Plastic Domain Decomposition Method for Parallel Elastic–Plastic Finite Element Computations in Geomechanics

Report UCD–CompGeoMech–03–07

by:

Boris Jeremić and Guanzhou Jie

Computational Geomechanics Group

Department of Civil and Environmental Engineering

University of California, Davis

Report version: 1. May, 2008, 15:22

The work presented in this short report was supported in part by the following grant sources: Civil and Mechanical System program, Directorate of Engineering of the National Science Foundation, award NSF–CMS–0201231 (cognizant program director Dr. Richard Fragaszy); award NSF–CMS–0337811 (cognizant program director Dr. Steve McCabe); Center for Information Technology Research in the Interest of Society (CITRIS); and by the Department of Civil and Environmental Engineering at the University of California at Davis.

Abstract

This report presents the development of the Plastic Domain Decomposition (PDD) algorithm, which features adaptive dynamic load balancing operations for nonlinear finite element simulations. This algorithm has been implemented using parts of OpenSees (Open System for Earthquake Engineering Simulation) framework. Performance study on SFSI (Soil Foundation Structure Interaction) analysis indicates the efficiency of the proposed algorithm. Large scale prototype SSI (Soil Structure Interaction) analysis have been performed using the developed software package. Results are presented and discussed.

The first part of the report introduces the theoretical background of the proposed PDD algorithm. Multi-objective and multi-constraint graph partitioning algorithms lay the foundation of the algorithm building. Parametric study has been conducted to tune the partitioning kernel for our specific application codes. The algorithm is implemented using Object-Oriented principle and using parts of the OpenSees framework. Design details are presented. Major class abstraction and related member functions are summarized. Interfaces to external libraries are also introduced. Comprehensive performance study on SFSI models has been conducted. A novel application cost model is proposed to deal with implementation-dependent overheads. Comparison of proposed PDD algorithm to classic Domain Decomposition (DD) shows significant performance gains for inelastic finite element simulations.

The second part of the report is devoted to the issue of parallel equation solving in large scale finite element simulations. The efficiency of projection-based iterative solvers and some popular direct solvers is investigated using equation systems extracted from SFSI simulations. Complete parallel implementation has been developed using a number of class and numerical libraries, and frameworks (ParMETIS, OpenSees, PETSc, nDarray, FEMtools...)

For sources or installation help of the parallel simulation system, please contact the first Author. Parallel simulation system is currently installed at our local parallel computer GeoWulf, as well as on SDSC, TACC and CU Boulder parallel machines.

Contents

1 Introduction – Adaptive Parallel Inelastic Finite Element Simulations **10**

1.1 Hypothesis . . . 10

1.2 Scope of Study . . . 12

1.3 Summary of Contents . . . 12

1.4 Development Platform CONSOLID8 . . . 14

I Theory and Implementation **15**

2 Plastic Domain Decomposition Algorithm **16**

2.1 Introduction . . . 16

2.2 Inelastic Parallel Finite Element . . . 18

2.2.1 Adaptive Computation . . . 20

2.2.2 Multiphase Computation . . . 20

2.2.3 Multiconstraint Graph Partitioning . . . 21

2.2.4 Adaptive PDD Algorithm . . . 23

2.3 Adaptive Multilevel Graph Partitioning Algorithm . . . 23

2.3.1 Unified Repartitioning Algorithm . . . 28

2.3.2 Study of ITR in ParMETIS . . . 29

3 Object-Oriented Design of PDD **30**

3.1 Introduction . . . 30

3.2 Object-Oriented Parallel Finite Element Algorithm . . . 30

3.2.1 Modeling Classes . . . 33

3.2.2 Finite Element Model Class . . . 33

3.2.3 Analysis . . . 35

3.2.4 Object-Oriented Domain Decomposition . . . 45

3.2.5 Parallel Object-Oriented Finite Element Design . . . 47

3.3 Dual-Phase Adaptive Load Balancing . . . 52

3.3.1 Elemental Level Load Balancing . . . 52
3.3.2 Equation Solving Load Balancing . . . 52
3.4 Object-Oriented Design of PDD . . . 53
3.4.1 MPI_Channel . . . 58
3.4.2 MPI_ChannelAddress . . . 59
3.4.3 FEM_ObjectBroker . . . 59
3.4.4 Domain . . . 60
3.4.5 PartitionedDomain . . . 60
3.4.6 Node & DOF_Group . . . 60
3.4.7 DomainPartitioner . . . 61
3.4.8 Shadow/ActorSubdomain . . . 62
3.4.9 Send/RecvSelf . . . 64
3.5 Graph Partitioning . . . 64
3.5.1 Construction of Element Graph . . . 65
3.5.2 Interface to ParMETIS/METIS . . . 65
3.6 Data Redistribution . . . 67

4 Performance Studies on PDD Algorithm 71
4.1 Introduction . . . 71
4.2 Parallel Computers . . . 71
4.3 Soil-Foundation Interaction Model . . . 73
4.4 Numerical Study for ITR . . . 74
4.5 Parallel Performance Analysis . . . 83
4.5.1 Soil-Foundation Model with 4,035 DOFs . . . 85
4.5.2 Soil-Foundation Model with 4,938 Elements, 17,604 DOFs . . . 88
4.5.3 Soil-Foundation Model with 9,297 Elements, 32,091 DOFs . . . 93
4.6 Algorithm Fine-Tuning . . . 99
4.7 Fine Tuning on Load Imbalance Tolerance . . . 99
4.8 Globally Adaptive PDD Algorithm . . . 104
4.8.1 Implementations . . . 106
4.8.2 Performance Results . . . 107
4.9 Scalability Study on Prototype Model . . . 112
4.9.1 3 Bent SFSI Finite Element Models . . . 112
4.9.2 Scalability Runs . . . 114
4.10 Conclusions . . . 115

II Parallel Equation Solving in Finite Element Calculations 123

5 Application of Project-Based Iterative Methods in SFSI Problems 124

5.1 Introduction . . . 124
5.2 Projection-Based Iterative Methods . . . 125
5.2.1 Conjugate Gradient Algorithm . . . 125
5.2.2 GMRES . . . 127
5.2.3 BiCGStab and QMR . . . 128
5.3 Preconditioning Techniques . . . 128
5.4 Preconditioners . . . 130
5.4.1 Jacobi Preconditioner . . . 130
5.4.2 Incomplete Cholesky Preconditioner . . . 130
5.4.3 Robust Incomplete Factorization . . . 131
5.5 Numerical Experiments . . . 134
5.6 Conclusion and Future Work . . . 144

6 Performance Study on Parallel Direct/Iterative Solving in SFSI 146

6.1 Parallel Sparse Direct Equation Solvers . . . 147
6.1.1 General Techniques – SPOOLES . . . 147
6.1.2 Frontal and Multifrontal Methods – MUMPS . . . 147
6.1.3 Supernodal Algorithm – SuperLU . . . 150
6.2 Performance Study on SFSI Systems . . . 152
6.2.1 Equation System . . . 152
6.2.2 Performance Results . . . 154
6.3 Conclusion . . . 154

III Bibliography and Appendices 156

A Compilation of Parallel Program (PDD–based) on GNU/Linux Clusters 165

A.1 MPICH . . . 165
A.1.1 SMP On-Board Communication Effective Benchmark . . . 165
A.1.2 Cluster Inter-Switch Communication Effective Benchmark . . . 172
A.2 PETSc . . . 178
A.3 ParMETIS . . . 178

B Import New Element/Material/Load etc. to PDD–based Parallel Program 179

B.1 MovableObject . . . 179
B.2 Send/RecvSelf . . . 179

B.3 Default Constructor . . . 179
B.4 FEM_ObjectBroker . . . 180
B.5 getObjectSize . . . 180

C Commands to Invoke Parallel Equation Solvers 181
C.1 Iterative Solvers . . . 181
C.1.1 Conjugate Gradient Method . . . 181
C.1.2 Precoditioned Conjugate Gradient . . . 181
C.1.3 GMRES Method . . . 181
C.1.4 Precoditioned GMRES . . . 182
C.2 Direct Solvers . . . 182
C.2.1 MUMPS . . . 182
C.2.2 SPOOLES . . . 182
C.2.3 SuperLU_DIST . . . 182

List of Figures

1.1 Nested Hierarchy in Nonlinear Solution Methods (Felippa, 2004) 11

2.1 Multilevel Graph Partitioning Scheme Karypis et al. (2003) 24
2.2 A diagram illustrating the execution of adaptive scientific simulations on high performance parallel computers Schloegel et al. (1999) 26

3.1 Rumbaugh Notation of-Object Oriented Design 31
3.2 Class Diagram of Finite Element Model Classes 32
3.3 Class Diagram of Analysis Aggregation 37
3.4 Overall Algorithm Flow Chart for Nonlinear Finite Element Analysis 39
3.5 Detailed View: *theIntegrator::newStep()* - Incremental Solution Techniques for Nonlinear Finite Element Analysis 40
3.6 Detailed View: Assembly of Global Equation System in *theIntegrator::newStep()* 41
3.7 Detailed View: *theAlgorithm::solveCurrentStep()* - Newton-Raphson Iterative Schemes for Nonlinear Finite Element Analysis 42
3.8 Parallel Activity Flow Diagram of Nonlinear Finite Element Analysis 44
3.9 Class Diagram of Domain Decomposition Analysis 46
3.10 Communication Pattern of Actor-Shadow Models McKenna (1997) 49
3.11 Class Diagram for Parallel Finite Element Analysis 51
3.12 Parallel Data Organization of SFSI Equation System 54
3.13 Master–Slave design used for PDD development. 55
3.14 An example of the parameters passed to PARMETIS in a three processor case Karypis et al. (2003). 66
3.15 Class Diagram: Major Container Classes for Data Redistribution 69

4.1 System Configuration of DataStar http://www.sdsc.edu/user_services/datastar/ 72
4.2 Example Finite Element Model of Soil-Foundation Interaction (Indication Only, Real Model Shown in Each Individual Section) 75
4.3 FE Models (1,968 Elements, 7,500 DOFs) for Studying Soil-Foundation Interaction Problems 76

4.4 FE Models (4,938 Elements, 17,604 DOFs) for Studying Soil-Foundation Interaction Problems 76
4.5 Partition and Repartition on 2 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs) . . . 77
4.6 Partition and Repartition on 2 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs) . . . 78
4.7 Partition and Repartition on 4 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs) . . . 78
4.8 Partition and Repartition on 4 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs) . . . 79
4.9 Partition and Repartition on 7 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs) . . . 79
4.10 Partition and Repartition on 7 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs) . . . 80
4.11 Partition and Repartition on 7 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (4,938 Elements, 17,604 DOFs) . . . 80
4.12 Partition and Repartition on 7 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (4,938 Elements, 17,604 DOFs) . . . 81
4.13 Timing Data of ITR Parametric Studies (1,968 Elements, 7,500 DOFs, Imbal. Tol. 5%) . 81
4.14 Relative Speedup of ITR=1e-3 over ITR=1e6 (1,968 Elements, 7,500 DOFs, Imbal. Tol. 5%) 82
4.15 Timing Data of ITR Parametric Studies (4,938 Elements, 17,604 DOFs, Imbal. Tol. 5%) . 82
4.16 4,035 DOFs Model, 2 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition 85
4.17 4,035 DOFs Model, 4 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition 85
4.18 4,035 DOFs Model, 8 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition 86
4.19 Timing Data of Parallel Runs on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5% 86
4.20 Absolute Speedup Data of Parallel Runs on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5% 87
4.21 Relative Speedup of PDD over Static DD on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5% 87
4.22 Finite Element Model of Soil-Foundation Interaction (4,938 Elements, 17,604 DOFs) . . . 88
4.23 Timing Data of Parallel Runs on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 89
4.24 Absolute Speedup Data of Parallel Runs on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 90
4.25 Relative Speedup of PDD over Static DD on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 91
4.26 4,938 Elements, 17,604 DOFs Model, 2 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 91

4.27 4,938 Elements, 17,604 DOFs Model, 4 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 92
4.28 4,938 Elements, 17,604 DOFs Model, 8 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 92
4.29 Finite Element Model of Soil-Foundation Interaction (9,297 Elements, 32,091 DOFs) . . . 93
4.30 Timing Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 94
4.31 Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 95
4.32 Relative Speedup of PDD over Static DD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 96
4.33 9,297 Elements, 32,091 DOFs Model, 3 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 96
4.34 9,297 Elements, 32,091 DOFs Model, 5 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 97
4.35 9,297 Elements, 32,091 DOFs Model, 7 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 97
4.36 9,297 Elements, 32,091 DOFs Model, 16 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 98
4.37 9,297 Elements, 32,091 DOFs Model, 32 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5% . . . 98
4.38 Timing Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20% . . . 100
4.39 Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20% . . . 101
4.40 Relative Speedup of PDD over Static DD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20% . . . 102
4.41 Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 105
4.42 Performance of Globally Adaptive PDD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 108
4.43 Performance of Globally Adaptive PDD on 20,476 Elements, 68,451 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 109
4.44 Scalability Study on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 110
4.45 Scalability Study on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5% . . . 111

4.46 Finite Element Model - 3 Bent SFSI, 56,481 DOFs, 13,220 Elements, Frequency Cutoff > 3Hz, Element Size 0.9m, Minimum G/G_{max} 0.08, Maximum Shear Strain γ 1% 112
4.47 Finite Element Model - 3 Bent SFSI, 484,104 DOFs, 151,264 Elements, Frequency Cutoff 10Hz, Element Size 0.3m, Minimum G/G_{max} 0.08, Maximum Shear Strain γ 1% 113
4.48 Finite Element Model - 3 Bent SFSI, 1,655,559 DOFs, 528,799 Elements, Frequency Cutoff 10Hz, Element Size 0.15m, Minimum G/G_{max} 0.02, Maximum Shear Strain γ 5% 113
4.49 Scalability Study on 3 Bent SFSI Models, DRM Earthquake Loading, Transient Analysis, ITR=1e-3, Imbal Tol 5%, Performance Downgrade Due to Increasing Network Overhead . 114
4.50 Relative Performance of PDD over DD, Shallow Foundation Model, Static Loading, ITR=1e-3, Imbal Tol 5% 115
4.51 Scalability of PDD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5% . 117
4.52 Relative Speedup of PDD over DD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5% 118
4.53 Full Range Scalability of PDD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5%, Performance Downgrade Due to Increasing Network Overhead 120

5.1 Finite Element Mesh of Soil-Structure Interaction Model 135
5.2 Finite Element Mesh of Soil-Structure Interaction Model 136
5.3 Matrices $N = 3336$ (Continuum FEM) 137
5.4 Matrices $N = 5373$ (Continuum FEM) 137
5.5 Matrices $N = 33081$ (Continuum FEM) 138
5.6 Matrices $N = 8842$ (Soil-Beam Static FEM) 138
5.7 Matrices $N = 8842$ (Soil-Beam Dynamic FEM) 139
5.8 Convergence of CG and PCG Method (3336 DOFs Model) 141
5.9 Convergence of CG and PCG Method (5373 DOFs Model) 141
5.10 Convergence of CG and PCG Method (33081 DOFs Model) 142
5.11 Convergence of CG and PCG Method (Soil-Beam Static Model) 143
5.12 Convergence of CG and PCG Method (Soil-Beam Dynamic Model) 143

6.1 Matrices $N = 33081$ (Continuum FEM) 153

List of Tables

4.1 Latency and Bandwidth Comparison (as of August 2004) 72
4.2 Technical Information of IA64 TeraGrid Cluster at SDSC 73
4.3 Material Constants for Soil-Foundation Interaction Model 74
4.4 Test Cases of Performance Studies 84
4.5 Observation on Load Imbalance Tolerance %5 103
4.6 Best Performance Observed for ITR=0.001, Load Imbalance Tolerance %5 121

5.1 Matrices in FEM Models 139
5.2 Performance of CG and PCG Method (Continuum FEM) 140
5.3 Performance of CG and PCG Method (Soil-Beam FEM) 144

6.1 Performance Study on SFSI Systems (N=33081) 154

Chapter 1

Introduction – Adaptive Parallel Inelastic Finite Element Simulations

1.1 Hypothesis

With the rapid development of computer software and hardware techniques, computing researchers nowadays have been exposed to a more demanding situation. More detailed and advanced models need to be developed and extensive parametric studies have to be conducted by computation in order to meet application requirements. Although the processing capability of the processor doubles every 18 months, perfectly following Moore's Law (Dongarra et al., 2003), unfortunately, the scaling of application performance has not matched the scaling of hardware peak speed. How to bring scalable performance into our applications has been addressed by many researchers in various fields.

Finite Element Method (FEM) is the most popular numerical method in computational mechanics. It has been successfully used to simulate soil-pile interaction problems by many researchers. In order to take advantage of advanced structural and geotechnical theories to simulate more realistic behaviors of soil-structure system, complete 3D finite element models have been developed in this report .

Inherently, advanced nonlinear elastic-plastic constitutive models, which are capable of simulating complicated system behaviors for either soil or structures, are computationally demanding. Sequential codes on a single CPU machine are not capable of handling detailed models and parallelism becomes a necessity to develop advanced realistic models which can meet the application requirements.

Many researchers have explored various approaches to parallelize sequential finite element codes. Generally speaking, these efforts can be regarded as follows: (Topping and Khan, 1996)

- The analysis domain may be physically divided into different subdomains and all subdomains will be subject to same instructions during finite element computations. This is so-called explicit domain decomposition approach, which is described as a *Divide and Conquer* algorithm.
- Alternatively, the system of equations for the whole analysis domain may be assembled and then

solved in parallel without recourse to a physical partitioning of the problem.

There are numerous techniques for domain decomposition analysis (McKenna, 1997), either non-overlapping or overlapping. Non-overlapping decomposition technique used in this research has naturally become more popular in finite element analysis due to its straightforward formulation. Substructuring (Przemieniecki, 1986), iterative substructuring and FETI (Farhat and Roux, 1991a) (*Finite Element Tearing and Interconnecting*) are among those most popular methods based on non-overlapping domain decomposition.

The most important feature for nonlinear finite element calculations is the iterations required to achieve equilibrium at both global and local computation levels. Elastic-plastic computation has been the most expensive part in nonlinear finite element analysis. For FEM, all solution procedures of practical importance are strongly rooted in the idea of advancing the solution by *continuation*. The basic idea is to follow the equilibrium response of the structure as the control and state parameters vary by small amounts, which is so-called *loading stages* for analysis. With each loading stage, iterative methods, such as Newton-Raphson, are employed to get numerical results within specified tolerance. This is so-called *global iterations*. *Increments* will be needed by iterative method to achieve convergence. In order to get material response compatible with specific nonlinear constitutive models, we finally need to carry out local level integration for classic incremental plasticity theory. Popular Backward Euler (Implicit) algorithm is always chosen, in which *local level iterations* are required to return the stress state on the yield surface (Jeremić, 2004b).

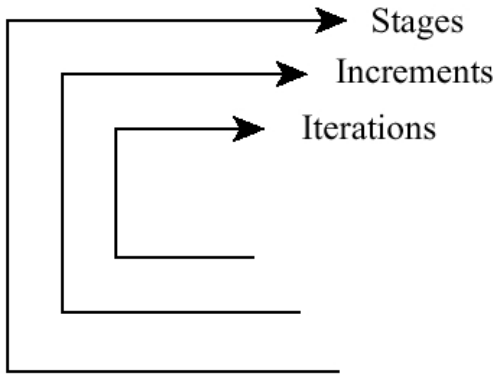


Figure 1.1: Nested Hierarchy in Nonlinear Solution Methods (Felippa, 2004)

In order to achieve high performance, parallelism has to be discovered for both global and local level iterations. Just one-step static domain decomposition on FE mesh graph at the very beginning of the analysis does not suffice for inelastic FE computation. Local level iterations can consume more than 70% of the total execution time and it imposes extra constraint on the graph partitioning algorithm.

A new *Plastic Domain Decomposition* (PDD) algorithm for inelastic finite element simulations is proposed in this report . This PDD algorithm aims at an adaptive finite element domain partitioning-repartitioning scheme to account for plasticity-induced local level load imbalance between processors.

1.2 Scope of Study

The aim of this work is to develop an efficient parallel finite element algorithm for nonlinear simulations. The proposed PDD algorithm is capable of detecting computational load imbalance as simulation advances. Adaptive graph partitioning will be triggered to guarantee even computational load distribution amongst processing units.

This report delivers a complete parallel implementation of OpenSees featuring the adaptive load balancing PDD algorithm. Object-Oriented principle has been strictly followed during software design and coding phases. Parallel functionality has been introduced and Object-Oriented design enables convenient future enhancement.

This report also includes the exploration of parallel equation solving in large scale SFSI problems. Popular parallel solvers have been tested against equation system from real world simulations.

Finally, the software tool developed in this report is used to study the dynamic behavior of a prototype SFSI system.

1.3 Summary of Contents

This report is divided into three parts: (I) theory and implementation, (II) parallel equation solving in finite element calculations and (III) parallel finite element modeling of SFSI. It consists of eight chapters and three appendices:

- **Chapter 1. Introduction – Adaptive Parallel Inelastic Finite Element Simulations** includes a brief discussion how inelastic parallel finite element calculations differ from linear cases. Graph partitioning as a powerful tool for parallel mesh-based scientific computing, is introduced in this chapter. The challenge of how to load balance inelastic finite element computations using graph partitioning algorithm is identified and the approach used in this report is previewed.

- **Chapter 2. Plastic Domain Decomposition Algorithm** introduces in details the multi-level multi-objective graph partitioning algorithm used in this report . The question of how to utilize graph partitioning algorithm in high performance mesh-based scientific computations is answered elaborately. This chapter also describes a unified graph-based algorithm to achieve load balance for parallel computations. The ParMETIS library provides fundamental functionality required by PDD algorithm. Implementation of this library is discussed and parametric studies have been performed to extract optimal number for the specific applications in this report .

- **Chapter 3. Object-Oriented Design of PDD** presents the complete implementation details of the proposed PDD algorithm. The Object-Oriented framework of OpenSees has been adopted as the foundation of proposed PDD algorithm. In order to maximize software reusing, the old sequential implementation of OpenSees has been introduced at the beginning of this chapter, followed by

the detailed discussion how parallel finite element implementation can be extended through an systematic Object-Oriented approach. C++ is solely used as the development language. Major class abstractions are discussed and the whole parallel finite element algorithm is pictured using activity diagrams. Interface implementation of OpenSees to ParMETIS is introduced and the element graph data structure is described in full details. The issue of how to properly design container class for data migration during parallel processing is also tackled in this chapter.

- **Chapter 4. Performance Studies on PDD Algorithm** shows the results of performance studies on the proposed PDD algorithm. The SFSI model used in performance studies is showed. IBM eServer and IA64 intel-based clusters are used to test the performance of the proposed PDD algorithm. The emphasis is to expose the advantage of PDD algorithm over the classic Domain Decomposition algorithm. Different model sizes and various number of CPUs have also been used to fully investigate scalability of the PDD algorithm. It has been shown that the naive implementation of PDD does not exhibit good scalability up to large number of CPUs. This chapter investigates the cause of deficiency and addresses several factors that contributed to the bad performance. A new globally adaptive algorithm has been added to the naive PDD implementation by introducing network communication costs and application-related overheads to the performance cost model. The performance and scalability of the newly improved algorithm has been investigated to show effectiveness of the improved algorithm.

- **Chapter 5. Application of Project-Based Iterative Methods in SFSI Problems** gives a brief overview of project-based iterative solvers. The algorithm details of some popular iterative solvers have been discussed. Preconditioning techniques are introduced to improve the convergence of CG and GMRES methods. Equation systems extracted from SFSI problems have been used to test the effectiveness of iterative solvers.

- **Chapter 6. Performance Study on Parallel Direct/Iterative Solving in SFSI** extends the iterative solving techniques to parallel equation system. Through the PETSc interface, consistent parallel equation solving is implemented as part of the proposed PDD parallel OpenSees package. As well as iterative solvers introduced in Chapter 5, popular parallel direct solvers such as MUMPS, SPOOLES and SuperLU_DIST are implemented in order to deliver the optimal equation solver for our prototype SFSI simulations.

- **Appendix A. Compilation of Parallel PDD on SMP-Clusters** describes some necessary steps to successfully compile PDD version of OpenSees on Linux/Unix clusters.

- **Appendix B. Import New Element/Material/Load etc. to PDD Version OpenSees** identifies the necessary functions for which users have to provide implementations in order to introduce new materials/element types to the parallel-ready PDD version of OpenSees.

- **Appendix C. Tcl Commands to Invoke Parallel Equation Solvers** shows example scripts to use the implemented parallel solvers.

1.4 Development Platform CONSOLID8

Developed parallel simulation program uses a number of numerical libraries. Namely. Graph partitioning is achieved using ParMETIS libraries (Karypis et al.). Parts of the OpenSees framework (McKenna, 1997) were used to connect the finite element domain. In particular, Finite Element Model Classes from OpenSees (namely, class abstractions Node, Element, Constraint, Load and Domain) where used to describe the finite element model and to store the results of the analysis performed on the model. In addition to that, an existing Analysis Classes were used as basis for development of parallel PDD framework which is then used to drive the global level finite element analysis, i.e., to form and solve the global system of equations in parallel. In addition to those, Actor, ShadowActor, Channel and other important constructs developed by McKenna (1997) were used as well (full description is available in this report). On a lower level, a set of Template3Dep numerical libraries (Jeremić and Yang Jeremić and Yang (2002)) were used for constitutive level integrations, nDarray numerical libraries (Jeremić and Sture Jeremić and Sture (1998)) were used to handle vector, matrix and tensor manipulations, while FEMtools element libraries from the UCD CompGeoMech toolset (Jeremić Jeremić (2004a)) were used to supply other necessary libraries and components. Parallel solution of the system of equations has been provided by PETSc set of numerical libraries (Balay et al. Balay et al. (2001, 2004, 1997)).

Part I

Theory and Implementation

Chapter 2

Plastic Domain Decomposition Algorithm

2.1 Introduction

Domain Decomposition approach is the most popular and effective method to implement parallel finite element method. The underlying idea is to physically divide the problem domain into subdomains and finite element calculations will be performed on each individual domain in parallel. Domain Decomposition can be overlapping or non-overlapping. The overlapping domain decomposition method divides the problem domain into several slightly overlapping subdomains. Non-overlapping domain decomposition is extensively used in continuum finite element modeling due to the relative ease to program and organize computations and is the one that will be examined in this report .

In general, a good non-overlapping decomposition algorithm should be able to

- handle irregular mesh of arbitrarily shaped domain.
- minimize the interface problem size by delivering minimum boundary connectivity, which will help reducing the communication overheads.

The well-known idea of domain decomposition method can be found in a 1870 paper by the father of domain decomposition, H.A. Schwarz (Rixena and Magoulès, 2007). Domain decomposition method is also the underlying paradigm of substructuring methods developed in the sixties, which aim at reducing the dimension of models in structural analysis by applying static condensation-type techniques to subdomains.

Other than static condensation, Farhat and Roux (1991b); Farhat (1991); Farhat and Geradin (1992) proposed FETI (Finite Element Tearing and Interconnecting) method for domain decomposition analysis. In FETI method, Lagrange multipliers are introduced to enforce compatibility at the interface nodes. Rigid body modes are eliminated in parallel from each local problem and a direct scheme is applied concurrently to all subdomains in order to recover each partial local solution. The contributions of these modes are then related to the Lagrange multipliers through an orthogonality condition. This FETI method has been shown that it can deliver high efficiency for parallel implicit transient simulations in structural mechanics (Crivelli and Farhat, 1993).

Domain decomposition itself has become a active topic as parallel processing techniques receive much more attention in mathematics and engineering world during recent years. Domain decomposition was revived as a natural paradigm for parallel solvers (Rixena and Magoulès, 2007). Many papers have discussed two algorithms that are currently receiving much research effort, namely the FETI-DP (or Dual Primal Finite Element Tearing and Interconnecting) method and the even more recent BDDC (or Balancing Domain Decomposition by Constraints).

FETI-DP is the third generation FETI method (Bavestrello et al., 2007) developed for the fast, scalable, and domain-decomposition-based iterative solution of symmetric systems of equations arising from the finite element (FE) discretization of static, dynamic, structural and acoustic problems (Farhat et al., 2001, 2000).

