# Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing

by

Francis Thomas McKenna

B.A. (Trinity College Dublin, Ireland) 1990
B.A.I. (Trinity College Dublin, Ireland) 1990
M.Sc. (Trinity College Dublin, Ireland) 1990

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 Gregory L. Fenves, Chair
Professor Stephen A. Mahin
Professor James W. Demmel

Fall 1997

The dissertation of Francis Thomas McKenna is approved:

______________________________________________
Chair                                                                                     Date

______________________________________________
Date

______________________________________________
Date

University of California, Berkeley

Fall 1997

# Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing

Copyright © Fall 1997

by

Francis Thomas McKenna

# Abstract

## Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing

by

Francis Thomas McKenna

Doctor of Philosophy in

Engineering—Civil Engineering

University of California, Berkeley

Professor Gregory L. Fenves, Chair

This dissertation presents a new design for object-oriented finite element software. The design provides for a variety of structural analysis methods to be performed on finite element models in both sequential and parallel computing environments.

In the first part of the dissertation a new design is presented for finite element analysis. In a traditional object-oriented design, an analyst constructs a single object, the analysis object, to perform the analysis. In the new design, the analysis class is broken into separate component classes: Integrator, ConstraintHandler, DOF_Numberer, AnalysisAlgorithm, AnalysisModel, FE_Element, DOF_Group, SystemOfEqn, and Solver. Using this design, an analyst creates an analysis procedure by providing objects of the component classes to the analysis object's constructor; the type of objects depending on the type of analysis the analyst wishes to perform. This approach offers great flexibility, because the analysis can be varied by changing the types of objects passed to the constructor, and extensibility, because the types of analysis procedures that can be performed is greatly increased by the introduction of a new component subclass. The functionality of the classes used in the new design are presented for an incremental displacement solution of the equilibrium equations.

Extensions to the design are then presented for modal analysis and modal transient analysis.

The design is evaluated by comparing an implementation using the object-oriented language C++ with a procedural program. The results show that the number of CPU cycles excluding disk i/o required by both programs are similar but that the object-oriented program uses more of the virtual address space.

The second part of the dissertation focuses on extending the design to allow non-overlapping domain decomposition methods to be used during the analysis. Two new classes, GraphPartitioner and DomainPartitioner, are introduced for partitioning the domain. New subclasses are introduced for the domain decomposition, including Subdomain, PartitionedDomain, DomainDecompAnalysis, DomainDecompAlgorithm and DomainSolver.

The third part of the dissertation focuses on parallel finite element solution procedures. New classes are introduced for parallel programming based on the actor programming model: Actor, Channel, Message, MovableObject, MachineBroker, ObjectBroker, and Shadow. The Shadow class is particularly important to the design, as it allows parallel processing to be introduced in a transparent manner. Shadow also permit non-computationally demanding methods, invoked on an object residing in a remote process, to be performed locally. Two subclasses, ShadowSubdomain and ActorSubdomain, are introduced for use in parallel finite element analysis.

A number of analyses using the substructuring domain-decomposition method are performed on two networks of workstations. The results show that the time taken to perform the analysis in parallel using multiple workstations can be faster than that required to perform the analysis using a single workstation, for certain examples the parallel program is over twenty times faster than the sequential program. This is due to the reduction in memory requirements on the workstations and the fact that large portions of the computation can be done concurrently.

The fourth part of the dissertation focuses on obtaining better performance on a parallel machine when performing finite element analysis by using dynamic load balancing. A new approach to dynamic load balancing for finite element analysis is presented in which elements migrate between subdomains to reduce the number of

wasted CPU cycles on the parallel machine. A new class, LoadBalancer, is introduced and existing classes are modified to allow for this new approach. Results are presented which show that the time taken to perform the analysis in a parallel environment can be reduced when using this approach to dynamic load balancing, the time being more than halved for certain examples.

---

Professor Gregory L. Fenves
Dissertation Committee Chair

**To my parents**

# Contents

**List of Figures** viii

**List of Tables** xii

**Acknowledgements** xiv

**1 Introduction** 1
- 1.1 Problem Statement . . . 1
- 1.2 Object Oriented Programming . . . 3
- 1.3 Objective of the Research . . . 4
- 1.4 Overview of Dissertation . . . 5

**2 Fundamental Object-Oriented Finite Element Design** 7
- 2.1 Introduction . . . 8
- 2.2 Modeling Classes . . . 8
- 2.3 Finite Element Model Classes . . . 10
  - 2.3.1 Node Class . . . 12
  - 2.3.2 Element Class . . . 14
  - 2.3.3 Constraint Classes . . . 17
  - 2.3.4 Load Classes . . . 18
  - 2.3.5 Domain Class . . . 22
- 2.4 Analysis Classes . . . 25
- 2.5 Numerical Classes . . . 25
  - 2.5.1 Matrix and Vector Classes . . . 26
  - 2.5.2 Tensor Classes . . . 27
  - 2.5.3 Linear System of Equation Classes . . . 27
- 2.6 Summary . . . 27

**3 Object-Oriented Analysis Algorithms** 29
- 3.1 Introduction . . . 30
- 3.2 The Incremental Solution of Nonlinear Finite Element Equations . . . 31
- 3.3 Existing Object-Oriented Approaches for Finite Element Analysis . . . 34
- 3.4 A New Object-Oriented Approach for the Analysis Algorithm . . . 43
  - 3.4.1 Analysis Class . . . 44

3.4.2 SolutionAlgorithm Class . . . 50
3.4.3 Integrator Class . . . 52
3.4.4 AnalysisModel Class . . . 58
3.4.5 DOF_Group Class . . . 61
3.4.6 FE_Element Class . . . 63
3.4.7 ConstraintHandler Class . . . 65
3.4.8 DOF_Numberer Class . . . 66
3.4.9 SystemOfEqn and Solver Classes . . . 67
3.5 Example Programs . . . 71
3.5.1 Flexibility . . . 71
3.5.2 Extensibility . . . 73
3.6 Extension of Framework to Other Types of Analysis Procedures . . . 76
3.6.1 Extensions for Eigenvalue Analysis . . . 76
3.6.2 Extensions for Modal Transient Analysis . . . 82

**4 Object-Oriented Domain Decomposition 88**
4.1 Introduction . . . 89
4.2 Domain Decomposition Methods for Finite Element Analysis . . . 89
4.2.1 Non-overlapping Domain Decomposition Methods . . . 90
4.2.2 Domain Partitioning . . . 95
4.3 Existing Object-Oriented Approaches to Domain Decomposition . . . 97
4.4 A New Object-Oriented Approach to Domain Decomposition . . . 100
4.4.1 PartitionedDomain Class . . . 101
4.4.2 DomainPartitioner Class . . . 103
4.4.3 GraphPartitioner Class . . . 103
4.4.4 Subdomain Class . . . 104
4.4.5 DomainDecompAnalysis Class . . . 107
4.4.6 DomainDecompAlgo Class . . . 109
4.4.7 DomainSolver Class . . . 109
4.5 Modifications to Classes for Domain Decomposition . . . 112
4.6 Example Programs using Domain Decomposition . . . 114

**5 Parallel Object-Oriented Finite Element Programming 116**
5.1 Introduction . . . 117
5.2 Summary of Parallel Computing . . . 119
5.2.1 Parallel Architectures . . . 119
5.2.2 Parallel Programming Models . . . 121
5.2.3 Parallel Programming . . . 123
5.2.4 Parallel Object-Oriented Computing . . . 126

5.3 Existing Approaches to Parallelizing the Finite Element Method . . . 129

5.3.1 The use of Domain Decomposition Methods in Parallel Finite Element Analysis . . . 130

5.3.2 Existing Approaches to the Parallel Solution of Linear System Of Equations . . . 131

5.3.3 Existing use of Parallel Object-Oriented Programming in Finite Element Analysis . . . 133

5.4 A Parallel Object-Oriented Programming Model for the Finite Element Method . . . 133

5.5 A Framework for Parallel Object-Oriented Finite Element Analysis . . . 137

5.5.1 Shadow Class . . . 139

5.5.2 Actor Class . . . 143

5.5.3 Channel Class . . . 146

5.5.4 Address Class . . . 147

5.5.5 Message Class . . . 149

5.5.6 MovableObject Class . . . 150

5.5.7 MachineBroker Class . . . 151

5.5.8 ObjectBroker Class . . . 152

5.6 Modification of Classes for Parallelism . . . 155

5.6.1 Modification to Class Interfaces . . . 155

5.6.2 Modification to Class Methods . . . 155

5.7 Example Parallel Programs . . . 159

**6 Example Structural Analysis and Performance Evaluation 161**

6.1 Introduction . . . 162

6.2 Example Structural Models . . . 163

6.3 Evaluation of the Object-Oriented Design on Uniprocessor Machines . 165

6.3.1 Introduction . . . 165

6.3.2 Procedural Program . . . 166

6.3.3 Results . . . 167

6.4 Evaluation of the Object-Oriented Design on Parallel Machines . . . 173

6.4.1 Introduction . . . 173

6.4.2 Results . . . 175

6.5 Summary . . . 183

**7 Dynamic Load Balancing for Finite Element Analysis 184**

7.1 Introduction . . . 185

7.2 Existing Approaches to Dynamic Load Balancing the Finite Element Analysis . . . 187

7.3 New Approaches for Dynamic Load Balancing . . . 188

7.4 Extension of Framework for Dynamic Load Balancing . . . . . 189
7.4.1 Modification to the PartitionedDomain Class . . . . . 189
7.4.2 Extension to the Subdomain Class . . . . . 189
7.4.3 Extension to the DomainPartitioner Class . . . . . 189
7.4.4 LoadBalancer Class . . . . . 191
7.5 Evaluation of the Effect of Dynamic Load Balancing on Performance . . . . . 192
7.5.1 Introduction . . . . . 192
7.5.2 Results . . . . . 194
7.6 Summary . . . . . 197

**8 Conclusions and Future Directions** **198**
8.1 Summary . . . . . 198
8.2 Future Directions . . . . . 201

**Bibliography** **203**

**A Matrix, Vector and ID Classes** **216**
A.1 Matrix Class . . . . . 216
A.2 Vector Class . . . . . 218
A.3 ID Class . . . . . 218

**B Detailed Performance Measurements** **221**
B.1 Sequential Performance . . . . . 221
B.2 Parallel Performance . . . . . 221

# List of Figures


2.1 Interface of the **ModelBuilder** Class . . . . . . . . . . 10
2.2 Class Diagram for the Finite Element Method . . . . . . . . . . 11
2.3 Interface for the **Node** Class . . . . . . . . . . 13
2.4 Interface for the **Element** Class . . . . . . . . . . 16
2.5 Interface for the **SP_ Constraint** Class . . . . . . . . . . 18
2.6 Interface for the **MP_Constraint** Class . . . . . . . . . . 19
2.7 Interface for the **LoadCase** Class . . . . . . . . . . 20
2.8 Interface for the **NodalLoad** Class . . . . . . . . . . 21
2.9 Interface for the **ElementalLoad** Class . . . . . . . . . . 21
2.10 Interface for the **Domain** Class . . . . . . . . . . 24

3.1 Class Diagram for Existing Analysis Frameworks . . . . . . . . . . 36
3.2 Class Diagram for Algorithmic Hierarchy in Miller and Rucki (1995) 41
3.3 Class Diagram for New Analysis Framework . . . . . . . . . . 45
3.4 Interface for the **Analysis** Class . . . . . . . . . . 46
3.5 Class Diagram for a Static Analysis . . . . . . . . . . 46
3.6 Pseudo-Code for Selected Methods for the **StaticAnalysis** Class . . . 48
3.7 Class Diagram for a Transient Analysis using a DirectIntegration Scheme . . . . . . . . . . 49
3.8 Interface for the **DirectIntegrationAnalysis** Class . . . . . . . . . . 49
3.9 Pseudo-Code for the **DirectIntegrationAnalysis** Classes Interface and analyze Method . . . . . . . . . . 50
3.10 Interface for the **SolutionAlgorithm** Class . . . . . . . . . . 50
3.11 Interface for the **EquiSolnAlgo** Class . . . . . . . . . . 51
3.12 Pseudo-Code for the **Linear** Classes solveCurrentStep Method . . . . . . 52
3.13 Pseudo-Code for the **NewtonRaphson** Classes solveCurrentStep Method 53
3.14 Interface for the **IncrementalIntegrator** Class . . . . . . . . . . 54
3.15 Pseudo-Code for Selected Methods for the **IncrementalIntegrator** Class . . . . . . . . . . 55
3.16 Interface for the **StaticIntegrator** Class . . . . . . . . . . 56
3.17 Pseudo-Code for Selected Methods for the **StaticIntegrator** Class . 57
3.18 Interface for the **DirectTransientIntegrator** Class . . . . . . . . . . 57

3.19 Pseudo-Code for the Methods of the **DirectTransientIntegrator** Class . . . 58
3.20 Interface and Pseudo-Code for Selected Methods of the **Newmark** Class . . . 59
3.21 Interface for the **AnalysisModel** Class . . . 60
3.22 Interface for the **DOF_Group** Class . . . 62
3.23 Interface for the **FE_Element** Class . . . 64
3.24 Interface for the **ConstraintHandler** Class . . . 66
3.25 Interface for the **DOF_Numberer** Class . . . 67
3.26 Interface for the **LinearSOE** Class . . . 69
3.27 Interface for the **LinearSolver** Class . . . 70
3.28 Pseudo-Code for the **DirectBandSPDLinSOESolver** classes solve Method . . . 71
3.29 An Element By Element Solvers Interface and solve Method . . . 72
3.30 Class Diagram for Eigenvalue Analysis . . . 77
3.31 Interface for the **EigenvalueAnalysis** Class . . . 78
3.32 Pseudo-Code for Selected Methods for the **EigenvalueAnalysis** Class 79
3.33 Pseudo-Code for the **Buckling** and **Frequency** Classes solveCurrentStep Method . . . 79
3.34 Interface for the **EigenvalueIntegrator** Class . . . 80
3.35 Pseudo-Code for Selected Methods of the **EigenvalueIntegrator** Class . . . 81
3.36 Interface for the **EigenvalueSOE** Class . . . 81
3.37 Class Diagram for Modal Transient Analysis . . . 83
3.38 Interface for the **ModalTransientAnalysis** Class . . . 84
3.39 Pseudo-Code for the **ModalTransientAnalysis** Classes analyze Method 84
3.40 Pseudo-Code for the **ModalTransientIntegrator** Classes update Method . . . 84
3.41 Pseudo-Code for the **LinearModal** Classes solveCurrentStep Method 85
3.42 Pseudo-Code for the **NewtonRaphsonModal** Classes solveCurrentStep Method . . . 86

4.1 Domain split into Two Subdomains . . . 90
4.2 Class Diagram for Existing Domain Decomposition Frameworks . . . 97
4.3 Class Diagram for New Domain Decomposition Framework . . . 101
4.4 Interface for the **PartitionedDomain** Class . . . 102
4.5 Pseudo-Code for the **PartitionedDomain** Classes partition Method 102
4.6 Interface for the **DomainPartitioner** Class . . . 103
4.7 Interface for the **GraphPartitioner** Class . . . 104
4.8 Interface for the **Subdomain** Class . . . 105
4.9 Psuedo-Code for Selected Methods of the **Subdomain** Class . . . 107
4.10 Interface for the **DomainDecompAnalysis** Class . . . 108

4.11 Pseudo-Code for Selected Methods for the **DomainDecompAnalysis** Class ... 109
4.12 Interface for the DomainDecompAlgo Class ... 110
4.13 Pseudo-Code for the **DomainDecompAlgo** Classes solveCurrentStep Method ... 110
4.14 Interface for the **DomainSolver** Class ... 111
4.15 Revised Interface for the **Element** Class ... 113
4.16 Pseudo-Code for the **FE_Element** Classes formTangent Method ... 113

5.1 Computer Architecture for Parallel Computers ... 120
5.2 Flow of Data using Shadow and Actor Objects ... 135
5.3 Flow of Data when making a Call in RPC ... 136
5.4 Class Diagram for Actor/Aggregate Framework for Parallel Finite Element Analysis ... 138
5.5 Interface for the **Shadow** Class ... 140
5.6 Pseudo-Code for Selected Methods for the **Shadow** Class ... 141
5.7 Interface for the **ShadowSubdomain** Class ... 141
5.8 Pseudo-Code for Selected Methods for the **ShadowSubdomain** Class 142
5.9 Pseudo-Code for an Actor Program ... 144
5.10 Interface for the **Actor** Class ... 145
5.11 Interface for the **ActorSubdomain** Class ... 145
5.12 Pseudo-Code for the **ActorSubdomain** Classes run Method ... 146
5.13 Interface for the **Channel** Class ... 148
5.14 Interface for the **Address** Class ... 149
5.15 Interface for the **Message** Class ... 150
5.16 Interface for the **MovableObject** Class ... 151
5.17 Interface for the **MachineBroker** Class ... 152
5.18 Interface for the **ObjectBroker** Class ... 154
5.19 Revised Interface for the **NodalLoad** Class ... 155
5.20 Pseudo-Code for the **IntcrementalIntegrator**s formTangent Method 156
5.21 Revised Pseudo-Code for the **IncrementalIntegrator**s formTangent Method ... 156
5.22 Time Line for Original formTangent Method ... 157
5.23 Time Line for Revised formTangent Method ... 158

6.1 Two and Three Dimensional Test Models ... 163
6.2 Profile of CPU Time for C++ Program on ALPHA ... 169
6.3 Profile of Page Faults for C++ Program on ALPHA ... 172
6.4 Parallel Performance on ALPHA ... 178
6.5 Parallel Performance on DEC ... 179
6.6 Parallel Performance on ALPHA for 25 Iterations ... 182

7.1 Revised Interface for the **DomainPartitioner** Class ... 190
7.2 Interface for the **LoadBalancer** Class ... 191

7.3 The **HeavierToLighterNeighbours** Class . . . . . . 193

A.1 Interface of the **Matrix** Class . . . . . . 217
A.2 Interface of the **Vector** Class . . . . . . 219
A.3 Interface of the **ID** Class . . . . . . 220

# List of Tables


6.1 Two Dimensional Frame Examples . . . 164
6.2 Three Dimensional Frame Examples . . . 164
6.3 Hardware Environments for Performance Measurements . . . 165
6.4 Performance Results on HOLDEN . . . 167
6.5 Performance Results on ALPHA . . . 168
6.6 Performance Results on DEC . . . 168
6.7 Performance Results on ALPHA Cluster . . . 176
6.8 Performance Results on DEC Cluster . . . 177

7.1 Effect of Dynamic Load Balancing on the Real Time using ALPHA for 25 Analysis Steps with the **HeavierToLighterNeighbours** algorithm . . . 195

B.1 % CPU time on HOLDEN for C++ Program . . . 223
B.2 % CPU time on ALPHA for C++ Program . . . 224
B.3 % CPU time on DEC for C++ Program . . . 225
B.4 % Page Faults on ALPHA for C++ Program . . . 226
B.5 % Page Faults on DEC for C++ Program . . . 227
B.6 Results for 2dF1 on ALPHA Network . . . 228
B.7 Results for 2dF2 on ALPHA Network . . . 229
B.8 Results for 2dF3 on ALPHA Network . . . 230
B.9 Results for 2dF4 on ALPHA Network . . . 231
B.10 Results for 2dF5 on ALPHA Network . . . 232
B.11 Results for 2dF6 on ALPHA Network . . . 233
B.12 Results for 3dF3 on ALPHA Network . . . 234
B.13 Results for 3dF4 on ALPHA Network . . . 235
B.14 Results for 3dF5 on ALPHA Network . . . 236
B.15 Results for 3dF6 on ALPHA Network . . . 237
B.16 Results for 2dF1 on DEC Network . . . 238
B.17 Results for 2dF2 on DEC Network . . . 239
B.18 Results for 2dF3 on DEC Network . . . 240
B.19 Results for 2dF4 on DEC Network . . . 241
B.20 Results for 2dF5 on DEC Network . . . 242

B.21 Results for 2dF6 on DEC Network . . . . . 243
B.22 Results for 3dF3 on DEC Network . . . . . 244
B.23 Results for 3dF4 on DEC Network . . . . . 245
B.24 Results for 3dF5 on DEC Network . . . . . 246
B.25 Results for 3dF6 on DEC Network . . . . . 247

# Acknowledgements

I would like to thank my parents for their help, encouragement and prayers through all these years.

I would also like to thank Professor Gregory Fenves for his help and guidance over the many years I have been at Cal.

I would like to extend my thanks to my dissertation committee: Professor Steve Mahin and Professor James Demmel.

I would also like to thank my fellow graduate students, especially: Reginald de-Roches, Dawn Lehman, Laura Lowes, Silvia Mazzoni, Abe Lynn, Garret Hall, and Edward Love.

# Chapter 1

# Introduction

The finite element method is a numerical procedure used in engineering analysis for computing the approximate response of a system, whose response is given by the solution of an initial boundary value problem. It is the most popular tool used by engineers to analyze problems in structural and continuum mechanics (Hughes, 1987; Zienkiewicz and Taylor, 1989; Bathe, 1996). The reason for the popularity of the finite element method over the other analysis methods has been the digital computer, for which the formulation of the finite element method is particularly suited.

## 1.1 Problem Statement

At the present time a wide gap exists between state of the art computing capabilities and the computing performed by practicing engineers. Advances in both hardware (particularly parallel and distributed computing) and software (particularly the plug-and-play object-oriented paradigm) are not reflected in the programs used by engineers. Practicing engineers today are typically forced to perform finite element analysis on single processor machines, using the elements and analysis algorithms that are provided by a finite element analysis package.

The reason for this is that typical finite element packages consist of several hundred thousand lines of procedural code, typically Fortran, for execution on a single processor. The codes are not designed to allow the analysts to experiment with their own elements and analysis algorithms. Neither are the codes designed in a manner

which will allow extensions for parallel and distributed environments.

The ability to modify and extend any finite element software package is essential for packages to keep current with finite element technology and computing systems, both of which are advancing rapidly. The problems with modifying and extending existing codes are the following:

1. To modify and extend the code requires users with an intimate knowledge of the data structures and what procedures affect what portions of the data structures. For programs consisting of several hundred thousand lines of code, this effectively limits the numbers of users who can modify and extend the code. To allow naive analysts to extend and modify the code can lead to the introduction of bugs into the program, and can lead to multiple and uncontrolled versions of the program.

2. The ability to reuse code from other sources is limited. This is because data structures vary wildly between programs. As a consequence, to introduce code from other sources often requires that the code be modified to suit the data structures used in the current program.

3. When performing non-linear analysis, history variables are often required for each element and material point in the model. The management of these history variables is typically provided by the base program in common finite element analysis codes. For this reason a limit is placed on the number of history variables that each element and material point may use. This limit may be insufficient for certain types of elements and material formulations an analyst may wish to use.

4. The existing codes are typically designed and written for execution on a uniprocessor machine. Those programs which have been written for execution an parallel machines have the same limitations as existing codes. This is because the programs use the same design as for uniprocessor machines and as a consequence they inherit the problems identified above.

## 1.2 Object Oriented Programming

Object-oriented programming forms the basis for a very large part of the software development industry, particularly in the area of computer graphics, databases, graphical interfaces and operating system development. Object-oriented finite element packages, particularly those written in C++, have been shown (Dubois-Pelerin and Zimmermann, 1993; Rucki and Miller, 1996) to have comparable performance to their Fortran counterparts and still provide for the maintainability and extensibility essential of modern day software packages. This is due to their support of abstraction (which separates behavior from implementation), encapsulation (which keeps the essential aspects of the data hidden from the user of the data), modularity (decomposition of the system into a set of loosely coupled objects) and code reuse.

An object-oriented program is composed of objects, each with a number of attributes, that define the state of the object. The behavior of an object is defined by methods, which are procedures for changing or returning the state of the object. An object's method is invoked when another object sends a message to that object. The function of an object-oriented program can be viewed as the interaction between the program's objects by the sending of messages. To aid in program development, objects with similar attributes and behavior are grouped into classes. The classes are implemented in a programming language. For an implementation the following are provided for each class: a class interface (defining the methods that can be invoked on each object of the class), private data (defining the attributes held privately by each object), and method implementations (code defining the sequence of operations that objects of the class perform in order to complete the method invoked on the object). To promote code reuse, object-oriented programming languages support the notion of class hierarchies, with data and methods of an ancestor class being inherited by descendent classes. This inheritance feature allows the programmer to define the common function and data used by several classes at the highest possible level in the hierarchy, which avoids the duplication of code at the lower levels. The descendent classes may add additional attributes and methods, and can redefine the methods of an ancestor class, termed operator overloading. Inheritance allows an object of the descendent class to be treated as an object of an ancestor class.

A well designed object-oriented programming system enables programmers to independently develop and validate new code, to maintain and revise existing code, and to be able to introduce new code into existing programs. A poorly designed system will not. Fenves (1990) identifies the three essential steps in the development of a object-oriented system as: identification of the classes, specification of the class interfaces, and implementation.

## 1.3 Objective of the Research

The work to date in applying object-oriented programming techniques to the finite element method has largely concentrated on identifying the main class abstractions, i.e. elements, nodes, loads, constraints, matrix, vector, and some analysis classes. The existing object-oriented work is weak in two areas:

1. The interface to the analysis class, and the classes that the analysis class uses to perform its work, limits the flexibility, extensibility and code reuse that object-oriented programming can provide. This is because, as in traditional programming, the analysis class is treated as a black box, an irreducible and opaque operation. The analyst chooses the type of analysis to be performed by instantiating an analysis object of the correct type. The analyst must, however, modify existing analysis classes or provide new analysis classes to change even the storage scheme used for the system of equations in an analysis.

2. The work to date has largely ignored discussing how the object-oriented designs can be utilized in parallel and distributed computing environments. The requirement that the design be able to encompass parallel and distributed environments, particularly networks of workstations, is essential. This is because, as is argued by Anderson et al. (1995b), these are certain to become the primary computing platform for engineers and scientists in the coming years.

It is these two areas that are addressed in this dissertation. The objective of the research is to develop an object-oriented design that provides the analyst the tools to make full use of their computing resources in a transparent manner. The interfaces to

the classes will allow the analysts to modify and extend the analysis algorithms, using existing and user provided code fragments. The design recognizes the interaction between computing environment and analysis procedures in such a way as to allow the development of optimal solution strategies for different computing environments.

## 1.4 Overview of Dissertation

**Chapter Two**: A review is made of the existing work that has made use of object-oriented programming principles for the finite element method. The fundamental classes that have been used are identified and discussed. Class interfaces for those basic classes that will be used later in the dissertation are presented.

**Chapter Three**: A review is made of existing object-oriented designs for the analysis algorithms, which highlights the shortcomings of these approaches. A new design is then presented which would provide the analyst with a flexible and extensible environment for finite element analysis.

**Chapter Four**: An introduction to the common domain decomposition methods used in finite element analysis and a review of the existing object-oriented approaches to domain decomposition are first presented. The design presented in chapter 3 is then extended to include domain decomposition methods.

**Chapter Five**: An introduction to parallel programming and a brief review of the approaches which have been used to parallelize the finite element analysis are given. A new design is then presented for parallel environments, which allows the flexibility and extensibility of the design presented in chapters 2 and 3 to be retained.

**Chapter Six**: Results for a test implementation of the design are given for both uniprocessor and parallel machines. The issue of performance when using the object-oriented design is then examined.

**Chapter Seven**: A review is made of existing approaches to performing dynamic load balancing during a finite element analysis on parallel machines. A new approach is presented. Modifications to the existing framework is given. Results showing the performance improvement that can be obtained using a simple load balancing scheme are then presented.

The dissertation concludes with a summary of the development, discusses implementation, and presents directions for future research and development.

# Chapter 2

# Fundamental Object-Oriented Finite Element Design

In this chapter a review is made of the existing work that has been presented on object-oriented programming for finite element analysis. The basic classes that have been used are identified and discussed. New class interfaces, similar to those outlined in the literature, are provided both to demonstrate the functionality of the classes and for use in subsequent chapters.

## 2.1 Introduction

A number of object-oriented finite element design and implementations have been presented in the literature over the past several years. In this chapter a brief review is made of this existing work. In the work that has been presented similar class abstractions have been identified, with different class interfaces depending on the type of problem being solved. The classes that have been presented can be grouped into four broad categories:

1. Modeling classes: classes used to create the finite element model for a given problem.

2. Finite Element Model classes: classes used to describe the finite element model and to store the results of an analysis on this model.

3. Analysis classes: classes used to perform the analysis of the finite element model, i.e. form and solve the governing equations.

4. Numerical classes: these are classes used to handle the numerical operations in the solution procedure.

In the following four sections the work that has been presented in the literature is discussed. In addition, class interfaces for the basic classes that will be used in later chapters are presented using pseudo-C++ code. The pseudo-code is similar to the code used in the implementation, the only difference being that private data is not shown.

## 2.2 Modeling Classes

The instances of modeling classes are objects used by the analyst to create the finite element model, i.e. create the nodes, elements, loads and constraints. For large problems, it is important that the analyst be able to create the model in a simple and descriptive manner. A number of approaches are used:

1. In some of the work presented, the analyst creates the model in the main driver of the program (Ross et al., 1992; Zeglinski and Han, 1994; Cardona et al., 1994;

Archer, 1996). The analyst does this by explicitly creating each object for each node, element, load and boundary condition in the finite element model. To overcome the problem of having to recompile the program for each new problem, Cardona et al. (1994) provides an interpreted environment.

2. In other work, the analyst creates an input file which is read by the object representing the finite element model (Forde et al., 1990; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993; Menetrey and Zimmermann, 1993). The input file is read either during the object's construction, as in Dubois-Pelerin and Zimmermann (1993), or by the analyst sending a message to the object to initialize the model, as in Forde et al. (1990).

3. In some of the work, the analyst interacts with a graphical interface object (Ostermann et al., 1995; Mackie, 1995). For example, in Mackie (1995) the analyst interacts with a **StructureModel** object. This object allows the analyst to create graphical objects (**KeyPoints** and **KeyLines**) and objects describing material and geometry. The analyst creates one or more closed regions, a region consisting a a series of **KeyLine** objects. Each region has a pointer to objects representing the material and a geometry of the region. These pointers are set by the analyst. The analyst creates point forces and restraints at **KeyPoint** objects, and distributed loads along **KeyLine** objects. Classes for mesh generation are provided with the package, whose objects create the finite element model components based on the regions identified by the analyst.

For the purposes of this research, a **ModelBuilder** object will be used to create the finite element model in a running program. The **ModelBuilder** class, whose interface is as shown in figure 2.1, is an abstract class. An abstract class is a class for which there is no code supplied to implement some of the methods defined in the interface and, consequently, no objects of that specific class can be created. Subclasses are introduced to provide the implementation of these methods. The **ModelBuilder** class defines one pure virtual method, buildFE_Model(), for which all subclasses must provide an implementation.

```
class ModelBuilder {
    public:
        ModelBuilder(Domain &theDomain);
        virtual ~ModelBuilder();
        virtual int buildFE_Model(void) =0;
};
```

Figure 2.1: Interface of the **ModelBuilder** Class

Each **ModelBuilder** object, as shown in figure 2.2, will be associated with a single **Domain** object. When buildFE_Model() is invoked on a **ModelBuilder** object, the object is responsible for building the components of the model and adding them to the **Domain** object, which acts as a repository for these components. The manner in which the **ModelBuilder** object creates the model components depends on the subclass of the **ModelBuilder** class that the analyst uses to construct the **ModelBuilder** object. This approach allows the analyst to build models using any of the three approaches outlined above, by choosing an appropriate subclass of **ModelBuilder**.

## 2.3 Finite Element Model Classes

In most of the work that has been presented, the main class abstractions used to describe the finite element model are: **Node**, **Element**, **Constraint**, **Load** and **Domain** (Forde et al., 1990; Zimmermann et al., 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993; Menetrey and Zimmermann, 1993; Pidaparti and Hudl, 1993; Cardona et al., 1994; Chudoba and Bittnar, 1995; Zahlten et al., 1995; Rucki and Miller, 1996; Archer, 1996). These are similar to the abstractions used in traditional finite element programming.

The classes and the relationship amongst them can be graphically seen in Figure 2.2, which shows the class diagram for the analysis using Rumbaugh notation (Rumbaugh et al., 1991). Class diagrams are used to show the existence of classes and the relationship between classes. In Rumbaugh notation, a class is represented by a rectangle, with the class name inside the rectangle, and the relationships be-

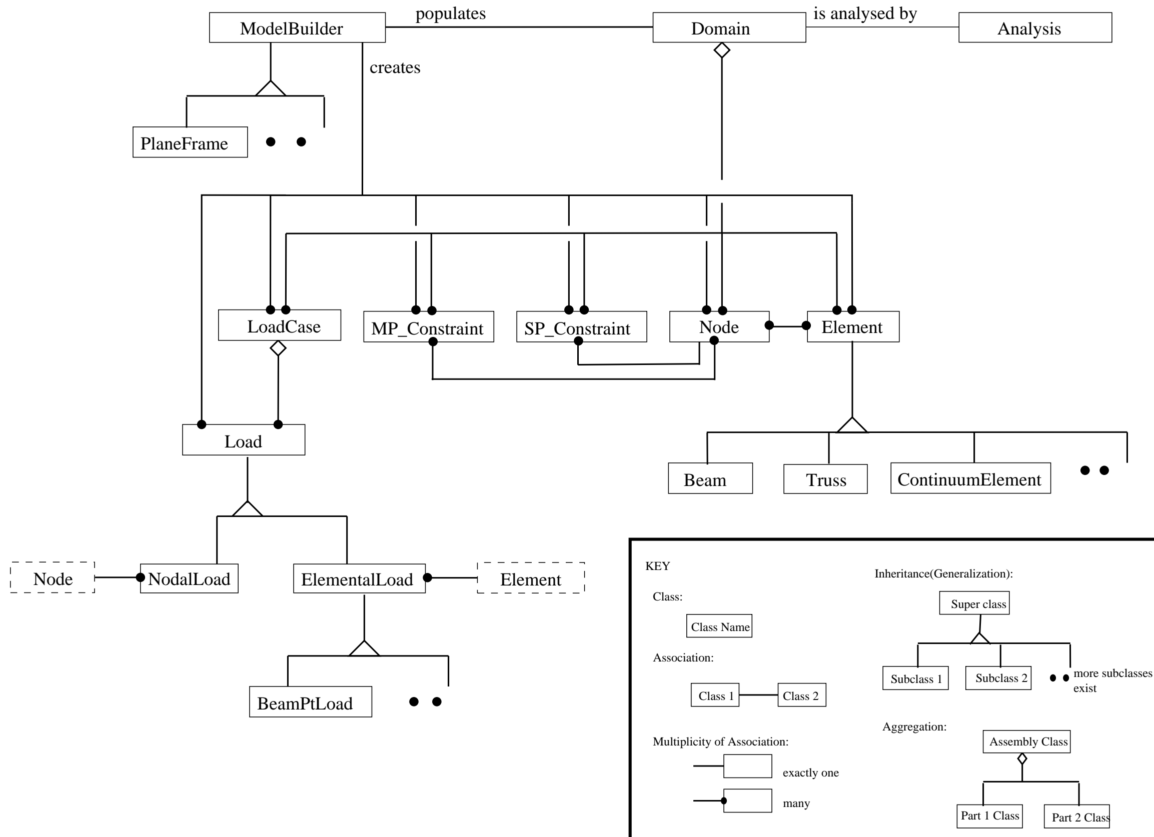



Figure 2.2: Class Diagram for the Finite Element Method

tween classes are represented by lines between the classes. There are three types of relationships in object-oriented programming:

1. **knows-a**: The knows-a relationship exists between classes when an object of one class knows about an object of another class. For example, an **Element** object knows about its **Node** objects. A knows-a relationship is represented by a line between two rectangles.

2. **is-a**: The is-a relationship, termed inheritance, exists when an instance of one class, the descendent class, can be treated as an instance of another class, the ancestor class. For example, a **Beam** object can be treated as an **Element** object by the **Domain**. The is-a relationship is represented by by a line with a triangle between the classes. Those classes which share a common parent class

are shown by lines connecting to the base of the triangle.

3. **has-a**: The has-a relationship, termed aggregation, exists when an object of one class is made up of component objects of other classes. For example, the **Domain** object is an aggregation of **Element**, **Node**, **Load** and **Constraint** objects. The has-a relationship is represented by a diamond at the aggregate class and a line from the diamond to the classes of the component objects.

In the following subsections, a review is made of the work presented for each of the main abstractions used to describe the finite element model.

### 2.3.1 Node Class

A **Node** object represents a discrete point in the domain at which response quantities (degrees-of-freedom) are defined. The function of a **Node** object is to store its coordinates within the domain, to store its response and unbalanced load information, and to provide methods to set and retrieve this information. In most of the work presented in the literature, the only response quantity that can be stored at a **Node** object is the displacement, though in some of the work, the velocity and acceleration can be stored as well (Cardona et al., 1994; Archer, 1996).

In much of the work that has been presented, each Node object is associated with a number of **DOF** objects. (Miller, 1991; Baugh and Rehak, 1992; Zimmermann et al., 1992; Dubois-Pelerin et al., 1992; Menetrey and Zimmermann, 1993; Rihaczek and Kroplin, 1993; Cardona et al., 1994; Zahlten et al., 1995; Rucki and Miller, 1996; Archer, 1996). The **DOF** objects are the repositories of the response and load quantities for each degree-of-freedom at a **Node** object. The **DOF** objects provide methods in their interface to set and retrieve these quantities. In some of the work, the **DOF** objects also keep track of their local coordinate systems (Miller, 1991; Baugh and Rehak, 1992; Miller and Rucki, 1993; Rucki and Miller, 1996; Archer, 1996).

For the purposes of this research, a **Node** class is introduced. The class, whose interface is as shown in figure 2.3, provides for the following:

1. The constructors allow **Node** objects to be built for one-, two-, and three-dimensional problems, with as many degrees-of-freedom associated with the

```
class Node : public DomainComponent {
    public:
        Node(int nodeTag); // to allow subclasses to be introduced
        Node(int nodeTag, int ndof, double Crd1);
        Node(int nodeTag, int ndof, double Crd1, double Crd2);
        Node(int nodeTag, int ndof, double Crd1, double Crd2, double Crd3);
        virtual ~Node();

        virtual const Vector &getCrds(void) const;
        virtual void setDOF_GroupPtr(DOF_Group *theDOF_Grp);
        virtual DOF_Group *getDOF_GroupPtr(void);
        virtual int getNumberDOF(void) const;

        virtual const Matrix &getMass(void) const;
        virtual void setMass(const Matrix &mass);

        // method to return the last committed response quantities
        virtual const Vector &getCommitDisp(void) const;
        virtual const Vector &getCommitVel(void) const;
        virtual const Vector &getCommitAccel(void) const;

        // methods to return the current trial responses
        virtual const Vector &getTrialDisp(void) const;
        virtual const Vector &getTrialVel(void) const;
        virtual const Vector &getTrialAccel(void) const;

        // methods to set the current trial response quantities
        virtual void setTrialDisp(const Vector &);
        virtual void setTrialVel(const Vector &);
        virtual void setTrialAccel(const Vector &);

        // trial response quantities equal committed plus increment
        virtual void setIncrTrialDisp(const Vector &);
        virtual void setIncrTrialVel(const Vector &);
        virtual void setIncrTrialAccel(const Vector &);

        // methods to set and retrieve the unbalanced load
        virtual void addUnbalancedLoad(const Vector &, double factor = 1.0);
        virtual const Vector &getUnbalancedLoad(void) const;
        virtual void zeroUnbalancedLoad(void);

        // method which sets current committed quantities
        // equal to the current trial quantities
        virtual int commitState();
};
```

Figure 2.3: Interface for the **Node** Class

objects as the analyst specifies.

2. Methods are provided to retrieve the nodal coordinates and information about the number of degrees-of-freedom at the **Node** object.

3. Methods are provided to set/retrieve trial response quantities at/from the nodal degrees-of-freedom. The response quantities, which include the displacement, velocity, and acceleration, can be set directly, e.g. setTrialDisp() or set incrementally, e.g. setIncrTrialDisp().

4. Methods are provided to set/retrieve the committed response quantities at/from the nodal degrees-of-freedom. The committed response quantities represent points along the solution path. The commit() method is used to set the committed values to be equal to the current trial values.

5. Methods are provided to zero out, increment, and retrieve the unbalanced load information at the **Node** object.

6. Methods are provided to set and retrieve the nodal mass matrix.

The **Node** class allows **Node** objects to be created that can be used in both linear and non-linear analysis. It does this by defining two sets of response quantities, committed and trial. The trial response quantities represent a point in the solution space that has not yet been accepted as being on the solution path. The committed quantities represent the last trial response that was accepted as being on the solution path. Archer (1996) presents a similar approach.

### 2.3.2 Element Class

The basic functionality of an **Element** object is to provide the current linearized stiffness, mass, and damping matrices, and the residual force vector due to the current stresses and element loads. In most of the work presented, the **Element** class is a abstract base class, which defines the interface that all subclasses must provide. This approach allows new **Element** subclasses to be introduced without changing the existing code in the program, as discussed in Mackie (1992). It should be noted

that, most finite element analysis programs written in procedural languages provide facilities for adding elements. This is because the interface is well established. An object-oriented approach, however, better isolates the element functions from analysis and solution algorithm functions, allows inheritance of common functions, and allows the **Element** objects to store as much private data as is required by the element.

In some of the work, the **Element** objects keep track of their local coordinate systems and are able to provide transformations between these local coordinate systems and the coordinate systems at their associated **Node** objects (Miller, 1991; Baugh and Rehak, 1992; Miller and Rucki, 1993; Rucki, 1996). This transformation is sometimes performed by an object acting as an interface between the analysis and the model objects (Lu et al., 1993; Chudoba and Bittnar, 1995; Archer, 1996).

For continuum elements, classes are sometimes provided to support the element formulation (Forde et al., 1990; Zimmermann et al., 1992; Pidaparti and Hudl, 1993; Chudoba and Bittnar, 1995; Zahlten et al., 1995). In Forde et al. (1990), where an implementation for linear static analysis problems is presented, each continuum **Element** object is associated with a **ShapeFunction** object, a **Material** object, and a number of **GaussPoint** objects. For problems involving non-linear materials, in Chudoba and Bittnar (1995) a **MaterialPoint** object is associated with each **GaussPoint** object. In Zahlten et al. (1995) class abstractions for the following are introduced: cross section, material point, material law, yield surface, hardening rule, and flow rule.

For the purposes of this research, an **Element** class is introduced. The class is an abstract base class. The class interface, which is as shown in figure 2.4, provides for the following:

1. Each **Element** object is associated with a number of **Node** objects, as shown in figure 2.2. Methods are provided to allow other objects in the program to obtain information about the number of **Node**s and the **Node** identifiers.

2. Methods are provided to return the current linearized mass, stiffness and damping matrices for the element given the current trial state at the **Node** objects.

3. Methods are provided to zero and return the current residual load information

and to allow **ElementalLoad** objects, to add their load contribution to the residual, as will be discussed in section 2.3.4

```
class Element: public DomainComponent {
    public:
        Element(int tag);
        virtual ~Element;

        // methods to get Node and DOF info
        virtual int getNumExternalNodes(void) const =0;
        virtual const ID &getExternalNodes(void) const =0;
        virtual int getNumDOF(void) const =0;

        // methods to obtain the mass, stiffness and damping matrices
        virtual const Matrix &getStiff(void)=0;
        virtual const Matrix &getDamp(void)=0;
        virtual const Matrix &getMass(void)=0;

        // methods for returning and applying loads
        virtual void zeroLoad(void) =0;
        virtual int addLoad(const Vector &load) =0;
        virtual const Vector &getResistingForce(void) =0;

        virtual void commitState(void) = 0;
};
```

Figure 2.4: Interface for the **Element** Class

4. The interface allows subclasses to be provided, which can be used by the analyst to create **Element** objects for use in linear and non-linear analysis. It does this by the provision of the commit() method. Each **Element** object in a non-linear analysis can use the invocation of this method to save current state information. Without this trial states could not be determined for certain non-linear **Element** subclasses for which state is path dependent.

All the methods in the interface are declared as being pure virtual, so no implementation of the methods are provided by the **Element** class. The subclasses of the **Element** class provide the implementation.

### 2.3.3 Constraint Classes

Constraints on a finite element model can be of two types:

1. Single-Point constraints: where the prescribed values of a degrees-of-freedom are specified.

2. Multi-Point constraints: where a relationship between the response quantities for a number of degrees-of-freedom at two nodes is specified.

In much of the work that has been presented, a **Constraint** class is introduced which is only capable of handling homogeneous single-point constraints (Forde et al., 1990; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993; Pidaparti and Hudl, 1993; Cardona et al., 1994). The constraint is enforced by the **Constraint** objects informing the associated **Node** objects that a certain degree-of-freedom has been constrained and should not be assigned an equation number in the system of equations.

In Baugh and Rehak (1992) constraints can be enforced by the analyst who provides auxiliary procedures. These procedures are invoked before a matrix operation is performed.

In some of the work, separate classes are introduced for single- and multi- point constraints (Miller and Rucki, 1993; Rucki, 1996; Archer, 1996). In Archer (1996) the **Analysis** object is responsible for enforcing the constraints by introducing the constraints into the system of equations. Rucki (1996) provides a **DOF_Filter** class, whose subclasses **LineFilter**, **PlaneFilter** and **TotalFilter** restrict the displacement to a line, a plane and totally. For multi-point constraints Rucki (1996) propose that a **DOF_Connector** class be developed.

For the purposes of this research, separate single and multi-point constraint classes are provided, **SP_Constraint** and **MP_Constraint**, respectively. The constraint objects in the system do not enforce the constraints, they merely inform the objects used in the analysis that constraints exist on the model and what the values of these constraints are.

Each **SP_Constraint** object is associated with a single **Node** object, as shown in figure 2.2. The **SP_Constraint** class, whose interface is as shown in figure 2.5,

provides methods to obtain the constrained node's number, the number of the degree-of-freedom that is constrained, and the value of the constraint for a specified time. A further method is provided to indicate whether or not the constraint is homogeneous, which is information needed to determine if the number of equations can be reduced. The **SP_Constraint** class interface only allows for time invariant constraints, time varying constraints can be provided by subclasses of **SP_Constraint**.

```
class SP_Constraint : public DomainComponent {
    public:
        SP_Constraint(); // to allow subclasses to be introduced
        SP_Constraint(int node, int ndof, double value, bool isHomo);
        virtual ~SP_Constraint();

        virtual int getNodeTag(void) const;
        virtual int getDOF_Number(void) const;
        virtual double getValue(double timeStep =0.0);

        virtual bool isHomogeneous(void);
};
```

Figure 2.5: Interface for the **SP_ Constraint** Class

Each **MP_Constraint** object is associated with two **Node** objects, as shown in figure 2.2. The **MP_Constraint** class, whose interface is as shown in figure 2.6, provides methods to obtain the constrained node number, the retained node number, those degrees-of-freedom at the constrained node that have constraints imposed on them, and the constraint matrix at a specified time. Like the **SP_Constraint** class interface, the **MP_Constraint** class interface would only allow for time invariant constraints, time varying constraints can be provided by subclasses of **MP_Constraint**.

### 2.3.4 Load Classes

Loads on a finite element model can be divided into two types:

1. Nodal Loads: these are loads acting on the nodes.

2. Element Loads: these are loads acting on the elements, which can be due to body forces, surface tractions, initial stresses, and temperature gradients. It is

```
class MP_Constraint : public DomainComponent {
    public:
        MP_Constraint(); // to allow subclasses to be introduced
        MP_Constraint(int nodeRetain,
                      int nodeConstr,
                      Matrix &constrnt,
                      ID &participatingDOF);
        virtual ~MP_Constraint();

        virtual int getNodeRetained(void) const;
        virtual int getNodeConstrained(void) const;
        virtual const ID & getConstrainedDOF(void) const;
        virtual const Matrix &getConstraint(double time =0.0);
};
```

Figure 2.6: Interface for the **MP_Constraint** Class

important that finite element programs provide element load abstractions, as without them the analyst is forced to determine the element load contributions by hand and add these to the nodal loads.

In much of the previous work, only classes for nodal loads are provided (Forde et al., 1990; Baugh and Rehak, 1992; Miller and Rucki, 1993; Mackie, 1995; Rucki and Miller, 1996). In other work, classes for both element and nodal loads are provided (Dubois-Pelerin et al., 1992; Chudoba and Bittnar, 1995; Zahlten et al., 1995; Archer, 1996).

The **Element** and **Node** objects typically keep a list of the loads acting on them, and are able to determine the loads contributions to the element residual and nodal unbalance by looping through these lists (Baugh and Rehak, 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993). In Archer (1996), where the **LoadCase** class is also introduced, each **Load** object in the current **LoadCase** makes itself known to its corresponding component. In Rucki (1996) the **Load** objects add their contributions to the **Element** and **Node** objects. In Forde et al. (1990) the objects performing the analysis work directly with the **Load** objects.

For the purposes of this work, three classes are introduced, **LoadCase**, **NodalLoad** and **ElementalLoad**. The **LoadCase** class interface, which is as shown in figure 2.7, provides methods in its interface to allow **NodalLoad** and **Elemental-**

**Load** objects to be added to a **LoadCase** object, to remove these objects from the **LoadCase** object, and to obtain iterators to these objects. The interface also provides the applyLoad() method, which will invoke applyLoad() on all **NodalLoad**s and **ElementalLoad**s in the **LoadCase** object.

```
class LoadCase : public DomainComponent {
    public:
        LoadCase(int tag);
        virtual ~LoadCase();

        virtual bool addNodalLoad(NodalLoad *loadPtr);
        virtual bool addElementalLoad(ElementalLoad *loadPtr);
        virtual NodalLoadIter &getNodalLoads(void);
        virtual ElementalLoadIter &getElementalLoads(void);
        virtual NodalLoad *removeNodalLoad(int tag);
        virtual ElementalLoad *removeElementalLoad(int tag);

        void applyLoad(double timestep = 0.0);
};
```

Figure 2.7: Interface for the **LoadCase** Class

Each **NodalLoad** object is associated with a **Node** object, as shown in figure 2.2, and is used by the analyst to apply nodal loads to that object. The **NodalLoad** class, whose interface is as shown in figure 2.8, provides methods to obtain the node number of its associated **Node** object and to apply the load at a specified time. When the applyLoad() method is invoked on a **NodalLoad** object, it will add its contribution to the **Node** objects load by invoking addUnbalancedLoad() on that **Node** object.

Each **ElementalLoad** object is associated with an **Element** object, as shown in figure 2.2, and is used by the analyst to apply element loads to that object. The **ElementalLoad** class, whose interface is as shown in figure 2.9, provides methods to obtain the element number and to apply the load at a specified time. When the applyLoad() method is invoked, the **ElementLoad** object will add its contribution to the **Element** object's load by invoking addLoad() on its associated **Element** object. The **ElementalLoad** class is an abstract class. Each **Element** class will have its own subclasses of **ElementalLoad**, which define how element loads can be added to **Element** objects of that type.

```
virtual class NodalLoad : public Load {
    public:
        NodalLoad(int nodeTag);
        NodalLoad(int nodeTag, const Vector &load);
        ~NodalLoad();

        virtual int getNodeTag(void) const;
        virtual void applyLoad(double timestep = 0.0);
};
```

Figure 2.8: Interface for the **NodalLoad** Class

```
virtual class ElementalLoad : public Load {
    public:
        ElementalLoad(int elementTag);
        ~ElementalLoad();

        virtual int getElementTag(void) const;
        virtual void applyLoad(double timestep = 0.0) =0;
};
```

Figure 2.9: Interface for the **ElementalLoad** Class

### 2.3.5 Domain Class

A **Domain** object is a container responsible for holding all the components of the finite element model, i.e. the **Node**, **Element**, **Constraint**, and **Load** objects. The class that is used for this purpose goes by many different names: NAP (Forde et al., 1990), LocalDB (Miller, 1991), Assemblage (Miller and Rucki, 1993), Partition (Rucki and Miller, 1996), FE_Model (Mackie, 1995), Model (Archer, 1996), and Domain (Zimmermann et al., 1992; Menetrey and Zimmermann, 1993; Cardona et al., 1994; Chudoba and Bittnar, 1995). For the purposes of this research, the class name **Domain** will be used.

The functionality of the **Domain** class, as presented in the literature, can be divided into two categories:

1. The **Domain** class provides methods to add components to the **Domain** object. This can be done by adding the components individually (Archer, 1996) or by adding the components of similar type, e.g. the **Element** objects, as a collection (Cardona et al., 1994). In some of the work, the interface does not have to provide such methods, as the domain components are created by the Domain object itself (Forde et al., 1990; Dubois-Pelerin and Zimmermann, 1993; Menetrey and Zimmermann, 1993).

2. The **Domain** class provides methods to access the domain components. In Archer (1996) the access is provided by iterators. In Rucki and Miller (1996) the access is provided through manager objects, which manage the individual collections. In those designs, where the **Domain** object is also responsible for performing the analysis of the model, no such access methods are provided by the **Domain** class (Forde et al., 1990; Zimmermann et al., 1992; Menetrey and Zimmermann, 1993).

For the purposes of this research, a **Domain** object is associated with a **ModelBuilder** object and an **Analysis** object, as shown in figure 2.2. The **Domain** class interface, which is as shown in figure 2.10, provides for the following:

1. Two constructors are provided, one of which takes hints as to the number of

components that are to be added to the **Domain**. This is for efficiency reasons in large problems.

2. Methods are provided to add and remove components to the domain. When adding objects to the domain, the analyst can ask that checks be performed to ensure the validity of the domain. For example, when adding an **Element** object the **Domain** object can check to ensure that all the **Element**'s **Node** objects exist in the **Domain**. This is done for efficiency reasons. It is important that checks be provided when adding the components, as it eliminates the needs for certain checks to be performed during the analysis algorithm, checks which may be repeated numerous times in iterative strategies.

3. Methods are provided to access the components of the domain. The access can be individually, e.g. getElementPtr(), or on the collection of like components, in which case an iter is returned, e.g. getElements().

4. Methods are provided which update the state of the components of the Domain. For example, the method commit() is provided, which when called will invoke commit() on all the **Element**s and **Node**s in the **Domain**. The method applyLoad() will invoke applyLoad() on all **Load** objects in the current **LoadCase**.

5. Methods are provided to obtain information about the current state of the **Domain**. This includes the time, current **LoadCase**, and connectivity information for the **Node**s and **Element**s. The connectivity information is returned in the form of **Graph** objects.

6. Methods are provided which allow the **Domain** to be modified by the analyst between analyses, which may be required in contact problems. For example, the method invokeChangeOnAnalysis() is a method which will invoke domainChanged() on its associated **Analysis** object, setAnalysis() is a method to set the link to the **Analysis** object associated with the **Domain**, and hasDomainChanged() is a method which returns true, if new components have been added to the **Domain** since the last call to this method.

```
class Domain {
    public:
        Domain();
        Domain(int numEle, int numNode, int numLC, int numSp, int numMp);
        virtual ~Domain();

        // methods to add/remove components to/from the domain
        virtual bool addElement(Element *elePtr, bool check = false);
        virtual bool addNode(Node *nodePtr, bool check = false);
        virtual bool addSP_Constraint(SP_Constraint *spPtr, bool check = false);
        virtual bool addMP_Constraint(MP_Constraint *mpPtr, bool check = false);
        virtual bool addLoadCase(LoadCase *lcPtr, bool check = false);
        virtual bool addNodalLoad(int lcTag, NodalLoad *ldPtr, bool check = false);
        virtual bool addElementalLoad(int lcTag, ElementalLoad *ldPtr, bool chk = false);
        virtual Element *removeElement(int tag);
        virtual Node *removeNode(int tag);
        virtual LoadCase *removeLoadCase(int tag);
        virtual SP_Constraint *removeSP_Constraint(int tag);
        virtual MP_Constraint *removeMP_Constraint(int tag);
        virtual NodalLoad *removeNodalLoad(int lc, int tag);
        virtual NodalLoad *removeElementalLoad(int lc, int tag);

        // methods to access the components of the domain
        virtual ElementIter &getElements();
        virtual NodeIter &getNodes();
        virtual LoadCaseIter &getLoadCases();
        virtual SP_ConstraintIter &getSPs();
        virtual MP_ConstraintIter &getMPs();
        virtual Element *getElementPtr(int tag) const;
        virtual Node *getNodePtr(int tag) const;
        virtual LoadCase *getLoadCasePtr(int tag) const;

        // methods to update/query the state of the domain and its components
        virtual void setCurrentLoadCase(int tag);
        virtual void setCurrentTime(double newTime);
        virtual void applyLoad(double timeStep = 0.0);
        virtual void commit(void);
        virtual Graph &getElementGraph(void);
        virtual Graph &getNodeGraph(void);
        virtual int getCurrentLoadCase(void) const;
        virtual double getCurrentTime(void) const;

        virtual bool hasDomainChanged(void);
        virtual int setAnalysis(Analysis &theAnalysis);
        virtual int invokeChangeOnAnalysis(void);
};
```

Figure 2.10: Interface for the **Domain** Class

## 2.4 Analysis Classes

The **Analysis** object forms and solves the governing equations for the finite element model. The type of analysis that can be performed by the analyst depends on the **Analysis** classes provided and the interface of the components of the finite element model. In some of the work, only linear static analysis can be performed on the model (Forde et al., 1990; Baugh and Rehak, 1992; Yu and Adeli, 1993). In other work linear static and transient analysis can be performed (Zimmermann et al., 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993; Pidaparti and Hudl, 1993; Cardona et al., 1994). In other work, both linear and non-linear static and transient analysis can be performed (Menetrey and Zimmermann, 1993; Miller and Rucki, 1993; Chudoba and Bittnar, 1995; Rucki and Miller, 1996; Archer, 1996). Pidaparti and Hudl (1993) allow for an eigenvalue analysis.

