coarse grid correction

$\hat{x} \leftarrow P_i \cdot \acute{d}$ – interpolate correction back to this grid i

$x \leftarrow x + \hat{x}$ – add correction

$r \leftarrow b - A_i \cdot x$ – form new residual

$\hat{x} \leftarrow Smooth(A_i, r)$ – smooth error

$x \leftarrow x + \hat{x}$ – add correction

Figure 8.4: Full Multigrid Cycle

8.5.2 Coarse grid size and density

To model the scale of our coarse matrices, we assume that each grid has a factor of $8(=2^D$ for $D=3$ dimensional problems) fewer vertices than the previous grid. This comes from the assumption that every *other* vertex is being "promoted" to the next grid *in each dimension* - this is true for regular meshes. This assumption is not always true for irregular meshes as even on a regular grid a maximal independent set (MIS) in §5.2 *could* pick every *third* vertex resulting in a factor of 27 reduction at each level.

We intentionally *randomize* the vertex order of the "interior" vertices in our MIS implementation (subject to the constraints in §5.3.5), to *increase* the reduction factor - thus we actually observe reduction factors of about 10 between the finest and first *coarse* grid and about 8 on the subsequent grids. Modeling this reduction factor is further complicated by the fact that the "graphs" on the coarse grid are not, in general, well defined in the sense that they are not graphs of valid finite element meshes (with either tetrahedral or hexahedral elements). In fact our code has *three* graphs associated with each coarse grid i:

- The non-zero structure of A_i $(= R_{i-1}A_{i-1}R_{i-1}^T$ in §2.4.2) which we call G_{exact}
- The graph of the Delaunay tessellation described in §5.3.5 - $G_{Delaunay}$ (note, this *is* a valid finite element graph)
- The approximate adjacencies that we create immediately after the MIS to facilitate efficient implementation of the "symbolic" phase of our code

These algorithmic complexities also present difficulties in making *a priori* estimates of the number of *non-zeros* in the coarse grid matrices. We have observed reduction rates in the number of non-zeros from grid i to grid $i+1$ of about 5 on problem $P1$ in chapter 9. Thus we use 5 for our non-zero (edge) reduction factor for all grids as an approximation.

Number of adjacent processor subdomains and size of "ghost" vertex lists

To model communication costs we first characterize the maximum number of adjacent neighboring processors and the number of "ghost" vertices with which any processor has contact - $N_{neigh} \equiv$ maximum number of neighbors of any processor domain. To estimate these quantities we take advantage of the fact that our problems come from physical domains. The mesh partitioner attempts to minimize edge cuts in the graph - this has the physical analogy of minimizing the *surface* to *volume* ratio of the partitions. Optimal partitions are hexagons for a 2D continuum and rhombic dodecahedrons a 3D continuum. Thus the "optimal" N_{neigh} is about 12 in a regular 3D mesh. Note that for 3D partitioning with cubes N_{neigh} is 26. We use $N_{neigh} = 22$ as this is typical of what we see in practice. That is, $N_{neigh} = 22$ is the *maximum* number of neighbors for any processor, with about $25,000$ degrees of freedom per processor and about 200 processors. Note that this number will most likely go down with increased dof per processor and better mesh partitioners, and up with more processors as statistical effects of nonuniform partitioning come into play.

Now we estimate the number of ghost vertices (off processor vertices which are connected to a local vertex) for each processor (e.g., the number of entries of x that we need, from other processors, to compute the local entries in $y \leftarrow Ax$ in a matrix vector product). Likewise, we need the number of local "boundary" vertices whose values must be sent to other processors. We can again use the physical properties of our graphs by assuming that a processor's subdomain is a sphere with radius r_p, and each vertex "fills" a unit volume. With $|V_p|$ processor vertices we have $|V_p| = \frac{4}{3} \cdot \pi \cdot r_p^3$. By assuming that the thickness of the boundary layer of vertices is 1, the interior vertices V_p^I have a radius $r_p - 1$, thus $\left|V_p^I\right| = \frac{4}{3} \cdot \pi \cdot (r_p - 1)^3$. After truncation of higher order terms we have

$$\frac{\left|V_p^I\right|}{|V_p|} \approx 1 - \frac{3}{r_p}$$

since we have

$$r_p \approx \left(\frac{3\,|V_p|}{4\pi}\right)^{\frac{1}{3}}$$

we get

$$\frac{\left|V_p^I\right|}{|V_p|} \approx 1 - \frac{3}{\left(\frac{3|V_p|}{4\pi}\right)^{\frac{1}{3}}} \tag{8.2}$$

Our numerical experiments use about $15,000$ to $25,000$ dof per processor (about $5,000$ to $8,000$ vertices per processor) on the fine grid. Substituting these partition sizes into equation (8.2) gives us about $70 - 75\%$ interior vertices, which is line with the values that we measure in our numerical experiments. To get an expression for the number of ghost vertices, we assume that the ghost layer is one unit thick, so that $\left|V_p^G\right| = (r_p + 1)^3 - r_p^3$, and after dropping low order terms, we get

$$\left|V_p^G\right| \approx 5 \cdot |V_p|^{\frac{2}{3}} + 8 \cdot |V_p|^{\frac{1}{3}}$$

and similarly as $\left|V_p^B\right| = r_p^3 - (r_p - 1)^3$

$$\left|V_p^B\right| \approx 5 \cdot |V_p|^{\frac{2}{3}} - 8 \cdot |V_p|^{\frac{1}{3}}$$

An alternative model is to assume that the subdomains are *cubes*, we can now us an explicit discrete graph with a regular grid of hexahedra, by assuming that each subdomain has N_p vertices on the side of its cube. The number of vertices per subdomain as $|V_p| = N_p^3$. Further we can model the number of interior vertices (vertices with no contact with other processors as

$$\left|V_p^I\right| = (N_p - 2)^3, \quad N_p > 1$$

and boundary vertices

$$\left|V_p^B\right| \approx 6 \cdot |V_p|^{\frac{2}{3}} - 12 \cdot |V_p|^{\frac{1}{3}}$$

and the number of ghost vertices

$$\left|V_p^G\right| \approx 6 \cdot |V_p|^{\frac{2}{3}} + 12 \cdot |V_p|^{\frac{1}{3}}$$

which is largely similar to the model based on spheres. Again for $\approx 5,000$ to $8,000$ vertices per processor problems we get about $70 - 75\%$ interior vertices.

8.5.3 Floating point costs

This section lists the floating point counts for the components in multigrid - these estimates assume that we are using eight node hexahedral trilinear elements [86]. We assume that our subdomains (for the block Jacobi preconditioner) are $4 \times 4 \times 4$ node cubes, which gives blocks with 192 equations. Further we are oriented toward thick body problems, as this is the type of problem that we use for our linear scalability studies, in chapter 9.

We need a flop *rate* to estimate the costs of floating point operations: Mflop/sec $= 10^6\cdot$ flops / sec. Mflop/sec is a machine dependent parameter, and even on any one machine we subdivide Mflop/sec into several types. Below are five types of $flop$ rates, with the Mflop/sec rate of the appropriate on a T3E and IBM PowerPC in Table 8.5.3, on a typical problem (i.e., about 20,000 unknowns on one processor).

- $Mflop_1/sec$: Dot products AXPYs, any norm, etc.
- $Mflop_2/sec$: Sparse matrix vector products and solves, with a *ndf* $\times$ *ndf* matrix of double precision scalars per block matrix entry

- $Mflop_{2a}/sec$: Sparse matrix vector products, 1 scalar per block matrix entry
- $Mflop_3/sec$: Sparse direct matrix factorization
- $Mflop_{3a}/sec$: Sparse matrix-matrix-matrix products

	T3E (625 Mflop/sec peak)	IBM PowerPC (258 Mflop/sec peak)
$Mflop_1/sec$	85	22
$Mflop_2/sec$	95	37
$Mflop_{2a}/sec$	19	15
$Mflop_3/sec$	193	173
$Mflop_{3a}/sec$	12	14

Table 8.3: MFlop rates for MFlop types in multigrid)

Floating point counts are estimated in some cases by measuring the flop counts of representative problems, otherwise they are models for typical hexahedral meshes. Figure 8.5 shows the floating point counts that we use in our analysis, ($\approx$) refers to values that are measured from numerical experiments, and ($=$) are modeled values.

$F(Mvec_0) = 2 \cdot 80 \cdot \cdot n_0 = 160 \cdot n_i$ - $Mflop_2$
$F(Mvec_i) \approx \frac{Mvec_0}{5^i}$
$F(Restrict_i) = 2 \cdot 4 \cdot n_i$ - $Mflop_{2a}$
$F(VecDot_i) = 2 \cdot n_i$ - $Mflop_1$
$F(VecNorm_i) = 2 \cdot n_i$ - $Mflop_1$
$F(Axpy_i) = 2 \cdot n_i$ - $Mflop_1$
$F(TriProd_i) \approx 5 \cdot Mvec_i \approx 1000 \cdot n_i$ - $Mflop_{3a}$
$F(F_i^j) \approx 30,000$ - $Mflop_3$
$F(S_i^j) \approx 18,000$ - $Mflop_2$
$F(F_{L-1}) \approx 7Mflop$ - $Mflop_3$
$F(S_{L-1}) \approx 0.4Mflop$ - $Mflop_2$

Figure 8.5: Floating point counts for multigrid operators - flop-rate type

8.5.4 Communication costs

Communication costs are difficult to model effectively; their difficulty will require that we use the LogP complexity models (overhead (o) and latency L) described at the beginning of this chapter. As with the flop rate above, there are different overhead o types for different operations; we include the time to "pack" messages, in the application layer, in overhead. Overhead is subdivided into α, the overhead for each processor with which we communicate, and β the overhead for each word (double) being communicated; i.e., the cost to send n words is $\alpha + n \cdot \beta$. We define three types of α

- α_1: Dot products, the *true* machine/system software below the application: $2 \cdot (o + L)$ for the reduction to compute the answer and a broadcast to disseminate it.
- α_2: Vector "scatter/gather" in sparse matrix vector products: posting receives and other per send or receive overhead.
- α_3: Matrix operations in the matrix construction: like α_2 setting up for each send and receive.

In general, N_{neigh} is multiplied by α to get time due to latency in matrix-vector products. Note we do not include the initial setup phase for the data structures in α_2 and α_3 (i.e., allocating buffers, communicating maps between local and non-local vertices for the send and receives, etc.).

For completeness we describe β in the same way as α. Dot product communication bandwidth has no dependence on numbers of vertices (we assume that all active processors on a level i have at least one vertex by definition), so we need only define β_2 and β_3. β_2 could be a function of ndf and V_p^G the number of ghost vertices, where ndf is the number of scalars associated with each vertex in a vector. β_3 could be a function of x, ndf^2, and V_p^G, where an average of x matrix entries are in a row associated with an off-processor vertex that must be communicated. We will estimates these parameters with experimental data, and as we only test $ndf = 3$ problems, we assume β_2 is a linear function of ndf and β_3 is a quadratic function of ndf (we could include ndf, as we do in §8.5.5 for V_p^G, to get a more accurate complexity).

Matrix vector products (β_2) use indirect indexing into a dense vector, to copy ("gather") values to buffer for sending to a neighbor processor; this process is reversed on the receiving side as processors unpack a message and "scatter" it into a sparse vector.

Matrix triple products calculate some off processors values and could accumulate them into a local sparse matrix; β_3 costs occur after the floating point work is complete, each processor will copy their off processor data into messages for sending, and will then receive corresponding messages and accumulate this data into their local sparse matrix. We will not measure α_3 or β_3, as these values are inherently very implementation and data structure dependent, and they are not as large a cost in the solve as the α_2 or β_2. We use α_3 and β_3 here, to derive an order of the complexity for matrix vector products; we provide quantitative measures for α_2 and β_2 in §8.5.5.

In constructing our complexity model we use the subdomain-as-cube model of §8.5.2 to give, on grid i with ndf dof per vertex

$$C(Mvec_i) \approx N_{neigh} \cdot \alpha_2 + ndf \cdot \beta_2 \left(6 \cdot |V_p|^{\frac{2}{3}} + 12 \cdot |V_p|^{\frac{1}{3}}\right) \tag{8.3}$$

Note we ignore latency L as we can overlap communication and computation effectively, at least on the finer grids (see §8.5.5 for details). Additionally we have

$$C(VecDot_i) = 2 \cdot \lceil log_2(P_i) \rceil \cdot (o + L)$$

using the raw machine overhead o and latency L, and

$$C(VecNorm_i) = C(VecDot_i)$$

$$C(Axpy_i) = 0 \tag{8.4}$$

The number of off-processor ghost vertices, for the restriction operator $C(Restrict_i)$, (for which we require values for the left hand side vector) is about the same as that for the matrix-vector product. Again as we have ndf dof per vertex and 4 coarse grid vertices, in general, connected to each fine grid vertex through the restriction operator.

$$C(Restrict_i) \approx N_{neigh} \cdot \alpha_2 + ndf \cdot \beta_2 \left(4 \cdot V_p\right) \tag{8.5}$$

For $C(TriProd_i)$ assume that a processor adds values into *all* graph edges (off-diagonal matrix entries) *to/from* their ghost vertices, and self edges (diagonal entries) *on* those vertices. Additionally a processor can potentially add values to the off-processor edges between two ghost vertices. With this we can estimate the number of off processor matrix entries that a processor sends as the number of ghost vertices (diagonal entries), and edges "between" processors (i.e., $\left|(v,w) \in E^S \mid v \in V_p\right|$ in §5.2.1), and the number of edges *between* ghost vertices. To simplify $C(TriProd_i)$ we assume that all ghost vertices of processor p on grid i have about half ($15 \cdot \left(\frac{8}{5}\right)^i$) of their matrix row entries touched by the processor p, thus

$$C(TriProd_i) \approx N_{neigh} \cdot \alpha_3 + ndf^2 \cdot \beta_3 \cdot 15 \cdot \left(\frac{8}{5}\right)^i \cdot \left|V_p^G\right| \tag{8.6}$$

$$\left|V_p^G\right| = 6 \cdot |V_p|^{\frac{2}{3}} + 12 \cdot |V_p|^{\frac{1}{3}}$$

This expression is in line with our observations on problem $P1$ (chapter 9).

8.5.5 Total cost of components

For most of the components in our multigrid model we can simply add the communication time to the floating point time as there is either no opportunity for *overlapping communication and computation*, or we (or PETSc) have not implemented it.