BDDC, on the other hand, derives its formulation from substructuring method by enforcing constraints associated with disjoint sets of nodes on substructure boundaries using constrained energy minimization concepts (Dohrmann, 2003; Mandel and Dohrmann, 2003).

An early endeavor on dynamic computational load balancing was presented by McKenna (1997). Limited number of examples show that run time, dynamic computational load balancing can indeed improve parallel program performance in some cases, particularly when nonlinearities are involved.

Although many works have been presented on domain decomposition methods, the most popular methods such as FETI-type and BDDC all stem from the root of subdomain interface constraints handling. The merging of iterative solving with domain decomposition-type preconditioning is promising as shown by many researchers (Pavarino, 2007; Li and Widlund, 2007). Schwartz-type preconditioners for parallel domain decomposition system solving have also shared part of the spotlight (Hwang and Cai, 2007; Sarkis and Szyld, 2007).

In solid finite element methods, it has been assumed that the equation solving is the most computational expensive part so it is totally reasonable that all focus has been set on equation solver during the past decades.

Work presented in this report , however, has originated from the observation that for highly nonlinear materials, the constitutive level computation can be at least equally costly as equation solving, if not more expensive. The novelty of this report is to break out of the existing substructuring or FETI frameworks to further address the fundamental load balance issue of parallel computing. Namely, in order to achieve better parallel performance, we want to keep all processors equally busy. Load imbalance issue resulted from nonlinear constitutive level computations is too important to be neglected. This report proposes the Plastic Domain Decomposition algorithm which focuses on adaptive load balancing operation for nonlinear finite elements.

From the implementation point of view, for mesh-based scientific computations, domain decomposition corresponds to the problem of mapping a mesh onto a set of processors, which is well defined as a graph partitioning problem (Schloegel et al., 1999).

Formally, the graph partitioning problem is as follows. Given a weighted, undirected graph $G = (V; E)$

for which each vertex and edge has an associated weight, the k-way graph partitioning problem is to split the vertices of V into k disjoint subsets (or subdomains) such that each subdomain has roughly an equal amount of vertex weight (referred to as the balance constraint), while minimizing the sum of the weights of the edges whose incident vertices belong to different subdomains (i. e., the edge-cut).

In computational solid mechanics, the element graph is naturally used in parallel finite element method due to the fact that elemental operation forms the basis of finite element method. On the other hand, for material nonlinearity simulations, the element calculations represent the most computationally expensive part. In order to facilitate consistent interfaces for computational load measuring and data migration, element graph has been utilized as fundamental graph structure in this report , although it has been shown that the node-graph can be used as well for structure dynamics problem and the element-cut partitioning can make certain algorithms simpler (Krysl and Bittnar, 2001).

The graph partitioning problem is known to be NP-complete [1]. Therefore, generally it is not possible to compute optimal partitioning for graphs of interesting size in a reasonable amount of time. Various heuristic approaches have been developed, which can be classified as either geometric, combinatorial, spectral, combinatorial optimization techniques, or multilevel methods (Dongarra et al., 2003).

In finite element simulations involving nonlinear material response, static graph partitioning mentioned above does not guarantee even load distribution among processors. Plastification introduces work load that is much heavier than pure elastic computation. So for this kind of multiphase simulation, adaptive computational load balancing scheme has to be considered to keep all processing units *equally* busy as much as possible. Traditional static graph partitioning algorithm is not adequate to do multiphase partition/repartitioning. A parallel multilevel graph partitioner has been introduced in this research to achieve dynamic load balancing for inelastic finite element simulations.

In this chapter, the algorithm of Plastic Domain Decomposition (PDD) is proposed. The adaptive multi-level graph partitioning kernel of the PDD algorithm is implemented through the ParMETIS interface. Studies are performed to extract optimal algorithmic parameters for our specific applications.

2.2 Inelastic Parallel Finite Element

The distinct feature of inelastic (elastic-plastic) finite element computations is the presence of two iteration levels. In a standard displacement based finite element implementation, constitutive driver at each Gauss point iterates in stress and internal variable space, computes the updated stress state, constitutive stiffness tensor and delivers them to the finite element functions. Finite element functions then use the updated stresses and stiffness tensors to integrate new (internal) nodal forces and element stiffness matrix. Then,

[1] The complexity class NP is the set of decision problems that can be solved by a non-deterministic Turing machine in polynomial time. the NP-complete problems are the most difficult problems in NP ("non-deterministic polynomial time") in the sense that they are the smallest subclass of NP that could conceivably remain outside of P, the class of deterministic polynomial-time problems. (http://en.wikipedia.org/wiki/NP-complete)

on global level, nonlinear equations are iterated on until equilibrium between internal and external forces is satisfied within some tolerance.

- **Elastic Computations**

  In the case of elastic computations constitutive driver has a simple task of computing increment in stresses ($\Delta\sigma_{ij}$) for a given deformation increment ($\Delta\epsilon_{kl}$), through a closed form equation ($\Delta\sigma_{ij} = E_{ijkl}\Delta\epsilon_{kl}$) It is important to note that in this case the amount of work per Gauss point is known in advance. The amount of computational work is the same for every integration point. If we assume the same number of integration points per element, it follows that the amount of computational work is the **same** for each element and it is **known in advance**.

- **Elastic-Plastic Computations**

  On the other hand, for elastic-plastic problems, for a given incremental deformation the constitutive driver is iterating in stress and internal variable space until consistency condition is satisfied ($F = 0$). The number of iterations is not known in advance. Initially, all Gauss points are in elastic range, but as we incrementally apply loads, the plastic zones develop. For Gauss points still in elastic range, there are no iterations, the constitutive driver just computes incremental stresses from closed form solution. Computational load will increase significantly for integration of constitutive equations in plastic range. In particular, constitutive level integration algorithms for soils, concrete, rocks, foams and other granular materials are very computationally demanding. More than 70% of wall clock time during an elastic-plastic finite element analysis is spent in constitutive level iterations. This is in sharp contrast with elastic computations where the dominant part is solving the system of equations which consumes about 80% of run time. The extent of additional, constitutive level iterations is not known before the actual computations are over. In other words, the extent of elastic-plastic domain is not known ahead of time.

  The traditional preprocessing type of Domain Decomposition method (also known as topological DD) splits domain based on the initial geometry and assigns roughly the same number of elements to every computational node and minimizes the size of subdomain boundaries. This approach might result in serious computational load imbalance for elastic-plastic problems. For example one domain might be assigned all of the elastic-plastic elements and spend large amount of time in constitutive level iterations. The other domains will have elements in elastic state and thus spend far less computational time in computing stress increments. This results in program having to wait for the slowest domain (the one with large number of elastic-plastic finite elements) to complete constitutive level iterations and only proceed with global system iterations after that.

This illustrates a two-fold challenge with computational load balancing for inelastic simulations in mechanics. These two challenges is described below in some more detail.

2.2.1 Adaptive Computation

First, these computations are dynamic in nature. That is, the structure of elastic and elastic-plastic domains changes dynamically and unpredictably during the course of the computation. For this reason, a static decomposition computed as a pre-processing step is not sufficient to ensure the computational load-balance of the entire computation. Instead, periodic computational load-balancing is required during the course of the computation. The problem of computing a dynamic decomposition shares the same requirements as that of computing the initial decomposition (i.e., balance the mesh elements and minimize the inter-processor communications), while also requiring that the cost associated with redistributing the data in order to balance the computational load is minimized. This last requirement prevents us from simply computing a whole new static partitioning from scratch each time computational load-balancing is required.

Often, the objective of minimizing the data redistribution cost is at odds with the objective of minimizing the inter-processor communications. For applications in which the computational requirements of different regions of the domain change rapidly, or the amount of state associated with each element is relatively high, minimizing the data redistribution cost is preferred over minimizing the communications incurred during parallel processing.

For applications in which computational load-balancing occurs very infrequently, the key objective of a load-balancing algorithm is in obtaining the minimal inter-processor communications. For many application domains, it is straightforward to select a primary objective to minimize (i.e., minimize whichever cost dominates). However, one of the key issues concerning the elastic-plastic computation is that the number of iterations between computational load-balancing phases is both unpredictable and dynamic. For example, in the case of static problems, zones in the 3D solid may become plastic and then unload to elastic (during increments of loading) so that the extent of plastic zone is changing. The change can be both slow and rapid. Slow change usually occurs during initial loading phases, while the later deformation tends to localize in narrow zones rapidly and the rest of the solid unloads rapidly (becomes elastic again). The narrow, localized zone has heavy computational load on the constitutive level (in each integration point within elements). Similar phenomena is observed in seismic soil-structure interaction computations where stiff structure interacts with soft soil and elastic and elasto-plastic zones change significantly during loading cycles. In this type of computation, it is extremely difficult to select the type of computational load-balancing algorithm to employ. Furthermore, the preferred computational load-balancing algorithm is liable to change during the course of the computation, and so the selection must be made dynamically.

2.2.2 Multiphase Computation

The second challenge associated with computational load-balancing elastic-plastic computations in geomechanics is that these are two-phase computations. That is, elastic-plastic computations follow up the elastic computations. There is a synchronization phase between the computations, as only after the elastic

computation is finished is it possible to check if the elastic-plastic computation is required for a given integration (Gauss) point within an element. For regions of the mesh in which this check indicates that the elastic-plastic computation is necessary, lengthy elastic-plastic computations are then performed. The existence of the synchronization step between the two phases of the computation requires that each phase be individually load balanced. That is, it is not sufficient to simply sum up the relative times required for each phase and to compute a decomposition based on this sum. Doing so may lead to some processors having too much work during the elastic computation (and so, these may still be working after other processors are idle), and not enough work during the elastic-plastic computation, (and so these may be idle while other processors are still working), and vice versa. Instead, it is critical that every processor have an equal amount of work from both of the phases of the computation.

2.2.3 Multiconstraint Graph Partitioning

Elastic-plastic FE computation can be understood as a two-phase calculation, which is also dynamic in nature. Traditional graph partitioning formulations are not adequate to ensure its efficient execution on high performance parallel computers. In this report very recent progresses from the graph partitioning algorithm research will be investigated. We need new adaptive graph partitioning formulations, which can compute adaptive partitioning-repartitionings that can satisfy an arbitrary number of balance constraints.

- Static Graph Partitioning
  Given a weighted, undirected graph $G = (V, E)$, for which each vertex and edge has an associated weight, the k-way graph partitioning problem is to split the vertices of V into k disjoint subsets (or *subdomains*) such that each subdomain has roughly an equal amount of vertex weight (referred to as the *balance constraint*), while minimizing the sum of the weights of the edges whose incident vertices belong to different subdomains (i.e., the edge-cut).

  1. Geometric Techniques
     Compute partitioning based solely on the coordinate information of the mesh nodes, without considering edge-cut. Popular methods include, *Coordinate Nested Dissection* (CND or Recursive Coordinate Bisection), *Recursive Inertial Bisection* (RIB), *Space-Filling Curve* techniques and *Sphere-Cutting* approach.
  2. Combinatorial Techniques
     Attempt to group together highly connected vertices whether or not these are near each other in space. That is combinatorial partitioning schemes compute a partitioning based only on the adjacency information of the graph; they do not consider the coordinates of the vertices. They tend to have lower edge-cuts but generally slower. Popular methods include, *Levelized Nested Dissection* (LND) and *Kernighan-Lin/Fiduccia-Mattheyses* (KL/FM) partitioning refinement algorithm, which needs an initial partition input to do swapping refinement.

3. Multilevel Schemes

   The multilevel paradigm consists of three phases: graph coarsening, initial partitioning, and multilevel refinement. Firstly, we form coarse graph by collapsing together selected vertices of the input graph. After rounds of coarsening, we get coarsest graph, on which an initial bisection will be performed. Then the KL/FM algorithm can be used to refine the partition back to the finest graph.

   The multilevel paradigm works well for two reasons. First, a good coarsening scheme can hide a large number of edges on the coarsest graph, which makes the task of computing high-quality partitioning easier. Second reason, incremental refinement schemes such as KL/FM become much more powerful in the multilevel context.

   Popular algorithms include *Multilevel Recursive Bisection* and *Multilevel k-Way Partitioning*.

- Adaptive Graph Partitioning

  For large scale elasto-plastic FE simulations, it is necessary to dynamically load-balance the computations as the analysis progresses due to unpredictable plastification inside the domain. This dynamic load balancing can be achieved by using a graph partitioning algorithm.

  Adaptive graph partitioning shares most of the requirements and characteristics of static graph partitioning but also adds an additional objective. That is, the amount of data that needs to be redistributed among the processors in order to balance the load should be minimized. If the vertex weight represents the computational cost of the work carried by the vertex, another metric, *size* of the vertex needs to be considered as well, which reflects distribution cost of the vertex. Thus, the repartitioner should attempt to balance the partitioning with respect to vertex weight while minimizing vertex migration with respect to vertex size.

  Different approaches are available. One can simply compute a new graph from scratch, so called *Scratch-Remap Repartitioner*, which expectedly introduces more data redistribution than necessary. *Diffusion-Based Repartitioner* attempt to minimize the difference between the original partitioning and the final repartitioning by making incremental changes in the partitioning to restore balance. This method has been an very active topic during recent years, Dongarra et al. (2003) gives up-to-date review.

- Multiconstraint Graph Partitioning

  We can see traditional graph partitioning typically balances only a single constraint (i.e., the vertex weight) and minimizes only a single objective (i.e., the edge-cut). If we replace the vertex weight, which is a single number, with a weight vector of size m, then the problem becomes that of finding a partitioning that minimizes the edge-cuts subject to the constraints that each of the m weights is balanced across subdomains.

  Multilevel graph partitioning algorithms for solving multiconstraint/multiobjective problems have

been very successful Schloegel et al. (1999). The software libraries METIS and ParMETIS are widely used in computational mechanics research.

2.2.4 Adaptive PDD Algorithm

In this report , the Plastic Domain Decomposition (PDD) has been developed using multi-level, multi-objective graph partitioning algorithm. This algorithm automatically monitors load balancing condition and updates element graph structure accordingly as the simulation progresses. Element redistribution will be triggered to achieve load balance when nonlinearity of materials brings down the parallel performance.

2.3 Adaptive Multilevel Graph Partitioning Algorithm

Kaypis and Kumar (1998) present a k-way multilevel partitioning algorithm whose run time is linear in the number of edges $|E|$ (i.e., O($|E|$)); whereas the run time of multilevel recursive bisection schemes is O($|E|$log$k$) for k-way partitioning. Kaypis and Kumar (1998) show that the proposed multilevel partitioning scheme produces partitioning that are of comparable or better quality than those produced by multilevel recursive bisection, while requiring substantially less time. This paradigm consists of three phases: graph coarsening, initial partitioning, and multilevel refinement. In the graph coarsening phase, a series of graphs is constructed by collapsing together selected vertices of the input graph in order to form a related coarser graph. This newly constructed graph then acts as the input graph for another round of graph coarsening, and so on, until a sufficiently small graph is obtained. Computation of the initial bisection is performed on the coarsest (and hence smallest) of these graphs, and so is very fast. Finally, partition refinement is performed on each level graph, from the coarsest to the nest (i.e., original graph) using a KL/FM-type algorithm Dongarra et al. (2003). Figure 2.1 illustrates the multilevel paradigm. This algorithm is available in METIS Karypis and Kumar (1998b) which is used in this research to provide initial static partitioning.

Adaptive graph repartitioning algorithm can be used to achieve dynamic load balancing of multiphase elastic-plastic finite element simulations. Adaptive graph partitioning differs from static graph partitioning algorithm in the sense that *one additional objective* has to be targeted. That is, the amount of data the needs to be redistributed among the processors in order to balance the load should be minimized. In order to measure this redistribution cost, not only does the weight of a vertex, but also its *size* have to be considered. In our implementation for the purpose of this research, the vertex weight represents the computational load of each finite element, while the size reflects its redistribution cost. Thus, the application of adaptive graph partitioning algorithm aims at balancing the partitioning with respect to vertex weight while minimizing vertex migration with respect to vertex size.

A repartitioning of a graph can be obtained simply by partitioning a new graph from a scratch, which tends to bring much more unnecessary communications because the old distribution has not been taken into account. Diffusion-based Repartitioner is more popular in which one attempts to minimize the dif-

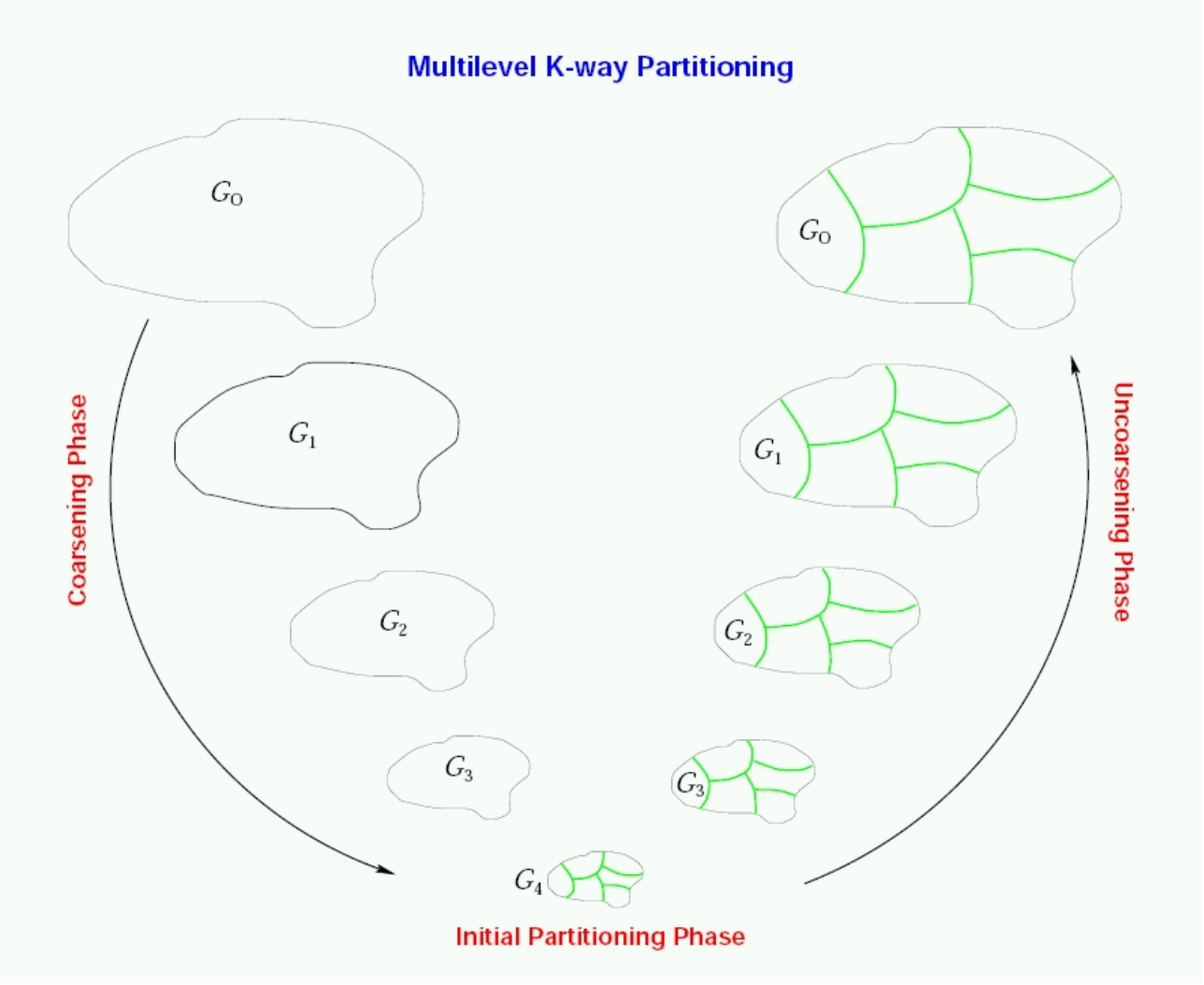


Figure 2.1: Multilevel Graph Partitioning Scheme Karypis et al. (2003)

ference between the original partitioning and the final repartitioning by making incremental changes in the partitioning to restore balance. Dongarra et al. (2003) gives a comprehensive review on this subject. Adaptive repartitioning is available in ParMETIS Karypis et al. (2003) and Jostle Warshaw (1998). The former is chosen in this research considering the fact that ParMETIS provides seamless interface for METIS 4.0 which makes the comparison between static and adaptive partitioning schemes more consistent.

PARMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs and for computing fill-reducing orderings of sparse matrices Karypis et al. (2003). PARMETIS is particularly suited for parallel numerical simulations involving large unstructured meshes. In this type of computation, PARMETIS dramatically reduces the time spent in communication by computing mesh decompositions such that the numbers of interface elements are minimized. The algorithms in PARMETIS are based on the multilevel partitioning and fill-reducing ordering algorithms that are implemented in the widely-used serial package METIS Karypis and Kumar (1998a). However, PARMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel computations and large-scale numerical simulations. In particular, PARMETIS provides the following functionality Karypis et al. (2003):

- Partition unstructured graphs and meshes.
- Repartition graphs that correspond to adaptively refined meshes.
- Partition graphs for multi-phase and multi-physics simulations.
- Improve the quality of existing partitioning.
- Compute fill-reducing orderings for sparse direct factorization.
- Construct the dual graphs of meshes.

Both METIS and PARMETIS are used in this research. METIS routines are called to construct static partitioning for commonly used one-step static domain decomposition, while adaptive load-balancing is achieved by calling PARMETIS routines regularly during the progress of nonlinear finite element simulations.

Adaptive load-balancing through domain repartitioning is a multi-objective optimization problem, in which repartitionings should minimize both the inter-processor communications incurred in the iterative mesh-based computation and the data redistribution costs required to balance the load. PARMETIS provides the routine *ParMETIS_V3_AdaptiveRepart* for repartitioning the previous unbalanced computational domain. This routine assumes that the existing decomposition is well distributed among the processors, but that (due to plastification of certain nonlinear elements) this distribution is poorly load balanced.

Figure 2.2 Schloegel et al. (2000) shows common steps involved in the execution of adaptive mesh-based simulations on parallel computers. Initially, the mesh is equally distributed on different processors. As all elements are elastic at the very beginning (carrying the same amount of elemental calculation

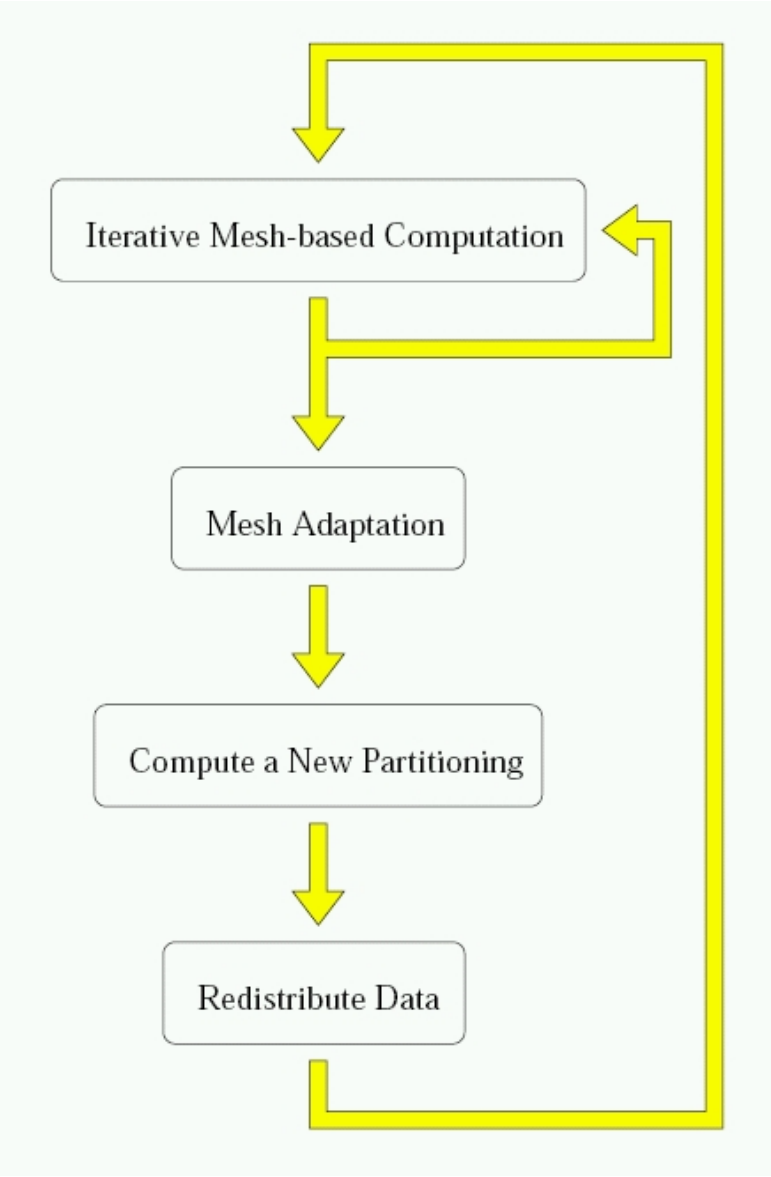


Figure 2.2: A diagram illustrating the execution of adaptive scientific simulations on high performance parallel computers Schloegel et al. (1999)

work), computation load balance can be guaranteed with a even distribution. A number of iterations of the simulation are performed in parallel, after which plasticity occurs in certain nonlinear elements thus introducing some amount of load imbalance. A new partitioning based on the unbalanced domain is computed to re-balance the load, and then the mesh is redistributed among the processors, respectively. The simulation can then continue for another number of iterations until either more mesh adaptation is required or the simulation terminates.

If we consider each round of executing a number of iterations of the simulation, mesh adaptation, and load-balancing to be an *epoch*, then the run time of an epoch can be described by, Schloegel et al. (2000)

$$(t_{comp} + f(|E_{cut}|))n + t_{repart} + g(|V_{move}|) \tag{2.1}$$

where n is the number of iterations executed, t_{comp} is the time to perform the computation for a single iteration of the simulation, $f(|E_{cut}|)$ is the time to perform the communications required for a single iteration of the simulation, and t_{repart} and $g(|V_{move}|)$ represent the times required to compute the new partitioning and to redistribute the data. Here, the inter-processor communication time is described as a function of the edge-cut of the partitioning and the data redistribution time is described as a function of the total amount of data that is required to be moved in order to realize the new partitioning. Adaptive repartitioning affects all of terms in Equation 2.1. How well the new partitioning is balanced influences t_{comp}. The inter-processor communications time is dependent on the edge-cut of the new partitioning. The data redistribution time is dependent on the total amount of data that is required to be moved in order to realize the new partitioning. It is critical for adaptive partitioning schemes to minimize both the edge-cut and the data redistribution when computing the new partitioning. Viewed in this way, adaptive graph partitioning is a multi-objective optimization problem.

There are various approaches how to handle this dual-objective problem. In general, two approaches have primarily been taken when designing adaptive partitioners. Schloegel et al. (2000) gives a comprehensive review on this topic. The first approach is to attempt to focus on minimizing the edge-cut and to minimize the data redistribution only as a secondary objective. This family of methods can be called *scratch-remap* repartitioner. These use some type of state-of-the-art graph partitioner to compute a new partitioning from scratch and then attempt to intelligently remap the subdomain labels to those of the original partitioning in order to minimize the data redistribution costs. Since a state-of-the-art graph partitioner is used to compute the partitioning, the resulting edge-cut tends to be extremely good. However, since there is no guarantee as to how similar the new partitioning will be to the original partitioning, data redistribution costs can be high, even after remapping. The second approach is to focus on minimizing the data redistribution cost and to minimize the edge-cut as a secondary objective, or so-called *diffusion-based* repartitioner. These schemes attempt to perturb the original partitioning just enough so as to balance it. This strategy usually leads to low data redistribution costs, especially when the partitioning is only slightly imbalanced. However, it can result in higher edge-cuts than scratch-remap methods because perturbing a partitioning in order to balance it also tends to adversely affect its quality.