The typical approach that has been taken to the **Analysis** class is the black box approach of traditional finite element programming (Forde et al., 1990; Zimmermann et al., 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993; Pidaparti and Hudl, 1993; Rucki and Miller, 1996; Archer, 1996). With this approach, a number of subclasses of **Analysis** are provided. The analyst constructs an **Analysis** object of the correct class to perform the analysis. Typically the **Analysis** object itself instantiates other objects to help with the analysis, the analyst however has no control over the classes used to create these objects. A more detailed review of the existing work will be presented in section 3.3, and a new approach to the design of the analysis class will be presented in section 3.4.

## 2.5 Numerical Classes

The finite element method requires intense numerical computations and the systems that have been proposed in the literature provide a number of numerical classes. These classes support numerical algorithms. In the following subsections, a review is made of these numerical classes.

### 2.5.1 Matrix and Vector Classes

In most of the work that has been presented, much use is made of **Matrix** and **Vector** classes in the formulation of the algorithms and as data units for passing information between the objects in the system (Forde et al., 1990; Baugh and Rehak, 1992; Mackie, 1992; Zimmermann et al., 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993; Menetrey and Zimmermann, 1993; Lu et al., 1993; Pidaparti and Hudl, 1993; Raphael and Krishnamoorthy, 1993; Cardona et al., 1994; Chudoba and Bittnar, 1995; Ostermann et al., 1995; Archer, 1996). For this reason, a number of researchers have focused their attention on developing these classes for integration into finite element analysis packages (Scholz, 1992; Zeglinski and Han, 1994; Lu et al., 1995). Scholz (1992) present the the interfaces for **Matrix** and **Vector** classes which is similar to the ones used by subsequent researchers. In some of the work, subclasses of **Matrix** for symmetric, upper, lower, sparse, band, symmetric band and profile storage schemes are presented (Baugh and Rehak, 1992; Zeglinski and Han, 1994; Lu et al., 1995). Dongarra et al. (1995) present the C++ version of LAPACK (Anderson et al., 1995a), which is shown to be as fast as the Fortran version.

Cardona et al. (1994) introduce special **Matrix** and **Vector** subclasses for use in a finite element analysis. These are **StructureMatrix** and **StructureVector**. The construction of a **StructureMatrix** object will actually perform the assembly of the tangent stiffness matrix. Objects created from subclasses of **StructureVector**, which are **genDispl**, **genVel**, **genAccel**, do not actually store the values, but hold the mapping to the correct nodal degrees-of-freedom. An assignment to the **Vector** object will cause the object to update the appropriate nodal response quantities at the **Node** object.

For the purposes of this work, three classes are provided: **Matrix**, **Vector**, and **ID**. The **Matrix** and **Vector** classes provide a similar interface to that presented in Scholz (1992). An **ID** object is a special form of vector for handling integers. There are no methods provided to add, subtract or multiply **ID** objects. Instead methods are provided to check if an integer value is in the **ID**, to return the location in the **ID** of an integer object, and to obtain the integer values held by the **ID**. The interfaces

for the **Matrix**, **Vector** and **ID** classes are presented in more detail in appendix A.

### 2.5.2 Tensor Classes

In some of the work, tensor classes are presented (Baugh and Rehak, 1992; Hoffmeister et al., 1993; Lu et al., 1993; Rucki, 1996). Tensors, unlike matrices and vectors, are aware that the quantities contained exist in three dimensional space. **Tensor** objects keep track of the current coordinate system and are able to transform their quantities into other coordinate systems. Hoffmeister et al. (1993) present classes for rank one through rank four tensors. Rucki (1996) have classes for rank one and rank two tensors.

### 2.5.3 Linear System of Equation Classes

In some of the work, linear system of equation classes, **LinearSOE**, are used (Zimmermann et al., 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993). Dubois-Pelerin et al. (1992) points out that these are classes which, in addition to having a regular matrix, have a right hand side and a solution. Methods to assign the next equation number, to add to the matrix, to add to the right hand side and to solve the system are supplied at the interface. For the purposes of this work, two classes are provided: **SystemOfEqn** and **Solver**. The **SystemOfEqn** object is responsible for storing the equations. The **Solver** object is responsible for performing the numerical operations on the equations. These two classes are presented in more detail in section 3.4.9.

## 2.6 Summary

This chapter reviewed the main class abstractions identified in the literature for the finite element method. The classes that have been identified are: **ModelBuilder**, **Domain**, **Element**, **Node**, **Constraint**, **Load**, **Analysis**, **Matrix**, **Vector**, **Tensor** and **LinearSOE**. For those classes that will be used subsequently in this research, a class interface was presented. These interfaces provide similar functionality to those

presented in the literature. Additional features have been provided in these interfaces for efficiency reasons, e.g. commit() in **Domain**.

It has been pointed out in the literature that the flexibility and extensibility of an object-oriented design for the finite element analysis is exemplified by the ease with which new **Element** subclasses can be introduced. This is due to the fact that the **Element** interface is well understood. By choosing the subclass of **Element** to use in an analysis, the analyst is choosing which boundary value problem is being solved for a given geometry. The analyst can change the boundary value problem by simply changing the type of **Element** subclass used to construct the **Element**s.

Just as important to an analyst, however, is changing the type of the analysis that is performed on the **Domain**. The typical approach, as discussed in section 2.4, is the black box approach in which the analyst creates an **Analysis** object which performs the analysis. The **Analysis** object will typically create objects to help it perform the analysis, e.g. **LinearSOE** objects. While this typical approach offers the analyst a flexible and extensible platform (flexible in the sense that the analyst can choose the type of analysis and extensible in the sense that the analyst can introduce a new analysis class), it does not lead to code re-use. This is because it does not allow the flexibility and extensibility that is inherent in a well designed object-oriented system. In a well designed system, the analyst would be able to create an **Analysis** object based on the objects the analyst supplies to this object. It is the design for such a system that will be presented in the next chapter.

# Chapter 3

# Object-Oriented Analysis Algorithms

In this chapter a new object-oriented design for finite element analysis is presented. The incremental solution strategies used for solving linear and non-linear static and transient problems are briefly introduced. A review is made of existing object-oriented designs for the analysis algorithm. A new framework is then presented, which is demonstrated by code examples, to provide a more flexible and extensible design than previous approaches. Extensions to the framework for other types of analysis are then discussed.

## 3.1 Introduction

There are three types of problems encountered in engineering analysis: steady-state or static problems, transient problems, and eigenvalue problems. Each of these can be further classified as being either linear or non-linear. For linear problems, the basic steps in a finite element analysis are:

1. Discretization of the domain into elements and nodes.

2. Formulation of the element matrices and vectors.

3. Formulation of the system of equations.

4. Incorporation of the boundary conditions.

5. Solution of the system of equations for the nodal degrees-of-freedom.

6. Computation of response within each element given values for the for the degrees-of-freedom in non-linear analysis.

The difference between the three types of problems are in the matrix system of equations that are formed. For each type of problem, there are many ways to solve these equations. When dealing with non-linear problems, the system of equations are non-linear, and iterative schemes are typically employed, e.g. Newton-Raphson, modified Newton, quasi Newton methods, to achieve a solution.

Step one of the analysis is performed by the analyst. Step two is performed by the element. The work presented in this chapter focuses on the analysis, which is considered to be steps three through six. In this work, particular attention is paid to the incremental solution techniques used to solve static or transient problems, as reviewed in section 3.2. In section 3.3 existing approaches for the analysis framework are reviewed. In section 3.4 the design of a new analysis framework is presented. This new approach uses object-oriented design principles to provide an analysis framework that provides more flexibility and extensibility than the current designs. In section 3.5 example code is presented, which demonstrates the new analysis approach. In section 3.6 a discussion is presented on how the framework can be extended to include other types of analysis.

## 3.2 The Incremental Solution of Nonlinear Finite Element Equations

The most general form of a non-linear equation for which a solution vector $\mathbf{U}$ is sought is:

$$\mathbf{R}(\mathbf{U}) = \mathbf{0} \tag{3.1}$$

where $\mathbf{R}(\mathbf{U})$ is a nonlinear vector function. For the finite element method, the general form for the transient problem, of which the static problem is a special case, seeks a solution ($\mathbf{U}$, $\dot{\mathbf{U}}$, $\ddot{\mathbf{U}}$) to the residual defined as:

$$\mathbf{R}(\mathbf{U}) = \mathop{\mathrm{A}}_{n=1}^{nodes} \left(\mathbf{P}_n(t) - \mathbf{M}_n\ddot{\mathbf{U}}_n\right) - \mathop{\mathrm{A}}_{e=1}^{elements} \left(\mathbf{F}_{I_e}(\ddot{\mathbf{U}}_e) + \mathbf{F}_{R_e}(\dot{\mathbf{U}}_e, \mathbf{U}_e) - \mathbf{P}_e(t)\right) \tag{3.2}$$

subject to the homogeneous multi-point constraints

$$\mathbf{C}(\mathbf{U}, t) = \mathbf{0} \tag{3.3}$$

where $\mathrm{A}$ is the assembly operator, $\mathbf{M}_n$ a nodal mass matrix, $\mathbf{P}_n(t)$ a nodal load vector, $\mathbf{F}_{I_e}(\ddot{\mathbf{U}}_e)$ an element inertia force, $\mathbf{F}_{R_e}(\dot{\mathbf{U}}_e, \mathbf{U}_e)$, an element resisting load due to internal stresses and $\mathbf{P}_e(t)$ an element load due to externally applied loads. The response quantities at the nodes ($\mathbf{U}_n$, $\dot{\mathbf{U}}_n$, $\ddot{\mathbf{U}}_n$) and elements ($\mathbf{U}_e$, $\dot{\mathbf{U}}_e$, $\ddot{\mathbf{U}}_e$)are given by:

$$\mathbf{U}_\varrho = \mathbf{T}_\varrho\mathbf{U} + \mathbf{T}_{s\varrho}\mathbf{U}_s \tag{3.4}$$

$$\dot{\mathbf{U}}_\varrho = \mathbf{T}_\varrho\dot{\mathbf{U}} + \mathbf{T}_{s\varrho}\dot{\mathbf{U}}_s \tag{3.5}$$

$$\ddot{\mathbf{U}}_\varrho = \mathbf{T}_\varrho\ddot{\mathbf{U}} + \mathbf{R}_{s\varrho}\ddot{\mathbf{U}}_s \tag{3.6}$$

where ($\mathbf{U}_s$, $\dot{\mathbf{U}}_s$, $\ddot{\mathbf{U}}_s$) are the specified support motions and ($\mathbf{T}_{s\varrho}$, $\mathbf{R}_{s\varrho}$) are transformation matrices for each node ($\varrho = n$) and element ($\varrho = e$). This general formulation covers the most important problems in structural engineering.

The most widely used technique for solving the non-linear finite element equation, equation 3.2, is to use an incremental formulation. In an incremental formulation, a solution to the equation is sought at successive time steps, or load increments in a static problem. The solution at each step, $t$, is in the form of a displacement increment $\Delta\mathbf{U}_t$. Given the solution at $\mathbf{U}_t$ satisfying $\mathbf{R}(\mathbf{U}_t) = \mathbf{0}$, we seek $\Delta\mathbf{U}_t$ such that $\mathbf{R}(\mathbf{U}_{t+\Delta t}) = \mathbf{0}$ where $\mathbf{U}_{t+\Delta t} = \mathbf{U}_t + \Delta\mathbf{U}_t$.

Various implicit direct integration schemes are used in dynamic analysis to relate velocities and accelerations to $\Delta\mathbf{U}$. The integration schemes provide two operators, $\mathbf{I}_1$ and $\mathbf{I}_2$, to give the velocity and displacement in terms of $\Delta\mathbf{U}_t$ and the response at previous steps:

$$\dot{\mathbf{U}}_{t+\Delta t} = \mathbf{I}_1(\Delta\mathbf{U}_t, \mathbf{U}_t, \dot{\mathbf{U}}_t, \ddot{\mathbf{U}}_t, \mathbf{U}_{t-\Delta t}, \dot{\mathbf{U}}_{t-\Delta t}, \ddot{\mathbf{U}}_{t-\Delta t} \ldots, ) \tag{3.7}$$

$$\ddot{\mathbf{U}}_{t+\Delta t} = \mathbf{I}_2(\Delta\mathbf{U}_t, \mathbf{U}_t, \dot{\mathbf{U}}_t, \ddot{\mathbf{U}}_t, \mathbf{U}_{t-\Delta t}, \dot{\mathbf{U}}_{t-\Delta t}, \ddot{\mathbf{U}}_{t-\Delta t} \ldots, ) \tag{3.8}$$

For example, using the Newmark_$\gamma\beta$ scheme

$$\dot{\mathbf{U}}_{t+\Delta t} = \dot{\mathbf{U}}_t + \frac{\gamma}{\beta\Delta t}\Delta\mathbf{U}_t - \frac{\gamma}{\beta}\dot{\mathbf{U}}_t + \Delta t\left(1 - \frac{\gamma}{2\beta}\right)\ddot{\mathbf{U}}_t \tag{3.9}$$

$$\ddot{\mathbf{U}}_{t+\Delta t} = \ddot{\mathbf{U}}_t + \frac{1}{\beta\Delta t^2}\Delta\mathbf{U} - \frac{1}{\beta\Delta t}\dot{\mathbf{U}}_t - \frac{1}{2\beta}\ddot{\mathbf{U}}_t \tag{3.10}$$

Using these direct integration schemes, a solution is sought for the nonlinear finite element equation, equation 3.2. This is typically done using an iterative procedure, i.e. making an initial prediction for $\mathbf{U}_{t+\Delta t}$, denoted $\mathbf{U}^{(0)}_{t+\Delta t}$ a sequence of approximations $\mathbf{U}^{(i)}_{t+\Delta t}$, $i = 1, 2, ..$ is obtained which converges (one hopes) to a solution $\mathbf{U}_{t+\Delta t}$. The most frequently used iterative schemes, such as Newton-Raphson, modified Newton, and quasi Newton schemes, are based on a Taylor expansion of equation 3.1 about $\mathbf{U}_{t+\Delta t}$:

$$\mathbf{R}(\mathbf{U}_{t+\Delta t}) = \mathbf{R}(\mathbf{U}^{(i)}_{t+\Delta t}) + \left[\frac{\partial\mathbf{R}}{\partial\mathbf{U}}\Big|_{\mathbf{U}^{(i)}_{t+\Delta t}}\right]\left(\mathbf{U}_{t+\Delta t} - \mathbf{U}^{(i)}_{t+\Delta t}\right) \tag{3.11}$$

which for the nonlinear finite element equation, equation 3.2, can be expressed as

$$\mathbf{R}(\mathbf{U}_{t+\Delta t}) = \mathop{\mathrm{A}}_{n} \left(\mathbf{P}_n(t+\Delta t) - \mathbf{M}_n \ddot{\mathbf{U}}^{(i)}_{n_{t+\Delta t}}\right) - \mathop{\mathrm{A}}_{n} (\mathbf{M}_n \mathbf{I}'_2) \left(\mathbf{U}_{n_{t+\Delta t}} - \mathbf{U}^{(i)}_{n_{t+\Delta t}}\right)$$
$$- \mathop{\mathrm{A}}_{e} \left(\mathbf{F}_{I_e}\left(\ddot{\mathbf{U}}^{(i)}_{e_{t+\Delta t}}\right) + \mathbf{F}_{R_e}\left(\dot{\mathbf{U}}^{(i)}_{e_{t+\Delta t}}, \mathbf{U}^{(i)}_{e_{t+\Delta t}}\right) - \mathbf{P}_e(t+\Delta t)\right)$$
$$- \mathop{\mathrm{A}}_{e} (\mathbf{M}_e \mathbf{I}'_2 + \mathbf{C}_e \mathbf{I}'_1 + \mathbf{K}_e) \left(\mathbf{U}_{e_{t+\Delta t}} - \mathbf{U}^{(i)}_{e_{t+\Delta t}}\right) \quad (3.12)$$

where

$$\mathbf{M}_e = \frac{\partial \mathbf{F}_{I_i}}{\partial \ddot{\mathbf{U}}}\Big|_{\mathbf{U}^{(i)}_{e_{t+\Delta t}}} \quad (3.13)$$

$$\mathbf{C}_e = \frac{\partial \mathbf{F}_{R_i}}{\partial \dot{\mathbf{U}}}\Big|_{\mathbf{U}^{(i)}_{e_{t+\Delta t}}} \quad (3.14)$$

$$\mathbf{K}_e = \frac{\partial \mathbf{F}_{R_i}}{\partial \mathbf{U}}\Big|_{\mathbf{U}^{(i)}_{e_{t+\Delta t}}} \quad (3.15)$$

The values of $\mathbf{I}'_1$ and $\mathbf{I}'_2$ depend on the direct integration scheme. For example, in the Newmark_$\gamma\beta$ scheme

$$\mathbf{I}'_1 = \frac{\partial\, \mathbf{I}_1}{\partial \Delta \mathbf{U}} = \frac{\gamma}{\beta \Delta t}\, \mathbf{I} \quad (3.16)$$

$$\mathbf{I}'_2 = \frac{\partial\, \mathbf{I}_2}{\partial \Delta \mathbf{U}} = \frac{1}{\beta \Delta t^2}\, \mathbf{I} \quad (3.17)$$

The solution sought for the nonlinear finite element system of equations, equation 3.12, at step $t + \Delta t$ can be expressed in matrix form as

$$\mathbf{K}^* \Delta \mathbf{U}^{(i)}_n = \mathbf{P}^* \quad (3.18)$$

where

$$\mathbf{K}^* = \mathbf{K}^*_n + \mathbf{K}^*_e \quad (3.19)$$

$$\mathbf{P}^* = \mathbf{P}^*_n + \mathbf{P}^*_e \quad (3.20)$$

and

$$\mathbf{K}_n^* = \mathop{\mathsf{A}}_n \left(\mathbf{M}_n \mathbf{I}_2'\right) \tag{3.21}$$

$$\mathbf{K}_e^* = \mathop{\mathsf{A}}_e \left(\mathbf{M}_e \mathbf{I}_2' + \mathbf{C}_e \mathbf{I}_1' + \mathbf{K}_e\right) \tag{3.22}$$

$$\mathbf{P}_n^* = \mathop{\mathsf{A}}_n \left(\mathbf{P}_n(t+\Delta t) - \mathbf{M}_n \ddot{\mathbf{U}}_{n_{t+\Delta t}}^{(i)}\right) \tag{3.23}$$

$$\mathbf{P}_e^* = \mathop{\mathsf{A}}_e \left(\mathbf{P}_e\left(t+\Delta t\right) - \mathbf{F}_{I_e}\left(\ddot{\mathbf{U}}_{e_{t+\Delta t}}^{(i)}\right) - \mathbf{F}_{R_e}\left(\dot{\mathbf{U}}_{e_{t+\Delta t}}^{(i)}, \mathbf{U}_{e_{t+\Delta t}}^{(i)}\right)\right) \tag{3.24}$$

$$\Delta \mathbf{U}_n^{(i)} = \left(\mathbf{U}_{t+\Delta t}^{i+1} - \mathbf{U}_{t+\Delta t}^{(i)}\right) \tag{3.25}$$

To start the iteration scheme, trial values for $\mathbf{U}_{t+\Delta t}$, $\dot{\mathbf{U}}_{t+\Delta t}$ and $\ddot{\mathbf{U}}_{t+\Delta t}$ are required. These are obtained by assuming $\mathbf{U}_{t+\Delta t}^{(0)} = \mathbf{U}_t$. The $\dot{\mathbf{U}}_{t+\Delta t}^{(0)}$ and $\ddot{\mathbf{U}}_{t+\Delta t}^{(0)}$ can then be obtained from the operators for the integration scheme.

## 3.3 Existing Object-Oriented Approaches for Finite Element Analysis

The analysis algorithm is responsible for forming the system of equations, applying the boundary conditions, solving the system of equations, and updating the response quantities at the nodes and elements. To do this, the analysis algorithm must perform a number of tasks. For an incremental solution algorithm these tasks are:

- Assign equation numbers and map these to the nodal degrees-of-freedom. The mapping can have a significant influence on the amount of computation required to solve the matrix system of equations, equation 3.18, and on the amount of memory required to store the matrix equations.

- Form the matrix equations using contributions from elements, given by equations 3.22 and 3.24, and nodes, given by equations 3.21 and 3.23. The contribution depends on the integration scheme chosen by the analyst.

- Apply the constraints, which may involve transforming the element and nodal contributions or adding additional terms and unknowns to the matrix equations.
- Solve the matrix equations for the incremental nodal displacements.
- Update the nodal degrees-of-freedom with the appropriate response quantities.
- Determine the internal state and stresses in the elements.

A well designed object-oriented analysis framework will allow the analyst to change and experiment with new solution algorithms easily and with the minimum amount of effort. The analyst should have the ability to use different numbering schemes, different storage schemes and solution strategies for the matrix equations, different methods for dealing with the constraints, different time integration schemes, and different non-linear iteration schemes. With this in mind, a review is made of some existing finite element object-oriented software implementations, in which different analysis algorithms can be employed.

1. **Zimmermann and co-workers**: In this work an object-oriented design for the dynamic analysis using direct integration schemes is presented (Dubois-Pelerin et al., 1992; Dubois-Pelerin and Zimmermann, 1993; Zimmermann et al., 1992). To incorporate material non-linearity into the design, some of the class interfaces are modified and some of the methods rewritten (Menetrey and Zimmermann, 1993). The principal classes with respect to the analysis algorithm that are used, as shown in figure 3.1, are **Domain**, **Element**, **Node**, **LinearSystem** and **TimeStep**. The **Domain** object is responsible for creating the finite element model and for performing the analysis on this model, as demonstrated in the following main program presented in Dubois-Pelerin and Zimmermann (1993):

   ```
   main {
       Domain structure;
       structure.solveYourself();
   }
   ```

   To help perform the analysis, the **Domain** object creates two other objects: a **TimeStep** object, which can be of type **Newmark** or **Static**, and a **LinearSystem** object, which can be of type **BandSystem** or **ProfileSystem**.

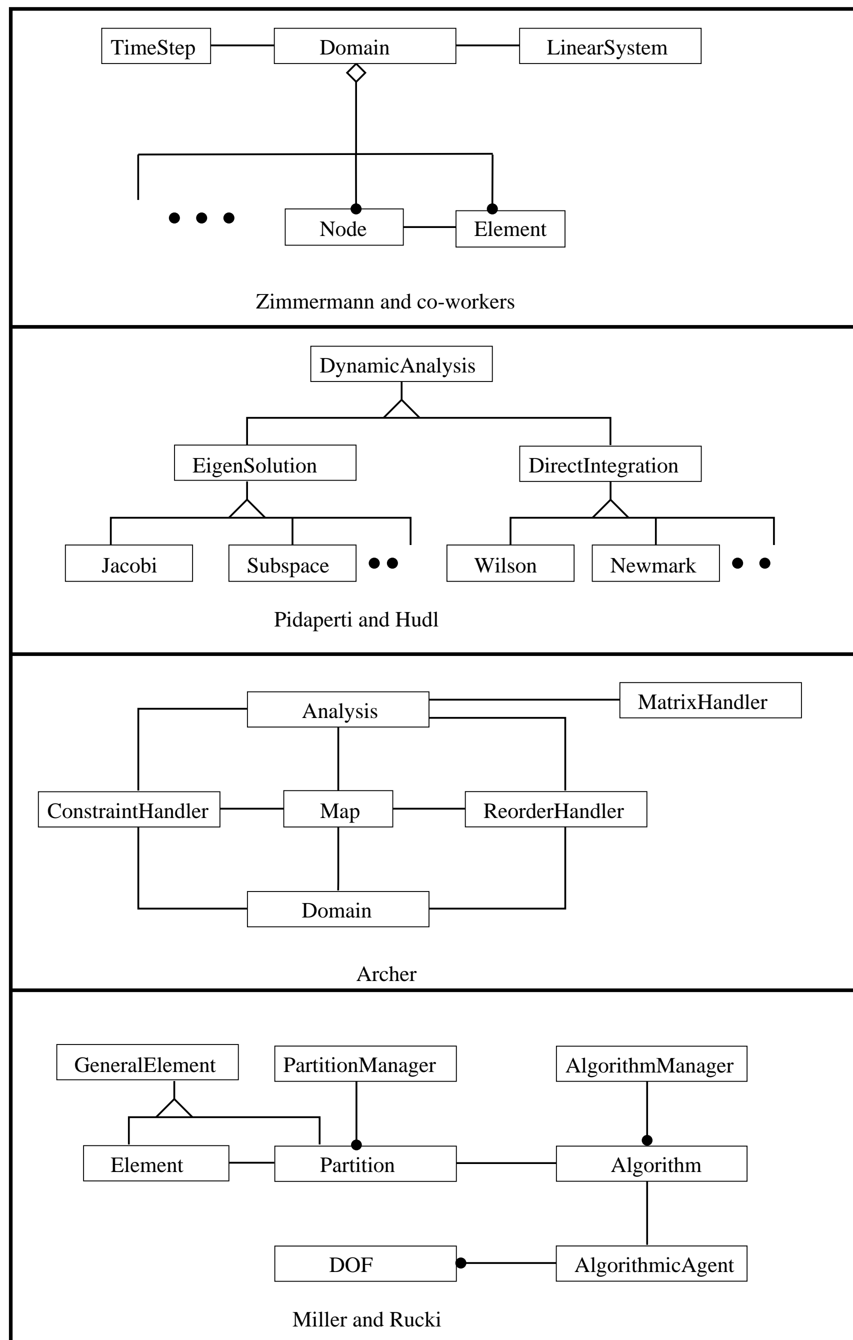



Figure 3.1: Class Diagram for Existing Analysis Frameworks

The analyst starts the analysis by invoking solveYourself() on the **Domain** object. solveYourself() causes the **Domain** object to step through the time interval invoking solveCurrentStep() on itself at each interval. When solveCurrentStep() is invoked the **Domain** object is responsible for forming and solving the system of equations at each time step, and updating the model. To facilitate this additional methods, formLHSat() and formRHSat(), are provided at the **Domain** interface. formRHSat() is a method which causes the **Domain** object to loop over all the **Node** objects invoking formRHSat(), which causes the **Node** to compute the residual and add it to the **LinearSystem** object. formLHSat() is a method which causes the **Domain** object to loop over all the **Element** objects invoking formLHSat(), which causes the **Element** to compute its tangent, and to add the tangent to the **LinearSystem** object. The forming of the **Element** tangent is done according to the **TimeStep** object, which is provided as the argument when the method is invoked.

The shortcomings with this design, in relation to an extensible and flexible platform, are the following:

(a) The analyst has no control over the types of **LinearSystem** and **TimeStep** objects that are created by the **Domain** object, as seen in the example code that was provided. The **Domain** class constructor must thus be rewritten for each different **TimeStep** and **LinearSystem** combination the analyst wishes to use.

(b) The **Domain** classes solveCurrentStep() method must be rewritten for each new iteration scheme the analyst wishes to use. For example, to incorporate material non-linearity the solveCurrentStep() method was rewritten to perform a Newton-Raphson iteration scheme (Menetrey and Zimmermann, 1993). To perform other iteration schemes, for example modified Newton or quasi-Newton schemes, would require the analyst to rewrite the method again.

(c) The mapping of equation numbers to degrees-of-freedom, which as presented is based on a first come first served approach, can result in excessive numerical computations when solving the equations. To overcome this

problem, the analyst would have to rewrite the solveCurrentStep() method, so that on its first invocation, more appropriate numbering schemes are employed to perform the mapping.

(d) The **Domain** classes constructor must be rewritten for each new **Element** subclass introduced by the analyst and for each new geometry. This means that inheritance for classes that share a common analysis algorithm, but do not share a common model building approach, cannot be used.

2. **Pidaparti and Hudl**: Pidaparti and Hudl (1993) presents a design for linear transient and eigenvalue analysis for problems with homogeneous single-point constraints. In this work, two abstract subclasses of **DynamicAnalysis** are provided, **EigenSolution** and **DirectIntegration**. For each of these, additional subclasses are provided, as shown in figure 3.1. The two base classes provide methods to read the input matrices. They also define pure virtual methods in their interface to perform the analysis. For example, the **EigenSolution** class defines computeEvec() and computeEval() as pure virtual, and the subclasses provide the implementation of these methods. For example, each subclass of **DirectIntegration** provides an implementation of the methods computeTangent(), TriangularizeTangent(), and computeResponse() which are called in the implementation of computeState().

   The shortcomings with this design, in relation to an extensible and flexible platform, are the following:

   (a) There is no code re-use between the subclasses. For example, each subclass of **DirectIntegration** provides a method to factorize the tangent matrix, when this method could have been provided by the **DirectIntegration** class.

   (b) To extend the design to include non-linear analysis the **computeState()** method would have to be rewritten for each class and for each iteration strategy used.

3. **Archer**: Archer (1996) presents five classes to allow for static and dynamic, linear and non-linear analysis. These classes are **Analysis**, **ConstraintHandler**,

**ReorderHandler**, **Map** and **MatrixHandler**. In this work an **Analysis** object is responsible for the following:

(a) Instantiating the type of **ConstraintHandler**, **ReorderHandler**, **MatrixHandler** and **Map** objects in the analysis.

(b) Performing the analysis algorithm. The steps of the analysis algorithm are performed on the invocation of the analyze() method. This method calls upon assembleK(), assembleResForce() and assembleLoad().

(c) Incorporate the constraints into the system of equations.

The **ConstraintHandler** object is responsible for providing an initial mapping between degrees-of-freedom at the **Node**s in the model and equation numbers of the analysis. It does not handle the constraints, as its name would suggest. The handling of the constraints is performed by the **Analysis** object. The **ReorderHandler** is responsible for changing the initial mapping assigned to the **ConstraintHandler**.

The **Map** is the object which stores the final mapping. The **Map** is the interface between the **Analysis** and the **Domain**, which is similar to the functionality of a **Connector** object described in Chudoba and Bittnar (1995). The **Map** has three functions:

(a) To store the mapping.

(b) To obtain stiffness, mass and damping matrices and load vectors from the **Element**s, which are provided in terms of the **Element**s local coordinate system, and apply the transformations necessary to transform them into the global coordinate system of the analysis.

(c) Given the results of the analysis, to update the response at the **Node**s.

The shortcomings with this architecture, in relation to an extensible and flexible platform, are the following:

(a) The handling of the constraints is non-uniform. For example, if the analyst uses the transformation method, the constraints are applied to **Element**

contributions before the **Analysis** receives them and so the analysis does not have to consider constraints. If using the Penalty method or Lagrange multipliers, the **Analysis** object must explicitly deal with the constraints. The class name **ConstraintHandler** is thus misleading, as it only handles constraints if using the transformation method.

(b) The design again leads to a shallow analysis hierarchy with no code reuse. To try new constraint handling methods, direct integration methods or iteration schemes requires the analyst to write a specific analysis class. For example, if given a nonlinear incremental analysis for a direct integration scheme which handles the constraints using one constraint handling method, the analyst must rewrite the constructor and analyze routines for each different constraint handling method. This rewrite must be done for each different integration scheme.

4. **Miller and Rucki**: In this work a novel approach to the analysis algorithm is presented (Rucki, 1996; Rucki and Miller, 1996). An **Algorithm** object works with the **Element**, **DOF**, **Load** and **Constraint** objects of a **Partition** to update the responses directly, there is no forming of the global system of equations. This approach allows the use of element-by-element iterative solvers, such as Jacobi and Gauss-Seidel, and DOF-by-DOF solvers. The DOF-by-DOF solvers can be both iterative, such as Gauss-Seidel, and direct, using a sparse solver. The direct sparse solvers work on the **DOF** objects in a **Partition**. To accommodate these solution strategies, the **Element** class defines methods to obtain the unbalanced load and to install the tangent stiffness coefficients at the **DOF** objects associated with an **Element**. For the sparse solver, the **DOF** class defines methods which allow **DOF** objects to connect with other **DOF** objects that they were not initially associated with.

   In this work three new classes are provided, **AlgorithmManager**, **Algorithm**, and **AlgorithmicAgent**, as shown in figure 3.1 to perform the analysis. The **AlgorithmManager** object is responsible for managing its contained **Algorithm** objects and for providing a method `updateAlgos()`, which is invoked to update the state of the model so that equilibrium is satisfied. To perform these

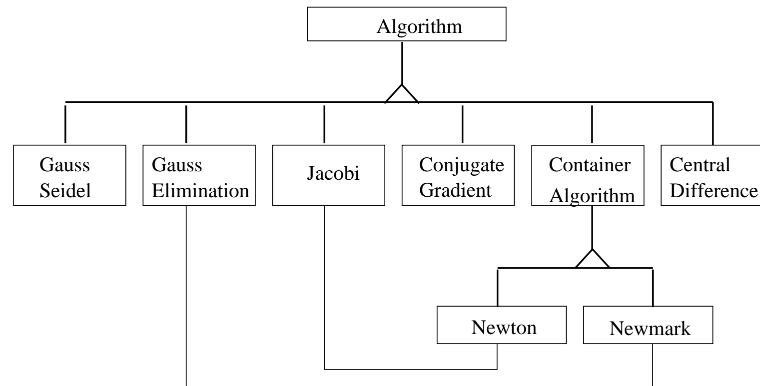



Figure 3.2: Class Diagram for Algorithmic Hierarchy in Miller and Rucki (1995)

operations, the **AlgorithmManager** will invoke updateState() on all its contained **Algorithm** objects. When this method is invoked on an **Algorithm** object, the **Algorithm** is responsible for orchestrating the update of the partition it is associated with. To do this, it will zero the unbalanced load, ask the **Load**s to apply() themselves, and will ask the **Constraint**s to modify the force vector. The **Algorithm** object will then perform the solution of the system of equations, using an element-by-element or DOF-by-DOF approach. To remove from the **DOF** objects the need to know anything about the solution algorithm, e.g. the operators of the integration schemes, **AlgorithmicAgents** are provided. The **AlgorithmicAgent** acts as an intermediary between the **Algorithm** and the **DOF** object it is associated with. Each **Algorithm** subclass will have at least one **AlgorithmicAgent** provided for it. The interface for the **AlgorithmicAgent** subclasses is unique to the **Algorithm** classes.

The shortcomings with this architecture, in relation to an extensible and flexible platform, are the following:

(a) The hierarchy presented for the **Algorithm**, which is shown in figure 3.2, mixes actual analysis algorithms, i.e. algorithms used to solve the governing nonlinear equation, equation 3.2, and numerical algorithms, i.e. algorithms used to solve the matrix equations, equation 3.18.

(b) When using the sparse solver, there is no way provided for analyst to spec-

ify the order in which the **DOF** objects are eliminated, i.e. a numbering scheme. This is required to reduce the amount of work in each updateState() when performing a direct solve on the linear system of equations.

(c) Constraints can only be enforced using the transformation method. This is because solvers work with the **Element** or **DOF** objects directly. The introduction of equations and coefficients, that a penalty or Lagrange multiplier method requires, cannot be handled.

(d) For transient analysis using a direct integration scheme, an element-by-element solution strategy can only be employed when $\mathbf{I}'_1 = \mathbf{I}'_2 = \mathbf{I}$. This is because the **Element**s are not required to know anything about each **Algorithm**, and no agents are provided to act between the **Element**s and the **Algorithm**. For example, an element-by-element solver cannot be used with the Newmark integration scheme.

In all of the above approaches, the analyst creates a single **Analysis** object to perform the analysis. The hierarchy representing the **Analysis** classes is very flat. While flat hierarchies, as pointed out by Rucki and Miller (1996), do not lead to inefficient code in terms of performance of execution they do not facilitate code-reuse, which can lead to efficiency in terms of program development time. Neither do flat hierarchies lead to modular code, which can be used to develop a library of re-usable objects, i.e. if an analyst wants to use a new analysis algorithm, the analyst must either write it from scratch or copy bits of code from other classes and hope they work together.

## 3.4 A New Object-Oriented Approach for the Analysis Algorithm

To facilitate code re-use and to provide for a design which is more flexible and extensible than those presented in the previous section, object-oriented design principles can be applied to the analysis algorithm. As discussed in section 1.2, this is first done by identifying the main tasks performed in a finite element analysis, abstracting them into separate classes, and then specifying the interface for these classes. It is important that the interfaces specified allow the classes to work together to perform the analysis and allow new classes to be introduced without the need to change existing classes. In the design presented for this research, an **Analysis** object is an aggregation, as shown in figure 3.3, of objects of the following types:

1. **SolnAlgorithm**: The solution algorithm object is responsible for orchestrating the steps performed in the analysis.

2. **AnalysisModel**: The AnalysisModel object is a container class for storing and providing access to the following types of objects:

   (a) **DOF_Group**: The **DOF_Group** objects represent the degrees-of-freedom at the **Node**s or new degrees-of-freedom introduced into the analysis to enforce the constraints.

   (b) **FE_Element**: The **FE_Element** objects represent the **Elements** in the **Domain** or they are introduced to add stiffness and/or load to the system of equations in order to enforce the constraints.

   The **FE_Element**s and **DOF_Group**s are important to the design because:

   (a) They remove from the **Node** and **Element** objects the need to worry about the mapping between degrees-of-freedoms and equation numbers. In this they provide the functionality of the **Map** (Archer, 1996) and **AlgorithmicAgent** (Rucki and Miller, 1996).

   (b) They also remove from the **Node** and **Element** class interfaces methods for forming tangent and residual vectors, that are used to form the system

of equations.

(c) The subclasses of **FE_Element** and **DOF_Group** are responsible for handling the constraints. This removes from the rest of the objects in the analysis aggregation the need to deal with the constraints.

3. **Integrator**: The **Integrator** object is responsible for defining the contributions of the **FE_Element**s and **DOF_Group**s to the system of equations and for updating the response quantities at the **DOF_Group**s with the appropriate values given the solution to the system of equations.

4. **ConstraintHandler**: The **ConstraintHandler** object is responsible for handling the constraints. It does this by creating **FE_Element**s and **DOF_Group**s of the correct type.

5. **DOF_Numberer**: The **DOF_Numberer** object is responsible for mapping equation numbers in the system of equations to the degrees-of-freedom in the **DOF_Group**s.

In the following subsections the purpose of each of these classes is outlined. Pseudo C++ code fragments are presented to demonstrate the function of some of the main classes and the interplay between these classes, when obtaining the solution of static and transient problems using the incremental solution technique.

### 3.4.1 Analysis Class

The **Analysis** object is responsible for verifying that the objects in the aggregation are of the correct type, for setting up the links required by the objects in the aggregation so that they can perform their function, for invoking start up methods on the objects, and for starting the analysis operation. The actual numerical computation performed is the responsibility of objects in the analysis aggregation. The **Analysis** class, whose interface is shown in figure 3.4, is an abstract class defining two pure virtual methods, analyze() and domainChanged(). analyze() is this method which is invoked by the analyst to perform an analysis on the **Domain** and domainChanged() is invoked to inform the **Analysis** that the **Domain** has changed, which

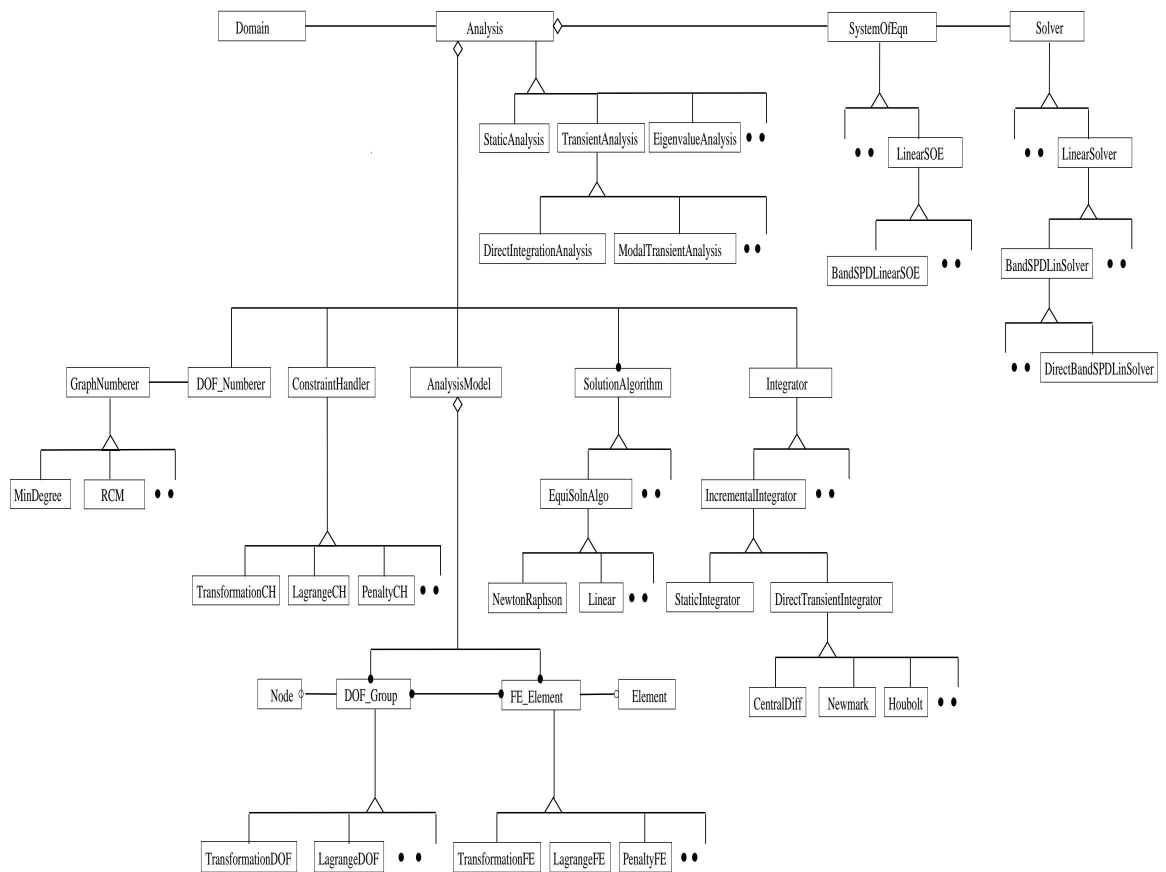



Figure 3.3: Class Diagram for New Analysis Framework

in turn is responsible for invoking the appropriate methods on the objects in the aggregation.

```
class Analysis: {
    public:
        Analysis(Domain &theDomain);
        virtual ~Analysis();

        virtual int analyze(void) =0;
        virtual int domainChanged(void) =0;
};
```

Figure 3.4: Interface for the **Analysis** Class

For the solution of static and transient problems by an incremental approach, two subclasses of **Analysis** are provided:

1. **StaticAnalysis**: The **StaticAnalysis** class is used by an analyst to perform a static analysis using an incremental solution technique. The **StaticAnalysis** object is an aggregation, as shown in figure 3.5, of objects of the following types: **ConstraintHandler**, **DOF_Numberer**, **AnalysisModel**, **LinearSOE**, **EquiSolnAlgo**, and **StaticIntegrator**. The objects in the aggregation are passed as arguments to the constructor. The **StaticAnalysis** class defines two methods: domainChanged() and analyze().

   (a) domainChanged(): The method, which is shown in figure 3.6, first invokes setLinks() on the objects in the aggregation to set up the links required by

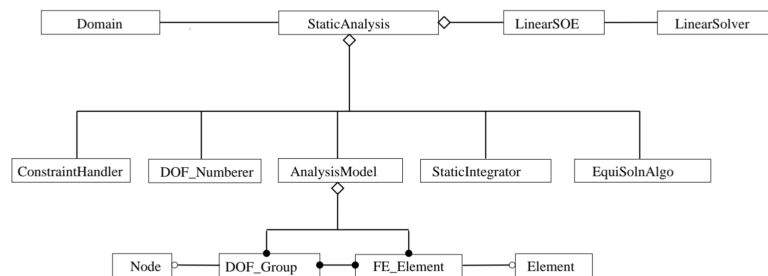



Figure 3.5: Class Diagram for a Static Analysis

these objects. It then invokes handle() on the **ConstraintHandler** object, which will create the **FE_Element** and **DOF_Group** objects, and it invokes number() on the **DOF_Numberer** object to create a mapping between degrees-of-freedom and the equation numbers. It then invokes setSize() on the **LinearSOE** object, using the **Graph** of the degree-of-freedom connectivity obtained from the **AnalysisModel**. Finally it invokes analysisModelChanged() on the EquiSolnAlgo and **StaticIntegrator** objects.

(b) analyze(): This is a method which is invoked by the analyst to perform the static analysis. The method, which is shown in figure 3.6, first invokes domainChanged() on itself. It then invokes applyLoad() on the **AnalysisModel** object to have the loads in the current loadcase apply themselves, and it invokes solveCurrentStep() on the **EquiSolnAlgo** object to perform the analysis. Finally it invokes commitDomain() on the **AnalysisModel** object to have commit() invoked on the **Domain** object.

2. **DirectIntegrationAnalysis**: The **DirectIntegrationAnalysis** class is used by an analyst to perform a transient analysis using an incremental solution technique. The **DirectIntegrationAnalysis** object is an aggregation, as shown in figure 3.7, of objects of the following types: **ConstraintHandler**, **AnalysisModel**, **LinearSOE**, **DOF_Numberer**, **EquiSolnAlgo**, and **TransientIntegrator**. The difference between a **DirectIntegrationAnalysis** object and a **StaticAnalysis** object is that the **DirectIntegrationAnalysis** object has a **TransientIntegrator**, as opposed to a **StaticIntegrator** object in a **StaticAnalysis**. The **DirectIntegrationAnalysis** class, whose interface is as shown in figure 3.8, provides the following methods:

   (a) domainChanged(): This is a method which performs the same operations as the **StaticAnalysis** classes domainChanged() method.

   (b) setTimeVar(): This is a method which is used by the analyst to set the start time, the finish time, and $\Delta t$ for an analysis. Subclasses of **DirectIntegrationAnalysis** can be introduced for adaptive schemes.

```
StaticAnalysis::domainChanged{
{
    // first set up the links needed by the elements in the
    // aggregation
    theConstraintHandler->setLinks(theDomain,theModel,theIntegrator);
    theDOF_Numberer->setLinks(theModel);
    theStaticIntegrator->setLinks(theModel,theSOE);
    theAlgorithm->setLinks(*this,theModel,theIntegrator,theSOE);
    theDomain->setAnalysis(*this);

    // now we invoke handle() on the constraint handler which
    // causes the creation of FE_Element and DOF_Group objects
    // and their addition to the AnalysisModel.
    theConstraintHandler->handle();

    // we now invoke number() on the numberer which causes
    // equation numbers to be assigned to all the DOFs in the
    // AnalysisModel.
    theDOF_Numberer->numberDOF();

    // we now invoke setSize() on the LinearSOE which
    // causes that object to determine its size
    theSysOfEqn->setSize(theAnalysisModel->getDOFGraph());

    // finally we ininvoke analysisMoedlChanged
    // on the Integrator and SolutionAlgo objects
    theStaticIntegrator->analysisModelChanged();
    theAlgorithm->analysisModelChanged();
}

StaticAnalysis::analyze{
    if (theDomain->hasDomainChanged() == true)
        this->domainChanged();
    thyeAnalysisModel->applyLoadDomain(0.0)
    theEquiSolnAlgo->solveCurrentStep()
    theAnalysisModel->commitDomain()
}
```

Figure 3.6: Pseudo-Code for Selected Methods for the **StaticAnalysis** Class

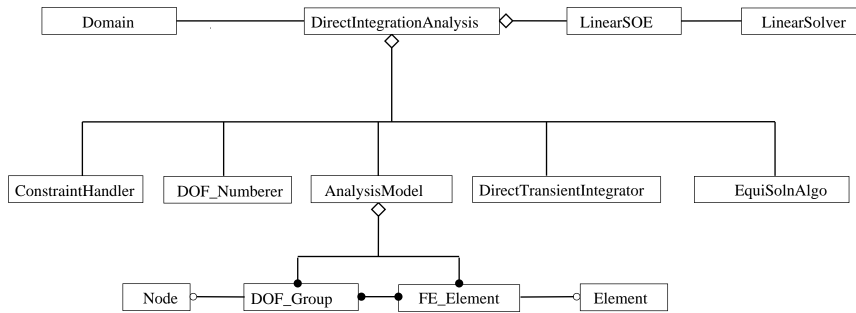



Figure 3.7: Class Diagram for a Transient Analysis using a DirectIntegration Scheme

(c) setAlgorithm(): This is a method which can be used by the analyst to change the **TransientIntegrator** object between analysis.

(d) analyze(): This is a method to perform the transient analysis. The analyze() method for the **DirectIntegrationAnalysis** class, which is shown in figure 3.9, first checks to see if the **Domain** has changed and it then incrementally steps through the time interval performing an equilibrium analysis at each time step.

```
class DirectIntegrationAnalysis: public TransientAnalysis {
    public:
        DirectIntegrationAnalysis(Domain &theDomain,
                                  ConstraintHandler &theHandler,
                                  DOF_Numberer &theNumberer,
                                  AnalysisModel &theModel,
                                  StaticEquiSolnAlgo &theSolnAlgo,
                                  LinearSOE &theSOE,
                                  TransientIntegrator &theIntegrator);
        virtual  DirectIntegrationAnalysis();

        virtual int domainChanged(void);
        virtual int analyze(void);
        virtual void setTimeVar(double tStart, double tFinish, double Δt);
        virtual void setIntegrator(TransientIntegrator &newIntegrator);
}
```

Figure 3.8: Interface for the **DirectIntegrationAnalysis** Class

```
DirectIntegrationAnalysis::analyze{
      if (theDomain->hasDomainChanged() == true)
            this->domainChanged();
      time = tStart
      while (time < tFinal) {
            theIntegrator->newStep(Δt)
            theAnalysisModel->applyLoadDomain(time)
            theEquiSolnAlgo->solveCurrentStep()
            theAnalysisModel->commitDomain()
            time += Δt
      }
}
```

Figure 3.9: Pseudo-Code for the **DirectIntegrationAnalysis** Classes Interface and analyze Method

### 3.4.2 SolutionAlgorithm Class

The **SolutionAlgorithm** object orchestrates the steps in the analysis. To do this it specifies the sequence of operations that are invoked on the different objects in the analysis aggregation. The **SolutionAlgorithm** class is an abstract class whose interface, which is shown in figure 3.10, defines the methods solveCurrentStep() and analysisModelChanged().

```
class SolutionAlgorithm: {
      public:
            SolutionAlgorithm();
            virtual ~SolutionAlgorithm();

            // pure virtual functions to be implemented by subclasses
            virtual int solveCurrentStep(void) =0;
            virtual int analysisModelChanged(void) =0;

};
```

Figure 3.10: Interface for the **SolutionAlgorithm** Class

For the solution of static and transient problems using the incremental approach, the **SolutionAlgorithm** object will be some subclass of **EquiSolnAlgo**. The **EquiSolnAlgo** class, whose interface is shown in figure 3.11, defines three methods:

```
class EquiSolnAlgo: public SolutionAlgorithm {
    public:
        EquiSolnAlgo();
        virtual ~EquiSolnAlgo();

        void setLinks(StaticAnalysis &theAnalysis,
                      AnalysisModel &theModel,
                      IncrementalIntegrator &theIntegrator,
                      LinearSOE &theSOE);

        // pure virtual functions to be implemented by subclasses
        virtual int solveCurrentStep(void) =0;
        virtual int analysisModelChanged(void);

    protected:
        StaticAnalysis *theAnalysis;
        AnalysisModel *theModel;
        IncrementalIntegrator *theIntegrator;
        LinearSOE *theSysOfEqn;
};
```

Figure 3.11: Interface for the **EquiSolnAlgo** Class

1. setLinks(): This method is called so that the **EquiSolnAlgo** object can learn of the **AnalysisModel**, **IncrementalIntegrator** and **LinearSOE** objects on which it will invoke operations when solveCurrentStep() is invoked.

2. analysisModelChanged(): This method is called so that the **EquiSolnAlgo** object can be informed that the analysis model has changed. The method for this class does nothing. It is declared as virtual to allow subclasses to provide their own implementation.

3. solveCurrentStep(): This is a pure virtual method, which is invoked on the object by the **Analysis** object. When invoked, the object will perform an equilibrium analysis on the **Domain** object to find an equilibrium state, given the current state of the **Domain**. The steps taken to obtain the equilibrium state depend on the implementation of this method provided by the subclass of **EquiSolnAlgo** chosen by the analyst. Examples of subclasses, as shown in figure 3.3, are **Linear** and **NewtonRaphson**. For example, if the analyst creates an object of

type **Linear** the **EquiSolnAlgo** object will invoke methods on the **Integrator** object to form the linear system of equations in the **LinearSOE** object. It will then invoke methods on the **LinearSOE** object to solve the system of equations and obtain the solution. Finally it will invoke updateIncr() on the **Integrator** object which will cause that object to update the **DOF_Group** objects with the appropriate response quantities. This is shown in figure 3.12.

```
Linear::solveCurrentStep{
    // form the system of equations
    theIntegrator->formUnbalance()
    theIntegrator->formTangent()

    // solve the system of equations
    theLinearSOE->solveX()
    Vector &U = theLinearSOE->getX()

    // update
    theIntegrator->updateIncr(U)
}
```

Figure 3.12: Pseudo-Code for the **Linear** Classes solveCurrentStep Method

If the analyst instead creates a **EquiSolnAlgo** object of type **NewtonRaphson**, a Newton-Raphson scheme would be used to obtain an equilibrium solution. The object would iterate through the steps outlined for the **Linear** class above until convergence is obtained, as shown in figure 3.13.

### 3.4.3 Integrator Class

The **Integrator** object is responsible for providing methods for forming the system of equations, for defining the contributions of the **FE_Element** and **DOF_Group** objects to the system of equations, and for updating the response at the **DOF_Group** objects. The **Integrator** class is an abstract base class.

For the solution of static and transient problems by an incremental approach the **Integrator** object will be a subclass of **IncrementalIntegrator**. The **IncrementalIntegrator** class, whose interface is shown in figure 3.14, is an abstract class. It provides for the following:

```
NewtonRaphson::solveCurrentStep{
    theIntegrator->formUnbalance()
    // iterate until convergence
    while (theLinearSOE->normRHS() > TOL) {
        theIntegrator->formTangent()
        theLinearSOE->solveX()
        Vector &ΔU = theLinearSOE->getX()
        theIntegrator->updateIncr(ΔU)
        theIntegrator->formUnbalance()
    }
}
```

Figure 3.13: Pseudo-Code for the **NewtonRaphson** Classes solveCurrentStep Method

1. The setLinks() method, which is invoked by the **Analysis** object to make the **IncrementalIntegrator** object aware of the **AnalysisModel** and **LinearSOE** objects in the aggregation.

2. The **analysisModelChanged()** method, which is invoked to inform the **IncrementalIntegrator** object that the analysis model has changed. This is provided so that subclasses can set any **Vector** or other objects used. The method for the **IncrementalIntegrator** class performs no operations.

3. Methods for forming the linear system of equations. These methods are formTangent(), formUnbalance(), formElementResidual(), and formNodalUnbalance(). These methods, which are shown in figure 3.15, iterate over the **FE_Element**s to form the tangent and element residual, and iterate over the **DOF_Group**s to form the nodal unbalance.