The only computational component that requires that we articulate the model further is $Mvec$, discussed below. The rest of the multigrid components are simply modeled as the sum of their computation and communication times shown in Figure 8.6.

$$T(Restrict_i) = N_{neigh} \cdot \alpha_2 + ndf \cdot \beta_2 \left|V_p^G\right| + \frac{2 \cdot 4}{Mflop_{2a}/sec} \cdot \frac{n_0}{8^i \cdot P_i}$$
$$T(VecDot_i) = \alpha_1 \cdot \lceil log(P_i) \rceil + \frac{2}{Mflop_1/sec} \cdot \frac{n_0}{8^i \cdot P_i}$$
$$T(VecNorm_i) = \alpha_1 \cdot \lceil log(P_i) \rceil + \frac{2}{Mflop_1/sec} \cdot \frac{n_0}{8^i \cdot P_i}$$
$$T(Axpy_i) = \frac{2}{Mflop_1/sec} \cdot \frac{n_0}{8^i \cdot P_i}$$
$$T(TriProd_i) \approx N_{neigh} \cdot \alpha_3 + ndf^2 \cdot \beta_3 \cdot 15 \cdot \left|V_p^G\right| + \frac{800}{Mflop_{3a}/sec} \cdot \frac{n_0}{8^i \cdot P_i}$$
$$T(F_i^j) \approx \frac{0.003}{Mflop_3/sec}$$
$$T(S_i^j) \approx \frac{0.0018}{Mflop_2/sec}$$
$$T(F_{L-1}) \approx \frac{7}{Mflop_3/sec}$$
$$T(S_{L-1}) \approx \frac{0.4}{Mflop_2/sec}$$
$$\left|V_p^G\right| = 6 \cdot \left|\frac{n_0}{8^i \cdot P_i}\right|^{\frac{2}{3}} + 12 \cdot \left|\frac{n_0}{8^i \cdot P_i}\right|^{\frac{1}{3}}$$

Figure 8.6: Costs for multigrid operators

Matrix vector product cost

To form the total cost of matrix-vector products we look at the matrix data structure in more detail; this is because matrix vector products are responsible for most of the time in a multigrid solve and the communication and computation patterns of our matrix vector products are not obvious and allow for some overlap of communication and computation. PETSc implements its parallel matrix class with two serial sparse matrices on each processor, plus a small amount of global data. The local "short fat" submatrix (block of global rows partitioned to processor p) is column partitioned into the diagonal block (A_p) of the global matrix, and the rest, or off diagonal, local part (B_p). The advantage of this scheme is that A_p only works on the local parts of the source and destination vectors (x and b, respectively), and can thus be done without any communication. The PETSc method of performing $b \leftarrow Ax$ on processor i, is as follows (note, matrix subscripts refer to processor submatrices and not grids)

- post receives for necessary parts (ghost values) of $x_j \mid j \neq i$ (actual receives in the fourth step)
- send necessary parts of x_i to all processors j that "touch" i (corresponding send of receives in the fourth step)
- $b_i \leftarrow A_i \cdot x_i$ (work on local data)
- receive all off-processor entries in x_j
- $b_i \leftarrow b_i + B_i \cdot x_j$

With this we can state a cost model for matrix vector products

$$T(Mvec_i) = 2 \cdot \left(N_{neigh} \cdot \alpha_2 + ndf \cdot \beta_2 \cdot \left|V_p^G\right|\right) +$$

$$ndf^2 \cdot \left(\frac{8}{5}\right)^i \cdot \frac{\left(9 \cdot \left|V_p^B\right|\right)}{Mflop_2/sec} + ndf^2 \cdot \left(\frac{8}{5}\right)^i \cdot \frac{\left(27 \cdot \left|V_p^I\right| + 18 \cdot \left|V_p^B\right|\right)}{Mflop_2/sec}$$

$$\left|V_p^G\right| = 6 \cdot \left|\frac{n_0}{8^i \cdot P_i}\right|^{\frac{2}{3}} + 12 \cdot \left|\frac{n_0}{8^i \cdot P_i}\right|^{\frac{1}{3}}$$

$$\left|V_p^B\right| = 6 \cdot \left|\frac{n_0}{8^i \cdot P_i}\right|^{\frac{2}{3}} - 12 \cdot \left|\frac{n_0}{8^i \cdot P_i}\right|^{\frac{1}{3}}$$

Here we assume that there is enough work in the $b_i \leftarrow A_i \cdot x_i$ term of the matrix vector product to "hide" the latency in the communication; this will not be the case in general, especially on the coarsest grids.

We can measure the maximum time that any processor spends in each of these phases, with varying numbers of processors to estimate the value of α_2 and β_2; table 8.5.5 shows these maximum times for a problem ($P1$ described in chapter 9) with about 25,000 equations per processor on the Cray T3E at NERSC.

Fine grid matrix vector product data, with 25,000 equations per processor			
# processors (# dof x1000)	2 (40)	64 (1,343)	256 (6,489)
Max. send time (max/min) t_{send}^{max}	0.00065 (1.6)	0.0033 (3.6)	0.0038 (2.9)
Max. diag. block mat-vec time (max/min)	0.0318 (1.0)	0.0341 (1.1)	0.041 (1.1)
Mflop/sec per proc. diag. block mat-vec	96	90	87
Max. receive time (max/min)	0.00035 (2.4)	0.0013 (2.9)	0.0018 (4.5)
Max. off-diag. block mat-vec time (max/min)	0.0018 (1.1)	0.0048 (1.9)	0.0054 (2.0)
Mflop/sec per proc. off-diag. block mat-vec	60	57	61
Max. neighbor processors N_{neigh}	1	18	20
Ave. number ghosts in each neighbor proc. n_g	583	180	183

Table 8.4: Matrix vector product phase times

Note, this data is from the standard "summary" output in PETSc, although we added ten lines of code to PETSc's matrix vector product routine (to start and stop timers for each of the five phases).

To use this data to estimate α_2 and β_2, we first notice that our model predicts that the send and receive phases should be equal - this is not the case. The sends are non-blocking, and the receives can potentially block, but for the fine grid this is not likely as there is much work (the local diagonal block matrix vector product) to hide the latency in the sends. We use the send times for this demonstration.

For a model of the slowest processors communication time, we will use the maximum time and the maximum number of neighbor processors (although we do not know if a processor with the largest number of neighbor processors was in fact the processor with the largest time, but it natural to assume so). We will also assume that all of the messages are of the same length, as the average length of messages is the only data that is available, and use this to calculate the average number of ghost vertices in each message n_g. We can now use this data from the three samples to give us three equations in two unknowns of the form $N_{neigh} \cdot \alpha_2 + N_{neigh} \cdot n_g \cdot ndf \cdot \beta_2 = t_{send}^{max}$.

$$\begin{bmatrix} 1 & 583 \\ 18 & 18 \cdot 180 \\ 20 & 20 \cdot 183 \end{bmatrix} \begin{bmatrix} \alpha_2 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0.00035 \\ 0.0013 \\ 0.0018 \end{bmatrix}$$

The least squares fit for this data gives us $\alpha_2 = -0.000025$ and $\beta_2 = 0.0000012$. Clearly, $n \cdot \beta >> \alpha$, so we can neglect the α term (on the fine grid) and calculate β_2 with least squares fit: $\beta_2 = 0.00000103$. Figure 8.7 shows a comparison of this model with the data.

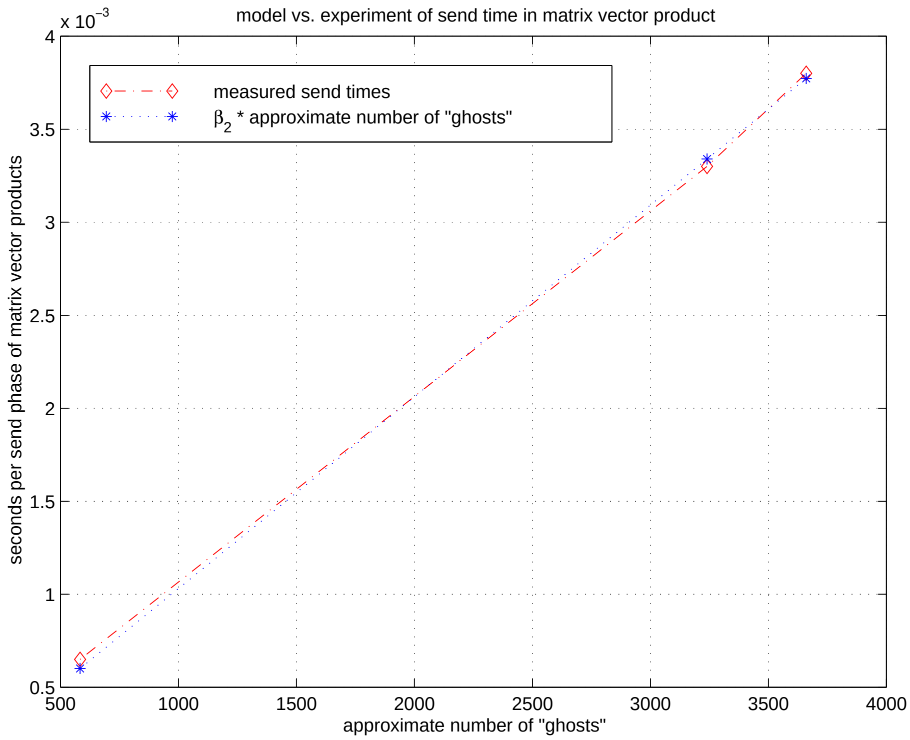


Figure 8.7: Comparison of model with experimental data for send phase of matrix vector product on fine grid

Thus we have an approximate measure of $\alpha_2 = 0$ and $\beta_2 = 1.03\mu$sec on a 440 MHz. Cray T3E, with 3 dof per vertex, and hexahedra mesh of a thick body problem. The next step would be to articulate β_2 to include a term for the average number of vertices per processor to allow for the modeling of the coarse grids - though we conclude the development of this model here.

This exercise is an example of the analyses that would have to be carried out for α_1, α_3, and β_3, for a machine and a problem class, to construct an initial complexity model of multigrid. We do not pursue this model further in this dissertation, but the next section concludes with a sketch of the path for pursuing this model.

8.6 Conclusion and future work in multigrid complexity modeling

This chapter has provided an introduction to the complexity issues of large scale unstructured multigrid solvers on the common parallel architecture and machines of todays. We have developed some of the infrastructure for our complexity model and have articulated some of the components of this model. We however, have not completed this model.

To complete this model one would first need to complete the expressions for α_2 and β_2, developed in the last section, for coarse grid matrices (i.e., extend the model, of α_2 and β_2, as a function of the number of equations per processor and of machine latency); one could then construct models for α_1, α_3 and β_3. These expressions could then be substituted into the component models presented in this chapter; these component models could then be substituted into an inventory (shown in Figure 8.8) of the applications of each component in the solver. The resulting cost estimate could then be compared to experimental data and refined as necessary to develop an accurate model.

After refining this model for thick body hexahedra meshes, one could extend the model to accommodate other types of problem classes (e.g., using thin body problems like shells), by changing some of the appropriate parameters, such as grid reduction rates (in §8.5.2). One would then continue to iteratively refine the model and perturb the problems to develop a more comprehensive accurate model of multigrid solvers on unstructured finite element meshes. Some of the payoffs in the modeling that we have introduced in this chapter would be (as stated in §8.2) to aid algorithmic design, design better computers, inform purchasing decisions, and verify optimal or correct code installations.

1 : Setup Multigrid preconditioner:

- 1: $TriProd_i$, $\forall i = 1, 2, 3, \cdots, L-1$
- 1: F_{L-1} - - factor coarsest grid
- $\forall i = 0, 1, 2, 3, \cdots, L-2 \quad \forall j = 1, 2, 3, \cdots, b_i$: F_i^j - - factor block diagonal matrix for block Jacobi:

k: Conjugate Gradient iterations, with a Multigrid preconditioner:

- 1: $Mvec_0$
- 2: $VecDot_0$
- 3: $Axpy_0$
- 1: $Norm_0$
- 1: Application of $MultiGrid\_Fcycle(A_0, b)$:
  - L: S_{L-1} - - solves on coarsest grid
  - 1: $Restrict_i \ \ \forall i = 0, 1, 2, 3, \cdots, L-1$
  - 1: $Mvec_i \ \ \forall i = 0, 1, 2, 3, \cdots, L-1$
  - 1: $Axpy_i \ \ \forall i = 0, 1, 2, 3, \cdots, L-1$
  - 1: Applications of $MultiGrid\_Vcycle(A_i, b) \ \ \forall i = 0, 1, 2, 3, \cdots, L-2$:
    - $2 \cdot (i+1)$: $Mvec_i$
    - $2 \cdot (i+1)$: $Restrict_i / Interpolate_i$
    - $3 \cdot (i+1)$: $Axpy_i$
    - $s \cdot (i+1)$: Applications of $Smooth(A_i, b)$:
      - 1: $Mvec_i$
      - 2: $VecDot_0$
      - 3: $Axpy_i$
      - $\forall j = 1, 2, 3, \cdots, b_i$: S_i^j - - Jacobi PC

Figure 8.8: Cost Inventory of CG with Full Multigrid Preconditioner

Chapter 9

Linear scalability studies

This chapter presents scalability studies of our solver on an IBM PowerPC cluster with 128 4-way SMPs and a Cray T3E with 512 processors, with problems up to 7.5 million degrees of freedom. We show comparative results on two different platforms, and look into some general and detailed solver performance issues.

9.1 Introduction

This chapter proceeds as follows, notation and solver configuration are introduced in §9.2. We introduce our linear test problem in §9.3, and present scalability studies of our solver on a Cray T3E §9.4 and an IBM PowerPC cluster §9.5. §9.6 shows numerical experiments for our grid agglomeration strategies and analysis of the time spent in each grid of a sample problem. We present "end to end" performance data in §9.7, to provide a view of the performance of our entire parallel finite element package.

Our overall results show that our algorithm is indeed scalable on problems up to 7.5 million degrees of freedom on 512 processors of a Cray T3E with about 50% solver parallel efficiency, and on 128 4-way SMPs nodes of an IBM PowerPC cluster with about 20% solver parallel efficiency.

9.2 Solver configuration and problem definitions

We denote a problem x by Px_k, k being the number of thousands of dof that we put on each processor. A *problem* is a geometry, or domain, with boundary conditions, and a finite element discretization (or element formulation), material properties, etc., for each subdomain - but does not include a mesh of the domain. For instance our first problem the "included sphere" (§5.2.5, §5.3.7, §6.4, §6.5, and §9.3), with a mesh and number of processors that results in about 25,000 dof per processor is referred to as $P1_{25}$.

Our solver uses a block Jacobi preconditioner for a CG smoother for our (full) multigrid preconditioner of a CG solver. Two pre and post smoothing steps are used throughout our numerical experiments, and we use a convergence tolerance of 10^{-6} (i.e., declare convergence of solution $\hat{x}$ when $\frac{||A\hat{x}-b||}{||b||} < 10^{-6}$) - unless otherwise stated.

9.3 Problem $P1$

Figure 9.1 shows one mesh (13,882 vertices), of a finite element model of a hard sphere (Poisson ratio of 0.3) included in a soft somewhat incompressible material (Poisson ratio of 0.49). $P1$ is made of eight vertex hexahedral trilinear "brick" elements and is almost logically regular. All materials are linear, with mixed displacement and pressure elements (Q1P0) [86]. A uniform pressure load is applied on the top surface. One octant is modeled with symmetric ("roller") boundary conditions on the "cut" surfaces, all other surfaces have homogeneous Neumann boundary conditions. The other meshes that we test are of the same physical model but with different scales of discretization.

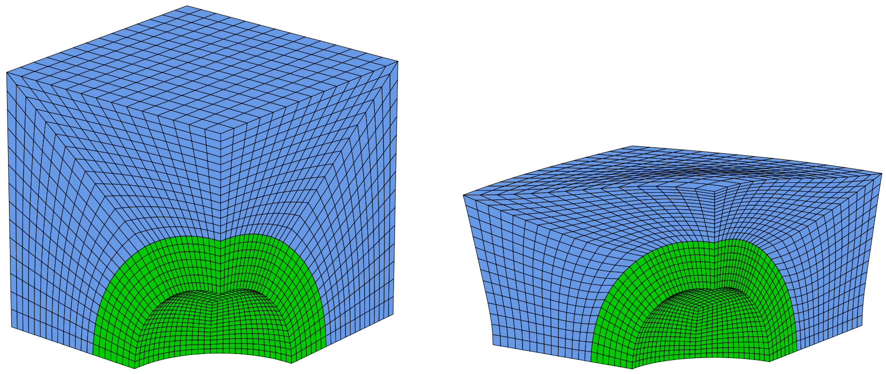


Figure 9.1: 13,882 Vertex 3D FE mesh and deformed shape

9.4 Scalability studies on a Cray T3E - $P1_{15}$

Figure 9.2 shows the times for the primary components (in Figure 8.2) of one linear finite element solution, after the (per configuration) set up phase, for a variety of problems from 15,000 dof to 7.5 million dof with 1 to 512 processors, run on a Cray T3E. The Cray T3E has 512 450 MHz single processor nodes, with 900 Mflop/sec theoretical peak, and 256 Mb memory per processor. For each instantiation of the problem we have chosen the number of processors so as to keep about 15,000 dof per processor, and to be a multiple of four (to be consistent with the IBM data in the next section).