These two types of repartitioner allow the user to compute partitioning that focus on minimizing either the edge-cut or the data redistribution costs, but give the user only a limited ability to control the tradeoffs among these objectives. This control of the tradeoffs is sufficient if the number of iterations that a simulation performs between load-balancing phases (i.e. the value of n in Equation 2.1) is either very high or very low. However, when n is neither very high nor very low, neither type of scheme precisely minimizes the combined costs of $f(E_{cut}|)n$ and $g(|V_{move}|)$. Another disadvantage exists for applications in which n is difficult to predict or those in which n can change dynamically throughout the course of the computation. As an example, one of the key issues concerning the elastic-plastic soil-structure interaction computations required for earthquake simulation is that the number of iterations between load-balancing phases is both unpredictable and dynamic. Here, zones in the 3D solid may become plastic and then unload (during increments of loading) so that the extent of the plastic zone is changing. The change can be both slow and rapid. Slow change usually occurs during initial loading phases, while the later deformation tends to localize in narrow zones rapidly and the rest of the solid unloads rapidly (becomes elastic again) Jeremić and Xenophontos (1999).

Schloegel et al. (2000) presents a parallel adaptive repartitioning scheme (called the *Unified Repartitioning Algorithm*) for the dynamic load-balancing of scientific simulations that attempts to solve the precise multi-objective optimization problem. By directly minimizing the combined costs of $f(E_{cut}|)n$ and $g(|V_{move}|)$, the proposed scheme is able to gracefully tradeoff one objective for the other as required by the specific application. The paper shows that when inter-processor communication costs are much greater in scale than data redistribution costs, the proposed scheme obtains results that are similar to those obtained by an optimized scratch-remap repartitioner and better than those obtained by an optimized diffusion-based repartitioner. When these two costs are of similar scale, the scheme obtains results that are similar to the diffusive repartitioner and better than the scratch-remap repartitioner. When the cost to perform data redistribution is much greater than the cost to perform inter-processor communication, the scheme obtains better results than the diffusive scheme and much better results than the scratch-remap scheme. They also show in the paper that the *Unified Repartitioning Algorithm* is fast and scalable to very large problems.

2.3.1 Unified Repartitioning Algorithm

A key parameter used in Unified Repartitioning Algorithm (URA) is the *Relative Cost Factor* (RCF). This parameter describes the relative times required for performing the inter-processor communications incurred during parallel processing and to perform the data redistribution associated with balancing the load. Using this parameter, it is possible to unify the two minimization objectives of the adaptive graph partitioning problem into the unified cost function

$$|E_{cut}| + \alpha|V_{move}| \tag{2.2}$$

where α is the Relative Cost Factor, $|E_{cut}|$ is the edge-cut of the partitioning, and $|V_{move}|$ is the total amount of data redistribution. The Unified Repartitioning Algorithm attempts to compute a repartitioning

while directly minimizing this cost function.

The Unified Repartitioning Algorithm is based upon the multilevel paradigm that is illustrated in Figure 2.1, which can be described as three phases: graph coarsening, initial partitioning, and uncoarsening/refinement Schloegel et al. (2000). In the graph coarsening phase, coarsening is performed using a purely local variant of heavy-edge matching. That is, vertices may be matched together only if they are in the same subdomain on the original partitioning. This matching scheme has been shown to be very effective at helping to minimize both the edge-cut and data redistribution costs and is also inherently more scalable than global matching schemes.

2.3.2 Study of ITR in ParMETIS

The RCF in the URA implementation controls the tradeoff between two objectives, minimizing data redistribution cost or edge-cut. In our application, ParMETIS library has been linked to updated OpenSees analysis model to facilitate the partitioning/adaptive repartitioning scheme. The RCF is defined as a single parameter ITR in ParMETIS Karypis et al. (2003). This parameter describes the ratio between the time required for performing the inter-processor communications incurred during parallel processing compared to the time to perform the data redistribution associated with balancing the load. As such, it allows us to compute a single metric that describes the quality of the repartitioning, even though adaptive repartitioning is a multi-objective optimization problem. As recommended by Karypis et al. (2003), appropriate values to pass for the ITR Factor parameter can be determined depending on the times required to perform

1. all inter-processor communications that have occurred since the last repartitioning, and

2. the data redistribution associated with the last repartitioning/load balancing phase.

Simply divide the first time measurement by the second time measurement. The result is the correct ITR Factor. In case these times cannot be ascertained (e.g., for the first repartitioning/load balancing phase), Karypis et al. (2003) suggests that values between 100 and 1000 work well for a variety of situations. By default ITR is between 0.001 and 1000000. If ITR is set high, a repartitioning with a low edge-cut will be computed. If it is set low, a repartitioning that requires little data redistribution will be computed.

Chapter 3

Object-Oriented Design of PDD

3.1 Introduction

In this report , proposed PDD algorithm has been implemented by reworking (improving, updating) an existing sequential OpenSees framework (**?**). At the beginning of this chapter, the Object-Oriented approach to programming the Finite Element Method is reviewed based on the existing (as of 2005) implementation of OpenSees. Object-Oriented parallel design is then extended from the existing framework. Parallel algorithm adopts Master-Slave paradigm and the new design of data structures have strictly followed the Object-Oriented principle using C++ language. External utility libraries such as ParMETIS and PETSc have been incorporated to provide seamless parallel numerical manipulations including partitioning/repartitioning and equation solving.

In this chapter, the algorithm overview will be presented first. Then the implementation details in C++ will follow. The challenges of achieving load balancing in parallel Finite Element simulation have been divided into two parts, global level equation solving and constitutive level iterations. This research presents the PDD algorithm to demonstrate how to balance each stage systematically in applications.

3.2 Object-Oriented Parallel Finite Element Algorithm

Parts of OpenSees software framework has been used in this report . Object-Oriented design of OpenSees enables software reuse that greatly shortens the development life cycle of application codes.

OpenSees is comprised of a set of classes and objects that represent models perform computations for solving the governing equations, and provide access to processing results. There are four types of class objects in OpenSees McKenna (1997).

- *Modeling Classes* are used to create the *Finite Element Model Classes* for a given problem.
- *Finite Element Model Classes* are used to describe the finite element model and to store the results of the analysis performed on the model. Main class abstractions used in OpenSees are **Node**, **Element**,

Constraint, **Load** and **Domain**. The relationship amongst these classes can be shown using the class diagram Figure 3.2 using the Rumbaugh notation as shown in Figure 3.1 Rumbaugh et al. (1991).

- *Analysis Classes* are used to perform the finite element analysis, i.e., to form and solve the global system of equations
- *Numerical Classes* are used to handle numerical operations in the solution procedure. Also included in this category are data structure classes such as **Vector**, **Matrix** and **Tensor**.

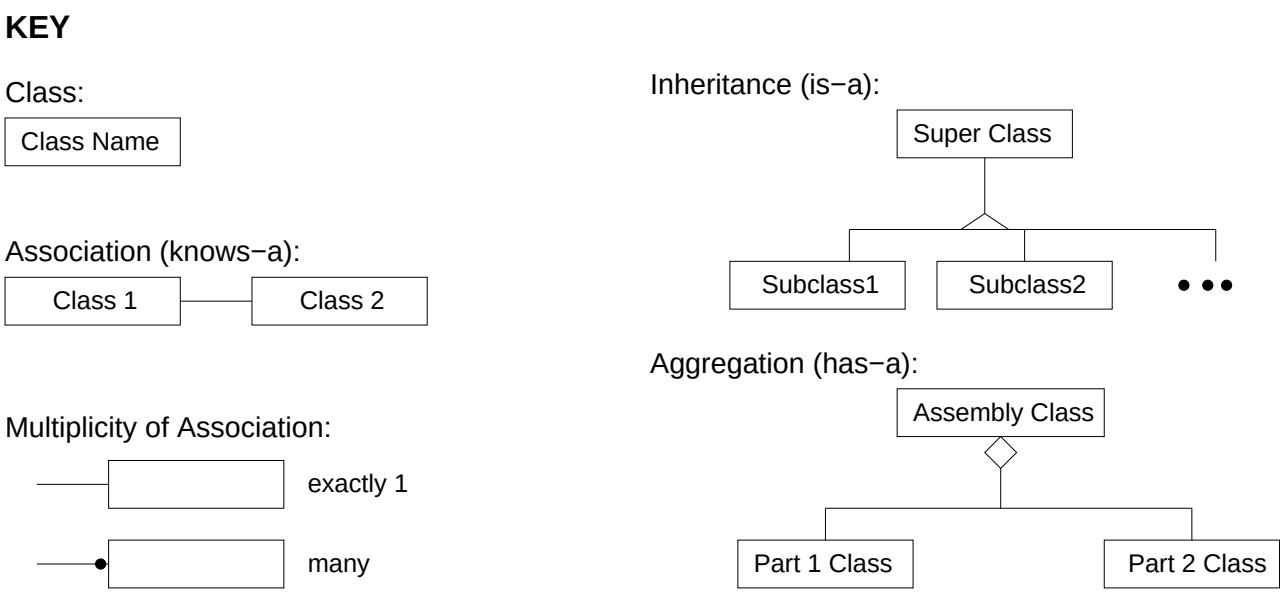


Figure 3.1: Rumbaugh Notation of-Object Oriented Design

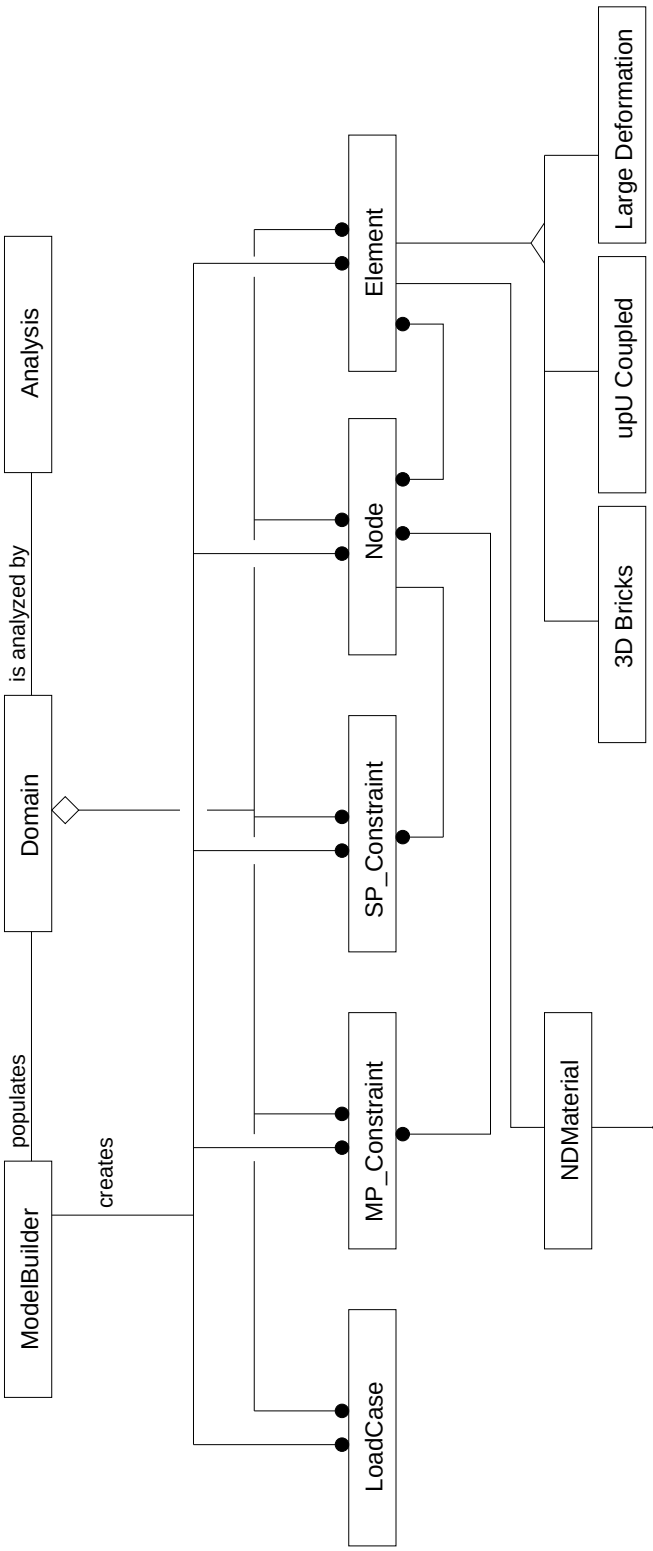


Figure 3.2: Class Diagram of Finite Element Model Classes

3.2.1 Modeling Classes

The modeling classes are responsible of creating the necessary components of the finite element model, such as nodes, elements, loads and constraints. There are a number of approaches proposed by various researchers. In some works, the user has the responsibility to create the finite element model in a single driver-type file Ross et al. (1992); Zeglinski et al. (1994); Cardona et al. (1994). In other works, an input file containing the model data is used to be read by the main program to create the model Forde et al. (1990); Dubois-Pelerin et al. (1992); Dubois-Pelerin and Zimmermann (1993); Menéntrey and Zimmermann (1993). Graphical interface for building models visually has also been proposed Ostermann et al. (1995); Mackie (1995).

In this research, the existing **ModelBuilder** interface class is reused to facilitate the finite element model construction. As shown in Figure 3.2, the **ModelBuilder** is associated with a single finite element **Domain** object. The interface (pure virtual function) *buildFE_Model()* must be redefined depending on the specific type of finite element model users want to build.

In parallel processing, **PartitionedModelBuilder** is used instead, in which the building process includes higher level control of building **Subdomain**s from the **PartitionedDomain**. For each **Subdomain**, the **PlaneFrameModelBuilder**-type is invoked to build the finite element model on each **Subdomain**.

The Object-Oriented interface design of **ModelBuilder** through pure virtual function provides a consistent framework from which all kinds of engineering model can be readily extended.

3.2.2 Finite Element Model Class

In most of the works presented, main class abstractions used to describe a 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); Menéntrey and Zimmermann (1993); Pidaparti and Hudli (1993); Cardona et al. (1994); Chudoba and Bittnar (1995); Zahlten et al. (1995); Rucki and Miller (1996).

Node

The most important feature of the **Node** class is the associativity with **DOF** class which contains the degree of freedoms of any specific instance of the **Node** class. The response quantities such as displacements of each **DOF** object will be stored in the **Node** class. Routines are available to set/get those solution quantities.

Element

The functionality of an **Element** object is to provide the tangent stiffness, mass and the residual force corresponding to current loadings. **Element** class contains reference to its associated **Node** objects.

Element class is one of the most fundamental abstractions in Object-Oriented finite element software design. In this research, **Element** also acts as a container for material models, which is critical for simulations with nonlinear materials. Chudoba and Bittnar (1995) proposed a **MaterialPoint** object which is associated with **GaussPoint** object. In Zahlten et al. (1995), class abstractions such as cross section, material point, material law, yield surface, hardening rule and flow rule are introduced to model complicated materials within the **Element** class in an Object-Oriented flavor.

Jeremić and Yang (2002) present the complete formulation of **Template3Dep** material class, which is wrapped inside the **Element** class to enable a consistent interface for complex elastic-plastic material modeling.

Constraint

There are two types of constraints in finite element simulations,

1. Single-Point constraints, which are applied to a specific **DOF** object;
2. Multi-Point constraints, which describe the relationship between more than one **DOF** objects.

In current implementation of OpenSees, the two classes **SP_Constraint** and **MP_Contraint** are designed but they do not handle the constraints. These two classes are responsible of setting up relations between **Node**s and the constrained **DOF_Group**s. This will be covered shortly in **Analysis** class design.

Load

There are also two types of loads that are commonly seen in finite element analysis:

1. node loads that act on specific **Node**s;
2. element loads that act on specific **Element**s, which can be due to body forces, surface tractions, initial stresses and temperature gradients.

In the current implementation of OpenSees, three extra classes are introduced to handle loading conditions, **LoadPattern**, **NodalLoad** and **ElementLoad**. The **LoadPattern** is a container class that provides methods in its interface to allow **NodalLoad** and **ElementalLoad** objects to be created, traversed and removed. As shown in Figure 3.2, each **NodalLoad** or **ElementalLoad** object is associated with a **Node** or **Element** object and is responsible of applying nodal or elemental loads to that object.

Domain

The **Domain** class is the most important container class that is responsible of holding all components of the finite element model, i.e. all the **Node**s, **Element**s, **Constraint**s and **Load**s. **Domain** class acts as the interface between **Analysis** class and all the individual components of the finite element model. The interface of **Domain** enables component creation, information access and component removal.

3.2.3 Analysis

The **Analysis** class (McKenna, 1997) is responsible for forming and solving the governing equations for the finite element model. As for nonlinear problems, incremental solution techniques are required and iterative schemes such as Newton-Raphson needed to solver the nonlinear system of equations.

For incremental solution algorithm, the computational tasks are more involved for the finite element analysis.

- Assign equation numbers and map these to the nodal DOFs. This step can be of significant influence on the bandwidth of the coefficient matrix, which is inherently sparse due to the compact support of finite element formulation.
- Form the matrix equations using contributions from elements and nodes.
- Apply the constraints, which may involve transforming the element and nodal contributions or adding additional terms and unknowns to the matrix equations depending the method employed to handle constraints.
- Solve the matrix equations for the incremental nodal displacements.
- Determine the internal state and stresses in the elements.

The Object-Oriented design of the **Analysis** class is done by firstly breaking down the main tasks performed in a finite element analysis, abstracting them into separate classes, and then specifying the interface for these classes. The **Analysis** class is an aggregation of all the sub-functionality classes of following types:

1. **SolutionAlgorithm** class describes the complete computation procedure (steps) in the analysis.
2. **AnalysisModel** is a container class that stores and provides access to the following types of classes:
   (a) **DOF_Group** class represents the DOF at the **Node**s or new DOF introduced into the analysis to enforce the constraints;
   (b) **FE_Element** class represents the real **Element**s in the **Domain** or they are introduced to add stiffness and/or load to the system of equations in order to enforce the constraints.
   It is worthwhile to mention that the **FE_Element**s and **DOF_Group**s have very important design implications although they might seem redundant at the first sight. The significance comes from the facts that
      i. they record the mapping between DOFs and equation numbers in the global system which greatly simplifies the interfaces of **Node** and **Element** class;
      ii. they also provide the interfaces for forming tangent and residual vectors which are used to form the global system of equations;

iii. they are major utility classes of handling constraints.

3. **Integrator** defines how the **FE_Element**s and **DOF_Group**s contribute to the system of equations and how the response quantities should be updated given the solution to the global system of equations.

4. **ConstraintHandler** handles the constraint by creating adequate **FE_Element**s and **DOF_Group**s.

5. **DOF_Numberer** maps the equation number to the DOFs in the **DOF_Group**s.

6. **SystemOfEqn** encapsulates the global system of equations.

The aggregation of the **Analysis** object is shown in Figure 3.3.

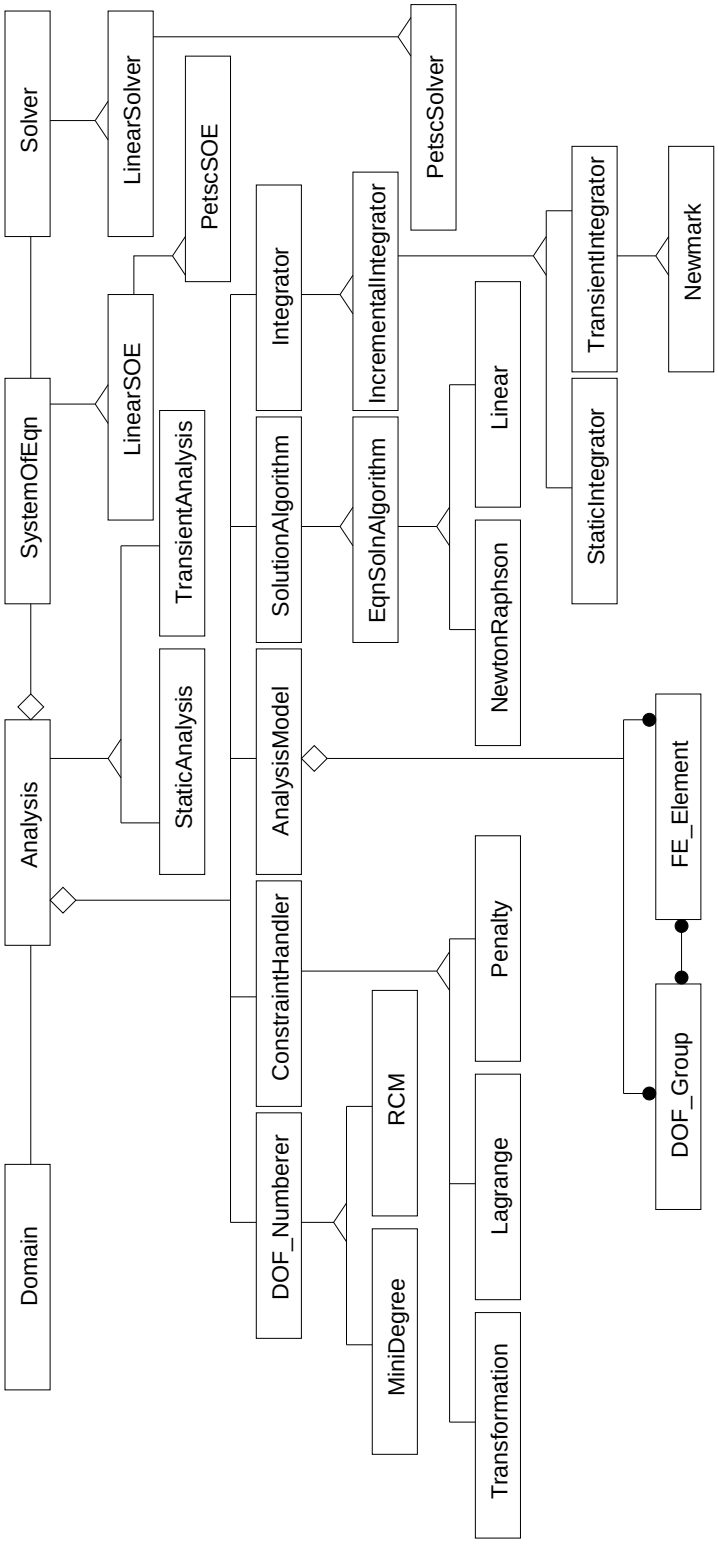


Figure 3.3: Class Diagram of Analysis Aggregation

Traditional program flow diagrams are used to describe how is the nonlinear finite element algorithm control flow implemented in OpenSees. These flow charts are organized as following:

Figure 3.4 shows the overall analysis algorithm flow for nonlinear finite elements.
Then this overall analysis flow is broken down into detailed subroutines, such as *theIntegrator::newStep()* and *theAlgorithm::solveCurrentStep()*.

- Figure 3.5 explains in detail the function flow of *theIntegrator::newStep()*, which illustrates the (fairly standard) incremental finite element solution techniques implemented in OpenSees.
- Figure 3.6 shows the function flow of forming the tangent stiffness matrix, which is a loop assembling the global equation system involved in function *theIntegrator::newStep()*.
- Figure 3.7 further describes the Newton-Raphson type iterative solution schemes involved in function *theAlgorithm::solveCurrentStep()*.

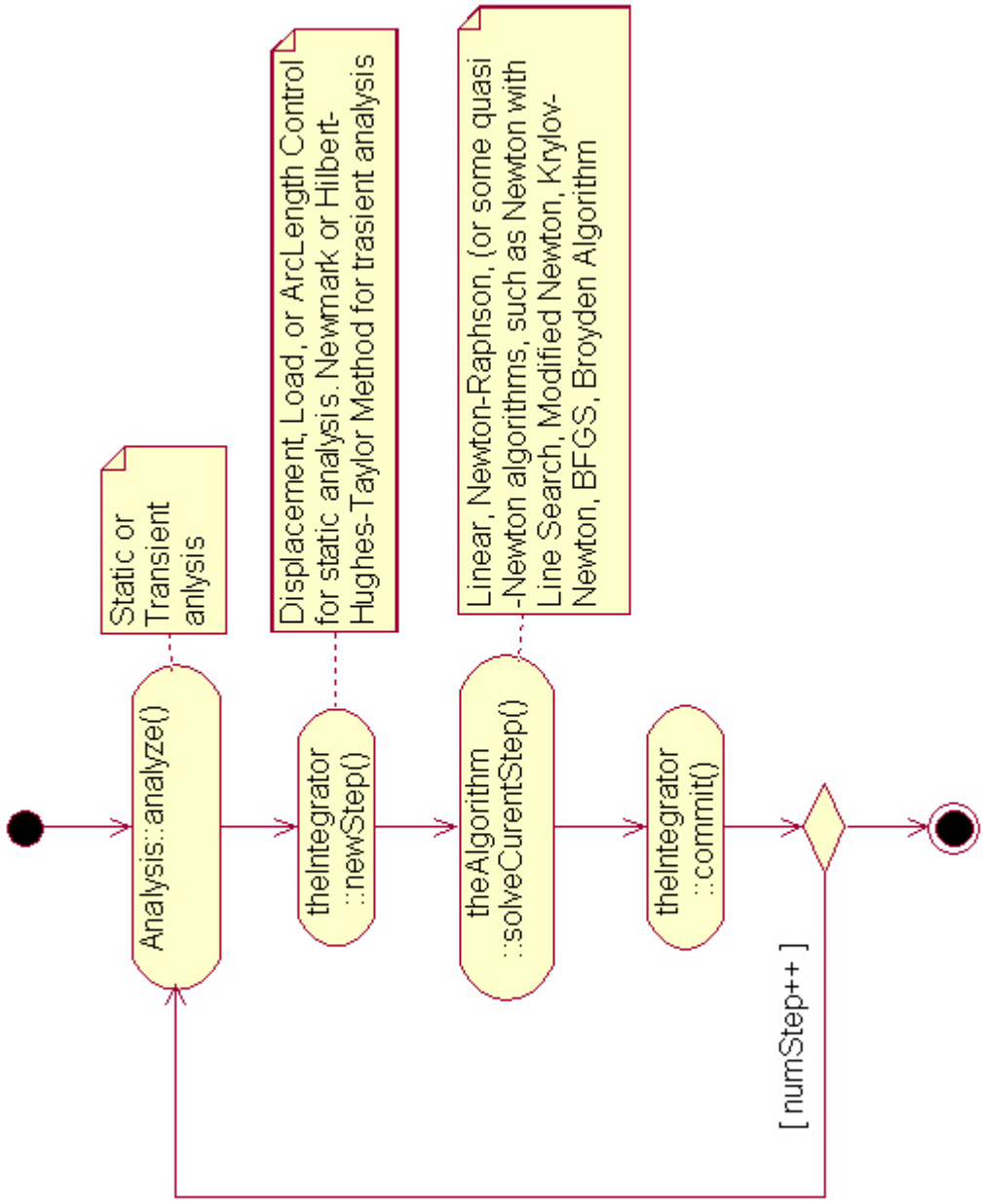


Figure 3.4: Overall Algorithm Flow Chart for Nonlinear Finite Element Analysis

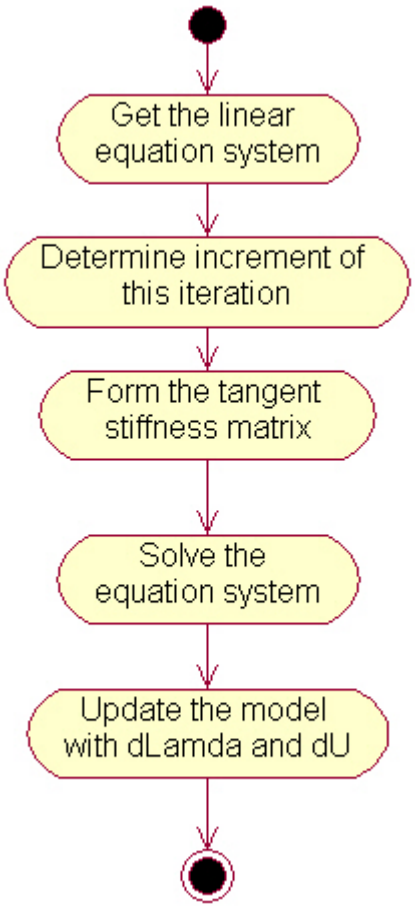


Figure 3.5: Detailed View: *theIntegrator::newStep()* - Incremental Solution Techniques for Nonlinear Finite Element Analysis