4. Methods which define each elements and nodes contribution to to the system of equations and how the system of equations are set up. For example when using the incremental displacement approach the choice of integration scheme defines each elements contribution to $\mathbf{K}_e^*$, equation 3.22, and $\mathbf{P}_e^*$, equation 3.24, and each nodes contribution to $\mathbf{K}_n^*$, equation 3.21, and $\mathbf{P}_n^*$, equation 3.23. These methods, formEleTangent(), formEleResidual(), formNodTangent() and formNodUnbalance(), are all declared as pure virtual.

```
class IncrementalIntegrator : public Integrator {
    public:
        IncrementalIntegrator();
        virtual ~IncrementalIntegrator();

        virtual void setLinks(AnalysisModel &theModel,
                              LinearSOE &theSysOfEqn);
        virtual analysisModelChanged(void);

        // methods to set up the system of equations
        virtual int formTangent(void);
        virtual int formUnbalance(void);
        virtual int formNodalUnbalance(void);
        virtual int formElementResidual(void);

        // methods to define what the FE_Ele and DOF_Groups add to
        // the system of equations, all are pure virtual.
        virtual int formEleTangent(FE_Element *theEle) =0;
        virtual int formEleResidual(FE_Element *theEle) =0;
        virtual int formNodTangent(DOF_Group *theDof) =0;
        virtual int formNodUnbalance(DOF_Group *theDof) =0;

        // methods to relate the solution of the system of equations
        // to the degrees-of-freedom in the DOF_Groups
        virtual int update(const Vector &U) =0;
        virtual int updateIncr(const Vector &ΔU) =0;

    protected:
        LinearSOE *mySOE;
        AnalysisModel *myModel;
};
```

Figure 3.14: Interface for the **IncrementalIntegrator** Class

```
IncrementalIntegrator::formTangent {
    FE_EleIter theEles = theAnalysisModel->getFEs()
    theLinearSOE->zeroA()
    while ((feElePtr = theEles()) ≠ 0) {
        feElePtr->formTangent(theIntegrator)
        theLinearSOE->addA(feElePtr->getTangent(), feElePtr->getID())
    }
}

IncrementalIntegrator::formUnbalance{
    theLinearSOE->zeroB()
    this->fromElementResidual()
    this->fromNodalUnbalance()


IncrementalIntegrator::formElementResidual {
    FE_EleIter theEles = theAnalysisModel->getFEs()
    while ((feElePtr = theEles()) ≠ 0) {
        feElePtr->formResidual(theIntegrator)
        theLinearSOE->addB(feElePtr->getResidual(), feElePtr->getID())
    }
}

IncrementalIntegrator::formNodalUnbalance{
    DOF_Iter theDofs = theAnalysisModel->getDOFs()
    while ((dofPtr = theDOFs()) ≠ 0) {
        dofPtr->formUnbalance(theIntegrator)
        theLinearSOE->addB(dofPtr->getUnbalance(), dofPtr->getID())
    }
}
```

Figure 3.15: Pseudo-Code for Selected Methods for the **IncrementalIntegrator** Class

5. Two methods to relate the solution of the system of equations to the degrees-of-freedom at the nodes. These methods, update() and updateIncr(), are both declared as pure virtual.

The **IncrementalIntegrator** class is an abstract class. Subclasses are provided which provide implementations of the pure virtual methods. Two subclasses of **IncrementalIntegrator** are provided for static and transient analysis:

1. **StaticIntegrator**: The **StaticIntegrator** class is used by the analyst when performing static analysis. The class provides an implementation of the pure virtual methods declared in the superclass, as the interface in figure 3.16 shows.

```
class StaticIntegrator : public IncrementalIntegrator {
    public:
        StaticIntegrator();
        virtual ~StaticIntegrator();

        // methods to define what the FE_Ele and DOF_Groups add to
        // the system of equations.
        virtual int formEleTangent(FE_Element *theEle);
        virtual int formEleResidual(FE_Element *theEle);
        virtual int formNodTangent(DOF_Group *theDof);
        virtual int formNodUnbalance(DOF_Group *theDof);
        virtual int update(const Vector &U);
        virtual int updateIncr(const Vector &ΔU);
};
```

Figure 3.16: Interface for the **StaticIntegrator** Class

The **StaticIntegrator** class inherits the methods formTangent(), formElementResidual(), and formNodalUnbalance() from the IncrementalIntegrator class. The StaticIntegrator class defines the methods update(), updateIncr(), formEleTangent(), formEleResidual(), formNodTangent(), and formNodUnbalance(). For example, the methods formEleTang(), update() and updateIncr() are as shown in figure 3.17,

2. **DirectTransientIntegrator**: The **DirectTransientIntegrator** class is used by the analyst when performing transient analysis. The class, whose interface

```
StaticIntegrator::formEleTang(FE_Element *theEle) {
      theEle->zeroTang()
      theEle->addKtoTang()
}

StaticIntegrator::update(Vector &U) {
      DOF_Iter theDofs = theAnalysisModel->getDOFs()
      while ((dofPtr = theDOFs()) ≠ 0)
            dofPtr->setNodeDisp(U)
}

StaticIntegrator::updateIncr(Vector &ΔU) {
      DOF_Iter theDofs = theAnalysisModel->getDOFs()
      while ((dofPtr = theDOFs()) ≠ 0)
            dofPtr->setNodeIncrDisp(ΔU)
}
```

Figure 3.17: Pseudo-Code for Selected Methods for the **StaticIntegrator** Class

is shown in figure 3.18, is an abstract class. Examples of subclasses, as shown in figure 3.3 are **Newmark**, **CentralDifference** and **Houbolt**.

```
class DirectTransientIntegrator : public IncrementalIntegrator {
      public:
            DirectTransientIntegrator();
            virtual ~DirectTransientIntegrator();

            // methods to set up the system of equations
            virtual int formTangent(void);
            virtual int newStep(double Δt) =0;
};
```

Figure 3.18: Interface for the **DirectTransientIntegrator** Class

The **DirectTransientIntegrator** class inherits the methods formElementResidual() and formNodalUnbalance() from the IncrementalIntegrator class, but it redefines the formTangent() method. This is because in transient analysis the nodes may have masses which are added to the tangent matrix. The method, which is shown in figure 3.19, performs the formTangent() method defined for the **IncrementalIntegrator** class and then loops over the **DOF_Group** objects

adding their contribution as well.

```
DirectTransientIntegrator::formTangent {
      // invoke IncrementalIntegrators formTangent method
      this->IncrementalIntegrator::formTangent()

      // now loop over the DOF_Groups
      DOF_Iter theDofs = theAnalysisModel->getDOFs()
      theLinearSOE->zeroA()
      while ((dofPtr = theDOFs()) ≠ 0) {
            dofPtr->formTangent(theIntegrator)
            theLinearSOE->addA(dofPtr->getTang(), dofPtr->getID())
      }
}
```

Figure 3.19: Pseudo-Code for the Methods of the **DirectTransientIntegrator** Class

The subclasses of **DirectTransientIntegrator** are responsible for individually defining the methods update(), updateIncr(),formEleTangent(), formEleResidual(), formEleResidual(), and formEleResidual(). For example, the updateIncr() and formEleTang() methods for the **Newmark** class are shown in figure 3.20.

### 3.4.4 AnalysisModel Class

The **AnalysisModel** object is responsible for holding and providing access to the **FE_Element** and **DOF_Group** objects. The **AnalysisModel** class, whose interface is shown in figure 3.21, provides for the following:

1. Methods are provided to allow the **ConstraintHandler** object to add the **FE_Element** and **DOF_Group** objects.

2. Methods are provided which allows the other objects in the analysis aggregation access to the **FE_Element** and **DOF_Group** objects. This access is provided in the form of iterators. **DOF_Group** objects can also be accessed individually using their unique identifier.

3. Methods are provided to return connectivity information, for both the individual degrees-of-freedom and the **DOF_Group**s. The information returned is in

```
class Newmark : public DirectTransientIntegrator {
    public:
        Newmark(double γ, double β);
        virtual ~Newmark();

        virtual int formEleTangent(FE_Element *theEle);
        virtual int formEleResidual(FE_Element *theEle);
        virtual int formNodUnbalance(DOF_Group *theDof);
        virtual int update(const Vector & U);
        virtual int updateIncr(const Vector &ΔU);

    protected:
};

Newmark::formEleTang(FE_Element *theEle) {
    theEle->zeroTang()
    theEle->addKtoTang()
    theEle->addMtoTang(γ/(βΔt))
    theEle->addCtoTang(1/(βΔt²))
}

Newmark::updateIncr(Vector & ΔU) {
    ΔU̇ = γ/(βΔt) ΔU_t − (γ/β) U̇_t + Δt (1 − γ/(2β)) Ü_t
    ΔÜ = 1/(βΔt²) ΔU − (β/Δt) U̇_t − (1/(2β)) Ü_t
    DOF_Iter theDofs = theAnalysisModel->getDOFs()
    while ((dofPtr = theDOFs()) ≠ 0) {
        dofPtr->setIncDisp(ΔU)
        dofPtr->setIncVel(ΔU̇)
        dofPtr->setIncAccel(ΔÜ)
    }
}
```

Figure 3.20: Interface and Pseudo-Code for Selected Methods of the **Newmark** Class

```
class AnalysisModel {
    public:
        AnalysisModel(Domain &theDomain);
        virtual ~AnalysisModel();

        // method to set the link to the domain
        virtual void setLinks(Domain &theDomain);

        // methods to populate the AnalysisModel
        virtual bool addFE_Element(FE_Element *theFE_Ele) =0;
        virtual bool addDOF_Group(DOF_Group *theDOF_Grp) =0;

        // methods to get the components of the AnalysisModel
        virtual FE_EleIter &getFEs() =0;
        virtual DOF_GrpIter &getDOFs() =0;
        virtual DOF_Group *getDOF_GroupPtr(int tag);

        // methods to get dof connectivities
        virtual Graph &getDOFGraph(void);
        virtual Graph &getDOFGroupGraph(void);

        // methods which trigger operations in the Domain
        virtual void applyLoadDomain(double time);
        virtual void commitDomain(void);
};
```

Figure 3.21: Interface for the **AnalysisModel** Class

the form of a **Graph** object. The degree-of-freedom graph, which is a graph for those degrees-of-freedom which have been assigned an equation number, is needed to determine the size and sparsity of the system of equations. The **DOF_Group** graph is used by the **DOF_Numberer** object to assign equation numbers to the individual degrees-of-freedom.

4. Methods are provided which trigger methods in the **Domain** object, on which the analysis is being performed. These methods include commitDomain(), which invokes commit() on the **Domain**, and applyLoadDomain(), which invokes applyLoad() on the **Domain**.

5. The method setLinks() is provided, which is called by the **Analysis** object to inform the object which **Domain** is being analyzed.

There are no load type or constraint type objects in an **Analysis** model. There are no load type objects because the **Load** objects apply themselves to the **Element** or **Node** objects in the call to applyLoadDomain(). There are no constraint type objects because the **FE_Element** and **DOF_Group** objects introduce the constraints into the system of equations.

### 3.4.5 DOF_Group Class

Each **DOF_Group** object in the **AnalysisModel** represents the degrees-of- freedom at a **Node** object in the analysis or represent new degrees-of-freedom introduced into the analysis by the **ConstraintHandler** object to enforce the constraints, such as when using Lagrange multipliers. The **DOF_Group** class, whose interface is shown in figure 3.22, provides for the following:

1. The **DOF_Group** objects are responsible for keeping track of the mapping between the degrees-of-freedom and the equation numbers. Methods are provided to set and retrieve this mapping information.

2. Methods are also provided to return the number of degrees-of-freedom represented by the **DOF_Group** object and also information about the number of

```
class DOF_Group {
    public:
        DOF_Group(int tag, Node *myNode);
        DOF_Group(int tag, int nDOF);
        virtual ~DOF_Group();

        // methods to set/obtain mapping information
        virtual void setID(int, int);
        virtual const ID &getID(void) const;

        // methods for obtaing info about the DOFs
        virtual int getTag(void) const;
        virtual int getNumDOF(void) const;
        virtual int getNumFreeDOF(void) const;
        virtual int getNumConstrainedDOF(void) const;

        // methods to form the tangent
        virtual const Matrix &getTangent(void);
        virtual void formTangent(Integrator *theIntegrator);
        virtual void zeroTangent(void);
        virtual void addMtoTang(double fact = 1.0);

        // methods to form the unbalance
        virtual const Vector &getUnbalance(void) const;
        virtual void formUnbalance(Integrator *theIntegrator);
        virtual void zeroUnbalance(void);
        virtual void addPtoUnbalance(double fact = 1.0);

        // methods to update the trial responses at the nodes
        virtual void setNodeDisp(const Vector &U);
        virtual void setNodeVel(const Vector &U̇);
        virtual void setNodeAccel(const Vector &Ü);
        virtual void setNodeIncrDisp(const Vector &ΔU);
        virtual void setNodeIncrVel(const Vector &ΔU̇);
        virtual void setNodeIncrAccel(const Vector &ΔÜ);
};
```

Figure 3.22: Interface for the **DOF_Group** Class

degrees-of-freedom represented by the object that have not been assigned equation numbers.

3. Methods are provided which allow the **Integrator** object to form a nodal tangent matrix, which is represented by equation 3.21 for an incremental solution. A method is also provided to return this matrix.

4. Method are provided which allow the **Integrator** object to form the nodal unbalance, which is represented by equation 3.23 for an incremental solution. A method is also provided to return this vector.

5. Methods are provided which will update the trial response quantities at the **Node** objects. The methods can either set the trial response, e.g. setNodeDisp(), or increment the trial response, e.g. setNodeIncrDisp(). The arguments to all these methods are **Vector** objects of size equal to the size of the system of equations. The **DOF_Group** objects, using the mapping information they hold, are able to extract the appropriate quantities from the **Vector** and update the **Node** with these values.

### 3.4.6 FE_Element Class

Each **FE_Element** object is associated with an **Element** in the **Domain** or is introduced due to some constraint, to add stiffness and/or load to the system of equations, such as when using penalty or Lagrange methods to enforce the constraints. The **FE_Element** class, whose interface is shown in figure 3.23, provides for the following:

1. The **FE_Element** object is responsible for determining the mapping between its global degrees-of-freedom and the equation numbers. The objects can determine this mapping from the **DOF_Group** objects associated with the **Node** objects. Methods are provided to instruct the **FE_Element** to determine this mapping and to return the mapping.

2. A method getDOFtags() is provided to return the identifiers of the **DOF_Group**s associated with an **FE_Element**. This information is needed to build the

```
class FE_Element {
   public:

      FE_Element(Element *);
      FE_Element(int numDOF_Group, int nDOF);
      virtual ~FE_Element();

      virtual const ID &getDOFtags(void) const;
      virtual void setID(void);
      virtual const ID &getID(void) const;

      // methods to form the tangent
      virtual const Matrix &getTangent(void);
      virtual void formTangent(Integrator *theIntegrator);
      virtual void zeroTangent(void);
      virtual void addKtoTang(double fact = 1.0);
      virtual void addCtoTang(double fact = 1.0);
      virtual void addMtoTang(double fact = 1.0);

      // methods to form the residual
      virtual const Vector &getResidual(void);
      virtual void formResidual(Integrator *theIntegrator);
      virtual void zeroResidual(void);
      virtual void addRtoResidual(double fact = 1.0);
      virtual void addK_Force(const Vector &U, double fact = 1.0);
      virtual void addD_Force(const Vector &U̇, double fact = 1.0);
      virtual void addM_Force(const Vector &Ü, double fact = 1.0);

      // methods to obtain forces for ele-by-ele solutions
      virtual const Vector &getTangForce(const Vector &x, double fact = 1.0);
      virtual const Vector &getK_Force(const Vector &U, double fcat = 1.0);
      virtual const Vector &getD_Force(const Vector &U̇, double fcat = 1.0);
      virtual const Vector &getM_Force(const Vector &Ü, double fcat = 1.0);
};
```

Figure 3.23: Interface for the **FE_Element** Class

degree-of-freedom graph and the **DOF_Group** graph.

3. Methods are provided which will allow an **Integrator** object to instruct the **FE_Element** objects how to form their contributions to the system of equations, which for the incremental solution strategy are given by equations 3.22 and 3.24. These contributions are determined with the help of the **Integrator** object, as **FE_Element** objects are not expected to be aware of all the possible direct integration schemes the analyst might choose to use.

4. Methods getTangent() and getResidual() are provided to obtain these contributions.

5. Methods to obtain the product of the elements tangent, stiffness, mass and damping matrices with a vector are provided for element by element solution strategies.

Examples of subclasses of **FE_Element** are **LagrangeFE_Element**, **PenaltyFE_Element** and **TransformationFE_Element**. Objects of these classes enforce the constraints. The **FE_Element** objects are created by the **ConstraintHandler** object.

### 3.4.7 ConstraintHandler Class

The **ConstraintHandler** object is responsible for creating the **DOF_Group** and **FE_Element** objects, and adding them to the **AnalysisModel**. The type of **DOF_Group** and **FE_Element** objects created depends on the **ConstraintHandler**. The **ConstraintHandler** is also responsible for assigning an initial mapping of degrees-of-freedom to equation numbers.

The **ConstraintHandler** class, whose interface is shown in figure 3.24, provides two methods: setLinks() which is invoked by the **Analysis** object to allow the **ConstraintHandler** to set pointers to the **Domain** and **AnalysisModel** objects, and handle(). The handle() method instructs the **ConstraintHandler** object to create the **FE_Element** and **DOF_Group** objects, add these objects to the **AnalysisModel**,

and assign an initial mapping between the degrees-of-freedom and the equation numbers. The **ConstraintHandler** class is an abstract class. Examples of subclasses the analyst may use to enforce the constraints are **Penalty**, **Lagrange** and **Transformation**.

```
class ConstraintHandler {
    public:
        ConstraintHandler();
        virtual ~ConstraintHandler();

        void setLinks(Domain &theDomain,
                      AnalysisModel &theModel);
        virtual int handle(const ID *nodesNumberedLast =0) =0;
};
```

Figure 3.24: Interface for the **ConstraintHandler** Class

### 3.4.8 DOF_Numberer Class

The **DOF_Numberer** object is responsible for assigning equation numbers to the degrees-of-freedom in each **DOF_Group** object. It is also responsible for getting the **FE_Element** objects to determine the mapping of their local degrees-of-freedom to the equation numbers. The **DOF_Numberer** class, whose interface is shown in figure 3.25, provides two methods: setLinks(), which is invoked by the **Analysis** object to inform the object of the **AnalysisModel** in the analysis aggregation, and numberDOF(). The numberDOF() method uses the **GraphNumberer** object passed in the constructor to number the vertices in the **DOF_Group** graph. Examples of **GraphNumberer** subclasses that could be used by the analyst, as shown in figure 3.3, are **MinDegree** and **RCM**, which implement the minimum degree (Tinney and Walker, 1967) and reverse Cuthill-McKee (George, 1971) numbering schemes. Based on the vertex numbering, the **DOF_Numberer** will then go through each individual **DOF_Group** object in the **AnalysisModel** and assign equation numbers to the degrees-of-freedom. The **DOF_Numberer** will then ask each **FE_Element** object to determine its mapping, based on the mapping now at the **DOF_Group**

objects.

```
class DOF_Numberer {
	public:
		DOF_Numberer();
		DOF_Numberer(GraphNumberer &theGraphNumberer);
		virtual ~DOF_Numberer();

		virtual void setLinks(AnalysisModel &theModel);
		virtual int numberDOF(void);
};
```

Figure 3.25: Interface for the **DOF_Numberer** Class

### 3.4.9 SystemOfEqn and Solver Classes

The **SystemOfEqn** object is responsible for storing the systems of equations used in the analysis. The **Solver** object is responsible for performing the numerical operations on the **SystemOfEqn** object. The **SystemOfEqn** and **Solver** classes are abstract base classes. For the solution of static and transient problems using an incremental solution strategy two subclasses are defined: **LinearSOE** and **LinearSolver**.

1. **LinearSOE**: The **LinearSOE** object is responsible for storing linear system of equations. For this research the linear system of equations stored by a **LinearSOE** object is of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a matrix, and $\mathbf{x}$ and $\mathbf{b}$ are vectors. The **LinearSOE** class, whose interface is shown in figure 3.26, provides the following:

   (a) setSize(): This is a method to allow the **LinearSOE** object to determine its storage requirements and sparsity pattern. A **Graph** object is used to supply this information. The vertices of the **Graph** object passed in a structural analysis are labeled based on the ordering of the degrees-of-freedom which has been determined by the **DOF_Numberer**. The **LinearSOE** object may internally renumber the equations in an attempt

to improve on the numbering scheme. This may be done provided the results are returned in the original ordering scheme and that a domain decomposition method, which will be discussed in chapter 4, is not being employed. The use of the **Graph** object, to provide the information on the connectivity of the matrix, is important, as it allows the analyst to specify the type of **LinearSOE** to use in the analysis. Without this the **LinearSOE** objects would have to be created inside the **Analysis** object, with the correct arguments for that particular subclass of **LinearSOE** provided by the **Analysis** object.

(b) Methods are provided to zero out and to add to **A** and **b**.

(c) setX(): This is a method to set the values of **x**. This is needed, for example, when using an iterative approach to solving the equations and an approximate solution is known.

(d) The constructor takes as an argument the **Solver** object. A method, setSolver(), is also provided to allow the analyst to change the **Solver** during the course of an analysis.

(e) Methods are provided to solve the system of equations and return the computed result.

(f) getNumEqn: This is a method which will return the number of equations in the system.

The **LinearSOE** class is an abstract class. Examples of subclasses which are provided are: **BandSPDLinearSOE**, **SparseSPDLinearSOE**, **BandGeneralLinearSOE**, and **EleByEleLinearSOE**. The **LinearSOE** objects do not have to actually store the components of the system. For example, an **EleByEleSOE** does not store the **A** matrix and no operations are performed by the object on invocations of the addA(), it does however provide an **EleByEleSolver** object with access to the **FE_Elements**.

2. **LinearSolver**: The **LinearSOESOlver** object is responsible for solving the system of equations stored in a **LinearSOE** object. The **LinearSolver** class

```
class LinearSOE : public SystemOfEqn {
    public:
        LinearSOE(LinearSolver &theSolver) ;
        virtual ~LinearSOE() ;

        virtual int setSize(Graph &theDOF_Graph);
        virtual int getNumEqn(void) const =0;

        virtual int addA(const Matrix &, const ID &, double fact = 1.0) =0;
        virtual int addB(const Vector &, const ID &, double fact = 1.0) =0;

        virtual void zeroA(void) =0;
        virtual void zeroB(void) =0;

        virtual void setX(int loc, double value) =0;
        virtual int setSolver(LinearSolver &newSolver);

        virtual int solve(void) = 0;
        virtual const Vector &getX(void) = 0;
        virtual const Vector &getB(void) = 0;
        virtual double normRHS(void) = 0;
};
```

Figure 3.26: Interface for the **LinearSOE** Class

is an abstract class. At least one subclass is provided for each **LinearSOE** subclass. The **LinearSolver** class, whose interface is shown in figure 3.27, define one method analyze(). While generic solvers could have been developed, i.e. **GaussianElimination**, **ConjugateGradient**, etc., specific solvers are written for each **LinearSOE** subclass. This is done to improve performance, which can be achieved by taking into account sparsity of the equations and allowing the solvers to work directly on the data without needing to go through an interface.

```
class LinearSolver: public Solver {
      public:
            Solver();
            virtual ~Solver();

            virtual int solve(void) = 0;
};
```

Figure 3.27: Interface for the **LinearSolver** Class

The system of equations is solved when the solve() routine is invoked. The **LinearSOE** and **LinearSolver** are purely numerical objects, that is their interface has no finite element specific arguments. This allows for the use of outside libraries and hence does not require the analyst to write **LinearSolver** subclasses. For example, the solver for the **BandSPDLinSOESolver** class calls the LAPACK (Anderson et al., 1995a) library, as shown in figure 3.28.

Analysts can of course still provide their own solvers, as in the case of the element-by-element conjugate gradient solver, whose interface and solve() method are as shown in figure 3.29.

The separation of the system of equation and solver into two separate classes allows the analyst to change between different solvers as the solution progresses. For example, the analyst may wish to use a direct solver initially and then go to an iterative solver later.

While **Matrix** and **Vector** objects could be used instead of the **SystemOfEqn** and **Solver** objects, as is done in Archer (1996), combining the matrix and vector

```
DirectBandSPDLinSOESolver::solve() {
      // solve AX = B
      if (factored == false)
            dpbsv_("U",&n,&kd,&nrhs,Aptr,&ldA,Bptr,&ldB,&info);
      else
            dpbtrs_("U",&n,&kd,&nrhs,Aptr,&ldA,Bptr,&ldB,&info);
      factored = true;
}
```

Figure 3.28: Pseudo-Code for the **DirectBandSPDLinSOESolver** classes solve Method

data into one object results in more efficient code, as the operations do not have to go through the **Matrix** and **Vector** interfaces to obtain components. Also, for common storage schemes, i.e. profile and sparse, the writer of a **LinearSOE** class will not have to implement methods for adding, subtracting, multiplying by other **LinearSOE** classes, methods which will probably never be used by the analyst.

## 3.5 Example Programs

Using the classes presented in sections 2.2 and 3.4, pseudo C++ code is now presented to demonstrate the flexibility and extensibility of the new approach.

### 3.5.1 Flexibility

To demonstrate flexibility, consider the changes that an analyst would have to make to modify the analysis performed on a structure. The main routine for a program to perform a static nonlinear analysis using the transformation method to enforce constraints, a reverse Cuthill-McKee DOF numbering scheme, the Newton-Raphson solution algorithm, and a banded symmetric positive definite system of equations, which is solved by a direct solver, is as follows:

```
003      Domain theDomain();
004      DeltaWing theModelBuilder(theDomain)
005      theModelBuilder.buildModel();
006
007      /* create the analysis */
```

```
class EleByEleCGStaticSolver: public EleByEleSolver
      public:
            EleByEleCGSolver(double tol = 1.0e-6);
            int solve(void);
      private:
            double TOL;
            EleByEleLinearSOE *theLinearSOE
}

EleByEleCGSolver::solve(void) {
      Vector &x = theLinearSOE->getX()
      Vector &r = theLinearSOE->getB()
      Vector x.Zero()
      Vector p(r)
      Vector Ap(p.Size())
      double rdotr = r * r
      while (sqrt(rdotr) > TOL) {
            Ap.zero()
            FE_Iter &theFes = theLinearSOE->getFEs()
            while ((FE_Element *theEle = theFEs()) ≠ 0) {
                  Ap.Assemble(theEle->getTangForce(p), theEle->getID())
            double γ = rdotr/(p*Ap)
            x += p*γ
            r -= Ap * γ
            double oldrdotr = rdotr
            rdotr = r * r
            double β = rdotr/oldrdotr
            p = r + (p * β)
      }
}
```

Figure 3.29: An Element By Element Solvers Interface and solve Method

.

```
008        Transformation theConstraintHandler;
009        RCM theGraphNumberer;
010        DOF_Numberer theDOFNumberer(theGraphNumberer);
011        AnalysisModel theModel;
012        DirectBandSPDSOE theSolver;
013        BandSPDSOE theLinearSOE(theSolver);
014        StaticIntegrator theIntegrator;
015        NewtonRaphson theSolnAlgo;
016        StaticAnalysis theAnalysis(theDomain, theConstraintHandler,
017            theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinearSOE);
018
019        /* perform the analysis */
020        theDomain.setLoadCase(1);
021        theAnalysis.analyze;
022
```

To change from a non-linear problem to a linear problem, the analyst replaces line 015 in the original with the following:

```
015        Linear theSolnAlgo;
```

The analyst now wishes instead to use the minimum-degree algorithm and a sparse matrix storage scheme and a conjugate gradient solver. This requires replacing lines 009, 012 and 013 in the original with the following:

```
009        MinDegree theGraphNumberer;
012        ConjugateGradientSparseSPDSOE theSolver;
013        SparseSPDSOE theLinearSOE(theSolver);
```

To perform a non-linear step-by-step dynamic analysis using the Newmark integration strategy, with constants $\beta = 1/4$ and $\gamma = 1/2$, on the domain, lines 014, 016 and 017 in the original are replaced with the following:

```
014        Newmark theIntegrator(1/4, 1/2);
016        DirectIntegrationAnalysis theAnalysis(theDomain,theConstraintHandler,
017            theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinearSOE);
```

### 3.5.2 Extensibility

To demonstrate the extensibility of the new framework, sample code to implement the BFGS solution algorithm, a quasi-Newton algorithm, is presented:

```
100  class BFGS: public EquiSolnAlgo
101       public:
102            BFGS(int maxSteps, int numIncr, double tol, double Gtol, double maxSearch);
103            int solveCurrentStep(void);
104            void computeDirection(int numStep);
105            void lineSearch();
106            void computeTrialUnbalance(Vector &ΔUtrial, Vector &trialUnbalance);
107       private:
108            double TOL;
109            int numLoadIncr, maxSteps;
110            Vector unbalance,oldUnbalance,d,a,b,δI,γI,ΔU
111  }

125  BFGS::solveCurrentStep{
126       theIntegrator->formUnbalance()
127       theIntegrator->formTangent()
128       unbalance = theLinearSOE->getRHS()
129       d.zero()
130       numStep = 0
131       while (unbalance.norm() > TOL && numSteps < maxSteps) {
132            this->computeDirection()
133            this->lineSearch()
134            theIntegrator->update(d)
135            theIntegrator->formUnbalance()
136            oldUnbalance = unbalance
137            unbalance = theLinearSOE->getRHS()
138            numStep++
139       }
140  }

150  BFGS::computeDirection(int numStep) {
151       if (numStep == 0) {
152            theLinearSOE->solve()
153            d = theLinearSOE->getX()
154       } else {
155            γI = unbalance - oldUnbalance
156            w[numStep] = (1/(δI*γI)) * δI
157            v[numStep] = (1-s*sqrt((d*γI)/(δI*oldUnbalance)))*oldUnbalance-unbalance
158            double γ = w[numStep]*unbalance
159            y = unbalance + γ*v[numStep]
160            for (int j=numStep-1; j>0; j−−) {
161                 γ = w[j] * y
162                 x = y + (v[j] * γ)
162                 y = x
163            }
164            theLinearSOE->solve(y)
165            x = theLinearSOE->getX()
166            γ = v[1]*x
167            y = x + γ * w[1]
168            for (int j=2; j <= numStep; j++)
169                 γ = v[j] * b
```

```
170                 x = y + (w[j] * γ)
171                 y = x
172             }
173             d = y
174         }
175 }
176
180 BFGS::lineSearch() {
181         // find an interval [sa,sb] containing a root
182         sa = 0.0
183         sb = 1.0
184         Ga = d*unbalance
185         ΔUtrial = ΔU + d
186         this->computeUnbalance(ΔUtrial,trialUnbalance)
187         Gb = d*trialUnbalance
188         while (Ga*Gb > 0 && sb < maxSearch) {
189             sa = sb
190             sb = 2.0*sa
191             Ga = Gb
192             ΔUtrial = ΔU + (d * sb)
193             this->computeUnbalance(ΔUtrial,trialUnbalance)
194             Gb = d*trialUnbalance
195         }
196
197         if (Ga * Gb > 0) {
198         // if no interval found use a regular Newton step
199             ΔU += d
200             δI = d
201         } else {
202             use regula-falsi to determine the root
203             numIter = 0
204             while (abs(Ga-Gb) < Gtol && (numIter < maxIter)) {
205                 step = sa - Ga*(sa-sb)/(Ga-Gb)
206                 ΔUtrial = ΔU + d*step
207                 this->computeUnbalance(ΔUtrial,trialUnbalance)
208                 Gstep = d*trialUnbalance
209                 if (Gs*Ga > 0) {
210                     sa = step
211                     Ga = Gstep
212                 } else {
213                     sb = step
214                     Gb = Gstep
215                 }
216             }
217             ΔU += d*sb
218         }
219 }
220
```

To change the original program to perform an analysis using this solution algorithm with a maximum of 100 iterations, a maximum of 10 Regula-Falsi steps, a convergence tolerance of 1e-3, and a maximum search length of 2.0, the analyst simply replaces line 015 in the original main program with:

```
015        BFGS theSolnAlgo(100,10,1e-3,2.0);
```

This example demonstrates that, with less than 100 lines of code, the analyst can introduce a new class into the system. The **BFGS** class provides a new **EquiSolnAlgo** subclass, which can be used in any analysis which requires a **EquiSolnAlgo** class. This greatly increases the number of possible analysis options that are available to the analyst.

## 3.6 Extension of Framework to Other Types of Analysis Procedures

The framework can be extended to include other types of analysis procedures, such as eigenvalue analysis, modal transient analysis, and static pushover. For each new analysis procedure, subclasses of the basic classes in the aggregation must be defined. This section describes what changes to the framework are made to include eigenvalue analysis and modal transient analysis.

### 3.6.1 Extensions for Eigenvalue Analysis

In solid mechanics and structural engineering, the determination of the buckling load and the determination of the natural frequencies and mode shapes of structures is a common operation. An eigenvalue analysis is concerned with finding these quantities. They are obtained from solving either of the following:

1. Standard eigenvalue problem $\mathbf{K\Phi} = \mathbf{\Phi\Lambda}$ for case of buckling problem. There are many ways to solve the problem, for example Singular Value Decomposition, Jacobi, Householder Tridiagonalization, Vector Iteration and Rayleigh Quotient Iteration.

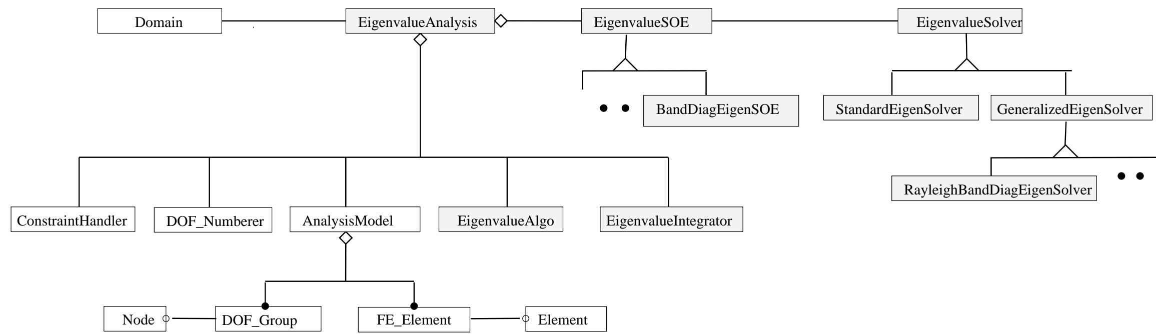



Figure 3.30: Class Diagram for Eigenvalue Analysis

2. Generalized eigenvalue problem $\mathbf{K\Phi} = \mathbf{M\Phi\Lambda}$ for case of natural modes and frequencies problem. There are again many ways to solve the problem: Vector iteration methods, Rayleigh quotient iteration, and Lanczos iteration methods. If $\mathbf{M}$ is symmetric positive definite, the problem can be transformed to the standard eigenvalue problem,

To extend the analysis framework to include eigenvalue analysis, the framework shown in figure 3.30 is used. The new classes introduced for this framework, which are shaded in figure 3.30, are:

1. **EigenvalueAnalysis**: The **EigenvalueAnalysis** object is created by the analyst to perform an eigenvalue analysis on the **Domain**. The object is, as shown in figure 3.30, an aggregation of objects of the following types: **ConstraintHandler**, **DOF_Numberer**, **EigenvalueAlgo**, **EigenvalueIntegrator**, **AnalysisModel**, and **EigenvalueSOE**. The constructor to the **EigenvalueAnalysis** class, whose interface is shown in figure 3.31, verifies the objects passed as arguments are of the correct type. The class provides three methods:

   (a) domainChanged(): This method sets the links needed by the objects in the aggregation, invokes handle() on the **ConstraintHandler**, invokes number() on the **DOF_Numberer**, setSize() on the **EigenSOE**, and analysisModelChanged() on the **EigenvalueAlgo** and **EigenvalueIntegrator** objects.

(b) analyze(): This is the method which is invoked to perform the eigenvalue analysis. To do this it invokes domainChanged() on itself, and then solveCurrentStep() on the **EigenvalueAlgo** object, as shown in figure 3.32.

(c) updateMode(): This is a method to update the nodal displacements with the values in the eigenvector for a specific mode. This is done by invoking update() on the **EigenvalueIntegrator** object with the correct eigenvector. The method returns the value of the eigenvalue for the specified mode. The method is as shown in figure 3.32.

```
class EigenvalueAnalysis: public Analysis {
    public:
        EigenvalueAnalysis(Domain &theDomain,
                           ConstraintHandler &theHandler,
                           DOF_Numberer &theNumberer,
                           AnalysisModel &theModel,
                           EigenvalueAlgo &theSolnAlgo,
                           EigenvalueSOE &theSOE,
                           EigenvalueSOESolver &theSolver,
                           EigenvalueIntegrator &theEigenIntegrator);
        virtual ~EigenvalueAnalysis();

        virtual int domainChanged(void);
        virtual int analyze(void);
        virtual double updateMode(int mode);
};
```

Figure 3.31: Interface for the **EigenvalueAnalysis** Class

2. **EigenvalueAlgo**: The **EigenvalueAlgo** object in the analysis aggregation orchestrates the steps in the analysis. Like the **EquiSolnAlgo** class, it is an abstract class. The class provides the method setLinks(), which sets the links needed by the object, and defines the method solveCurrentStep() to be pure virtual. solveCurrentStep() is invoked to perform the eigenvalue analysis. The steps taken in the analysis depend on the type of subclass of **EigenvalueAlgo** used by the analyst. Examples of subclasses are **Frequency** and **Buckling**. The solveCurrentStep() of the **Buckling** class, which is shown in figure 3.33, will determine the buckling mode shapes of a **Domain**. The solveCurrentStep() of

```
EigenvalueAnalysis::analyze(void){
    if (theDomain->hasDomainChanged() == true)
        this->domainChanged();
    theAlgorithm->solveCurrentStep();
    analysisDone = true;
}

EigenvalueAnalysis::updateMode(int mode){
    if (analysisDone == flase)
        this->analyze();
    const Vector &φ = theSOE->getφ(mode);
    theIntegrator->update(φ);
    return theSOE->getλ(mode);
}
```

Figure 3.32: Pseudo-Code for Selected Methods for the **EigenvalueAnalysis** Class

the **Frequency** class, which is shown in figure 3.33, will determine the natural modes and frequencies of a **Domain**.

```
Buckling::solveCurrentStep{
    theIntegrator->formK();
    theEigenvalueSOE->solve();
}


Frequency::solveCurrentStep{
    theIntegrator->formK();
    theIntegrator->formM();
    theEigenvalueSOE->solve();
}
```

Figure 3.33: Pseudo-Code for the **Buckling** and **Frequency** Classes solveCurrentStep Method

3. **EigenvalueIntegrator**: The **EigenvalueIntegrator** object is responsible providing methods that the **EigenvalueAlgo** object can use. The **EigenvalueIntegrator** class, whose interface is as shown in figure 3.34, provides methods to form the $K$ and $M$ matrices, and methods to instruct the **FE_Element** and **DOF_Group** objects how to determine their contribution to these matrices.

The class also provides a method **update()**, which will set the trial displacement at the **Node** objects equal to those values specified in the eigenvector passed as the argument.

```
class EigenvalueIntegrator : public Integrator {
    public:
        EigenvalueIntegrator();
        virtual ~EigenvalueIntegrator();

        virtual void setLinks(AnalysisModel &, EigenvalueSOE &);
        virtual void analysisModelChanged(void);

        // methods to from the M and K matrices
        virtual int formK();
        virtual int formM();

        // methods to instruct the FE_Elements and DOF_Group
        // objects how to determine their constributions to M and K
        virtual int formEleTangK(FE_Element *theFeEle);
        virtual int formEleTangM(FE_Element *theFeEle);
        virtual int formNodTangM(DOF_Group *theDOFgrp);
        virtual int update(const Vector &φ);
};
```

Figure 3.34: Interface for the **EigenvalueIntegrator** Class

4. **EigenvalueSOE**: The **EigenvalueSOE** object is responsible for storing the **K**, **M**, and $\Phi$ matrices, for storing the $\Lambda$ vector, and for providing access to $\Phi$ and $\Lambda$. The **EigenvalueSOE** class, whose interface is shown in figure 3.36, is an abstract class which defines the methods that all subclasses must provide. These include methods to add to the $K$ and $M$ matrices. In addition they include the methods get$\Phi$(), which will return all the eigenvectors, get$\phi$(), which will return a single eigenvector, get$\Lambda$(), which will return all the eigenvalues, get$\lambda$(), which will return a single eigenvalue, and solve(), which will invoke solve() on the associated **EigenvalueSolver** object.

   Subclasses of **EigenvalueSOE** are written for the different matrix storage schemes, as were provided for the **LinearSOE** class in section 3.4.9.

5. **EigenvalueSolver**: The **EigenvalueSolver** object is responsible for comput-

```
EigenvalueIntegrator::formEleTangK(FE_Element *theEle) {
    theEle->zeroTang();
    theEle->addKtoTang();
}

EigenvalueIntegrator::update(Vector & φ) {
    DOF_Iter theDofs = theAnalysisModel->getDOFs();
    while ((dofPtr = theDOFs()) ≠ 0)
        dofPtr->setDisp(φ);
}
```

Figure 3.35: Pseudo-Code for Selected Methods of the **EigenvalueIntegrator** Class

```
class EigenvalueSOE : public SystemOfEqn {
    public:
        EigenvalueSOE(EigenSolver &theSolver);
        virtual ~EigenvalueSOE();

        virtual int setSize(Graph &theDOFGraph) =0;
        virtual int addK(const Matrix &, const ID &, double fact = 1.0) =0;
        virtual int addM(const Vector &, const ID &, double fact = 1.0) =0;

        virtual void zeroK(void) =0;
        virtual void zeroM(void) =0;

        virtual int solve(void) = 0;
        virtual const Matrix &getΦ() = 0;
        virtual const Vector &getφ(int mode) = 0;
        virtual const Vector &getΛ() = 0;
        virtual const double &getλ(int mode) = 0;
};
```

Figure 3.36: Interface for the **EigenvalueSOE** Class

ing $\Phi$ and $\Lambda$, given the **K** and **M** matrices stored in the **EigenvalueSOE** object. The **EigenvalueSolver** class, which is an abstract class, has two subclasses **StandardEigenSolver** and **GeneralizedEigenSolver**, which also are abstract classes. Subclasses of these two classes are provided for each **EigenvalueSOE** subclass. Each subclass provides its own implementation of the solve() method. For **EigenvalueSOE** classes, which use standard storage schemes, calls can be made to existing libraries, e.g. LAPACK (Anderson et al., 1995a), to determine the eigenvalues and eigenvectors. The design for this class closely resembles the design for the **LinearSolver** class, which was presented in section 3.4.9.

### 3.6.2 Extensions for Modal Transient Analysis

A modal transient analysis performs a transient analysis, as outlined in section 3.2, but uses a transformed system of equations. The system of equations are transformed so that: (1) the system is less computationally expensive to solve, for example $K^*$ of equation 3.18 will be diagonal in the transformed system and/or (2) the transformed system of equations has significantly fewer equations.

In a modal transient analysis we replace the original system with a transformed system. In the case of a linear transient problem, the original system of equations can be expressed in matrix form as:

$$\mathbf{M}\ddot{\mathbf{U}}(t) + \mathbf{C}\dot{\mathbf{U}}(t) + \mathbf{K}\mathbf{U}(t) = \mathbf{P}(t) \tag{3.26}$$

This system is replaced by the transformed system:

$$\left(\mathbf{T}^T\mathbf{M}\mathbf{T}\right)\ddot{\mathbf{X}}(t) + \left(\mathbf{T}^T\mathbf{C}\mathbf{T}\right)\dot{\mathbf{X}}(t) + \left(\mathbf{T}^T\mathbf{K}\mathbf{T}\right)\mathbf{X}(t) = \mathbf{T}^T\mathbf{P}(t) \tag{3.27}$$

where the two systems are related through the expression $\mathbf{U}(t) = \mathbf{T}\mathbf{X}(t)$. Once the system has been transformed, a transient analysis, similar to that outlined in section 3.2, can be performed. There are many types of transformation matrices **T** which can be used. In practice **T** is either the eigenvectors for $\mathbf{K}\Phi = \mathbf{M}\Phi\Lambda$, or derived Ritz or Lanczos vectors (Chopra, 1995).

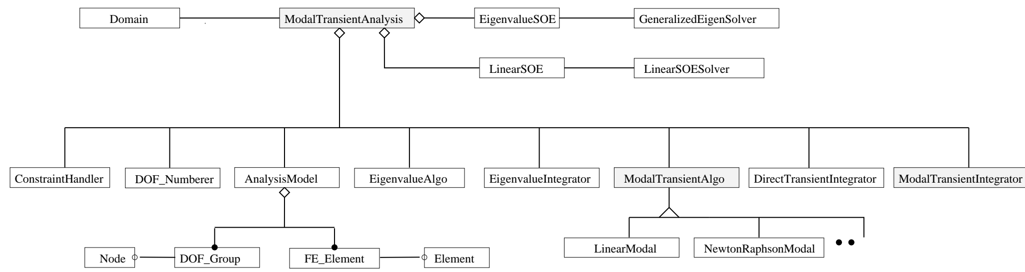



Figure 3.37: Class Diagram for Modal Transient Analysis

To extend the framework to include modal transient analysis, the framework shown in figure 3.37 is used. The new classes introduced for this framework, which are shaded in figure 3.37, are:

1. **ModalTransientAnalysis**: A **ModalTransientAnalysis** object is created by the analyst to perform a modal transient analysis on the **Domain**. The object, as shown in figure 3.37, is an aggregation consisting of objects of the following type: **ConstraintHandler**, **DOF_Numberer**, **AnalysisModel**, **ModalTransientAlgo**, **LinearSOE**, **DirectTransientIntegrator**, **EigenvalueSOE**, **EigenvalueIntegrator** and **ModalTransientIntegrator**. The **ModalTransientAnalysis** class, whose interface is shown in figure 3.38, provides a constructor which will verify the correct types of objects are passed as arguments. In addition the methods domainChange(), which sets up the links needed and invokes start up methods in the aggregate objects, and analyze(), which is invoked by the analyst to perform the analysis, are provided. The analyze method is shown in figure 3.39.

2. **ModalTransientIntegrator**: The **ModalTransientIntegrator** class has two methods: setLinks() which sets up a link to the **AnalysisModel** object, and update(). The update() method, as shown in figure 3.40, is responsible for setting the transformation matrix **T** at each **FE_Element** and **DOF_Group** object in the **AnalysisModel**.

3. **ModalTransientAlgo**: The **ModalTransientAlgo** object specifies the steps to be performed in the modal transient analysis. The **ModalTransientAlgo**

```
class ModalTransientAnalysis: public TransientAnalysis {
  public:
      ModalTransientAnalysis(Domain &theDomain,
                             ConstraintHandler &theHandler,
                             DOF_Numberer &theNumberer,
                             AnalysisModel &theModel,
                             ModalTransientAlgo &theSolnAlgo,
                             LinearSOE &theSOE,
                             DirectTransientIntegrator &theDirectTransientIntegrator);
                             EigenvalueSOE &theSOE,
                             EigenvalueIntegrator &theEigenIntegrator);
      virtual ~EigenvalueAnalysis();

      virtual int domainChanged(void);
      virtual int analyze(void);
};
```

Figure 3.38: Interface for the **ModalTransientAnalysis** Class

```
ModalTransientAnalysis::analyze{
    double time = tStart;
    while (time < tFinal) {
        theIntegrator->newStep(Δt);
        theAnalysisModel->applyLoadDomain(time);
        theAlgorithm->solveCurrentStep();
        theAnalysisModel->commitDomain();
        time += Δt;
    }
}
```

Figure 3.39: Pseudo-Code for the **ModalTransientAnalysis** Classes analyze Method

```
ModalTransientIntegrator::update(const Matrix &Φ) {
    FE_EleIter theEles = theAnalysisModel->getFEs();
    while ((feElePtr = theEles()) ≠ 0)
        feElePtr->setTransformation(Φ);
    DOF_Iter theDofs = theAnalysisModel->getDOFs();
    while ((dofPtr = theDOFs()) ≠ 0)
        dofPtr->setTransformation(Φ);
}
```

Figure 3.40: Pseudo-Code for the **ModalTransientIntegrator** Classes update Method

class is an abstract class with two methods: setLinks(), which sets up links to the objects in the aggregation; and solveCurrentStep, which is declared as pure virtual. Examples of subclasses of **ModalTransientAlgo** are, as shown in figure 3.37, **LinearModal** and **NewtonRaphsonModal**. For the subclass **LinearModal**, the method, which is shown in figure 3.41, will form the transformation matrix on the first invocation of the method, and will thereafter simply perform the linear iteration scheme defined for the **Linear** class.

```
LinearModal::solveCurrentStep{
    // transform the system if not already done so
    if (transformedSystem == false) {
        theEigenIntegrator->formK(); // we use eigen integrtor to
        theEigenIntegrator->formM(); // set up the eigen equations
        theEigenSOE->solve();
        theModalTransientIntegrator->update(theEigenSOE->getΦ());
        transformedSystem = true;
    }
    theDirectTransientIntegrator->formNodalUnbalance();
    theDirectTransientIntegrator->formElementResidual();
    theDirectTransientIntegrator->formTangent();
    theLinearSOE->solveX();
    Vector &ΔU = theLinearSOE->getX();
    theDirectTransientIntegrator->updateIncr(ΔU);
}
```

Figure 3.41: Pseudo-Code for the **LinearModal** Classes solveCurrentStep Method

For the subclass **NewtonRaphsonModal**, wherein the transformation matrix is computed at the beginning of each step, the method is as shown in figure 3.42. The method is similar to the **NewtonRaphson** classes solveCurrentStep() method, the difference being that a transformationmatrix is calculated at the start of each iteration.

```
NewtonRaphsonModal::solveCurrentStep{
      // update the mode shapes for the current step if convergence slow
      if (numIterationsLast > numIterationsRecalcEigen) {
            theEigenIntegrator->formK();
            theEigenIntegrator->formM();
            theEigenSOE->solve();
            theModalTransientIntegrator->update(theEigenSOE->getΦ());
      }
      // now perform NR on the reduced system
      theDirectTransientIntegrator->formUnbalance();
      int numIterationsLast = 0;
      while (theLinearSOE->normRHS() > TOL) {
            theDirectTransientIntegrator->formTangent();
            theLinearSOE->solveX();
            Vector &ΔU = theLinearSOE->getX();
            theDirectTransientIntegrator->updateIncr(ΔU);
            theDirectTransientIntegrator->formUnbalance();
            numIterationsLast += 1;
      }
}
```

Figure 3.42: Pseudo-Code for the **NewtonRaphsonModal** Classes solveCurrentStep Method

The interface for the **FE_Element** and **DOF_Group** classes must also be modified. This is so the transformed system, equation 3.27, is solved and not the original system, equation 3.26. In a modal transient analysis, the **FE_Element**s are required to return $\mathbf{T}^T\mathbf{K}_e^*\mathbf{T}$ and $\mathbf{T}^T\mathbf{P}_e^*$ on invocation of getTang() and getResidual(). To allow this, an additional method setTransformation() is added to the **FE_Element** interface. The **DOF_Group**s return $\mathbf{T}^T\mathbf{K}_n^*\mathbf{T}$ and $\mathbf{T}^T\mathbf{P}_n^*$ on invocation of getTang() and getUnbalance(). They also update the nodal response quantities with $\mathbf{TX}$, $\mathbf{T\dot{X}}$ and $\mathbf{T\ddot{X}}$, when setting the trial response quantities at the nodes. To allow this, the method setTransformation() is added to the **DOF_Group** interface.

# Chapter 4

# Object-Oriented Domain Decomposition

In this chapter an object-oriented design for domain decomposition is presented. The design provides for the implementation of a number of domain decomposition methods used in finite element analysis, using a hierarchy similar to that presented in chapter 3. Many of the classes defined in chapter 3 are re-used for domain decomposition, further demonstrating the extensibility of the object-oriented design presented for the analysis algorithms.

## 4.1 Introduction

Divide and conquer is an age old technique used to solve large problems. In structural analysis, the divide and conquer approach is typically associated with the substructuring method, a domain decomposition method for solving the the large systems of equations, given by equation 3.18, arising from refined finite element discretizations of a problem. Domain decomposition techniques are commonly employed because: (1) for linear problems, in which the geometry repeats itself, the subdomain information required in the analysis algorithm need only be formulated once for identical subdomains; (2) if memory is limited there is the potential to solve much larger problems using secondary storage; (3) superior convergence rate of domain decomposition methods over other iterative methods; and (4) potential for efficient parallelization because of data locality.

This chapter presents a brief review of the domain decomposition methods commonly employed for finite element analysis. A review is made of existing object-oriented approaches to domain decomposition. Finally a new object-oriented approach for domain decomposition is presented.

## 4.2 Domain Decomposition Methods for Finite Element Analysis

Domain decomposition methods are methods used for solving linear or non-linear problems. What distinguishes domain decomposition methods from other methods, such as the multifront method, is that information about the discretization of the domain is used, i.e. the element connectivity is used explicitly to determine the subdomains. Domain decomposition methods are classified into two groups:

1. *Non-overlapping methods*: The domain $\Omega$ is decomposed into several disjoint subdomains $\Omega_i$ such that $\Omega = \bigcup_i \Omega_i$. Examples of non-overlapping methods are substructuring, iterative substructuring, and finite element tearing and interconnecting (FETI) (Farhat and Roux, 1991; Farhat and Crivelli, 1994).

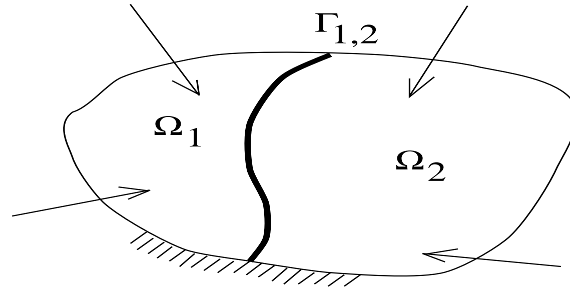



Figure 4.1: Domain split into Two Subdomains

2. *Overlapping methods*: The domain is decomposed into several slightly overlapping subdomains. Examples of overlapping methods are the Schwartz methods.

Of the two groups of methods, the non-overlapping methods are typically the ones used in finite element analysis, and are the ones that will be examined in this chapter.

### 4.2.1 Non-overlapping Domain Decomposition Methods

Consider a domain $\Omega$ split into two disjoint subdomains $\Omega_1$ and $\Omega_2$, with a boundary surface between the domains $\Gamma_{1,2}$, as shown in the figure 4.1. The unknowns in the matrix equation 3.18 are partitioned into three sets: those corresponding to the unknowns in $\Omega_1$ denoted $\mathbf{U}_1$, those corresponding to the unknowns in $\Omega_2$ denoted $\mathbf{U}_2$, and those corresponding to the unknowns on $\Gamma_{1,2}$ denoted $\mathbf{U}_3$. The matrix equation can be rewritten in block partitioned form as

$$\begin{bmatrix} \mathbf{K}_{11} & \mathbf{0} & \mathbf{K}_{13} \\ \mathbf{0} & \mathbf{K}_{22} & \mathbf{K}_{23} \\ \mathbf{K}_{31} & \mathbf{K}_{31} & \mathbf{K}_{33} \end{bmatrix} \left\{ \begin{array}{c} \mathbf{U}_1 \\ \mathbf{U}_2 \\ \mathbf{U}_3 \end{array} \right\} = \left\{ \begin{array}{c} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{array} \right\} \tag{4.1}$$

where $\mathbf{K}_{33} = \mathbf{K}_{33}^1 + \mathbf{K}_{33}^2$, i.e. $\mathbf{K}_{33}$ contains contributions from elements in both $\Omega_1$ and $\Omega_2$. The unknowns $\mathbf{U}_1$ and $\mathbf{U}_2$ can be eliminated from the system, using the Schur complement of $\mathbf{K}_{33}$ in $\mathbf{K}$. The resulting Schur complement system is

$$\mathbf{K}_{33}^* \mathbf{U}_3 = \mathbf{P}_3^* \tag{4.2}$$

where

$$\mathbf{K}_{33}^{*} = \mathbf{K}_{33}^{1} - \mathbf{K}_{31}\mathbf{K}_{11}{}^{-1}\mathbf{K}_{13} + \mathbf{K}_{33}^{2} - \mathbf{K}_{32}\mathbf{K}_{22}{}^{-1}\mathbf{K}_{23}$$
$$= \mathbf{K}_{33}^{1*} + \mathbf{K}_{33}^{2*} \tag{4.3}$$

and

$$\mathbf{P}_{3}^{*} = \mathbf{P}_{3}^{1} - \mathbf{K}_{31}\mathbf{K}_{11}^{-1}\mathbf{P}_{1} + \mathbf{P}_{3}^{2} - \mathbf{K}_{32}\mathbf{K}_{22}^{-1}\mathbf{P}_{2}$$
$$= \mathbf{P}_{3}^{1*} + \mathbf{P}_{3}^{2*} \tag{4.4}$$

This concept can be extended to domains wherein the domain is split into an arbitrary number of subdomains, $ns$. For each subdomain, the unknowns are partitioned into two sets: those corresponding to the internal unknowns, $\mathbf{U}_i$, and those corresponding to the external unknowns, $\mathbf{U}_e$. For each subdomain, $s = 1, 2, ..ns$, the subdomain equation $\mathbf{K}_s\mathbf{U}_s = \mathbf{P}_s$ can be expressed in a partitioned form:

$$\begin{bmatrix} \mathbf{K}_{ii} & \mathbf{K}_{ie} \\ \mathbf{K}_{ei} & \mathbf{K}_{ee} \end{bmatrix}_s \begin{Bmatrix} \mathbf{U}_i \\ \mathbf{U}_e \end{Bmatrix}_s = \begin{Bmatrix} \mathbf{P}_i \\ \mathbf{P}_e \end{Bmatrix}_s \tag{4.5}$$

The resulting Schur complement system, or the interface problem as it is sometimes called, as given by equation 4.2 for the two subdomain problem, can be expressed as:

$$\left( \mathop{\mathcal{A}}_{s=1}^{ns} \mathbf{K}_{ee}^{*} \right) \mathbf{U} = \mathop{\mathcal{A}}_{s} (\mathbf{P}_{e}^{*}) \tag{4.6}$$

where for each subdomain $s$:

$$\mathbf{K}_{ee}^{*} = \mathbf{K}_{ee} - \mathbf{K}_{ei}\mathbf{K}_{ii}^{-1}\mathbf{K}_{ie} \tag{4.7}$$

$$\mathbf{P}_{e}^{*} = \mathbf{P}_{e} - \mathbf{K}_{ei}\mathbf{K}_{ii}^{-1}\mathbf{P}_{i} \tag{4.8}$$

The internal unknowns for each subdomain $s$, $\mathbf{U}_i$, can be determined once $\mathbf{U}_e$ are known using

$$\mathbf{U}_i = \mathbf{K}_{ii}^{-1}(\mathbf{P}_i - \mathbf{K}_{ie}\mathbf{U}_e) \tag{4.9}$$

The reason that the domain decomposition method is useful for larger problems is that the Schur complement system, given in equation 4.6, is much smaller than the

original system, given in equation 3.18, and, with the good choice of subdomains, is better conditioned. There are a number of non-overlapping domain decomposition methods uses in finite element analysis. The most popular of these are substructuring, iterative substructuring, and FETI.