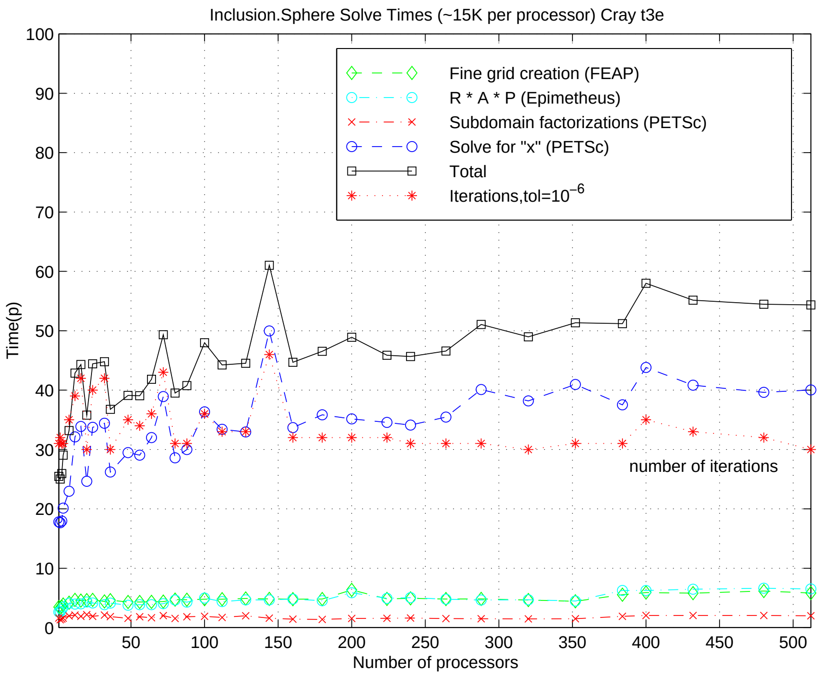


Figure 9.2: 15,000 dof per processor, included sphere times, on a 512 PE Cray T3E

From this data we can make a few observations about the performance characteristics of our solver. First and foremost we see an overall trend of our solver requiring a constant number of iterations (the first solve required 31 iterations and the last required 30). This means that multigrid converges as quickly as expected. The enthusiasm for the success of this algorithm is tempered by seemingly random and indeterminate increases of the iteration count, particularly on the smaller problems (in this data, the 35 different solves have iteration counts between 30 and 46). This is a shortcoming of the current algo-

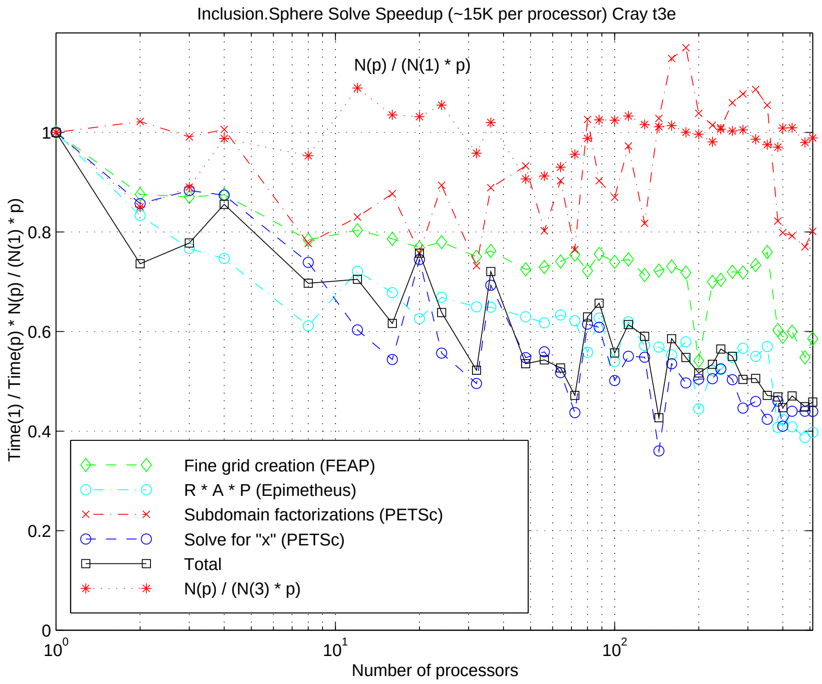


Figure 9.3: 15,000 dof per processor, included sphere efficiency, on a Cray T3E

rithm - although we have found (since this data was collected) that this variability can be significantly damped by *restarting* CG every 20 or 30 iterations.

We see two possible sources of these fluctuations in iteration counts: the non-global coarse grids meshes (§5.4) lead to restriction operators that are not computed with one valid global finite element mesh, and the "random" selection of facets in our face identification algorithm (§5.3.3) is probably not optimal. These issues require a parallel Delaunay mesh generator [12], and that we use a more rational selection of initial facets with some global view of the problems - the design of such an algorithm that is both effective and has acceptable parallel characteristics is a subject of future research.

Figure 9.3 shows the *efficiency* of the data in Figure 9.2; we have also plotted the factor $\frac{N(p)}{N(1)\cdot p}$ by which we multiply the data, to account for the fact that these problems come in fixed sizes which are not in general an integer multiple of the smallest (1 processor) version of the problem. Figures 9.2 and 9.3 show that the formation of the fine grid stiffness matrix ("Fine grid [stiffness matrix] creation") is scaling well - this is to be expected as no communication is required and the only sources of inefficiency are load imbalance, and the redundant work done on elements that straddle our vertex partitioning. Thus, our efficiency (defined in §8.4.1) for the fine grid construction is: communication efficiency $c = 1.0$, scale efficiency $z \approx 1.0$, and work efficiency $w \approx 0.8$ and is the ratio of the number of elements in the problem divided by the number of processors, to the maximum number of element evaluations on any one processor. Load balance l, for the element evaluations is not explicitly enforced with our partitioner, though they are inherently well balanced as all elements do the same amount of work (in this linear example), and the element load balance is reasonably good as the number of elements on a processor is closely related to the number of non-zeros on the processor (which is explicitly optimized by the partitioner). Also, on large problems ParMetis has a tendency to put multiple disconnected small subdomains on a few processors, resulting in good load balance in the matrix vector products, but large "surface areas" on these few processors - these processors evaluate more than the average number of elements per processor leading to larger load imbalance in the fine grid matrix creation.

We can also see that the coarse grid creation (the matrix triple product, RAP) is scaling reasonably well, and is a small part of the overall solve time. The matrix triple product times in Figure 9.2 are hindered by what seems to be non-optimal matrix assembly implementation in PETSc. We have implemented our own assembly "wrappers" for the PETSc assemble routines that use hash tables to cache the accumulation of matrix entries until the floating point work is done in the matrix triple product. We then add our cached values to the PETSc matrix, with the PETSc assemble routines, once for each matrix entry per processor. This optimization is necessary for the off-processor matrix entries as PETSc does not implement this well, but even for the on-processor entries, using our assembly wrapper has more than doubled the performance of the matrix triple product on the IBM (the increase on the Cray T3E is not as dramatic).

To understand the parallel efficiency of the actual solve we remove the scale o efficiency *noise* (number of iterations) from this data to ascertain trends in the solver performance (Mflop rate). Figure 9.4 shows two different decompositions of the solve (time to solve for "x") parallel efficiency, using flop rate and flop count, utilizing our efficiency models, from §8.4.1, to visualize the contributing components. Note, the lower curve of each of these plots represents the efficiency of the solve with a fixed number of iterations. Also the small difference between the "average flop/iteration" and the horizontal line at efficiency = 1.0 (the horizontal axis), in the left plot of Figure 9.4, is the small amount of sub-linear (below the axis) flop efficiency in *this* set of experiments. See §10.4 for a more

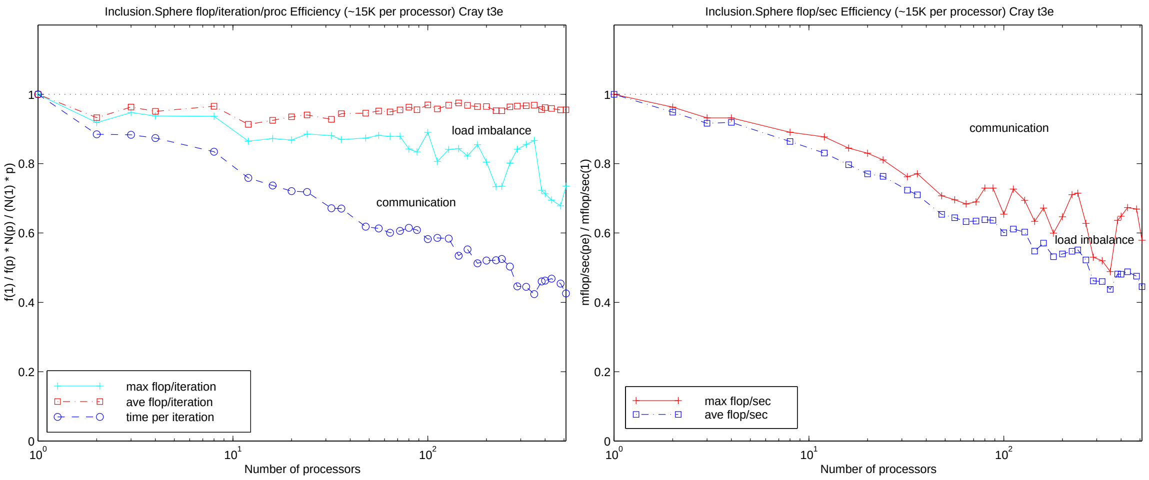


Figure 9.4: 15,000 dof per processor, included sphere, efficiency on a Cray T3E

detailed discussion of the efficiency plots such as those in Figure 9.4. From this data we can see that the parallel efficiency of the T3E is almost 50% for the solve.

The *serial efficiency* of this code can be calculated by dividing the Mflop rate for the entire solve and matrix setup (submatrix factorizations and matrix triple products), by the serial peak flop rate. The Serial Mflop rate for the 15,000 dof problem is 61 Mflop/sec, and the peak flop rate (appendix B) is 662 Mflop/sec to giving a serial efficiency $s = 0.10$.

9.5 Scalability studies on an IBM PowerPC cluster - $P1_{30}$

Performance is a machine dependent quantity, thus we look at this same study, as the last section, on a different machine. Note, to be consistent with $P1_{15}$ on the Cray, we put 60 k dof per 4-way SMP node on the IMB; we, however, use only two processors per node as this gives us better performance on the largest problems (the ones of interest) - resulting in $P1_{30}$. Figures 9.5,9.6,9.7 show the same experiments, as in the previous section, run on an IBM PowerPC cluster at LLNL. Each node has four 332 MHz PowerPC 604e processors, with 512 Mb of memory per node, and a peak Mflop rate of 258 Mflop/sec (appendix B). We only use two processors per node and run in a *flat* MPI programming model (thus we are *not* explicitly taking advantage of the shared address space on each node).

This data shows that the efficiency of the solve is not scaling well as we are only

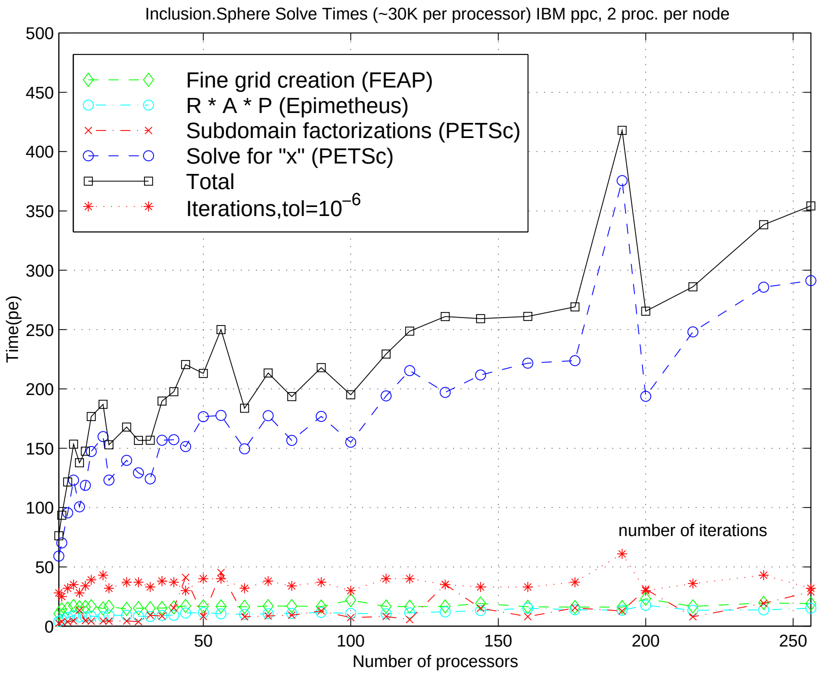


Figure 9.5: 30,000 dof per processor, 2 active processors per node, included sphere times, on a IBM PowerPC cluster

running a bit above 20% efficiency (Figure 9.6) for the largest problem. The subdomain factorizations is running very poorly in some cases - this is most likely due to paging caused by some non-scalable constructs in PETSc, which means we should have run these problems with fewer equations per node (however we wanted to remain consistent with the Cray data on this problem).

To understand the parallel efficiency of the actual solve we remove the scale o efficiency *noise* (number of iterations) from this data to ascertain trends in the solver performance (Mflop rate). Figure 9.7 shows two different decompositions of the solve (time to solve for "x") parallel efficiency, using flop rate and flop count, utilizing our efficiency models to visualize the contributing components from §8.4.1. See §10.4 for a more detailed discussion of the efficiency plots such as those in Figure 9.7.

The serial efficiency of this code can be calculated by dividing the Mflop rate for the entire solve and matrix setup (submatrix factorizations and matrix triple products). The Serial Mflop rate for the 15,000 dof problem is 30 Mflop/sec, peak flop rate (appendix B) is 258 Mflop/sec, to give a serial efficiency $s = 0.12$.

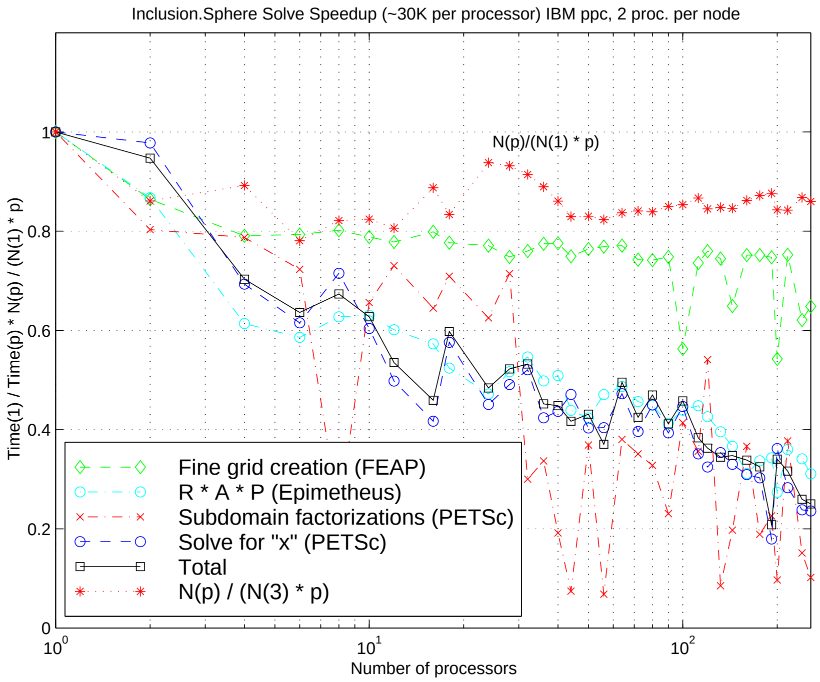


Figure 9.6: 30,000 dof per processor, 2 processors per node, included sphere efficiency, on a IBM PowerPC cluster

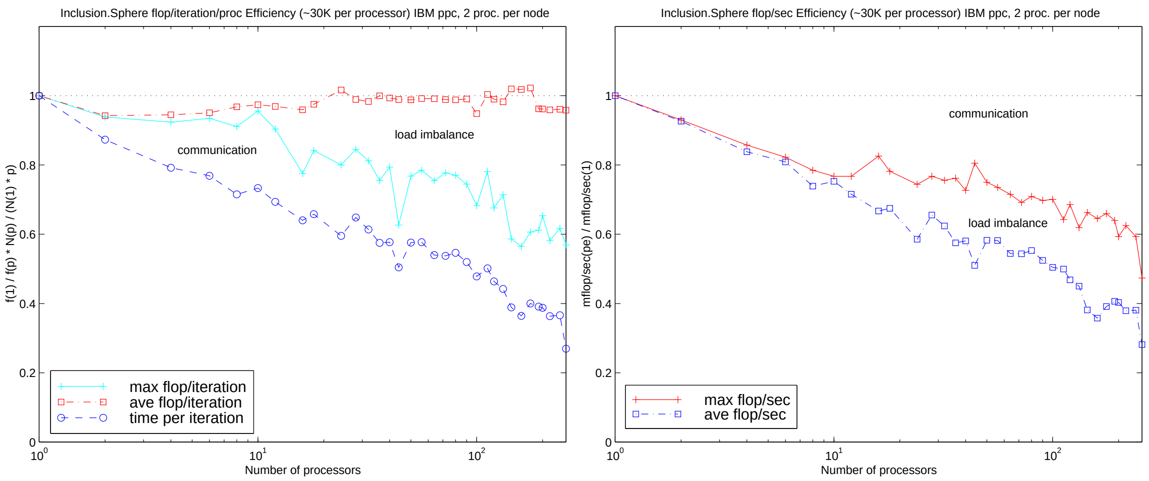


Figure 9.7: 30,000 dof per processor, 2 processors per node, included sphere, efficiency on a IBM PowerPC cluster

9.6 Agglomeration and level performance

This section looks into two related areas, time spent on each multigrid level and the effect of processor agglomeration described in §7.4.