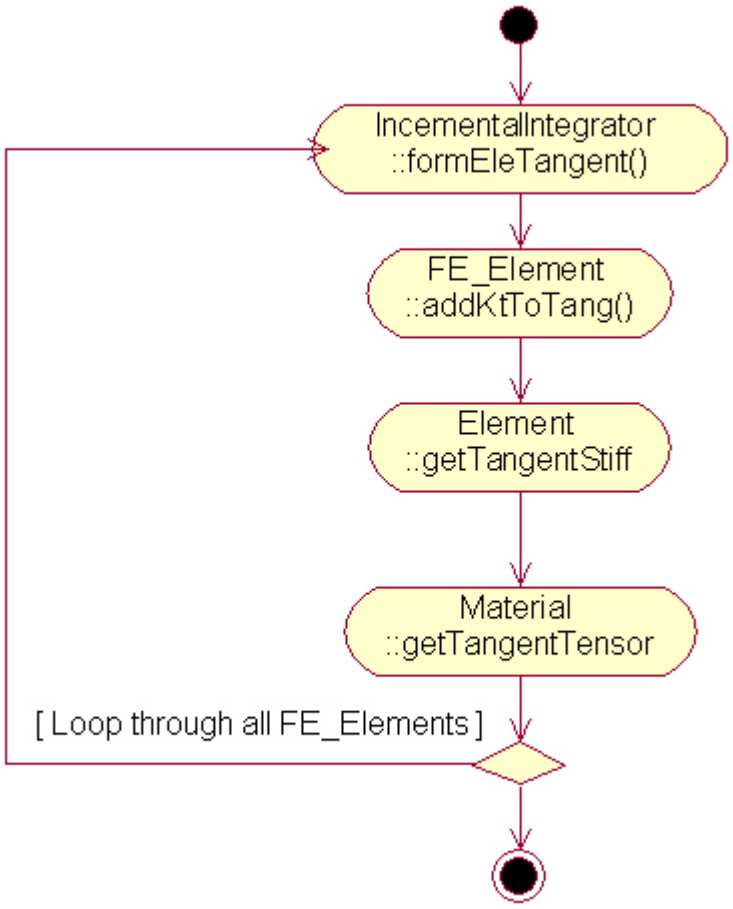


Figure 3.6: Detailed View: Assembly of Global Equation System in *theIntegrator::newStep()*

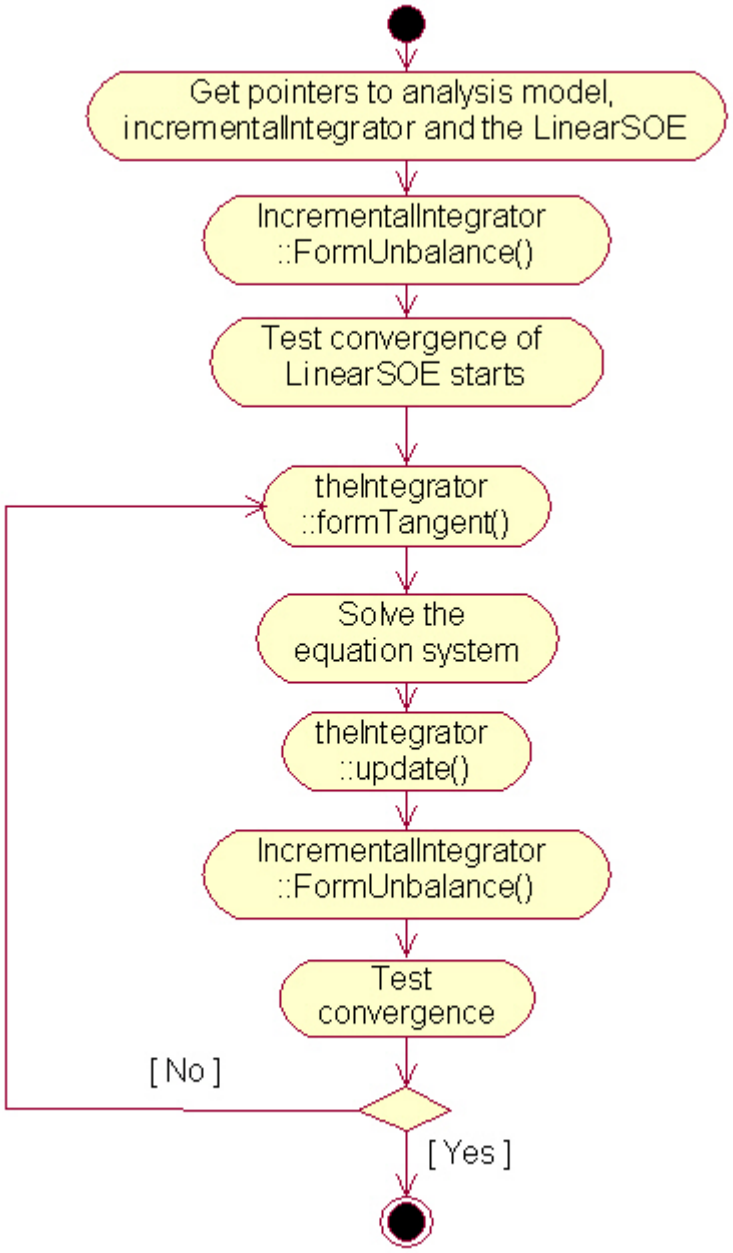


Figure 3.7: Detailed View: *theAlgorithm::solveCurrentStep()* - Newton-Raphson Iterative Schemes for Nonlinear Finite Element Analysis

Finite element simulations inherently are element-based operations, so little modification is needed to parallelize the algorithms described above, although special attention has to be paid to synchronize the computation among different processors. Figure 3.8 shows the activity flow for parallel nonlinear finite element simulations.

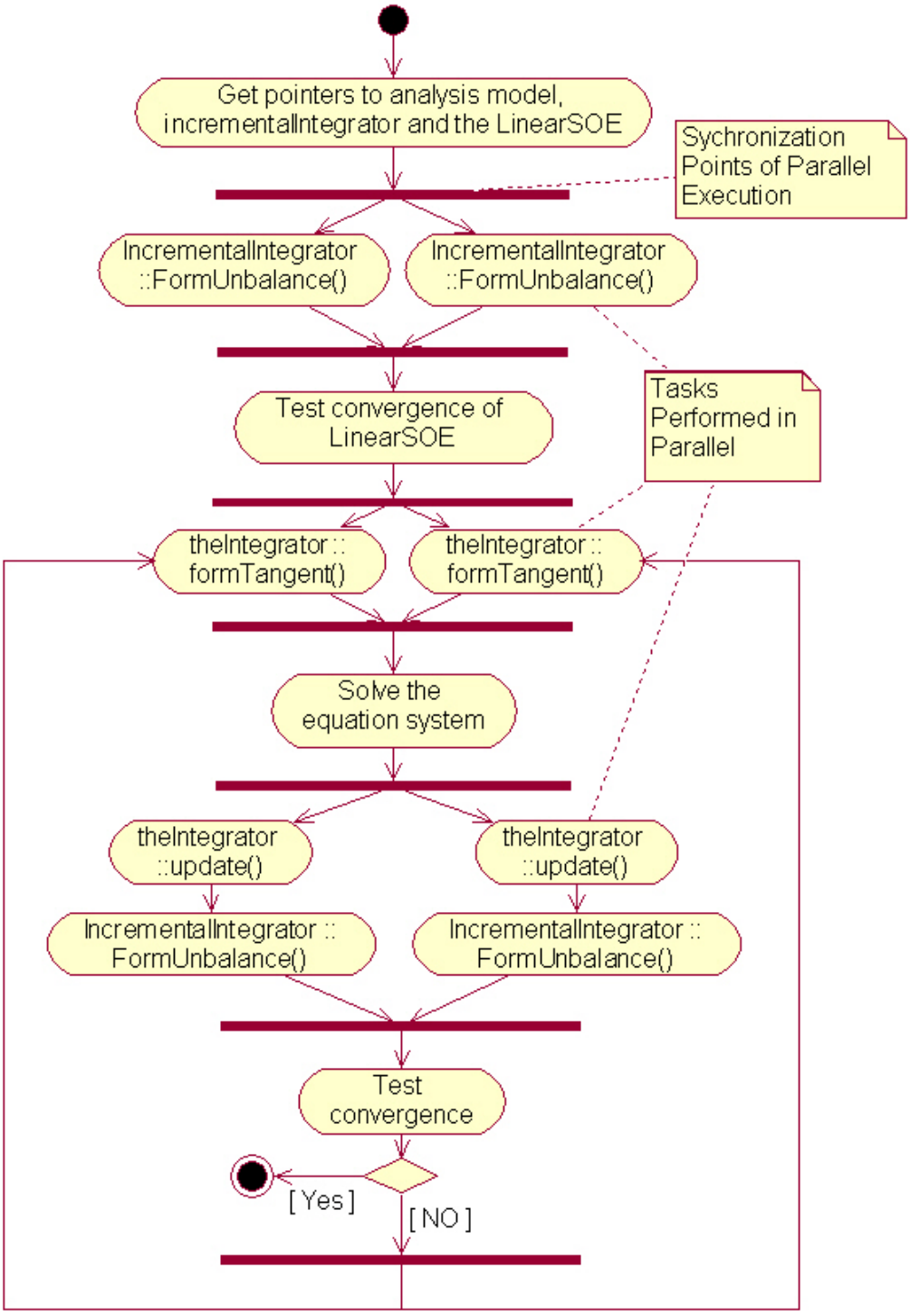


Figure 3.8: Parallel Activity Flow Diagram of Nonlinear Finite Element Analysis

3.2.4 Object-Oriented Domain Decomposition

There are three most notable designs of Domain Decomposition method in literature McKenna (1997).

1. Sause and Song (1994) presents an Object-Oriented design for linear static analysis using substructuring. The interface is restricted to substructuring or FETI Farhat and Roux (1991a) only, and repeated geometry limits the applicability of this design to large problems.

2. Archer (1996) proposes a **SuperElement** class that is a subclass of **Element** and has a **Domain** class aggregated. This design is conceptually inappropriate and it results excessive method calls as methods that are for the **SuperElement** must be called by the **SuperElement** on the associated **Domain** McKenna (1997).

3. Miller and Rucki (1993) introduces the **Partition** class which is associated with an **Algorithm** class. The **Algorithm** class is responsible for updating the state of a **Partition** so that it will be in equilibrium. Again, this design is good for substructuring type Domain Decomposition analysis. If we want to solve a problem before the interface solution can be determined, the design fails.

The current design of OpenSees McKenna (1997) proposes many new classes to facilitate flexible Object-Oriented Domain Decomposition. The main abstractions include **PartitionedDomain**, **DomainDecompAnalysis**, **DomainDecompSolver Subdomain**, **DomainPartitioner** and **GraphPartitioner**. The class diagram is shown in Figure 3.9.

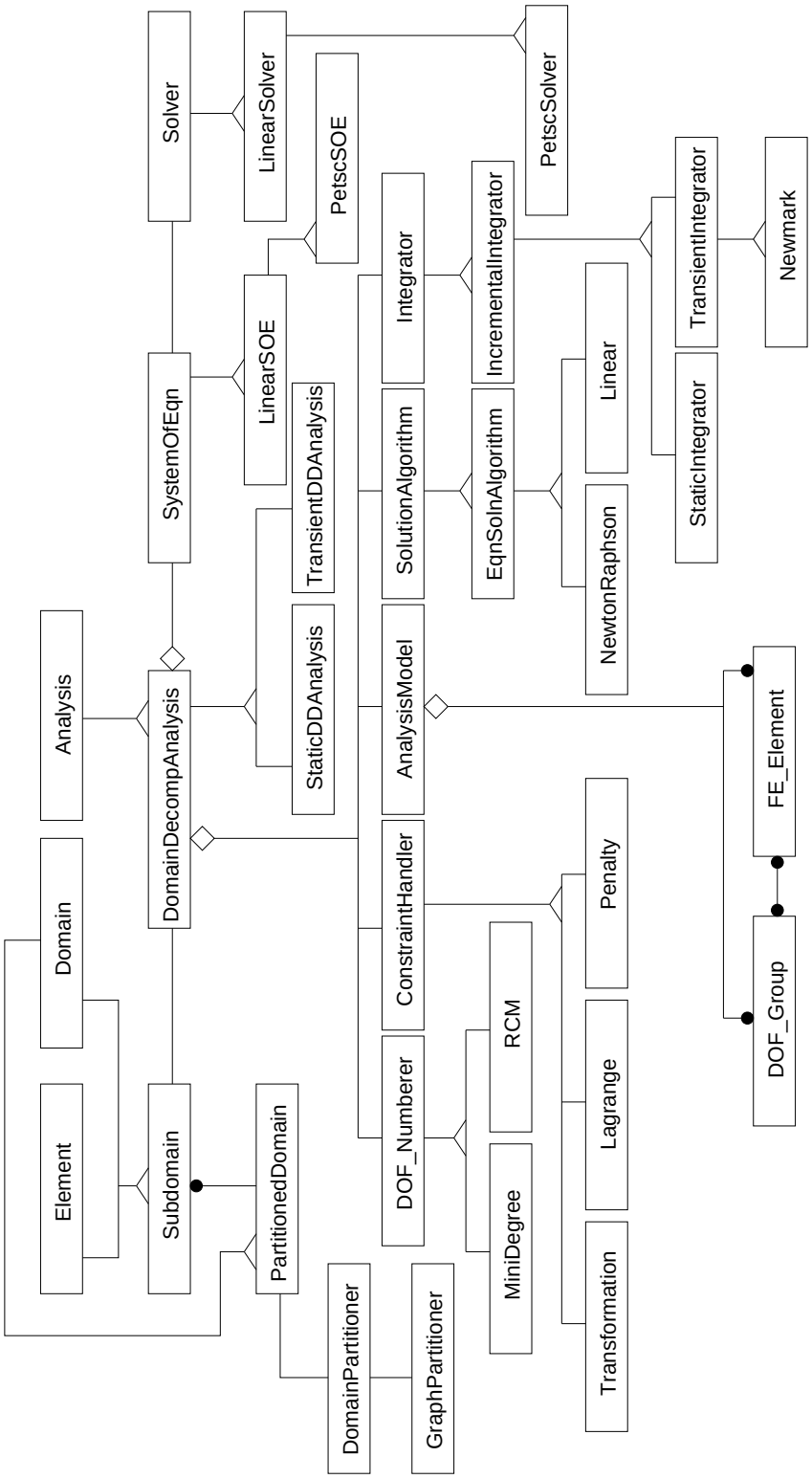


Figure 3.9: Class Diagram of Domain Decomposition Analysis

PartitionedDomain

The **PartitionedDomain** class is a subclass of **Domain** whose objects can be partitioned into **Subdomain** objects. Aside from common functionality inherited from **Domain**, **PartitionedDomain** class provides methods for partitioning the **Domain** and retrieving information from **Subdomain**s. **PartitionedDomain** the aggregation of **Subdomain**s and is the major containing class in master process.

DomainPartitioner

The **DomainPartitioner** class is responsible for performing the actual operation to split the **PartitionedDomain**. The **DomainPartitioner** will call its associated **GraphPartitioner** to partition the **PartitionedDomain**. It also provides the methods to migrate **Element**s, **Node**s, **Constraint**s, **Load**s amongst **Subdomain**s.

DomainPartitioner is one of the most important utility class in OpenSees in the sense that all partitioning routine and data migration operations will be rooted from this class.

GraphPartitioner

This class utilizes external graph partitioner to color the finite element connectivity graph, which will be constructed from the **PartitionedDomain**. The result will be fed back to **DomainPartitioner** to facilitate subsequent data distribution.

GraphPartitioner introduces graph partitioning into OpenSees and the main functionality of this class is to call API and provide necessary data structures from the specific application.

Subdomain

The **Subdomain** class inherits from both **Element** and **Domain**. This has a dual-level design implication:

1. for the top **PartitionedDomain**, superclass **Element** is a proxy class of subclass **Subdomain**, in the sense that all the relevant operations on **Element**s invoked by **PartitionedDomain** will be redirected to the specific **Subdomain**;

2. for any specific **Subdomain**, it inherits all the interfaces of **Domain** to do all the computations required by **PartitionedDomain**.

3.2.5 Parallel Object-Oriented Finite Element Design

There has been much effort by researchers on parallel implementation of finite element computations, which can be categorized into either domain decomposition methods or parallel equation solving.

Domain decomposition is favored by many researchers due to its nice "*divide and conquer*" approach. The subdomains in the domain decomposition method are each assigned to a processing node, which will perform all the computations on that subdomain.

Of the domain decomposition methods, the substructuring method has been the most popular choice although other methods such as iterative substructuring Carter et al. (1989) and FETI (Finite Element Tearing and Interconnecting) Farhat and Roux (1991a); Farhat and Crivelli (1994) have also been used. In the substructuring method presented, static condensation is typically performed on the assembled system of equations.

Earlier works on parallel processing for inelastic mechanics focused on structural problems. We mention work by Noor et al. (1978); Utku et al. (1982); Storaasli and Bergan (1987) in which they used substructuring to achieve partitions. Fulton and Su (1992) developed techniques to account for different types of elements but used substructures of same element types (non–balanced computations). Hajjar and Abel (1988) developed techniques for dynamic analysis of framed structures with the objective of minimizing communications. Klaas et al. (1994) developed parallel computational techniques for elastic–plastic problems but tied the algorithm to the specific multiprocessor computers used (and specific network connectivity architecture). Farhat (1987) developed the so–called Greedy domain partitioning algorithm but stayed short of using redistribution of domains as a function of developed nonlinearities.

The major parallel programming model in OpenSees (McKenna, 1997) is the so-called **Actor** model, which is a mathematical model of concurrent computation that has its origins in Hewitt et al. (1973) . **Actor**s Agha (1984) are autonomous and concurrently executing objects which execute asynchronously. **Actor**s can create new actors and can send messages to other actors. The **Actor** model is an Object-Oriented version of message passing in which the **Actor**s represent processes and the methods sent between **Actor**s represent communications.

The **Actor** model adopts the philosophy that *everything is an* **Actor**. This is similar to the *everything is an Object* philosophy used by object-oriented programming languages, but differs in that object-oriented software is typically executed sequentially, while the Actor model is inherently concurrent (http://en.wikipedia.org/wiki/Actor_model).

An Actor is a computational entity with a behavior such that in response to each message received it can concurrently:

- send a finite number of messages to (other) Actors;
- create a finite number of new Actors;
- designate the behavior to be used for the next message received.

Note that there is no assumed sequence to above actions and that they could in fact be carried out in parallel.

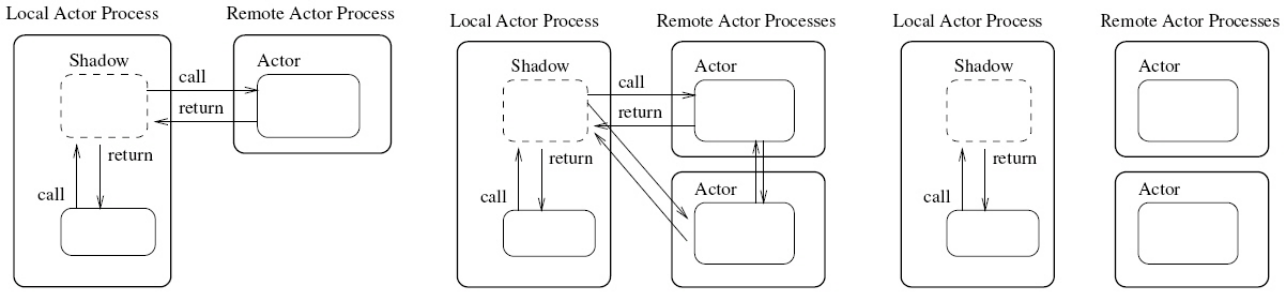
Communications with other **Actor**s occur asynchronously (i.e. the sending **Actor** does not wait until the message has been received before proceeding with computation), which is the unblocking behavior.

Messages are sent to specific **Actor**s, identified by address (sometimes referred to as the **Actor**'s "mailing address"). As a result, an **Actor** can only communicate with **Actor**s for which it has an address which it might obtain in the following ways:

- The address is in the message received;
- The address is one that the **Actor** already had, i.e. it was already an "acquaintance";
- The address is for a just created **Actor**.

The **Actor** model is characterized by inherent concurrency of computation within and among **Actor**s, dynamic creation of **Actor**s, inclusion of **Actor** addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

In order to minimize the changes to the sequential Domain Decomposition design presented in previous sections, McKenna (1997) introduces the **Shadow** class. A **Shadow** object is an object in an **Actor**'s local address space. Each **Shadow** is associated with one **Actor** or multiple **Actor**s in the case of an aggregation. The **Shadow** object represents the remote object to the objects in the local **Actor**'s space. The **Shadow** object is responsible for sending an appropriate message to the remote **Actor** or **Actor**s if broadcasting. 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 3.10.



(a) Actor - Communication Required

(b) Aggregate - Communication Required

(c) No Communication Required

Figure 3.10: Communication Pattern of Actor-Shadow Models McKenna (1997)

Some other new classes of parallel finite element programming are:

- **Channel** is the bridge through which the **Actor**s and **Shadow**s can communicate.

- **Address** represents the location of a **Channel** object in the machine space. **Channel** objects send/receive information to/from other **Channel** objects, whose locations are given by the **Address** objects.
- **MovableObject** is an object which can send its state from one actor process to another.
- **ObjectBroker** is an object in a local actor process for creating new objects.
- **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.

The relation between these classes is shown in Figure 3.11.

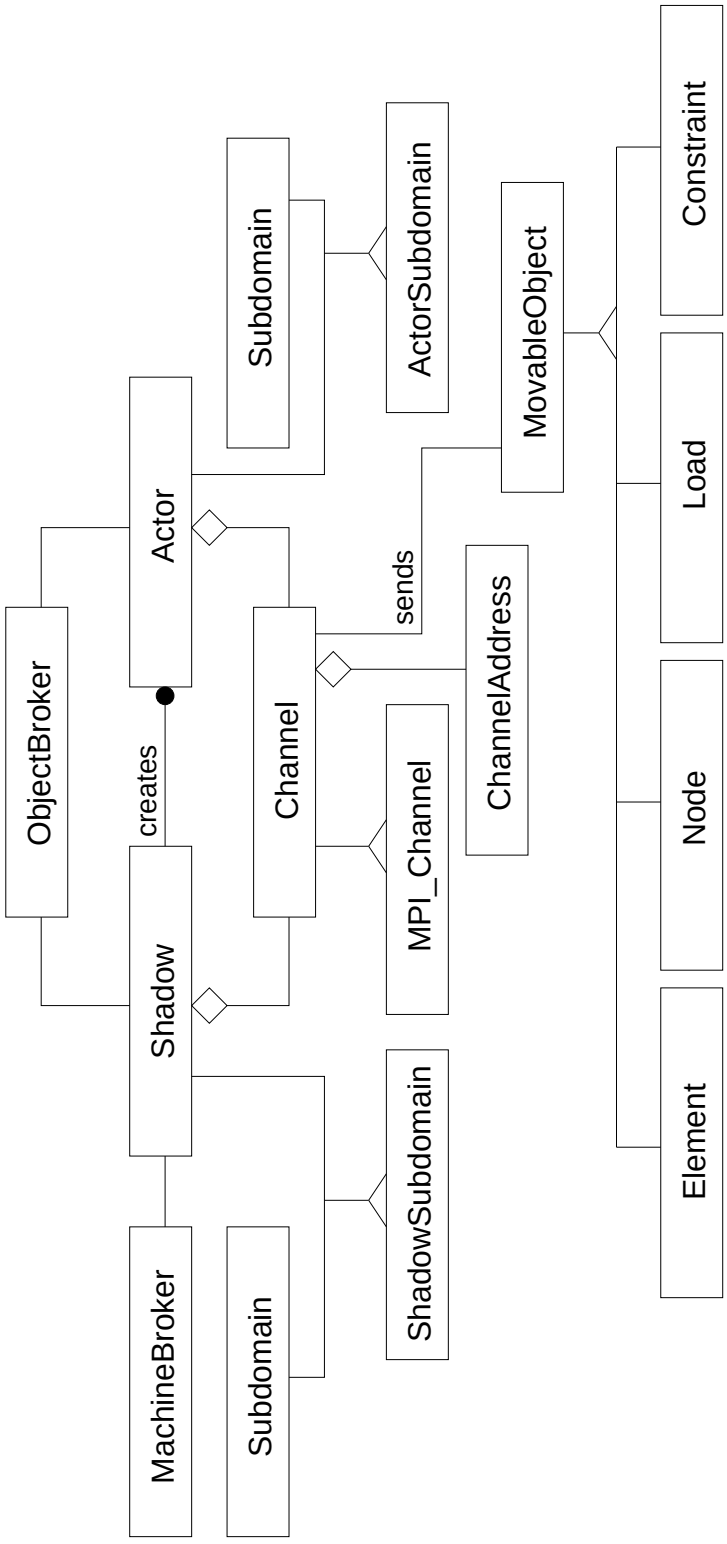


Figure 3.11: Class Diagram for Parallel Finite Element Analysis

3.3 Dual-Phase Adaptive Load Balancing

From the Figure 3.7, one can easily identify two computational phases that are fundamental to nonlinear elastic-plastic finite element simulations. One is well known as global level equation solving and the other is local level elemental calculations during which the elemental update happens for each element. In nonlinear elastic-plastic finite element simulations, the local computational phase can be much more expensive than the global equation solving phase due to the presence of complex material models and nonlinearity.

In this report , the implementation of proposed PDD algorithm has considered load balancing issues on both elemental level elastic-plastic computations and global level equation solving.

3.3.1 Elemental Level Load Balancing

The load balancing operation on constitutive level is built on the foundation of adaptive multilevel graph partitioning algorithm available through ParMETIS.

In this report , element-based graph is constructed from the Finite Element mesh on which the graph partitioning algorithm acts on to obtain partitions and/or repartitions. Each element will be assigned a vertex tag for identification.

When two elements at least share a single node, we assign an edge to both vertices because the element graph is deemed to be undirected, which means the edge is equally identified by two vertices without ordering required.

We creatively specify vertex weight to represent elemental level computational load for each vertex (element). In the implementation of this report , the vertex weight will be automatically updated as simulation progresses to reflect element computation cost. Performance timing has been added for constitutive update routines and the graph data structure will be refreshed every single iteration.

The last metric used is the vertex size of each vertex which basically contains the information that how much memory each vertex (element) requires in order to reproduce itself to other processes during data distribution. Adaptive load balancing is a multi-objective operation in the sense that both edge cut and data migration cost must be minimized simultaneously. The vertex size exactly describes the size of data that need to be shipped via communication. This metric must be correctly obtained for all available element types in order for the multi-objective load balancing algorithm to ensure the best performance.

3.3.2 Equation Solving Load Balancing

Parallel equation solving algorithm falls into two major different categories, direct solver and iterative solver.

Direct solver stems from Gaussian-type elimination and effective elimination tree is determined by the sparsity pattern of the stiffness matrix. Load balancing issue is addressed inherently when forming the elimination tree. Various packages such as SPOOLES and SuperLU provide scalable direct solutions to parallel equation systems. Chapter 5 discusses in further details about parallel direct solvers that are

available as part of the release of this report .

Iterative solver has been the focus of this report in the sense that special care has been paid to achieve dynamic load balancing for each partition/repartition. The kernel of project-based iterative solvers is matrix-vector multiply. The issues of how to evenly distribute the stiffness matrix in parallel among different processors and how to reorder the sparse matrix to reduce data communications have been the focus of this report .

In order to achieve load balancing for parallel iterative solvers, parallel matrix/vector storage scheme and sparse matrix ordering are key factors. In the implementation of this report , even row-distribution of stiffness matrix among processing units is assumed. As shown in Figure 3.12, each processing unit has equal number of rows stored locally. The right hand side of the system is the force vector, which will be replicated for each processing unit. In this way, one can expect fastest matrix-vector multiply with the least amount of data needed to be communicated through network. As matrix-vector multiply is performed in parallel, load balancing issue is related to the number of nonzero numbers of the sparse stiffness matrix, which directly determines how many floating point multiplications are needed. In finite element computations, this nonzero pattern is determined by DOF numbering. Bandwidth reducing numbering scheme, or matrix ordering scheme, such as RCM Dongarra et al. (2003), can effectively lead to a sparse pattern that has similar number of nonzero elements on majority of rows as shown in Figure 3.12.