### Substructuring

The substructuring method is a direct approach to solving the equations. In this method the solution is obtained in three steps:

1. The unknowns not on the interfaces are eliminated in a process known as static condensation, to form for each subdomain $\mathbf{K}^*_{ee}$, given in equation 4.7, and $\mathbf{P}^*_e$, given in equation 4.8. In parallel processing this step can be performed concurrently among subdomains located on different processors.

2. The Schur complement system, equation 4.6, is formed by assembling the contributions from all subdomains. This system is then solved for the interface unknowns.

3. The internal unknowns in each subdomain are then determined using equation 4.9. In parallel processing this step can again be performed concurrently.

### Iterative Substructuring

The substructuring method can be a computationally expensive method, due to the need to form the Schur complement $\mathbf{K}^*_{ee}$, given by equation 4.7, for each substructure. Also, as the number of substructures grows so does the size of the Schur complement system, equation 4.6. The iterative substructuring method assumes an element-by-element Krylov subspace iteration method, e.g. the conjugate gradient method, is being used to solve the interface problem. The reduced system is solved without explicitly forming the Schur complement. Using an element-by-element solver, each subdomain is responsible for providing the matrix-vector product $\mathbf{K}^*_{ee}\mathbf{X}_e$, as seen in the example code in figure 3.29. This matrix-vector product can be obtained without explicitly forming $\mathbf{K}^*_{ee}$, using the following matrix-vector operations:

$$\mathbf{K}_{ee}^{*}\mathbf{X}_e = \mathbf{K}_{ee}\mathbf{X}_e - \mathbf{K}_{ei}\left(\mathbf{K}_{ii}^{-1}\left(\mathbf{K}_{ie}\mathbf{X}_e\right)\right) \tag{4.10}$$

Once convergence has been achieved for the external unknowns, the internal unknowns can be determined using equation 4.9.

### FETI - Finite element tearing and interconnecting

This is a domain decomposition approach which has been shown to be quite useful. In this method the subdomains are considered to be independent of each other and the Lagrange multiplier method is used to enforce the displacement continuity on the subdomains interfaces, i.e. to ensure $\mathbf{U}_3^1 = \mathbf{U}_3^2$ for the two subdomain problem.

The system of equations for the two subdomain example can be expressed as:

$$\begin{bmatrix} \mathbf{K}_1 & \mathbf{0} & \mathbf{B}_1 \\ \mathbf{0} & \mathbf{K}_2 & \mathbf{B}_2 \\ \mathbf{B}_1^T & \mathbf{B}_2^T & \mathbf{0} \end{bmatrix} \begin{Bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \lambda_{12} \end{Bmatrix} = \begin{Bmatrix} \mathbf{Q}_1 \\ \mathbf{Q}_2 \\ \mathbf{0} \end{Bmatrix} \tag{4.11}$$

where $\mathbf{B} = [\mathbf{B}_1\mathbf{B}_2]^T$ is a constraint matrix, $\mathbf{V}_1 = [\mathbf{U}_1\mathbf{U}_3^1]^T$, $\mathbf{Q}_1 = [\mathbf{P}_1\mathbf{P}_1^3]^T$ and $\mathbf{K}_\delta$ is the uncondensed tangent matrix, represented by equation 3.22, for subdomain $\Omega_\delta$. Looking at the individual equations expressed in equation 4.11 and rearranging:

$$\mathbf{V}_1 = \mathbf{K}_1^{-1}\left(\mathbf{Q}_1 - \mathbf{B}_1\lambda_{12}\right) \tag{4.12}$$

$$\mathbf{V}_2 = \mathbf{K}_2^{-1}\left(\mathbf{Q}_2 - \mathbf{B}_2\lambda_{12}\right) \tag{4.13}$$

$$\mathbf{B}_1^T\mathbf{V}_1 + \mathbf{B}_2^T\mathbf{V}_2 = \mathbf{0} \tag{4.14}$$

multiplying equation 4.12 by $\mathbf{B}_1^T$ and equation 4.13 by $\mathbf{B}_2^T$ and rearranging:

$$\mathbf{B}_1^T\mathbf{K}_1^{-1}\mathbf{B}_1\lambda_{12} = \mathbf{B}_1^T\mathbf{K}_1^{-1}\mathbf{Q}_1 - \mathbf{B}_1^T\mathbf{V}_1 \tag{4.15}$$

$$\mathbf{B}_2^T\mathbf{K}_2^{-1}\mathbf{B}_2\lambda_{12} = \mathbf{B}_2^T\mathbf{K}_2^{-1}\mathbf{Q}_2 - \mathbf{B}_2^T\mathbf{V}_2 \tag{4.16}$$

adding equations 4.15 and 4.16 and using equation 4.14, an expression is obtained for the Lagrange multiplier $\lambda_{12}$:

$$\left(\mathbf{B}_1^T\mathbf{K}_1^{-1}\mathbf{B}_1 + \mathbf{B}_2^T\mathbf{K}_2^{-1}\mathbf{B}_2\right)\lambda_{12} = \mathbf{B}_1^T\mathbf{K}_1^{-1}\mathbf{Q}_1 + \mathbf{B}_2^T\mathbf{K}_2^{-1}\mathbf{Q}_2 \tag{4.17}$$

This concept can be extended to include multiple subdomains. The resulting system of equations are of the form:

$$\left(\sum_{s=1}^{ns} \mathbf{B}_s^T \mathbf{K}_s^{-1} \mathbf{B}_s\right) \lambda = \sum \mathbf{B}_s^T \mathbf{K}_s^{-1} \mathbf{Q}_s \tag{4.18}$$

Once the Lagrange multipliers have been determined, the solution for the unknowns $\mathbf{V}_s$ for each subdomain can be determined using:

$$\mathbf{V}_s = \mathbf{K}_s^{-1} \left(\mathbf{Q}_s - \mathbf{B}_s \lambda_s\right) \tag{4.19}$$

Equations 4.18 and 4.19 require the use of $\mathbf{K}_s^{-1}$. For a subdomain on which there is an insufficient number of boundary conditions imposed to remove the rigid body motion, $\mathbf{K}_s$ is singular, being positive semi-definite. In Farhat and Roux (1991) the solution taken to overcome the problem of a positive semi-definite $K_s$ uses the pseudo-inverse $\mathbf{K}_s^+$ for $\mathbf{K}_s^{-1}$ which satisfies the Moore-Penrose conditions (Golub and VanLoan, 1989), i.e. $\mathbf{K}_s = \mathbf{K}_s \mathbf{K}_s^+ \mathbf{K}_s$, $\mathbf{K}_s^+ = \mathbf{K}_s^+ \mathbf{K}_s \mathbf{K}_s^+$, and $\mathbf{K}_s \mathbf{K}_s^+$ and $\mathbf{K}_s^+ \mathbf{K}_s$ are Hermitian. For the two subdomain case, where $\Omega_2$ has rigid body modes, the solution presented in Farhat and Roux (1991) proceeds as follows:

A general solution to equation 4.13 is first expressed as:

$$\mathbf{V}_2 = \mathbf{K}_2^+ \left(\mathbf{Q}_2 - \mathbf{B}_2 \lambda_{12}\right) + \mathbf{R}_2 \alpha \tag{4.20}$$

where $\mathbf{R}_2$ is a rectangular matrix whose columns form a basis of the null space of $\mathbf{K}_2$ and $\alpha$ is a vector. $\mathbf{R}_2$ represents the rigid body modes of $\Omega_2$, and $\alpha$ a combination of these modes. Substituting equation 4.20 for equation 4.13, a new interface problem for the two subdomain case can be expressed as:

$$\left(\mathbf{B}_1^T \mathbf{K}_1^{-1} \mathbf{B}_1 + \mathbf{B}_2^T \mathbf{K}_2^+ \mathbf{B}_2\right) \lambda_{12} = \mathbf{B}_1^T \mathbf{K}_1^{-1} \mathbf{Q}_1 + \mathbf{B}_2^T \left(\mathbf{K}_2^+ \mathbf{Q}_2 + \mathbf{R}_2 \alpha\right) \tag{4.21}$$

Since $\mathbf{K}_2$ is symmetric, equation 4.13 has at least one solution if and only if the vector $(\mathbf{Q}_2 - \mathbf{B}_2^T \lambda_{12})$ has no component in the null space of $\mathbf{K}_2$, that is

$$\mathbf{R}_2^T \left(\mathbf{Q}_2 - \mathbf{B}_2 \lambda_{12}\right) = 0 \tag{4.22}$$

Equations 4.21 and 4.22 can be expressed as

$$\begin{bmatrix} \mathbf{B}_1^T\mathbf{K}_1^{-1}\mathbf{B}_1 + \mathbf{B}_2^T\mathbf{K}_2^{+}\mathbf{B}_2 & -\mathbf{B}_2^T\mathbf{R}_2 \\ -\mathbf{R}_2^T\mathbf{B}_2 & \mathbf{0} \end{bmatrix} \left\{ \begin{array}{c} \lambda_{12} \\ \alpha \end{array} \right\} = \left\{ \begin{array}{c} \mathbf{B}_1^T\mathbf{K}_1^{-1}\mathbf{Q}_1 + \mathbf{B}_2^T\mathbf{K}_2^{+}\mathbf{Q}_2 \\ -\mathbf{R}_2^T\mathbf{Q}_2 \end{array} \right\} \tag{4.23}$$

where the matrix is symmetric and nonsingular and, as pointed out in Farhat and Roux (1991), the solution of which uniquely determines $\mathbf{V}_1$ and $\mathbf{V}_2$.

It is interesting to note that no longer is a solution to the interface problem sought in terms of any of the original unknowns. A direct or iterative approach to the solution of the equations can be employed, using the same procedures outlined for substructuring and iterative substructuring. The advantage of an iterative FETI method over the traditional iterative substructuring method is that, for small numbers of subdomains, it converges in fewer iterations. This is due to the fact that the eigenvalues of the FETI interface problem tend to accumulate near the lowest eigenvalue and there are relatively few high eigenvalues (Farhat and Roux, 1994).

### 4.2.2 Domain Partitioning

In order to implement the domain decomposition methods there are several practical matters to be faced by the analyst:

1. How many subdomains to use? The number of subdomains to use depends on the problem size and geometry and the computing resources available. For this work, the number of subdomains is assumed to be at the discretion of the analyst.

2. How to partition the domain? Given the finite element discretization, how is it decomposed into subdomains. The partitioning can be done by the analyst, but for complicated geometries or very large models this is tedious. The domain partitioning can be looked at as a graph partitioning problem. A good partitioning is generally considered to be one that results in the minimum size of the interface with partitions with roughly equal number of elements. Except for the simplest of regular meshes this is an NP-complete problem.

Partitioning algorithms are used in finite element analysis to partition the mesh into subdomains for a non-overlapping domain decomposition methods. Partitioning the mesh is done in one of two ways:

1. **Edge Separator**: Find a subset of element edges such that tearing the mesh along these element edges breaks the mesh into two disconnected submeshes. Popular edge separator algorithms are inertia, greedy (Farhat, 1988), and spectral bisection (Simon, 1991)

2. **Node Separator**: Find a subset nodes such that removing the nodes from the mesh breaks the mesh into two disconnected submeshes. A popular algorithm is nested dissection (George, 1973; George and Liu, 1978).

Most algorithms which have been developed for graph partitioning provide for a two-way partitioning if by edge separator or a three-way partitioning if by vertex separator. To generate further partitions, if more than two subdomains are required by the analyst, the algorithms can be applied recursively. Two recent trends in graph partitioning are:

1. **Local Improvement**: Given an initial partitioning, local improvement heuristics are used to improve the initial partition. A cost function is needed to determine what is an improvement. These local algorithms include Kernighan-Lin (Kernighan and Lin, 1970), Simulated Annealing (Kirkpatrick et al., 1983; Flower et al., 1987; Vanderstraeten and Keunings, 1995), and Stochastic Evolution (Saab and Rao, 1991; Vanderstraeten and Keunings, 1995).

2. **Multilevel Approach**: As graphs get larger and algorithms become more computationally demanding, a considerable amount of time can be spent in the partitioning phase. Instead of working on the fine mesh, the mesh is first coarsened, a partitioning is done on the coarser mesh and the partitioning of the finer mesh is determined from an un-coarsening of the portioned coarsened mesh (Bui and Jones, 1993; Bernard and Simon, 1994; Karypis and Kumar, 1995b)

## 4.3 Existing Object-Oriented Approaches to Domain Decomposition

A review is now made of research on object-oriented applications to domain decomposition. Of interest is how the designs allow for the implementation of various domain decomposition methods and how the domain partitioning is handled. The most notable work in this area has been presented by three groups of researchers:

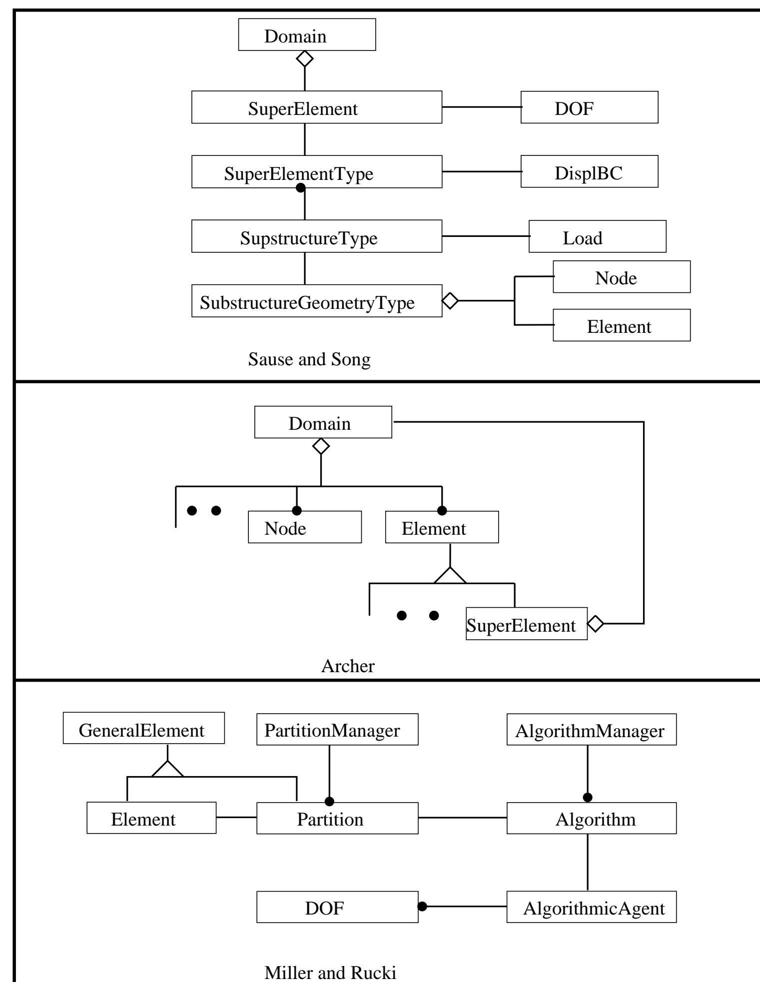



Figure 4.2: Class Diagram for Existing Domain Decomposition Frameworks

1. **Sause and Song**: Sause and Song (1994) presents an object-oriented design for linear static analysis using substructuring. The design handles subdomains with

repeated geometries, which limits the memory requirements for large problems. The classes introduced for substructuring are: **SuperElement**, **SuperElementType**, **SubstructureType**, and **SubstructureGeometryType**. The relationships amongst them and with the other classes in the design are as shown in figure 4.2. The **SubstructureGeometryType** object represents the geometry of a group of substructures. The **SubstructureType** object represents geometry, material and loads. If two substructures have identical load and geometry, they are related by one **SubstructureType** object. The **SuperElementType** represents geometry, material, loads and boundary degrees-of-freedom. It performs the static condensation on the stiffness matrix and residual load vector, which are created by the associated **SubstructureType** object. There exists one **SuperElementType** object for each **SuperElement** object. The **SuperElement** object is responsible for returning the condensed stiffness matrix $\mathbf{K}^*_{ee}$, given in equation 4.7, and condensed residual load vector $\mathbf{P}^*_E$, given in equation 4.8, when getStiff() and getResidual() are invoked by the **Domain** object, which, as in Zimmermann et al. (1992), performs the analysis.

The shortcomings with this architecture, in relation to allowing multiple domain decomposition methods, are:

(a) The design is restricted to linear static analysis problems. The interface could be modified for linear transient analysis but not to non-linear problems involving non-linearities within subdomains.

(b) The interface is restricted to the substructuring and FETI methods, as only getTang() and getResidual() methods are provided at the interface. The iterative substructuring method would require an additional method to get the product of the tangent matrix and a vector.

2. **Archer**: Archer (1996) presents, as an example of the extensibility of the system briefly described in section 3.3, a **Superlement** class. The **Superelement** is a subclass of **Element**, that has a **Domain**. This is as shown in figure 4.2. The interface allows for both linear and non-linear static and transient analysis.

The shortcomings with this architecture, in relation to allowing multiple domain

decomposition methods, are:

(a) The conceptual design is flawed. A **SuperElement** is both an **Element** and a **Domain**. A **SuperElement** does not have a **Domain**, as represented in the relationship between **SuperElement** and **Domain**. This results in excessive method calls as methods that are for the **SuperElement** must be called by the **SuperElement** on the associated **Domain**.

(b) For an incremental transient analysis, as outlined in section 3.2, the design causes excessive numerical computations and numerically inaccurate results. When performing a transient analysis, the **SuperElement** will be asked for its stiffness, mass and damping matrices. The matrices that are returned are $\mathbf{T}^T\mathbf{KT}$, $\mathbf{T}^T\mathbf{MT}$ and $\mathbf{T}^T\mathbf{CT}$, where $\mathbf{T}$ is the transformation matrix obtained in performing the static condensation process on the stiffness matrix. These matrices are then added to the interface problem using:

$$\mathbf{T}^T\mathbf{MTI}_2' + \mathbf{T}^T\mathbf{CTI}_1' + \mathbf{T}^T\mathbf{KT} \tag{4.24}$$

Numerically the correct addition to the interface problem is:

$$\mathbf{T}^{*T}\left(\mathbf{MI}_2' + C\mathbf{I}_1' + \mathbf{K}\right)\mathbf{T}^* \tag{4.25}$$

where $\mathbf{T}^*$ would be the transformation matrix in performing the static condensation on $\mathbf{MI}_2' + \mathbf{CI}_1' + \mathbf{K}$. The two contributions are not the same. Furthermore the amount of computation required to form the contribution given by equation 4.24 is considerably more than that required to form the contribution given by equation 4.25. This can be seen from the simple fact that to obtain $\mathbf{T}$ a static condensation must be performed and then there are the subsequent operations to be performed on $\mathbf{M}$ and $\mathbf{C}$, whereas the contribution given by equation 4.25 is obtained in a static condensation.

3. **Miller and Rucki**: In this work (Rucki, 1996; Rucki and Miller, 1996), the **Partition** class is a subclass of **GeneralElement**, as shown in figure 4.2. Each **Partition** object is associated with a **Algorithm** object, which is responsible for updating the state of a **Partition** so that it will be in equilibrium, as was

discussed in section 3.3. The **Partition** interface was modified to allow a **Partition** to return the unbalanced load and to allow it to install its tangent stiffness at the **DOF** objects (Rucki, 1996). In order for a **Partition** to determine its unbalanced load contribution to the interface problem an analysis is carried out by the **Algorithm** using fixed-point constraints on the interface **DOF** objects. The unbalanced load for the **DOF**s is equal to the reaction at the **DOF**s that such an analysis determines. No discussion is presented for how the **Algorithm** object determines the tangent stiffness coefficients that are required to be installed at the **DOF** objects in a DOF-by-DOF solution strategy.

The shortcomings of this design for performing the analysis was discussed in section 3.3. The need to perform an actual analysis on the **Partition** before the interface solution can be determined is an additional failing with this work.

Also, none of the above mentioned designs have classes to help in the partitioning of the domain into subdomains.

## 4.4 A New Object-Oriented Approach to Domain Decomposition

To overcome the deficiencies in the previous works, the framework shown in figure 4.3 is introduced for domain decomposition. In figure 4.3 new classes not discussed in previous chapters are shaded. The main new classes here are **PartitionedDomain**, **Subdomain**, **DomainDecompAnalysis**, **DomainDecompSolver**, **DomainPartitioner**, and **GraphPartitioner**.

The figure 4.3 also shows some of the descendents, or subclasses, of the new classes. In the following subsections the purpose of each of these new classes is outlined, with pseudo C++ code being presented to demonstrate the functionality of the classes and the interplay between them.

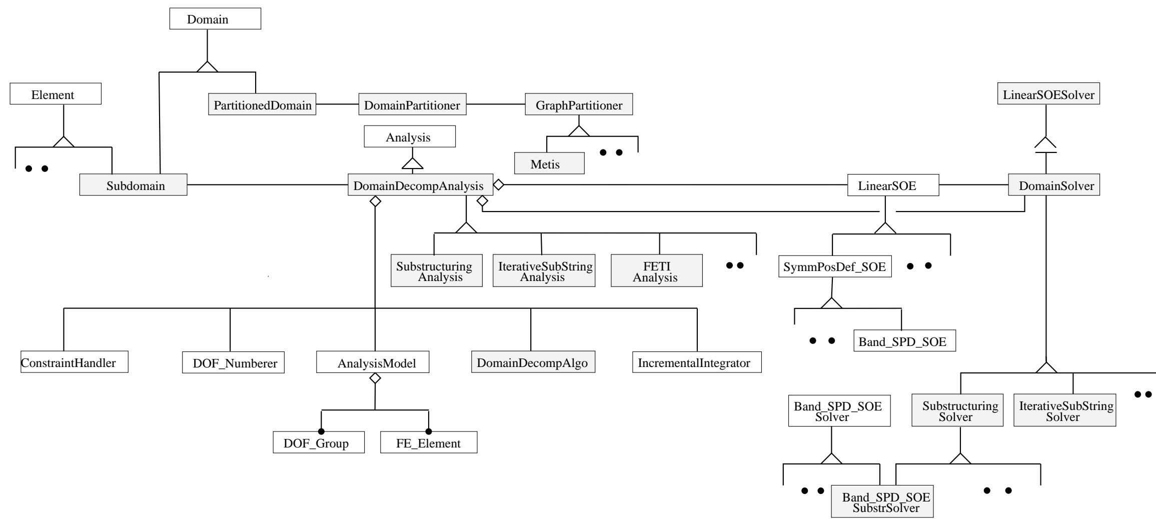



Figure 4.3: Class Diagram for New Domain Decomposition Framework

### 4.4.1 PartitionedDomain Class

A **PartitionedDomain** object is a domain that can be partitioned into **Subdomain** objects. The **PartitionedDomain** class is, as shown in figure 4.3, a subclass of **Domain**. The **PartitionedDomain** class, whose interface is shown in figure 4.4, inherits the regular **Domain** class interface and provides some additional methods:

1. partition(): This is a method for partitioning the **Domain** which invokes partition() on the **DomainPartitioner** object passed as an argument to the constructor, as shown in figure 4.5.

2. addSubdomain(): This is a method for adding **Subdomain**s.

3. getSubdomainPtr() and getSubdomains: These are are methods for accessing the **Subdomains**.

**Subdomain**s, while they could have been treated as regular elements, are treated specially in the design for two reasons: (1) efficiency, as the **DomainPartitioner** needs quick access to the **Subdomain**s; and (2) identification, as in the implementation used to validate the design, the elements in the Domain must have unique identifiers and the **DomainPartitioner** assumes that the **Subdomain**s have tags 1

```
class PartitionedDomain: public Domain {
    public:
        PartitionedDomain(DomainPartitioner &thePartitioner);
        virtual ~PartitionedDomain();

        // public member functions in addition to the standard domain
        virtual int partition(int numPartitions);
        virtual bool addSubdomain(Subdomain *subPtr);
        virtual int getNumSubdomains(void);
        virtual Subdomain *getSubdomainPtr(int tag);
        virtual SubdomainIter &getSubdomains(void);
};
```

Figure 4.4: Interface for the **PartitionedDomain** Class

```
PartitionedDomain::partition(int numPartitions) {
    theDomainPartitioner->partition(numPartitions);
}
```

Figure 4.5: Pseudo-Code for the **PartitionedDomain** Classes partition Method

through $ns$, which would have meant that none of the elements previously added to the **PartitionedDomain** would have been permitted to have used these identifiers.

### 4.4.2 DomainPartitioner Class

The **DomainPartitioner** object is responsible for partitioning a **PartitionedDomain** object. The **DomainPartitioner** class, whose interface is shown in figure 4.6, provides two methods: setPartitionedDomain() which is invoked by the **PartitionedDomain** object during its construction to set the link between the two objects; and partition(). To partition the **PartitionedDomain**, the **DomainPartitioner** obtains the element graph from the **PartitionedDomain** and then uses its associated **GraphPartitioner** object, which is passed as an argument to the constructor, to partition this graph. With the partititioned element graph, the **DomainPartitioner** is responsible for removing **Node**s, **Element**s, **Load**s, and **Constraint**s from the **PartitionedDomain** and adding them to the appropriate **Subdomain**s.

```
class DomainPartitioner {
    public:

        DomainPartitioner(GraphPartitioner &theGraphPartitioner);
        virtual ~DomainPartitioner();

        virtual void setPartitionedDomain(PartitionedDomain &theDomain);
        virtual int partition(int numParts);
};
```

Figure 4.6: Interface for the **DomainPartitioner** Class

### 4.4.3 GraphPartitioner Class

The **GraphPartitioner** object is responsible for partitioning a **Graph** object, i.e. coloring the **Graph**. The **GraphPartitioner** class, whose interface is shown in figure 4.7, is an abstract class which defines the single method partition() as being pure virtual. It is the subclasses of **GraphPartitioner** that provide the implementation of this method. The partitioning strategies that can be used by these subclasses were

discussed in section 4.2.2. Additional partitioning strategies can also be developed by the analyst, to take advantage of the particular geometry of the model being analyzed.

```
class GraphPartitioner {
      public:
            GraphPartitioner();
            virtual ~GraphPartitioner();

            virtual int partition(Graph &theGraph, int numPart) =0;
};
```

Figure 4.7: Interface for the **GraphPartitioner** Class

As with the **SystemOfEqn** and **Solver** subclasses, the **GraphPartitioner** interface was designed so that the subclasses do not need to know anything about the finite element method. This allows work provided by researchers in other fields to be used. For example, for the test implementation to verify the design presented in this research, an object-oriented interface for the METIS partitioning library (Karypis and Kumar, 1995a), **Metis**, was developed. In the current design, an unweighted **Graph** object is supplied to the **GraphPartitioner**. This could be changed to a weighted **Graph**, in which the vertex weights correspond to the cost of element computations or represent the state of the element, and the edge weights are related to boundary conditions.

### 4.4.4 Subdomain Class

The **Subdomain** class inherits from both **Element** and **Domain**, as shown in figure 4.3, that is a **Subdomain** is both an **Element** and a **Domain**. The class interface, which is as shown in figure 4.8, provides a number of additional methods in the interface. These additional methods can be split into two groups:

1. For efficiency when performing the domain decomposition analysis, the methods addNode(), addExternalNode(), getInternalNodeIter(), and getExternalNodeIter are introduced. These allow the **Subdomain** objects to keep track of which nodes are internal and which nodes are on the interface. This information is available

```
class Subdomain: public Element, public Domain {
    public:
        Subdomain(int tag);
        virtual ~Subdomain();

        // public methods which are new or modified in their meaning
        // from a regular Domain for efficiency reasons
        virtual NodeIter &getInternalNodeIter(void);
        virtual NodeIter &getExternalNodeIter(void);
        virtual bool addNode(Node *);
        virtual bool addExternalNode(Node *);

        // methods which are new for Subdomain type Elements
        virtual void setAnalysis(DomainDecompAnalysisAnalysis &theAnalysis);
        virtual int computeTang(void); // to form (4.25)
        virtual int computeResidual(void);
        virtual int computeTangForce(const Vector &x);
        virtual const Vector &getLastTangForce(void);
        virtual const Matrix &getTang(void);
        virtual const Matrix &getResidual(void);

        // methods to compute the response at the subdomain nodes
        // once the interface problem has been solved
        void setFE_ElementPtr(FE_Element *theFE_Ele);
        const Vector &getLastExternalSysResponse(void);
        int computeNodalResponse(void);
};
```

Figure 4.8: Interface for the **Subdomain** Class

in the **DomainPartitioner** object, as a result of the partitioning. Providing this information to the **Subdomain** means that the **Subdomain**s do not have to independently determine the interface **Node**s.

2. Additional element type methods have been added to the interface to overcome the numerical problems, discussed in section 4.3, that inheriting from the **Element** class would cause. The new Element type methods are:

   (a) computeTang(), computeResidual(), and computeTangForce(), which are methods to perform the computation of the tangent $\mathbf{K}^*_{ee}$, given by equation 4.7, the residual $\mathbf{P}^*_e$, given by equation 4.8, and the tangent-vector product $\mathbf{K}^*_{ee}\mathbf{X}_e$, given by equation 4.10.

   (b) getTang(), getResidual(), and getTangForce(), which are methods to return the results of the computations performed by the three previous methods.

   (c) setFE_elementPtr(), getLastExternalSysResponse() and computeNodalResponse() which are methods used to compute the response at the **Node**s in the **Subdomain** given the solution to the interface problem.

   (d) setAnalysis() which is used to set the associated **DomainDecompAnalysis** object.

The Subdomain class itself does not perform the numerical computations associated with these methods. Invocations of these methods cause similar methods to be invoked on the **DomainDecompAnalysis** object associated with the **Subdomain**, as shown in figure 4.9, were code fragments are presented for formTang() and getTang() methods. With this design the analyst is able to experiment with different domain decomposition methods without the need to modify the **Subdomain** class. This follows the design principle behind associating an **Analysis** object with the **Domain** object seen in chapter 2.

```
int Subdomain::formTang(void) {
    return theAnalysis->condenseA();
}

const Matrix & Subdomain::getTang(void) {
    return theAnalysis->getCondensedA();
}
```

Figure 4.9: Psuedo-Code for Selected Methods of the **Subdomain** Class

### 4.4.5 DomainDecompAnalysis Class

The **DomainDecompAnalysis** object associated with the **Subdomain** is an aggregation of objects of the following types: **ConstraintHandler**, **DOF_Numberer**, **AnalysisModel**, **IncrementalIntegrator**, **LinearSOE**, **DomainDecompAlgo**, and **DomainSolver**. It is the objects in the aggregation that perform the numerical computations required of a domain decomposition analysis. These objects are created by the analyst and passed to the **DomainDecompAnalysis** as arguments in the constructor. This approach, which is similar to the approach taken to the **Analysis** classes presented in chapter 3, allows the analyst to select the domain decomposition method, storage scheme for the system of equations, and solver used in the domain decomposition analysis.

The **DomainDecompAnalysis** class, whose interface is shown in figure 4.10, provides:

1. a constructor which will verify the objects passed as arguments are of the correct type.

2. domainChanged(), a method which is invoked by the **DomainPartitioner** object on completion of the partitioning to set links and invoke set up functions on the objects in the aggregation.

3. getNumExternalEqn(), formTangent(), formResidual(), formTangVectProduct(), getTangent(), getResidual(), and getTangVectProduct(), which are the methods invoked by the **Subdomain** object, as was discussed in the previous section, to

perform the numerical computations and to return the results of these computations.

```
class DomainDecompAnalysis: public StaticAnalysis {
    public:
        DomainDecompAnalysis(Subdomain &theDomain,
                             ConstraintHandler &theHandler,
                             DOF_Numberer &theNumberer,
                             AnalysisModel &theModel,
                             DomainDecompAlgo &theSolnAlgo,
                             LinearSOE &theSOE,
                             IncrementalIntegrator &theIntegrator,
                             DomainSolver &theSolver);

        virtual ~StaticCondensationAnalysis();

        virtual int analyze(void);
        virtual int domainChanged(void);

        // methods called by the subdomain object
        virtual int getNumExternalEqn(void);
        virtual int computeInternalResponse(void);
        virtual int formTangent(void);
        virtual int formResidual(void);
        virtual int formTangVectProduct(Vector &force);
        virtual const Matrix &getTangent(void);
        virtual const Vector &getResidual(void);
        virtual const Vector &getTangVectProduct(void);
};
```

Figure 4.10: Interface for the **DomainDecompAnalysis** Class

.

The actual numerical computations are performed by the objects in the aggregation at the request of the **DomainDecompAnalysis** object, as shown in figure 4.11. For example the **IncrementalIntegrator** object is responsible for forming $\mathbf{K}$, equation 3.19, and $\mathbf{P}$, equation 3.20. The subsequent forming and retrieval of of $\mathbf{K}^*_{ee}$, equation 4.7, $\mathbf{P}^*_e$, equation 4.8, and the product $\mathbf{K}^*_{ee}\mathbf{X}_e$, equation 4.10, are the responsibility of the **DomainSolver** object.

```
int DomainDecompAnalysis::formTangent(void) {
    theIntegrator->formTangent();
    return theDomainSolver->condenseA(numIntEqn);
}

const Matrix & DomainDecompAnalysis::getTangent(void) {
    return theDomainSolver->getCondensedA();
}

int DomainDecompAnalysisAnalysis::computeInternalResponse(void) {
    return theAlgo->solveCurrentStep();
}
```

Figure 4.11: Pseudo-Code for Selected Methods for the **DomainDecompAnalysis** Class

### 4.4.6 DomainDecompAlgo Class

The **DomainDecompAlgo** object is responsible for updating the response at the **Node** objects in the **Subdomain**.The **DomainDecompAlgo**, whose interface is shown in figure 4.12, provides the methods: setLinks(), which will set the links to the other objects; and solveCurrentStep(), which will cause the response quantities at the **Node**s in the **Subdomain** to be updated. To update the nodal responses, the **DomainDecompAlgo** object asks the **Subdomain** for the last interface response for the **Subdomain**s external degrees-of-freedom. It then invokes methods in the **DomainSolver** object to set the response at the external degrees-of-freedom (the $\lambda$ if a **FETI_Solver** is being used) and to determine the response at the internal degrees-of-freedom (the internal and external degrees-of-freedom if a **FETI_Solver**). The **Integrator** object is then asked to update the nodal responses. This is as shown in figure 4.13.

### 4.4.7 DomainSolver Class

The **DomainSolver** object is responsible for performing all the numerical computations required for a domain decomposition analysis. As the computations depend on the type of domain decomposition method, a subclass is provided for each type of

```
class DomainDecompAlgo: public StaticEquiSolnAlgo {
    public:
        DomainDecompAlgo();
        virtual ~DomainDecompAlgo();

        virtual int solveCurrentStep(void);

        void setLinks(DomainDecompAnalysis &theAnalysis,
                      AnalysisModel &theModel,
                      IncrementalIntegrator &theIntegrator,
                      LinearSOE &theSOE,
                      DomainSolver &theDomainSolver,
                      Subdomain &theSubdomain);
};
```

Figure 4.12: Interface for the DomainDecompAlgo Class

```
DomainDecompAlgo::solveCurrentStep(void) {

    const Vector &extResponse = theSubdomain->getLastExternalSysResponse();

    theDomainSolver->setComputedXext(extResponse);
    theDomainSolver->solveXint();

    theIntegrator->update(theLinearSOE->getX());
}
```

Figure 4.13: Pseudo-Code for the **DomainDecompAlgo** Classes solveCurrentStep Method

domain decomposition method the analyst would wish to employ. Examples of the subclasses, as shown in the figure 4.3, include **SubstructuringSolver**, **IterativeSolver**, and **FETI_Solver**. Each of these are themselves abstract classes. Subclasses must be written for each subclass of **LinearSOE** that the analyst wishes to use for efficiency reasons. This is because the operations performed by the **DomainSolver** objects are numerically intensive and for large problems will dominate the computation. While generic solvers could be supplied, they would be too slow. Specific solvers for each subclass can take advantage of the sparsity of each type of **LinearSOE** class.

```
class DomainSolver : public LinearSOESolver {
    public:
        DomainSolver();
        virtual ~DomainSolver();

        virtual int condenseA(int numInt) =0;
        virtual int condenseRHS(int numInt, Vector *anotherRHS = 0) =0;
        virtual int computeCondensedMatVect(int numInt, const Vector &u) =0;

        virtual const Matrix &getCondensedA(void) =0;
        virtual const Vector &getCondensedRHS(void) =0;
        virtual const Vector &getCondensedMatVect(void) =0;

        virtual int setComputedXext(const Vector &) =0;
        virtual int solveXint(void) =0;
};
```

Figure 4.14: Interface for the **DomainSolver** Class

The **DomainSolver** class, whose interface is shown in figure 4.14, provides methods for the following:

1. Methods are provided to form and retrieve $\mathbf{K}^*_{ee}$, equation 4.7, $\mathbf{P}^*_e$, equation 4.8, and the product $\mathbf{K}^*_{ee}\mathbf{X}_e$, equation 4.10. If a FETI method is being used, the **FETI_Solver** will instead form $B_s^T K_s^{-1} B_s$, equation (4.18), $B_s^T K_s^{-1} Q_s$, equation (4.18), and $B_s^T K_s^{-1} B_s \mathbf{\Lambda}$ for the **Subdomain** $s$.

2. setComputedXext(): This is a method which is used by the **DomainDecompAlgo** object to set $\mathbf{X}_e$ in the **LinearSOE** object. If a **FETI_Solver** has been employed, this will set $\lambda$.

3. solveXint(): This is a method to solve for $\mathbf{X}_i$, equation (4.9), a **FETI_Solver** will solve for $\mathbf{X}$, equation (4.19).

## 4.5 Modifications to Classes for Domain Decomposition

To introduce domain decomposition methods into the analysis framework, certain modifications to existing classes are necessary to avoid the numerical problems discussed in section 4.3. The **Element** class interface has to be extended and the **FE_Element** methods have to be modified.

The revised **Element** interface, which is shown in figure 4.15 has an additional method, isSubdomain(). This method allows **FE_Elements** to determine if the **Element** they are associated with is a **Subdomain**.

The **FE_Element** routines are now modified to account for this additional method and to account for the methods provided in the **Substructure** interface for domain decomposition. For example when formEleTang() is invoked on a **FE_Element** object, the **FE_Element** object, will ask the **Integrator** object to form the **Element's** contribution to the tangent, or if the **Element** is a **Subdomain** it will ask the **Subdomain** to form this contribution. This is shown in figure 4.16.

```
class Element : public DomainComponent {
    public:
        Element(int tag);
        virtual ~Element() ;

        virtual int getNumExternalNodes(void) const =0;
        virtual const ID &getExternalNodes(void) =0;

        // pure virtual functions
        virtual int getNumDOF(void) =0;
        virtual int computeState(void) = 0;
        virtual void commitState(void) = 0;

        virtual const Matrix &getStiff(void)=0;
        virtual const Matrix &getDamp(void)=0;
        virtual const Matrix &getMass(void)=0;

        virtual void zeroLoad(void) =0;
        virtual int addLoad(const Vector &load) =0;
        virtual const Vector &getResistingForce(void) =0;

        virtual void commitState(void) = 0;

        virtual bool isSubdomain(void) =0;
};
```

Figure 4.15: Revised Interface for the **Element** Class

```
FE_Element::formTangent(Integrator *theNewIntegrator) {
    if (myEle ≠ 0) // myEle is a pointer to the associated element
        if (myEle->isSubdomain() == false)
            theIntegrator->formEleTangent(this);
        else {
            Subdomain *theSub = (Subdomain *)myEle;
            theSub->computeTang();
        }
}
```

Figure 4.16: Pseudo-Code for the **FE_Element** Classes formTangent Method

## 4.6 Example Programs using Domain Decomposition

To demonstrate the flexibility of this approach, a base pseudo C++ program is first presented. Modifications to this program are then made to produce new programs, which perform different analysis to that performed by the base program. The base program performs a transient analysis of a space shuttle model using the Newmark integration strategy, a Newton-Raphson iteration at each time step, and the substructuring method. Four subdomains are created, each of which uses a reverse Cuthill-McKee numbering scheme to order the degrees-of-freedom and a profile storage scheme to store the subdomain equations. The interface problem, which uses a reverse Cuthill-McKee numbering scheme and a banded storage scheme to store the equations, is solved by a direct method. The pseudo-code for the base program is as follows:

```
001    numSubdomains = 4;
002    /* create the partitioned domain and model builder */
003    Metis theGraphPartitioner;
004    DomainPartitioner thePartitioner(theGraphPartitioner);
005    PartitionedDomain theDomain(thePartitioner);
006
007    /* create the subdomain and add to the domain
008
009
010    for (int i=1; i<=numSubdomains; i++) {
011
012        Subdomain theSubdomain(i)
013        Transformation theConstraintHandler;
014        RCM theDOFNumberer;
015        AnalysisModel theModel;
016        ProfileSPDSOE_Substr_Solver theSolver;
017        ProfileSPDSOE theLinearSOE(theSolver);
018        Newmark theIntegrator(1/4, 1/2);
019        DomainDecompAlgo theSolnAlgo;
020        DomainDecompAnalysis theAnalysis(theSubdomain, theConstraintHandler,
021            theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinearSOE);
022        theDomain.addSubdomain(theSubdomain);
023    }
024
025    /* create a model builder and build the model */
026    SpaceShuttle theModelBuilder(theDomain);
027    theModelBuilder.buildModel();
028
```

```
029       /* partition the domain into the subdomains */
030       theDomain.partition(numSubdomains);
031
032       /* create the analysis */
033       Transformation theConstraintHandler;
034       RCM theDOFNumberer;
035       AnalysisModel theModel;
036       DirectBandSPDSOE theSolver;
037       BandSPDSOE theLinearSOE(theSolver);
038       Newmark theIntegrator(1/4, 1/2);
039       NewtonRaphson theSolnAlgo;
040       DirectIntegrationAnalysis theAnalysis(theDomain,theConstraintHandler,
041           theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinearSOE);
042
043       /* perform the analysis */
044       theDomain.setLoadCase(1);
045       theAnalysis.analyze;
046
```

To change this code to use the FETI domain decomposition method the analyst replaces line 016 with the following:

```
016       ProfileSPDSOE_FETI_Solver theSolver;
```

If the analyst wishes instead to use an iterative substructuring approach, which uses the element-by-element conjugate gradient method to solve the interface problem, lines 016, 036 and 037 are replaced with the following:

```
016       ProfileSPDSOE_IterativeSubstr_Solver theSolver;
036       EleByEleSPDSOE theSolver;
037       EleByEleSOE theLinearSOE(theSolver, theModel);
```

# Chapter 5

# Parallel Object-Oriented Finite Element Programming

This chapter presents an object-oriented approach for parallelizing the finite element method. A brief introduction to parallel programming is first presented. A review is then made of existing research on parallelizing the finite element method to identify the approaches typically used. A parallel object-oriented programming model suitable for the finite element method is presented, which is a modified version of the actor model, a popular parallel object-oriented programming model. A framework is provided for this new model and a discussion of what changes are required to the design presented in the previous chapters is given. The chapter concludes with an example of parallel object-oriented analysis in the prototype implementation.

## 5.1 Introduction

The finite element analysis of large problems can easily exhaust both the patience of the analyst and the CPU and memory resources of conventional single processor computers. In a period of a few short years parallel computers have become a reality as a variety of parallel computers have become commercially available and, more importantly for practicing engineers, software has become available which allows networks of workstations and microcomputers to be programmed as parallel machines, e.g. PVM (Sunderam, 1990; Sunderam et al., 1994), PARMACS (Calkin et al., 1994), Berkeley Sockets (Stevens, 1990), Munin (Carter et al., 1995), and LAM (Nevin, 1996) and MPICH (Gropp et al., 1996), which support the message passing interface standard MPI (Walker, 1994).

A considerable amount of research has been presented describing finite element analysis for parallel computers; Mackerle (1996) identifies over seven hundred papers on the subject. At the present time, however, there exists only a small number of commercially available packages which run on a few parallel machines. These packages include ADINA, which offers a version for the SGI PowerChallenge and the HP Exemplar, ANSYS which offer versions for the Cray C90 and Y-MP machines, the HP Exemplar and a variety of workstation networks running under Unix, MARC which offer versions for the Cray C90 and Y-MP machines, the IBM SP2, the SGI Origin 2000, the HP Exemplar and a variety of workstation networks, and MSC/NASTRAN which offer versions for the Cray C90 and Y-MP machines and the HP Exemplar machine. The reason for the small number of software packages and the limitation of these software packages to a few machines can be attributed to a number of factors:

1. Much of the research was performed using machines that are no longer in use today. This is because many current workstations can outperform these parallel machines of yesterday.

2. The ability to port code from one machine to another is limited. This is because compilers and libraries available on different parallel machines are often vendor proprietary, for example IBM offers EUI for programming on its SP1 and SP2 machines, Intel offers NX2 for its iPSC and Paragon machines, Thinking Ma-

chines offered CMMD for its CM2 and CM5 machines and nCUBE offered PSE for its nCUBE1 and nCUBE2 machines.

3. The differences in the relative hardware performance of the components of the parallel machines, i.e. processors, memory and communication interconnection, result in algorithms performing well on certain machines but performing poorly on other machines.

4. The code was written in procedural languages, which result, as was discussed in section 1.1, in programs which are inflexible and difficult to extend.

As languages, e.g. Split-C (Culler et al., 1993) and HPF (HPF Forum, 1993), and software packages, e.g. PVM, LAM and MPICH, become more widely available for a variety of parallel machines, parallel programs can now be written which can be ported to different parallel machines. In order to make efficient use of these machines, however, it is not enough that the packages be portable. This is because the performance of the analysis algorithms varies greatly between parallel machines (Farhat, 1990b). For this reason the software packages that are developed for parallel machines must be flexible, allowing the analyst to choose the appropriate algorithm for specific problems and specific machines. The software packages must also be extensible, allowing the analyst to introduce new routines into the package to account for changes in hardware. Current parallel packages do not provide this flexibility and extensibility. For example, MARC only provides a parallel sparse solver for each parallel machine it is supported on.

It will be demonstrated in this chapter that a flexible and extensible approach for programming on parallel machines can be achieved using an object-oriented design. The design is based on the classes presented in the previous chapters and new classes which are introduced for parallel programming. The flexibility and extensibility of the design presented in the previous chapters is maintained in this new design. In addition, the design allows for flexibility and extensibility where machine hardware and software are concerned, allowing the analyst to choose the communication software and protocols to use, and to determine on which processors of the parallel machine to run the processes. In section 5.2 a brief introduction to parallel computing is

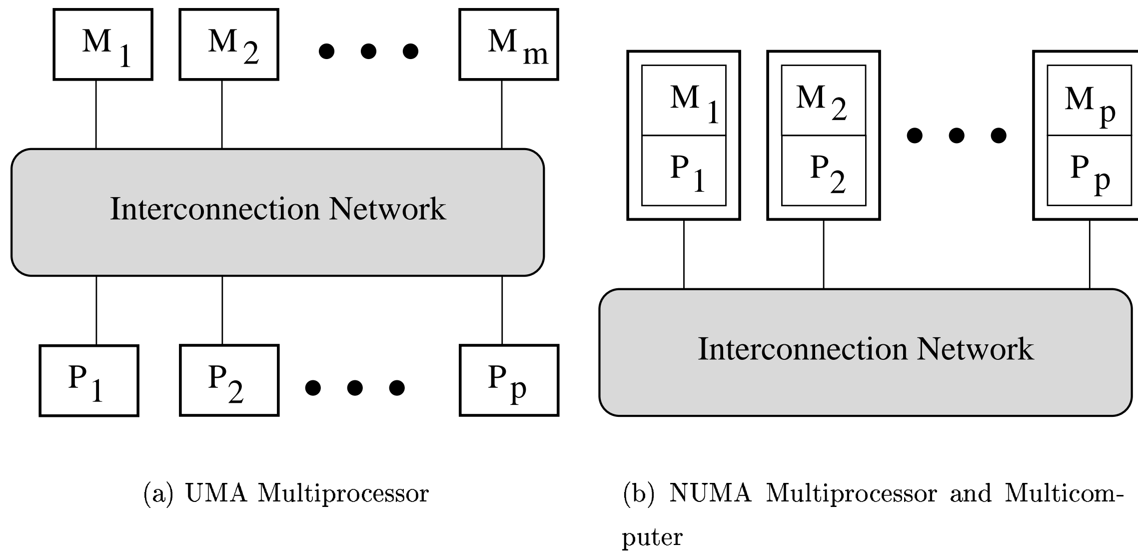
presented, in which a review of current parallel architectures, programming models and programming methods is given. In section 5.3 a review is made of some existing work on parallel finite element analysis. This review of existing work is presented in order to identify the approaches that are typically used to parallelize the finite element analysis. In section 5.4 a slightly modified parallel programming model is presented for the object-oriented approach that is presented in this dissertation. The modified model minimizes changes to the existing design and allows for more efficient programs. In section 5.5 the new classes introduced for a parallel object-oriented finite element framework are defined. In section 5.6 a discussion is presented on the changes to existing class interfaces and the issues the analyst must be aware of when programming in a parallel environment. In section 5.7 a prototype implementation demonstrates the new design.

## 5.2 Summary of Parallel Computing

### 5.2.1 Parallel Architectures

A parallel computer, as defined by Almasi and Gottlieb (1989), is *a collection of processing elements that cooperate and communicate to solve large problems fast.* Most of today's parallel computers can be viewed as a collection of processors and memory units which are linked together by an interconnection network. There are three prevalent parallel architectures in use today: uniform memory access (UMA) multiprocessor; non-uniform memory access (NUMA) multiprocessor; and multicomputer.

1. **UMA Multiprocessor**: A UMA multiprocessor is a parallel architecture in which $p$ processors and $m$ memory units are linked together by an interconnection network, as shown in figure 5.1a. To the processors in a UMA multiprocessor, the memory units make up one large memory to/from which all processors can store/get data. The access time for each process to any location in memory is the same for all processors. Examples of parallel machines with this architecture are the Alliant FX-8, the Cray X-MP, the Cray Y-MP, the Cray C90, the HP Exemplar and the Sequent Symmetry.




(a) UMA Multiprocessor

(b) NUMA Multiprocessor and Multicomputer

Figure 5.1: Computer Architecture for Parallel Computers

2. **NUMA Multiprocessor**: A NUMA multiprocessor is a parallel architecture in which a collection of processing units is interconnected via an interconnection network, as shown in figure 5.1b. Each processing unit consists of a memory unit and at least one processor, sometimes more. To the processors, as in the UMA multiprocessor, the memory units make up one large memory to/from which all processors can store/get data. The difference with the UMA multiprocessor is that the time to access data in a local memory unit is considerably faster than accessing data in a remote memory unit. Typically in such systems when a page fault occurs, the page is copied from remote memory and a coherency policy is enforced on duplicate pages in the system. This memory management is all performed by the hardware.

   The NUMA multiprocessor architecture is more popular than the UMA multiprocessor architecture for the building of machines with large numbers of processors because the architecture scales better. Examples of parallel machines with this architecture are the Cray T3D, the SGI Origin 2000, and the Stanford Dash.

3. **Multicomputer**: The multicomputer is similar to that of a NUMA multiprocessor. A multicomputer is a parallel architecture in which a number of processing units is interconnected via an interconnection network, as shown in figure 5.1b. Each processing unit consists of a memory unit and a processor. In a multicomputer, unlike the NUMA multiprocessor, each processing units can only store/get data to/from its local memory unit. Each processing unit acts as a standard sequential computer. In order for a process, running on a processing unit, to look at data in another process'es address space communication between the processes must occur.

   The advantage of the multicomputer architecture over the NUMA multicomputer is that off the shelf components can be used. For example, the IBM SP1, the IBM SP2, and the HP 735CL are parallel machines comprised of several workstations bundled inside a single cabinet. Examples of parallel machines with this architecture are the Intel Paragon, the Intel iPSC, the nCUBE nCUBE2, the Thinking Machines CM-5, the IBM SP1, the IBM SP2, the HP 735CL, and networks of workstations (NOWs).

The parallel computer architectures can be further classified depending on the number of processors in the machine. The machines can be classified as being:

1. **Coarse Grained**: The machine offers the user only a small number of powerful processors, such as the Cray-XMP(4 processors), the Cray-YMP(8 processors), the Cray C90(16 processors), and the Alliant FX-8(8 processors).

2. **Fine Grained**: The machine offers a large number of small processors, e.g. the CM-2 had up to 65536 processors.

3. **Medium Grained**: In between fine and coarse grained, the machine offers from a few dozen to a few thousand processors, such as the nCUBE(1024 processors), the iPSC(128 processors), and the Cray T3D(2048 processors).

### 5.2.2 Parallel Programming Models

A parallel machine, like all computers, requires an operating system, compilers and software packages in order to be useful. Collectively they provide the program-

mer with a programming model. For parallel machines there are primarily three programming models:

1. **Message Passing**: A running program is viewed as a collection of independent communicating processes. Each process executes in its own address space and has a unique identifier which allows the other processes to identify it for purposes of communication. The processes communicate with each other by the sending and receiving of data, i.e. message passing. The sending process specifies the local data to be transmitted and the address of the receiving process(s). The receiving process specifies the sending process and where the data is to be placed in its local address space.

   The operating system, compilers, and software packages provide functions that the processes can use to perform the communication. The operating system and software packages also provide functions which allow processes to create other processes. The functions appear to the processes as normal library calls. Examples of operating systems which provide this functionality are Unix and Plan-9. Examples of software packages which provide this are PVM, LAM and MPICH.

2. **Shared Memory with Threads**: The running program is viewed as a collection of processes each sharing a portion of its virtual address space with the other processes, i.e. a collection of threads. Each process has private data, such as the process stack, and shared data, which is in the same region of the virtual address space of each process. Access to the shared data must be synchronized between processes to prevent race conditions. This synchronization, the communication between processes in a shared memory programming model, is performed by processes using locks, barriers, condition variables, and other synchronization operators. Examples of software packages that support this are Munin and Express. An Example of a language which supports this model is Split-C.

3. **Data Parallel**: The running program is viewed as a single process. Parallel processing occurs when the process performs an operation on an array. Op-

erations on arrays are performed in parallel on a number of processors. The communication between processors is implicit, the programmer is not responsible for sending data between processors or synchronizing the access to the data in the arrays. Examples of languages which support this model are HPF and CM Fortran.

Certain programming models are suited to certain architectures, such as message passing for multicomputers and shared memory for multiprocessors. A number of software packages allow programmers to use a certain programming model on either multiprocessor or multicomputer machines. For example, PVM (Beguelin et al., 1993), a software package that supports the message-passing model, is available on both multiprocessor machines, such as NOWs, the Intel iPSC, and the Thinking Machines CM-5, and multiprocessor machines, such as the Cray Y-MP and the Sequent Symmetry. An example of software packages which provide programmers with the shared memory model on multicomputer NOWs are IVY (Li and Hudak, 1989) and Munin (Carter et al., 1995).