9.6.1 Agglomeration

We use the IBM PowerPC cluster of this experiment as the utility of agglomeration techniques are most pronounced on this machine. We use $P1_{15}$ on 60 nodes (using all 4 processors per node), with and without processor agglomeration. Table 9.6.1 shows approximate flop rates for dot products, and the total flop rate for the actual solve (i.e. the iterations).

		Total Mflop per sec. (MFs) for solve			
3,594 k dof Included Sphere		With agglomeration **1509**		Without agglomeration **249**	
grid	≈ equations	np	dot MFs	np	dot MFs
1	3,594 k	**240**	≈ 35	**240**	≈ 42
2	314k	**120**	≈ 11	**240**	≈ 3
3	45k	**30**	≈ 8	**240**	≈ 0
4	6k	**10**	≈ 2	**240**	≈ 0
5	600	**1**	NA	**1**	NA

Table 9.1: Flat and "graded" processor groups, IBM PowerPC cluster

This data shows that, for this machine, dramatic savings can be achieved with processor group agglomeration.

9.6.2 Performance on different multigrid levels

We provide an approximate measurement of the time spent on each level, for a particular problem on the Cray T3E. We use the 9,594,879 dof version of problem $P2$, that is similar to $P1$, introduced in chapter 10. Table 9.6.2 shows the time spent on each grid, in the accelerator, and the total time for the actual solve - after the preconditioners have be factored and the coarse grids created. Note, we are only able to measure the total solve time accurately - given the PETSc output. We thus approximate the time on each grid by adding the *maximum* time spent in the matrix-vector product and subdomain solves (of the block Jacobi preconditioner), and the *minimum* time spent in the dot products (as the dot products accumulate the load imbalance accounted for in the previous terms). We then throw out the time on the grid that we have the least confidence in (grid 4) and assign it the time required to add up to our (reliable) total solve time.

level	vertices	active processors	$\approx$ time (sec)
Krylov accelerator	3,227,206	512	1.1
1	3,227,206	512	7.68
2	262,909	512	3.62
3	35,286	512	2.86
4	5,309	64	4.04
5	543	8	1.59
6	46	1	0.21
	Total solve time	21.1	21.1

Table 9.2: Time for each grid on Cray T3E, 9.6 million dof problem

Note, this data may indicate that fourth grid is not using an optimal number of processors, though we are not able to improve the overall solve time (the quantity that we can measure accurately) by varying the number of processors, so the source of the apparent inefficiency in this grid is not clear.

9.7 End to end performance

This section shows the total “end to end” performance of our parallel finite element implementation with our solver to do just *one* solve - from beginning to end. We measure the high level components of the total parallel finite element system, as diagramed in Figure 8.2. Note, we do not measure time for FEAP to setup the local data structures for the fine grid, after the partitioning but before the construction of the restriction operators (Prometheus), as this is not optimal now (we do not use memory resident files) but it is small (about 20 seconds on the T3E).

9.7.1 Cray T3E

Figures 9.8 and 9.9 show the times for major components in the solution of one solve on the Cray T3E at NERSC.

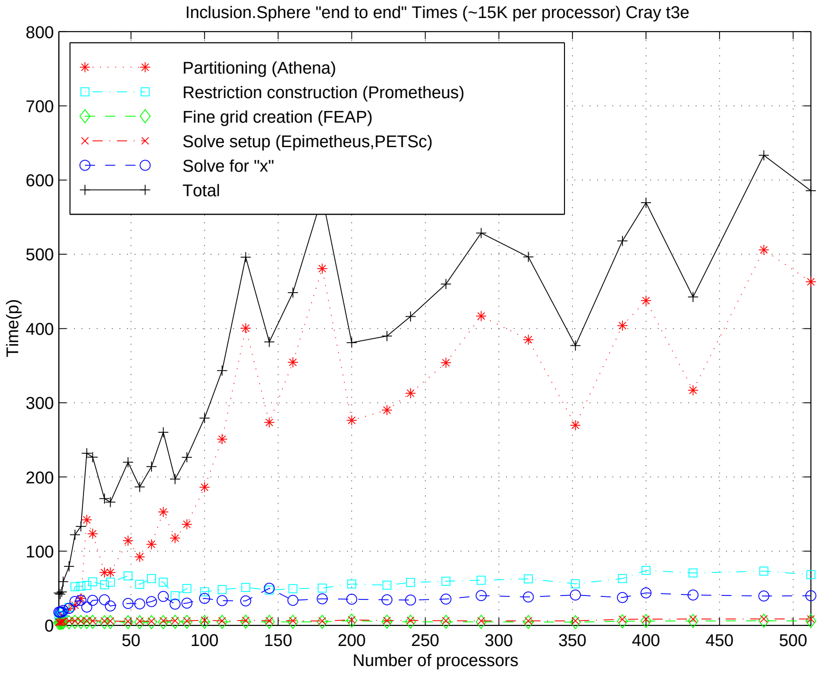


Figure 9.8: 15,000 dof per processor "end to end" times on a Cray T3E

From this data we can see that the parallel finite element codes setup phase and partitioning (Athena), costs about ten times the cost of one solve (of $P1$ solved to a relative residual tolerance of 10^{-6}). Also the restriction operator construction (Prometheus), i.e. coarse grid construction, shape function evaluation, and restriction matrix construction, costs at most twice as much as a solve. The time for the restriction operator construction is hindered by a few PETSc inefficiencies, namely the the assembly routines are probably not optimal (§9.4) and so assembling the restriction operator could probably be improved. The restriction construction is also hindered by inefficient submatrix extraction and symbolic factorization in the additive Schwarz preconditioners. Thus the restriction construction phase probably has opportunity to be further optimized by optimizing some of the general sparse matrix operations in PETSc; though these costs are "per configuration" cost and thus are amortized in nonlinear and transient analyses.

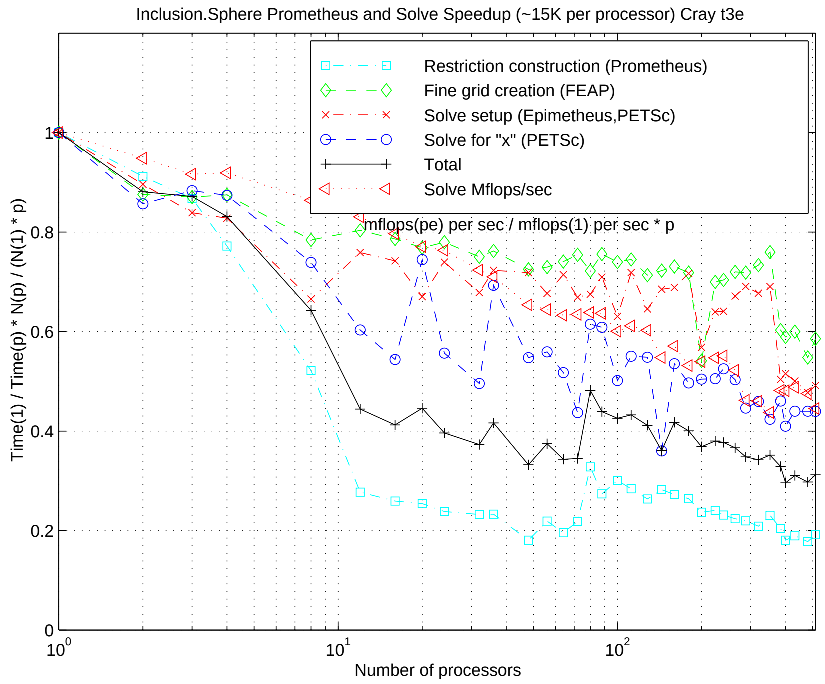


Figure 9.9: 15,000 dof per processor "end to end" times on a Cray T3E

Also the FEAP construction and Epimetheus assembly of the fine grid stiffness matrix is very small relative the solve cost. The solve setup costs includes the Jacobi block factorizations, but is predominately the matrix triple product for the coarse grid operator construction - this cost is about one fifth that of the actual solve (solve for "x"). Finally "End to End" the sum of all of the components. Additionally Figure 9.9 shows the efficiency of the Mflop rate in the solve, which is about 50%.

9.7.2 IBM PowerPC cluster

This section shows the same data as the last section for the IBM PowerPC cluster at LLNL. It is interesting to note that the finite element setup (Athena) is faster on the IBM - though this is somewhat misleading. Each node on the IBM has a local disk, the Cray processors do not have a local disk, so that writing the local FEAP input file, and reading it in (inside of FEAP on each processor) is much more expensive on the Cray. Also the Cray does not support I/O in C very well, requiring that Cray specific routines be called for reading and writing files; this data may not have had the optimal system I/O parameters set, so the Cray Athena data may not be optimal.

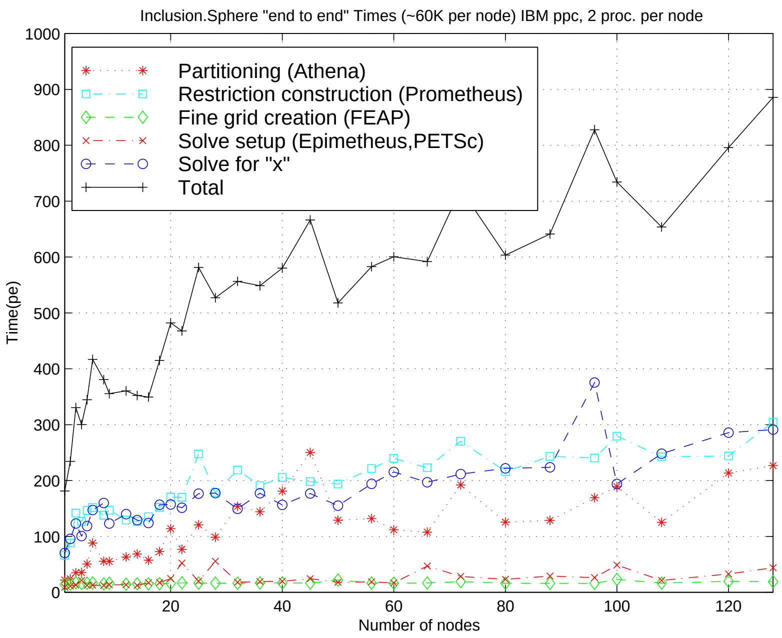


Figure 9.10: 60 k dof per node, 2 proc. per node “end to end” times, IBM PowerPC cluster

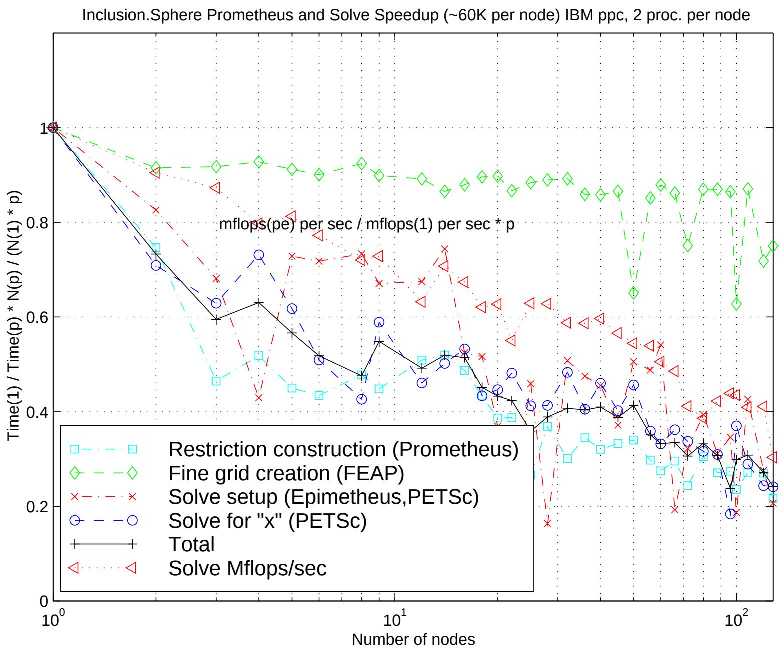


Figure 9.11: 60 k dof per node, 2 proc. per node, solve times, IBM PowerPC cluster

9.7.3 Conclusion

Form this data we can see that the initial parallel finite element setup phase is the biggest bottleneck for *one* solve; but as its cost can be amortized for typical problems, and as it is not the subject of this work, this is not of primary concern. The performance of the restriction construction (Prometheus) may be somewhat diminished by non-optimal assembly of the restriction operators in PETSc - this is partially due the fact that we repartition the coarse grids, requiring more communication than if we did not repartition. Also, though ParMetis minimizes the data movement in repartitioning, it is not clear how effective it is (we are using the second or third alpha release of version 2.0). The restriction construction phase is also hindered by inefficient sparse matrix operations (submatrix extraction and symbolic factorization) in the additive Schwarz preconditioners §9.7. These problems are again only "per configuration" costs and have not been investigated throughly. Additionally we can see the very different performance characteristics of the IBM and the Cray, on unstructured multigrid codes. We attribute this to the poor support of "flat" MPI codes on the IBM.

Chapter 10

Large scale nonlinear results, and indefinite systems

This chapter discusses the application of our solver to non-linear problems that arise in computational plasticity and finite deformation materials, with a study that presents our largest solve to data - a 16.6 million degree of freedom solve with up to 512 processors and about 60% parallel communication efficiency. We also discuss the use of our linear solver (for symmetric positive definite matrices) to constrained problems with Lagrange multipliers - namely contact problems in linear elasticity.

10.1 Introduction

This chapter proceeds as follows, §10.2 describes the large scale test problem $P2$, and the non-linear solution procedure is discuss in §10.3. A scalability study for one linear solve is presented in 10.4, with problems of up to 16.6 million degrees of freedom, with about 60% parallel efficiency on 405 processors of a Cray T3E. The non-linear numerical results are discussed in §10.5 with problems up to 16.6 million degrees of freedom on 512 processors of a Cray T3E and about 60% parallel efficiency on 405 processors of a Cray T3E. We discuss contact problems with serial numerical results in §10.6, and conclude in §10.7.

10.2 Non-linear problem - $P2$

Problem $P2$ is similar to $P1$, from chapter 9, with one important difference - the included sphere has been subdivided into seventeen layers of *alternating* materials. This change to $P1$ has two important effects on the character of the problem. First it introduces "thin body" features which our heuristics (§5.3) are designed to accommodate, and second these heuristics alters the degree to which vertices are coarsened in the sphere but not in the rest of the domain - leading to severe load imbalance if one does not repartition.