Finite element method inherently possesses compact support. Off-diagonal data of the stiffness matrix need to be synchronized among different processors. In order to reduce the extra overhead involved, in this report , several implementation solutions have been considered.

- **Graph Partitioning Phase**. As stated in previous chapters, minimizing edge-cut is one of the main objectives of the partitioning operation on the element graph. One extra benefit is that the bandwidth of the stiffness matrix will be greatly reduced. The number of nodes that need to be synchronized will be greatly reduced.

- **DOF Numbering Phase**. This phase is to renumber the DOFs of the finite element model after data redistribution in order to make sure contributions from local elements will sit on rows that are stored locally. This is done every time when the data migration is triggered. The idea is to start numbering the DOFs from local elements in Processor 1 to local elements in Processor N. In this way, when the global matrix is formed, local element stiffness matrix will always become clustered along the diagonal.

3.4 Object-Oriented Design of PDD

The parallel design of PDD basically follows *Master-Slave* algorithm structure as shown in Figure 3.13 and MPI has been adopted to facilitate inter-processor communications. The Actor/Shadow model described in previous sections is the used in PDD implementation and does nicely interact with parts of OpenSees

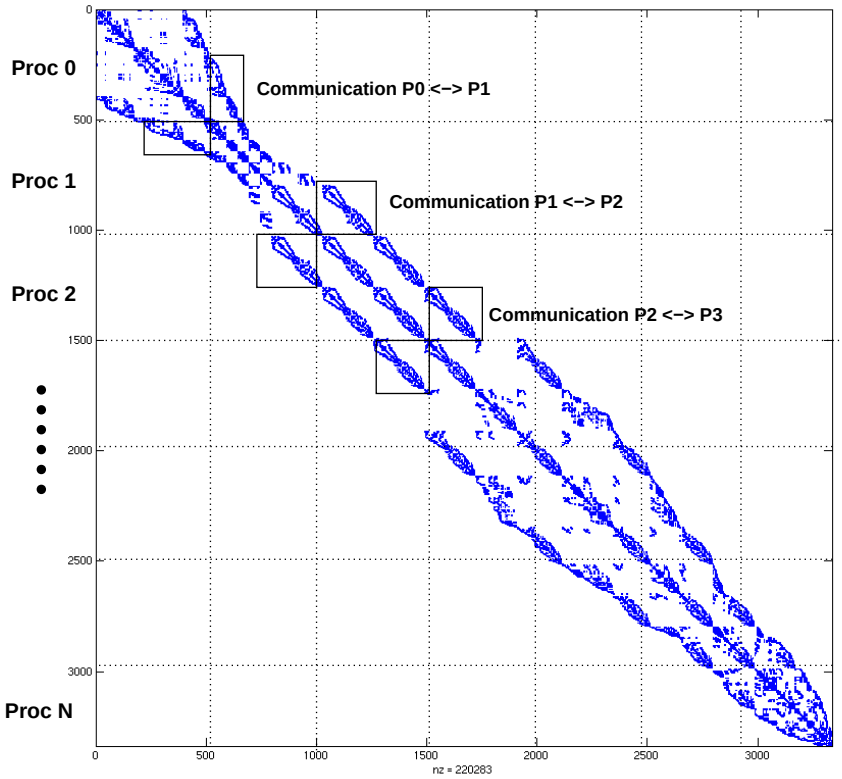


Figure 3.12: Parallel Data Organization of SFSI Equation System

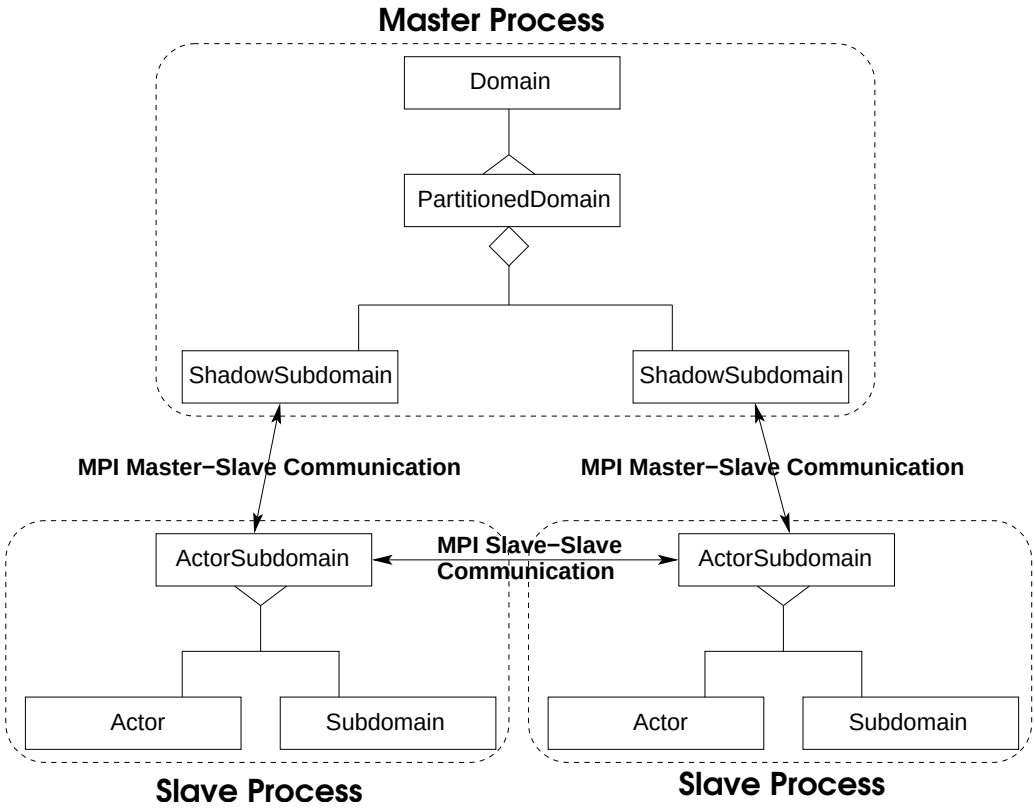


Figure 3.13: Master–Slave design used for PDD development.

framework, which uses **Actor** and **Shadow** classes to facilitate the inter-process communication between master and slave processes.

- **Master Process**

  Master process assumes the role to *orchestrate* the whole computation process. OpenSees uses *tcl* as an interpreter (or any other interpreted language that can be embedded into c or C++) to read input scripts from user. In parallel implementation of described here, master process is responsible for establishing the whole model for analysis and then distributing data among sub-processors. An important improvement in this report is that the master does not actually create all finite element objects, whose memory space will only be allocated after they are sent to subdomains. This design helps avoid the high memory requirement on the master side. Initial partitioning is done solely by master process or in parallel by all working processes. Data movement is coordinated by the master process, in which a complete element graph is kept intact.

  As for repartitioning, the master process is still responsible for issuing commands to migrate data from this subprocessor to another even though the data is not in master.

- **Slave Process**

  The actor model has been used and modifications have been added to avoid unnecessary data communications. Basically speaking, actors in slave processes will be *waiting for orders* till master issues one and then do corresponding work on their own copy of data. The original design in OpenSees framework has disabled slave to initiate communication, which means in order for a sub-processor to communicate with another sub-processor, it has to send all the data back to the master process first. This is highly inefficient and needed to be redeveloped (improved). In this research actor model has been implemented to enable direct communications between sub-processes and this improvement greatly reduced unnecessary communications.

All of the class designs for sequential version of OpenSees can be reused in parallel version following the Object-Oriented paradigm. There are some very important additions however in order to facilitate master-slave parallel processing. In this section, these classes will be revisited and updated/changes/improvements originally developed during this research will be explained thereafter.

- **PartitionedDomain**

  The **PartitionedDomain** class basically inherits all functionality from the **Domain** class in sequential version. This class acts as a container class in the master process. It differs from **Domain** class in the respect that all actions performed on the domain will be propagated to all subdomains when doing parallel processing.

- **Subdomain**

  The **Subdomain** is a child class of **Domain**. This class will be instanced by each slave process and

it covers all functionality of the **Domain** class in sequential version. It can be called as an instance of **Domain** taking care of components only for the local slave process.

- **ActorSubdomain & ShadowSubdomain**

  The **Actor/Shadow Subdomain** classes are the most important classes for parallel version OpenSees. They are assuming the roles to initiate and facilitate all communications between master and slave processes. Both **Actor/Shadow Subdomain** will be instanced automatically when user creates slave processes.

  ShadowSubdomain sits on master process. The function of this class is to represent a specific slave process in master process. Master does not directly interact with slave. Whatever action that needs to be performed by the slave process will be issued to **ShadowSubdomain**. This extra layer smooths the communication between master and slave nodes.

  On the other hand, **ActorSubdomain** sits on slave process and it hides master from slave nodes. All commands from master node will be received by **ActorSubdomain** and **ActorSubdomain** will match the command with some actions performed by **Subdomain**.

  Actor/Shadow Subdomain are extremely important classes in the parallel implementation of this report . They carry all communication functionality required to finish the partition and adaptive repartition.

- **Channel**

  Channel is the class that really does the job of sending/receiving data between processors. Only MPI channel has been used in this report . Specific data structure, such as ID (integer array), vector (double array) or matrix needs to provide its own implementation for send/receive functionality.

- **FEMObjectBroker**

  This class is instanced only at slave processors, which is in charge of creating new model data for subdomains. This design isolates model creation from communication classes.

- **Address**

  Address class identifies parallel processes. With MPI channel used, the address corresponds to global process ID.

- **DomainPartitioner**

  DomainPartitioner assumes the responsibilities of invoking the **GraphPartitioner** and feeding necessary data to finish the partition/repartition. This class will also be in charge of data migration after partition/repartition is done.

- **SendSelf & RecvSelf**

  These two should be called functions rather than classes. **SendSelf & RecvSelf** are functions implemented to provide copy of model data to finish sending/receiving operations.

The old parallel design of OpenSees is not capable of performing elastic–plastic computations since it was designed and implemented for a single stage loading only. This single stage loading works fine for elastic analysis, but since elastic–plastic materials do have memory, staged loading is essential for any realistic computations with elastic–plastic material. This is particularly true for geotechnical and structural models, where simulations support for staged loading (self weight of soil medium for initial stress, construction process and subsequent static or dynamic loading) is essential if any modeling accuracy is to be achieved. One of new developments in this report was the addition of multi-stage elastic-plastic analysis. This improvement included modification of 3D solid and beam elements, Template3Dep/NewTemplate3Dep material models and DRM loading pattern for seismic analysis. Some of the old utility commands, such as "wipeAnalysis", were improved/redeveloped to enable parallel multi-stage analysis.

The most significant significant improvement developed during research over the old parallel design of OpenSees is the introduction of load balancing technique by adaptive graph partitioning algorithm through ParMETIS. Major improvements/updates have been introduced in **PartitionedDomain**, **Actor/ShadowSubdomain**, **DomainPartitioner**, **FEM_ObjectBroker** and **Subdomain**. Modifications done in this report also focus very much on performance issue. In order to reduce unnecessary data communication during partitioning/repartitioning, some functions have been rewritten. The functionality of **Actor** and **ShadowSubdomain** have been expanded so that any **ActorSubdomain** can initiate communication to another **ActorSubdomain**. The old design of OpenSees had to use master process as intermediate layer if subdomains want to exchange information.

For example, if Subdomain No. 1 needs to migrate an Element to Subdomain No. 2, the old design would issue a "remove Element" command from master PartitionedDomain to Subdomain No. 1, then Subdomain No. 1 would remove the Element and send the Element back to master process, finally the Element would be migrated to Subdomain No. 2. We can clearly recognize the communication to master is not necessary here. In order to develop adaptive load balancing while minimizing data redistribution cost, the improvement in this report is to allow ActorSubdomain at source Subdomain initiates communication with ActorSubdomain at target Subdomains and they can exchange information without recourse to master process. So the new communication pattern will be, again for the "migrate element" case, the master process will issue an "export element" command to Subdomain No. 1 and a "receive element from Subdomain No. 1" command to Subdomain No. 2, and then the element information will be directly sent from Subdomain No. 1 to No. 2.

Details of implementation are given in following sections.

3.4.1 MPI_Channel

- Functions **sendnDarray** and **recvnDarray** have been added to facilitate the data communication of Template3D material classes, which are based on **nDarray** tensor data structures.

  int MPI_Channel::sendnDarray(int,int, const nDarray&, ChannelAddress*)

int MPI_Channel::recvnDarray(int,int, const nDarray&, ChannelAddress*)

3.4.2 MPI_ChannelAddress

- Function **getOtherTag** has been added to get MPI global ID for the specific **MPI_Channel**. This function is mainly used for data migration. It provides the MPI global communicator ID of the target process which the next communication will be directed to.
  int MPI_ChannelAddress::getOtherTag(void)

3.4.3 FEM_ObjectBroker

- New functionality to instance 3D continuum brick elements has been added to **getNewElement** function.
  Element* FEM_ObjectBroker::getNewElement(EightNodeBrickTag)
- New functionality to instance **Template3D/NewTemplate3D** material models for continuum brick elements has been added to **getNewNDMaterial** function.
  NDMaterial* FEM_ObjectBroker::getNewNDMaterial(int)
- **Template3D** material is a stand-alone material library designed for general elastic-plastic materials. User can define separately **YieldSurface**, **PotentialSurface**, **Scalar Evolution Law** and **Tensorial Evolution Law**. Various material models have been implemented in OpenSees Jeremić and Yang (2002), such as Cam Clay, Drucker Prager and von Mises yield/potential surfaces, Armstrong Frederick nonlinear kinematic hardening law and bounding surface plasticity. All the material models have to be instanced by FEM_ObjectBroker during parallel processing.
  YieldSurface* FEM_ObjectBroker::getYieldSurfacePtr(int)
  PotentialSurface* FEM_ObjectBroker::getPotentialSurfacePtr(int)
  EvolutionLaw_S* FEM_ObjectBroker::getEL_S(int)
  EvolutionLaw_T* FEM_ObjectBroker::getEL_T(int)
- **NewTemplate3D** material is a newly designed material library which includes more advanced elastic-plastic constitutive models for geomaterials, such as Dafalias and Manzari 2004 model. The design of **NewTemplate3D** extends the principle of **Template3D**, in which key parameters describing plasticity model are abstracted as different class objects, such as **YieldFunction**, **PlasticFlow**, etc. In order to reduce unnecessary data allocation, new **MaterialParameter** class has been developed to carry all material parameters. New **ElasticState** has been used to store all intermediate and/or committed stress/strain data. All these material classes have to be instanced by FEM_ObjectBroker during parallel processing and new functions have been implemented in this report .
  MaterialParameter* FEM_ObjectBroker::getNewMaterialParameterPtr(void)

ElasticState* FEM_ObjectBroker::getNewElasticStatePtr(int)
YieldFunction* FEM_ObjectBroker::getNewYieldFunctionPtr(int)
PlasticFlow* FEM_ObjectBroker::getNewPlasticFlowPtr(int)
ScalarEvolution* FEM_ObjectBroker::getNewScalarEvolutionPtr(int)
TensorEvolution* FEM_ObjectBroker::getNewTensorEvolutionPtr(int)

3.4.4 Domain

- Timing routines have been added to **update** function to measure computation time of constitutive level iterations for each element during every single loading increment. This metric will be assigned to the corresponding vertex of the element graph as the vertex weight. This metric represents element-level computational load against which subsequent load balancing techniques will be applied.

3.4.5 PartitionedDomain

- **addElementalLoad** function has been added to add **ElementalLoad** into **LoadPattern**, which was not supported in the old design.
  bool PartitionedDomain::addElementalLoad(ElementalLoad*, int)
- **repartition** function has been implemented to initiate adaptive repartitioning on the element graph of the **Domain** after every loading increment.
  int PartitionedDomain::repartition(int)

3.4.6 Node & DOF_Group

- **sendSelf** and **recvSelf** functions for **Node** class have been changed mainly to deal with the **DOF_Group** object associated with the **Node**. In the old design of parallel version of OpenSees, only one-step static domain partitioning would be invoked so that there is no need to pass the **DOF_Group**. But in this report , adaptive load balancing is developed to achieve better performance. The **Node** class should keep the information of its own **DOF_Group**, which guarantees the consistency of the **DOF_Graph** of the whole **Domain**. This point is extremely important when user tries to invoke **Transformation** constraint handler on the **DOF_Graph**. The addition of this feature in **Node** improved the robustness of the whole program.

- **DOF_Group** is a class carries information about the **DOF_Graph** of the analysis model, which will be used to finish assembling the stiff/mass/damping matrices. Each **Node** has its own **DOF_Group** to record the IDs of degree of freedoms in the global analysis model. Function **unSetMyNode** has been introduced to avoid segmentation fault. The reason is that after each round of repartitioning, if data movement is required, the **AnalysisModel** will be wiped off but **Node**s are still in existence.

Introduction of **unSetMyNode** function separates **Node** from its **DOF_Group** so the **DOF_Group** can be wiped and regenerated for the new model. **void DOF_Group::unSetMyNode(void)**

3.4.7 DomainPartitioner

DomainPartitioner is one of the most extensively changed classes in this report . This class acts as the entry point for **PartitionedDomain** to do domain decomposition and it basically has been rewritten to introduce new partition/repartition functionality and new data structures.

- Function **repartition** is implemented to do repartitioning after each loading increment. Partition and repartition are both implemented in parallel through ParMETIS library in this report . This function will collect **ElementGraph** from each **Subdomain** and pass them to **GraphPartitioner**. The global **ElementGraph** will be kept intact from which connectivity/adjacency information will be gathered to assemble child **ElementGraph**s and provide initial graph distribution data for repartition routines. After repartitioning by ParMETIS finishes, the function will verify the new partition against the original one to see if data redistribution is required to achieve load balancing. This **repartition** function also acts as a commander to control the data migration for adaptive load balancing. It issues commands to **ShadowSubdomain** to export/import **Node**s, **Element**s, **Constraints**s, **Load**s, etc.

- The old design of OpenSees used multiplication of prime numbers as index number to record which partitions a specific node belongs to. This is a very good idea because with this approach, we only need one integer for each node to keep track of node partitions, which can be called as an index number for the node. The idea was to name each **Subdomain** with one specific prime number, if a node belongs to this **Subdomain**, we would multiply the index number of the node with the prime number of this **Subdomain**. In order to determine if a node belongs to on specific **Subdomain**, all we need is to divide the index number of the node with the prime number the **Subdomain** represents to see if we can get zero residual.

- The drawback of the old data structure based on prime numbers is that it only works when the number of processing units is small, say less than 16. In 3D continuum models, a single node might belong to up to 8 partitions simultaneously, which happens when a corner node sits on intersections of different **Subdomain**s. As we know, prime numbers grow up very fast, multiplication with 8 prime numbers can easily overflow the index number of the node. A new data structure inspired by the Compressed Sparse Row (**CSR**) storage format popular in sparse matrix calculations has been introduced into in this report to solve the problem. One integer array has been used to store the partition data of all nodes, i.e. which partitions this node belongs to. Another integer array has been employed to record the count of partitions for each node. With these two arrays, we can load as many partitions as we want in our parallel processing.

3.4.8 Shadow/ActorSubdomain

As mentioned in previous sections of this report , **Shadow/ActorSubdomain** are the most important classes in parallel design of OpenSees McKenna (1997). **ShadowSubdomain**s represent **Subdomain**s in the master **PartitionedDomain**. If **PartitionedDomain** requires one specific **Subdomain** to carry out some operations, it will send out orders to the **ShadowSubdomain** associated with the target **Subdomain**. Then the **ShadowSubdomain** sets up communication channel to communicate with the **Subdomain** through **ActorSubdomain**. **ActorSubdomain**, on the other hand, sits on each child process as an agent receiving and processing incoming operation requests. The major improvements in this report include new functionality for adaptive repartitioning and data migration, and several other minor changes to reduce unnecessary data communications, such as when the **Subdomain** is required to **removeElement**, the new design won't send the element information out, etc. New features will be introduced in this section.

- **ShadowActorSubdomain_Partition**
  New design used ParMETIS to do parallel graph partitioning instead of sequential partitioning by METIS in old design. This improvement helps to reduce partition/repartition overhead and enable the parallel adaptive repartitioning for PDD algorithms proposed in this report .

- **ShadowActorSubdomain_BuildElementGraph**
  In order to provide input graphs for adaptive load balancing, all **Subdomain**s have to construct their own sub**ElementGraph**, which will be fed into ParMETIS routines for repartitioning.

- **ShadowActorSubdomain_Repartition**
  The repartitioning is implemented in parallel in this report so this entry point is set in the **ActorSubdomain** for each **Subdomain**.

- **ShadowActorSubdomain_reDistributeData**
  If data migration is needed to achieve load balancing, the master process will orchestrate the data redistribution process and the functionality here helps to facilitate the data communications between processes. This is one of the major additions to the existing design. Starting from this point, the **ActorSubdomain** is able to handle all required data movement on its own and **ActorSubdomain**s representing other **Subdomain**s will connect to the current working **ActorSubdomain** to receive/send data. Logically only one **ActorSubdomain** will be doing **ShadowActorSubdomain_reDistributeData** while others including the master will be listening to separate MPI port for data migration requests.

- **ShadowActorSubdomain_recvChangedNodeList**
  This function is used to simplify the data migration routine. With this function, only **Node**s/**Element**s and their associated **Constraint**s, **Load**s etc. need to be moved between processors.

- **ShadowActorSubdomain_ChangeMPIChannel**
  This function prepares the current **ActorSubdomain** for messages from some specific processes. It changes the destination/source for subsequent outgoing/incoming communications, which helps redistributing data after load balancing.

- **ShadowActorSubdomain_restoreChannel**
  The default communication pattern in the old design of OpenSees was one to one, master to slave. This function helps restoring communication patterns of the whole model after data redistribution finishes.

- **ShadowActorSubdomain_swapNodeFromInternalToExternal**
  Nodes that only belongs to one single **Subdomain** is called internal nodes whose information will be stored only in that specific **Subdomain**. While for those nodes that belong to more than one **Subdomain**s, their information should be accessible from all **Subdomain**s with which the nodes are associated. Those nodes are called external nodes instead. It is possible that former internal nodes to one **Subdomain** become external after the adaptive repartitioning. What the old design would do is to remove the internal nodes from that **Subdomain**, gather the information back to the master process and then distribute it externally among those **Subdomain**s as indicated by the newly obtained partitions. The improvement in this report avoids unnecessary data communication between current working **Subdomain** and the master **Domain**. We can just swap the node in working **Subdomain** from internal status to external status and then export them to other specified **Subdomain**s. This new design can improve performance if the data migration is extensive by avoiding unnecessary communications.

- **ShadowActorSubdomain_swapNodeFromExternalToInternal**
  This function is introduced due to the same reason as described previously although now the swapping direction is in reverse. It is noted that along with the swapping, removing operations must be invoked for those **Subdomain**s that does not contain the node anymore.

- **ShadowActorSubdomain_exportInternalNode**
  This function handles the situation when a **Node** does not belong to the current **Subdomain** after adaptive repartitioning. The node will be removed from current **Subdomain** and exported to other **Subdomain**s specified by the graph repartitioning. This again avoids the unnecessary data communication to/from master process by directly sending data to other **Subdomain**s.

- **ShadowActorSubdomain_resetRecorders**
  The **Recorder**s have to be reset after data migration to reflect component changes in each **Subdomain**.

3.4.9 Send/RecvSelf

As stated in previous sections, **Send/RecvSelf** must be provided by all domain components to finish data communication operations, such as **Node**s, **Element**s, **Load**s, **Constraint**s, **Material**s etc. In this report , new communication functions have been developed for **EightNodeBrick** element, **ElasticIsotropic3D** material, **Template3Dep/NewTemplate3Dep** material. The basic requirement to implement **Send/RecvSelf** is to replicate the source object instance in target process. For the old design, only one-step initial partitioning is performed and thus greatly simplifies the **Send/RecvSelf** routines because all the analysis-related information is null or void and only geometry-related data need to be transferred. But in this report , data migration is needed periodically to achieve load balance so the **Send/RecvSelf** has to be redesigned to carry analysis-related information besides the geometry model data. This is extremely important for **Element** and **Material** classes because they contain intermediate iteration/solution data of nonlinear finite element simulations. Figure 3.15 shows the class diagrams of brick **Element** and the associated **Template3Dep** material model. **Send/RecvSelf** operations have been implemented also for all classes associated with **Template3Dep** which are necessary to define a complete material model, such as Cam Clay, Drucker Prager and von Mises **PotentialSurface**s, Cam Clay, Drucker Prager and von Mises **YieldSurface**s, linear and nonlinear isotropic and kinematic hardening rules, etc.

3.5 Graph Partitioning

Graph partitioning approach has been extensively used in implementing domain decomposition type parallel finite element method. The element-based graph naturally becomes the favorite due to the fact that elemental operation forms the fundamental calculation unit in finite element analysis.

In this report , element graph has been constructed upon which graph partitioning algorithm acts to get domain decomposition for parallel finite element analysis. In the current implementation of this report , vertices of the element graph represent elements of the analysis model. Vertex weight is then specified as the computational load of each element. In elastic-plastic finite element simulations, the most expensive part has shown to be the elemental level calculations, which include constitutive-level stress update (strain-driven constitutive driver assumed) and formulation of elastic-plastic modulus (or so-called tangent stiffness tensor/matrix). In this research, the wall clock time used by elemental calculations has been dynamically collected and specified as the corresponding vertex weight for each element. The elemental calculation time clearly tells whether the element is elastic or plastified. With this timing metric, the graph can effectively reflect load distribution among elements thus load balancing repartition can be triggered on the graph to redistribute element between processors to achieve more balanced elastic-plastic calculation.

On the other hand, vertex size has to be defined for repartitioning problem as mentioned in previous sections. In this research, vertex size has been specified to be redistribution cost associated with each element. This information depends on the parallel implementation of the software and is discussed in the

section immediately following.

3.5.1 Construction of Element Graph

Each element is considered as one vertex in the element graph. An edge is formed when two elements share a common node. In this report , the graph structure is assumed to be undirected, which means the same edge will be added to both vertex ends. The edge is weightless in our application considering the fact that the purpose of minimizing edge-cut is to reduce the data migration when assembling global stiffness matrix. In that sense, the edge of element graph should carry the same weight or, more directly no weight at all.

3.5.2 Interface to ParMETIS/METIS

Interfaces to both ParMETIS and METIS have been implemented in this report . ParMETIS is the parallel implementation of METIS and new adaptive repartitioning functionality is only available through ParMETIS.

All of the graph routines in ParMETIS/METIS take as input the adjacency structure of the graph, the weights of the vertices and edges (if any), and an array describing how the graph is distributed among the processors Karypis et al. (2003). The structure of the graph is represented by the compressed storage format (CSR), extended for the context of parallel distributed-memory computing. We will first describe the CSR format for serial graphs and then describe how it has been extended for storing graphs that are distributed among processors.

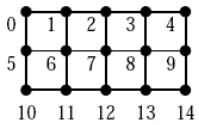
- **Serial CSR Format** The CSR format is a widely-used scheme for storing sparse graphs. Here, the adjacency structure of a graph is represented by two arrays, $xadj$ and $adjncy$. Weights on the vertices and edges (if any) are represented by using two additional arrays, $vwgt$ and $adjwgt$. For example, consider a graph with n vertices and m edges. In the CSR format, this graph can be described using arrays of the following sizes:

$$xadj[n+1], \quad vwgt[n], \quad adjncy[2m], \quad \text{and} \quad adjwgt[2m] \tag{3.1}$$

  Note that the reason both $adjncy$ and $adjwgt$ are of size $2m$ is because every edge is listed twice (i.e., as (v, u) and (u, v)). Also note that in the case in which the graph is unweighted (i.e., all vertices and/or edges have the same weight), then either or both of the arrays $vwgt$ and $adjwgt$ can be set to $NULL$. *ParMETIS_V3_AdaptiveRepart* additionally requires a $vsize$ array. This array is similar to the $vwgt$ array, except that instead of describing the amount of work that is associated with each vertex, it describes the amount of memory that is associated with each vertex.