### 5.2.3 Parallel Programming

It is more difficult to write fast programs for parallel machines than it is to write fast sequential programs for uniprocessor machines. The reason for this is that the programmer has a lot more resources available than just a single processor and a single memory unit. In addition to the usual issues that need to be addressed in developing fast programs, e.g. data locality, the programmer in a parallel environment must also consider:

1. **Load Balance**: To make efficient use of the parallel machine, the programmer must keep all the processors busy at all times. A load imbalance occurs when some processors sit idle while others remain busy, degrading the efficiency of the computation.

2. **Communication**: The processes running on the parallel machine need to communicate to perform the work required of them. The communication between processes is expensive compared to normal process operations. The programmer

should minimize communication between processes and must also coordinate the timing of the communication between processes.

To aid in the development of parallel programs, as Culler et al. (1997) discuss, four steps can be identified:

1. **Decomposition**: The computation is broken down into a number of tasks representing discrete portions of the computation. For example, in a finite element analysis, the tasks include the formation of the system of equations, the solution of this system and the updating of the nodal response quantities. Each of these tasks can be subdivided into smaller tasks. For example, the formulation of the system of equations may be divided into the separate tasks at the element level, i.e. the formulation of mass, stiffness and damping matrices, the formulation of the element tangent matrix and its assembly into the system of equations. An important factor in determining what level of decomposition is necessary, is that the decomposition must identify tasks that can execute concurrently.

2. **Assignment**: The programmer is responsible for creating the processes in which the tasks will be performed. In order to reduce communication between processes a number of tasks are assigned to each process. It is important in the assignment that:

   (a) To minimize load imbalance, that the tasks that can be executed concurrently should be assigned to different processes and that the workload among the processes is balanced.

   (b) To reduce communication between processes, tasks which reference common data should be assigned to the same process.

3. **Orchestration**: The processes must communicate to perform the tasks assigned to them. The communication is in the form of messages in a message passing environment and synchronization in a shared memory environment. The programmer is responsible for determining the order in which tasks within a process execute and when the processes communicate on behalf of the tasks.

4. **Mapping**: The processes must be assigned to the physical processors. For coarse and medium grained machines, where the number of processes may exceed the number of processors or in heterogeneous environments where the relative performance of processing units may vary, this mapping can have a significant impact on the overall performance.

It is important to decompose the problem into tasks that can be performed concurrently; that those tasks that can be executed concurrently be assigned to different processes or threads; and that processes or threads requiring the most CPU time be assigned to the faster processors. As Amdahl (1968) pointed out, if the computation is divided into the serial portion, that portion for which no tasks can execute concurrently, and the concurrent portion, that portion for which tasks can execute concurrently, then no matter how high the degree of concurrency in the concurrent portion, the performance will be limited by the serial portion. This is commonly referred to as Amdahl's law.

If the time to perform the computation, $T$, is divided into the serial time $T_s$, the time to execute the serial portion, and the concurrent time $T_c$, the time to execute the concurrent portion, then the total time to execute the program on a single processor can be expressed as

$$T_1 = T_s + T_c$$

The time to execute on $p$ processors, *assuming that communication costs are negligible and access to data at all levels of the memory hierarchy is uniform*, can be expressed as

$$T_p = T_s + \frac{T_c}{p}$$

Letting $r = T_c/T_s$, the *algorithmic speedup* for p processes $AS_p$ is defined as

$$AS_p = \frac{T_1}{T_p} = \frac{1+r}{1+r/p}$$

An upper-bound on the algorithmic speedup that can be obtained is expressed as

$$\lim_{p \to \infty} AS_p = 1 + r$$

For example if $r = 0$, i.e. no concurrency exists in the program, then no matter how many processes are used, no increase in performance by using more than one processor

will be obtained. If $r = 1$, i.e. 50% of program is sequential and 50% concurrent, the limit on the amount of algorithmic speedup that can be obtained is 2 and for two processors the algorithmic speedup that can be obtained is 1.6, which increases only to 1.8 for eight processors. If $r = 4$ the limit is increased to 5, with an algorithmic speedup of 1.7 being obtained for two processors and 3.3 for eight processors.

For this reason the best sequential algorithm is not always the best parallel algorithm. There may exist an algorithm which is slower on a sequential machine but due to concurrency in the algorithm, it will perform better in a parallel environment. A metric that is used to compare the performance of various algorithms is the speedup. The *speedup* of an algorithm using p processors $S_p$ is defined as (Codenotti and Leoncini, 1993; Kumar et al., 1994; Quinn, 1994):

$$S_p = \frac{T}{T_p}$$

where T is the time taken to perform the computation using the *best* sequential algorithm on a single processor and $T_p$ is the time taken to perform the computation using the algorithm under consideration on $p$ processors.

The algorithmic speedup and speedup of an algorithm will vary from one parallel machine to another. This is because the relative performance of the processors, memory, and communication can be substantially different on different parallel machines. These speedup metrics will also vary with the problem even on the same parallel machine because for different problems the ratio of communication to numerical operations, the ratio of the serial portion to the concurrent portion, and the number of page faults and cache misses change.

### 5.2.4 Parallel Object-Oriented Computing

The object-oriented paradigm is ideally suited to the development of parallel programs because the tasks can be identified as the invocation of the object methods, and the assignment of tasks that share common data to a process can be identified as assigning an object to a process. For the development of parallel object-oriented programs, a number of experimental object-oriented languages have been proposed in the literature. These languages support two main programming models:

1. **Actor Model**: Actors (Agha, 1984) are autonomous and concurrently executing objects which execute asynchronously. Actors can create new actors and can send messages to other actors. The messages sent to actors are tasks requested of the object. A slightly modified version of the actor model is the **aggregate model** (Chien and Dally, 1990). An aggregate is a collection of actors. A message sent to an aggregate is sent to all of the members of the aggregate. The method invoked can be performed by one of the actors or by some of actor objects in the aggregate working together.

   The actor model is an object-oriented version of message passing in which the actors represent processes and the methods sent between actors represent communication. There are a number of languages which support the actor model, some examples of which will be briefly discussed to show how the languages support for the development of parallel programs:

   (a) POOL (America, 1987) is a language in which each object has a body, which is a local process. The object executes in parallel with other objects. Communication between objects occurs when objects invoke methods on other objects. The object that invokes the method is blocked until the called object performs the method and returns a result.

   (b) Charm++ (Kale and Krishnan, 1993) is an extension of C++ in which actor objects must be created by the programmer. The actor classes can have no public member functions, access to the actor methods is provided through special entry points defined for each actor. New syntax in the language is provided to access these member functions. For example to invoke a method in a remote actor pointed to by actorPtr, the programmer creates a message and sends it to the actor by invoking actorPtr=> EP(msgPtr). A **Message** is a class introduced for the sending and receiving of messages between actors. Charm++ also introduces the notion of replicated objects, which can be accessed by all the objects in the system.

   (c) ProperCAD (Parkes et al., 1994) and ACT++ (Kafura and Lee, 1989) are C++ library based implementations of the actor model. Libraries provide classes that can be used in standard C++ programs to create

parallel programs which use the actor model. In ACT++ an **Actor** class is provided, and in ProperCAD both the **Actor** and **Aggregate** classes are provided. Local objects in an actor process cannot invoke methods on actor objects in the normal way, because C++ compilers do not support message passing between processes, so special abstractions are provided. For example, in ProperCAD actors have special methods, ActorMethods, which can be invoked using **Continuation** objects.

2. **Shared Object Model**: This is an object-oriented version of the shared memory model. In this model threads exist in a world of serial objects. The difference between this model and the shared memory model is that the threads can themselves be treated as objects and the data on which the threads execute are objects.

   There are a number of languages that support this model. Examples of such languages are: CC++ (Chandy and Kesselman, 1993), pC++ (Lee and Gannon, 1991), COOL (Chandra et al., 1993), pSather (Feldman et al., 1993), and PRESTO (Bershad et al., 1988) Each of the languages provides constructs for the following:

   (a) The creation of threads: In CC++ threads are created for each statement within a par{}, parfor{}, and spawn{} statement. In pC++ threads are created when objects of subclasses of **TEClass** are created or when a method is invoked on a collection. In COOL methods can be identified as being parallel; invoking such a method which creates a thread, which by default is located on the processor on which the object resides. In pSather a forAll statement is provided which is similar to the par statement in CC++. In PRESTO, like pC++, objects of type **Thread** can be constructed.

   (b) The placement of objects: In CC++ the serial objects reside inside **Processor** objects, one **Processor** object for each processor. In pC++ serial objects reside in **Collections**; the constructor for a **Collection** is passed a **Distribution** object as an argument which defines how objects in the collection are to be mapped to processors. In COOL objects are located

on the processors where they were created or the programmer can provide an additional argument to new() specifying a processor. For load balancing, the programmer can invoke a migrate function on the serial objects to move objects from one processor to another.

(c) The access to remote data: CC++ uses global pointers; the invocation of a method on an object pointed to by a global pointer is carried out on the remote processor holding the object. In pC++ the objects on a local processor can get a copy of the remote object on which they can invoke the method. In pSather each object has a global address; as in CC++ the invocation of a method on a remote object is carried out remotely.

(d) Synchronization: Because par{} and parFor{} statements in CC++ do not return control until all threads created in the statement are finished, barriers are provided. Also in CC++ Sync variables are provided to synchronize access to shared data. In COOL operations on shared objects can be mutex and event synchronization through operations on conditions variables. The language also has a waitFor{} statement; all threads created inside a waitFor{} must terminate before next statement can begin execution. In pSather locks and monitors are provided. PRESTO provides a number of synchronization classes, such as **Lock**, **Monitor** and **ConditionVariable**.

## 5.3 Existing Approaches to Parallelizing the Finite Element Method

There has been a considerable amount of research on the parallel implementation of the finite element method, with some recent work focusing on networks of workstations (Hajjar and Abel, 1988; Burman, 1990; Sharma and Baugh Jr., 1992; Kumar and Adeli, 1995; Santiago and Law, 1996). Most of the research has focused on the actual analysis algorithm, though Farhat et al. (1989) looks at methods to parallelize the I/O and in some of the work efficient methods to obtain good partitions are discussed (Farhat, 1988; Malone, 1988; Kamal and Adeli, 1990; Bernard

and Simon, 1994). The type of analysis algorithms performed includes static (Law, 1986; Farhat et al., 1987), transient using direct integration (Ortiz and Nour-Omid, 1986; Hajjar and Abel, 1988; Ou and Fulton, 1988), and modal transient (Farhat and Wilson, 1986). A number of researchers present results for the application of the analysis algorithms on different parallel machines. Those that do comment on the fact that one algorithm might perform better than another algorithm on one machine, but may perform worse on another machine (Farhat, 1990b). As discussed in section 5.2.3, this is due to the fact that the performance of the processors, memory units and communication network can vary substantially between different parallel machines.

The focus of the majority of the work has been on the methods used to parallelize the finite element analysis. The work has typically concentrated on either domain decomposition methods or on the solution of the linear system of equations. In the following subsections, a brief review of some of the work that has been presented in these areas is given. In addition, a review of the work presented for parallel object-oriented finite element analysis is also presented.

### 5.3.1 The use of Domain Decomposition Methods in Parallel Finite Element Analysis

Domain Decomposition methods are commonly used in parallel finite element analysis. This can be attributed to the fact that they are divide and conquer methods, which makes the task of assignment particularly easy for the programmer. The subdomains in the domain decomposition method are each assigned to a separate process, with the tasks required of the subdomain typically being performed in that process, though in some of the work these tasks are performed in parallel using multiple processes (Fulton and Su, 1992; Synn and Fulton, 1995).

Of the domain decomposition methods, the substructuring method has been the most commonly used, though the other domain decomposition methods have also been used, for example Carter et al. (1989) use iterative substructuring and Farhat and Roux (1994) use FETI. In the work presented for the substructuring method, the static condensation is typically performed on the assembled system of equations.

However, the frontal method is also sometimes used, in which case the system of equations is never fully formed (Zhang and Lui, 1991; Roa et al., 1994; Synn and Fulton, 1995). The solution of the interface problem is typically obtained using a direct approach, although iterative methods are also employed (Hajjar and Abel, 1988). In the direct solution of the interface problem a number of approaches have been taken:

1. El-Sayed and Hsiung (1990) solve the interface problem in each process. To do this each subdomain determines its contribution to the interface problem and then sends this information to all the other subdomains.

2. The interface problem is generally solved in a single process (Zhang and Lui, 1991; Foley and Vinnakota, 1994). All subdomain processes send their contributions to that process, which determines the interface solution and sends the solution back to the subdomain processes.

3. The interface problem can be solved in parallel using the algorithms that were described previously for the direct methods (Farhat et al., 1987; Fulton and Su, 1992; Roa et al., 1994; Baugh Jr. and Sharma, 1994; Synn and Fulton, 1995).

### 5.3.2 Existing Approaches to the Parallel Solution of Linear System Of Equations

In a lot of the research in parallel finite element analysis the focus has been on the parallel methods used to solve the linear system of equations. A brief review of the direct and iterative methods commonly used to solve these equations is now presented.

1. *Direct:* Work in this group focuses on solving the fully formed linear system of equations using a direct method. Typically some form of Gaussian elimination is used, though Berry and Plemmons (1987) use a $QR$ factorization for ill-conditioned problems. Gaussian elimination is performed using a variety of matrix storage schemes. Some popular storage schemes are the banded matrix scheme (Ou and Fulton, 1988; Goehlich et al., 1989; Lai and Chen, 1992), the

profile storage scheme (Farhat and Wilson, 1988; Farhat, 1990a; Agarwal et al., 1994) and a sparse storage scheme, (Law and Mackay, 1993; Taylor and Nour-Omid, 1994). In a lot of the work the assignment of the matrix elements to the processes is different. For example, for the profiled storage scheme, Farhat and Wilson (1988) assign the columns of the matrix in a column cyclic manner among the processors, and to improve on the performance Farhat (1990a) assign the columns in a block cyclic manner to the processes, adding zeros to the profile to obtain a blocked profile scheme.

There has also been a number of papers presented by mathematicians in which they look at the parallel solution of linear equations by direct means for band and sparse storage schemes. For banded schemes there are two approaches advocated. For banded matrices with a very small band, a divide and conquer approach is suggested, in which the equations are renumbered to allow the substructuring method to be employed (Dongarra and Johnsson, 1987; Cleary and Dongarra, 1997). For banded matrices with a very large band, no renumbering of the equations occurs and the elements of the matrix are assigned in a column cyclic or block column cyclic manner to the processes (Dongarra and Johnsson, 1987). For sparse schemes, a supernode approach is suggested in which the unknowns are grouped together to allow the computation to proceed on a blocked sparse matrix. The block sparse matrix can then be solved using a general direct sparse approach, in which the blocks are assigned to individual processes (Rothberg and Gupta, 1993; Rothberg and Schreiber, 1994; Li, 1996), or using a multifrontal approach (Duff and Reid, 1983), in which processes get work to do from a task pool containing the blocks which can be eliminated (Duff and Reid, 1986; Benner et al., 1987; Gupta and Kumar, 1994).

2. *Iterative:* Work in this group focuses on solving the fully formed linear system of equations using an iterative method. A number of iterative methods are used. Law (1986) uses the conjugate gradient algorithm. The preconditioned method is also used (Hughes et al., 1987; Nour-Omid and Park, 1987; Y. DeRoeck and Vidrascu, 1992; Baugh Jr. and Sharma, 1994; Baddourah and Nguyen, 1994; Barragy et al., 1994). A popular approach when implementing the conjugate

gradient method is the element-by-element approach (Law, 1986; Berry and Plemmons, 1987; Hughes et al., 1987). Ou and Fulton (1988) use a mixed Jacobi/Gauss-Seidel method; Gauss-Seidel within a processor and Jacobi across processors.

### 5.3.3 Existing use of Parallel Object-Oriented Programming in Finite Element Analysis

Mukunda et al. (1996) present an object-oriented framework, PFE++, for parallel finite element analysis using the substructuring approach. The analyst creates a program which is run on each of the processors of the parallel machine. The program contains two objects: a substructure object and an analysis object. Each substructure object reads its data from an input file containing the model information, the partitioning information, the mapping of the external degrees-of-freedom to equation numbers, and the type of analysis to be performed. In each process the static condensation is performed and the processes contribution to the interface problem is stored. The interface problem is then solved in parallel, each process working on that part of the matrix equation stored in that process. The framework uses an object-oriented matrix library created for PFE++ (Modak et al., 1997), which in turn employs an object-oriented communication library, PPI++ (Hsieh and Sotelino, 1997).

## 5.4 A Parallel Object-Oriented Programming Model for the Finite Element Method

In this work the actor model is used to extend the object-oriented design of the finite element method presented in the previous chapters to a parallel computing environment. The actor model was chosen instead of the shared object model for three reasons:

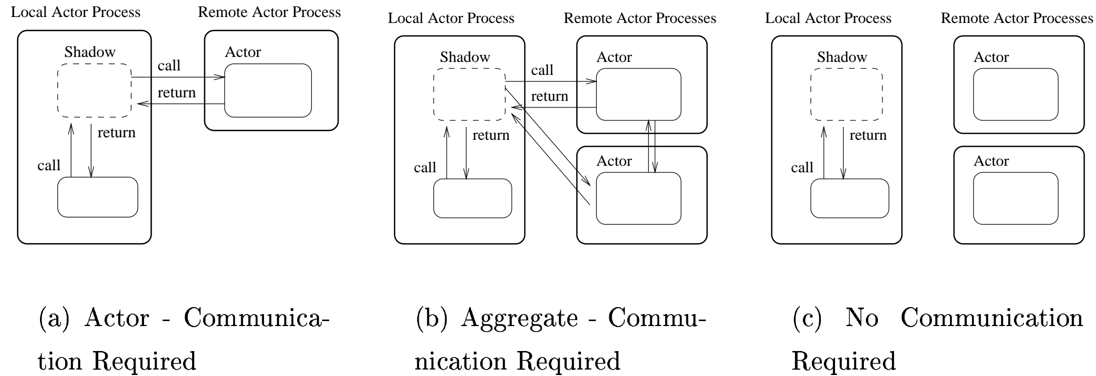
1. Synchronization in the actor model occurs naturally. The message passing between objects provides the synchronization in the parallel program. If the shared object model had been used, it would have been necessary to identify the critical sections (those objects and object methods in which only a single process could be executing at any one time) and barrier points (those methods requiring all threads created inside the method to complete before control is returned to the calling environment).

2. The creation of processes occurs naturally in the actor model. This is because a process is created whenever a new actor object is created. In the shared object model the programmer is responsible for identifying when to create the threads.

3. A number of reliable software packages supporting the message passing model existed in the public domain at the time the research started.

A slightly modified form of the Actor model is developed in this work to both minimize changes to the sequential design presented in the previous chapters and to allow for more efficient parallel finite element programming. The modification involves the introduction of **Shadow** objects. A **Shadow** object is an object in an actor's local address space. Each **Shadow** object is associated with an actor, or with multiple actors if an aggregate. The **Shadow** object represents the remote object to the objects in the local actor's space. A message intended for a remote actor is sent to the local **Shadow** object. The **Shadow** object is responsible for sending an appropriate message to the remote actor, or actors if an aggregate. The remote actor(s) receive the message and processes it. The remote actor(s) will then, if required, return the result to the local **Shadow** object, which in turn replies to the local object. The communication process is shown in figure 5.2a for the case of a single actor object and in figure 5.2b for the case of an aggregate. The important aspect of this approach is that the local object, which initiated the request, is unaware that the processing is being done remotely.

The advantage of this approach over the traditional actor approach, in which all requests are processed remotely, is that it allows the local shadow object to cache often used data and data that has not changed since the last call to that method.




(a) Actor - Communication Required

(b) Aggregate - Communication Required

(c) No Communication Required

Figure 5.2: Flow of Data using Shadow and Actor Objects

The caching of data is particularly important in a parallel environment where communication is slow, such as a NOW. This is because when a local object requests data that is being cached by the **Shadow** object, the **Shadow** object can return the data to the local object without any communication with the actor objects, as shown in figure 5.2c. The disadvantages of this approach are that it can result in an additional method call, as shown in figures 5.2a and 5.2c, and that it requires that two classes be developed, one for the **Shadow** object and another for the **Actor** object.

The **Shadow** objects perform a function similar to the stubs in remote procedure call (RPC) (Birrell and Nelson, 1984; Bershad et al., 1987). RPC is a mechanism which enables the remote performing of procedural requests made by client code in a local process. The requests are processed in a manner which makes the remoteness transparent to the local client code. The key to making the remote calls transparent are the RPC stubs. When making a remote procedural call five elements are involved: the user and user-stub, the RPC communication package (RPCruntime), and the server and server-stub. The user and user-stub exist in a single process on the local machine, as do the server and server-stub on the remote machine. When the user makes a call to the user-stub, the user-stub calls the local RPCruntime. The local RPCruntime determines which server machine is capable of servicing the request and makes a call to the RPCruntime on that machine. The remote RCPruntime makes a call on the server-stub which in turn invokes a procedure on the server. The server

processes the request and the result is sent back to the client along the reverse path of the original call. This is as shown in figure 5.3. As far as the client code is concerned the processing of the request was performed locally by code in the user-stub.

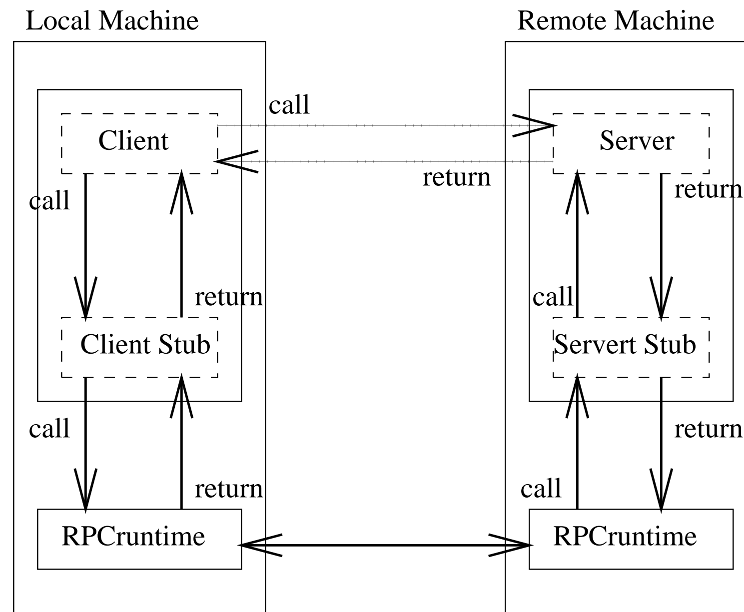



Figure 5.3: Flow of Data when making a Call in RPC

The differences between RPC stubs and **Shadow** objects are:

1. The **Shadow** objects act as both the stub and the RPCruntime, which reduces a level of indirection in the communication.

2. The **Shadow** objects can maintain history variables and a local cache of data, which can be used to determine if a remote call is actually necessary.

3. In RPC the calling process is suspended until the remote operation is performed. This need not be the case using **Shadow** objects.

## 5.5 A Framework for Parallel Object-Oriented Finite Element Analysis

The previous research into parallelizing the finite element method, summarized in section 5.3, identified the abstractions for which actor and aggregate objects are required:

1. **Actor**: It is necessary that the analyst be able to create subdomain actor objects. Actor processes store the subdomain information and process the operations required of the subdomains.

2. **Aggregate**: It is also necessary that the analyst be able to create an aggregate object for a system of equations. These are collections of actor processes which store the system of equations in parallel and in which **Solver** objects, of a type specified by the analyst, are created. The **Solver**s orchestrate the communication between the processes to perform operations on the system of equations, such as solving the linear system of equations, forming the Schur complement, or determining the eigenvalues and eigenvectors. The design of these aggregate objects permits the use of existing numerical libraries, such as ScaLAPACK (Blackford et al., 1997) and PETSc (Balay et al., 1995).

To facilitate the development of parallel object-oriented finite element programs, a new framework is presented in this section. The classes in the framework allow for the development of the actor and aggregate objects identified above. The new classes and the relationship amongst them are shown in figure 5.4. The new classes that have not been introduced in the previous chapters are shaded in figure 5.4.

The new classes for parallel finite element programming are:

- **Shadow** - A **Shadow** object represents a remote actor object in the local actor process.

- **Actor** - An **Actor** object is a local object in the remote actor process. It performs the operations requested of it by the **Shadow** object. The actor objects in an aggregation collectively perform the analysis operations by communicating between themselves.

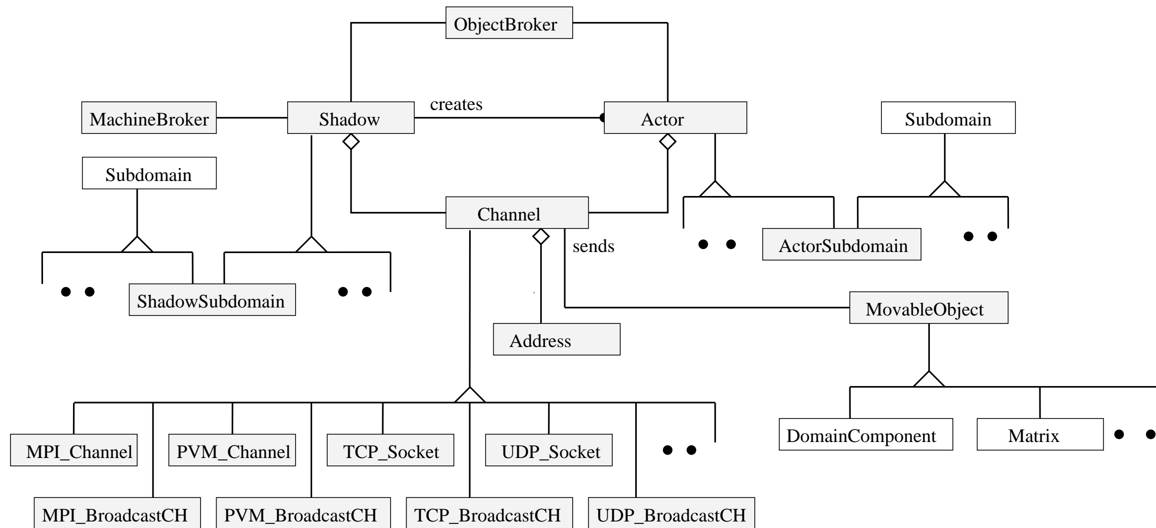



Figure 5.4: Class Diagram for Actor/Aggregate Framework for Parallel Finite Element Analysis

- **Channel** - The **Shadow** and **Actor** objects communicate with each other through **Channel** objects. A **Channel** object represents a point in a local actor process through which a local object can send and receive information.

- **Address** - An **Address** object represents the location of a **Channel** object in the machine space. **Channel** objects send information to other **Channel** objects, whose locations are given by an **Address** object. **Channel** objects also receive information from other **Channel** objects, whose locations are given by an **Address** object.

- **MovableObject** - A **MovableObject** is an object which can send its state from one actor process to another.

- **ObjectBroker** - An **ObjectBroker** is an object in a local actor process for creating new objects.

- **MachineBroker** - A **MachineBroker** is an object in a local actor process that is responsible for creating remote actor processes at the request of **Shadow** objects in the same local process.

In the following subsections the purpose of each of the new classes is outlined. Pseudo C++ code fragments are used to demonstrate the function of the classes and the interplay between them.

### 5.5.1 Shadow Class

A **Shadow** object is a local object in an actor's address space. A **Shadow** object has two main functions:

1. Create the remote actor processes, or processes if an aggregate.
2. Represent the remote actor object(s) to all other objects in the local actor process. In servicing the requests made by the local objects, the **Shadow** object can either communicate with the remote actor object(s), which performs the operation, or process the operation itself with information it caches locally on behalf of the remote object.

The **Shadow** class is a base class, as shown in the interface in figure 5.5, and it is responsible for providing the following:

1. The constructor for the **Shadow** class is responsible for starting remote actor processes. The number of processes to run and the name of the program to use are passed as arguments for the constructor along with the **ObjectBroker** object and the **MachineBroker** object. As shown in figure 5.6, it does this by repeatedly invoking startActor() on the **MachineBroker**. The constructor is also responsible for starting the handshaking required by the local **Channel** object to enable communication with the **Channel** objects associated with the remote **Actor** objects.
2. The **Shadow** class also is responsible for providing methods that allow objects of descendent classes to communicate with their associated **Actor** objects. One method is sendObject(), which is implemented as shown in figure 5.6. The communication methods are all declared as protected, as shown in figure 5.5, because the fact that the **Actor** object is on a remote process is transparent to all other objects in the local process.

```
class Shadow {
    public:
        Shadow(char *theProgram,
            Channel &theChannel,
            ObjectBroker &theObjectBroker,
            MachineBroker &theMachineBroker,
            int numActorProcesses,
                    int compDemand = 0);
        virtual ~Shadow();

    protected:
        // methods for subclasses to use for communication
        virtual int sendObject(MovableObject &theObject);
        virtual int recvObject(MovableObject &theObject);
        virtual int sendMessage(Message &theMessage);
        virtual int recvMessage(Message &theMessage);
        virtual int sendMatrix(Matrix &theMatrix);
        virtual int recvMatrix(Matrix &theMatrix);
        virtual int sendVector(Vector &theVector);
        virtual int recvVector(Vector &theVector);
        virtual int sendID(ID &theID);
        virtual int recvID(ID &theID);
};
```

Figure 5.5: Interface for the **Shadow** Class

```
Shadow::Shadow(char *program,
          Channel &theChannel,
          ObjectBroker &theObjectBroker,
          MachineBroker &theMachineBroker,
          int numActorProcesses,
          int compDemand ) {
     // start the remote actor process running
     for (int i=0; i<=numActorProcesses; i++)
          theMachineBroker.startActor(program,theChannel,compDemand);

     // now call setUpShadow on the channel
     theChannel.setUpShadow();
     theRemoteActorsAddress = theChannel.getLastSendersAddress();
}

Shadow::sendObject(MovableObject &theObject) {
     theChannel->sendObj(theObject,theObjectBroker,theRemoteActorsAddress);
}
```

Figure 5.6: Pseudo-Code for Selected Methods for the **Shadow** Class

The **Shadow** class is an abstract base class. The subclasses of **Shadow** are used by the analyst to create parallel programs. One such subclass, as shown in figure 5.4, is **ShadowSubdomain**. The **ShadowSubdomain** class, whose interface is as shown in figure 5.7, uses multiple inheritance to inherit from both the **Shadow** and **Subdomain** classes.

```
class ShadowSubdomain: public Shadow, public Subdomain {
     public:
          ShadowSubdomain(int tag,
                         Channel &theChannel,
                         ObjectBroker &theObjectBroker,
                         MachineBroker &theMachineBroker);
          virtual ~ShadowSubdomain();

          // most of the subdomain methods must be re-implemented
};
```

Figure 5.7: Interface for the **ShadowSubdomain** Class

```
ShadowSubdomain::addElement(Element *theElement) {
    // check that it is not already added
    if (theElements.getLocation(theElement->getTag() ≥ 0)
        return -1;
    // send a message to the actor telling it to get
    // ready to add an element of the appropriate class.
    theData(0) = addElement; // addElement an int specified in common header
    theData(1) = theElement->getClassTag();
    this->sendID(theData);
    // get the Element object to send its state
    this->sendObject(*theElement);
    delete theElement;
};

ShadowSubdomain::formTang(void) {
    theData(0) = formTang;
    this->sendID(theData);
    haveComputedTang = false;
};

ShadowSubdomain::getTang(void) {
    // check if can do a quick return
    if (haveComputedTang == true)
        return *theMatrix;

    // send a message telling actor object to return the tangent
    msgData(0) = getTang;
    this->sendID(theData);

    // receive the tangent matrix from the actor object
    this->recvMatrix(*theMatrix);
    haveComputedTang = true;
    return *theMatrix;
}
```

Figure 5.8: Pseudo-Code for Selected Methods for the **ShadowSubdomain** Class

The **ShadowSubdomain** class re-implements the methods provided by the **Subdomain** class. This is because many of the operations must now be processed remotely by an **ActorSubdomain** object. For example in the addElement(), formTang() and getTang() methods, which are shown in figure 5.8, the **ShadowSubdomain** object will send messages to its remote **ActorSubdomain** object to have it process the request, if it cannot process the request itself, using data it has cached. In the implementation of these methods the **ShadowSubdomain** class uses the communication methods provided by its parent **Shadow** class.

### 5.5.2 Actor Class

An **Actor** object is the object in the remote process which processes the requests made by the **Shadow** objects. The **Actor** object is responsible for receiving incoming messages from its associated **Channel** object and acting upon them. The **Actor** object continues processing incoming messages until it is told to stop, at which point the actor process terminates. The processing of the incoming messages is performed by the method run(), which is invoked on the **Actor** object when it is ready to process the incoming messages. This is demonstrated in figure 5.9, in which the pseudo-code for the remote actor program invoked by the creation of each **ShadowSubdomain** object is presented.

The **Actor** class is an abstract base class, as shown by the interface in figure 5.10. The **Actor** class, like the **Shadow** class, is responsible for providing its descendents with methods for sending and receiving data. The class constructor is responsible for starting the handshaking that the local **Channel** object needs to perform to enable it to communicate with the **Channel** object associated with the **Shadow** object.

The **ActorSubdomain** class is an example of a subclass of **Actor**, as shown in the class diagram presented in figure 5.4. Each **ActorSubdomain** object is associated with an **ShadowSubdomain** object. The **ActorSubdomain** object performs the operations requested of it by the **ShadowSubdomain** object. The **ActorSubdomain** class, as shown in the interface presented in figure 5.11, inherits from both the **Actor** and **Subdomain** classes.

The **ActorSubdomain** class provides an implementation of the run() method

```
int main(int argv, char **argc) {
    // create the channel object using info passed in the args
    int channelType = atoi(argc[1]);
    if (channelType == 1) {
        char *machine = argc[2];
        int port = atoi(argc[3]);
        theChannel = new TCP_Socket(port,machine);
    } else if (channelType == 2) {
        // code to deal with the init of other Channel types
    }

    // create an object broker
    ObjectBroker theObjectBroker;

    // create the actor object and start it running
    ActorSubdomain theActor(*theChannel,theObjectBroker);
    theActor.run();

    // exit normally
    exit(0);
}
```

Figure 5.9: Pseudo-Code for an Actor Program

```
class Actor {
  public:
    Actor (Channel &theChannel,
                ObjectBroker &theObjectBroker);
    virtual ~Actor();

    virtual int run(void);

  protected:
    virtual int sendObject(MovableObject &theObject, ChannelAddress *theAddress =0);
    virtual int recvObject(MovableObject &theObject, ChannelAddress *theAddress =0);
    virtual int sendMessage(Message &theMessage, ChannelAddress *theAddress =0);
    virtual int recvMessage(Message &theMessage, ChannelAddress *theAddress =0);
    virtual int sendMatrix(Matrix &theMatrix, ChannelAddress *theAddress =0);
    virtual int recvMatrix(Matrix &theMatrix, ChannelAddress *theAddress =0);
    virtual int sendVector(Vector &theVector, ChannelAddress *theAddress =0);
    virtual int recvVector(Vector &theVector, ChannelAddress *theAddress =0);
    virtual int sendID(ID &theID, ChannelAddress *theAddress =0);
    virtual int recvID(ID &theID, ChannelAddress *theAddress =0);
};
```

Figure 5.10: Interface for the **Actor** Class

```
class ActorSubdomain: public Subdomain, public Actor {
    public:
        ActorSubdomain(Channel &theChannel,
                    ObjectBroker &theObjectBroker);
        virtual ~ActorSubdomain();

        virtual int run(void);
        virtual const Vector &getLastExternalSysResponse(void);
}
```

Figure 5.11: Interface for the **ActorSubdomain** Class

that was declared as pure virtual in the **Actor** interface, figure 5.10. The method, portions of which are as shown in figure 5.12, uses the previously defined methods in both its ancestor classes. The class also re-implements the getLastExternalSysResponse() method. This is because this information, which is passed by the **ShadowSubdomain** objects when an update() method is invoked, is stored locally by **ActorSubdomain** object. This was done in an effort to reduce the number of messages sent between processes during the analysis.

```
ActorSubdomain::run(void) {
    bool exitYet = false;
    while (exitYet == false) {
        this->recvID(theData);
        int action = theData(0);
        switch (action) {
            case addElement: // get an element and add it to the Subdomain
                    theType = msgData(1);
                    theEle = theObjectBroker->getPtrNewElement(theType);
                    this->recvObject(*theEle);
                    result = this->addElement(theEle);
                    break;
            case getTang: // send the tangent to the ShadowSubdomain
                    theMatrix = &(this->getTang());
                    this->sendMatrix(*theMatrix);
                    break;
            case formTang: // compute the tangent
                    this->computeTang();
                    break;
            // the pseudo-code for the other cases not shown
}
```

Figure 5.12: Pseudo-Code for the **ActorSubdomain** Classes run Method

### 5.5.3 Channel Class

A **Channel** object is a point of communication in a local address space through which data enters and leaves the local process. The **Channel** object establishes the lines of communication between **Shadow** and **Actor** objects, and sends and receives the data. The **Channel** class, whose interface is as shown in figure 5.13, is an abstract base class. The methods in the interface, all of which must be implemented by the

subclasses, provide for the following:

1. Two methods are provided, which are invoked in the construction of **Shadow** and **Actor** objects, respectively, which allow the **Actor** and **Shadow** objects to determine the address of each other.

2. Methods are also provided to allow the transmission of data between actor processes. The data that can be sent is in the form of objects, such as **Message** objects, **Matrix** objects, **Vector** objects and **ID** objects. This communication of objects in this approach is different from the that used in PPI++, in which raw data is passed. This current approach is more object-oriented approach and, as will be discussed in section 5.5.5, it controls the access of the **Channel** objects to the data.

The subclasses of **Channel** provide the implementations of these methods. Examples of subclasses, as shown in figure 5.4, are **PVM_Channel**, **MPI_Channel**, **TCP_Socket**, and **PVM_BroadcastCH**. The broadcast channel objects are used by **Shadow** objects to communicate with the actor objects in an aggregation. When a **Shadow** object sends data to one of these broadcast objects, the data is sent to all the processes given by the **Address** specified. By instantiating an object of a particular type, the analyst chooses the protocol for communication between the **Shadow** and **Actor** objects.

### 5.5.4 Address Class

Each **Channel** object has a unique address, which identifies the **Channel** among the processes running on the parallel machine. The **Address** class, whose interface is as shown in figure 5.14, is a base class defining a single pure virtual method getClassTag(). A subclass of **Address** will exist for each subclass of **Channel** because each communication protocol has its own means of identifying the addresses. The **Channel** objects use getClassTag() to verify the type of **Address** object passed as an argument to the **Channel** object's send and receive routines.

```
class Channel {
    public:
        Channel ();
        virtual ~Channel();

        virtual int setUpShadow(void) =0;
        virtual int setUpForActor(void) =0;
        virtual int setNextAddress(const ChannelAddress &theAddress) =0;
        virtual ChannelAddress *getLastSendersAddress(void) =0;
        virtual char *addToProgram(void) =0;

        // methods to send/receive data to/from a channel
        virtual int sendObj(MovableObject &theObject,
                            const ObjectBroker &theObjectBroker
                            const ChannelAddress *theAdress =0) =0;
        virtual int recvObj(MovableObject &theObject,
                            const ObjectBroker &theObjectBroker
                            const ChannelAddress *theAdress =0) =0;
        virtual int sendMsg(Message &theMessage,
                            const ChannelAddress *theAdress =0) = 0;
        virtual int recvMsg(Message &theMessage,
                            const ChannelAddress *theAdress =0) = 0;
        virtual int sendMatrix(Matrix &theMatrix,
                            const ChannelAddress *theAdress =0) = 0;
        virtual int recvMatrix(Matrix &theMatrix,
                            const ChannelAddress *theAdress =0) = 0;
        virtual int sendVector(Vector &theVector,
                            const ChannelAddress *theAdress =0) = 0;
        virtual int recvVector(Vector &theVector,
                            const ChannelAddress *theAdress =0) = 0;
        virtual int sendID(ID &theID,
                            const ChannelAddress *theAdress =0) = 0;
        virtual int recvID(ID &theID,
                            const ChannelAddress *theAdress =0) = 0;
}
```

Figure 5.13: Interface for the **Channel** Class

```
class Address {
      public:
            Address(int classTag);
            virtual ~Address();

            int getClassTag(void) const;
};
```

Figure 5.14: Interface for the **Address** Class

### 5.5.5 Message Class

Processes communicate by passing messages between **Channel** objects. As discussed in section 5.2.2, the sending process specifies the location and size of the data to be transmitted, and the receiving process specifies the size of data to be received and where the data is to be placed. There are two approaches that could be used for sending/receiving data to/from the **Channel** objects:

1. The arguments to send() and recv() specify the data size and its location in the processes virtual address space, as is done in PPI++ (Hsieh and Sotelino, 1997). For example the pseudo-code to send an array of double to a **Channel** object would be theChannel.send(dataPtr,sizeOfData). The problem with this approach is that it leads to uncontrolled access to private data by the **Channel** objects. For example, a poorly written **Channel** subclass can to overflow the data buffer associated with the incoming messages, which will lead to program errors and possibly segmentation faults.

2. Provide a single class to control the access by **Channel** objects to the data. This will prevent **Channel** objects from both modifying the data that is to be sent from the sending process, and from overflowing the data buffer in the receiving process, which can lead to segmentation faults.

It is the second approach that is used in this framework. A **Message** object is an object controlling the access to the data that can be sent/received by a **Channel** object in one actor process to/from a **Channel** object in another actor process. The **Message** class, whose interface is as shown in figure 5.15, provides: constructors

for messages from a range of data types; a method to obtain the size of the data; a method to place data into the message in a controlled way; and a method to return a const pointer to the data.

```
class Message {
    public:
        // constructors
        Message(double*dataPtr, int sizeOfData);
        Message(int    *dataPtr, int sizeOfData);
        Message(char   *dataPtr, int sizeOfData);
        virtual ~Message();

        // methods to allow Channel objects to obtain
        // controlled access to the data passed in the constructors.
        virtual int putData(char *theData, int startLoc, int endLoc);
        virtual const char *getData(void);
        virtual int getSize(void);
};
```

Figure 5.15: Interface for the **Message** Class

### 5.5.6 MovableObject Class

A **MovableObject** object is an object which can send its state from one actor process to an object in another actor process. The transmission of the objects state is initialized when an object in the local sending actor process invokes a sendSelf() on the local object and it is completed when an object in the remote process invokes a recvSelf() on the remote object. The operations sendSelf() and recvSelf() are pairwise operations.

The **MovableObject** class, whose interface is as shown in figure 5.16, is an abstract base class. The methods sendSelf() and recvSelf() are pure virtual. The subclasses of **MovableObject**, such as **Element**, **Node**, **Matrix** and **Vector**, must implement these methods. The **MovableObject** class is also responsible for providing a method getClassTag() which will allow the objects type to be uniquely determined. As will be discussed in section 5.5.8, this is so that objects of the sending objects type can be constructed in the remote actor process to receive the state information

being sent.

```
class MovableObject {
    public:
        MovableObject(int classTag);
        virtual ~MovableObject();

        int getClassTag(void) const;
        virtual int sendSelf(Channel &theChannel,
                             ObjectBroker &theBroker) =0;
        virtual int recvSelf(Channel &theChannel,
                             ObjectBroker &theBroker) =0;
};
```

Figure 5.16: Interface for the **MovableObject** Class

### 5.5.7 MachineBroker Class

A **MachineBroker** object is responsible for starting actor processes running on the parallel machine. The **MachineBroker** class is an abstract base class, as shown by the interface in figure 5.17. The class has one method, startActor(). This method is invoked in the construction of a **Shadow** object, and it starts a remote actor process. A descendent class must be provided for each parallel machine that the analyst wishes to use. Each subclass must provide its own implementation of the startActor() method.

The decision of which processor to start the actor process on is left to the discretion of the **MachineBroker** object. In considering which processor to use the **MachineBroker** object may look at a number of factors:

1. The workload currently on each processor in the parallel machine, information which can be obtained from the system.

2. The number of users currently on each processor, information which can again be obtained from the system.

```
class MachineBroker {
    public:
        MachineBroker();
        virtual ~MachineBroker();

        // method to start the actor program
        // running on next a processor
        virtual int startActor(char *actorProgram,
                               Channel &theChannel,
                               int compDemand =0) =0;
};
```

Figure 5.17: Interface for the **MachineBroker** Class

3. The relative performance of the available processors, if in a heterogeneous environment. The information about the performance of the processors being built into the particular **MachineBroker** subclass provided for the parallel machine.

4. The expected computational demand of the actor process, which is given by the compDemand value passed in the startActor() method.

### 5.5.8 ObjectBroker Class

In a parallel environment **Shadow** and **Actor** objects construct objects to facilitate transmission of other objects from one actor process to another. The **Shadow** or **Actor** object about to receive the transmitted object requires a newly constructed object of the correct type before it can invoke **recvSelf()**. For example, when an **ActorSubdomain** object is receiving an element that is to be added to the subdomain, it must invoke recvSelf() on a newly constructed element object of the correct type, as shown in figure 5.12. Two approaches could be used to construct recipient objects:

1. Each **Actor** and **Shadow** object knows about all the possible types of objects that it may be asked to receive. For example, an **ActorSubdomain** object knows about all the subclasses of **Element** that may be added to the subdomain. The problem with this approach is that the **Actor** and **Subdomain**

object's classes and their descendents would have to be rewritten for each new **MovableObject** subclass introduced.

2. Provide a single class that knows all the types of objects that may be required to move from one actor process to another. In this approach the introduction of new subclasses only requires that this one class be modified or a subclass of this class be introduced.

Using the second approach, an **ObjectBroker** object instantiates and returns pointers to these objects. The **ObjectBroker** class, whose interface is given in figure 5.18, is responsible for providing methods to perform the following operations:

1. Methods are provided to construct and return pointers to all the main class abstractions that are used in the modeling (**Element**, **Node**, **MP_Constraint**, **SP_Constraint**, **LoadCase**, **NodalLoad**, **ElementalLoad**, **Matrix**, **Vector**, and **ID** objects).

2. Methods are also provided to construct and return pointers to all the main class abstractions used in domain decomposition analysis, (**ConstraintHandler**, **AnalysisModel**, **DomainDecompAlgo**, **IncrementalIntegrator**, **DomianSolver**, **LinearSOE**), **DOF_Numberer** and **DomainDecompositionAnalysis**. These methods are required so that the analyst only has to specify the type of domain decomposition method to be used in the main program, and not in the programs for the **ActorSubdomain**.

The exact type of object that is to be created and returned by the **ObjectBroker** is identified by the unique classTag identifier supplied as an argument to the methods. The provision of the **MovableObject** class allows the partitioning of the **Domain** to occur in the same running process as the analysis, which cannot be done in PFE++ (Mukunda et al., 1996). In addition it allows the model to change as the analysis proceeds, which may be required in hp-refinement and multigrid, and it allows dynamic load balancing between subdomains, as shall be demonstrated in chapter 7.

```
class ObjectBroker {
    public:
        ObjectBroker();
        virtual ~ObjectBroker();

        // methods to create new uninitialized model component
        // objects and return pointers to them
        virtual Element *getPtrNewElement(int classTag);
        virtual Node *getPtrNewNode(int classTag);
        virtual MP_Constraint *getPtrNewMP(int classTag);
        virtual SP_Constraint *getPtrNewSP(int classTag);
        virtual LoadCase *getPtrNewLC(int classTag, int tag);
        virtual NodalLoad *getPtrNewNodalLoad(int classTag);
        virtual ElementalLoad *getPtrNewElementalLoad(int classTag);
        virtual Matrix *getPtrNewMatrix(int classTag, int noRows, int noCols);
        virtual Vector *getPtrNewVector(int classTag, int size);
        virtual ID *getPtrNewID(int classTag, int size);

        // methods to get new unitialised objects
        // for the analysis algorithms and returns a pointer to them
        virtual ConstraintHandler *getPtrNewConstraintHandler(int classTag);
        virtual DOF_Numberer *getPtrNewNumberer(int classTag);
        virtual AnalysisModel *getPtrNewAnalysisModel(int classTag);
        virtual EquiSolnAlgo *getPtrNewEquiSolnAlgo(int classTag);
        virtual DomainDecompAlgo *getPtrNewDomainDecompAlgo(int classTag);
        virtual StaticIntegrator *getPtrNewStaticIntegrator(int classTag);
        virtual TransientIntegrator *getPtrNewTransientIntegrator(int classTag);
        virtual IncrementalIntegrator *getPtrNewIncrementalIntegrator(int classTag);

        virtual int getPtrNewLinearSOE(int classTagSOE, int classTagSolver,
              LinearSOE *, LinearSOESolver *);
        virtual int getPtrNewDDLinearSOE(int classTagSOE, int classTagDDSolver,
              LinearSOE *, DomainSolver *);
        virtual DomainDecompositionAnalysis *
              getPtrNewDomainDecompAnalysis(int classTag, Subdomain &theDomain);
        virtual Subdomain *getSubdomainPtr(int classTag);
};
```

Figure 5.18: Interface for the **ObjectBroker** Class

## 5.6 Modification of Classes for Parallelism

In this section a review is made of what changes to the design presented in the previous chapters for parallel finite element analysis. The changes occur in both the class interfaces and the class methods.

### 5.6.1 Modification to Class Interfaces

The descendents of **MovableObject** have to be modified to allow for the object to send and receive itself. Each of the classes introduced in the previous chapters must be extended to include the methods sendSelf() and recvSelf(). For example, the revised **NodalLoad** interface is shown in figure 5.19.

```
virtual class NodalLoad : public Load {
    public:
        NodalLoad(int nodeTag);
        NodalLoad(int nodeTag, const Vector &load);
        ~NodalLoad();
        virtual int getNodeTag(void) const;
        virtual void applyLoad(double timestep = 0.0);
        virtual int sendSelf(Channel &theChannel,ObjectBroker &theBroker);
        virtual int recvSelf(Channel &theChannel,ObjectBroker &theBroker);
};
```

Figure 5.19: Revised Interface for the **NodalLoad** Class

### 5.6.2 Modification to Class Methods

As discussed in section 5.2.3, it is important not just to identify those tasks that can be executed concurrently but to orchestrate the tasks that can be performed concurrently. To orchestrate effectively, some of the existing code has to be modified. For example, consider the pseudo-code used by an **IncrementalIntegrator** when forming the tangent matrix, which is repeated in figure 5.20.

A time line for the pseudo-code in figure 5.20 for the execution of an analysis with two **ActorSubdomain**s is shown in figure 5.22. Even though the tangent for each subdomain is being processed by separate actor processes, there is no overlap

```
IncrementalIntegrator::formTangent {
      FE_EleIter theEles = theAnalysisModel->getFEs()
      theLinearSOE->zeroA()
      while ((feElePtr = theEles()) ≠ 0) {
            feElePtr->formTangent(theIntegrator)
            theLinearSOE->addA(feElePtr->getTangent(), feElePtr->getID())
      }
}
```

Figure 5.20: Pseudo-Code for the **IntcrementalIntegrators** formTangent Method

of computation. As discussed in section 5.2.3, this program would perform no better than a sequential program for this phase of the computation, although both **ActorSubdomain**s could be forming their tangents concurrently. To orchestrate the tasks concurrently, the operations in the formTangent() method must be re-sequenced. The sequence of pseudo-code for the revised operation is presented in figure 5.21. In the revised method each FE_Element is asked to compute its tangent before each FE_Element is asked for its tangent. The resulting improvement in performance can be seen in figure 5.23.

```
IncrementalIntegrator::formTangent {
      FE_EleIter theEles = theAnalysisModel->getFEs()
      theLinearSOE->zeroA()
      while ((feElePtr = theEles()) ≠ 0)
            feElePtr->formTangent(theIntegrator)
      while ((feElePtr = theEles()) ≠ 0)
            theLinearSOE->addA(feElePtr->getTangent(), feElePtr->getID())
}
```

Figure 5.21: Revised Pseudo-Code for the **IncrementalIntegrators** formTangent Method

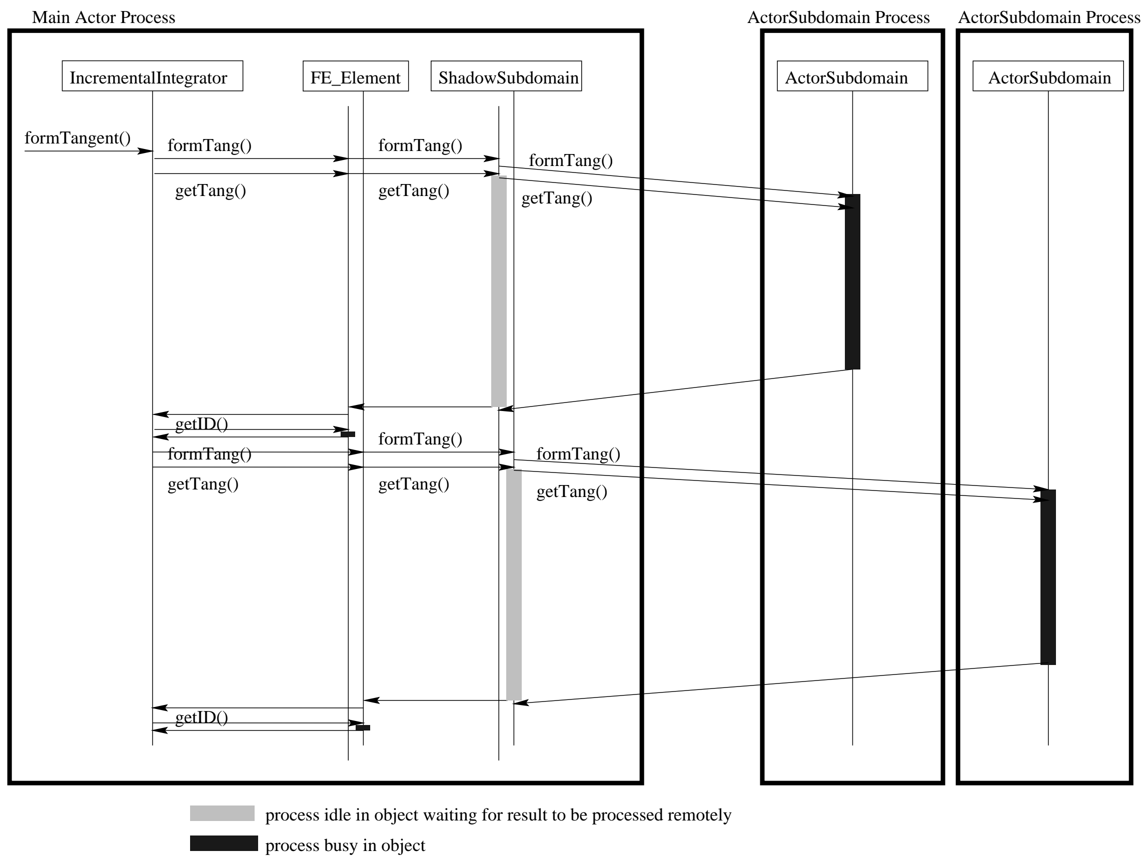



Figure 5.22: Time Line for Original formTangent Method

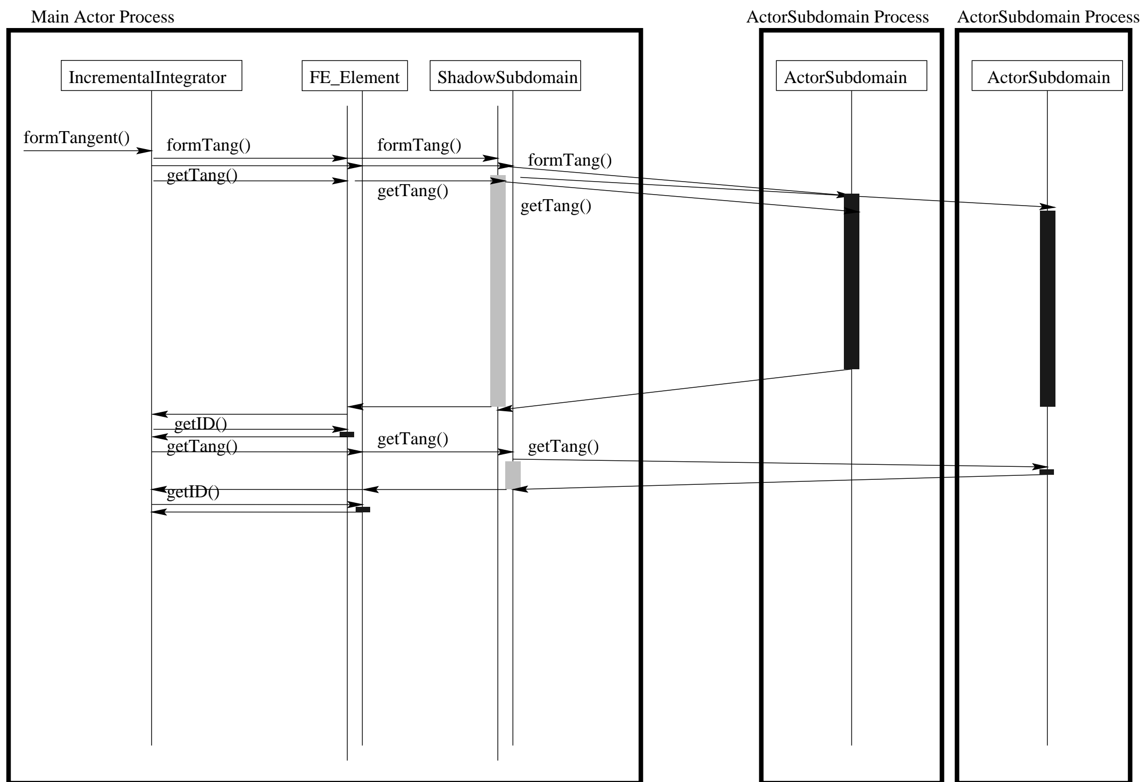



Figure 5.23: Time Line for Revised formTangent Method

## 5.7 Example Parallel Programs

To demonstrate the flexibility and the transparency of this approach, the base sequential pseudo C++ program presented in section 4.6 is repeated here. Modifications to this program are then made to produce new programs which perform parallel analysis. The base program performs a transient analysis of a space shuttle model using the Newmark integration strategy, a Newton-Raphson iteration at each time step, and the substructuring method on a uniprocessor. Four subdomains are created, each of which uses a reverse Cuthill-McKee numbering scheme to order the degrees-of-freedom and a profile storage scheme to store the subdomain equations. The interface problem, which uses a reverse Cuthill-McKee numbering scheme and a banded storage scheme to store the equations, is solved by a direct method. The pseudo-code for the base program is as follows:

```
001    numSubdomains = 4;
002    /* create the partitioned domain and model builder */
003    Metis theGraphPartitioner;
004    DomainPartitioner thePartitioner(theGraphPartitioner);
005    PartitionedDomain theDomain(thePartitioner);
006
007    /* create the subdomain and add to the domain
008
009
010    for (int i=1; i<=numSubdomains; i++) {
011
012        Subdomain theSubdomain(i)
013        Transformation theConstraintHandler;
014        RCM theDOFNumberer;
015        AnalysisModel theModel;
016        ProfileSPDSOE_Substr_Solver theSolver;
017        ProfileSPDSOE theLinearSOE(theSolver);
018        Newmark theIntegrator(1/4, 1/2);
019        DomainDecompAlgo theSolnAlgo;
020        DomainDecompAnalysis theAnalysis(theSubdomain, theConstraintHandler,
021            theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinearSOE);
022        theDomain.addSubdomain(theSubdomain);
023    }
024
025    /* create a model builder and build the model */
026    SpaceShuttle theModelBuilder(theDomain);
027    theModelBuilder.buildModel();
028
029    /* partition the domain into the subdomains */
030    theDomain.partition(numSubdomains);
```

```
031
032      /* create the analysis */
033      Transformation theConstraintHandler;
034      RCM theDOFNumberer;
035      AnalysisModel theModel;
036      DirectBandSPDSOE theSolver;
037      BandSPDSOE theLinearSOE(theSolver);
038      Newmark theIntegrator(1/4, 1/2);
039      NewtonRaphson theSolnAlgo;
040      DirectIntegrationAnalysis theAnalysis(theDomain,theConstraintHandler,
041           theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinearSOE);
042
043      /* perform the analysis */
044      theDomain.setLoadCase(1);
045      theAnalysis.analyze;
046
```

To change the code so that the substructuring is done in parallel, the analyst would replace lines 008, 009, 011 and 012 with the following:

```
008      ObjectBroker theObjectBroker;
009      AlphaMachineBroker theMachineBroker;
011      PVM_Channel theChannel;
012      ShadowSubdomain theSubdomain(theChannel, theObjectBroker);
```

If the analyst finds that the resulting program is still too slow the analyst could instead partition the domain into 32 subdomains and try a parallel system of equations and solver. The analyst would replace lines 001, 036 and 037 in the revised program with the following:

```
001      numSubdomains = 32;
036      ShadowDirectBandSPDSOE theSolver;
037      ShadowBandSPDSOE theLinearSOE(theSolver);
```

# Chapter 6

# Example Structural Analysis and Performance Evaluation

In this chapter the performance of the object-oriented design presented in the previous chapters is discussed. The design is evaluated by performing, both sequentially and in parallel, the analysis of a number of simple structural models using a test implementation of the design. The performance of the test implementation is then compared with a procedural program.