We again scale this problem, Figure 10.2 shows the smallest (base) version of $P2$ with 80 k dof. $P2$ has been discretized by using a similar scale of discretization as $P1$'s

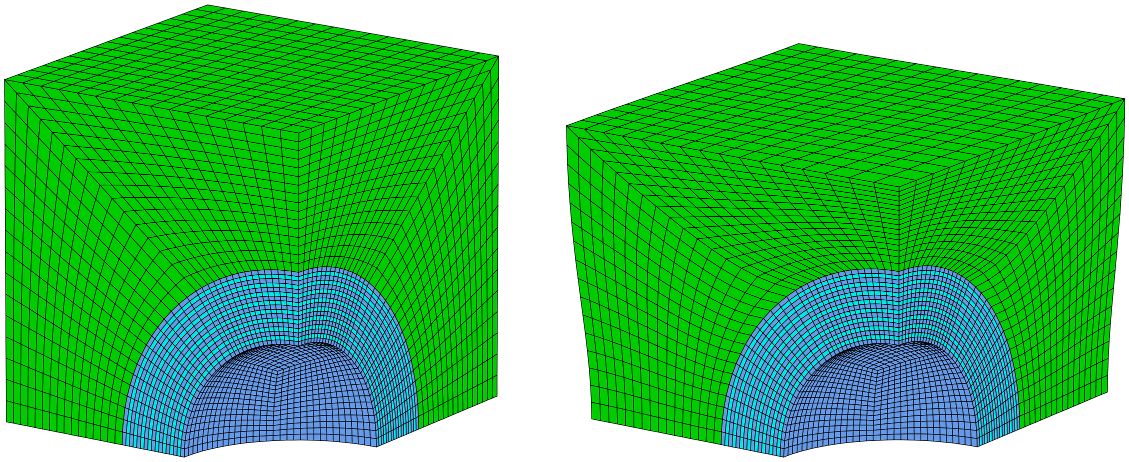


Figure 10.1: 80,000 dof concentric spheres problem

83 k problem, for the base case. Each successive problem has one more layer of elements through each of the seventeen shell layers, with an appropriate (i.e., similar) refinement in the other two directions, and in the outer soft domain - resulting in problems of size: 80 k, 621 k, 2,086 k, 4,924 k, 9,595 k, 16,554 k, and 26,257 k degrees of freedom (the largest problem has not yet been run).

We use similar materials as those in $P1$, but with non-linear constitutive models, and the interior concentric spheres alternate "hard" and "soft" layers with the hard material in the inner and outer shells. The loading and boundary conditions have been changed from $P1$ to an imposed uniform displacement (down), on the top surface.

Table 10.2 shows a summary of the constitution of our two material types.

The hard material is a simple J_2 plasticity material with kinematic hardening [75].

Material	Elastic mod.	Poisson ratio	deformation type	yield stress	hardening mod.
soft	10^{-4}	0.49	large	∞	NA
hard	1	0.3	small	0.002	0.002

Table 10.1: Non-linear materials

The soft material is a large deformation (Neo-Hookean) hypoelastic material [86].

10.3 Non-linear solver

We use a full Newton non-linear solution method. Convergence is declared when the energy norm of the correction, in an iteration, is 10^{-16} that of the first correction. This means in Newton iteration m, we declare convergence when $\left|x_m^T \cdot (b - Ax_m)\right| < 10^{-16} \cdot \left|x_0^T \cdot (b - Ax_0)\right|$. Our linear solver, within each Newton iteration, is conjugate gradient preconditioned by our multigrid solver, with a block Jacobi preconditioned conjugate gradient smoother. We use 6 blocks for every 1,000 unknowns in the block Jacobi preconditioner.

FEAP calls our linear solver at each Newton iteration, with the current residual $r_m = b - Ax_m$, thus the linear solve is for the (negative) increment $\Delta x \approx A^{-1} r_m$. We use a dynamic convergence tolerance ($rtol$) for the linear solve, in each Newton iteration, of $rtol_1 = rtol_2 = 10^{-4}$ in the first and second iteration, and $rtol_m = min(10^{-3}, \frac{\|r_m\|}{\|r_{m-1}\|} \cdot 10^{-1})$ on all subsequent iterations ($m > 2$). This heuristic is intended to minimize the number of total iteration required in the Newton solve in each time step by only solving each linear solve to the degree that it "deserves" to be solved. That is, if the true (non-linear) residual is not converging quickly then solving the linear system to an accuracy far in excess of the reduction in the residual, that is expected in the outer Newton iteration, is not like to be economical.

The reason for hardwiring the tolerance, for the second Newton iteration, is that the residual for the first iteration of this problem tends to drop by about three orders of magnitude. The second step of this problem tends to have the residual reduced by about one order of magnitude or less and then continues with super linear, but not quadratic convergence rate (as we use a non-exact solver). Our dynamic tolerance heuristic ($min(10^{-3}, \frac{\|r_m\|}{\|r_{m-1}\|} \cdot 10^{-1})$) specifies too small of a tolerance, on the second iteration of this problem, so we hardwired the tolerance for the sake of efficiency.

10.4 Cray T3E - large scale linear solves

We run one linear solve with about 41,000 degrees of freedom per processor (i.e. $P2_{41}$) and a convergence tolerance of 10^{-4} (the first linear solve tolerance in the non-linear solver), so as to investigate the efficiency of our largest solves to date ($16.6 \cdot 10^6$ dof). We want to show our solver in its best light by running with as many equations per processor as possible, as parallel efficiency will in general increase as the number of degrees of freedom per processor goes up. Thus, this material is intended to illustrate the issues of scale in isolation of the issue of non-linear performance discussed in the §10.5. Figure 10.2 shows the times for the major subcomponents of the solver, and Figure 10.3 shows the efficiency diagram of the same data.

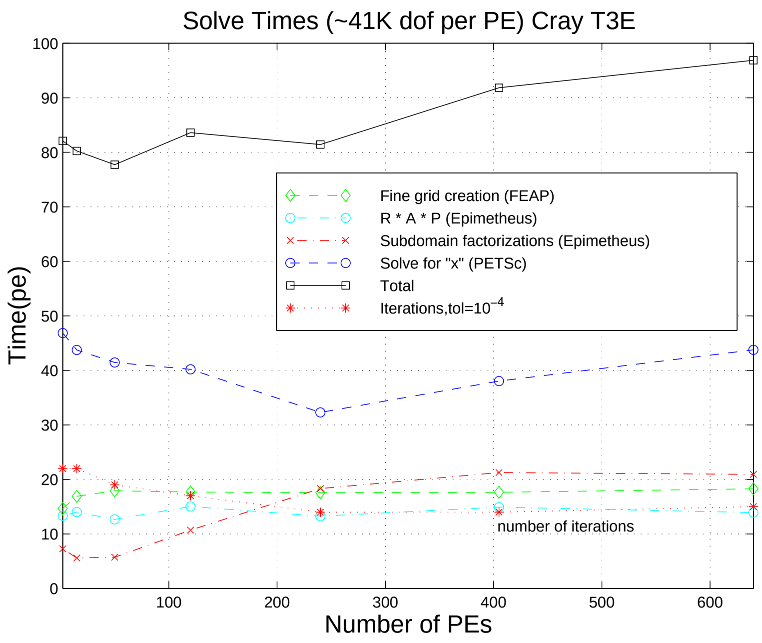


Figure 10.2: 41,000 dof per processor, included concentric sphere times, on a Cray T3E

We see super-linear efficiency in the solve times, in Figures 10.2 and 10.3, for two reasons. First, we have super-linear convergence rates, as shown in Figures 10.2. The decrease in the iteration count in Figure 10.2 shows super-linear scale efficiency $z > 1.0$, in our efficiency measures from §8.4.1.

Second, the vertices added in each successive scale problem have a higher percentage of *interior* vertices than the base (two processor) problem, leading to higher rates

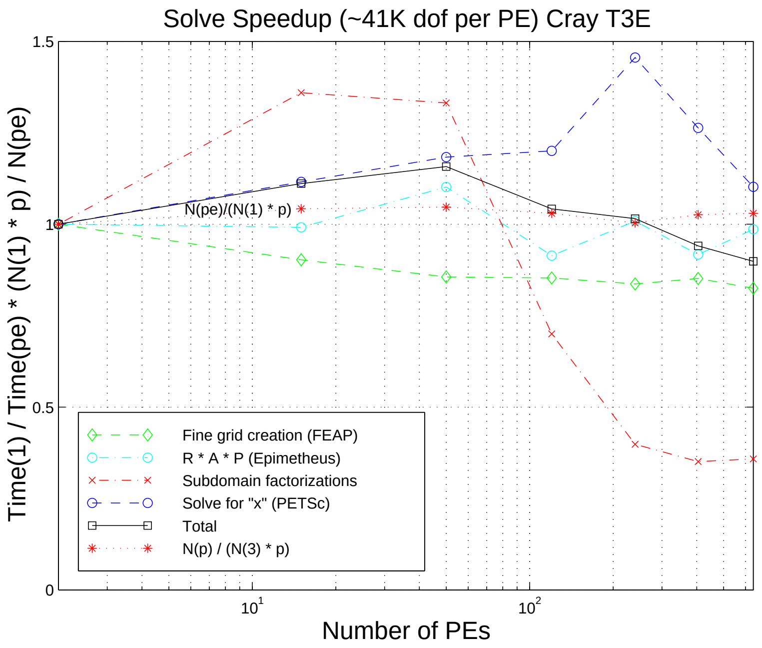


Figure 10.3: 41,000 dof per processor, included concentric sphere efficiency, on a Cray T3E

of vertex reduction in the coarse grids. This is because, as the number of unknowns n increases, the "surface area" increases by $n^{\frac{2}{3}}$ whereas the interior will increase by n; thus, the ratio of interior vertices to surface vertices will increase as the scale of discretization decreases (n increases) on the larger problems. Our heuristics (§5.3.1) try to articulate the surfaces (boundary and material interfaces) well, resulting in a higher ratio of surface vertices being promoted to the coarse grid. Thus, the *rate* of vertex reduction is higher on the larger problems as they have proportionally higher interior vertices, hence proportionally larger reduction rates leading to less work per processor per fine grid vertex on the large problems. For instance, the total reduction factor of the first coarse grid is about seven on the base case (80 k dof problem) and about thirteen on the larger problems. Therefore, there are in fact fewer flops - *per fine grid vertex* - on the coarse grids for the larger problems as can be seen in Figure 10.4. The work efficiency w from §8.4.1, shown in the "average flops per iteration" per processor in Figure 10.4, is super-linear $w > 1.0$.

Figures 10.2 and 10.3 also show that "subdomain factorizations" perform poorly. On inspection of the PETSc output we see that the copying of the subdomain data, from the grid stiffness matrix, and the symbolic factorizations of these submatrices are the cause of these large and increasing (with number of processors) times. We are not able to explain this poor performance in this data, though older data shown in Figure 9.2 on problems of similar scale show much smaller times that are in line with our expectations. Additionally, there is no need for communication in the subdomain factorization phase, hence the 25% parallel efficiency in Figure 10.3 is clearly not endemic to our algorithm or implementation.

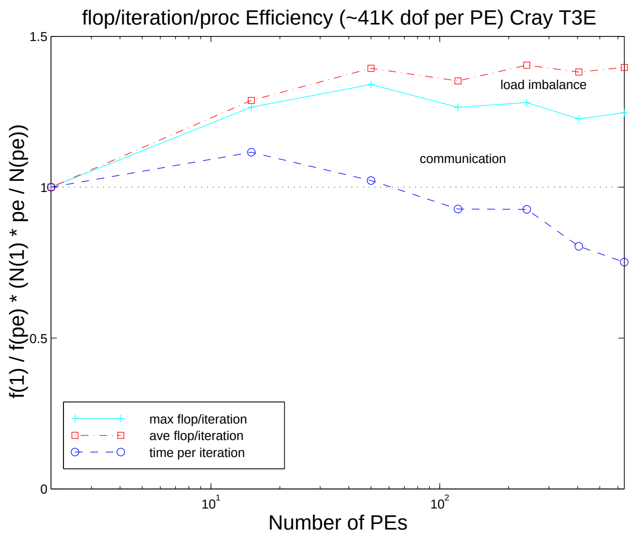


Figure 10.4: 41 k dof per processor, concentric sphere, flop efficiency on a Cray T3E

Figure 10.5 shows that we have above 60% parallel efficiency - in the Mflop rate - up to 405 processors on the Cray T3E, in the solve.

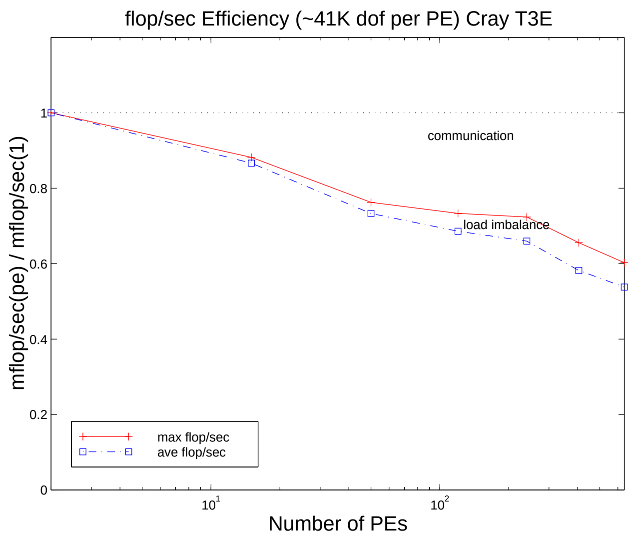


Figure 10.5: 41 k dof per processor, concentric sphere, Mflop/sec efficiency on a Cray T3E

10.5 Cray T3E - non-linear solution

For the non-linear problem $P2_{32}$ we run five time steps that result in a displacement down of 3.6 inches, the cube is 12.5 inches on a side, and the top "soft" section is 5 inches at the central (z) axis. Thus, as the hard interior core does not displace significantly at 3.6 in top displacement we have about 72% compression of the soft material at the interior corner. This percentage decreases away from the interior corner as the depth of the soft material increases as we move away from the top of the sphere. We keep about 32,000 degrees of freedom per processor, and run problems of size 80 k (on 3 processors) up to 16,554 k (on 512 processors). Figure 10.6 show the number of multigrid iterations, stacked on one another to show the total number of multigrid iterations to solve each problem.

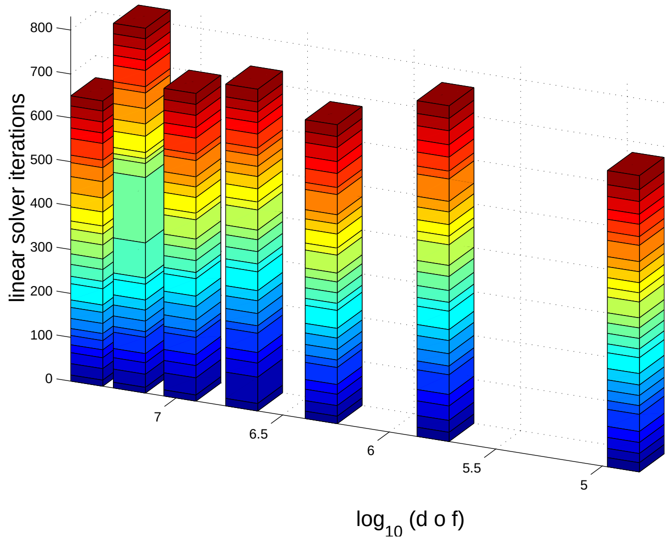


Figure 10.6: Multigrid iterations per Newton iteration

From this data we can see that the total number of iterations is staying about constant as the scale of the problem increases. Figure 10.3 shows that the number of iterations, to reduce the residual by a fixed amount in the first solve of the first time step, *decreases* as the problem size increases. Thus the data in Figure 10.6 suggests that the nonlinear problem is getting *harder* to solve as the discretization is refined; this is not a surprising result as there is likely more nonlinear behavior in the finer discretizations, but more work remains to investigate this issue further.

Figure 10.7 show a histogram of the number of *inner* iterations for each Newton *outer* iteration of each time step.