  The adjacency structure of the graph is stored as follows. Assuming that vertex numbering starts from 0 (C style), the adjacency list of vertex i is stored in array $adjncy$ starting at index $xadj[i]$ and ending at (but not including) index $xadj[i+1]$ (in other words, $adjncy[xadj[i]]$ up through and

including $adjncy[xadj[i+1]-1]$). Hence, the adjacency lists for each vertex are stored consecutively in the array $adjncy$. The array $xadj$ is used to point to where the list for each specific vertex begins and ends. Figure 3.14(a) illustrates the CSR format for the 15-vertex graph shown in Figure 3.14(b). If the graph has weights on the vertices, then $vwgt[i]$ is used to store the weight of vertex i . Similarly, if the graph has weights on the edges, then the weight of edge $adjncy[j]$ is stored in $adjwgt[j]$. This is the format that is used by (serial) METIS library routines.



(a) A sample graph

Description of the graph on a serial computer (serial MeTiS)

xadj	0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44
adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13

(b) Serial CSR format

Description of the graph on a parallel computer with 3 processors (ParMeTiS)

Processor 0:	xadj	0 2 5 8 11 13
	adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9
	vtxdist	0 5 10 15
Processor 1:	xadj	0 3 7 11 15 18
	adjncy	0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14
	vtxdist	0 5 10 15
Processor 2:	xadj	0 2 5 8 11 13
	adjncy	5 11 6 10 12 7 11 13 8 12 14 9 13
	vtxdist	0 5 10 15

(c) Distributed CSR format

Figure 3.14: An example of the parameters passed to PARMETIS in a three processor case Karypis et al. (2003).

- **Distributed CSR Format** ParMETIS uses an extension of the CSR format that allows the vertices of the graph and their adjacency lists to be distributed among the processors. In particular, PARMETIS assumes that each processor P_i stores n_i consecutive vertices of the graph and the corresponding m_i edges, so that $n = \sum_i n_i$, and $2m = \sum_i m_i$. Here, each processor stores its local part of the

graph in the four arrays $xadj[n_i + 1]$, $vwgt[n_i]$, $adjncy[m_i]$, and $adjwgt[m_i]$, using the CSR storage scheme. Again, if the graph is unweighted, the arrays $vwgt$ and $adjwgt$ can be set to $NULL$. The straightforward way to distribute the graph for PARMETIS is to take n/p consecutive adjacency lists from $adjncy$ and store them on consecutive processors (where p is the number of processors). In addition, each processor needs its local $xadj$ array to point to where each of its local vertices' adjacency lists begin and end. Thus, if we take all the local $adjncy$ arrays and concatenate them, we will get exactly the same $adjncy$ array that is used in the serial CSR. However, concatenating the local $xadj$ arrays will not give us the serial $xadj$ array. This is because the entries in each local $xadj$ must point to their local $adjncy$ array, and so, $xadj[0]$ is zero for all processors. In addition to these four arrays, each processor also requires the array $vtxdist[p + 1]$ that indicates the range of vertices that are local to each processor. In particular, processor P_i stores the vertices from $vtxdist[i]$ up to (but not including) vertex $vtxdist[i + 1]$.

Figure 3.14(c) illustrates the distributed CSR format by an example on a three-processor system. The 15-vertex graph in Figure 3.14(a) is distributed among the processors so that each processor gets 5 vertices and their corresponding adjacency lists. That is, Processor Zero gets vertices 0 through 4, Processor One gets vertices 5 through 9, and Processor Two gets vertices 10 through 14. This figure shows the $xadj$, $adjncy$, and $vtxdist$ arrays for each processor. Note that the $vtxdist$ array will always be identical for every processor. All five arrays that describe the distributed CSR format are defined in PARMETIS to be of type $idxtype$. By default $idxtype$ is set to be equivalent to type int (i.e., integers). However, $idxtype$ can be made to be equivalent to a $short\ int$ for certain architectures that use 64-bit integers by default. (Note that doing so will cut the memory usage and communication time required approximately in half.) The conversion of $idxtype$ from int to $short$ can be done by modifying the file *parmetis.h*. (Instructions are included there.) The same $idxtype$ is used for the arrays that store the computed partitioning and permutation vectors.

When multiple vertex weights are used for multi-constraint partitioning, the c vertex weights for each vertex are stored contiguously in the $vwgt$ array. In this case, the $vwgt$ array is of size nc, where n is the number of locally stored vertices and c is the number of vertex weights (and also the number of balance constraints).

New **GraphPartitioner** class **ParMETIS** has been developed in this report to provide seamless interface to adaptive partitioning/repartitioning routines.

3.6 Data Redistribution

Data redistribution after repartitioning has been a challenging problem which needs careful study to guarantee correctness of subsequent analysis. In this research, Object-Oriented philosophy has been followed to abstract container classes to facilitate analysis and model data redistribution after repartition. As for the

initial partitioning, only model data, such as geometry parameters, has to be exported to sub-processors, while in adaptive repartitioning finite element simulation, analysis data has to be moved as well. It is extremely important to have well-designed container classes to carry data around. Basic units of finite element analysis, such as nodes and elements naturally become our first choices. Not to give up generality, the design in OpenSees adopts basic iterative approach for nonlinear finite element analysis Crisfield (1997), important intermediate analysis data include trial data, commit data, incremental data, element residual, element tangent stiffness, etc. Vertex size of each element has been defined as the total number of bytes that have to be transferred between sub-processors.

1. **Node**

   Other than geometric data such as node coordinates and number of degree of freedoms, the **Node** class contains nodal displacement data which should be sent together with the node to preserve continuity of the analysis model.

2. **Element**

   Element class is the basic construction unit in finite element model. In the design of this research, **Element** class keeps internal links to **Template3D** material class Jeremić and Yang (2002). In order to facilitate elastic-plastic simulation, **EPState** class is constructed to hold all the intermediate response data. This object-oriented abstraction greatly systematize the communication pattern. The information on class design is shown in the class diagram Figure 3.15 by Rational Rose Boggs and Boggs (2002).

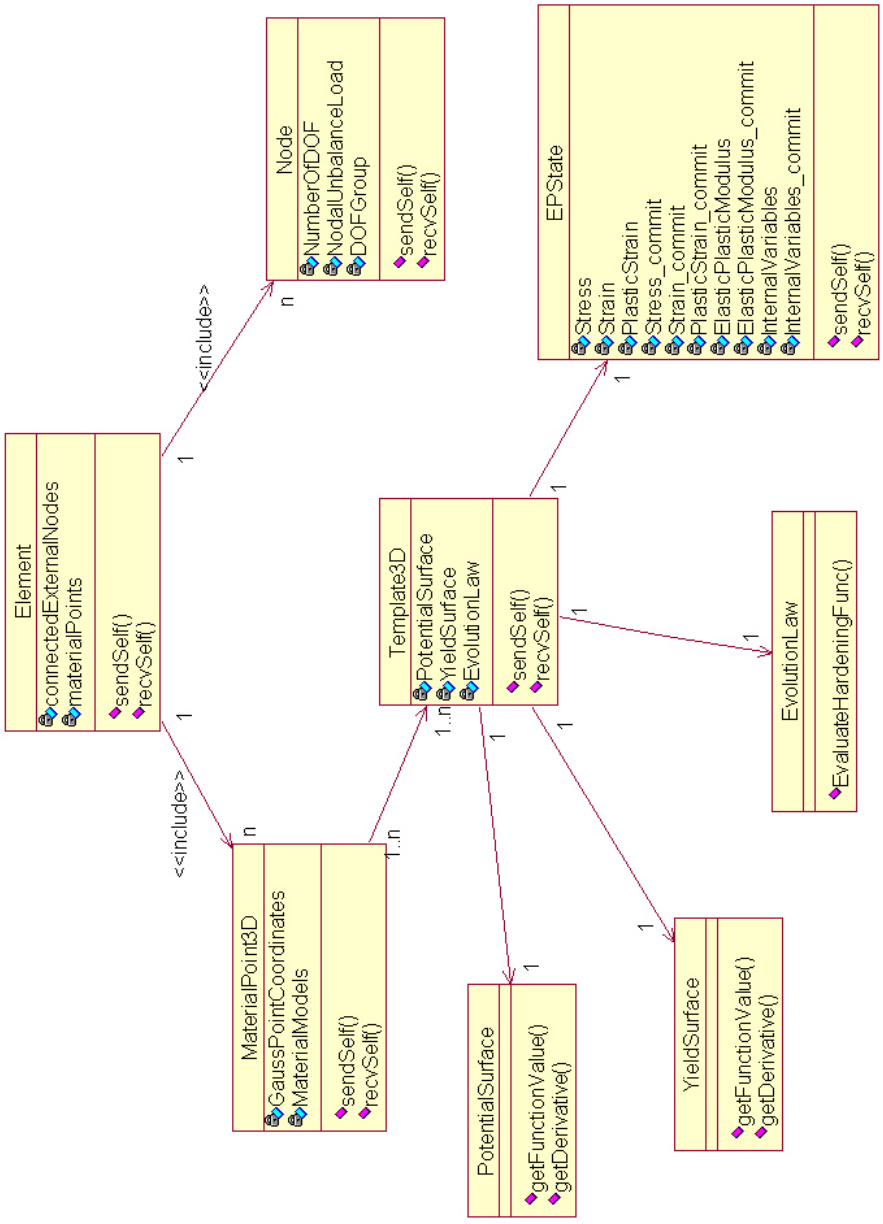


Figure 3.15: Class Diagram: Major Container Classes for Data Redistribution

All data communication operations have been implemented through the standard **Send/RecvSelf** interface, which forms a complete set of consistent point-to-point communication patterns and is convenient for future additions of new element/materials.

Chapter 4

Performance Studies on PDD Algorithm

4.1 Introduction

In this chapter, parallel performance of the proposed PDD algorithm is thoroughly investigated. There are two major focuses for the timing analysis. Firstly we want to see how much performance gain we can have by introducing the PDD algorithm into inelastic finite element calculations. Secondly, we also want to show how scalable the proposed PDD algorithm is.

As our final objective is to apply PDD in large scale SFSI finite element simulations, finite element models of SFSI have been set up to study the parallel performance of the PDD based parallel program. Implicit constitutive integration scheme Jeremić and Sture (1997) has been used to expose the load imbalance by plasticity calculation. Only continuum element has been studied due to the fact they can be easily visualized to obtain partition and/or repartition figures.

Distributed memory Linux/Unix clusters are major platforms used in this report for speed up analysis.

4.2 Parallel Computers

Performance measurement has been carried out on two SMP-based clusters.

- **IBM eServer p655**
  The DataStar IBM eServer p655 cluster consists of 176 8-way P655+ nodes at San Diego Supercomputer Center. System configuration is shown in Fig 4.1. The network benchmark is shown in Table 4.1.

- **TeraGrid IA-64 Intel-Based Linux Cluster**
  The TeraGrid project was launched by the the National Science Foundation with $53 million in funding to four sites: the National Center for Supercomputing Applications (NCSA) at the University of Illinois, Urbana-Champaign, the San Diego Supercomputer Center (SDSC) at the University of California, San Diego, Argonne National Laboratory in Argonne, IL, and Center for Advanced Computing

Table 4.1: Latency and Bandwidth Comparison (as of August 2004)

	MPI Latencies (μsec)	Bandwidth (MBs)
Intra-node	3.9	3120.4
Inter-node	7.65	1379.1

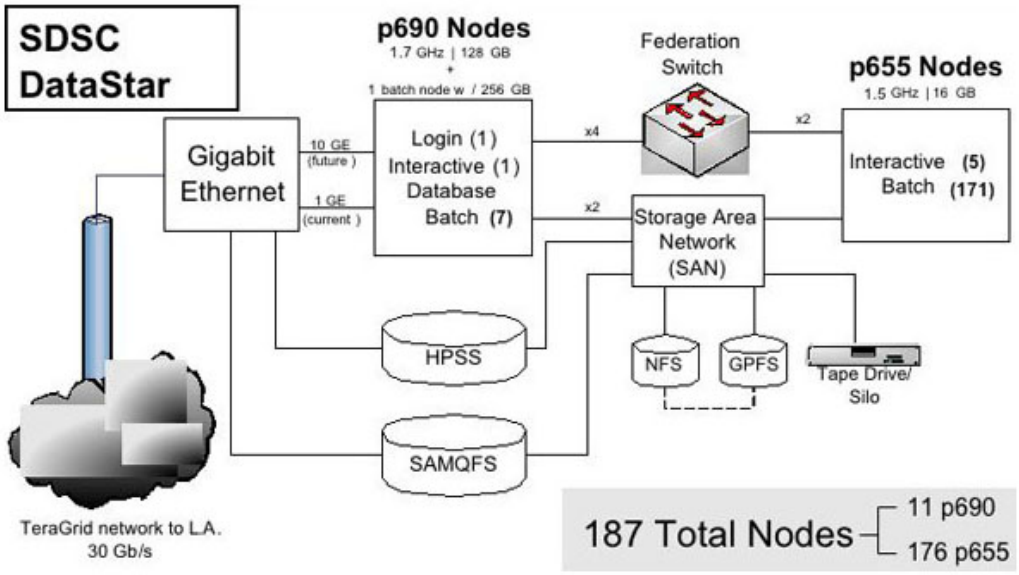


Figure 4.1: System Configuration of DataStar http://www.sdsc.edu/user_services/datastar/

Research (CACR) at the California Institute of Technology in Pasadena.

SDSC's TeraGrid cluster currently consists of 256 IBM cluster nodes, each with dual 1.5 GHz Intel® Itanium® 2 processors, for a peak performance of 3.1 teraflops. The nodes are equipped with four gigabytes (GBs) of physical memory per node. The cluster is running SuSE Linux and is using Myricom's Myrinet cluster interconnect network. Table 4.2 shows the technical configuration of the IA64 cluster, on which the second part of the performance study has been done.

Table 4.2: Technical Information of IA64 TeraGrid Cluster at SDSC

IA-64 Cluster (tg-login.sdsc.teragrid.org)	
COMPONENT	**DESCRIPTION**
Architecture	Linux Cluster
Access Nodes	⋆ quad-processor ⋆ ECC SDRAM memory: 8 GB ⋆ 2 nodes (8 processors)
Compute Nodes	⋆ dual-processor ⋆ ECC SDRAM memory: 4 GB ⋆ 262 nodes (524 processors)
Processor	⋆ Intel® Itanium® 2, 1.5 GHz ⋆ Integrated 6 MB L3 cache ⋆ Peak performance 3.1 Tflops
Network Interconnect	Myrinet 2000, Gigabit Ethernet, Fiber Channel
Disk	1.7 TB of NFS, 50 TB of GPFS (Parallel File System)
Operating System	Linux 2.4-SMP (SuSE SLES 8.0)
Compilers	⋆ Intel: Fortran77/90/95 C C++ ⋆ GNU: Fortran77 C C++
Batch System	Portable Batch System (PBS) with Catalina Scheduler

4.3 Soil-Foundation Interaction Model

A soil-shallow-foundation interaction model as shown in Figure 4.2 has been set up to study the parallel performance. 3D brick element with 8 integration (Gaussian) points is used. The soil is modeled by **Template3D** elasto-plastic material model (Drucker-Prager model with Armstrong Frederick nonlinear kinematic hardening rule) and linear elasticity is assumed for the foundation. More advanced constitutive laws can be applied through **Template3D** model although the model used here suffices the purpose of this research to show repartitioning triggered by plastification. It is shown in this research that the speedup by

adaptive load balancing is significant even for seemingly simple constitutive model. The material properties are shown in Table 4.3 and the vertical loading is applied at 5.0kN increments. The performance analysis has been carried out on DataStar supercomputer at San Diego Supercomputing Center (P655+ 8-way nodes).

Table 4.3: Material Constants for Soil-Foundation Interaction Model

Soil	
Elastic modulus	$E = 17400$kPa
Poisson ratio	$\nu = 0.35$
Friction angle	$\phi = 37.1^{\circ}$
Cohesion	$c = 0$
Isotropic Hardening	Linear
Kinematic Hardening	A/F nonlinear ($h_a = 116.0$, $C_r = 80.0$)
Foundation	
Elastic modulus	$E = 21$GPa
Poisson ratio	$\nu = 0.2$

4.4 Numerical Study for ITR

As described in Section 2.3.2, the parameter ITR in ParMETIS describes the ratio between the time required for performing the inter-processor communications incurred during parallel processing compared to the time to perform the data redistribution associated with balancing the load. It acts like a switch on algorithmic approaches of ParMETIS repartitioning kernel. With ITR factor being very small, the ParMETIS tends to do that repartitioning which can minimize data redistribution cost. If the ITR factor is set to be very large, ParMETIS tends to minimize edge-cut of the final repartition.

In parallel design of PDD, if repartitioning is necessary to achieve load balance after each load increment, the whole **AnalysisModel** McKenna (1997) has to be wiped off thus a new analysis container can be defined to reload subsequent analysis steps. The data redistribution cost can be much higher than communication overhead only. In order to determine an adequate ITR value for our application, preliminary study needs to be performed to investigate the effectiveness of the URA. In this research, two extreme values of the ITR (0.001 and $1,000,000$) are prescribed and then parallel analysis is carried out on 2, 4 and 8 processors to see how the partition/repartition algorithm behaves. Two soil-structure interaction models as shown in Figure 4.4 have been used in this parametric study. Timing data and partition figures have been collected to investigate the performance of different approaches. The one that tends to bring better performance will be adopted in subsequent parallel analysis for prototype 3D soil structure interaction problems. Figure 4.5

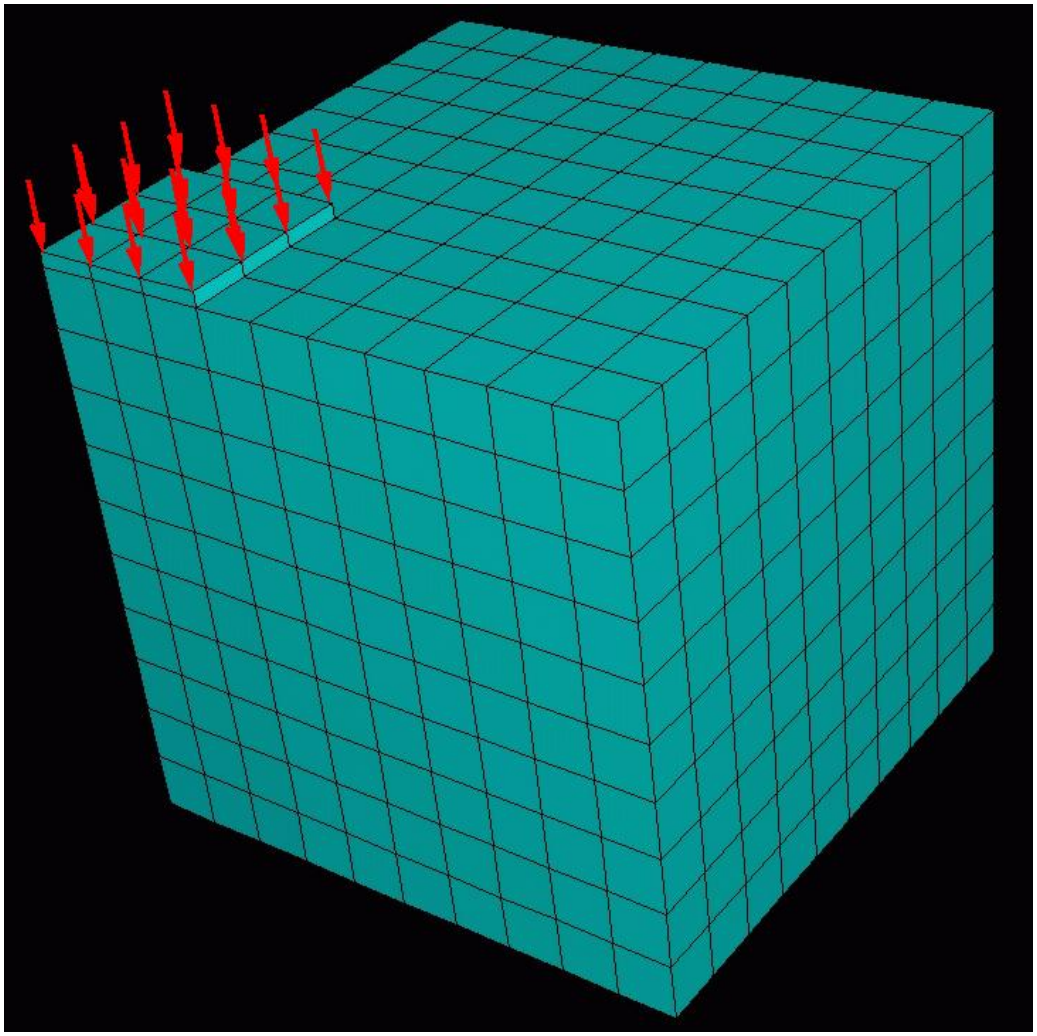


Figure 4.2: Example Finite Element Model of Soil-Foundation Interaction (Indication Only, Real Model Shown in Each Individual Section)

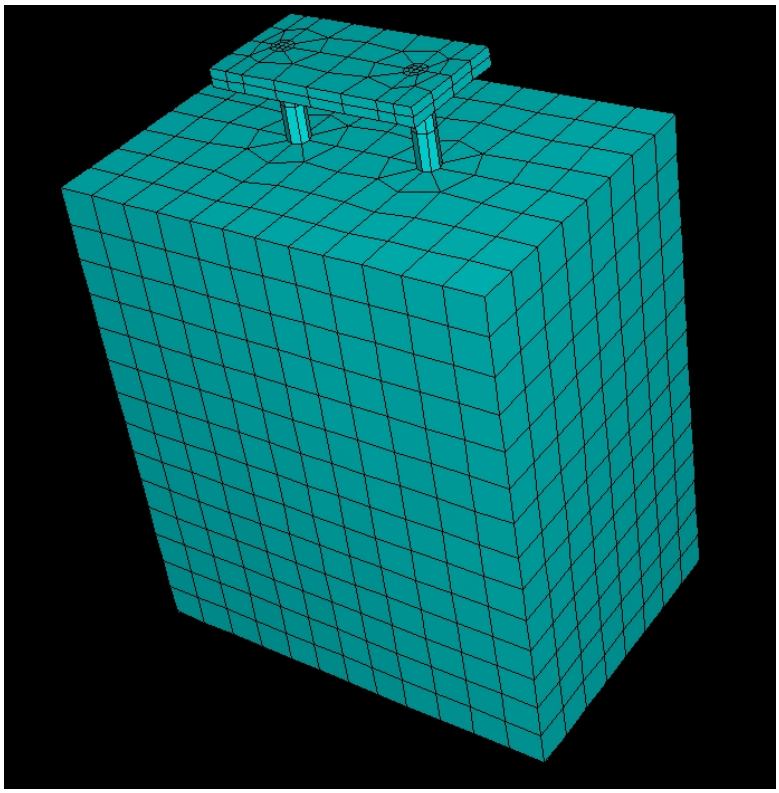


Figure 4.3: FE Models (1,968 Elements, 7,500 DOFs) for Studying Soil-Foundation Interaction Problems

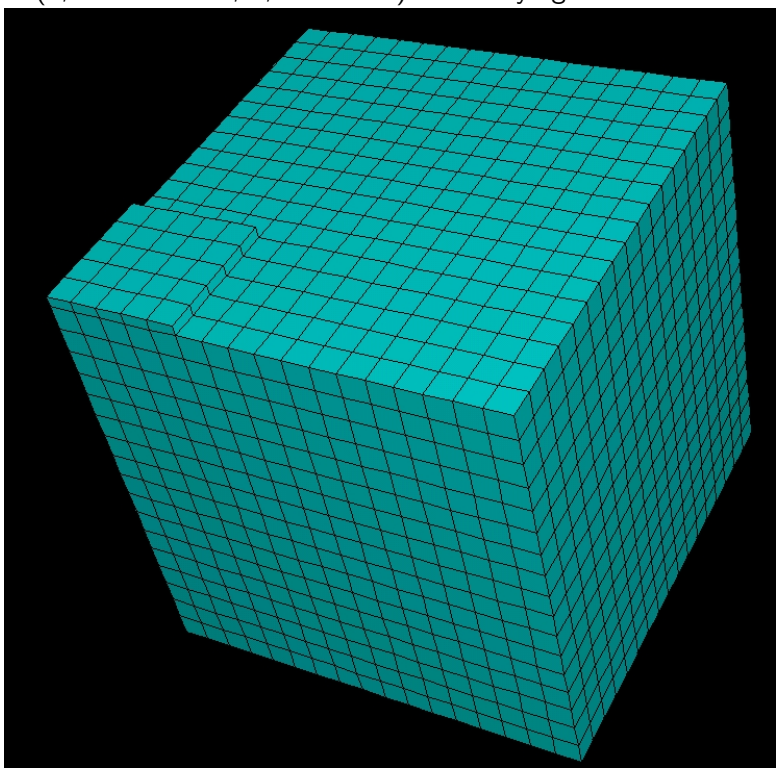


Figure 4.4: FE Models (4,938 Elements, 17,604 DOFs) for Studying Soil-Foundation Interaction Problems

to Figure 4.10 shows the initial partition and final repartition figures for two different types of algorithms. With ITR factor to be very small, the URA tends to present results that minimize data redistribution cost, in which diffusive repartitioning approach is used. On the other hand, if the ITR factor is set to be very large, then the URA algorithm tends to give repartitioning with lowest edge cut but with considerably higher data redistribution cost.