## 6.1 Introduction

In order to be practical a software design must allow for efficiency in terms of both time and memory requirements. Object-oriented programs have been shown from a programmers point of view to have a shorter development time than conventional procedural programs (Forde et al., 1990), and to require less time to both maintain and extend (Dubois-Pelerin and Zimmermann, 1993; Zeglinski and Han, 1994; Rucki, 1996). The resulting executables have also been shown to be smaller for object-oriented programs than comparable procedural programs (Forde et al., 1990).

In order to be practical, however, it is more important that the software design be efficient from a users point of view. A software design must allow finite element packages to be developed which are as fast as conventional packages and which are able to accommodate problems as large as these conventional packages.

In this chapter the efficiency of the design presented in the previous chapters is evaluated from a users point of view. To do this a test implementation of the design is developed using the object-oriented programming language C++, the test implementation containing over 50,000 lines of code. The efficiency of this program is then evaluated in two steps:

1. A number of analyses are performed on some structural models using a uniprocessor machine. The performance of the object-oriented program is then compared with the performance of a procedural program.

2. The analysis are repeated in parallel using two different parallel machines, two different NOWs. The performance of the object-oriented program is then compared with the performance of the object-oriented program on the uniprocessor machines.

The results will show that while the performance is comparable in terms of CPU time, the object-oriented program suffers in terms of memory requirements. This memory problem is alleviated on parallel machines, as the results will show.

## 6.2 Example Structural Models

To evaluate the performance of the test implementation a number of linear static analysis are performed on typical structural models. The models are of two basic types: plane frame and three dimensional frame. These two types are chosen for a number of reasons: they allow example models to be generated easily, they allow models to be generated which vary in respect to the percentage of operations performed in the various stages of the analysis, and the topologies of the models are representative of two- and three- structural models.

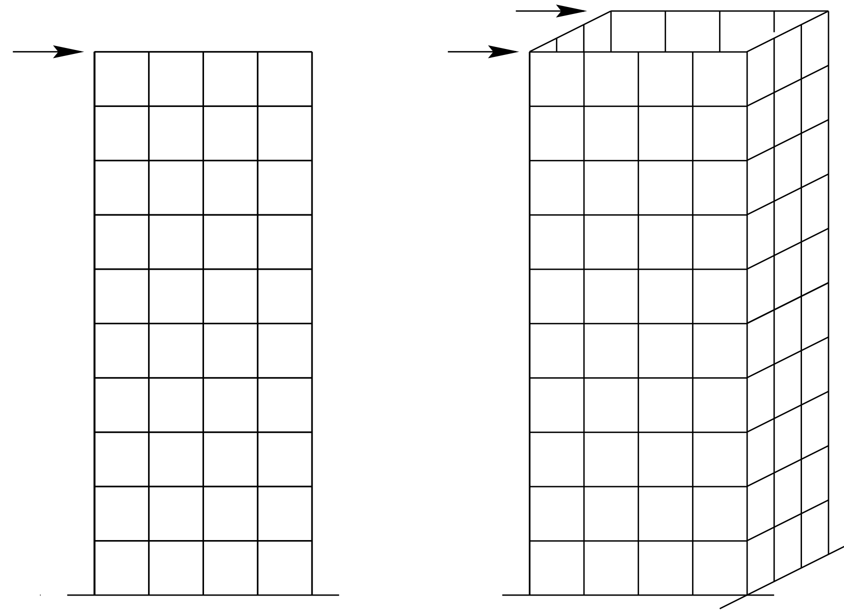


Figure 6.1: Two and Three Dimensional Test Models

Two **ModelBuilder** classes are developed, **Quick2dFrame** and **Quick3dFrame** respectively. Objects of these classes generate models which use linear two- and three-dimensional beam elements to connect the nodes. Loads are applied at the roof levels and the base of each model is fully restrained, as shown in figure 6.1. The two **ModelBuilder** classes prompt the user for the number of bays and number of stories of the model to be generated. The resulting models which, while not corresponding to any actual structures, are useful for measuring the performance of the software.

A number of example models, as described in tables 6.1 and 6.2, are analyzed.

The examples chosen vary both in respect to the memory requirements and with regard to the percentage of the time required by the program to solve the system of equations relative to the time to form the equations. The examples 3dF3, 3dF4, 3dF5, and 3dF6, while they have similar numbers of degrees-of-freedom as examples 2dF3, 2dF4, 2dF5, and 2dF6, require more memory, as can be seen from the fact the size of the profile is greater. These examples also require a greater percentage of the CPU time to solve these equations. This can be seen from the fact that, while the average column height of the upper triangular portion of the symmetric profile matrix required to store the system of equations (Avg. Height) is increased by 56% for these 3d examples, the number of elements and nodes in the examples are less than half those in the plane frame examples.

| Example | # Elements | # Nodes | # DOF | Profile Size | Avg. Height |
|---|---|---|---|---|---|
| 2dF1 | 3050 | 1581 | 4650 | 43371 | 93 |
| 2dF2 | 4050 | 2091 | 6150 | 753981 | 122 |
| 2dF3 | 5050 | 2601 | 7650 | 1162791 | 151 |
| 2dF4 | 6060 | 3111 | 9180 | 1399941 | 152 |
| 2dF5 | 7070 | 3622 | 10710 | 1637091 | 152 |
| 2dF6 | 8080 | 4131 | 12240 | 1874241 | 153 |

Table 6.1: Two Dimensional Frame Examples

| Example | # Elements | # Nodes | # DOF | Profile Size | Avg. Height |
|---|---|---|---|---|---|
| 3dF3 | 2560 | 1320 | 7680 | 1815252 | 239 |
| 3dF4 | 3120 | 1600 | 9360 | 2224332 | 238 |
| 3dF5 | 3600 | 1840 | 10800 | 2574972 | 237 |
| 3dF6 | 4080 | 2080 | 12240 | 2925612 | 236 |

Table 6.2: Three Dimensional Frame Examples

# 6.3 Evaluation of the Object-Oriented Design on Uniprocessor Machines

## 6.3.1 Introduction

In this section the performance of the object-oriented program is compared with a procedural program. The performance of each program is evaluated by measuring the real time, the CPU time and the number of page faults required to perform a linear static analysis of the models presented in section 6.2. To demonstrate the effects of memory size and processor speed on the performance of the programs, several different machines, as indicated in table 6.3, are used to perform the analysis.

| | HOLDEN | ALPHA | DEC |
|---|---|---|---|
| Workstation | ALPHAstation 500 | ALPHAstation 255 | DECstation 3100 |
| Processor | ALPHA | ALPHA | MIPS |
| Clock Speed | 266 MHz | 233 MHz | 16.67 MHz |
| Cache on Chip | 8KB-I, 8KB-D 96KB - L2 | 16KB-I | 64KB-I 64KB-D |
| Cache on Board | 2MB -L3 | 1MB - L3 | None |
| RAM | 64MB | 32MB | 12MB |
| OS | Digital Unix V3.2D | Digital Unix V3.2D | Ultrix V4.4 |
| Page Size | 8192 | 8192 | 4096 |
| Compiler | DEC C++ | DEC C++ | GNU C++ |
| # Machines | 1 | 8 | 8 |
| Connection | NA | 10Mb Ethernet | 10 Mb Ethernet |
| Linpack MFLOPS 100x100 | 123.7 | 46.5 | 1.6 |
| Cost of Page Fault | 7ms | 10ms | 40ms |

Table 6.3: Hardware Environments for Performance Measurements

For the static analysis of the models, the system of equations are stored using a profile storage scheme and solved directly. No equations are assigned to the degrees-of-freedom that are constrained and no renumbering of the equations is performed to reduce the profile. The driver of the C++ program using the new framework is:

```
001  main() {
002          Domain theDomain();
003          Quick2dFrame theModelBuilder(theDomain)
004          theModelBuilder.buildModel();
005
006          /* start measuring the system resources */
007          Timer theTimer();
008          theTimer.start();
009
010          /* create the analysis */
011          Transformation theConstraintHandler;
012          PlainNumberer theGraphNumberer;
013          DOF_Numberer theDOFNumberer(theGraphNumberer);
014          AnalysisModel theModel;
015          DirectProfileSPDSOE theSolver;
016          ProfileSPDSOE theLinSOE(theSolver);
017          StaticIntegrator theIntegrator;
018          Linear theSolnAlgo;
019          StaticAnalysis theAnalysis(theDomain, theConstraintHandler, theDOFNumberer,
020                  theModel, theSolnAlgo, theIntegrator, theLinSOE);
021
022          /* perform the analysis */
023          theDomain.setLoadCase(1);
024          theAnalysis.analyze;
025
026          /* print system resources used */
027          theTimer.pause();
028          theTimer.print(cout);
029  }
```

### 6.3.2 Procedural Program

A reference implementation is developed for a uniprocessor machine. This reference implementation is written in the procedural language C, using the typical procedural design (Zienkiewicz and Taylor, 1989). The procedural program is written to perform a linear static analysis of the plane frame examples with many features hard coded. For example, the profile of the system of equations can be determined from the number of bays and stories supplied as input. Also, as there are no element loads and the analysis is static no element residuals need be calculated. The hard coding was done to ensure the fastest possible reference procedural implementation. As a consequence, the program is also limited to analyzing the simple frame examples presented in section 6.2. Other two- and three- dimensional frame examples cannot

be analyzed by this program.

### 6.3.3 Results

Tables 6.4 through 6.6 show the performance results on the various uniprocessor machines. These tables show the real and CPU times in seconds for the analysis on the different machines, using both programs. In addition, these tables also indicate the number of page faults, requiring pages to be read from disk, that occur during the analysis. Tables B.1 through B.3 provided in appendix B give the profile information showing what percentage of the CPU time was spent in the main components of the domainChanged(), solveCurrentStep(), and update() method calls, which are the methods invoked when analyze() is invoked on a **StaticAnalysis** object. Similar profile information for the page faults on the limited memory ALPHA and DEC machines are shown in tables B.4 and B.5.

| | Object-Oriented | | | Procedural | | | |
|---|---|---|---|---|---|---|---|
| Example | Real | CPU | # Page | Real | CPU | # Page | $\frac{CPU{+}{+}}{CPU}$ |
| 2dF1 | 1.08 | 1.01 | 0 | 0.75 | 0.73 | 0 | 1.38 |
| 2dF2 | 2.13 | 2.00 | 0 | 1.66 | 1.61 | 0 | 1.24 |
| 2dF3 | 3.73 | 3.53 | 0 | 3.25 | 3.15 | 0 | 1.12 |
| 2dF4 | 4.50 | 4.25 | 0 | 3.93 | 3.80 | 0 | 1.12 |
| 2dF5 | 5.26 | 4.98 | 0 | 4.58 | 4.45 | 0 | 1.12 |
| 2dF6 | 6.05 | 5.75 | 0 | 5.26 | 5.08 | 0 | 1.13 |
| 3dF3 | 9.01 | 8.75 | 0 | No Program Developed | | | |
| 3dF4 | 11.13 | 10.77 | 0 | No Program Developed | | | |
| 3dF5 | 12.98 | 12.51 | 0 | No Program Developed | | | |
| 3dF6 | 14.90 | 14.38 | 0 | No Program Developed | | | |

Table 6.4: Performance Results on HOLDEN

The CPU time required by the object-oriented program is greater than that required by the procedural program. This is because the object-oriented program spends more CPU cycles setting up and forming the systems of equations than the procedural program, as shown in figure 6.2. There are a number of reasons for the extra CPU time required by the object-oriented program:

1. The object-oriented program performs more work than the procedural program

| Example | Object-Oriented | | | Procedural | | | $\frac{CPU++}{CPU}$ |
|---|---|---|---|---|---|---|---|
| | Real | CPU | # Page | Real | CPU | # Page | |
| 2dF1 | 2.1 | 1.9 | 0 | 1.7 | 1.6 | 0 | 1.23 |
| 2dF2 | 4.1 | 3.8 | 0 | 3.70 | 3.5 | 0 | 1.08 |
| 2dF3 | 23.1 | 6.9 | 985 | 6.9 | 6.4 | 0 | 1.07 |
| 2dF4 | 57.2 | 8.6 | 4220 | 8.7 | 8.0 | 0 | 1.09 |
| 2dF5 | 81.5 | 10.1 | 6377 | 10.0 | 9.3 | 0 | 1.09 |
| 2dF6 | 105.0 | 11.4 | 8220 | 13.0 | 10.7 | 50 | 1.07 |
| 3dF3 | 83.2 | 17.0 | 6235 | No Program Developed | | | |
| 3dF4 | 130.0 | 20.7 | 10163 | No Program Developed | | | |
| 3dF5 | 147.6 | 21.8 | 11653 | No Program Developed | | | |
| 3dF6 | 173.4 | 25.7 | 13783 | No Program Developed | | | |

Table 6.5: Performance Results on ALPHA

| Example | Object-Oriented | | | Procedural | | | $\frac{CPU++}{CPU}$ |
|---|---|---|---|---|---|---|---|
| | Real | CPU | # Page | Real | CPU | # Page | |
| 2dF1 | 26 | 24 | 0 | 20 | 20 | 0 | 1.20 |
| 2dF2 | 87 | 63 | 14 | 57 | 56 | 0 | 1.12 |
| 2dF3 | 362 | 131 | 4146 | 123 | 118 | 0 | 1.11 |
| 2dF4 | 550 | 160 | 8908 | 150 | 142 | 0 | 1.12 |
| 2dF5 | 758 | 188 | 13889 | 184 | 167 | 0 | 1.13 |
| 2dF6 | 894 | 215 | 16837 | 391 | 194 | 3699 | 1.11 |
| 3dF3 | 875 | 341 | 13156 | No Program Developed | | | |
| 3dF4 | 1119 | 419 | 17983 | No Program Developed | | | |
| 3dF5 | Out of Memory | | | No Program Developed | | | |
| 3dF6 | Out of Memory | | | No Program Developed | | | |

Table 6.6: Performance Results on DEC

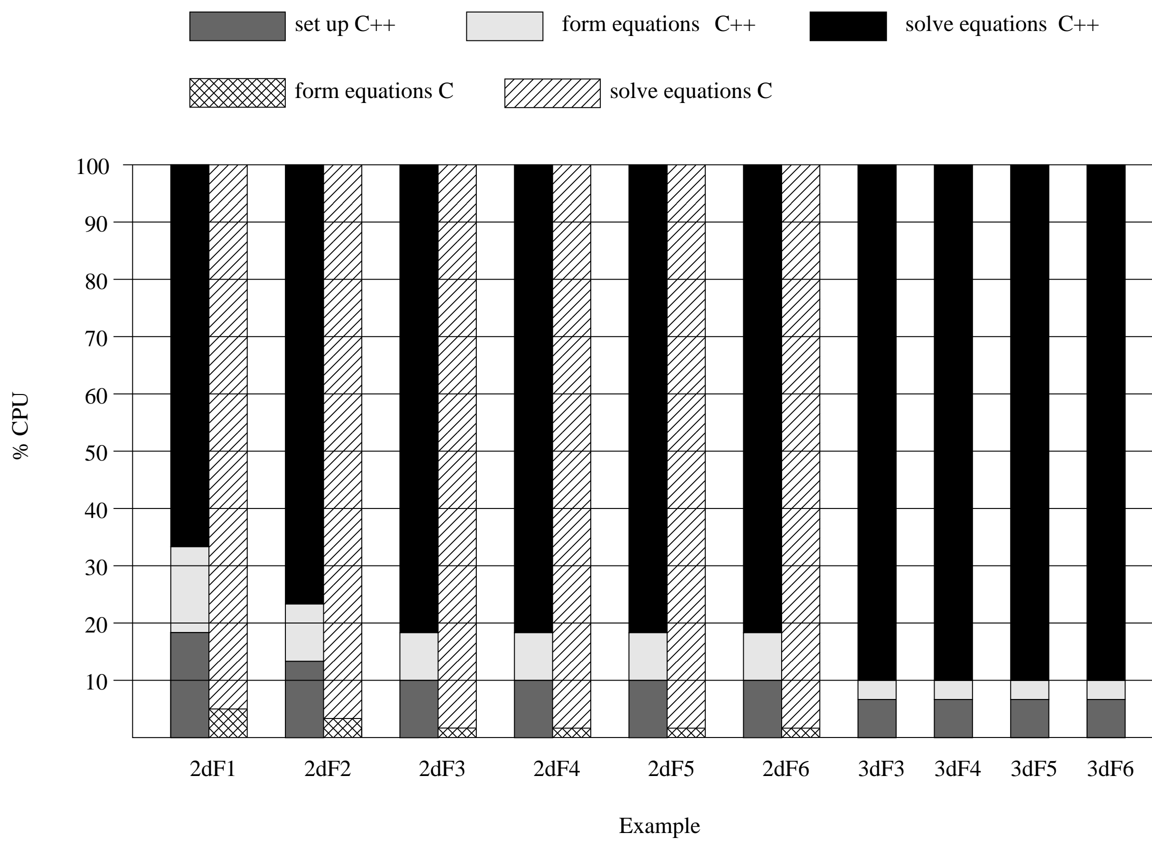



Figure 6.2: Profile of CPU Time for C++ Program on ALPHA

in setting up and forming the system of equations. For example, the object-oriented program spends upwards of 2% of the CPU time forming the element residual and 3% of its time forming the degree-of-freedom graph. These are operations not performed in the procedural program.

2. There are considerably more method calls in the object-oriented program than procedural calls in the procedural programs. For example, whereas the procedural program performs 3064 and 8094 procedure calls to analyze models 2dF1 and 2dF6, the object-oriented program requires over 2.6 million and 7.6 million method calls when analyzing these same two models.

   This is typical of object-oriented programs and is due to the fact that there are many more levels of abstraction, and hence indirection, in object-oriented programs than in procedural programs. For example, the methods invoked in the object-oriented program when formTangent() is invoked on a **FE_Element** object. This method will cause formElementTanget() to be invoked on the **Integrator** object, which will in turn will cause zeroTang() and addKtoTang() to be invoked on the **FE_Element** object. addKtoTang() invokes getStiff() on the **FE_Element**s associated **Element** object. This in turn invokes getCrd() on all the **ELement**s **Node** objects. In the procedural program, there is only a single procedural call for each element. For problems involving more complicated element calculations, the relative additional cost of the element calculations in an object-oriented enviroment to that in a procedural enviroment will be diminished.

3. In the object-oriented program it is sometimes necessary for the system to determine which method is to be invoked, when a method is invoked on an object. This is known as dynamic binding and is due to subclassing. For example when getStiff() is invoked on each **Element** object it is necessary for the system to determine which **Element** subclasses getStiff() method to invoke.

4. The object-oriented program spends more time managing memory than does the procedural program. For example, in the procedural program there are 10 calls to the kernel's memory allocation function malloc() for all the examples.

In the object-oriented program model 2dF1 requires the creation of over 85,740 objects and model 2dF6 over 224,740 objects.

This is typical of object-oriented programs and is a consequence of the dynamic allocation and destruction of the many objects used in the object-oriented programs, which is a result of the high degree of abstraction used. For example, in a procedural program one large section of memory can be allocated for holding all the element data. In the object-oriented program a separate location is allocated for each Element object. In addition for each **Element** object memory is allocated for the additional objects internal to the **Element**, such as an **ID** to identify the nodes and a **Matrix** object.

The results for the limited memory ALPHA and DEC machines also show that the object-oriented programs requires more memory, as seen by comparing page faults in tables 6.5 and 6.6. The increased memory requirement of the object-oriented program means that the procedural program will be able to handle larger problems than the object-oriented program for a fixed amount of memory. In addition, when page faulting occurs, especially for the machines with faster processors, the real time taken to perform the analysis degrades considerably. There are a number of reasons for the additional memory requirements of the object-oriented program:

1. The use of the **FE_Element** objects requires that a **Matrix** object and **ID** object be created for each **FE_Element** object. In addition, the creation of a **Graph** object, needed to provide the **LinearSOE** object with information about the degree-of-freedom connectivity, requires a **Vertex** object be created for each degree-of-freedom. Each **Vertex** requires an **ID** object to store it's adjacency information. This allocation of memory is not required in the procedural program.

2. Due to the large number of dynamically created objects, the demands on the heap tends to be greater for an object-oriented programs than for a procedural programs.

3. Due to the large number of method calls which can be invoked when a method

is invoked on an object, the requirements on the stack are greater for an object-oriented program than for a procedural program.

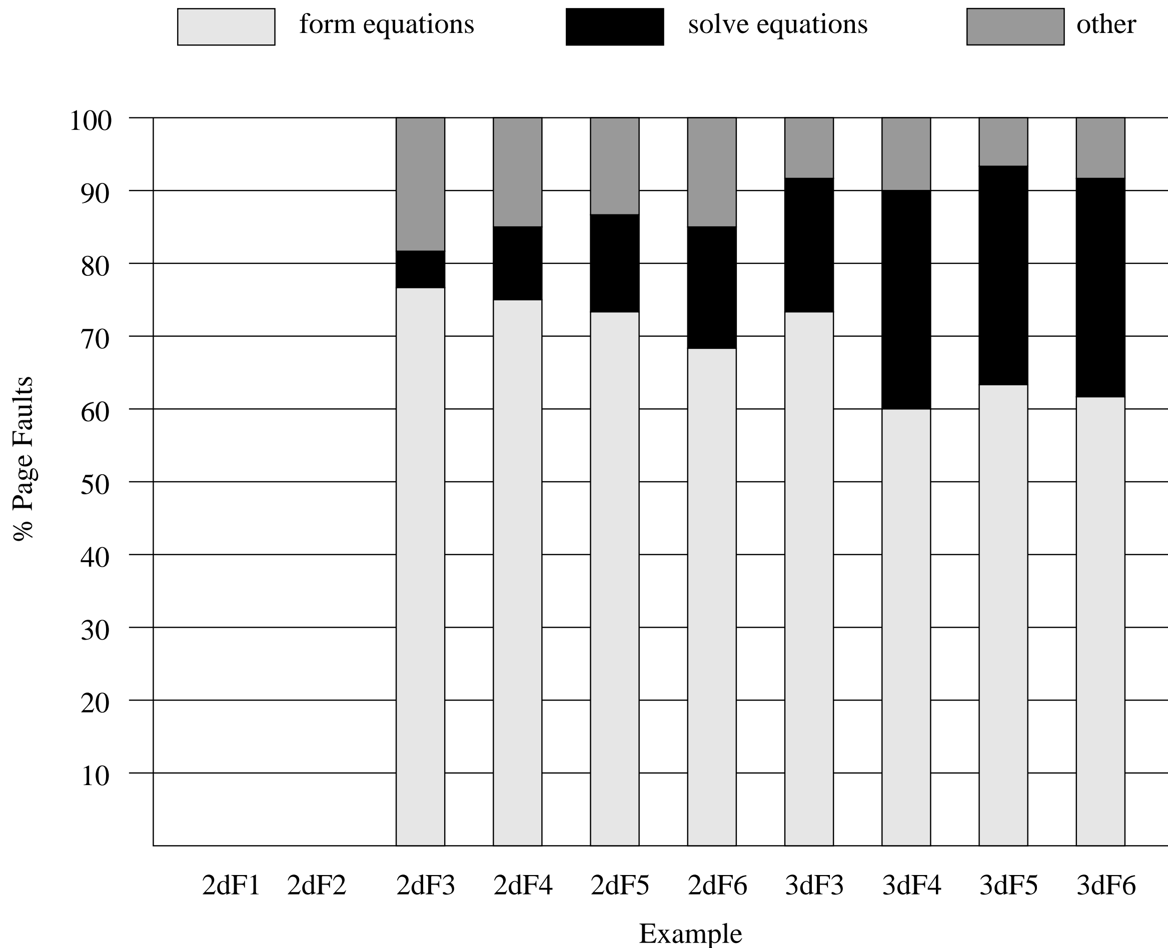



Figure 6.3: Profile of Page Faults for C++ Program on ALPHA

When paging does occur, the number of page faults is quite high. As can be seen in figure 6.3, the page faulting is not restricted to that portion of the program solving the system of equations. The high page rates are due to the fact that the programmer has little control over the location in the virtual address space allocated to each of the many objects, which results in poor data locality. For example, while only at most 7% of the CPU time is spent in `formTangent()` over 60% of the page faults occur here. This is because this method, as previously discussed, is invoking further methods on objects that are scattered all over the virtual address space of the program. To reduce the

number of page faults, certain classes could be rewritten to provide their own memory management (Stroustrup, 1991). The revised classes would be responsible for looking after large contiguous portions of memory, allocating portions of this memory to their objects and to the aggregate objects used by these objects. For example, the virtual address allocated to the **Element** objects of a certain class, along with their **Matrix**, **Vector** and **ID** objects, would be in a contiguous portion of the virtual address space.

## 6.4 Evaluation of the Object-Oriented Design on Parallel Machines

### 6.4.1 Introduction

In this section the performance of the object-oriented program when executing on a parallel machine is evaluated by comparing it with the execution of the object-oriented program executing on a single processing unit of the parallel machine. The parallel machines that are used are the networked ALPHA and DEC machines presented in table 6.3. The performance of the program is evaluated by measuring the real time taken to perform the analysis and the number of page faults that occur. These are then compared to the real time and CPU time taken to solve the problem on a single processing unit. The real time is used as the metric for the algorithmic speedup and the CPU time for the speedup:

$$AS_p = \frac{T_1}{T_p}$$

$$S_p = \frac{CPU_1}{T_p}$$

The CPU time is used for the speedup because page faults cause a serious degradation in performance and frontal solvers could be used to solve the equations, which would reduce considerably the number of page faults. This is a conservative measure as frontal solvers would be slower in terms of CPU time for the examples analyzed, which generate systems of equations which have relatively constant and narrow band profiles, and the reported CPU time is always less than the reported real time because of system overhead.

To perform the static analysis in parallel the substructuring domain decomposition method is used. The domain is partitioned using a multilevel strategy and the spectral bisection method. The subdomain equations and the interface equation are stored using a profile storage scheme. The interface problem is solved directly. This is as shown in the C++ program shown below:

```
001  main() {
002          /* ask user for the number of subdomains */
003          int numSubdomains;
004          cout << "Enter Number of Subdomains: ";
005          cin >> numSubdomains;
006
007          /* create the partitioned domain and model builder */
008          Metis theGraphPartitioner;
010          DomainPartitioner thePartitioner(theGraphPartitioner);
011          PartitionedDomain theDomain(thePartitioner);
012
013          /* create the subdomains and add to the domain
014          ObjectBroker theObjectBroker;
015          AlphaMachineBroker theMachineBroker;
016          for (int i=1; i<=numSubdomains; i++) {
017                  TCP_Socket theChannel;
018                  ShadowSubdomain theSubdomain(theChannel, theObjectBroker);
019                  Transformation theConstraintHandler;
020                  RCM theGraphNumberer;
021                  DOF_Numberer theDOFNumberer(theGraphNumberer);
022                  AnalysisModel theModel;
023                  ProfileSPDSOE_Substr_Solver theSolver;
024                  ProfileSPDSOE theLinSOE(theSolver);
025                  StaticIntegrator theIntegrator;
026                  DomainDecompAlgo theSolnAlgo;
027                  DomainDecompAnalysis theAnalysis(theSubdomain, theConstraintHandler,
028                          theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinSOE);
029                  theDomain.addSubdomain(theSubdomain);
030          }
031
032          /* create a model builder and build the model */
033          Quick2dFrame theModelBuilder(theDomain);
034          theModelBuilder.buildModel();
035
036          /* create a timer and start it running */
037          Timer theTimer;
038          theTimer.start();
039
040          /* partition the domain into the subdomains */
041          theDomain.partition(numSubdomains);
042
043          /* create the analysis */
044          Transformation theConstraintHandler;
```

```
045        RCM theGraphNumberer;
046        DOF_Numberer theDOFNumberer(theGraphNumberer);
047        AnalysisModel theModel;
048        DirectProfileSPDSOE theSolver;
049        ProfileSPDSOE theLinSOE(theSolver);
050        StaticIntegrator theIntegrator;
051        Linear theSolnAlgo;
052        StaticAnalysis theAnalysis(theDomain,theConstraintHandler,
053              theDOFNumberer, theModel, theSolnAlgo, theIntegrator, theLinSOE);
054
055        /* perform the analysis */
056        theDomain.setLoadCase(1);
057        theAnalysis.analyze;
058
059        /* stop the timer and print out the results */
059        theTimer.pause();
060        theTimer.print(cout);
061   }
```

### 6.4.2 Results

The results showing the performance obtained on these machines is as shown in tables 6.7 and 6.8. The speedup and algorithmic speedup are as shown graphically in figures 6.4 and 6.5. These figures also show the load imbalance among the subdomains, by providing the maximum ratio of the result of invoking getCost() on the **ActorSubdomain** objects, and they provide an indication of the percentage of the total time can be attributed to page faulting, communication and computation. In tables B.6 through B.21 the time taken to perform the computation is divided into the time taken to partition the domain and the time taken to perform the analysis. In addition these tables also provide information about the CPU time required to solve the interface problem and the times required by the individual subdomain processes during the analysis.

The results demonstrate the following:

1. A parallel environment allows problems to be analyzed which cannot be analyzed on a single uniprocessor machine, because of memory limitations. For example, problems 3dF5 and 3dF6, cannot be analyzed on a single DEC workstation, but they can be analyzed using a number of these machines in parallel.

2. Large problems can be analyzed faster in a parallel environment than in a

| Example | NP=1 | | | NP=3 | | NP=4 | | NP=5 | | NP=7 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Real | CPU | # PF | REAL | # PF | Real | # PF | Real | # PF | Real | # PF |
| 2dF1 | 2.1 | 1.9 | 0 | 2.85 | 0 | 3.65 | 0 | 3.05 | 0 | 4.45 | 0 |
| 2dF2 | 4.1 | 3.8 | 0 | 4.05 | 0 | 4.28 | 0 | 3.65 | 0 | 5.85 | 0 |
| 2dF3 | 23.1 | 6.9 | 985 | 6.08 | 0 | 6.05 | 0 | 5.05 | 0 | 7.42 | 0 |
| 2dF4 | 57.2 | 8.6 | 4220 | 8.05 | 0 | 8.23 | 0 | 5.98 | 0 | 9.28 | 8 |
| 2dF5 | 81.5 | 10.1 | 6377 | 16.15 | 45 | 8.78 | 0 | 8.62 | 0 | 10.85 | 17 |
| 2dF6 | 105.0 | 11.4 | 8220 | 15.40 | 125 | 28.37 | 8 | 10.30 | 5 | 13.93 | 25 |
| 3dF3 | 83.2 | 17.0 | 6235 | 10.15 | 0 | 22.12 | 0 | 16.70 | 0 | 15.23 | 18 |
| 3dF4 | 130.0 | 20.7 | 10163 | 16.40 | 152 | 33.92 | 74 | 22.50 | 21 | 20.58 | 16 |
| 3dF5 | 147.5 | 21.8 | 11653 | 23.82 | 547 | 56.00 | 1176 | 32.88 | 62 | 29.57 | 198 |
| 3dF6 | 173.4 | 25.7 | 13787 | 52.55 | 6312 | 84.25 | 2770 | 38.63 | 69 | 45.63 | 577 |

Table 6.7: Performance Results on ALPHA Cluster

| Example | NP=1 | | | NP=3 | | NP=4 | | NP=5 | | NP=7 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Real | CPU | # PF | REAL | # PF | Real | # PF | Real | # PF | Real | # PF |
| 2dF1 | 26 | 24 | 0 | 23 | 0 | 40 | 0 | 22 | 0 | 38 | 2 |
| 2dF2 | 87 | 63 | 14 | 38 | 0 | 45 | 1 | 35 | 1 | 51 | 1 |
| 2dF3 | 362 | 131 | 4146 | 60 | 0 | 71 | 0 | 51 | 2 | 81 | 24 |
| 2dF4 | 550 | 160 | 8908 | 74 | 0 | 94 | 4 | 70 | 0 | 118 | 155 |
| 2dF5 | 758 | 188 | 13889 | 94 | 13 | 148 | 1 | 86 | 6 | 141 | 208 |
| 2dF6 | 894 | 216 | 16837 | 123 | 72 | 306 | 278 | 143 | 208 | 185 | 170 |
| 3dF3 | 875 | 341 | 13156 | 140 | 4 | 296 | 80 | 247 | 623 | 236 | 804 |
| 3dF4 | 1119 | 419 | 17983 | 286 | 4872 | 517 | 1000 | 325 | 860 | 309 | 1090 |
| 3dF5 | Not Enough Memory | | | 416 | 2800 | 614 | 2619 | 513 | 1042 | 415 | 1532 |
| 3dF6 | Not Enough Memory | | | 554 | 5186 | 881 | 7449 | 564 | 1422 | 648 | 1890 |

Table 6.8: Performance Results on DEC Cluster

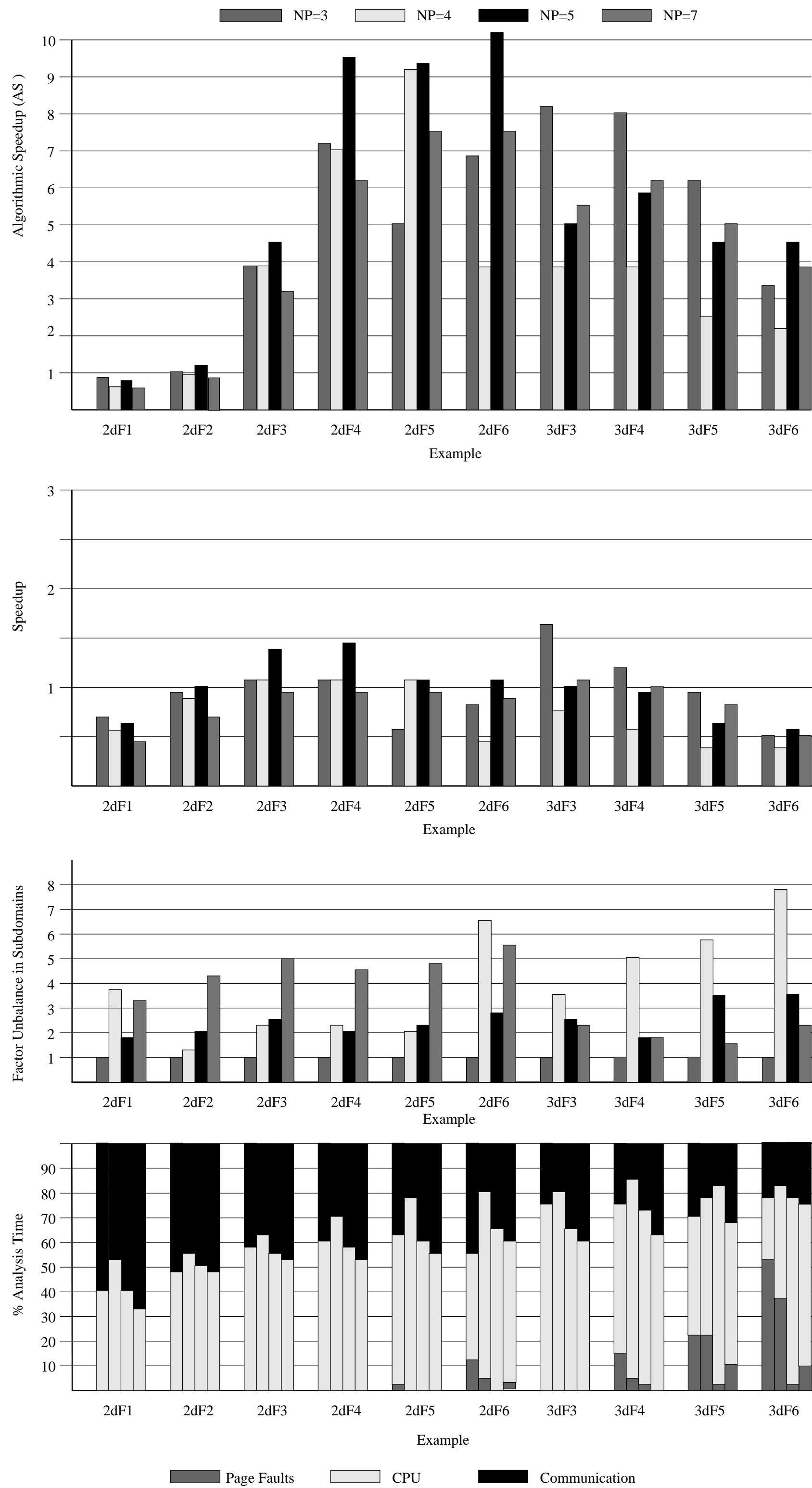



Figure 6.4: Parallel Performance on ALPHA

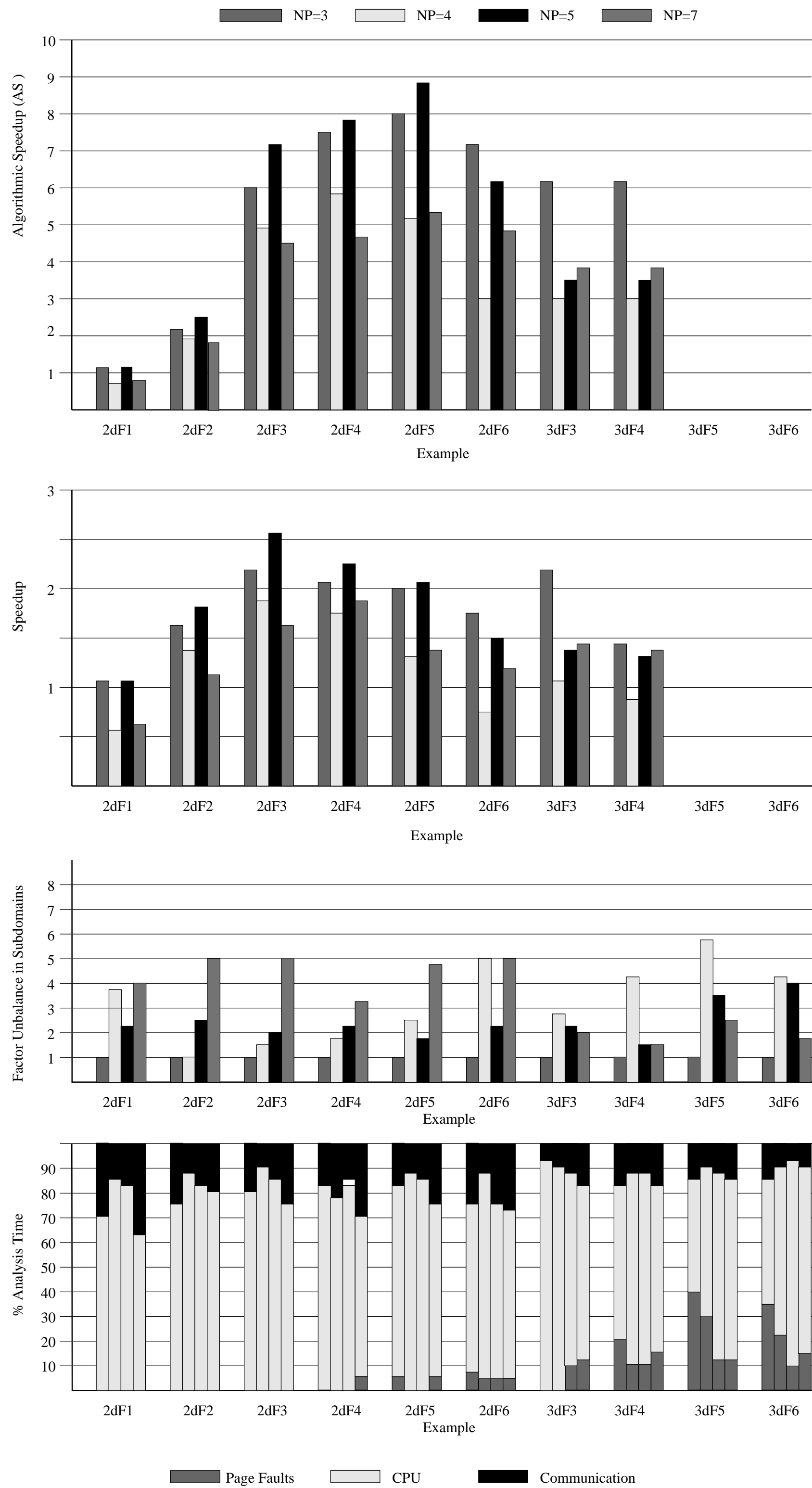



Figure 6.5: Parallel Performance on DEC

sequential environment. This is despite the overhead of partitioning the problem required in the parallel analysis, which accounts, on average, for over 35% of the time taken to perform the analysis. For example for all examples but 2dF1 on the ALPHA cluster, algorithmic speedup is obtained.

The primary reason for this marked increase in performance is in the reduction in the number of page faults that occur in the parallel environment. This is demonstrated by the fact that the algorithmic speedup is considerably more than the speedup when page faulting occurs, as demonstrated in figures 6.4 and 6.5. The figures also demonstrate that the effect of reducing the number of page faults is greater for the ALPHA machine than the DEC machine. This is due to the fact that the ratio of processor speed to disk access is greater for the ALPHA machine than the DEC machine.

3. The ratio of processor speed to the cost of communication has a significant influence on the performance of the parallel machine. As the ratio of processor speed to communication speed reduces, the speedup obtained will increase. This is demonstrated by the fact that the speedup values for the DEC machines are greater than those for the ALPHA machines.

4. The performance does not necessarily improve as more processors are added. There are a number of reasons for this:

   (a) As more processors are added, the time required to partition the problem increases, accounting for over 50% of the total time required to perform the computation for some examples. This additional time is due both to the extra time required to partition the element graph and the extra time required to construct the degree-of-freedom graph given the increase in connectivity. To overcome this problem a parallel graph partitioning package could be used, e.g. ParMetis (Schloegel et al., 1997), and more efficient graph classes could be developed.

   The overhead associated with the partitioning phase will be negated when an iterative solution strategy is specified by the analyst, for example in non-linear or transient analysis. The reason for this is that the partition-

ing is only done once, while the setting up and solving of the equations are done numerous times. This is as seen in figure 6.6, where the algorithmic speedup and speedup values when 25 iterations are performed on the ALPHA machine is presented. The results show that the speedup values obtained are almost double that obtained for the single iteration case, presented in figure 6.4.

(b) As more processors are added, the size of the interface problem grows. The larger interface problem requires additional CPU cycles to solve and can also result in page faulting. To overcome this problem the interface problem could be solved in parallel.

(c) As more processors are added, the amount of communication required to perform the analysis increases, as shown in figures 6.4 and 6.5. For smaller problems and slower communication networks, the computation can be dominated by the communication, for example the communication accounts for over 60% of the time spent when analyzing 2dF1 with six subdomains on the ALPHA machine.

(d) When using more than two processors there is sometimes a problem of load imbalance on the processors. In certain instances one processor is taking almost five times longer than another processor to perform the subdomain calculations. This is due to both the partitioning of the domain, generated by the **Metis** object, and the numbering scheme used in assigning equation numbers to the degrees-of-freedom, generated by the **RCM** object. The partitioner does not take into account that, when using the substructuring method, the amount of work to be done in a subdomain depends not only on the number of elements in a subdomain, but also on the relationship between the interior and exterior degrees-of-freedom. The algorithm employed by the **RCM** object does not take into account that, when using substructuring, the degrees-of-freedom in a partition need to be split up into two groups, interior and exterior, and numbered accordingly.

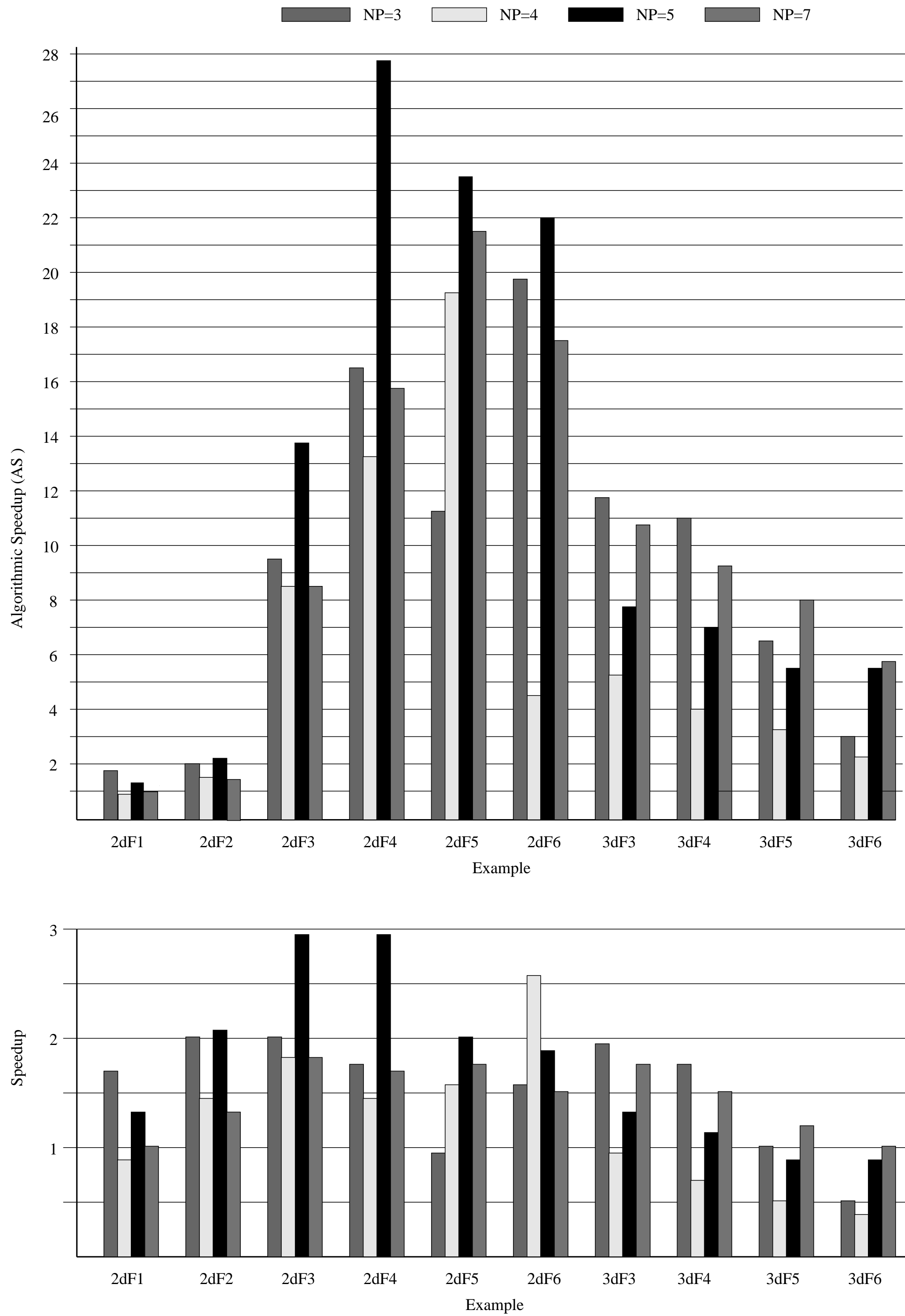



Figure 6.6: Parallel Performance on ALPHA for 25 Iterations

## 6.5 Summary

In this chapter it has been shown that an object-oriented program, based on the design presented in the previous chapters, will require more CPU time than a corresponding procedural program. For large problems, the additional CPU time required by the object-oriented program will be less than 5% of the total CPU time. This is acceptable given the flexibility and extensibility offered to the analyst as a consequence of the object-oriented design.

Of greater concern to the analyst, however, is the greater memory requirements of the object-oriented program over that of a procedural program. The additional memory requirements limit the size of the problems that can be analyzed and can result in significantly slower execution times if page faulting occurs.

To overcome this memory problem, the design presented allows the analyst to use a parallel machine, such as a network of computers commonly found in most offices. The results show that if enough machines are made available the memory problems associated with the object-oriented program can be negated.

A problem of load imbalance among processors can however limit the performance of the parallel program as it results in CPU cycles being wasted. As discussed, the load imbalance can be a result of the partitioning scheme and degree-of-freedom numbering algorithm used by the analyst. New **GraphPartitioner** and **GraphNumberer** subclasses could be introduced to overcome this problem. The object design presented allows the analyst to choose the specific classes of **GraphPartitioner** and **GraphNumberer** most suitable for the problem at hand, further demonstrating the flexibility and extensibilty of the design. An alternative solution to the load imbalance problem is, for non-linear and transient problems, to use dynamic load balancing. This will be looked at in chapter 7.

# Chapter 7

# Dynamic Load Balancing for Finite Element Analysis

In this chapter the design presented in the previous chapters is extended to allow for dynamic load balancing during the analysis. A review of existing approaches for dynamic load balancing in a finite element analysis is first presented. A new approach is then given, along with the extensions and modifications to the design to accommodate this new approach. Finally, results showing the performance improvements that can be obtained using this new approach are presented and discussed.

## 7.1 Introduction

Static load balancing, where the work is assigned to the processors at the start of the computation and does not vary as the computation proceeds, is not always adequate for parallel processing. This is especially true for finite element analysis on networks of workstations. There are a number of reasons for this:

1. The initial partitioning of the domain may lead to load imbalance among the processors, as seen in chapter 6.

2. In a non-linear analysis, certain regions of the domain may go non-linear at any stage in the solution. For elements in the non-linear range, the amount of computation required for the state determination can be significantly greater than when in the linear range. This can result in load imbalance among the processors, if a large number of elements in the non-linear range exist on a single processor.

3. On a traditional network of workstations with which the users share the system resources, the load on the individual processors can change at any moment as new users log on, or as existing users begin new or terminate old processes. On dedicated parallel computers, this is not an issue.

4. On a network of workstations, the performance of the workstations may vary. This can lead to load imbalance if the initial partitioning does not take into account the processing capabilities of the individual processors.

Dynamic load balancing, wherein the work assigned to the individual processors varies as the computation progresses, can overcome the performance problems associated with static load balancing. Dynamic load balancing techniques have been used in this and other areas, particularly in parallel computational fluid dynamics (CFD) and distributed and parallel operating systems research. In CFD applications where adaptive mesh control is used, quasi-static load balancing is typically employed (Nicol and Reynolds Jr., 1990; Williams, 1991; Pramono et al., 1994; Chien et al., 1994; Van Driessche and Roose, 1995). That is, load balancing is done whenever changes to the

mesh would cause a severe load imbalance. The typical strategy employed is to repartition the new mesh, that is to perform static load balancing again. In distributed and parallel operating systems research, the goal is to attempt to balance the workload of all runnable processes among the processors. This is typically done in one of two ways:

1. Task queue approach (Wang and Morris, 1985): The waiting runnable processes are placed in a queue. When a processor becomes free, the processor is assigned a process from the queue.

2. Diffusion approach (Wang and Morris, 1985; Ni et al., 1985; Barak and Shiloh, 1985; Eager et al., 1986; Lin and Keller, 1987; Cybenko, 1989; Douglis and Ousterhout, 1989; Chowdhury, 1990; Xu and Hwang, 1993): The processes, assigned initially to a specific processor, move around as processes change and/or as more load enters the system. Typically background processes run on each processor, to monitor the load on the processor, and they communicate with a managing process, which monitors the load in the system. As unbalance occurs and certain processors become overloaded, processes are moved from one processor to another in the system.

3. Switch approach (Chakrabarti et al., 1997): Tasks, which represent work to be done, are placed in a queue when ready to run. Each task contains data parallel operations, that is operations which can be performed in parallel on arrays, and has a speedup profile, which is a function of the task size, the machine and the type of work being performed by the task. The tasks are placed into one of two queues based on their speedup profiles; one queue, $J_1$, contains all tasks that will only be assigned a single processor and the other queue, $J_P$, contains tasks that will be assigned to run all all the processors. Initially, all the tasks in $J_P$ are run and, when $J_P$ is empty, a switch is made and all the processes in $J_1$ are run, using a greedy algorithm to assign the tasks to the processors. When $J_1$ is empty, a switch back to $J_P$ is made and the process continues.

In this chapter a review of an existing approach for dynamic load balancing a finite element analysis is presented, with a discussion on how the software design can

be modified for this approach. Two new approaches for dynamic load balancing an analysis are then given. The modifications to the existing framework to accommodate one of these new approaches is then presented. Finally, empirical results demonstrate that the performance of the parallel program can be improved using simple load balancing algorithms.

## 7.2 Existing Approaches to Dynamic Load Balancing the Finite Element Analysis

Santiago and Law (1996) perform dynamic load balancing using a task queue approach. In this work the domain is broken down into many subdomains, with the number of subdomains being much greater than the number of processors. The subdomain operations, i.e. formTangent(), represent tasks. The subdomain data, such as elements, nodes and loads, are kept by the main process. Also running on the parallel machine are a number of slave processes. A slave process asks the main process for work when the current subdomain task assigned to the process has been completed. During the analysis, as the main process iterates over the subdomains it will assign a subdomain task to next available slave process.