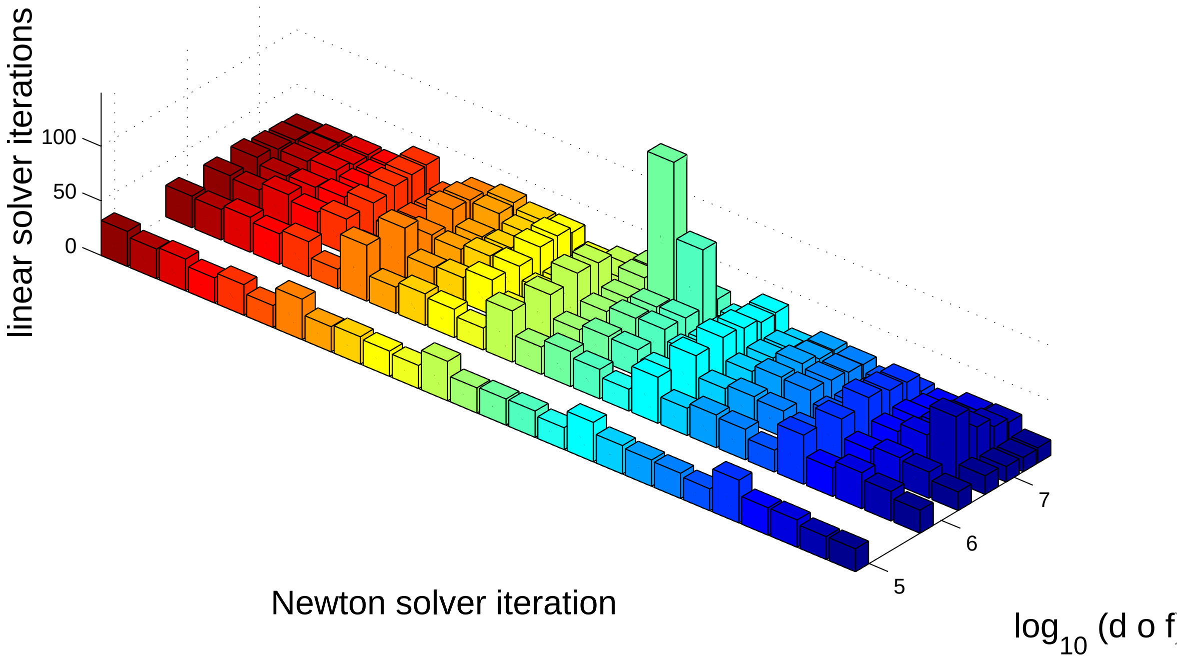


Figure 10.7: Histogram of the number iteration per Newton step in all (5) time steps

Figure 10.8 show the total "wall-clock" times for these problems run on an IBM PowerPC cluster (note, this data was not in the dissertation and solves problems of up to 26.3 million dof, the largest solve also slipped into the figures above).

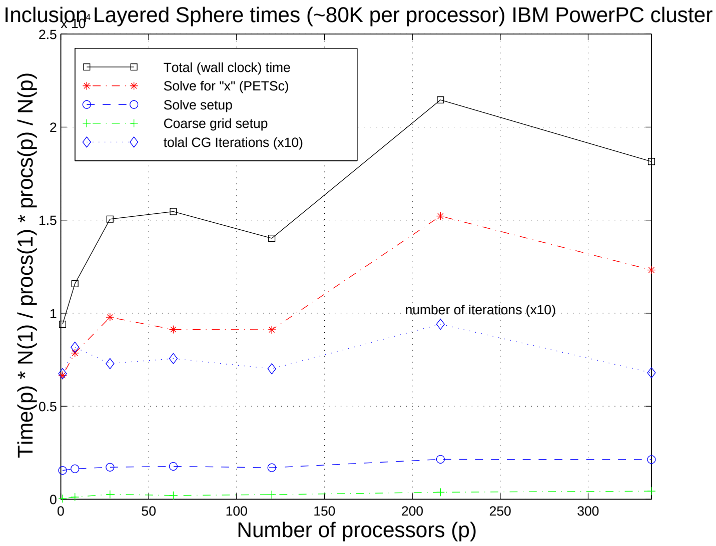


Figure 10.8: End to end times of non-linear solve with 32,000 dof per processor

This data shows that the overall solution times are staying about constant and that the initial partitioning cost are about equal to the cost of the five non-linear solves on the Cray T3E up to 512 processors. Additionally, the base case (3 processor case), solve time (in the actual iterations) ran at 71 Mflops, and the 512 processor case ran at 20,823 Mflops (about 58% efficiency). This (71 Mflops) is the fastest Mflop rate per processor that we have recorded, thus our communication efficiency is 58% ($c = 0.58$) on 512 processors.

10.6 Lagrange multiplier problems

For problems with multiple bodies, or bodies that fold onto themselves, the system of differential equations in continuum mechanics must be augmented with algebraic constraints to prevent penetration between bodies - *contact* forces must be introduced to maintain a physically meaningful solution i.e., satisfy conservation laws. Contact constraints can be implemented in one of two ways. First, "stiff" springs can be added to the system judiciously to *approximate* contact i.e., some "penetration" is permitted and the stiffer the springs the smaller this error - this is known as a *penalty* approach. Second, the system of differential equations may be augmented with an explicit set of algebraic equations, that specify the lengths of prescribed "gaps" - i.e., the normal to a plane of a body through a point on another body, with a specified gap (e.g., 0 for true contact) defines one equation. This specified relative displacement requires that a *force* be applied, in the direction of the "gap", to maintain balance of linear momentum. Force terms are thus added to the existing displacement equations as an applied load, and the magnitude of this load is the value of the *Lagrange multiplier*.

For example consider a 1D problem with a spring (k_1) attached to a rigid wall and a mass, and an applied *constant* load, gives rise to a simple equation for the steady state response, $k_1 x_1 = f_1$. If another mass-spring were added to the system we would get the trivial system of equations

$$\begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

If we specify that the two masses must stay in contact with each other (i.e., $x_1 = x_2$ or $x_1 - x_2 = 0$), we need to augment each equation with the contact force between the masses: $k_1 x_1 = f_1 + \lambda$ and $k_2 x_2 = f_2 - \lambda$ to get

$$\begin{pmatrix} k_1 & 0 & -1 \\ 0 & k_2 & 1 \\ -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \lambda \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ 0 \end{pmatrix}$$

or

$$\begin{pmatrix} A & C^T \\ C & 0 \end{pmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix} \tag{10.1}$$

A is a standard stiffness matrix. Note that we could also assume that $k_1 = 0$ and A would only be positive semi-definite but, in general, the problem is still well posed if $\frac{v^T A v}{v^T v} \neq 0$, $\forall v \in Ker(C), v \neq 0$ [16].

We use Lagrange multiplier formulations to impose contact constraints, thus resulting in an indefinite system of equations. To solve the set of equations we use a simple Uzawa method [5]. The Uzawa algorithm, or augmented Lagrange method, decouples the solution for the displacements x, and the Lagrange multipliers λ. *Outer* iterations, on the Lagrange multipliers, call our multigrid solver which will conduct *inner* iterations, with a modified right hand side, see Figure 10.9.

Our stiffness matrix A can be positive semidefinite. We know A is positive definite over the kernel of C, thus we can *regularize* A to get a symmetric positive definite matrix $\bar{A} = A + \rho C^T DC$. Here ρ is a scalar scaling parameter and D is a scaling matrix. We use a diagonal matrix for D, Note,

- the solution x^* satisfies $Cx^* = 0$, so $Ax^* = \bar{A}x^*$, and we have the correct solution even though we are *not* using the true stiffness matrix to calculate the residual.
- our Lagrange multipliers are formed with a simple (slave) point on (master) surface formulation [57].
- in many texts authors implicitly assume that D is the identity [17].
- penalty methods solves with $\bar{A}$ and pick ρ to be vary large (e.g., 10^6 times larger than the largest diagonal entry).

The vector from the slave point, to the surface (normal to the surface) is called the "gap" vector g; its length is called the gap. We define D, with the gap vector g and the diagonal entries d of the stiffness matrix associated with the "slave" node j in each constraint i, as $D_{ii} = g^T \cdot d$ (an alternative definition of D is $D_{ii} = g^t A_{jj} g$). The difficulty with penalty methods i.e., large ρ, is that the system becomes very "stiff" and difficult to solve with iterative methods. The advantage of augmented Lagrange formulations is that they are more accurate, and the regularized systems ($\bar{A}$) are much better conditioned as a relatively small ρ can be used.

Picking ρ leads to a typical *ad hoc* minimization process as there are no reliable methods that we are aware of for picking ρ *a priori*, for the problems that are of interest to this dissertation. If ρ is too small then the outer iterations will converge slowly, and if ρ is too large then the inner iteration (our solver) will converge slowly. We chose $\rho = 5.0$, as this seems to be about optimal for our test problem. Note, the solution times do not seem to be very sensitive to this parameter. We use a basic Uzawa, with a user provide relative tolerance *rtol* and absolute tolerance *atol*, algorithm to solve for x and λ in equation (10.1) as shown in Figure 10.9

$\lambda_0 = \lambda$ from previous time step

$x = 0$

$i = 1$

while $\left|Ax + C^T\lambda_{i-1} - f\right| > rtol \cdot |f|$ or $|Cx_i| > atol$

solve for x_i: $\bar{A}x_i = f - C^T\lambda_{i-1}$

$\lambda_i \leftarrow \lambda_{i-1} + \rho C x_i$

$i \leftarrow i + 1$

Figure 10.9: Uzawa algorithm

10.6.1 Numerical results

We test our solver with a test problem in linear elasticity. Figure 10.10 shows a problem with three concentric spheres, this is somewhat similar to $P2$ with out the cube cover material, and we call it $P7$. The loading is a thermal load on the middle sphere - i.e., the middle sphere is hotter than the inner and outer sphere, and hence expands. The two interface conditions (between the inner and the middle sphere and the middle and the outer sphere), are enforced with a contact constraint. The problem has 22,092 dof and about 300 equations in C for each of the two contact surfaces; our standard solver configuration is used with 200 blocks of block Jacobi preconditioner for the smoother, four multigrid levels, and a relative tolerance of 10^{-12} or 10^{-6} on the residual. These solves have two separate influences

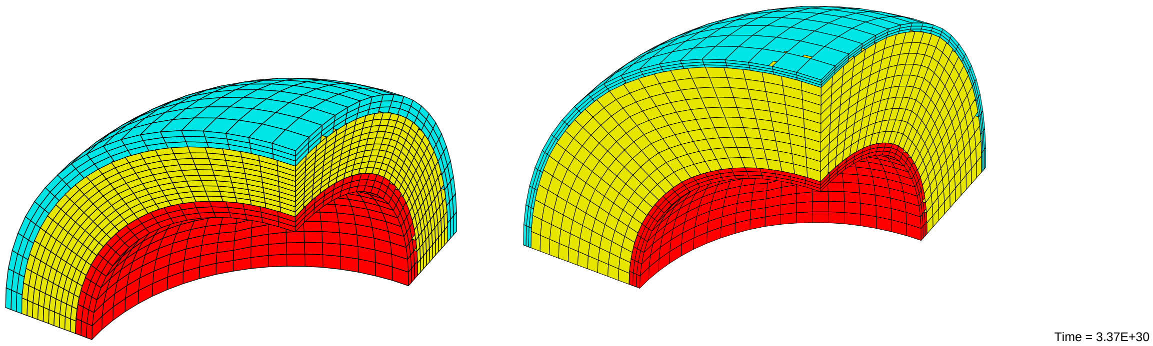


Figure 10.10: 22,092 dof concentric spheres with contact, undeformed and deformed shape

on their performance, the number of outer iterations and the cost of each iteration. We provide a "control" case - that is a similar problems with no contact. Figure 10.11 shows the concentric sphere problem without contact, both the undeformed and deformed shape.

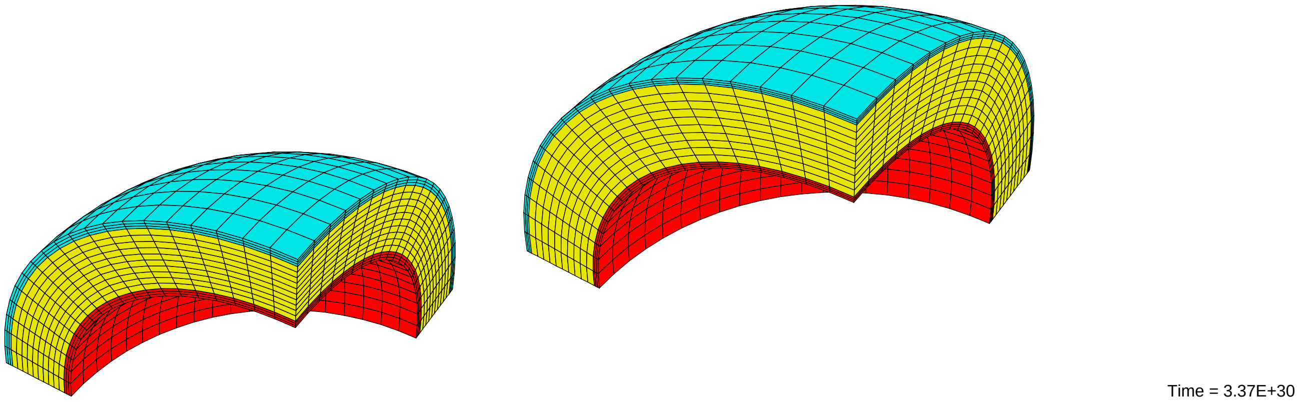


Figure 10.11: 15,810 dof concentric spheres without contact, undeformed and deformed shape

Table 10.2 shows the results of this experiment. The *average* convergence rate for each Uzawa iteration is about the same as that of the no contact problem. As we use an adaptive convergence tolerance in the inner iterations ($rtol^{m_i}_{inner} = |Cx^m_i|$), the regularized problem is in fact a bit more difficult to solve than our control problem.

problem	inner iterations (total)		outer iterations	
rtol	10^{-12}	10^{-6}	10^{-12}	10^{-6}
no contact	27	12	1	1
contact	313	85	14	7

Table 10.2: Multigrid preconditioned CG iteration counts for contact problem

From this data we can see that the cost is growing at about the logarithm of the convergence tolerance, with about twice as many outer iterations and about twice as many inner iterations, per outer iteration. This is probably not optimal and may be the subject of future research. A potential avenue to improve this algorithm is to use preconditioned Uzawa methods [33].

10.7 Conclusion

We conclude that our linear solver can be effectively used in solving non-linear problems, and that it remains robust for problems with thin body features and up to 16.6 million equations on up to 512 processors of a Cray T3E, with about 60% parallel efficiency. Our non-linear problems also present some the the most challenging operators, namely with geometric stiffness, which lowers the lowest eigenvalues (hence increases the condition number of the matrix), and "softening" materials from yielding plasticity materials resulting in areas of strain localization with highly incompressible constitution. Thus, we are able to show that our solver remains robust on large scale problems, in the presents very challenging materials. Also we show that our solver has the potential to be useful is solving contact problems with Lagrange multipliers.

Chapter 11

Conclusion

This dissertation has developed a promising method for solving the linear set of equations arising from implicit finite element applications in solid mechanics. Our approach, a 3D and parallel extension to an existing serial 2D algorithm, is to our knowledge unique in that it is a fully automatic (i.e. the user need only provide the fine grid, which is easily available in most finite element codes) standard geometric multigrid method for unstructured finite element problems.

We have developed heuristics for the automatic parallel construction of the coarse grids for 3D problems in solid mechanics; this work represents some of the most original contributions of this dissertation. Our method is the most scalable and robust (i.e., with respect to convergence rate *and* breadth of problems on which it is effective) multigrid algorithm on unstructured finite element problems that we are aware of. Additionally we have developed and analyzed a new parallel maximal independent set algorithm that has superior PRAM complexity on finite element meshes, than the commonly used random algorithms, and is very practical.

We have developed a fully parallel and portable prototype solver that shows promising results, both in terms of convergence rates and parallel efficiency, for some modestly complex geometries with challenging materials in large deformation elasticity and plasticity. Our prototype has solved problems of up to 16.6 million degrees of freedom on problems with large deformation elasticity and plasticity on a Cray T3E with up to 512 processors, and on an IBM PowerPC cluster with up to 128 4-way SMP nodes processors; we have also run on a network of workstations [68] and a network of PC SMPs [67]. We have developed a complexity theory, and have begun to develop a complexity model, of multigrid equation solvers for 3D finite element problems on parallel computers. Additionally we have developed agglomeration strategies for the optimal selection of active processor sets on the coarse grids of multigrid, as this is essential for optimal scalability.

We have also developed a highly parallel finite element implementation, built on an existing state-of-the-art serial research "legacy" finite element implementation that uses our parallel solver. By necessity we have developed a novel, domain decomposition based parallel finite element programming model, that builds a *complete* finite element problem on each processor as its primary abstraction.