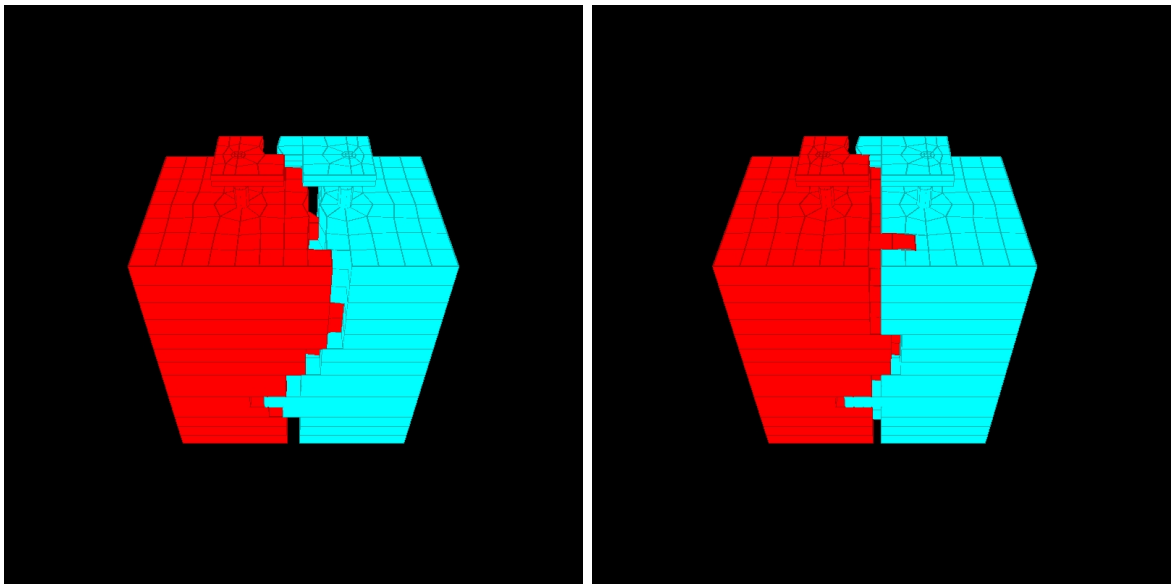


Figure 4.5: Partition and Repartition on 2 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

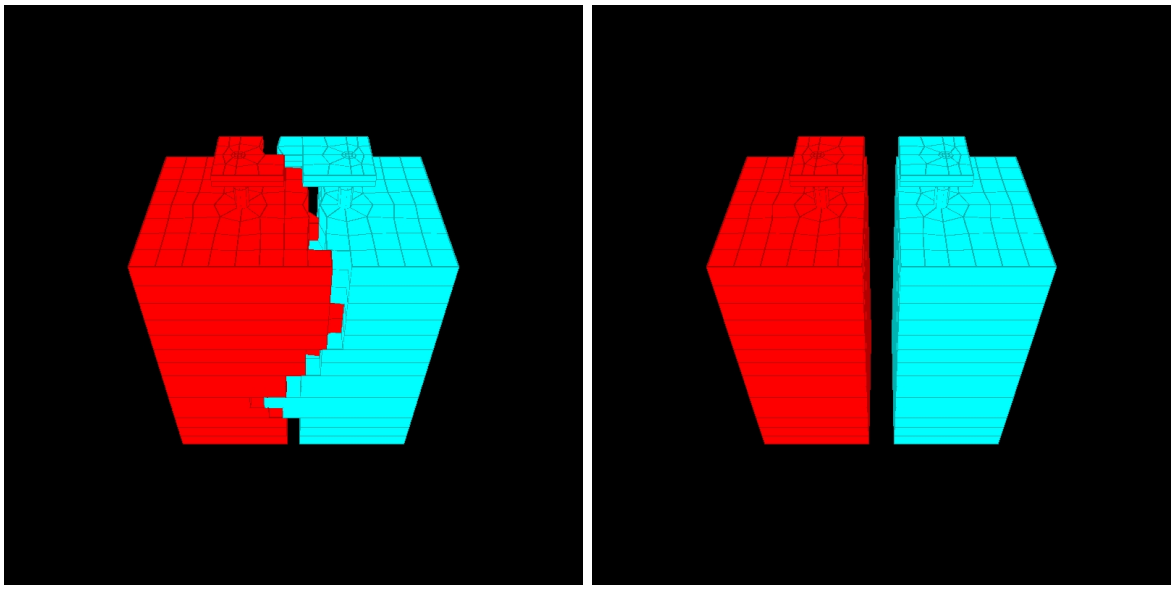


Figure 4.6: Partition and Repartition on 2 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

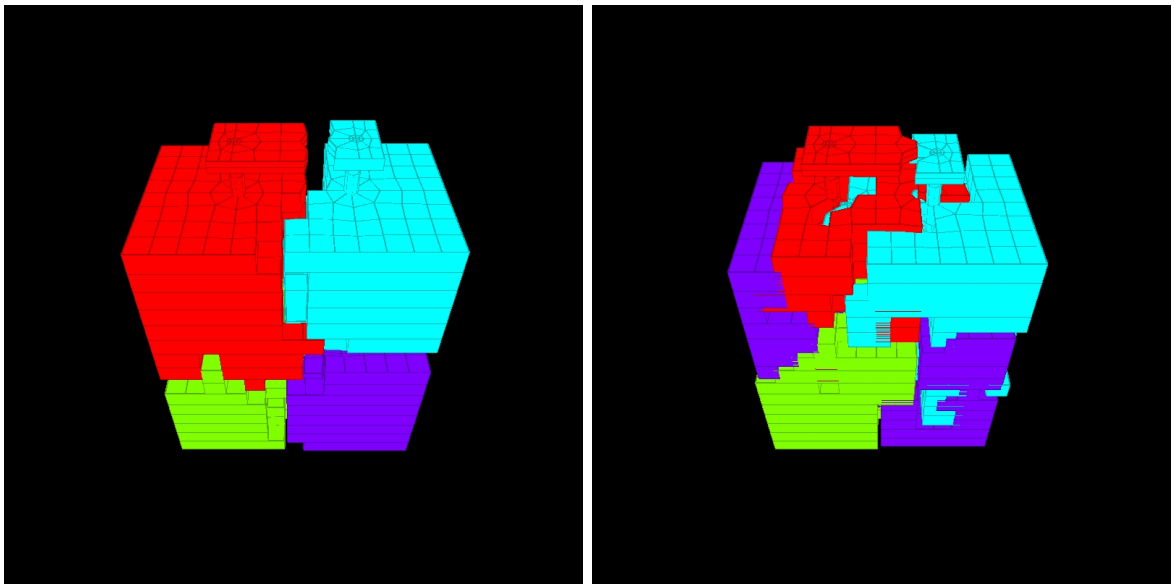


Figure 4.7: Partition and Repartition on 4 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

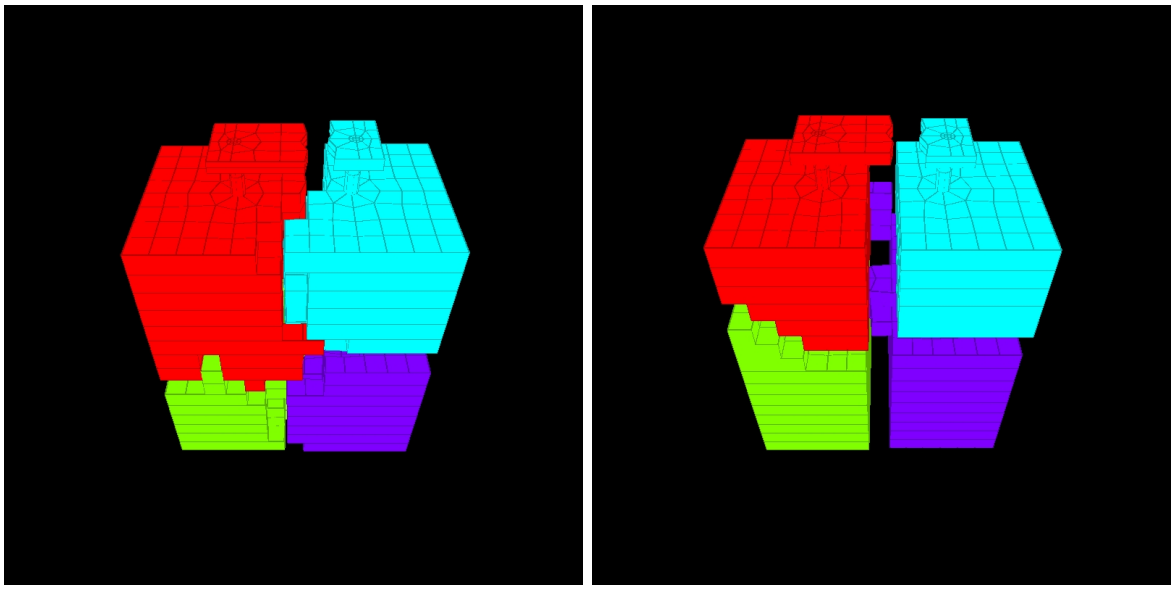


Figure 4.8: Partition and Repartition on 4 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

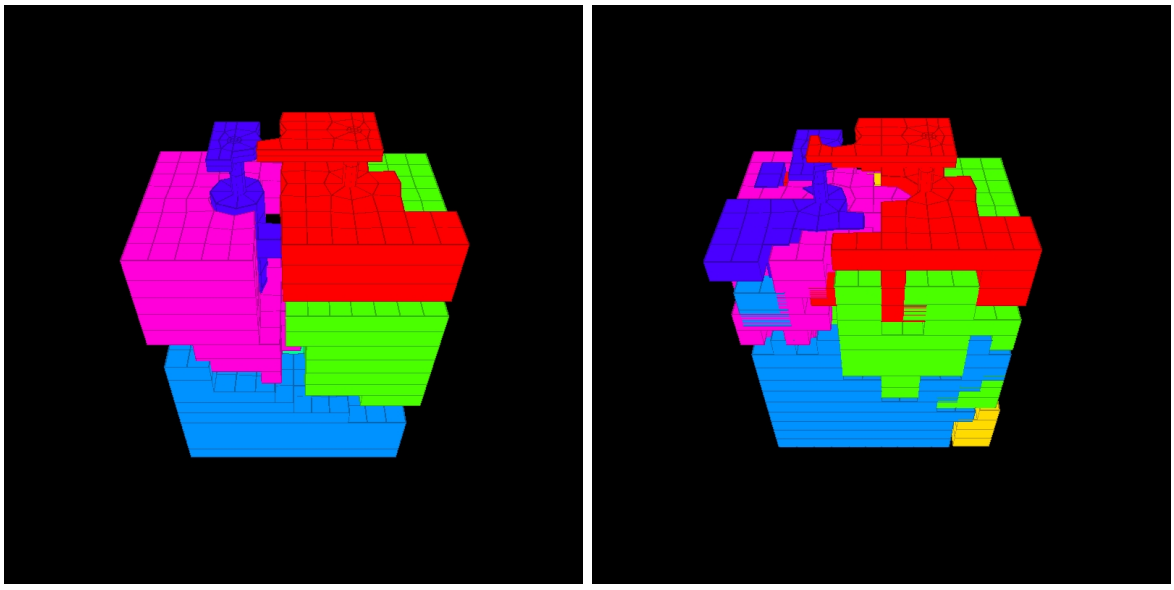


Figure 4.9: Partition and Repartition on 7 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

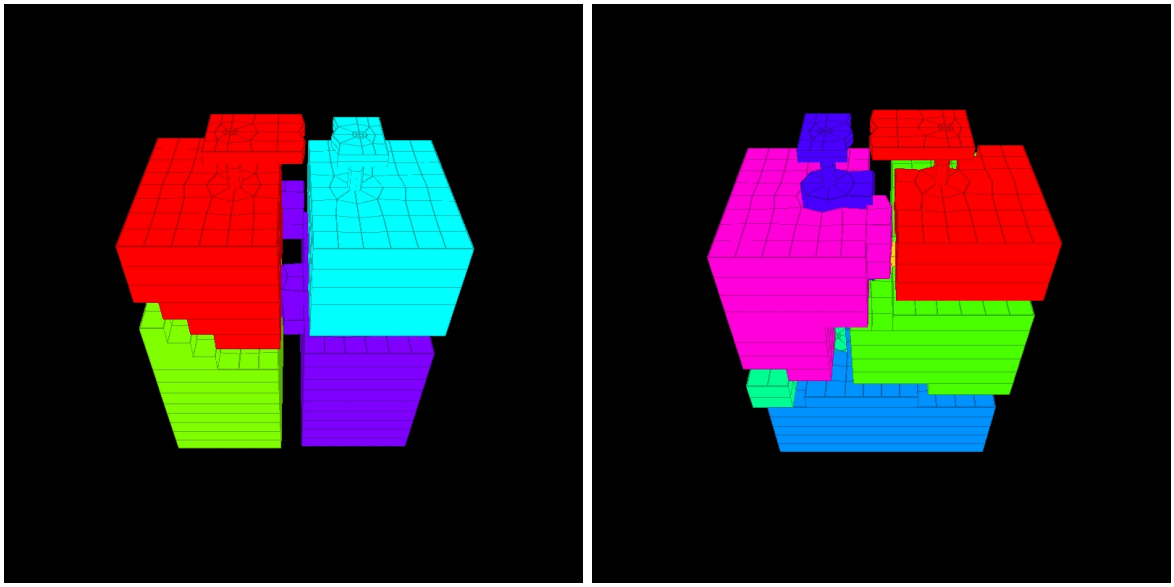


Figure 4.10: Partition and Repartition on 7 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

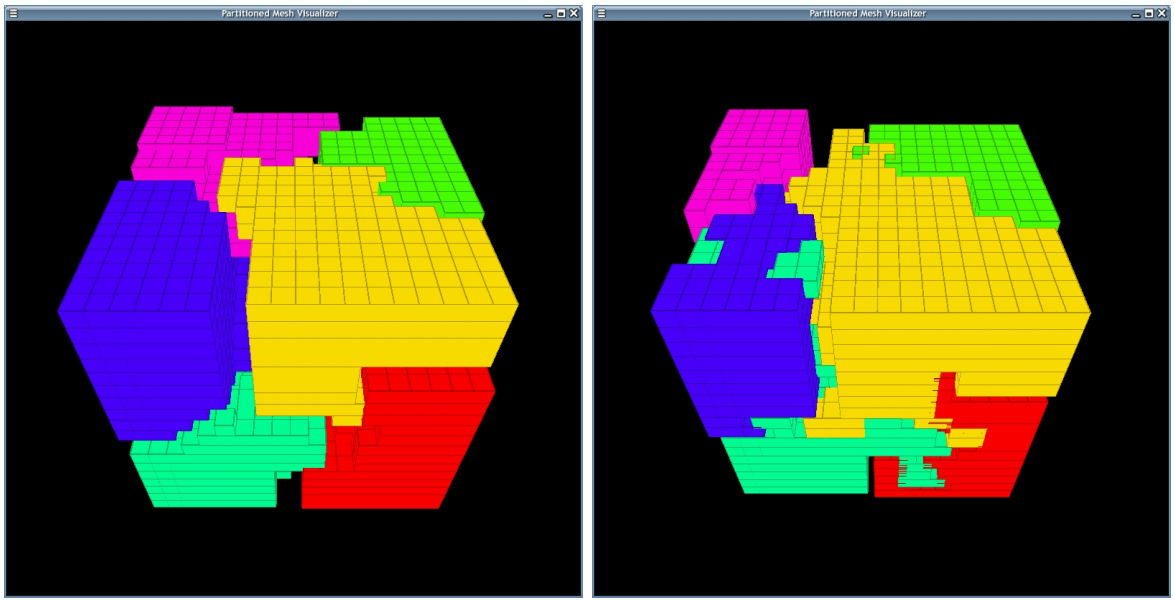


Figure 4.11: Partition and Repartition on 7 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (4,938 Elements, 17,604 DOFs)

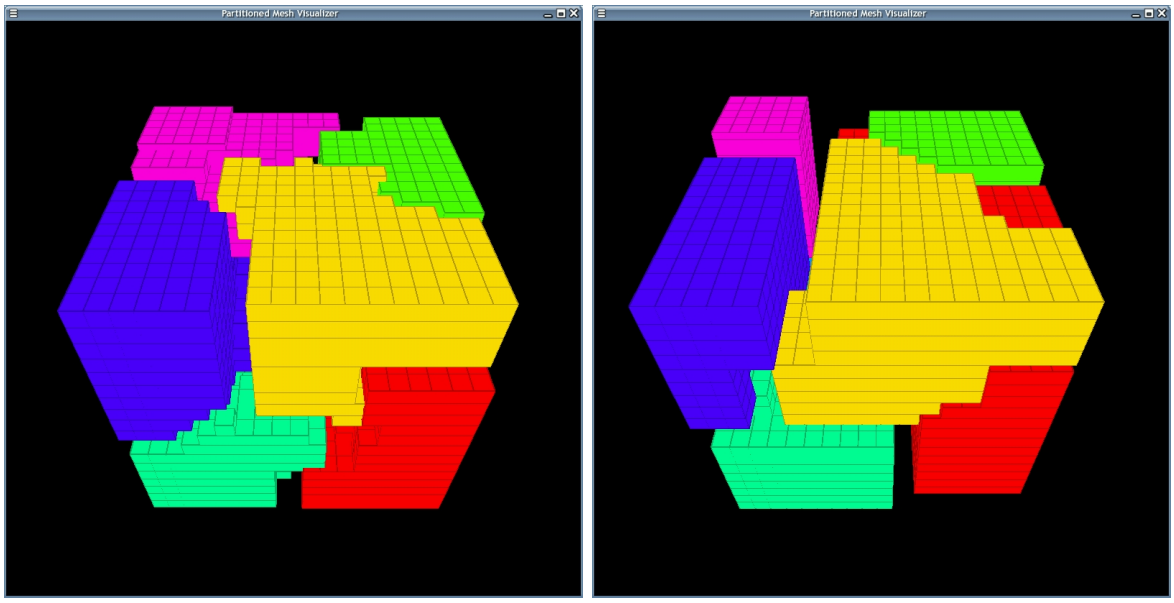


Figure 4.12: Partition and Repartition on 7 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (4,938 Elements, 17,604 DOFs)

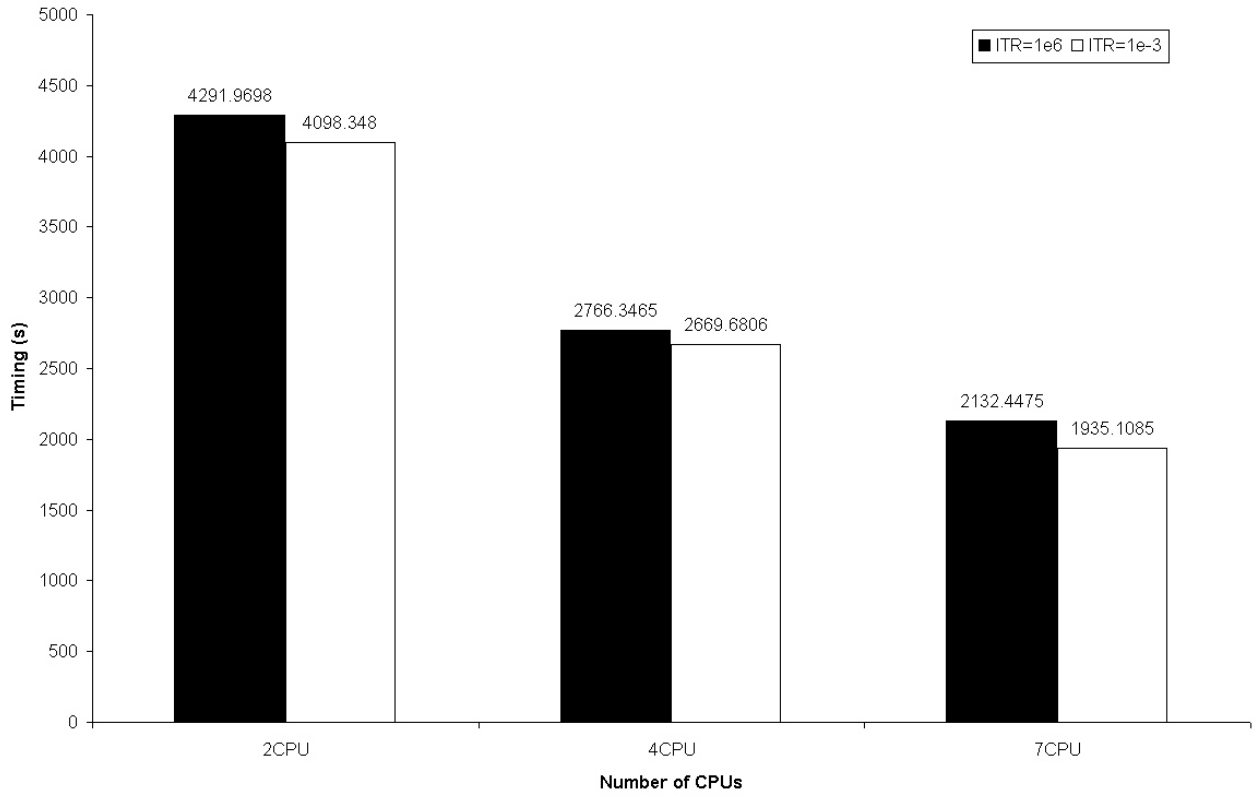


Figure 4.13: Timing Data of ITR Parametric Studies (1,968 Elements, 7,500 DOFs, Imbal. Tol. 5%)

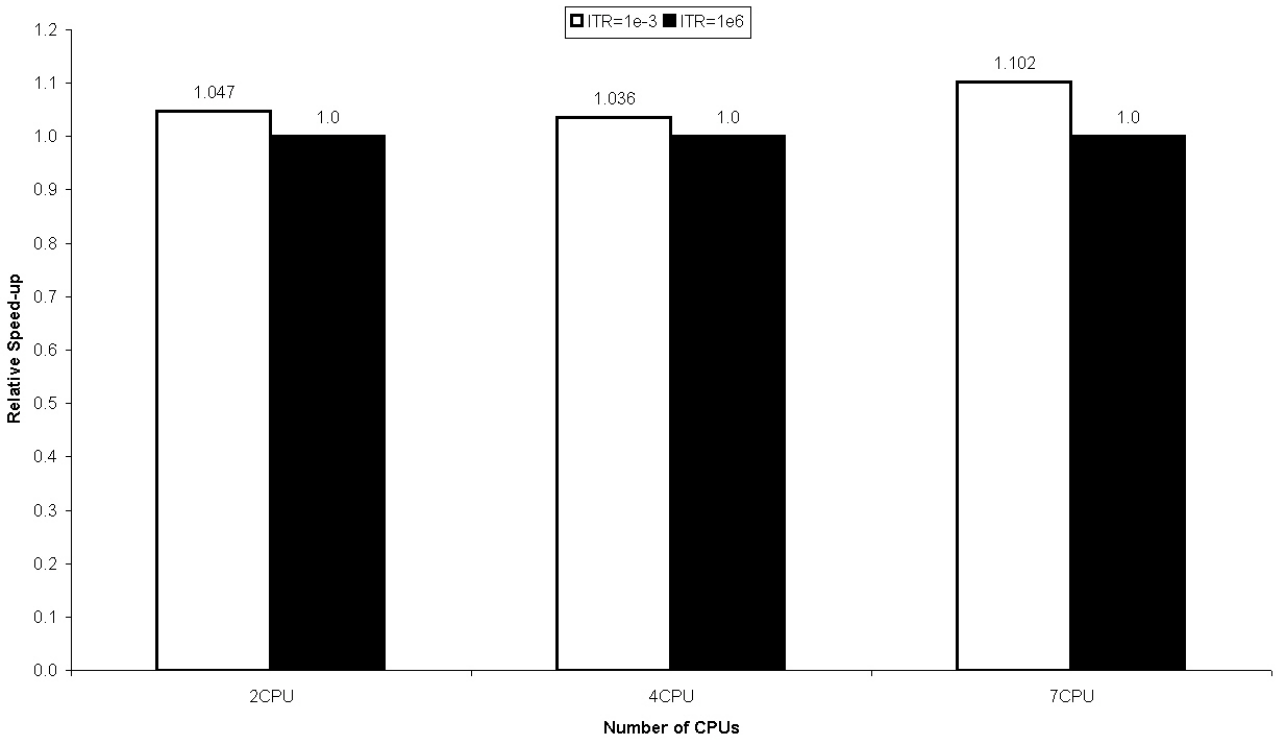


Figure 4.14: Relative Speedup of ITR=1e-3 over ITR=1e6 (1,968 Elements, 7,500 DOFs, Imbal. Tol. 5%)

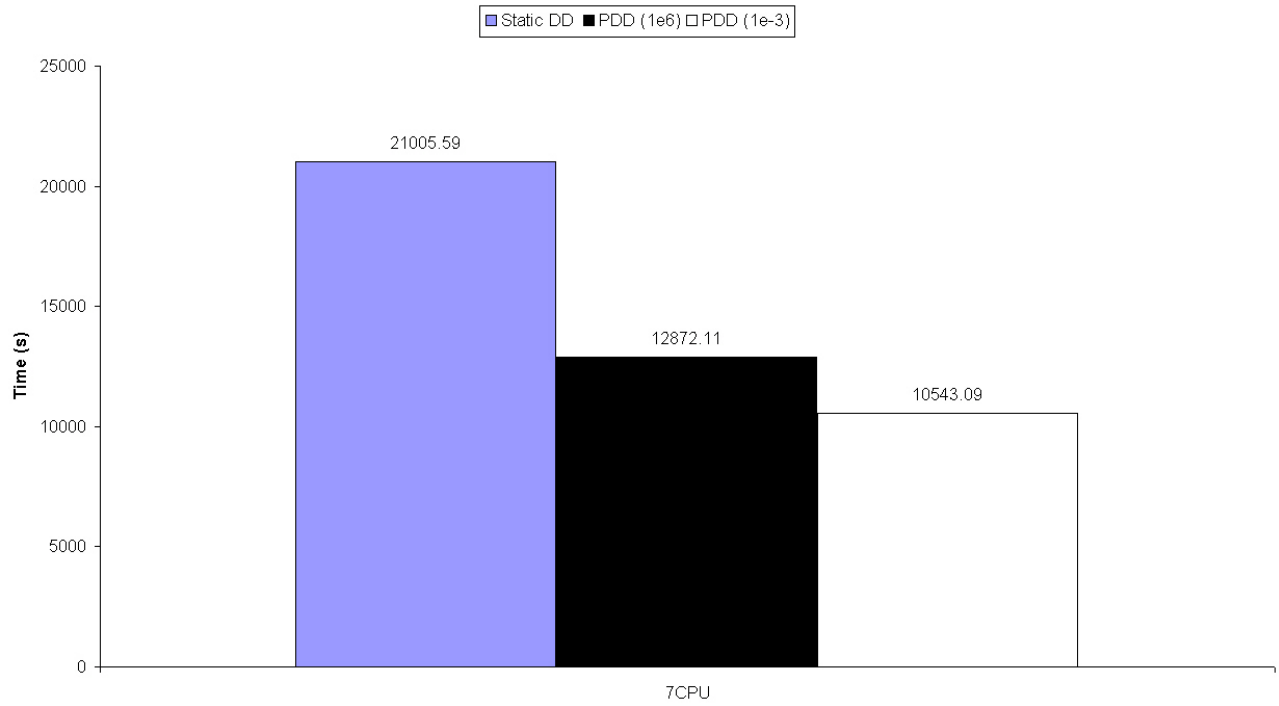


Figure 4.15: Timing Data of ITR Parametric Studies (4,938 Elements, 17,604 DOFs, Imbal. Tol. 5%)

Figures 4.13, 4.14 and 4.15 show the speedup data of parametric study on ITR factors. The purpose is to expose the the more efficient approach to do repartitioning for our specific parallel SFSI simulations, either scratch-remap approach ($ITR = 1e6$) or diffusive approach ($ITR = 1e-3$). Through the study of this report , some conclusions can be drawn.

1. Smaller value of ITR (1e-3) outperforms larger value (1e6). The performance gain is up to 22.1% for 7 processors. As the model gets larger, the speedup tends to get better.

2. With small ITR value, the URA algorithm tends to give results for diffusive partition/repartitioning scheme, which is good for performance for our application in overall due to the fact that the overhead associated with data redistribution in this research is very high. Diffusive approach minimizes possible data movement thus delivers better performance. The drawback is the diffusive approach typically gives very bad or even disconnected graphs with very high edge-cut as shown in Figures 4.5, 4.7 and 4.9. So careful attention must be paid to these graph structures when programing the finite element calculation. In this sense, the diffusive algorithm is not as robust as scratch/remapping. One very important observation was that repetitive repartitionings tend to yield totally ill-connected graph.

3. With large ITR value, the URA algorithm adopts the scratch/remapping scheme which inevitably introduce huge data redistribution cost. But this approach gives high quality graph and the integrity of original graph is well preserved as shown Figures 4.6, 4.8 and 4.10. This will be of great meaning for parallel finite element method based on substructure-type methods. Another important observation was, the scratch/remapping approach performed much more repartitionings than diffusive approach for same analysis. Repetitive repartitionings by scratch/remapping method tends to totally migrate all elements out of their initial partitioning and repartitioning never stops even though the computation is stabilized (in the sense of formation of plastic zones). This also explains in part why the diffusive approach can substantially outperform scratch/remapping.

4. Based on the timing analysis performed in this report , ITR=1e-3 is the best choice that brings substantially better performance over large ITR values. With the increase of number of processing units or the model size, the performance gain is more significant as shown in Figures 4.13, 4.14 and 4.15. Robustness of the diffusive approach has not caused much trouble in our application.

4.5 Parallel Performance Analysis

Timing routines have been implemented in PDD (and parts of OpenSees framework and other used libraries, such as Template3DEP/NewTemplate3Dep) to study the parallel performance. The preprocessing unit, like reading model data from file, has not been timed so the speed up here reflects only algorithmic gain by graph partitioning. In the current phase of this research, the equation solving problem has not been

addressed yet. More meaningful perspective would be to consider performance gains by simply switching from plain graph partitioning to adaptive graph partitioning, which is also the basic aim of this research. As we can see from the results below, adaptive graph partitioning improves the overall performance of elasto-plastic finite element computations. The partitioning/repartitioning overhead has been minimized by using parallel partitioner.

As stated in previous sections of this report , there are a couple of key parameters that control performance of the adaptive load balancing algorithm. One is the ITR factor, and the other is the computational load imbalance tolerance.

1. **ITR** is the key parameter which determines the algorithmic approach of the adaptive load balancing scheme. Depending on different applications and network interconnections, this value can be set to very small (0.001) or very large (up to 1,000,000) and algorithm focus will be set to minimizing data redistribution or edge-cut respectively as explained in previous sections.

2. **Computational Imbalance Load Tolerance** is the other key factor affecting greatly the overall performance of the whole application codes. Basically speaking, with larger finite element model, the tolerance should be set higher due to the fact that data redistribution and subsequent analysis-restarting overhead can be substantially higher as the finite element model size increases.