The problems with this approach are:

1. There is a considerable amount of communication. For each task assigned to a slave process, the master process must send all the subdomain data to the slave process. Upon completion, the slave process sends the results and the subdomain data back to the main process.

2. The approach will not scale well to larger problems because the master process must hold all the model data.

To accommodate this approach in the frameworks presented in the previous chapters, a new **SubdomainProcess** class and two new **Subdomain** subclasses, **ShadowSubdomainTask** and **ActorSubdomainTask** could be introduced. The **ShadowSubdomainTask** object resides in the main process and holds all the subdomain data. When a computationally demanding method, such as getTangent(), is invoked

on the object it would invoke a method in the **SubdomainProcess** object to obtain a **Channel** object with which to communicate with the next available **ActorSubdomainTask** object. The **ActorSubdomainTask** object, which resides in a remote actor process, is responsible for receiving the subdomain data, performing the operation and returning the results and subdomain data to the **ShadowSubdomainTask** object via its **Channel** object. When done, the **ActorSubdomainTask** object sends a message to the **SubdomainProcess** object telling that object that it is done and is ready to work on another subdomain.

## 7.3 New Approaches for Dynamic Load Balancing

A number of different approaches than that used by Santiago and Law (1996) could be employed for performing dynamic load balancing during a finite element analysis. In this section two new approaches are presented:

1. One approach is to partition the domain into subdomains, with the number of subdomains being less than the number of available processors. Initially each subdomain is assigned a number of processors based on the *a-priori* determination of the workload in each subdomain (Synn and Fulton, 1995). As the computation progresses and load imbalance occurs, the number of processors assigned to each subdomain is modified to reduce the load imbalance.

2. An alternative approach is to partition the domain into subdomains, with the number of subdomains being equal to the number of processors. Each subdomain is assigned to a single processor. As the computation progresses and load imbalance occurs, the original partitioning is modified, by migrating elements between subdomains to reduce the load imbalance.

The first approach, a multilevel subdomain approach, can result in situations in which numerous processors try to communicate at once, which causes an considerable degradation in the performance of a NOW, which relies on an ethernet for communication, (Cabrera et al., 1988). The second approach does not generate such communication patterns. For this reason and due to the fact that the NOWs available for this research use ethernets, the second approach is developed in this research.

## 7.4 Extension of Framework for Dynamic Load Balancing

To incorporate the second approach to dynamic load balancing described in the previous section, the software design presented in the previous chapters is modified and extended. A new class **LoadBalancer** is introduced and the **PartitionedDomain**, **Subdomain** and **DomainPartitioner** classes are modified or extended.

### 7.4.1 Modification to the PartitionedDomain Class

The commit() method of the **PartitionedDomain** class is modified so that after the normal commit operations are performed, the **PartitionedDomain** object can gather from the **Subdomain** objects the real time taken by them to perform their tasks during the previous iteration step. Given the load information the **PartitionedDomain** will invoke balance() on the **DomainPartitioner** object associated with it.

### 7.4.2 Extension to the Subdomain Class

The **Subdomain** class is extended to include a method getCost(). This method will return the real time spent by the **Subdomain** processing operations since the last invocation of the **Subdomain**'s getCost() method.

### 7.4.3 Extension to the DomainPartitioner Class

The **DomainPartitioner** class is now a subclass of the **GraphBalancer** class, which defines methods for balancing a weighted graph. This requires that additional methods, as shown in figure 7.1, are implemented for the **DomainPartitioner** class. The additional methods can be divided into two categories:

1. A method balance() which is invoked when a load balancing step is to be performed. If no **LoadBalancer** object is associated with the **DomainPartitioner** no load balancing is performed. If a **LoadBalancer** is associated with the object, the **DomainPartitioner** invokes balance() on the **LoadBalancer**

```
class DomainPartitioner: public GraphBalancer {
    public:

        DomainPartitioner(GraphPartitioner &theGraphPartitioner);
        DomainPartitioner(GraphPartitioner &theGraphPartitioner
                          LoadBalancer &theGraphLoadBalancer);
        virtual ~DomainPartitioner();

        virtual void setPartitionedDomain(PartitionedDomain &theDomain);
        virtual int partition(int numParts);

        // new methods for load balancing defined in GraphBalancer
        virtual int balance(Graph &theWeightedPartitionGraph);
        virtual int swapVertex(int from, int to, int vertexTag);
        virtual int swapBoundary(int from, int to);
        virtual int releaseVertex(int from,
                                  int vertexTag,
                                  const Graph &theWeightedPartitionGraph,
                                  bool okToReleaseToLighter);
        virtual int releaseBoundary(int from,
                                  const Graph &theWeightedPartitionGraph,
                                  bool okToReleaseToLighter);
        virtual Graph &getColoredGraph(void);
};
```

Figure 7.1: Revised Interface for the **DomainPartitioner** Class

object. When this object is done, the **DomainPartitioner** will check to see if the **Subdomain** objects have been modified. If modified, it invokes domainChanged() on them and on the **PartitionedDomain** object.

2. Methods swapVertex(), swapBoundary, releaseVertex(), releaseBoundary(), getColoredGraph(), and getPartitionGraph() which are invoked by the **LoadBalancer** object during the load balancing. While these methods are responsible for moving the **Element**, **Node**, **Load** and **Constraint** objects among the **Subdomains**, the methods provide a simple interface which can be used for other graph balancing problems. This would allow the **LoadBalancer** classes to be used in other frameworks which have a **GraphBalancer** class.

### 7.4.4 LoadBalancer Class

The **LoadBalancer** object is responsible for modifying the partition to obtain a better load balance among the partitions. The **LoadBalancer** class, whose interface is shown in figure 7.2, is an abstract base class. It defines one method, balance(), for which all subclasses must provide an implementation.

```
class LoadBalancer {
    public:
        LoadBalancer();
        virtual ~LoadBalancer();

        virtual void setLinks(GraphBalancer &theBalancer)
        // method to balance the load among the processors
        virtual int balance(graph &weightedPartitionGraph) =0;

    protected:
        GraphBalancer *theGraphBalancer;
};
```

Figure 7.2: Interface for the **LoadBalancer** Class

When balancing, the **LoadBalancer** object uses the methods defined in the **DomainPartitioner** object. For example, the **HeavierToLighterNeighbours** subclass, which is shown in figure 7.3, uses the swapBoundary() method. The load bal-

ancing algorithm employed by this subclass will shed the elements on the boundary between all partitions and their less loaded neighboring partitions subject to the following conditions:

1. No shedding of elements will occur between a partition and its neighbor if the ratio of the load in the partition and its neighbor is less the a certain factor. This factor, loadFactor, is used to recognize that there is a point where the costs associated with the load balancing will be greater than the benefits obtained. The cost of swapping elements between partitions is due to both the communication costs involved in sending **Element**, **Node** and **Load** objects and the cost of invoking domainChanged() on the **Subdomains** when the load balancing is completed.

2. No swapping of elements between partitions occurs if elements were moved in the previous load balancing step. This is done to ensure that page faulting, which can greatly increases in number when domainChanged() is invoked on a **Subdomain**, does not cause load balancing to occur when the next analysis step would not require any due to computationally balanced **Subdomains**.

## 7.5 Evaluation of the Effect of Dynamic Load Balancing on Performance

### 7.5.1 Introduction

This section examines the effect of dynamic load balancing on the performance of the parallel program, when the load imbalance on the processors is a consequence of the initial partitioning. This is done by comparing the time taken to perform 25 analysis steps in which no dynamic load balancing is performed to 25 steps in which the **HeavierToLighterNeighbours** dynamic load balancing strategy, presented in figure 7.3, is used. In the analysis, the elements remain in the elastic range and the only processes running on the machine are due to the parallel program. To perform 25 steps with no load balancing, line 052 of the code presented in section 6.4.1 is replaced with the following:

```
class HeavierToLighterNeighbours: public LoadBalancer {
    public:
        HeavierToLighterNeighbours(double loadFactor);
        virtual ~HeavierToLighterNeighbours();

        int balance(Graph &weightedPartitionGraph);
};

HeavierToLighterNeighbours::balance(Graph &weightedPartitionGraph){
    // only do the algorithm if no swapping last time
    if (lastSwap == true) {
        lastSwap = false;
        return 0;
    }

    // iterate through the vertices
    VertexIter &theVertices = weightedPartitionGraph->getVertices();
    while ((vertexPtr == theVertices()) ! = 0) {
        int vertexTag = vertexPtr->getTag();
        double vertexLoad = vertexPtr->getWeight();
        const ID &neighbours = vertexPtr->getAdjacency();
        // look at all the vertices neighbours
        for (int i=0; i<neighbours.Size(); i++) {
            int otherTag = neighbours.Size();
            vertex *otherPtr = weightedPartitionGraph->getVertexPtr(otherTag);
            double otherLoad = otherPtr->getWeight();
            // shed load to lighter neighbor if unbalance large enough
            if (vertexLoad/otherLoad > loadFactor) {
                theGraphBalancer->swapBoundary(vertexTag,otherTag);
                lastSwap = true;
            }
        }
    }
}
```

Figure 7.3: The **HeavierToLighterNeighbours** Class

```
052        PDeltaAnalysis theAnalysis(25,theDomain,theConstraintHandler,
```

To perform 25 analysis steps using the **HeavierToLighterNeighbours** dynamic load balancing algorithm, lines 009 and 010 of the code just presented for the 25 steps with no load balancing is replaced with the following:

```
009        HeavierToLighterNeighbours theLoadBalancer(factor);
010        DomainPartitioner(thePartitioner, theLoadBalancer);
```

### 7.5.2 Results

Table 7.1 shows the effect of performing dynamic load balancing on the ALPHA machine for the examples presented in section 6.2 using two different values for the loadFactor of 1.5 and 2.0. In table 7.1 both the times taken to perform all 25 steps of the analysis, $T_{1-25}$, and the time taken to perform the 25'th step of the analysis, $T_{@25}$.

The results demonstrate the following:

1. Dynamic load balancing can improve the performance of the parallel program. For example, the performance of the parallel program, $T_{1-25}$, is improved by a factor of 2.5 when using 3 subdomains (NP=4) and a loadFactor of 1.5 to analyze example 3dF5 on the ALPHA machine.

2. Dynamic load balancing can also degrade the performance. For example, the parallel program $T_{1-25}$, is slowed down by a factor of 1.6 when using 6 subdomains (NP=7) and a loadFactor of 1.5 to analyze example 3dF3 on the ALPHA machine.

3. The potential for improvement in the parallel performance that can be achieved using dynamic load balancing is greater when the communication costs, relative to computation costs, are low. This is demonstrated by the fact that the improvement in performance is, typically, greater for the 3 subdomain (NP=4) case than the 6 subdomain (NP=7) case. When using 3 subdomains (NP=4) processors and a loadFactor of 1.5 example 2dF1 requires 481 sec($T_{@25} = 14$ sec) on the DEC machine. Without load balancing the same example requires

| NP | Example | No Balancing | | Dynamic Load Balancing | | | |
|---|---|---|---|---|---|---|---|
| | | | | loadFactor = 1.5 | | loadFactor = 2.0 | |
| | | $T_{1-25}$ | $T_{@25}$ | $T_{1-25}$ | $T_{@25}$ | $T_{1-25}$ | $T_{@25}$ |
| 4 | 2dF1 | 47.93 | 1.80 | 35.90 | 1.17 | 37.88 | 1.28 |
| | 2dF2 | 58.13 | 2.17 | No Balancing | | No Balancing | |
| | 2dF3 | 86.27 | 3.35 | 73.50 | 2.70 | 82.20 | 3.07 |
| | 2dF4 | 140.72 | 5.45 | 112.43 | 4.22 | 114.42 | 4.30 |
| | 2dF5 | 139.27 | 5.33 | 130.02 | 4.73 | 135.93 | 5.13 |
| | 2dF6 | 664.67 | 27.77 | 352.47 | 5.83 | 364.00 | 7.58 |
| | 3dF3 | 457.75 | 18.08 | 392.63 | 14.75 | 403.82 | 14.48 |
| | 3dF4 | 732.48 | 28.10 | 395.50 | 13.06 | 437.05 | 15.45 |
| | 3dF5 | 1156.43 | 55.47 | 461.05 | 14.93 | 566.58 | 20.30 |
| | 3dF6 | 1753.20 | 72.38 | 919.48 | 31.32 | 730.92 | 20.53 |
| 5 | 2dF1 | 28.98 | 1.15 | 29.95 | 1.08 | 28.96 | 1.07 |
| | 2dF2 | 38.97 | 1.42 | 39.70 | 1.36 | 38.53 | 1.36 |
| | 2dF3 | 54.02 | 2.02 | 59.87 | 2.03 | 53.75 | 1.95 |
| | 2dF4 | 67.63 | 2.53 | 68.32 | 2.50 | 66.98 | 2.45 |
| | 2dF5 | 113.20 | 4.15 | 116.08 | 4.20 | 110.67 | 4.22 |
| | 2dF6 | 137.28 | 5.13 | 119.75 | 4.33 | 117.73 | 4.33 |
| | 3dF3 | 306.38 | 11.95 | 310.62 | 11.32 | 286.43 | 10.75 |
| | 3dF4 | 413.12 | 16.22 | 408.05 | 15.82 | 402.10 | 15.73 |
| | 3dF5 | 647.06 | 25.45 | 651.85 | 22.48 | 607.47 | 23.10 |
| | 3dF6 | 736.65 | 28.92 | 646.40 | 19.18 | 634.62 | 23.90 |
| 7 | 2dF1 | 41.85 | 1.30 | 41.55 | 1.55 | 37.90 | 1.28 |
| | 2dF2 | 61.90 | 2.17 | 62.30 | 2.28 | 59.03 | 2.05 |
| | 2dF3 | 87.85 | 3.13 | 85.96 | 3.03 | 85.38 | 3.06 |
| | 2dF4 | 118.92 | 4.47 | 114.13 | 4.26 | 118.45 | 4.23 |
| | 2dF5 | 125.82 | 4.65 | 130.76 | 4.50 | 119.06 | 4.03 |
| | 2dF6 | 171.88 | 6.40 | 200.23 | 8.67 | 160.72 | 5.50 |
| | 3dF3 | 222.32 | 8.65 | 356.57 | 10.97 | 242.83 | 9.03 |
| | 3dF4 | 311.12 | 11.95 | 338.00 | 12.70 | No Balancing | |
| | 3dF5 | 456.07 | 17.10 | 461.11 | 16.80 | No Balancing | |
| | 3dF6 | 704.23 | 26.92 | 688.95 | 23.90 | 684.48 | 23.83 |

Table 7.1: Effect of Dynamic Load Balancing on the Real Time using ALPHA for 25 Analysis Steps with the **HeavierToLighterNeighbours** algorithm

763 sec($T_{@25}$ = 30 sec). This is a gain in performance on the DEC machine of 1.58(2.14), while the gain in performance on the ALPHA machine is only 1.33(1.55).

4. The improvement in the parallel performance is greater when the imbalance among the processors is large. This is demonstrated by the fact that, for the 3 subdomain (NP=4) case, the gain in performance for example 2dF1 is greater than that for examples 2dF3, 2dF4, and 2dF5, even though the later problems require a smaller percentage of overall communication time.

5. If a considerable amount of load balancing occurs, the parallel program can spend a significant amount of time performing the load balancing. For example, for a number of examples, even though the performance at the 25'th iteration is better when dynamic load balancing is used, there is an overall loss in performance. This loss in performance would not have occurred if a large enough number of iterations is performed, such as for a time history analysis in which several thousand time steps are used.

6. The load balancing employed can result in a partitioning which requires more computations to solve than the original partitioning. This occurs in some of the analysis using 6 subdomains (NP=7) for the smaller two and three-dimensional problems with a small loadFactor. This is a consequence of the load balancing strategy that is employed, which does not attempt to minimize the interface boundaries between the partitions. The problem can be overcome for smaller problems by specifying larger values for loadFactor, for example a loadFactor of 2.0 for these problems, or by using a different load balancing strategy.

7. For certain problems that are just large enough to initiate page faulting on a processor, variability in the number of page faults occurring at each iteration can result in dynamic load balancing when none is really needed. The problem is overcome by setting a large loadFactor for these problems, or using parallel machines with memory large enough that page faulting is not an issue.

## 7.6 Summary

In this chapter the finite element design that had been presented in the previous chapters was extended to allow for dynamic load balancing a finite element analysis. It was shown that the extensions and modifications to the existing design to accomplish this was minimal.

It was also shown that dynamic load balancing a finite element analysis can improve the performance of the parallel program. These improvements were observed for the analysis of a number of examples using simple elements for situations in which no other users were running processes on the parallel machine. The potential for even greater improvements in performance exist for analysis involving more complex elements and non-linear state determination, for larger problems requiring more computation running on machines with large enough memories so page faulting is not an issue, and for the situations in which other users are on the system.

# Chapter 8

# Conclusions and Future Directions

## 8.1 Summary

The main objectives of this research were:

1. To develop, using object-oriented software design techniques, a design which would allow for flexible and extensible finite element programming.

2. To extend the design to accommodate efficient parallel finite element processing.

In chapter 1, a review of the fundamental class abstractions that have been identified for object-oriented finite element programming was presented. These classes are: **Element**, **Node**, **Constraint**, **Domain**, **ModelBuilder**, **Matrix**, and **Vector**.

In chapter 2, the existing approaches taken for the **Analysis** class were reviewed and their limitations identified. A new design for the finite element analysis was presented which provides greater flexibility and extensibility than the current approaches. The new approach breaks the **Analysis** class into several component classes. An analyst builds an analysis procedure by providing objects of the component classes to the analysis objects constructor. This approach is different than the traditional approach, in which the analyst constructs a single object, the **Analysis** object, to perform the analysis. The new approach offers great flexibility, because the analyst can vary the analysis by changing the types of objects passed to the constructor, and extensibility, because the types of analysis procedures that can be performed is greatly increased by the introduction of a new component subclass. The component classes introduced

are: **Integrator**, **AnalysisAlgorithm**, **ConstraintHandler**, **DOF_Numberer**, and **FE_Model**. The **FE_Model** object is a repository for the **FE_Element** and **DOF_Group** objects. The **FE_Element** and **DOF_Group** classes allow different constraint handling algorithms to be employed in a manner that is transparent to the **Analysis**.

In chapter 2, a different approach to the storing and solving of the system of equations was also introduced. In the traditional approach, a single class is used to both store and solve the system of equations. In this work, two classes are introduced, **SystemOfEqn** and **Solver** classes, which are responsible for storing the equations and solving the equations respectively. This approach allows different equation solving algorithms to be employed in a single analysis. The interface for the **SystemOfEqn** subclasses, i.e. **LinearSOE**, are specified so that different storage schemes can be employed without the **Analysis** object needing to be aware of the specific type of **SystemOfEqn** object.

Finally, the design was extended to allow modal analysis and modal transient analysis. In chapter 3, the design was extended to allow non-overlapping domain decomposition methods. Two new classes, **GraphPartitioner** and **DomainPartitioner**, were introduced for partitioning the domain, and new classes were introduced for the domain decomposition. These new classes for the domain decomposition, all subclasses of existing classes, are: **Subdomain**, **PartitionedDomain**, **DomainDecompAnalysis**, **DomainDecompAlgorithm** and **DomainSolver**.

Empirical results showing the performance of an implementation of the design was presented in the first part of chapter 6. By comparing the performance of the new implementation with that of a specially written procedural program, it was shown that the the CPU requirements of the object-oriented design was comparable with that of the procedural design. However, the object-oriented design required more of the virtual address space than the procedural program, which is an issue for larger problems on limited memory machines. Also, due to the lack of spatial locality in the mapping of the objects to the virtual address space of the object-oriented program, the number of page faults are quite high when page faulting does occur.

Chapter 5 introduced concepts in parallel programming and discussed techniques that have been used to parallelize the finite element method. New classes for parallel finite element programming using the actor programming model were then presented: **Actor**, **Channel**, **Message**, **MovableObject**, **Address**, **MachineBroker**, **ObjectBroker**, and **Shadow**. The **Shadow** class allows the introduction of parallel processing to the design, in a manner which is virtually transparent to the existing classes. The **Shadow** class provides for efficient parallel processing because data can be cached locally in a **Shadow** object, which in certain non-computationally demanding situations allows the local processing of methods by **Shadow** objects on behalf of objects that reside in a remote process. The ability to cache data locally proved to be particularly useful for implementing efficient **ShadowSubdomain** and **ActorSubdomain** classes. This is because when performing dynamic load balancing the **DomainPartioner** object must iterate through all the **Loads** in a **Subdomain** from which an **Element** is being removed. By caching a local copy of these load objects with the **ShadowSubdomain** object the performance of the load balancing was greatly improved.

Results showing the performance of the implementation was presented in the second part of chapter 6 for two NOWs. The results showed that significant performance improvements could be obtained, in one instance five workstations provided almost twenty-eight times the performance of a single workstation.

In chapter 7, dynamic load balancing to improve the performance of a parallel programming was presented. An existing approach to dynamic load balancing for finite element analysis was presented. Two alternative approaches were proposed, one of which is better suited to a NOW using an ethernet for communication between the workstations. A new class, **LoadBalancer**, was presented and the modifications required to some existing classes, **PartitionedDomain**, **Subdomain**, and **DomainPartitioner** was given. Empirical results showed that the performance of the parallel program can be improved when a simple load balancing scheme is employed, in some instances more than doubling the original performance.

## 8.2 Future Directions

This research achieved its main objectives of providing a software design that allows for flexible and extensible finite element analysis to be performed in an efficient way in both sequential and parallel computing environments. In the process of developing and implementing the new design, future research directions have been identified. These include:

1. As the design allows for great flexibility and extensibility it is essential that an interpreted environment, similar to that provided for Matlab, be developed. Such an environment would allow the analyst to develop new classes and perform different analysis without the need to recompile the program for each new class and each different type of analysis procedure.

2. The current design allows the creation of the finite element model in the main process of the parallel program. For very large problems the model itself may exhaust the memory resources of the machines. It is necessary that parallel **ModelBuilder** classes be used. A review of the changes that would be required to allow for efficient parallel model generation is required.

3. A non-linear or transient analysis of a very large problem, whether by an object-oriented program or a procedural program, can generate enormous amounts of history data that the analyst may wish to save. It is inefficient for the **Element** and **Node** objects to store this information. A review of the changes that would be required to allow this information to be stored in sequential and/or parallel databases is required.

4. It is important that the analyst be able to build the finite element models and review the results of the analysis graphically. A review of the new classes that would be required for a graphical user interface is required.

5. It was demonstrated in chapter 6 that the partitioning strategy does not always lead to well balanced partitions, when using the substructuring method for small numbers of subdomains. New partitioning strategies need to be developed for

such situations which take into account not only the number of elements but also the number of external and internal nodes.

6. It was noticed in the experiments that the amount of processing in the subdomains was somewhat sensitive to the **DOF_Numberer** used. The reason for this is that typical numbering schemes do not take into account that, when using the domain decomposition methods, the degrees-of-freedom need to be split into two groups, internal and external, and numbered accordingly. New numbering schemes need to be developed for the substructuring method.

7. Only a few dynamic load balancing schemes were examined in this work. Many other load balancing schemes could be developed and tested.

# Bibliography

Agarwal, T. K., Storaasli, O. O., and Nguyen, D. T. (1994). "A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers," *Computers and Structures*, Vol. 51(No. 5):pp. 503–512.

Agha, G., editor (1984). *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press.

Almasi, G. S. and Gottlieb, A., editors (1989). *Highly Parallel Computing*, Benjamin/Cummings, Redwood CA.

Amdahl, G. M. (1968). "The Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities," In *Proceedings of the American Federation of Information Processing Society, April 18-20, Atlantic City, NJ.*, pp. 483–485.

America, P. (1987). "Inheritance and Subtyping in a Parallel Object-Oriented Language," In *Proceedings of ECOOP'87, volume 276 of Lecture Notes in Computer Science*, pp. 234–242, Springer-Verlag.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., J. Du Croz, A. Greenbaum, S. H., McKenny, A., Ostrouchov, S., and Sorenson, D. (1995). *LAPACK Users' Guide - 2nd ed.*, Society for Industrial and Applied Mathematics, Philadelphia, PA.

Anderson, T., Culler, D., and Patterson, D. (1995). "A Case for NOW (Networks of Workstations)," *IEEE Micro*, Vol. 15(No. 1):pp. 54–64.

Archer, G. (1996). "Object-Oriented Nonlinear Dynamic Finite Element Analysis," PhD thesis, University of California at Berkeley, May.

Baddourah, M. A. and Nguyen, D. T. (1994). "Parallel-Vector Computations for Geometrically Nonlinear Finite Element Analysis," *Computers and Structures*, Vol. 51(No. 6):pp. 785–789.

Balay, S., McInnes, L. C., Gropp, W., and Smith, B. (1995). "PETSc 2.0 Users Manual," *Report No. ANL 95/11*, Argonne National Laboratory, Argonne, Il.

Barak, A. and Shiloh, A. (1985). “A Distributed Load Balancing Policy for a Multicomputer,” *Software - Practice and Experience*, Vol. 15(No. 9):pp. 901–913.

Barragy, E., Carey, G. F., and Van Ge Geijn, R. (1994). “Performance and Scalability of Finite Element Analysis for Distributed Parallel Computation,” *Journal of Parallel and Distributed Computing*, Vol. 21(No. 2):pp. 202–212.

Bathe, K. J. (1996). *Finite Element Procedures*, Prentice-Hall, Englewood Cliffs, NJ.

Baugh, J. W. and Rehak, D. E. (1992). “Data Abstraction in Engineering Software Development,” *Journal of Computing in Civil Engineering*, Vol. 6(No. 3):pp. 282–301.

Baugh Jr., J. W. and Sharma, S. K. (1994). “Evaluation of Distributed Finite Element Algorithms on a Workstation Network,” *Engineering with Computers*, Vol. 10(No. 1):pp. 45–62.

Beguelin, A., Dongarra, J., Geist, A., Manchek, R., Moore, K., and Sunderam, V. (1993). “PVM and HeNCE: Tools for Heterogeneous Network Computing,” In Dongarra, J. and Tourancheau, B., editors, *Enviroments and Tools for Parallel Scientific Computing*, pp. 139–153, North-Holland.

Benner, R. E., Montrey, G. R., Weigand, G. G., and Duff, I. (1987). “Concurrent Multifrontal Methods: Shared Memory, Cache and Frontwidth Issues,” *The International Journal of Supercomputer Applications*, Vol. 1(No. 3):pp. 26–44.

Bernard, S. T. and Simon, H. D. (1994). “A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Meshes,” *Concurrency: Practice and Experience*, Vol. 6(No. 2):pp. 101–117.

Berry, M. W. and Plemmons, R. J. (1987). “Algorithms and Experiments for Structural Mechanics on High-Performance Architectures,” *Computer Methods in Applied Mechanics and Engineering*, Vol. 64(No. 1-3):pp. 1987.

Bershad, B., Ching, D., Lazowska, E., Sanislo, J., and Schwartz, M. (1987). “A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems,” *IEEE Transactions on Software Engineering*, Vol. 13(No. 8):pp. 880–894.

Bershad, B. N., Lazowska, E., and Levy, H. M. (1988). “PRESTO: A System for Object-Orientd Parallel Programming,” *Software: Practice and Experience*, Vol. 18(No. 8):pp. 713–732.

Birrell, A. and Nelson, B. (1984). “Implementing Remote Procedure Calls,” *ACM Transactions on Computer Systems*, Vol. 2(No. 1):pp. 39–59.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., , and Whaley, R. C. (1997). "ScaLAPACK: A Linear Algebra Library for Message-Passing Computers," In *SIAM Conference on Parallel Processing*.

Bui, T. N. and Jones, C. (1993). "A Heuristic for Reducing Fill-In in Sparse Matrix Factorization," In Sincovec, R. F., Keyes, D. E., Leuze, M. P., Potzold, L. R., and Reed, D. A., editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 445–452, SIAM.

Burman, A. (1990). "On Increasing the Throughput in a Computer Network," In Kinzel, G. L. and Rohde, S. M., editors, *Computers in Engineering: Proceedings of the 1990 ASME International Computers in Engineering Conference and Exposition, Sugust 5-9, Boston, MA*, pp. 441–448, ASME.

Cabrera, L., Hunter, E., Karels, M. J., and Mosher, D. A. (1988). "User-Process Communication Performance in Networks of Computers," *IEEE Transactions on Software Engineering*, Vol. 14(No. 1):pp. 38–53.

Calkin, R., Hempel, R., Hoppe, H. C., and Wypior, R. (1994). "Portable Programming with the PARMACS Message-Passing Library," *Parallel Computing*, Vol. 20(No. 4):pp. 615–632.

Cardona, A., Klapka, I., and Geradin, M. (1994). "Design of a New Finite Element Programming Environment," *Engineering Computations*, Vol. 11(No. 4):pp. 365–381.

Carter, W. T., Sham, T. L., and Law, K. H. (1989). "A Parallel Finite Element Method and it's Prototype Implementation on a Hypercube," *Computers and Structures*, Vol. 31(No. 6):pp. 921–934.

Carter, J. B., Bennett, J. K., and Zwaenepoel, W. (1995). "Techniques for Reducing Consistency and Communication in Distributed Shared Memory Systems," *ACM Transactions on Computer Systems*, Vol. 13(No. 3):pp. 205–243.

Chakrabarti, S., Demmel, J., and Yelick, K., editors (1997). *Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs*, to apppear in: Journal of Parallel and Distributed Computing.

Chandra, R., Gupta, A., and Hennessy, J. L. (1993). "Data Locality and Load Balancing in COOL," *ACM SIGPLAN Notices*, Vol. 28(No. 7):pp. 249–259.

Chandy, K. M. and Kesselman, C. (1993). "CC++: A Declerative Concurrent Object-Orientd Programming Notation," In Agha, G., Wegner, P., and Yonezawa, A., editors, *Research Directions on Concurrent Object-Oriented Programming*, pp. 281–313, MIT Press.

Chien, A. A. and Dally, W. J. (1990). "Concurrent Aggregates (CA)," *ACM SIGPLAN Notices*, Vol. 25(No. 3):pp. 187–196.

Chien, Y. P., Ecer, A., Akay, H. V., Carpenter, F., and Blach, R. A. (1994). "Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems," *Computer Methods in Applied Mechanics and Engineering*, Vol. 119(No. 1-2):pp. 17–33.

Chopra, A. K. (1995). *Dynamics of Structures : Theory and Applications to Earthquake Engineering*, Prentice-Hall, Englewood Cliffs, NJ.

Chowdhury, S. (1990). "The Greedy Load Sharing Algorithm," *Journal of Parallel and Distributed Computing*, Vol. 9(No. 1):pp. 93–99.

Chudoba, R. and Bittnar, Z. (1995). "Explicit Finite Element Computation: An Object-Oriented Approach," In Pahl, P. J. and Werner, H., editors, *Computing in Civil and Building Engineering: Proceedings of the Sixth International Conference on Computing in Civil and Building Engineering, Berlin, Germany, July 12-15 1995*, pp. 139–145, A. A. Balkema, Brrokfield, VT 05036.

Cleary, A. and Dongarra, J. (1997). "Implementation in ScaLAPACK of Divide-and-Conquer Algorithms for Banded and Tridiagonal Linear Systems," *Report No. CS-97-358*, University of Tennessee, Knoxville, TN.

Codenotti, B. and Leoncini, M., editors (1993). *Introduction to Parallel Processing*, Addison-Wesley.

Culler, D., Dusseau, A., Goldstein, S., Krishnamurthy, A., Lumetta, S., von Eiken, T., and Yelik, K. (1993). "Parallel Programs in Split-C," In Werner, P., editor, *Supercomputing'93, Nov 15-19, Portland, Oregon.*, IEEE Computer Society Press.

Culler, D., Singh, J. P., and Gupta, A., editors (1997). *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers.

Cybenko, G. (1989). "Dynamic Load Balancing for Distributed Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 7(No. 2):pp. 279–301.

Dongarra, J. J. and Johnsson, L. (1987). "Solving BAnded Systems on a Parallel Processor," *Parallel Computing*, Vol. 5(No. 1-2):pp. 219–246.

Dongarra, J., Pozo, R., and Walker, D. (1995). "Lapack++ v1.0: High Performance Linear Algebra Users' Guide," *Report No. CS-95-290*, University of Tennessee, Knoxville, TN.

Douglis, F. and Ousterhout, J. (1989). "Transparent Process Migration for Personal Workstations," *Report No. UCB/CSD 89/540*, Computer Science Division (EECS), University of California, Berkeley, CA 94720, November.

Dubois-Pelerin, Y. and Zimmermann, T. (1993). "Object-Oriented Finite Element Programming: III. An Efficient Implementation in C++," *Computer Methods in Applied Mechanics and Engineering*, Vol. 108(No. 1-2):pp. 165–183.

Dubois-Pelerin, Y., Zimmermann, T., and Bomme, P. (1992). "Object-Oriented Finite Element Programming: II. A Prototype Program in Smalltalk," *Computer Methods in Applied Mechanics and Engineering*, Vol. 98(No. 3):pp. 361–397.

Duff, I. S. and Reid, J. K. (1983). "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations," *ACM Transactions on Mathematical Software*, Vol. 9(No. 3):pp. 302–325.

Duff, I. S. and Reid, J. K. (1986). "Parallel Implementation of Multifrontal Schemes," *Parallel Computing*, Vol. 3(No. 2):pp. 193–204.

Eager, D. L., Lazowska, E. D., and Zahorjan, J. (1986). "Adaptive Load in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12(No. 5):pp. 662–675.

El-Sayed, M. E. M. and Hsiung, C. K. (1990). "Parallel Finite Element Computation with Separate Substructures," *Computers and Structures*, Vol. 36(No. 2):pp. 261–265.

Farhat, C. and Crivelli, L. (1994). "A Transient FETI Methodology for Large-Scale Parallel Implicit Computations in Structural Mechanics," *International Journal for Numerical Methods in Engineering*, Vol. 37(No. 11):pp. 1945–1975.

Farhat, C. and Roux, F. X. (1991). "A Method of Finite Element Tearing and Interconnecting and its Parallel Solution Algorithm," *International Journal for Numerical Methods in Engineering*, Vol. 32(No. 6):pp. 1205–1227.

Farhat, C. and Roux, F. (1994). "Implicit Parallel Processing in Structural Mechanics," *Computational Mechanics Advances*, Vol. 2(No. 1):pp. 1–124.

Farhat, C. and Wilson, E. (1986). "Modal Superposition Dynamic Analysis on Concurrent Multiprocessors," *Engineering Computations*, Vol. 3(No. 4):pp. 305–311.

Farhat, C. and Wilson, E. (1988). "A Parallel Active Column Equation Solver," *Computers and Structures*, Vol. 28(No. 2):pp. 289–304.

Farhat, C., Wilson, E., and Powell, G. (1987). "Solution of Finite Element Systems on Concurrent Processing Computers," *Engineering with Computers*, Vol. 2(No. 3):pp. 157–165.

Farhat, C., Pramono, E., and Felippa, C. (1989). "Torwards Parallel I/O in Finite Element Simulations," *International Journal for Numerical Methods in Engineering*, Vol. 28(No. 11):pp. 2541–2553.

Farhat, C. (1988). "A Simple and Efficient Automatic FEM Domain Decomposer," *Computers and Structures*, Vol. 28(No. 5):pp. 579–602.

Farhat, C. (1990). "Redesigning the Skyline Solver for Parallel/Vector Supercomputers," *The International Journal for High Speed Computations*, 2:pp. 223–238.

Farhat, C. (1990). "Which Parallel Finite Element Algorithm for which Architecture and which Problem?," *Engineering Computations*, Vol. 7(No. 3):pp. 186–195.

Feldman, J. A., Lim, C., and Rauber, T. (1993). "The Shared Memory Language pSather on a Distributed Memory Multiprocessor," *ACM SIGPLAN Notices*, Vol. 28(No. 1):pp. 17–20.

Fenves, G. L. (1990). "Object-Oriented Programming for Engineering Software Development," *Engineering with Computers*, Vol. 6(No. 1):pp. 1–15.

Flower, J., Otto, S., and Salama, M. (1987). "Optimal Mapping of Irregular Finite Element Domains to Parallel Processors," In Noor, A. K., editor, *Parallel Computations and their Impact on Mechanics*, pp. 239–250, The American Society of Mechanical Engineers.

Foley, C. M. and Vinnakota, S. (1994). "Parallel Processing in the Elastic Nonlinear Analysis of High-Rise Frameworks," *Computers and Structures*, Vol. 52(No. 6):pp. 1169–1179.

Forde, B. W. R., Foschi, R. O., and Steimer, S. F. (1990). "Object-Oriented Finite Element Analysis," *Computers and Structures*, Vol. 34(No. 3):pp. 355–374.

Fulton, R. F. and Su, R. S. (1992). *Parallel Substructure Approach for Massively Parallel Computers*, ASME.

George, J. A. and Liu, J. W. H. (1978). "An Automatic Nested Dissection Algorithm for Irregular Finite Element Problems," *SIAM Journal on Numerical Analysis*, Vol. 15(No. 5):pp. 1053–1069.

George, J. A. (1971). "Computer Implementation of the Finite Element Method," PhD thesis, Stanford University, May.

George, J. A. (1973). "Nested Dissection of a Regular Finite Element Mesh," *SIAM Journal on Numerical Analysis*, Vol. 10(No. 2):pp. 345–363.

Goehlich, D., Komzsik, L., and Fulton, R. E. (1989). "Applications of a Parallel Equation Solver to Static FEM Problems," *Computers and Structures*, Vol. 31(No. 2):pp. 121–129.

Golub, G. H. and VanLoan, C. F. (1989). *Matrix Computations*, The John Hopkins University Press, Baltomre, Maryland.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A., editors (1996). *A High Performance, Portable Implementation of the MPI Message Passing Interface Standard*, URL: http://www.mcs.anl.gov/mpi/mpich/mpicharticle.ps.

Gupta, A. and Kumar, V. (1994). "A Scalable Parallel Algorithm for Sparse Cholesky Factorization," In Werner, B., editor, *Supercomputing'94, Nov 14-18, 1994, Washington, D.C.*, pp. 793–802, IEEE Computer Society Press.

Hajjar, J. F. and Abel, J. F. (1988). "Parallel Processing for Transient Nonlinear Structural Dynamics of Three-Dimensional Framed Structures using Domain Decomposition," *Computers and Structures*, Vol. 30(No. 6):pp. 1237–1254.

Hoffmeister, P., Zahlten, W., and Kratzig, W. B. (1993). "Object-Oriented Finite Element Modeling," In Cohn, L. F., editor, *Computing in Civil and Building Engineering: proceedings of the fifth International Conference V_ICCCBE, Anaheim, CA, June 7-9*, pp. 537–544, ASCE, New York, NY 10017.

HPF Forum (1993). "High Performance Fortran Language Specification, version 1,0," *Report No. CRPC-TR92225*, Rice University.

Hsieh, S. H. and Sotelino, E. D. (1997). "A Message-Passing Class Library C++ for Portable Parallel Programming," *Engineering with Computers*, Vol. 13(No. 1):pp. 20–34.

Hughes, T. J. R., Ferencz, R. M., and Hallquist, J. O. (1987). "Large-Scale Vectorized Implicit Calculations in Solid Mechanics an a Cray X-MP/48 Utilizing EBE Preconditioned Conjugate Gradients," *Computer Methods in Applied Mechanics and Engineering*, Vol. 61(No. 2):pp. 215–248.

Hughes, T. J. R. (1987). *The Finite Element Method*, Prentice-Hall, Englewood Cliffs, NJ.

Kafura, D. G. and Lee, K. H. (1989). "Inheritance in Actor Based Concurrent Object-Oriented Languages," In *Proceedings of ECOOP'89*, pp. 131–145, Cambridge University Press.

Kale, L. V. and Krishnan, S. (1993). "CHARM++: A Portable Concurrent Object Oriented System based on C++," *ACM SIGPLAN Notices*, Vol. 28(No. 10):pp. 91–108.

Kamal, O. and Adeli, H. (1990). "Automatic Partitioning of Frame Structures for Concurrent Processing," *Microcomputers in Civil Engineering*, Vol. 5(No. 4):pp. 269–283.

Karypis, G. and Kumar, V. (1995). "METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System Version 2.0," *anonymous ftp to ftp.cs.sandia.gov in the file pub/papers/bahendr/guide.ps.Z.*

Karypis, G. and Kumar, V. (1995). "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Report No. 96-064*, Univiversity of Minnesota, Deptartment of Computer Science, Minneapolis, MN.

Kernighan, B. and Lin, S. (1970). "An effective heuristic procedure for partitioning graphs," *The Bell System Technial Journal*, pp. 291–308.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). "Optimization by Simulated Annealing," *Science*, 220:pp. 671–680.

Kumar, S. and Adeli, H. (1995). "Distributed Finite-Element Analysis on Network of Workstations - Implementation and Application," *Journal of Structural Engineering*, Vol. 121(No. 10):pp. 1456–1462.

Kumar, V., Grama, A., Gupta, A., and Karypis, G., editors (1994). *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings Publishing Company, Redwood, CA.

Lai, G. and Chen, H. (1992). "Parallelization of Linear Finite Element Analysis," In Goodno, B. J. and Wright, J. R., editors, *Computing in Civil Engineering, Eighth Conference held in conjunction with A/E/C Systems '92, Dallas, Texas, June 7-9*, pp. 655–662, ASCE.

Law, K. H. and Mackay, D. R. (1993). "A Parallel Row-Oriented Sparse Solution Method for Finite Element Structural Analysis," *International Journal for Numerical Methods in Engineering*, Vol. 36(No. 17):pp. 2895–2919.

Law, K. H. (1986). "A Parallel Finite Element Solution Method," *Computers and Structures*, Vol. 23(No. 6):pp. 845–858.

Lee, J. K. and Gannon, D. (1991). "Object Oriented Parallel Programming Experiments and Results," In Copeland, A., editor, *Supercomputing'91, Nov 18-22, Albuquerque, New Mexico*, pp. 273–282, IEEE Computer Society Press.

Li, K. and Hudak, P. (1989). "Memory Coherence in Shared Virtual Memory Machines," *ACM Transactions on Computer Systems*, Vol. 7(No. 4):pp. 321–359.

Li, X. S. (1996). "Sparse Gaussian Elimination on High Performance Computers," PhD thesis, University of California at Berkeley, September.

Lin, F. C. H. and Keller, R. M. (1987). "The Gradient Model Load Balancing Method," *IEEE Transactions on Software Engineering*, Vol. SE-13(No. 1):pp. 32–38.

Lu, J., White, D. W., and Chen, W. F. (1993). "Applying Object-Oriented Design to Finite Element Programming," In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing, Indianapolis, Indiana*, pp. 424–492, ACM.

Lu, J., White, D. W., Chen, W. F., and Dunsmore, H. E. (1995). "A Matrix Class Library in C++ for Structural Engineering Computing," *Computers and Structures*, Vol. 55(No. 1):pp. 95–111.

Mackerle, J. (1996). "Implementing Finite Element Methods on Supercomputers, Workstations and PCs: A Bibliography," *Engineering Computations*, Vol. 13(No. 1):pp. 33–85.

Mackie, R. J. (1992). "Object-Oriented Programming of the Finite Element Method," *International Journal for Numerical Methods in Engineering*, Vol. 35(No. 2):pp. 425–436.

Mackie, R. I. (1995). "Object-Oriented Methods - Finite Element Programming and Engineering Software Design," In Pahl, P. J. and Werner, H., editors, *Computing in Civil and Building Engineering: Proceedings of the Sixth International Conference on Computing in Civil and Building Engineering, Berlin, Germany, July 12-15 1995*, pp. 133–138, A. A. Balkema, Brrokfield, VT 05036.

Malone, J. G. (1988). "Automated Mesh Decomposition and Concurrent Finite Element Analysis for Hypercube Multiprocessor Computers," *Compute Methods in Applied Mechanics and Engineering*, Vol. 70(No. 1):pp. 27–58.

Menetrey, P. and Zimmermann, T. (1993). "Object-Oriented Non-Linear Finite Element Analysis: Application to J2 Plasticity," *Computers and Structures*, Vol. 49(No. 5):pp. 767–777.

Miller, G. R. and Rucki, M. D. (1993). "A Program Architecture for Interactive Nonlinear Dynamic Analysis of Structures," In Cohn, L. F., editor, *Computing in Civil and Building Engineering: Proceedings of the Fifth International Conference V_ICCCBE, Anaheim , CA, June 7-9*, pp. 529–536, ASCE, New York, NY 10017.

Miller, G. R. (1991). "An Object-Oriented Approach to Structural Analysis and Design," *Computers and Structures*, Vol. 40(No. 1):pp. 75–82.

Modak, S., Sotelino, E. D., and Hsieh, S. H. (1997). "A Parallel Matrix Class Library in C++ for Computational Mechanics Applications," *Microcomputers in Civil Engineering*, Vol. 12(No. 1):pp. 83–99.

Mukunda, G. R., Sotelino, E. D., and Hsieh, S. H. (1996). "An Object-Oriented Finite Element Analysis Framework," *Report No. CE-STD-96-4*, Civil Engineering Purdue University, West Lafayette, IN.

Nevin, N., editor (1996). *The Performance of LAM 6,0 and MPICH 1.0.12 on a Workstation Cluster*, URL: http://www.osc.edu/Lam/lam/lam60-perf.html.

Ni, L. M., Xu, C. W., and Gendreau, T. B. (1985). "A Distributed Drafting Algorithm for Load Balancing," *IEEE Transactions on Software Engineering*, Vol. SE-11(No. 10):pp. 1153–1161.

Nicol, D. M. and Reynolds Jr., P. F. (1990). "Optimal Dynamic Remapping of Data Parallel Computations," *IEEE Transactions on Computers*, Vol. 39(No. 2):pp. 206–219.

Nour-Omid, B. and Park, K. C. (1987). "Solving Structural Mechanics Problems on the Caltech Hypercube Machine," *Computer Methods in Applied Mechanics and Engineering*, Vol. 61(No. 2):pp. 161–176.

Ortiz, M. and Nour-Omid, B. (1986). "Unconditionally Stable Concurrent Procedures for Transient Finite Element Analysis," *Computer Methods in Applied Mechanics and Engineering*, Vol. 58(No. 2):pp. 151–174.

Ostermann, W., Wunderlich, W., and Cramer, H. (1995). "Object-Oriented Tools for the Development of User Interfaces for Interactive Teachware," In Pahl, P. J. and Werner, H., editors, *Computing in Civil and Building Engineering: Proceedings of the Sixth International Conference on Computing in Civil and Building Engineering, Berlin, Germany, July 12-15 1995*, pp. 139–145, A. A. Balkema, Brrokfield, VT 05036.

Ou, R. and Fulton, R. E. (1988). "An Investigation of Parallel Numerical Integration Methods for Nonlinear Dynamics," *Computers and Structures*, Vol. 30(No. 1-2):pp. 403–409.

Parkes, S., Chandy, J. A., and Banerjee, P. (1994). "A Library Based Approach to Portable, Parallel, Object-Oriented Programming Interface, Implementation and Application," In Werner, B., editor, *Supercomputing'94, Nov 14-18, 1994, Washington, D.C.*, pp. 69–78, IEEE Computer Society Press.

Pidaparti, R. M. V. and Hudl, A. V. (1993). "Dynamic Analysis of Structures using Object-Oriented Techniques," *Computers and Structures*, Vol. 49(No. 1):pp. 149–156.

Pramono, E., Simon, H. D., and Sohn, A. (1994). "Dynamic Load Balancing for Finite Element Calculations on Parallel Computers," In Bailey, D. H., Bjorstad, P. E., Gilbert, J. P., Mascagni, M. V., Schreiber, R. S., Simon, H. D., Torczon, J. T., and Watson, L. T., editors, *Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pp. 599–604, SIAM.

Quinn, M. J., editor (1994). *Parallel Computing: Theory and Practice*, McGraw-Hill, Inc.

Raphael, B. and Krishnamoorthy, C. S. (1993). "Automatic Finite Element Development using Object Oriented Techniques," *Engineering Computations*, Vol. 10(No. 3):pp. 267–278.

Rihaczek, C. and Kroplin, B. (1993). "Object-Oriented Finite Element Modeling," In Cohn, L. F., editor, *Computing in Civil and Building Engineering: proceedings of the fifth International Conference V ICCCBE, Anaheim , CA, June 7-9*, pp. 545–552, ASCE, New York, NY 10017.

Roa, M., Logarathan, K., and Raman, N. V. (1994). "Multifrontal Based Approach for Concurrent Finite Element Analysis," *Computers and Structures*, Vol. 52(No. 4):pp. 841–846.

Ross, T. J., Wagner, L. R., and Luger, G. F. (1992). "Object-Oriented Programming for Scientific Codes. II: Examples in C++," *Journal of Computing in Civil Engineering*, Vol. 6(No. 4):pp. 497–514.

Rothberg, E. and Gupta, A. (1993). "An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization," In Werner, P., editor, *Supercomputing'93, Nov 15-19, Portland, Oregon*, pp. 503–512, IEEE Computer Society Press.

Rothberg, E. and Schreiber, R. (1994). "Improved Load Distribution in Parallel Sparse Cholesky Factorization," In Werner, B., editor, *Supercomputing'94, Nov 14-18, 1994, Washington, D.C.*, pp. 783–792, IEEE Computer Society Press.

Rucki, M. D. and Miller, G. R. (1996). "An Algorithmic Framework for Flexible Finite Element Element-Based Structural Modeling," *Computer Methods in Applied Mechanics and Engineering*, Vol. 136(No. 3-4):pp. 363–384.

Rucki, M. D. (1996). "An Algorithmic Framework for Flexible Finite Element Modelling," PhD thesis, University of Washington, May.

Rumbaugh, J., Blaha, M., Premerhani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey 07632.

Saab, Y. G. and Rao, V. B. (1991). "Combinatorial Optimization by Stochastic Evolution," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 10(No. 4):pp. 525–535.

Santiago, E. D. and Law, K. H. (1996). "An Implementation of Finite Element Method on Distributed Workstations," In Cheng, F. Y., editor, *Analysis and Computation: proceedings of the twelfth conference held in conjunction with Structures Congress XIV, Chicago, Illinois, April 15-18*, pp. 188–199, ASCE.

Sause, R. and Song, J. (1994). "Object-Oriented Structural Analysis with Substructures," In Khozeimeh, K., editor, *Computing in Civil Engineering: Proceedings of the First Conference held in Conjunction with A/E/C Systems '94, Washington, D.C., June 20-22, 1994*, pp. 153–160, ASCE.

Schloegel, A., Karypis, G., and Kumar, V. (1997). *ParMetis: Parallel Library for Unstructured Meshes (Re)Partitioning and Sparse Matrix Ordering*, URL: http://www-users.cs.umn.edu/ karypis/talks/parmetis/index.htm.

Scholz, S. P. (1992). "Elements of an Object-Oriented FEM++ Program in C++," *Computers and Structures*, Vol. 43(No. 3):pp. 517–529.

Sharma, S. K. and Baugh Jr., J. W. (1992). "LAN Ho! Structural Analysis on a Network," In Goodno, B. J. and Wright, J. R., editors, *Computing in Civil Engineering, Eighth Conference held in conjunction with A/E/C Systems '92, Dallas, Texas, June 7-9*, pp. 639–646, ASCE.

Simon, H. D. (1991). "Partitioning of Unstructured Problems for Parallel Problems," *Computing Systems in Engineering*, Vol. 2(No. 2/3):pp. 135–148.

Stevens, W. R., editor (1990). *Unix Network Programming*, Prentice Hall, Inglewood Cliffs, New Jersey.

Stroustrup, B. (1991). *The C++ Programming Language*, Addison-Wesley.

Sunderam, V. S., Geist, G. A., and Dongarra, J. J. (1994). "The PVM Concurrent Computing System: Evolution, Experiences and Trends," *Parallel Computing*, Vol. 20(No. 4):pp. 531–546.

Sunderam, V. S. (1990). "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, Vol. 2(No. 4):pp. 315–339.

Synn, S. Y. and Fulton, R. E. (1995). "Practical Strategy for Concurrent Substructure Analysis," *Computers and Structures*, Vol. 54(No. 5):pp. 939–944.

Taylor, V. E. and Nour-Omid, B. A. (1994). "A Study of the Factorization Fill-In for a Parallel Implementation of the Finite Element Method," *International Journal for Numerical Methods in Engineering*, Vol. 37(No. 22):pp. 3809–3823.

Tinney, W. F. and Walker, J. W. (1967). "Direct Solution of Sparse Network Equations by Optimally Ordered Triangular Factorization," *Proc. IEEE*, Vol. 55(No. 11):pp. 1801–1809.

Van Driessche, R. and Roose, D. (1995). "An Improved Spectral Bisection Algorithm and its Application to Dynamic Load Balancing," *Parallel Computing*, Vol. 21(No. 1):pp. 29–48.

Vanderstraeten, D. and Keunings, R. (1995). “Optimized Partitioning of Unstructured Finite Element Meshes,” *International Journal for Numerical Methods in Engineering*, Vol. 38(No. 3):pp. 433–450.

Walker, D. W. (1994). “The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers,” *Parallel Computing*, Vol. 20(No. 4):pp. 657–673.

Wang, Y. T. and Morris, R. J. T. (1985). “Load Sharing in Distributed Systems,” *IEEE Transactions on Computers*, Vol. C-34(No. 3):pp. 204–217.

Williams, R. D. (1991). “Performance of Dynamic Load Balancing Algorithms for Unstructured Meshes,” *Concurrency: Practice and Experience*, Vol. 3(No. 5):pp. 457–481.

Xu, J. and Hwang, K. (1993). “Heuristic Methods for Dynamic Load Balancing in a Messgage-Passing Multicomputer,” *Journal of Parallel and Distributed Computing*, Vol. 18(No. 1):pp. 1–13.

Y. DeRoeck, P. L. and Vidrascu, M. (1992). “A Domain Decomposed Solver for Nonlinear Elasticity,” *Computer Methods in Applied Mechanics and Engineering*, Vol. 99(No. 2-3):pp. 187–207.

Yu, G. and Adeli, H. (1993). “Object-Oriented Finite Element Analysis using EER Model,” *Journal of Structural Engineering*, Vol. 119(No. 9):pp. 2763–2781.

Zahlten, W., Demmert, P., and Kratzig, W. B. (1995). “An Object-Oriented Approach to Physically Nonlinear Problems in Computational Mechanics,” In Pahl, P. J. and Werner, H., editors, *Computing in Civil and Building Engineering: Proceedings of the Sixth International Conference on Computing in Civil and Building Engineering, Berlin, Germany, July 12-15 1995*, pp. 139–145, A. A. Balkema, Brrokfield, VT 05036.

Zeglinski, G. W. and Han, R. P. S. (1994). “Object Oriented Matrix Classes for Use in a Finite Element Code using C++,” *International Journal for Numerical Methods in Engineering*, Vol. 37(No. 22):pp. 3921–3937.

Zhang, W. and Lui, E. M. (1991). “A Parallel Frontal Solver on the Alliant FX/80,” *Computers and Structures*, Vol. 38(No. 2):pp. 203–215.

Zienkiewicz, O. C. and Taylor, R. L. (1989). *The Finite Element Method - 4thEd.*, McGraw-Hill, London.

Zimmermann, T., Dubois-Pelerin, Y., and Bomme, P. (1992). “Object-Oriented Finite Element Programming: I. Governing Principles,” *Computer Methods in Applied Mechanics and Engineering*, Vol. 98(No. 2):pp. 291–303.

# Appendix A

# Matrix, Vector and ID Classes

In this appendix the **Matrix**, **Vector** and **ID** classes are presented. **Matrix**, **Vector** and **ID** objects are primarily used in the design to pass information, e.g. stiffness and load information, between objects. The **Matrix** and **Vector** classes also provide a full range of numerical functions, typically in the form of overloaded operator functions. These functions were found to be useful in the primary stages of element development, however, for efficiency reasons many of the element methods that used these functions were later rewritten.