The implementation of our prototype system (ParFeap) required about 30,000 lines of our own C++ code, plus several large packages: PETSc (160,000 lines of C), FEAP (105,000 lines of FORTRAN), METIS/ParMetis (30,000 lines of C), and geometric predicates (4,000 lines of C) [73]. This complexity was required because

- efficient parallel multigrid solvers for unstructured meshes are inherently complex,
- algorithm development and verification of success in this area is highly *experimentally* based, and thus requires a flexible full featured computational substrate to enable this type of research, and
- we require portable implementations, so we use explicit message passing (MPI), as our parallel programming paradigm, while accommodating both "flat" and hierarchical memory architectures (clusters of SMPs, CLUMPs).

Additionally this work is rather broad, in the sense that the emphasis has been on getting a prototype of the best finite element linear equation solver possible - this has limited the amount of time that could be devoted to investigating more optimized approaches to each aspect of our algorithm. In fact the vary exploratory nature of this work *demands* a non-optimal implementation, in that there is no sense in optimizing any system (e.g., a parallel linear solver for unstructured finite element problems) before the overall practical quality of the algorithm and a particular application area have been determined. Thus, there is much more work to be done.

11.1 Future Work

1. To be useful to a more general finite element community we need to extend the algorithm and its features:

- Investigate more sophisticated face identification algorithms, to increase robustness of solver on arbitrary complex domains.
- Incorporate a parallel Delaunay tessellation algorithm [12] so as to develop more robust and globally consistent implementations.
- Incorporate a parallel direct solver for the coarsest grid [58].
- Extend the implementation for more element types: shells, beams, trusses, etc.
- Accommodate higher order elements such as, supporting higher dof per vertex for p-adaptive methods and multi-physics problems.

2. Develop solution strategies and implementations to extend application domains.

- Investigate highly nonlinear problems to evaluate solver characteristics on problems in plasticity, large deformation elasticity as well as other areas as one approaches the limit load, so as to develop strategies to effectively solve these highly non-linear finite element problems.
- Investigate non-CG Krylov subspace methods for indefinite systems from large deformation elasticity and plasticity.
- Develop parallel and preconditioned Uzawa solvers for indefinite systems from constrained problems with Lagrange multipliers.
- Investigate multigrid algorithms for differential and algebraic systems from constrained problems with Lagrange multipliers.

3. Develop PETSc to more fully support large scale "algebraic" multigrid applications efficiently.

- Incorporate a general RAP (sparse matrix triple product) in PETSc (we have implemented a specialized RAP, requiring a copy of the R matrix within Epimetheus).
- Implement faster off-processor, and on-processor, matrix assembly routines.
- Improve the efficiency of the additive Schwarz preconditioner setup phase.
- Add "left" and "right" communicators to restriction matrices to fully support coarse grid agglomeration.
- Optimize numerical kernels for shared memory architectures.

4. Rebuild and cleanup the prototype's infrastructure for distributed multilevel unstructured grid classes.

   - Redesign the parallel distributed multi-level classes.
   - Redesign the data structures, and clean the code up.
   - Separate the distributed multilevel unstructured grid classes from Prometheus so as to provide a class library to the public.

5. Add and test other *promising* "algebraic" multigrid algorithms e.g.

   - Agglomeration with rigid body modes: Bulgakov and Kuhn (1995) §4.2 [19].
   - Smoothed agglomeration: Vanek and Mandel (1995) §4.2 [83].

6. Build, maintain, document, and support a Library interface.

   - Encapsulate Prometheus into a PETSc "PC" object (low level).
   - Develop a higher level interface (e.g., FEI [37]).

Bibliography

[1] Mark Adams. Heuristics for the automatic construction of coarse grids in multigrid solvers for finite element matrices. Technical Report UCB//CSD-98-994, University of California, Berkeley, 1998.

[2] Mark Adams. A parallel maximal independent set algorithm. In *Proceedings 5th copper mountain conference on iterative methods*, 1998.

[3] Anderson, Bai, Bischof, Demmel, Dongarra, DuCroz, Greenbaum, Hammarling, McKenney, Ostrouchov, and Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[4] J.H Argyris and S. Kelsey. Energy theorems and structural analysis. *Aircraft engineering*, 1955.

[5] K. Arrow, Hurwicz L., and Uzawa H. *Studies in nonlinear programming*. Stanford University Press, 1958.

[6] ASCI. http://www.llnl.gov/asci.

[7] ASCI Red machine at Sandia National Laboratory. http://www.sandia.gov/ASCI/Red.

[8] S. B. Baden. Software infrastructure for non-uniform scientific computations on parallel processors. *Applied Computing Review, ACM*, 4(1):7–10, Spring 1996.

[9] T. Baker. Mesh generation for the computation of flow fields over complex aerodynamic shapes. *Computers Math. Applic.*, 24(5/6), 1992.

[10] S. Balay, W.D. Gropp, L. C. McInnes, and B.F. Smith. PETSc 2.0 users manual. Technical report, Argonne National Laboratory, 1996.

[11] Marshall Bern and David Eppstein. Mesh generation and optimal triangulation. In *Computing in Euclidean Geometry, World Scientific Publishing*, 1992.

[12] Guy Blelloch, Gary L Miller, and Dafna Talmor. Design and implementation of a practical parallel Delaunay algorithm. *To appear in Algorithmica*, 1998.

[13] A. Brandt. Multi-level adaptive solutions to boundary value problems. *Math. Comput.*, 31:333–390, 1977.

[14] D. Brelaz. New method to color the vertices of a graph. *Comm ACM*, 22:251–256, 1979.

[15] Eric A. Brewer. High-level optimization via automated statistical modeling. *PPoPP*, 1995.

[16] Franco Brezzi and Klaus-Jurgen Bathe. A discourse on the stability conditions for mixed finite element formulations. *Computer methods in applied mechanics and engineering*, 82:27–57, 1990.

[17] Franco Brezzi and Michel Fortin. *Mixed and Hybrid Finite Element Methods*. Springer-Verlag, 1991.

[18] W. Briggs. *A Multigrid Tutorial*. SIAM, 1987.

[19] V.E. Bulgakov and G. Kuhn. High-performance multilevel iterative aggregation solver for large finite-element structural analysis problems. *International Journal for Numerical Methods in Engineering*, 38, 1995.

[20] Edward K. Buratynski. A fully automatic three-dimensional mesh generator for complex geometries. *International Journal for Numerical Methods in Engineering*, 30:931–952, 1990.

[21] James C. Cavendish. Automatic triangulation of arbitrary planar domains for the finite element method. *International Journal for Numerical Methods in Engineering*, 8, 1974.

[22] T.F. Chan and Y. Saad. Multigrid algorithms on the hypercube multiprocessor. *IEEE Trans. Comput.*, C-35:969–977, 1986.

[23] Tony F. Chan and Barry F. Smith. Multigrid and domain decomposition on unstructured grids. In David F. Keyes and Jinchao Xu, editors, *Seventh Annual International Conference on Domain Decomposition Methods in Scientific and Engineering Computing*. AMS, 1995. A revised version of this paper has appeared in ETNA, 2:171-182, December 1994.

[24] Philippe G Ciarlet. *The finite element method for elliptic problems*. North-Holland Pub. Co., 1978.

[25] R. Courant. Variational methods for the solution of problems of equalibrium and vibration. *Bulletin of the American Mathematical Society*, 49:1–23, 1943.

[26] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *PPOPP*, May 1993.

[27] James Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[28] J.E. Dendy, M.P. Ida, and Rutledge J.M. A semicoarsening multigrid algorithm for SIMD machines. *SIAM J. Sci. Stat. Comput.*, 13:1460–1469, 1992.

[29] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, Knoxville, 1998.

[30] M.C. Dracopoulos and M.A Crisfield. Coarse/fine mesh preconditioners for the iterative solution of finite element problems. *International Journal for Numerical Methods in Engineering*, 38:3297–3313, 1995.

[31] Maksymillian Dryja, Barry F. Smith, and Olof B. Widlund. Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions. *SIAM J. Numer. Anal.*, 31(6):1662–1694, December 1994.

[32] Laura C. Dutto, Wagdi G. Habashi, and Michel Fortin. An algebraic multilevel parallelizable preconditioner for large scale CFD problems. *Comput. Methods Appl. Mech. Engrg.*, 149:303–3018, 1997.

[33] Howard C. Elman and Gene H. Golub. Inexact and preconditioned Uzawa algorithms for saddle point problems. *SAIM J. Numer Anal.*, 31(6):1645–1661, 1994.

[34] Charbel Farhat, Jan Mandel, and Francois-Xavier Roux. Optimal convergence properties of the FETI domain decomposition method. *Computer Methods in Applied Mechanics and Engineering*, 115:365–385, 1994.

[35] Charbel Farhat and Francois-Xavier Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32:1205–1227, 1991.

[36] FEAP. http://www.ce.berkeley.edu/∼ rlt.

[37] FEI. http://z.ca.sandia.gov/fei.

[38] D. A. Field. Implementing Watson's algorithm in three dimensions. In *Proc. Second Ann. ACM Symp. Comp. Geom.*, 1986.

[39] J. Fish, V. Belsky, and S. Gomma. Unstructured multigrid method for shells. *International Journal for Numerical Methods in Engineering*, 39:1181–1197, 1996.

[40] J. Fish, M. Pandheeradi, and V. Belsky. An efficient multilevel solution scheme for large scale non-linear systems. *International Journal for Numerical Methods in Engineering*, 38:1597–1610, 1995.

[41] S. Fortune and J. Wyllie. Parallelism in random access machines. In *ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[42] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. Technical Report CSL-94-13, Xerox Palo Alto Research Center, 1994.

[43] Gene H. Golub and Charles Van Loan. *Matrix Computations*. John Hopkins University Press, 1983.

[44] Herve Guillard. Node-nested multi-grid with Delaunay coarsening. Technical Report 1898, Institute National de Recherche en Informatique et en Automatique, 1993.

[45] O. Hassan, K. Morgan, E.J. Probert, and J. Peraire. Unstructured tetrahedral mesh generation for three-dimensional viscous flows. *International Journal for Numerical Methods in Engineering*, 39, 1996.

[46] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*, 1993. Formerly, Technical Report SAND93-1301 (1993).

[47] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Stat. Comput.*, 16(2):452–469, 1995.

[48] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Natl. Bur. Stand.*, 49:409–436, 1954.

[49] Ellis Horowititz and Sartaj Sahni. *Fundamentals of computer algorithms.* Galgotia Publications, 1988.

[50] Mark T. Jones and Paul E. Plassman. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[51] S. Kacau and I. D. Parsons. A parallel multigrid method for history-dependent elastoplacticity computations. *Computer methods in applied mechanics and engineering*, 108, 1993.

[52] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, To appear.

[53] George Karypis and Kumar Vipin. Parallel multilevel k-way partitioning scheme for irregular graphs. *Supercomputing*, 1996.

[54] B.W. Kernigham and S Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.

[55] D. E. Keyes. Aerodynamic applications of Newton-Krylov-Schwarz solvers. In M. Deshpande, S. Desai, and R. Narasimha, editors, *Proceedings of the 14th International Conference on Numerical Methods in Fluid Dynamics*, pages 1–20, Springer,New York, 1995.

[56] David Kincaid and Ward Cheney. *Numerical Analysis.* Brooks/Cole Publishing Company, 1991.

[57] N. Kirkuchi and J. T. Oden. *Contact Problems in Elasticity.* SIAM, 1988.

[58] Xiaoye S. Li, J. Demmel, and J. Gilbert. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *To appear in SIAM J. Matrix Anal. Appl.*, 1998.

[59] R. Lohner. Parallel unstructured grid generation. *Comp. Meth. App. Mech. Eng.*, 1992.

[60] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 4:1036–1053, 1986.

[61] David G. Luenberger. *Introduction to Linear and Nonlinear Programming*. Addison-Wesley, 1973.

[62] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, California, November 1997.

[63] J. Mandel. Balancing domain decomposition. *Comm. Numer. Meth. Eng.*, 9:233–241, 1993.

[64] Dimitri J. Mavriplis. Directional agglomeration multigrid techniques for high-Reynolds number viscous flows. Technical Report 98-7, Institute for Computer Applications in Science and Engineering Mail Stop 403, NASA Langley Research Center Hampton, VA 23681-0001, January 1998.

[65] Dimitri J. Mavriplis. Multigrid strategies viscous flow solvers on anisotropic unstructured meshes. Technical Report 98-6, Institute for Computer Applications in Science and Engineering Mail Stop 403, NASA Langley Research Center Hampton, VA 23681-0001, January 1998.

[66] MGNet. http://www.mgnet.org.

[67] Millennium project. http://www.millennium.berkeley.edu.

[68] NOW project. http://now.cs.berkeley.edu.

[69] D.R.J. Owen, Y.T. Feng, and D Peric. A multi-grid enhanced GMRES algorithm for elasto-plastic problems. *International Journal for Numerical Methods in Engineering*, 42:1441–1462, 1998.

[70] J.S. Przemieniecki. Matrix structural analysis of substructures. *Am. Inst. Aero. Astro. J.*, 1:138,147, 1963.

[71] J. Ruge. AMG for problems of elasticity. *Applied Mathematics and Computation*, 23:293–309, 1986.

[72] Y. Saad and H Schultz. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[73] Jonathan Richard Shewchuk. Adaptive precision floating point arithmetic and fast robust geometric predicates. Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996. Submitted to Discrete and Computational Geometry.

[74] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1997. Available as Technical Report CMU-CS-97-137.

[75] J.C. Simo and T.J.R. Hughes. *Computational Inelasticity.* Springer-Verlag, 1998.

[76] J.C. Simo, Taylor R.L., and Pister K.S. Variational and projection methods for the volume constraint in finite deformation elasto-plasticity. *Computer Methods in Applied Mechanics and Engineering*, 51:177–208, 1985.

[77] Barry Smith. A parallel implementation of an iterative substructuring algorithm in three dimensions. *SIAM J. Sci. Comput.*, 13(1):406–423, 1993.

[78] Barry Smith, Petter Bjorstad, and William Gropp. *Domain Decomposition.* Cambridge University Press, 1996.

[79] Ken Stanley. *Execution Time of Symmetric Eigensolvers.* PhD thesis, University of California at Berkeley, 1997.

[80] Made Suarjana and Kincho Law. A robust incomplete factorization based on value and space constraints. *Methods in Engineering*, 38:1703–1719, 1995.

[81] Dafna Talmor. *Well spaced points for numerical methods.* PhD thesis, Carnegie Mellon University, Pittsburgh PA 15213-3891, 1997.

[82] Sivan Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[83] P. Vanek, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. In *7th Copper Mountain Conference on Multigrid Methods*, 1995.

[84] David E. Womble and Brenton C. Young. A model and implementation of multigrid for massively parallel computers. Technical Report SAND89-2781J, Sandia National Laboratory, 1989.

[85] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing, Stanford, California*, February 1998.

[86] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method*, volume 1. McGraw-Hill, London, Fourth edition, 1989.

Appendix A

Test problems

This section lists all of our test problems here with some statistics about them. This is meant to provide a reference with more detail about the problems and to provide a single location for reference to the test problems.

Materials

All materials are mixed pressure-displacement, eight node trilinear finite elements elements.

Material name	elastic modulus	Poisson ratio	yield stress	hardening modulus	deformation type
hard-linear	1	0.3	∞	NA	small
soft-linear	10^{-4}	0.49	∞	NA	small
hard-nonlinear	1	0.3	0.002	0.002	large
soft-nonlinear	10^{-4}	0.49	∞	NA	large

Table A.1: Materials for test problems

Problem 1: Included sphere

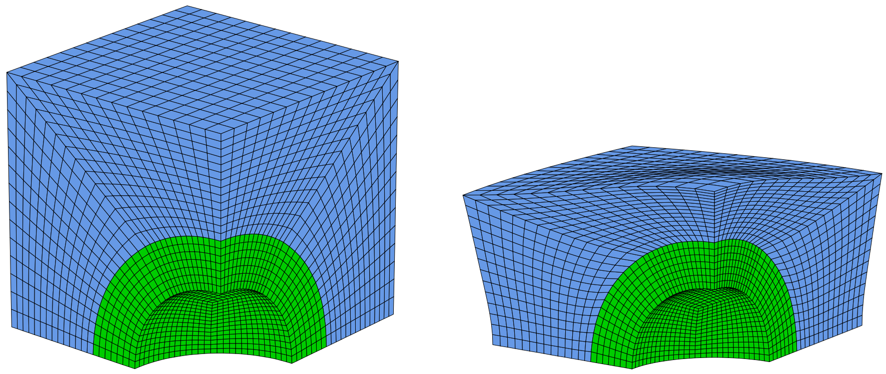


Figure A.1: 13,882 Vertex 3D FE mesh and deformed shape

Materials: *hard-linear* sphere, *soft-linear* cover.