## A.1 Matrix Class

The **Matrix** class, whose interface is showing in figure A.1, provides methods to obtain information about the size of the **Matrix**, to zero out the **Matrix**, to assemble a **Matrix** into the **Matrix**, and to set and retrieve components of the **Matrix**. In addition many of the operator functions are overloaded. The overloaded functions can be split into two groups: those that change the original **Matrix** ($=$, $+=$, $-=$, $*=$, $/=$), an those that return a new **Matrix** ($+$, $-$, $*$, $/$). The latter functions require that a Matrix constructor be called twice and so are expensive to use in large problems. This is because a constructor must first be called inside the function to create a **Matrix** into which to store the results of the operation, and a constructor is called again when this **Matrix** is returned. A reference to the new **Matrix** created inside the function cannot be returned because a destructor cannot be called.

```
class Matrix {
    public:
        // constructors and destructors
        Matrix(); // for subclasses
        Matrix(int nRows, int nCols);
        Matrix(int nRows, int nCols, double *theData);
        Matrix(const Matrix &M);
        virtual ~Matrix();

        // utility methods
        virtual int noRows() const;
        virtual int noCols() const;
        virtual void Zero(void);
        virtual Vector Solve(const Vector &V);
        virtual Matrix &Assemble(const Matrix &,
                   const ID &rows, const ID &cols, double fact = 1.0);
        friend ostream &operator<<(ostream &s, const Matrix &V);
        friend istream &operator>>(istream &s, Matrix &V);

        // operator overloaded functions
        virtual double &operator()(int row, int col)
        virtual double &operator()(int row, int col) const
        virtual Matrix operator()(const ID &rows, const ID & cols) const;
        virtual Matrix &operator=(double value);
        virtual Matrix &operator=(const Matrix &M);
        virtual Matrix &operator+=(double fact);
        virtual Matrix &operator-=(double fact);
        virtual Matrix &operator*=(double fact);
        virtual Matrix &operator/=(double fact);
        virtual Matrix &operator*=(const Matrix &other);
        virtual Matrix &operator+=(const Matrix &other);
        virtual Matrix &operator-=(const Matrix &other);

        virtual Matrix operator+(double fact) const;
        virtual Matrix operator-(double fact) const;
        virtual Matrix operator*(double fact) const;
        virtual Matrix operator/(double fact) const;
        virtual Vector operator*(const Vector &V) const;
        virtual Vector operator^(const Vector &V) const; // ^ used for transpose
        virtual Matrix operator+(const Matrix &M) const;
        virtual Matrix operator-(const Matrix &M) const;
        virtual Matrix operator*(const Matrix &M) const;
        virtual Matrix operator/(const Matrix &M) const;
        virtual Matrix operator^(const Matrix &M) const;
};
```

Figure A.1: Interface of the **Matrix** Class

## A.2 Vector Class

The **Vector** class, whose interface is shown in figure A.2, provides methods to obtain information about the size of the **Vector**, to zero out the **Vector**, to assemble a **Vector** into the **Vector**, to obtain the norm of the **Vector**, and to set and retrieve components of the **Vector**. In addition many of the operator functions are overloaded. The overloaded functions can be split into two groups: those that change the original **Vector** (=,+=,-=,*=,/=), and those that return a new **Vector** (+,-,*,/). The latter functions should be avoided because they require that a **Vector** constructor be called twice and are therefore expensive to use in large problems.

## A.3 ID Class

The **ID** class, whose interface is shown in figure A.3, provides controlled access to integer arrays. Methods are provided to obtain the size of the array, to zero out the array, and to set and retrieve components of the array. Methods are also provided to see if an integer is in the array and to remove an integer value from the array.

```
class Vector {
    public:
        // constructors and destructors
        Vector(); // for subclasses
        Vector(int size);
        Vector(const Vector &);
        Vector(double *data, int size);
        ~Vector();

        // utility methods
        virtual double Norm(void) const;
        virtual int Size(void) const;
        virtual void Zero(void);
        virtual void Assemble(const Vector &V, const ID &I, double fact = 1.0);
        virtual void addVector(const Vector &other, double fact = 1.0);
        virtual void addMatrixVector(const Matrix &m, Vector &v, double fact = 1.0);
        friend ostream &operator<<(ostream &s, const Matrix &V);
        friend istream &operator>>(istream &s, Matrix &V);

        // operator overloaded functions
        virtual double &operator()(int x);
        virtual double &operator()(int x);
        virtual Vector operator()(const ID &rows) const;
        virtual Vector &operator=(const Vector &V);

        virtual Vector &operator+=(const Vector &V);
        virtual Vector &operator-=(const Vector &V);
        virtual Vector &operator+=(double fact);
        virtual Vector &operator-=(double fact);
        virtual Vector &operator*=(double fact);
        virtual Vector &operator/=(double fact);

        virtual Vector operator+(const Vector &V) const;
        virtual Vector operator-(const Vector &V) const;
        virtual Vector operator+(double fact) const;
        virtual Vector operator-(double fact) const;
        virtual Vector operator*(double fact) const;
        virtual Vector operator/(double fact) const;
        virtual Vector operator/(const Matrix &M) const;
        virtual double operator^(const Vector &V) const;
        virtual Matrix operator^(const Matrix &M) const;
};
```

Figure A.2: Interface of the **Vector** Class

```
class ID {
    public:
        // constructors and destructors
        ID(); // for subclasses
        ID(int size);
        ID(int size, int *theData);
        ID(const ID &);
        virtual ~ID();

        // utility methods
        virtual int Size(void) const;
        virtual void Zero(void);
        virtual int getLocation(int value);
        virtual int removeValue(int value);
        virtual friend ostream &operator<<(ostream &s, const ID &V);
        virtual friend istream &operator>>(istream &s, ID &V);

        // operator overloaded functions
        virtual int operator()(int x);
        virtual int operator[](int x); // if x > size, makes larger
        virtual int operator()(int x) const;
        virtual ID &operator=(const ID &I);
};
```

Figure A.3: Interface of the **ID** Class

# Appendix B

# Detailed Performance Measurements

In this appendix detailed information about the performance presented in chapter 6 is given.

## B.1 Sequential Performance

In this section tables are presented profiling the CPU time and number of page faults for the sequential program presented in section 6.3.1. Tables B.1 through B.3 give the profile information showing what percentage of the CPU time was spent in the main components of the domainChanged(), solveCurrentStep(), and update() method calls, which are the methods invoked when analyze() is invoked on a **StaticAnalysis** object. Similar profile information for the percentage of page faults on the limited memory ALPHA and DEC machines are shown in tables B.4 and B.5.

## B.2 Parallel Performance

Tables B.6 through B.21 provide detailed information on the performance of the parallel program presented in section 6.4.1. In the tables the time taken to perform the analysis is broken into two: the time taken to partition the model, and the time taken to perform the analysis once the partitioning has been performed. For each,

the CPU time and number of page faults are also provided. In addition, for each subdomain information showing the size of the problem and the cost in terms of real time, CPU time and number of page faults, to perform the subdomain computations are also provided.

| | Example | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2dF1 | 2dF2 | 2dF3 | 2dF4 | 2dF5 | 2dF6 | 3dF3 | 3dF4 | 3dF5 | 3dF6 |
| **domainChanged()** | **16** | **11** | **9** | **9** | **9** | **9** | **6** | **5** | **5** | **6** |
| handle() | 6 | 4 | 4 | 4 | 4 | 4 | 1 | 1 | 1 | 1 |
| numberDOF() | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| getDOFGraph() | 7 | 5 | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 3 |
| setSize() | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **applyLoad()** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **solveCurrentStep()** | **82** | **87** | **90** | **90** | **90** | **90** | **94** | **94** | **94** | **94** |
| formNodalUnbalance() | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| formElementResidual() | 5 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| formTangent() | 7 | 5 | 4 | 3 | 4 | 4 | 2 | 2 | 2 | 2 |
| solve() | 69 | 79 | 84 | 84 | 84 | 84 | 91 | 91 | 91 | 91 |
| update() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **commit()** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

Table B.1: % CPU time on HOLDEN for C++ Program

| | Example | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2dF1 | 2dF2 | 2dF3 | 2dF4 | 2dF5 | 2dF6 | 3dF3 | 3dF4 | 3dF5 | 3dF6 |
| **domainChanged()** | **18** | **14** | **10** | **10** | **10** | **10** | **6** | **6** | **6** | **6** |
| handle() | 8 | 7 | 5 | 5 | 5 | 5 | 2 | 2 | 2 | 2 |
| numberDOF() | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| getDOFGraph() | 8 | 5 | 4 | 4 | 4 | 4 | 3 | 3 | 3 | 3 |
| setSize() | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **applyLoad()** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **solveCurrentStep()** | **80** | **86** | **89** | **90** | **90** | **89** | **94** | **94** | **93** | **93** |
| formNodalUnbalance() | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| formElementResidual() | 6 | 4 | 3 | 3 | 3 | 2 | 1 | 1 | 1 | 1 |
| formTangent() | 7 | 5 | 4 | 6 | 4 | 4 | 3 | 2 | 3 | 3 |
| solve() | 66 | 76 | 81 | 82 | 82 | 82 | 90 | 89 | 89 | 89 |
| update() | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **commit()** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

Table B.2: % CPU time on ALPHA for C++ Program

| | Example | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2dF1 | 2dF2 | 2dF3 | 2dF4 | 2dF5 | 2dF6 | 3dF3 | 3dF4 |
| **domainChanged()** | **14** | **8** | **5** | **5** | **5** | **5** | **3** | **3** |
| handle() | 5 | 3 | 1 | 2 | 2 | 2 | 1 | 1 |
| numberDOF() | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| getDOFGraph() | 6 | 3 | 2 | 2 | 2 | 2 | 2 | 2 |
| setSize() | 2 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| **applyLoad()** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **solveCurrentStep()** | **85** | **92** | **95** | **95** | **95** | **95** | **97** | **97** |
| formNodalUnbalance() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| formElementResidual() | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| formTangent() | 5 | 3 | 2 | 2 | 2 | 2 | 1 | 1 |
| solve() | 77 | 87 | 92 | 91 | 91 | 91 | 95 | 95 |
| update() | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **commit()** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

Table B.3: % CPU time on DEC for C++ Program

| | Example | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2dF1 | 2dF2 | 2dF3 | 2dF4 | 2dF5 | 2dF6 | 3dF3 | 3dF4 | 3dF5 | 3dF6 |
| **domainChanged()** | **-** | **-** | **1** | **0** | **0** | **0** | **1** | **1** | **0** | **0** |
| handle() | - | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| numberDOF() | - | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| getDOFGraph() | - | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| setSize() | - | - | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| **applyLoad()** | **-** | **-** | **1** | **4** | **3** | **6** | **3** | **2** | **3** | **2** |
| **solveCurrentStep()** | **-** | **-** | **96** | **89** | **91** | **89** | **92** | **92** | **95** | **95** |
| formNodalUnbalance() | - | - | 2 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| formElementResidual() | - | - | 22 | 9 | 8 | 5 | 8 | 5 | 6 | 6 |
| formTangent() | - | - | 53 | 65 | 65 | 63 | 65 | 55 | 58 | 57 |
| solve() | - | - | 4 | 10 | 13 | 16 | 17 | 30 | 30 | 29 |
| update() | - | - | 15 | 4 | 3 | 3 | 1 | 1 | 1 | 1 |
| **commit()** | **-** | **-** | **2** | **7** | **6** | **5** | **4** | **3** | **2** | **2** |

Table B.4: % Page Faults on ALPHA for C++ Program

| | Example | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2dF1 | 2dF2 | 2dF3 | 2dF4 | 2dF5 | 2dF6 | 3dF3 | 3dF4 |
| **domainChanged()** | **-** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| handle() | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| numberDOF() | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| getDOFGraph() | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| setSize() | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **applyLoad()** | **-** | **0** | **5** | **1** | **5** | **5** | **3** | **3** |
| **solveCurrentStep()** | **-** | **93** | **89** | **94** | **91** | **91** | **96** | **96** |
| formNodalUnbalance() | - | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| formElementResidual() | - | 28 | 4 | 5 | 4 | 4 | 2 | 2 |
| formTangent() | - | 64 | 78 | 64 | 61 | 60 | 64 | 61 |
| solve() | - | 0 | 2 | 21 | 23 | 24 | 28 | 31 |
| update() | - | 0 | 4 | 2 | 2 | 2 | 1 | 1 |
| **commit()** | **-** | **7** | **5** | **5** | **3** | **3** | **1** | **1** |

Table B.5: % Page Faults on DEC for C++ Program

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 2.10 | - | - | - | 2.10 | 1.92 | 0 | 4650 | 93 | - | - | - | - | - | - | - | - |
| 3 | 2.85 | 0.90 | 0.45 | 0 | 1.95 | 0.02 | 0 | 96 | 48 | 0.65 | 0.63 | 0 | 2394 | 96 | 79 | 1525 | 798 |
| | | | | | | | | | | 0.68 | 0.67 | 0 | 2352 | 96 | 82 | 1525 | 815 |
| 4 | 3.65 | 1.13 | 0.57 | 0 | 2.52 | 0.08 | 0 | 186 | 93 | 0.37 | 0.35 | 0 | 1614 | 93 | 72 | 1017 | 538 |
| | | | | | | | | | | 1.27 | 1.27 | 0 | 1653 | 186 | 139 | 1016 | 551 |
| | | | | | | | | | | 0.33 | 0.32 | 0 | 1569 | 93 | 70 | 1017 | 554 |
| 5 | 3.05 | 1.17 | 0.68 | 0 | 1.88 | 0.12 | 0 | 243 | 100 | 0.37 | 0.35 | 0 | 1233 | 117 | 85 | 762 | 411 |
| | | | | | | | | | | 0.25 | 0.25 | 0 | 1230 | 120 | 68 | 763 | 410 |
| | | | | | | | | | | 0.23 | 0.25 | 0 | 1224 | 132 | 67 | 763 | 421 |
| | | | | | | | | | | 0.43 | 0.40 | 0 | 1206 | 117 | 95 | 762 | 420 |
| 7 | 4.45 | 1.47 | 1.03 | 0 | 2.98 | 0.20 | 0 | 345 | 113 | 0.17 | 0.17 | 0 | 840 | 99 | 74 | 508 | 280 |
| | | | | | | | | | | 0.18 | 0.18 | 0 | 831 | 93 | 70 | 509 | 277 |
| | | | | | | | | | | 0.40 | 0.40 | 0 | 855 | 159 | 113 | 509 | 285 |
| | | | | | | | | | | 0.18 | 0.18 | 0 | 816 | 105 | 72 | 508 | 291 |
| | | | | | | | | | | 0.12 | 0.12 | 0 | 816 | 90 | 53 | 508 | 284 |
| | | | | | | | | | | 0.35 | 0.35 | 0 | 846 | 153 | 108 | 508 | 282 |

Table B.6: Results for 2dF1 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 4.10 | - | - | - | 4.10 | 3.80 | 0 | 6150 | 122 | - | - | - | - | - | - | - | - |
| 3 | 4.05 | 1.28 | 0.66 | 0 | 2.77 | 0.03 | 0 | 126 | 63 | 1.25 | 1.23 | 0 | 3171 | 126 | 95 | 2025 | 1057 |
| | | | | | | | | | | 1.17 | 1.15 | 0 | 3105 | 126 | 96 | 2025 | 1076 |
| 4 | 4.28 | 1.55 | 0.87 | 0 | 2.73 | 0.12 | 0 | 222 | 111 | 1.07 | 1.07 | 0 | 2088 | 138 | 114 | 1350 | 729 |
| | | | | | | | | | | 1.43 | 1.43 | 0 | 2154 | 171 | 125 | 1350 | 726 |
| | | | | | | | | | | 1.13 | 1.12 | 0 | 2133 | 138 | 116 | 1350 | 711 |
| 5 | 3.65 | 1.58 | 0.97 | 0 | 2.07 | 0.15 | 0 | 267 | 114 | 0.35 | 0.35 | 0 | 1620 | 144 | 72 | 1013 | 540 |
| | | | | | | | | | | 0.72 | 0.72 | 0 | 1626 | 132 | 107 | 1013 | 542 |
| | | | | | | | | | | 0.37 | 0.37 | 0 | 1596 | 135 | 75 | 1012 | 550 |
| | | | | | | | | | | 0.70 | 0.70 | 0 | 1581 | 129 | 110 | 1012 | 550 |
| 7 | 5.85 | 2.30 | 1.63 | 0 | 1.85 | 0.42 | 0 | 414 | 139 | 0.28 | 0.28 | 0 | 1101 | 108 | 79 | 675 | 367 |
| | | | | | | | | | | 0.77 | 0.77 | 0 | 1122 | 195 | 136 | 675 | 374 |
| | | | | | | | | | | 0.18 | 0.18 | 0 | 1098 | 120 | 62 | 675 | 366 |
| | | | | | | | | | | 0.32 | 0.32 | 0 | 1068 | 105 | 87 | 675 | 372 |
| | | | | | | | | | | 0.77 | 0.75 | 0 | 1116 | 186 | 132 | 675 | 372 |
| | | | | | | | | | | 0.18 | 0.18 | 0 | 1062 | 117 | 61 | 675 | 379 |

Table B.7: Results for 2dF2 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 23.10 | - | - | - | 23.10 | 6.92 | 985 | 7650 | 151 | - | - | - | - | - | - | - | - |
| 3 | 6.08 | 1.83 | 0.92 | 0 | 4.25 | 0.05 | 0 | 150 | 75 | 2.72 | 2.68 | 0 | 3897 | 150 | 115 | 2525 | 1325 |
| | | | | | | | | | | 2.65 | 2.62 | 0 | 3903 | 150 | 113 | 2525 | 1326 |
| 4 | 6.05 | 2.02 | 1.22 | 0 | 4.03 | 0.17 | 0 | 252 | 126 | 1.30 | 1.28 | 0 | 2589 | 153 | 107 | 1684 | 914 |
| | | | | | | | | | | 2.37 | 2.35 | 0 | 2664 | 171 | 142 | 1683 | 888 |
| | | | | | | | | | | 1.05 | 1.03 | 0 | 2652 | 183 | 97 | 1683 | 884 |
| 5 | 5.05 | 2.10 | 1.28 | 0 | 2.95 | 0.23 | 0 | 300 | 133 | 0.58 | 0.58 | 0 | 2007 | 153 | 84 | 1262 | 669 |
| | | | | | | | | | | 0.52 | 0.52 | 0 | 1968 | 147 | 79 | 1263 | 682 |
| | | | | | | | | | | 1.30 | 1.28 | 0 | 2013 | 156 | 129 | 1263 | 682 |
| | | | | | | | | | | 1.17 | 1.17 | 0 | 1968 | 150 | 122 | 1262 | 681 |
| 7 | 7.42 | 2.63 | 1.92 | 0 | 4.38 | 0.47 | 0 | 444 | 163 | 0.47 | 0.45 | 0 | 1311 | 117 | 94 | 841 | 463 |
| | | | | | | | | | | 1.22 | 1.22 | 0 | 1389 | 207 | 152 | 842 | 463 |
| | | | | | | | | | | 0.28 | 0.28 | 0 | 1356 | 138 | 70 | 842 | 452 |
| | | | | | | | | | | 0.52 | 0.50 | 0 | 1362 | 120 | 98 | 842 | 454 |
| | | | | | | | | | | 0.32 | 0.28 | 0 | 1326 | 141 | 73 | 841 | 467 |
| | | | | | | | | | | 1.57 | 1.55 | 0 | 1362 | 177 | 141 | 842 | 454 |

Table B.8: Results for 2dF3 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 57.20 | - | - | - | 57.20 | 8.63 | 4220 | 9180 | 152 | - | - | - | - | - | - | - | - |
| 3 | 8.05 | 2.07 | 1.15 | 0 | 5.98 | 0.05 | 0 | 168 | 84 | 3.75 | 3.72 | 0 | 4638 | 168 | 122 | 3030 | 1597 |
| | | | | | | | | | | 3.53 | 3.52 | 0 | 4710 | 168 | 119 | 3030 | 1570 |
| 4 | 8.23 | 2.48 | 1.57 | 0 | 3.38 | 0.25 | 0 | 288 | 144 | 1.67 | 1.67 | 0 | 3105 | 162 | 110 | 2020 | 1086 |
| | | | | | | | | | | 3.98 | 3.93 | 0 | 3189 | 195 | 157 | 2020 | 1063 |
| | | | | | | | | | | 1.83 | 1.78 | 0 | 3177 | 222 | 113 | 2020 | 1059 |
| 5 | 5.98 | 2.60 | 1.65 | 0 | 3.38 | 0.28 | 0 | 330 | 144 | 0.87 | 0.83 | 0 | 2361 | 180 | 92 | 1515 | 811 |
| | | | | | | | | | | 1.40 | 1.40 | 0 | 2358 | 147 | 121 | 1515 | 797 |
| | | | | | | | | | | 1.53 | 1.53 | 0 | 2391 | 159 | 127 | 1515 | 813 |
| | | | | | | | | | | 0.80 | 0.78 | 0 | 2406 | 180 | 90 | 1515 | 802 |
| 7 | 9.28 | 4.08 | 2.43 | 8 | 5.20 | 0.75 | 0 | 525 | 192 | 0.72 | 0.72 | 0 | 1602 | 135 | 108 | 1010 | 548 |
| | | | | | | | | | | 1.88 | 1.87 | 0 | 1614 | 231 | 169 | 1010 | 560 |
| | | | | | | | | | | 0.40 | 0.40 | 0 | 1605 | 159 | 77 | 1010 | 550 |
| | | | | | | | | | | 0.45 | 0.45 | 0 | 1623 | 150 | 83 | 1010 | 541 |
| | | | | | | | | | | 1.10 | 1.10 | 0 | 1638 | 234 | 134 | 1010 | 546 |
| | | | | | | | | | | 0.43 | 0.43 | 0 | 1632 | 150 | 78 | 1010 | 544 |

Table B.9: Results for 2dF4 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 81.53 | - | - | - | 81.50 | 10.10 | 6377 | 10710 | 152 | - | - | - | - | - | - | - | - |
| 3 | 16.15 | 2.58 | 1.58 | 0 | 13.57 | 0.13 | 0 | 219 | 110 | 8.12 | 7.78 | 45 | 5487 | 219 | 152 | 3535 | 1851 |
| | | | | | | | | | | 8.33 | 7.98 | 19 | 5442 | 219 | 152 | 3535 | 1843 |
| 4 | 8.78 | 3.10 | 1.95 | 0 | 5.68 | 0.30 | 0 | 306 | 153 | 2.22 | 2.20 | 0 | 3615 | 156 | 113 | 2356 | 1256 |
| | | | | | | | | | | 2.53 | 2.52 | 0 | 3711 | 246 | 120 | 2357 | 1237 |
| | | | | | | | | | | 4.60 | 4.53 | 0 | 3693 | 213 | 154 | 2357 | 1231 |
| 5 | 8.62 | 3.10 | 2.03 | 0 | 5.52 | 0.35 | 0 | 363 | 157 | 2.53 | 2.53 | 0 | 2769 | 177 | 142 | 1768 | 945 |
| | | | | | | | | | | 2.88 | 2.88 | 0 | 2781 | 174 | 147 | 1767 | 927 |
| | | | | | | | | | | 1.23 | 1.22 | 0 | 2799 | 213 | 103 | 1768 | 933 |
| | | | | | | | | | | 2.37 | 2.35 | 0 | 2730 | 168 | 138 | 1767 | 939 |
| 7 | 10.85 | 4.95 | 2.97 | 17 | 5.90 | 0.72 | 0 | 525 | 179 | 0.50 | 0.48 | 0 | 1857 | 156 | 81 | 1179 | 641 |
| | | | | | | | | | | 0.60 | 0.56 | 0 | 1890 | 159 | 87 | 1179 | 641 |
| | | | | | | | | | | 2.43 | 2.42 | 0 | 1896 | 216 | 176 | 1179 | 630 |
| | | | | | | | | | | 1.02 | 1.02 | 0 | 1878 | 144 | 118 | 1178 | 626 |
| | | | | | | | | | | 1.10 | 1.10 | 0 | 1836 | 147 | 122 | 1178 | 641 |
| | | | | | | | | | | 2.42 | 2.42 | 0 | 1884 | 234 | 174 | 1178 | 628 |

Table B.10: Results for 2dF5 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 105.00 | - | - | - | 105.00 | 11.40 | 8220 | 12240 | 153 | - | - | - | - | - | - | - | - |
| 3 | 15.40 | 2.82 | 1.55 | 0 | 12.58 | 0.07 | 0 | 159 | 80 | 7.15 | 5.02 | 125 | 6168 | 159 | 126 | 4040 | 2107 |
| | | | | | | | | | | 6.63 | 4.60 | 102 | 6231 | 159 | 123 | 4040 | 2077 |
| 4 | 28.37 | 3.38 | 2.23 | 0 | 24.98 | 0.37 | 0 | 321 | 161 | 2.82 | 2.78 | 0 | 4197 | 156 | 115 | 2694 | 1399 |
| | | | | | | | | | | 18.55 | 17.02 | 90 | 4242 | 321 | 248 | 2693 | 1414 |
| | | | | | | | | | | 3.43 | 3.38 | 0 | 4122 | 165 | 123 | 2693 | 1425 |
| 5 | 10.30 | 4.22 | 2.45 | 8 | 6.08 | 0.40 | 0 | 393 | 161 | 1.43 | 1.43 | 0 | 3147 | 198 | 105 | 2020 | 1072 |
| | | | | | | | | | | 3.67 | 3.65 | 0 | 3141 | 189 | 149 | 2020 | 1075 |
| | | | | | | | | | | 3.98 | 3.93 | 0 | 3171 | 198 | 158 | 2020 | 1057 |
| | | | | | | | | | | 1.62 | 1.62 | 0 | 3180 | 207 | 109 | 2020 | 1060 |
| 7 | 13.93 | 6.35 | 4.03 | 25 | 7.57 | 0.80 | 0 | 579 | 192 | 0.72 | 0.72 | 0 | 2124 | 186 | 90 | 1346 | 724 |
| | | | | | | | | | | 3.65 | 3.63 | 0 | 2160 | 243 | 189 | 1346 | 720 |
| | | | | | | | | | | 1.35 | 1.35 | 0 | 2091 | 156 | 129 | 1347 | 732 |
| | | | | | | | | | | 0.73 | 0.73 | 0 | 2139 | 177 | 92 | 1347 | 713 |
| | | | | | | | | | | 1.93 | 1.93 | 0 | 2169 | 249 | 147 | 1347 | 723 |
| | | | | | | | | | | 0.65 | 0.63 | 0 | 2148 | 159 | 86 | 1347 | 716 |

Table B.11: Results for 2dF6 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 83.18 | - | - | - | 83.18 | 17.03 | 6235 | 7680 | 239 | - | - | - | - | - | - | - | - |
| 3 | 10.15 | 1.47 | 0.78 | 0 | 8.68 | 0.15 | 0 | 240 | 120 | 6.57 | 6.50 | 0 | 4008 | 240 | 194 | 1280 | 668 |
| | | | | | | | | | | 6.87 | 6.78 | 0 | 3912 | 240 | 198 | 1280 | 692 |
| 4 | 22.12 | 3.67 | 2.80 | 0 | 18.45 | 1.73 | 0 | 486 | 243 | 3.83 | 3.83 | 0 | 2616 | 240 | 184 | 853 | 476 |
| | | | | | | | | | | 13.30 | 13.20 | 0 | 2784 | 384 | 293 | 854 | 464 |
| | | | | | | | | | | 12.58 | 12.45 | 0 | 2778 | 384 | 285 | 853 | 463 |
| 5 | 16.70 | 4.33 | 3.22 | 0 | 12.37 | 1.80 | 0 | 606 | 248 | 5.85 | 5.85 | 0 | 2112 | 312 | 250 | 640 | 352 |
| | | | | | | | | | | 5.86 | 5.86 | 0 | 2094 | 312 | 241 | 640 | 349 |
| | | | | | | | | | | 2.58 | 2.58 | 0 | 2064 | 306 | 186 | 640 | 363 |
| | | | | | | | | | | 2.42 | 2.40 | 0 | 2034 | 300 | 178 | 640 | 360 |
| 7 | 15.23 | 5.22 | 3.75 | 15 | 9.98 | 2.98 | 3 | 780 | 281 | 1.35 | 1.32 | 0 | 1434 | 282 | 168 | 427 | 239 |
| | | | | | | | | | | 1.20 | 1.15 | 0 | 1434 | 270 | 154 | 427 | 239 |
| | | | | | | | | | | 1.07 | 1.05 | 0 | 1428 | 252 | 148 | 426 | 238 |
| | | | | | | | | | | 2.10 | 2.08 | 0 | 1392 | 258 | 203 | 427 | 246 |
| | | | | | | | | | | 2.03 | 2.03 | 0 | 1386 | 258 | 201 | 427 | 244 |
| | | | | | | | | | | 2.30 | 2.30 | 0 | 1416 | 270 | 207 | 426 | 249 |

Table B.12: Results for 3dF3 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 130.01 | - | - | - | 130.01 | 20.70 | 10163 | 9360 | 238 | - | - | - | - | - | - | - | - |
| 3 | 16.40 | 1.62 | 0.87 | 0 | 14.78 | 0.15 | 0 | 240 | 120 | 10.35 | 8.70 | 152 | 4872 | 240 | 203 | 1560 | 812 |
| | | | | | | | | | | 11.62 | 9.23 | 101 | 4728 | 240 | 209 | 1560 | 828 |
| 4 | 33.92 | 3.78 | 2.90 | 0 | 30.13 | 1.45 | 0 | 486 | 243 | 5.40 | 5.33 | 0 | 3174 | 246 | 193 | 1040 | 569 |
| | | | | | | | | | | 24.68 | 23.47 | 74 | 3360 | 486 | 348 | 1040 | 560 |
| | | | | | | | | | | 4.97 | 4.93 | 0 | 3312 | 240 | 185 | 1040 | 552 |
| 5 | 22.50 | 5.43 | 3.85 | 6 | 17.03 | 2.23 | 1 | 648 | 260 | 8.98 | 8.97 | 0 | 2538 | 330 | 263 | 780 | 423 |
| | | | | | | | | | | 10.20 | 10.02 | 14 | 2544 | 342 | 271 | 780 | 424 |
| | | | | | | | | | | 7.73 | 7.72 | 0 | 2448 | 306 | 253 | 780 | 432 |
| | | | | | | | | | | 5.40 | 5.38 | 0 | 2496 | 336 | 216 | 780 | 432 |
| 7 | 20.58 | 7.87 | 5.45 | 13 | 12.70 | 4.27 | 3 | 900 | 305 | 1.92 | 1.90 | 0 | 1740 | 300 | 173 | 520 | 290 |
| | | | | | | | | | | 2.78 | 2.73 | 0 | 1758 | 342 | 199 | 520 | 293 |
| | | | | | | | | | | 3.33 | 3.30 | 0 | 1722 | 288 | 219 | 520 | 287 |
| | | | | | | | | | | 3.35 | 3.33 | 0 | 1692 | 300 | 222 | 520 | 294 |
| | | | | | | | | | | 3.40 | 3.38 | 0 | 1692 | 318 | 222 | 520 | 294 |
| | | | | | | | | | | 3.35 | 3.35 | 0 | 1686 | 282 | 217 | 520 | 297 |

Table B.13: Results for 3dF4 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 147.57 | - | - | - | 147.57 | 21.78 | 11653 | 10800 | 237 | - | - | - | - | - | - | - | - |
| 3 | 23.82 | 1.87 | 0.93 | 0 | 21.95 | .17 | 0 | 240 | 120 | 13.38 | 10.45 | 342 | 5586 | 240 | 210 | 1800 | 931 |
| | | | | | | | | | | 15.78 | 10.65 | 547 | 5454 | 240 | 214 | 1800 | 949 |
| 4 | 56.00 | 4.28 | 3.25 | 0 | 51.72 | 1.97 | 0 | 504 | 252 | 6.90 | 6.88 | 0 | 3774 | 246 | 194 | 1200 | 629 |
| | | | | | | | | | | 38.08 | 25.62 | 1176 | 3870 | 504 | 336 | 1200 | 645 |
| | | | | | | | | | | 7.95 | 7.86 | 82 | 3660 | 258 | 206 | 1200 | 650 |
| 5 | 32.88 | 7.18 | 5.62 | 7 | 25.50 | 2.90 | 11 | 732 | 283 | 14.27 | 14.15 | 0 | 2922 | 372 | 294 | 900 | 487 |
| | | | | | | | | | | 14.40 | 14.03 | 0 | 2928 | 384 | 290 | 900 | 488 |
| | | | | | | | | | | 4.22 | 4.20 | 0 | 2748 | 240 | 185 | 900 | 498 |
| | | | | | | | | | | 19.77 | 19.23 | 44 | 2946 | 480 | 341 | 900 | 491 |
| 7 | 29.57 | 11.68 | 7.60 | 46 | 17.60 | 5.57 | 152 | 1008 | 329 | 3.13 | 3.15 | 0 | 1992 | 354 | 200 | 600 | 332 |
| | | | | | | | | | | 3.23 | 3.18 | 0 | 2004 | 306 | 200 | 600 | 334 |
| | | | | | | | | | | 3.57 | 3.55 | 0 | 1986 | 366 | 217 | 600 | 331 |
| | | | | | | | | | | 4.25 | 4.25 | 0 | 1992 | 360 | 235 | 600 | 332 |
| | | | | | | | | | | 5.03 | 5.01 | 0 | 2004 | 396 | 249 | 600 | 334 |
| | | | | | | | | | | 2.48 | 2.48 | 0 | 1866 | 270 | 184 | 600 | 351 |

Table B.14: Results for 3dF5 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 173.45 | - | - | - | 173.45 | 25.67 | 13783 | 12240 | 236 | - | - | - | - | - | - | - | - |
| 3 | 52.55 | 2.07 | 1.00 | 0 | 50.48 | 0.17 | 0 | 240 | 120 | 40.12 | 12.05 | 2034 | 6312 | 240 | 214 | 2040 | 1052 |
| | | | | | | | | | | 39.53 | 12.42 | 2313 | 6168 | 240 | 219 | 2040 | 1068 |
| 4 | 84.25 | 4.08 | 2.97 | 0 | 80.17 | 1.68 | 0 | 480 | 240 | 14.17 | 7.26 | 1314 | 4242 | 240 | 196 | 1360 | 707 |
| | | | | | | | | | | 65.98 | 33.98 | 2770 | 4344 | 480 | 360 | 1360 | 724 |
| | | | | | | | | | | 7.57 | 7.40 | 9 | 4134 | 240 | 200 | 1360 | 729 |
| 5 | 38.63 | 7.87 | 5.82 | 2 | 30.75 | 2.82 | 26 | 720 | 280 | 20.40 | 20.03 | 23 | 3300 | 480 | 328 | 1020 | 550 |
| | | | | | | | | | | 5.93 | 5.47 | 17 | 3252 | 240 | 185 | 1020 | 542 |
| | | | | | | | | | | 20.68 | 19.82 | 41 | 3288 | 480 | 329 | 1020 | 548 |
| | | | | | | | | | | 5.07 | 5.05 | 0 | 3120 | 240 | 189 | 1020 | 560 |
| 7 | 45.63 | 15.33 | 10.50 | 40 | 30.12 | 6.90 | 470 | 1098 | 338 | 4.65 | 4.60 | 0 | 2208 | 252 | 217 | 680 | 368 |
| | | | | | | | | | | 12.85 | 12.12 | 67 | 2310 | 504 | 325 | 680 | 385 |
| | | | | | | | | | | 4.13 | 4.10 | 0 | 2220 | 330 | 214 | 680 | 370 |
| | | | | | | | | | | 8.98 | 8.88 | 0 | 2202 | 408 | 285 | 680 | 381 |
| | | | | | | | | | | 6.93 | 6.88 | 0 | 2196 | 348 | 258 | 680 | 382 |
| | | | | | | | | | | 7.37 | 7.28 | 0 | 2220 | 372 | 263 | 680 | 380 |

Table B.15: Results for 3dF6 on ALPHA Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 26 | - | - | - | 26 | 24.3 | 0 | 4650 | 93 | - | - | - | - | - | - | - | - |
| 3 | 23 | 6 | 4 | 0 | 17 | 0.2 | 0 | 96 | 48 | 11 | 11 | 0 | 2394 | 96 | 79 | 1525 | 798 |
| | | | | | | | | | | 12 | 11 | 0 | 2352 | 96 | 82 | 1525 | 815 |
| 4 | 40 | 10 | 6 | 0 | 30 | 1.4 | 0 | 186 | 93 | 7 | 7 | 0 | 1614 | 93 | 72 | 1017 | 538 |
| | | | | | | | | | | 27 | 27 | 0 | 1653 | 186 | 139 | 1016 | 551 |
| | | | | | | | | | | 7 | 6 | 0 | 1569 | 93 | 70 | 1017 | 554 |
| 5 | 22 | 10 | 7 | 0 | 12 | 2.0 | 0 | 243 | 100 | 8 | 7 | 0 | 1233 | 117 | 85 | 762 | 411 |
| | | | | | | | | | | 5 | 4 | 0 | 1230 | 120 | 68 | 763 | 410 |
| | | | | | | | | | | 4 | 4 | 0 | 1224 | 132 | 67 | 763 | 421 |
| | | | | | | | | | | 9 | 9 | 0 | 1206 | 117 | 95 | 762 | 420 |
| 7 | 38 | 20 | 12 | 2 | 18 | 3.6 | 0 | 345 | 113 | 4 | 4 | 0 | 840 | 99 | 74 | 508 | 280 |
| | | | | | | | | | | 3 | 3 | 0 | 831 | 93 | 70 | 509 | 277 |
| | | | | | | | | | | 8 | 8 | 0 | 855 | 159 | 113 | 509 | 285 |
| | | | | | | | | | | 4 | 3 | 0 | 816 | 90 | 72 | 508 | 284 |
| | | | | | | | | | | 2 | 2 | 0 | 816 | 105 | 53 | 508 | 291 |
| | | | | | | | | | | 8 | 8 | 0 | 846 | 153 | 108 | 508 | 282 |

Table B.16: Results for 2dF1 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 87 | - | - | - | 87 | 62.7 | 14 | 6150 | 122 | - | - | - | - | - | - | - | - |
| 3 | 38 | 10 | 6 | 0 | 28 | 0.4 | 0 | 126 | 63 | 22 | 21 | 0 | 3171 | 126 | 95 | 2025 | 1057 |
| | | | | | | | | | | 22 | 22 | 0 | 3105 | 126 | 96 | 2025 | 1076 |
| 4 | 45 | 14 | 11 | 1 | 31 | 3.0 | 0 | 231 | 116 | 23 | 23 | 0 | 2088 | 144 | 117 | 1350 | 729 |
| | | | | | | | | | | 26 | 26 | 0 | 2154 | 162 | 124 | 1350 | 726 |
| | | | | | | | | | | 26 | 26 | 0 | 2142 | 159 | 117 | 1350 | 714 |
| 5 | 35 | 15 | 11 | 1 | 20 | 3.0 | 0 | 267 | 114 | 6 | 6 | 0 | 1620 | 144 | 72 | 1013 | 540 |
| | | | | | | | | | | 15 | 15 | 0 | 1626 | 132 | 107 | 1013 | 542 |
| | | | | | | | | | | 7 | 7 | 0 | 1596 | 135 | 75 | 1012 | 550 |
| | | | | | | | | | | 15 | 15 | 0 | 1581 | 129 | 110 | 1012 | 550 |
| 7 | 51 | 26 | 20 | 1 | 25 | 6.8 | 1 | 414 | 139 | 6 | 6 | 0 | 1101 | 108 | 79 | 675 | 367 |
| | | | | | | | | | | 11 | 11 | 0 | 1122 | 192 | 113 | 675 | 374 |
| | | | | | | | | | | 3 | 3 | 0 | 1098 | 123 | 62 | 675 | 366 |
| | | | | | | | | | | 6 | 6 | 0 | 1068 | 105 | 87 | 675 | 372 |
| | | | | | | | | | | 15 | 15 | 0 | 1116 | 186 | 132 | 675 | 372 |
| | | | | | | | | | | 3 | 3 | 0 | 1062 | 117 | 61 | 675 | 379 |

Table B.17: Results for 2dF2 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 362 | - | - | - | 362 | 131.4 | 4146 | 7650 | 151 | - | - | - | - | - | - | - | - |
| 3 | 60 | 13 | 9 | 0 | 47 | 0.7 | 0 | 150 | 75 | 38 | 38 | 0 | 3897 | 150 | 112 | 2525 | 1327 |
| | | | | | | | | | | 37 | 37 | 0 | 3903 | 150 | 112 | 2525 | 1324 |
| 4 | 71 | 18 | 14 | 0 | 53 | 3.8 | 0 | 249 | 125 | 30 | 29 | 0 | 2664 | 156 | 109 | 1683 | 888 |
| | | | | | | | | | | 47 | 46 | 0 | 2631 | 177 | 147 | 1684 | 902 |
| | | | | | | | | | | 42 | 41 | 0 | 2604 | 165 | 142 | 1683 | 894 |
| 5 | 51 | 20 | 14 | 2 | 31 | 4.9 | 0 | 300 | 133 | 23 | 23 | 0 | 2007 | 144 | 121 | 1262 | 669 |
| | | | | | | | | | | 12 | 12 | 0 | 1971 | 162 | 88 | 1263 | 685 |
| | | | | | | | | | | 24 | 24 | 0 | 1980 | 153 | 125 | 1262 | 683 |
| | | | | | | | | | | 23 | 23 | 0 | 1995 | 144 | 120 | 1263 | 665 |
| 7 | 81 | 37 | 25 | 1 | 43 | 10.4 | 23 | 444 | 162 | 10 | 9 | 0 | 1350 | 111 | 93 | 841 | 450 |
| | | | | | | | | | | 6 | 5 | 0 | 1326 | 129 | 68 | 841 | 450 |
| | | | | | | | | | | 25 | 24 | 0 | 1371 | 213 | 149 | 842 | 457 |
| | | | | | | | | | | 7 | 6 | 0 | 1362 | 132 | 72 | 842 | 454 |
| | | | | | | | | | | 5 | 5 | 0 | 1314 | 120 | 67 | 842 | 461 |
| | | | | | | | | | | 15 | 14 | 0 | 1380 | 192 | 115 | 842 | 460 |

Table B.18: Results for 2dF3 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 550 | - | - | - | 550 | 160.3 | 8908 | 9180 | 152 | - | - | - | - | - | - | - | - |
| 3 | 74 | 16 | 11 | 0 | 58 | 0.8 | 0 | 156 | 78 | 48 | 47 | 0 | 4638 | 156 | 117 | 3030 | 1597 |
| | | | | | | | | | | 49 | 48 | 0 | 4698 | 156 | 117 | 3030 | 1566 |
| 4 | 94 | 32 | 17 | 2 | 61 | 4.7 | 0 | 267 | 134 | 32 | 32 | 0 | 3099 | 153 | 110 | 2020 | 1084 |
| | | | | | | | | | | 30 | 29 | 0 | 3177 | 198 | 108 | 2020 | 1059 |
| | | | | | | | | | | 53 | 53 | 2 | 3174 | 186 | 145 | 2020 | 1058 |
| 5 | 70 | 24 | 19 | 0 | 46 | 6.3 | 0 | 333 | 143 | 35 | 35 | 0 | 2364 | 162 | 137 | 1515 | 815 |
| | | | | | | | | | | 33 | 33 | 0 | 2361 | 165 | 134 | 1515 | 811 |
| | | | | | | | | | | 28 | 27 | 0 | 2400 | 147 | 122 | 1515 | 800 |
| | | | | | | | | | | 15 | 15 | 0 | 2394 | 198 | 91 | 1515 | 798 |
| 7 | 118 | 58 | 31 | 41 | 57 | 14.3 | 114 | 501 | 179 | 15 | 14 | 0 | 1590 | 129 | 105 | 1010 | 551 |
| | | | | | | | | | | 32 | 31 | 0 | 1608 | 216 | 156 | 1010 | 554 |
| | | | | | | | | | | 10 | 10 | 0 | 1614 | 159 | 85 | 1010 | 550 |
| | | | | | | | | | | 15 | 14 | 0 | 1617 | 141 | 104 | 1010 | 539 |
| | | | | | | | | | | 30 | 29 | 0 | 1644 | 237 | 148 | 1010 | 548 |
| | | | | | | | | | | 13 | 12 | 0 | 1614 | 126 | 95 | 1010 | 538 |

Table B.19: Results for 2dF4 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 758 | - | - | - | 758 | 188.3 | 13889 | 10710 | 152 | - | - | - | - | - | - | - | - |
| 3 | 94 | 19 | 14 | 1 | 74 | 0.8 | 0 | 153 | 77 | 58 | 57 | 2 | 5472 | 153 | 120 | 3535 | 1824 |
| | | | | | | | | | | 63 | 59 | 12 | 5391 | 153 | 121 | 3535 | 1848 |
| 4 | 148 | 38 | 24 | 1 | 110 | 6.9 | 0 | 303 | 152 | 40 | 39 | 0 | 3696 | 156 | 115 | 2357 | 1232 |
| | | | | | | | | | | 99 | 98 | 0 | 3660 | 225 | 184 | 2356 | 1250 |
| | | | | | | | | | | 40 | 39 | 2 | 3660 | 228 | 115 | 2357 | 1241 |
| 5 | 86 | 32 | 25 | 6 | 53 | 7.3 | 0 | 357 | 149 | 24 | 24 | 0 | 2790 | 192 | 104 | 1768 | 930 |
| | | | | | | | | | | 42 | 42 | 0 | 2784 | 165 | 141 | 1768 | 928 |
| | | | | | | | | | | 36 | 36 | 0 | 2733 | 162 | 132 | 1767 | 942 |
| | | | | | | | | | | 27 | 27 | 0 | 2763 | 198 | 110 | 1767 | 941 |
| 7 | 141 | 66 | 44 | 27 | 76 | 9.4 | 164 | 537 | 181 | 10 | 10 | 0 | 1881 | 150 | 81 | 1178 | 627 |
| | | | | | | | | | | 16 | 16 | 0 | 1878 | 141 | 104 | 1178 | 626 |
| | | | | | | | | | | 31 | 31 | 0 | 1908 | 252 | 144 | 1178 | 636 |
| | | | | | | | | | | 20 | 20 | 0 | 1854 | 147 | 121 | 1179 | 637 |
| | | | | | | | | | | 11 | 11 | 0 | 1842 | 168 | 85 | 1179 | 646 |
| | | | | | | | | | | 47 | 47 | 0 | 1893 | 225 | 179 | 1178 | 631 |

Table B.20: Results for 2dF5 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 894 | - | - | - | 894 | 215.6 | 16387 | 12240 | 153 | - | - | - | - | - | - | - | - |
| 3 | 123 | 23 | 214 | 9 | 100 | 1.0 | 1 | 159 | 80 | 73 | 71 | 8 | 6168 | 159 | 126 | 4040 | 2107 |
| | | | | | | | | | | 76 | 68 | 62 | 6231 | 159 | 123 | 4040 | 2077 |
| 4 | 306 | 53 | 27 | 24 | 252 | 8.3 | 189 | 321 | 161 | 43 | 43 | 0 | 4197 | 156 | 115 | 2694 | 1399 |
| | | | | | | | | | | 233 | 224 | 65 | 4242 | 321 | 248 | 2693 | 1414 |
| | | | | | | | | | | 52 | 51 | 1 | 4122 | 165 | 123 | 2693 | 1425 |
| 5 | 143 | 61 | 28 | 42 | 82 | 9.4 | 164 | 393 | 161 | 27 | 27 | 0 | 3147 | 198 | 105 | 2020 | 66 |
| | | | | | | | | | | 58 | 57 | 0 | 3141 | 189 | 149 | 2020 | 63 |
| | | | | | | | | | | 63 | 62 | 0 | 3171 | 198 | 158 | 2020 | 66 |
| | | | | | | | | | | 29 | 29 | 0 | 3180 | 207 | 109 | 2020 | 69 |
| 7 | 185 | 90 | 51 | 2 | 94 | 18.8 | 168 | 579 | 192 | 14 | 13 | 0 | 2124 | 186 | 90 | 1346 | 724 |
| | | | | | | | | | | 61 | 61 | 0 | 2160 | 243 | 89 | 1346 | 720 |
| | | | | | | | | | | 14 | 14 | 0 | 2091 | 156 | 129 | 1347 | 732 |
| | | | | | | | | | | 14 | 14 | 0 | 2139 | 177 | 92 | 1347 | 713 |
| | | | | | | | | | | 36 | 36 | 0 | 2169 | 249 | 147 | 1347 | 723 |
| | | | | | | | | | | 12 | 11 | 0 | 2148 | 159 | 86 | 1347 | 716 |

Table B.21: Results for 2dF6 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 875 | - | - | - | 875 | 341.5 | 13156 | 7680 | 239 | - | - | - | - | - | - | - | - |
| 3 | 140 | 14 | 10 | 0 | 126 | 3.4 | 0 | 240 | 120 | 116 | 115 | 2 | 4008 | 240 | 194 | 1280 | 668 |
| | | | | | | | | | | 119 | 118 | 4 | 3912 | 240 | 198 | 1280 | 692 |
| 4 | 296 | 61 | 50 | 5 | 230 | 28.7 | 73 | 486 | 243 | 67 | 67 | 1 | 2616 | 240 | 184 | 853 | 476 |
| | | | | | | | | | | 192 | 191 | 2 | 2784 | 384 | 293 | 854 | 464 |
| | | | | | | | | | | 178 | 176 | 2 | 2778 | 360 | 285 | 853 | 463 |
| 5 | 247 | 83 | 60 | 32 | 162 | 34.4 | 591 | 606 | 248 | 102 | 101 | 0 | 2112 | 312 | 250 | 640 | 352 |
| | | | | | | | | | | 94 | 93 | 2 | 2094 | 312 | 241 | 640 | 349 |
| | | | | | | | | | | 53 | 52 | 0 | 2064 | 306 | 186 | 640 | 363 |
| | | | | | | | | | | 48 | 47 | 2 | 2034 | 300 | 178 | 640 | 360 |
| 7 | 236 | 101 | 68 | 86 | 133 | 55.1 | 718 | 780 | 281 | 29 | 28 | 0 | 1434 | 282 | 168 | 427 | 239 |
| | | | | | | | | | | 24 | 24 | 0 | 1434 | 270 | 154 | 427 | 239 |
| | | | | | | | | | | 22 | 22 | 0 | 1428 | 252 | 148 | 426 | 238 |
| | | | | | | | | | | 43 | 43 | 0 | 1392 | 258 | 203 | 427 | 246 |
| | | | | | | | | | | 42 | 42 | 1 | 1386 | 258 | 201 | 427 | 244 |
| | | | | | | | | | | 46 | 46 | 1 | 1416 | 270 | 207 | 426 | 249 |

Table B.22: Results for 3dF3 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | 1119 | - | - | - | 1119 | 418.8 | 17983 | 9360 | 238 | - | - | - | - | - | - | - | - |
| 3 | 286 | 14 | 11 | 0 | 272 | 3.5 | 0 | 240 | 120 | 202 | 155 | 667 | 4872 | 240 | 203 | 1560 | 812 |
| | | | | | | | | | | 221 | 160 | 967 | 4728 | 240 | 209 | 1560 | 828 |
| 4 | 517 | 75 | 52 | 0 | 441 | 28.7 | 213 | 486 | 243 | 90 | 89 | 0 | 3174 | 246 | 193 | 1040 | 569 |
| | | | | | | | | | | 368 | 321 | 787 | 3360 | 486 | 348 | 1040 | 560 |
| | | | | | | | | | | 86 | 86 | 0 | 3312 | 240 | 185 | 1040 | 552 |
| 5 | 325 | 111 | 72 | 50 | 213 | 40.4 | 810 | 648 | 260 | 136 | 136 | 0 | 2538 | 330 | 263 | 780 | 423 |
| | | | | | | | | | | 144 | 142 | 0 | 2544 | 342 | 271 | 780 | 424 |
| | | | | | | | | | | 122 | 121 | 0 | 2448 | 306 | 253 | 780 | 432 |
| | | | | | | | | | | 85 | 85 | 0 | 2496 | 336 | 216 | 780 | 432 |
| 7 | 309 | 133 | 100 | 74 | 174 | 74.0 | 1016 | 900 | 305 | 43 | 40 | 0 | 1740 | 300 | 173 | 520 | 290 |
| | | | | | | | | | | 53 | 52 | 0 | 1758 | 342 | 199 | 520 | 293 |
| | | | | | | | | | | 64 | 63 | 0 | 1722 | 288 | 219 | 520 | 287 |
| | | | | | | | | | | 65 | 64 | 0 | 1692 | 300 | 222 | 520 | 294 |
| | | | | | | | | | | 65 | 64 | 0 | 1692 | 318 | 222 | 520 | 294 |
| | | | | | | | | | | 62 | 61 | 0 | 1686 | 282 | 217 | 520 | 297 |

Table B.23: Results for 3dF4 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | WOULD NOT RUN ON SINGLE DEC WORKSTATION DUE TO MEMORY LIMITS ON USERS | | | | | | | | | | | | | | | | |
| 3 | 416 | 16 | 12 | 0 | 400 | 3.4 | 0 | 240 | 120 | 316 | 191 | 2644 | 5586 | 240 | 210 | 1800 | 931 |
| | | | | | | | | | | 341 | 195 | 2800 | 5454 | 240 | 214 | 1800 | 949 |
| 4 | 614 | 86 | 57 | 18 | 527 | 32.1 | 284 | 504 | 252 | 108 | 108 | 0 | 3774 | 246 | 194 | 1200 | 629 |
| | | | | | | | | | | 468 | 346 | 2317 | 3870 | 504 | 336 | 1200 | 645 |
| | | | | | | | | | | 121 | 119 | 2 | 3660 | 258 | 206 | 1200 | 650 |
| 5 | 513 | 137 | 107 | 25 | 374 | 55.3 | 895 | 738 | 285 | 202 | 200 | 2 | 2922 | 378 | 294 | 900 | 487 |
| | | | | | | | | | | 194 | 193 | 2 | 2922 | 390 | 290 | 899 | 487 |
| | | | | | | | | | | 72 | 72 | 0 | 2760 | 240 | 185 | 901 | 500 |
| | | | | | | | | | | 283 | 272 | 122 | 2946 | 480 | 342 | 900 | 491 |
| 7 | 415 | 176 | 140 | 53 | 239 | 96.4 | 1479 | 1008 | 329 | 61 | 60 | 0 | 1992 | 354 | 200 | 600 | 332 |
| | | | | | | | | | | 60 | 60 | 0 | 2004 | 306 | 200 | 600 | 334 |
| | | | | | | | | | | 70 | 68 | 0 | 1986 | 366 | 217 | 600 | 331 |
| | | | | | | | | | | 80 | 80 | 0 | 1992 | 360 | 235 | 600 | 332 |
| | | | | | | | | | | 91 | 90 | 0 | 2004 | 396 | 249 | 600 | 334 |
| | | | | | | | | | | 49 | 48 | 0 | 1866 | 270 | 184 | 600 | 351 |

Table B.24: Results for 3dF5 on DEC Network

| NP | Total Time | Partition | | | Analyze | | | | | Subdomains | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | REAL | CPU | # PF | REAL | CPU | # PF | DOF | AvgHt | REAL | CPU | # PF | DOF | extDOF | AvgHt | # Ele | # Nod |
| 1 | WOULD NOT RUN ON SINGLE DEC WORKSTATION DUE TO MEMORY LIMITS ON USERS | | | | | | | | | | | | | | | | |
| 3 | 554 | 17 | 13 | 0 | 536 | 3.4 | 0 | 240 | 120 | 427 | 224 | 5055 | 6312 | 240 | 214 | 2040 | 1052 |
| | | | | | | | | | | 459 | 229 | 5186 | 6168 | 240 | 219 | 2040 | 1068 |
| 4 | 881 | 72 | 51 | 17 | 809 | 27.7 | 290 | 480 | 240 | 125 | 124 | 17 | 4242 | 240 | 196 | 1360 | 707 |
| | | | | | | | | | | 726 | 459 | 7142 | 4344 | 480 | 360 | 1360 | 724 |
| | | | | | | | | | | 132 | 127 | 56 | 4134 | 240 | 200 | 1360 | 729 |
| 5 | 564 | 138 | 104 | 34 | 426 | 52.1 | 1097 | 720 | 280 | 309 | 280 | 418 | 3300 | 480 | 328 | 1020 | 550 |
| | | | | | | | | | | 85 | 85 | 0 | 3252 | 240 | 185 | 1020 | 542 |
| | | | | | | | | | | 303 | 278 | 291 | 3288 | 480 | 329 | 1020 | 548 |
| | | | | | | | | | | 86 | 85 | 0 | 3120 | 240 | 189 | 1020 | 560 |
| 7 | 648 | 259 | 190 | 65 | 387 | 111.1 | 1814 | 1098 | 338 | 79 | 79 | 4 | 2208 | 252 | 217 | 680 | 368 |
| | | | | | | | | | | 189 | 183 | 11 | 2310 | 504 | 325 | 680 | 385 |
| | | | | | | | | | | 77 | 76 | 1 | 2220 | 330 | 263 | 680 | 370 |
| | | | | | | | | | | 142 | 138 | 71 | 2202 | 408 | 285 | 680 | 381 |
| | | | | | | | | | | 115 | 114 | 4 | 2196 | 348 | 258 | 680 | 382 |
| | | | | | | | | | | 112 | 119 | 4 | 2220 | 372 | 263 | 680 | 380 |

Table B.25: Results for 3dF6 on DEC Network