Boundary conditions: Symmetry on sides, uniform load on top.

dof	$\approx$ condition number	non-zeros (x9)
3410	$1.75 \cdot 10^6$	
14880	$3.5 \cdot 10^6$	128386
25308	$7.2 \cdot 10^6$	219835
39732		346816
58800		515161
113460		999271
194472		1718821
246480		2181466
307020		2720467
376740		3341656
456288		4050865
546312		4853926
647460		5756671
760380		6764932
885720		7884541
1024128		9121330
1176252		10481131
1342740		11969776
1524240		13593097
1721400		15356926
1934868		17267095
2167221		19329436
2413320		21549781
2679600		23933962
2964780		26487811
3269508		29217160
3594432		32127841
3940200		35225686
4307460		38516527
4696860		42006196
5109048		45700525
5544672		49605346
7534488		67446190

Table A.2: Problem P1 statistics

Problem 2: Included layered sphere

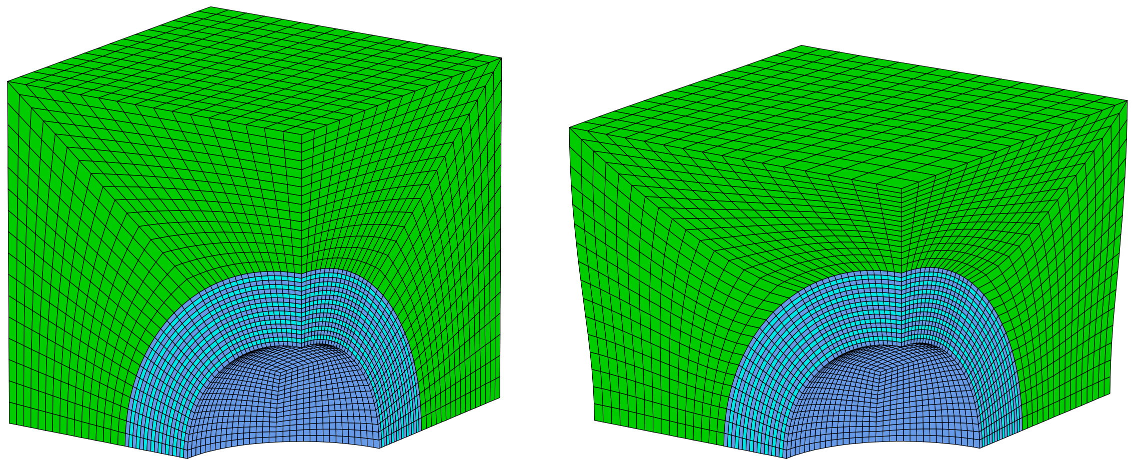


Figure A.2: 80,000 dof concentric spheres problem

Materials: Nine *hard-nonlinear* layers and eight *soft-nonlinear* layers in spheres, *soft-nonlinear* cover.

Boundary conditions: Symmetry on sides, uniform imposed displacement on top.

dof	≈ condition number	non-zeros (x9)
79679		705693
622815		5559456
2085599		18667011
4924223		44134086
9594879		86066409
16553759		
26257055		

Table A.3: Problem P2 statistics

Problem 3: Cantilever beam

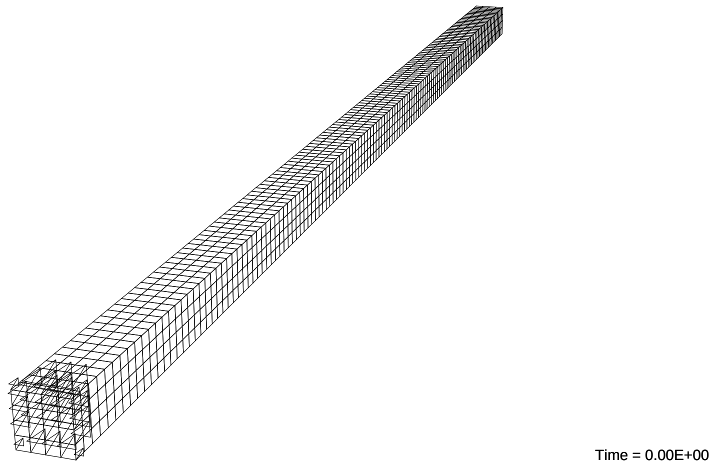


Figure A.3: Cantilever with uniform mesh and end load, $4 \times 4 \times 128$ element mesh, $N = 4$

Materials: *hard-linear*.

Boundary conditions: One end fixed, uniform, off-axis load on the other end.

dof	$\approx$ condition number	non-zeros (x9)
1,728	$2.9 \cdot 10^7$	
9,600	$1.2 \cdot 10^8$	
62,208	$4.3 \cdot 10^8$	

Table A.4: Problem P3 statistics

Problem 4: Cone

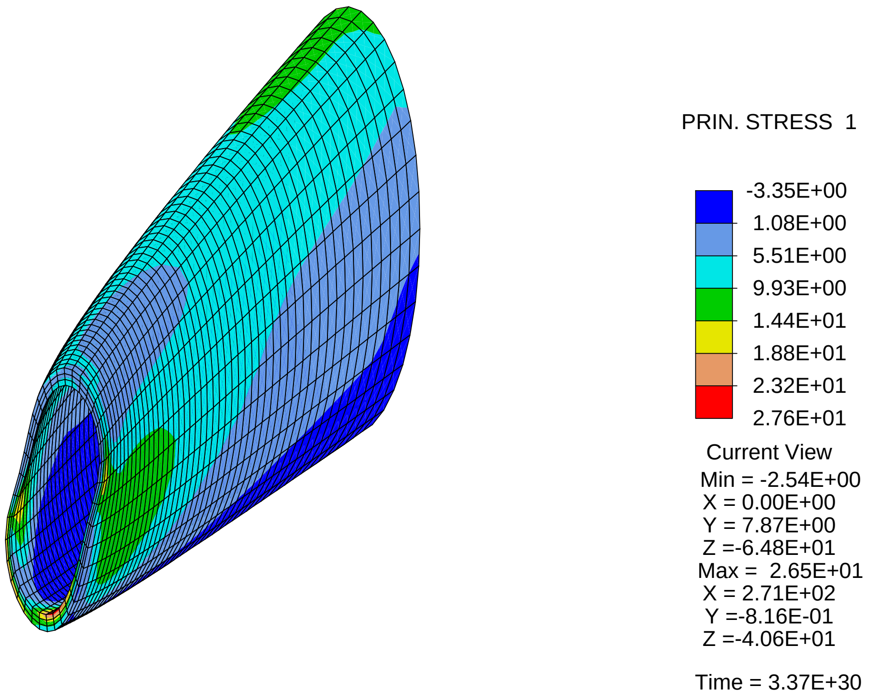


Figure A.4: Truncated hollow cone

Materials: *hard-linear.*

Boundary conditions: One end fixed, off-axis load, with a torque, on the other end.

dof	≈ condition number	non-zeros (x9)
21,700	$3.6 \cdot 10^7$	

Table A.5: Problem P4 statistics

Problem 5: Tube

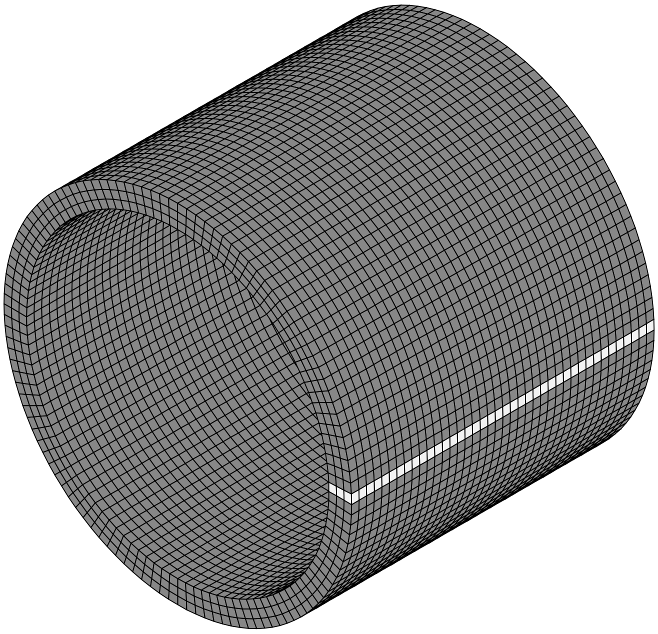


Figure A.5: Cantilevered tube

Materials: *hard-linear.*

Boundary conditions: One end fixed, uniform, off-axis load on the other end.

dof	$\approx$ condition number	non-zeros (x9)
57,600	$1.8 \cdot 10^5$	428,880

Table A.6: Problem P5 statistics

Problem 6: Beam-column

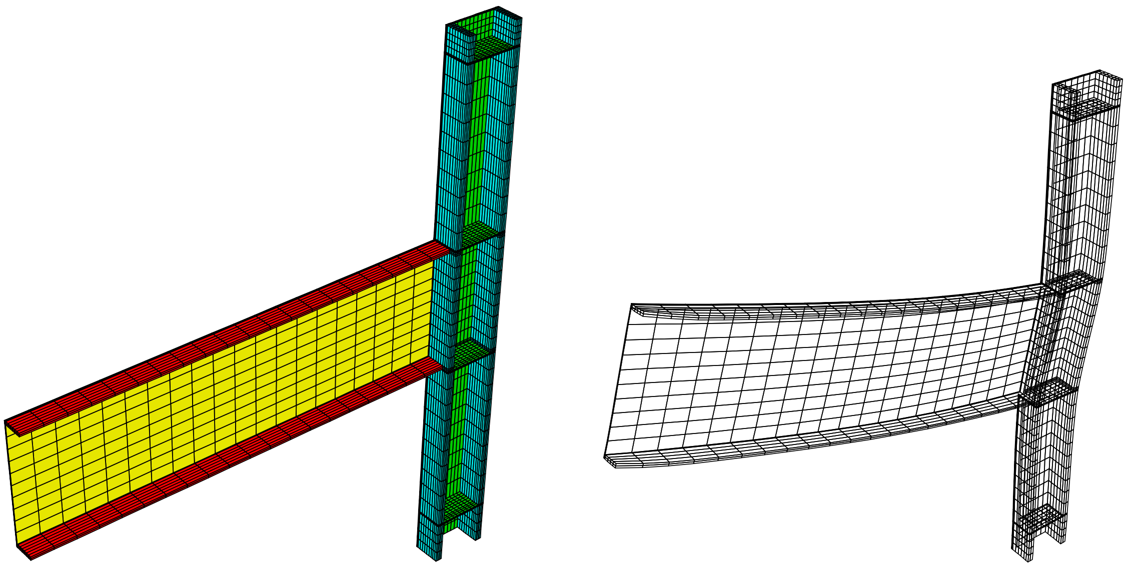


Figure A.6: Beam-column

Materials: *hard-linear*.

Boundary conditions: Top and bottom of column fixed and uniform load down on end of beam.

dof	$\approx$ condition number	non-zeros (x9)
34,460	$1.0 \cdot 10^8$	268,813

Table A.7: Problem P6 statistics

Problem 7: Concentric spheres without contact

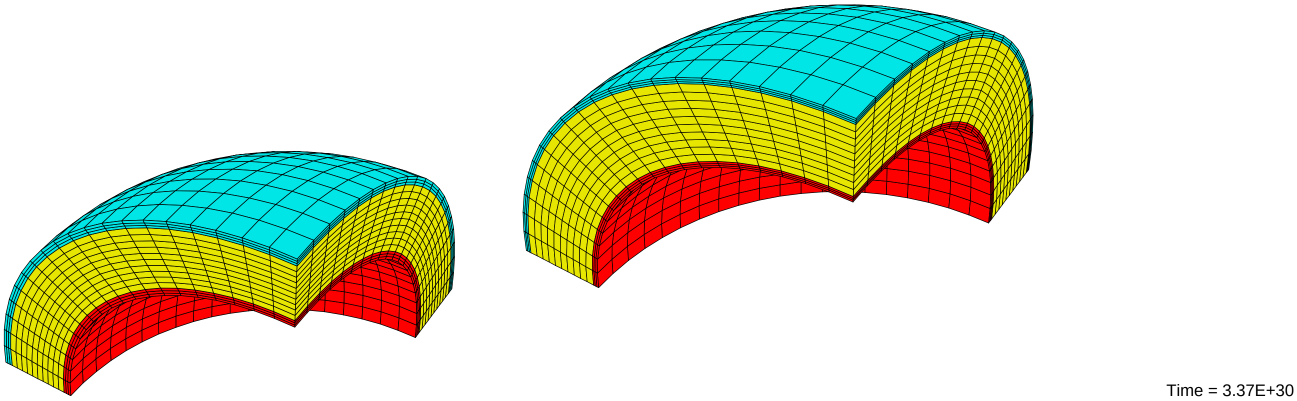


Figure A.7: Concentric spheres without contact

Materials: ALE3D matrix, steel on inner and outer shell, aluminum middle section.

Boundary conditions: Sphere symmetry boundary conditions, with thermal load on middle section.

dof	$\approx$ condition number	non-zeros (x9)
15,810		

Table A.8: Problem P7 statistics

Problem 8: Concentric spheres with contact

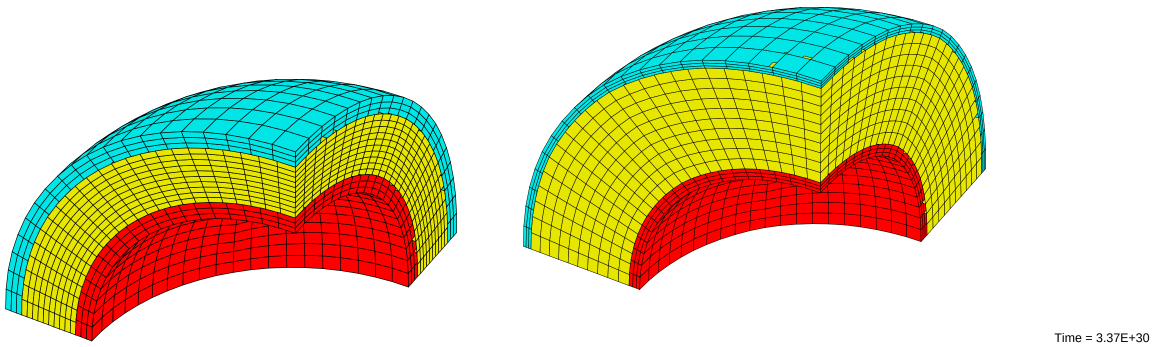


Figure A.8: Concentric spheres with contact

Materials: ALE3D matrix, steel on inner and outer shell, aluminum middle section.

Boundary conditions: Sphere symmetry boundary conditions, with thermal load on middle section. Contact between shells enforces with Lagrange multipliers.

dof	≈ condition number	non-zeros (x9)
22,092		

Table A.9: Problem P8 statistics

Appendix B

Machines

640 processor Cray T3E, at NERSC

The Cray T3E at NERSC has 696 single processor nodes total (up to 692 processors have been used for a single job), 450 MHz., 900 Mflop/sec theoretical peak, 256 MB memory per processor, and a peak Mflop rate of 662 Mflop/sec (1/2 of 2 processor Linpack R_{max}).

150 Node/ 600 processor PowerPC cluster, at LLNL

The IBM at LLNL, has about 150 4-way-SMPs available to users, 332 MHz PowerPC 604e processors, 664 Mflop/sec theoretical peak, 512 MB of memory per node, and a peak Mflop rate of 258 Mflop/sec (1/2 of 2 processor Linpack "toward perfect parallelism" as reported at http://www.rs6000.ibm.com/hardware/largescale/index.html).