The performance tunings on ITR factor tend to yield consistent results as stated previously that smaller ITR (0.001) brings better performance over large ITR values. Diffusive repartitioning algorithm outperforms scratch/remapping in our application.

While on the other hand, tuning on load imbalance studies has been more illusive. The first conclusion is that load imbalance tolerance larger than 5% was not able to work robustly as the size of finite element model increases in the application study of this report .

Detailed parametric studies have been performed on DataStar IBM Power4 and IA64 Intel clusters to indicate the effectiveness of the proposed adaptive PDD algorithm. Models with different sizes have been tested on various number of processors to show the scalability of computational performance. All results will be compared with static one-step Domain Decomposition approach to investigate the advantage of proposed PDD algorithm in nonlinear elastic-plastic finite element calculations.

Table 4.4: Test Cases of Performance Studies

Model Sizes (DOF)	4,035, 17,604, 32,091, 68,451
# of CPUs	3, 5, 7, 16, 32, 64
ITR Factors	0.001, 1,000,000
Imbalance Tolerance	5%, 10%, 20%

In the following sections, timing data and partition/repartition figures will be presented and results will be discussed at the end of this chapter.

4.5.1 Soil-Foundation Model with 4,035 DOFs

The partition/repartition figures by PDD have been shown in Figure 4.16, 4.17, 4.18.

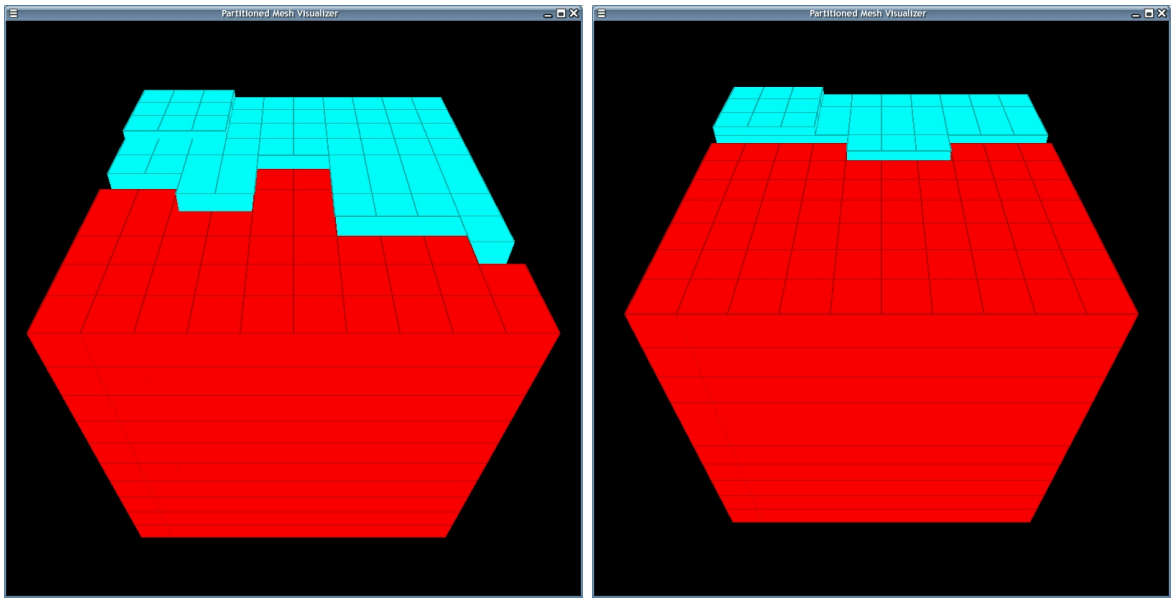


Figure 4.16: 4,035 DOFs Model, 2 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition

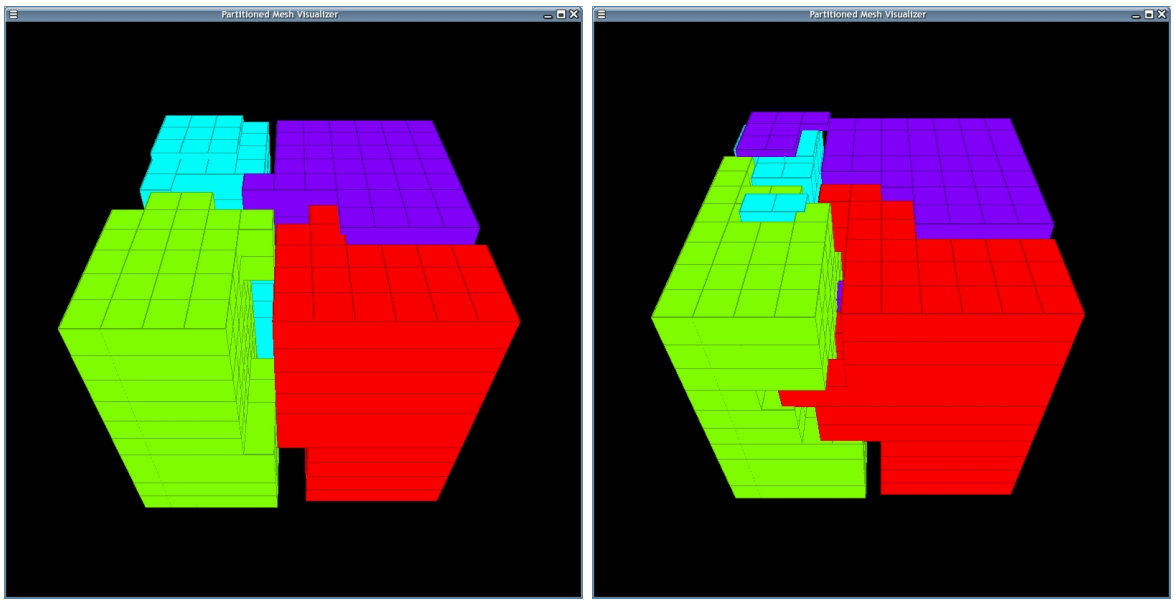


Figure 4.17: 4,035 DOFs Model, 4 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition

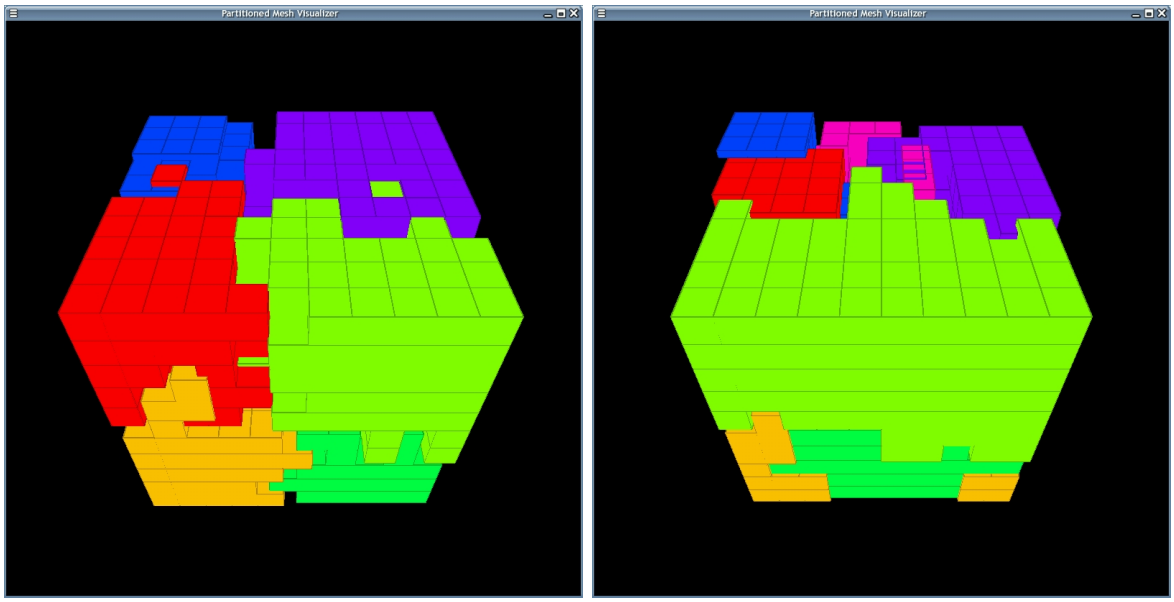


Figure 4.18: 4,035 DOFs Model, 8 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition

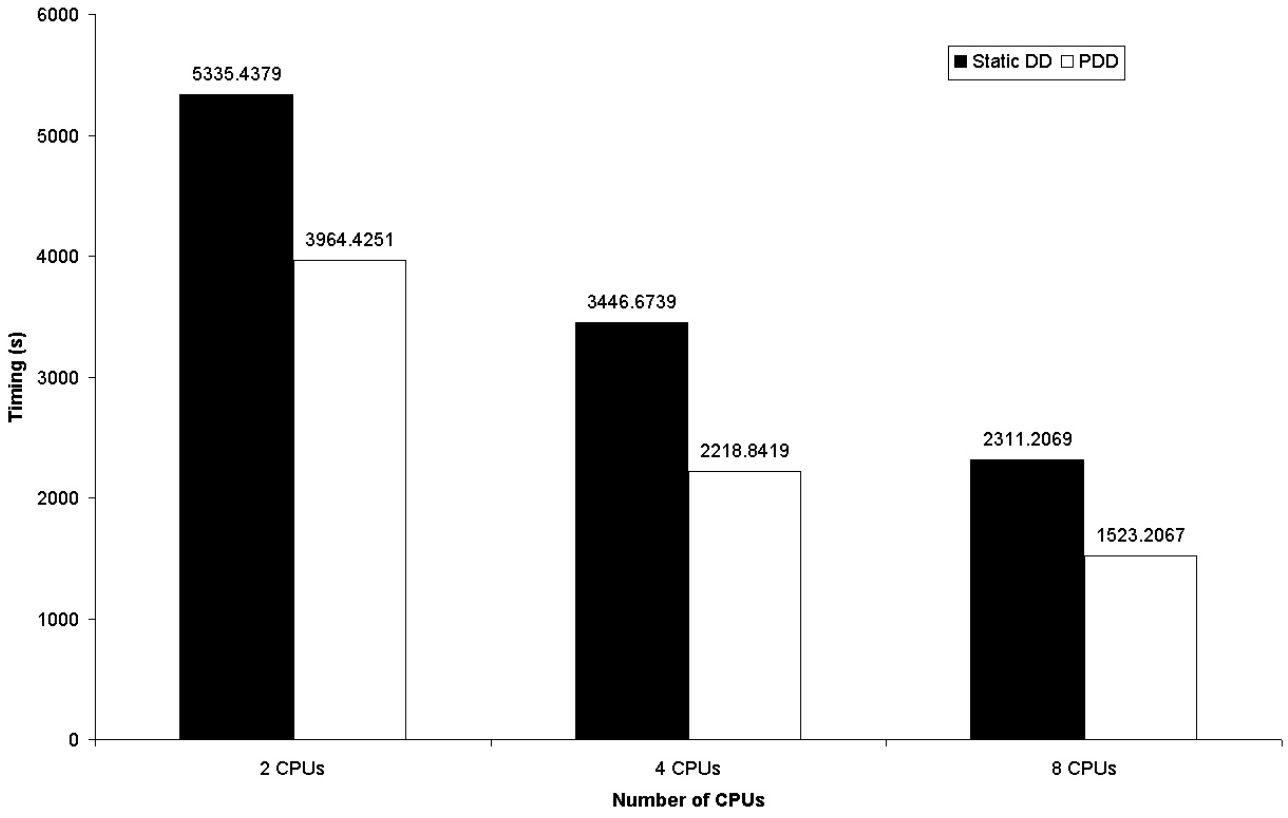


Figure 4.19: Timing Data of Parallel Runs on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5%

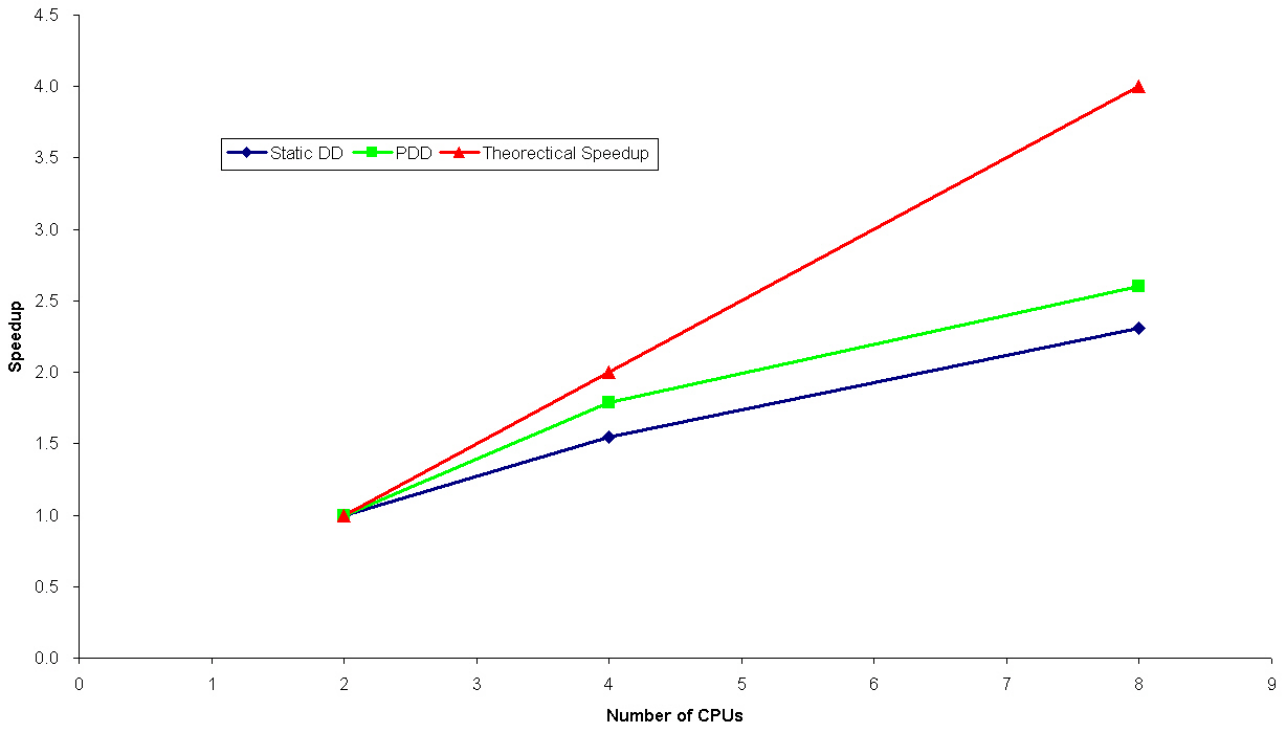


Figure 4.20: Absolute Speedup Data of Parallel Runs on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5%

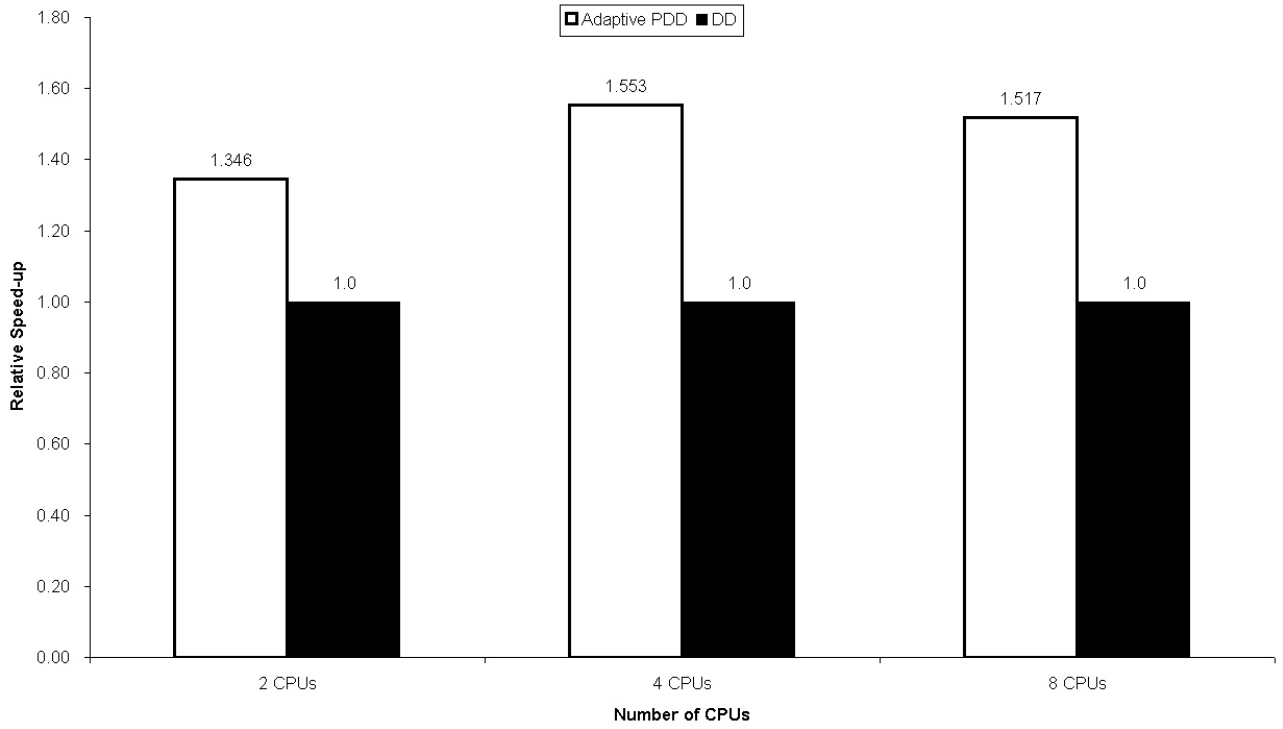


Figure 4.21: Relative Speedup of PDD over Static DD on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5%

4.5.2 Soil-Foundation Model with 4,938 Elements, 17,604 DOFs

This is the same model as described before but with more elements as shown in 4.22. Timing data has been collected to indicate performance gains by adaptive load balancing Partition and repartition figures are shown from Figure 4.26 to 4.28. The partition/repartition figures by PDD have been shown in

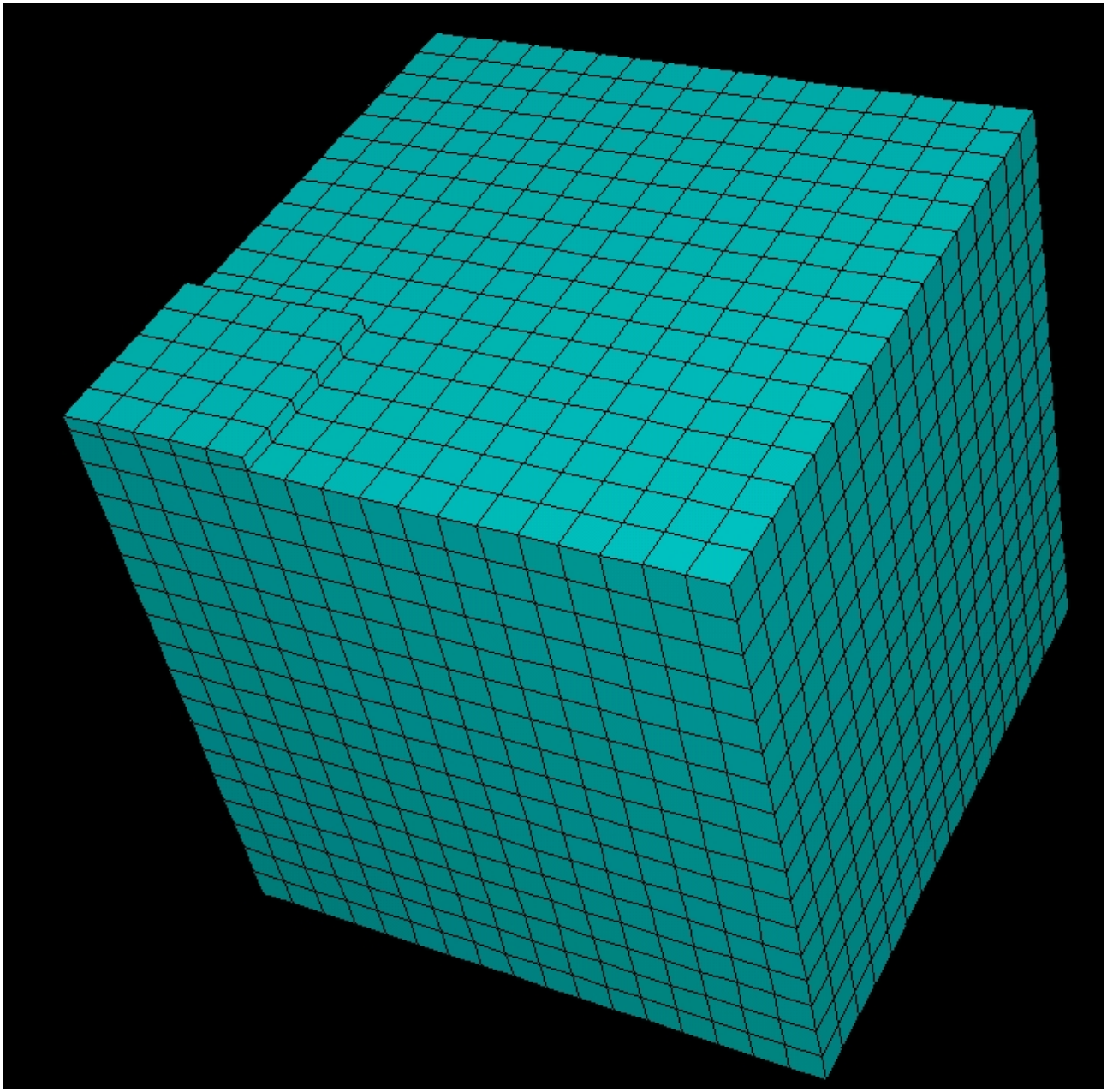


Figure 4.22: Finite Element Model of Soil-Foundation Interaction (4,938 Elements, 17,604 DOFs)

Figure 4.26, 4.27, 4.28.

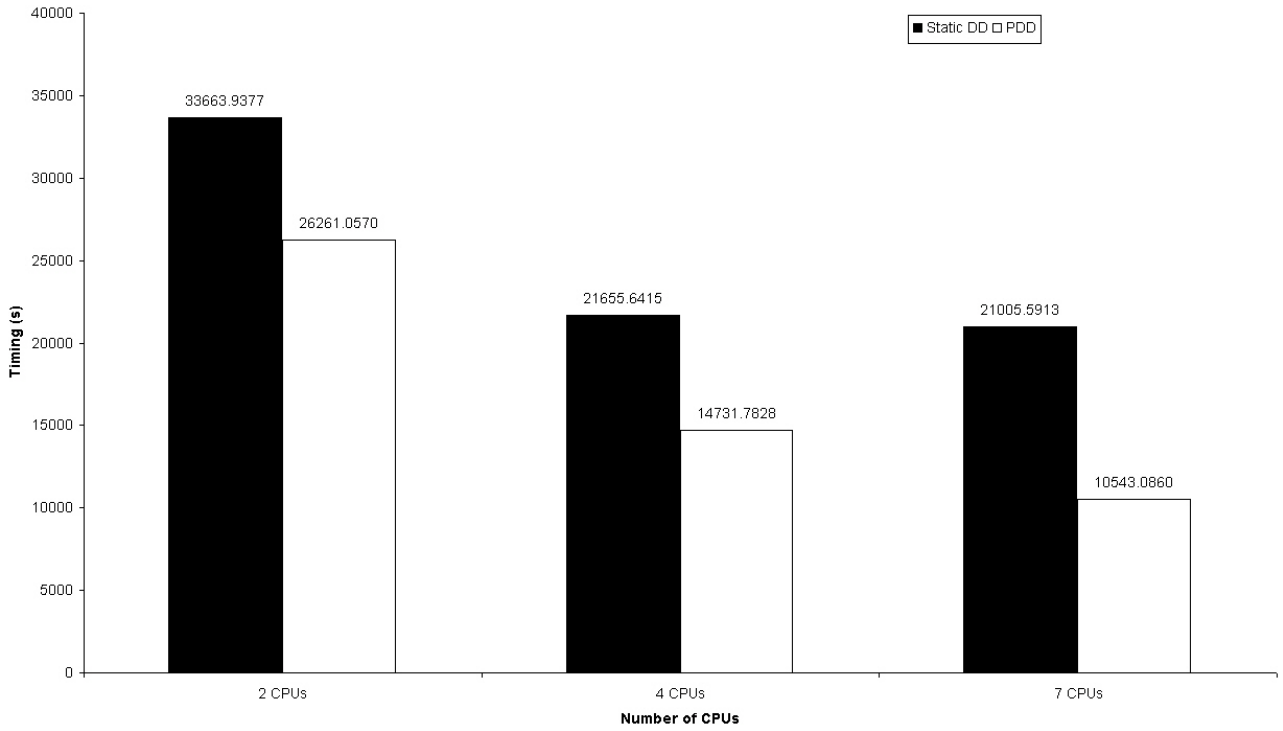


Figure 4.23: Timing Data of Parallel Runs on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

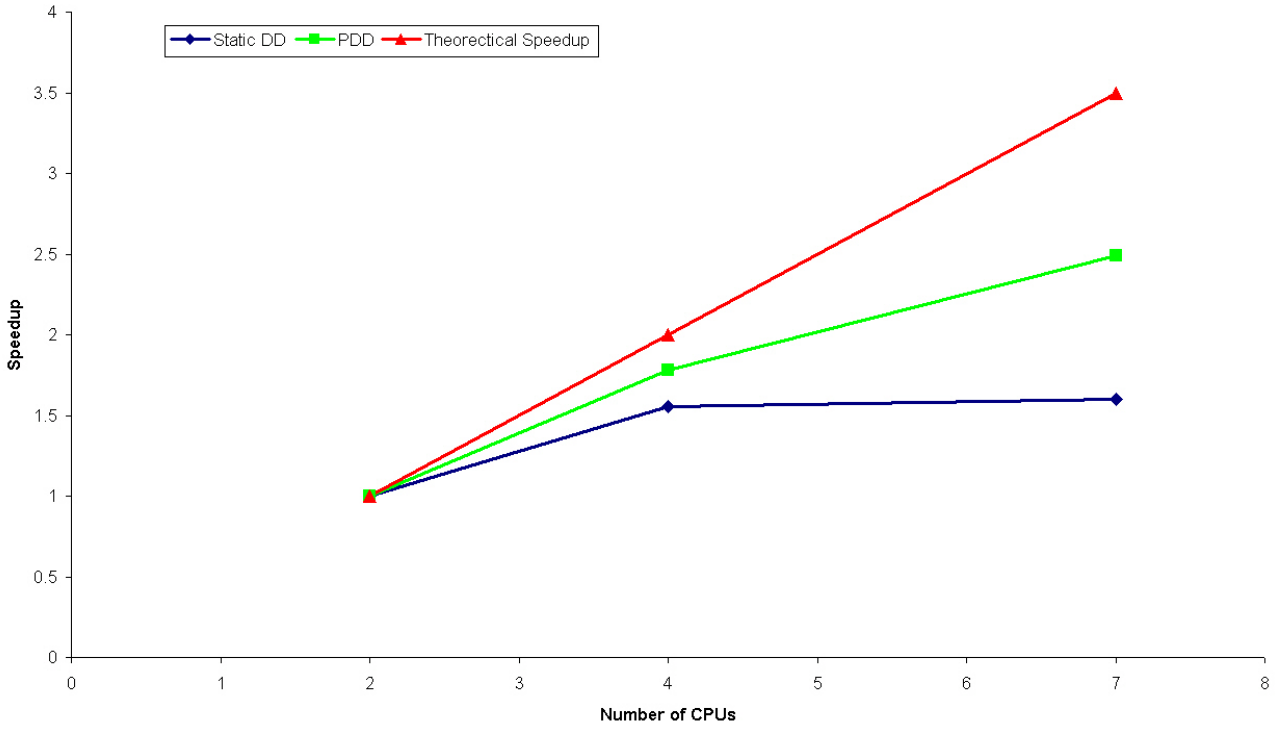


Figure 4.24: Absolute Speedup Data of Parallel Runs on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

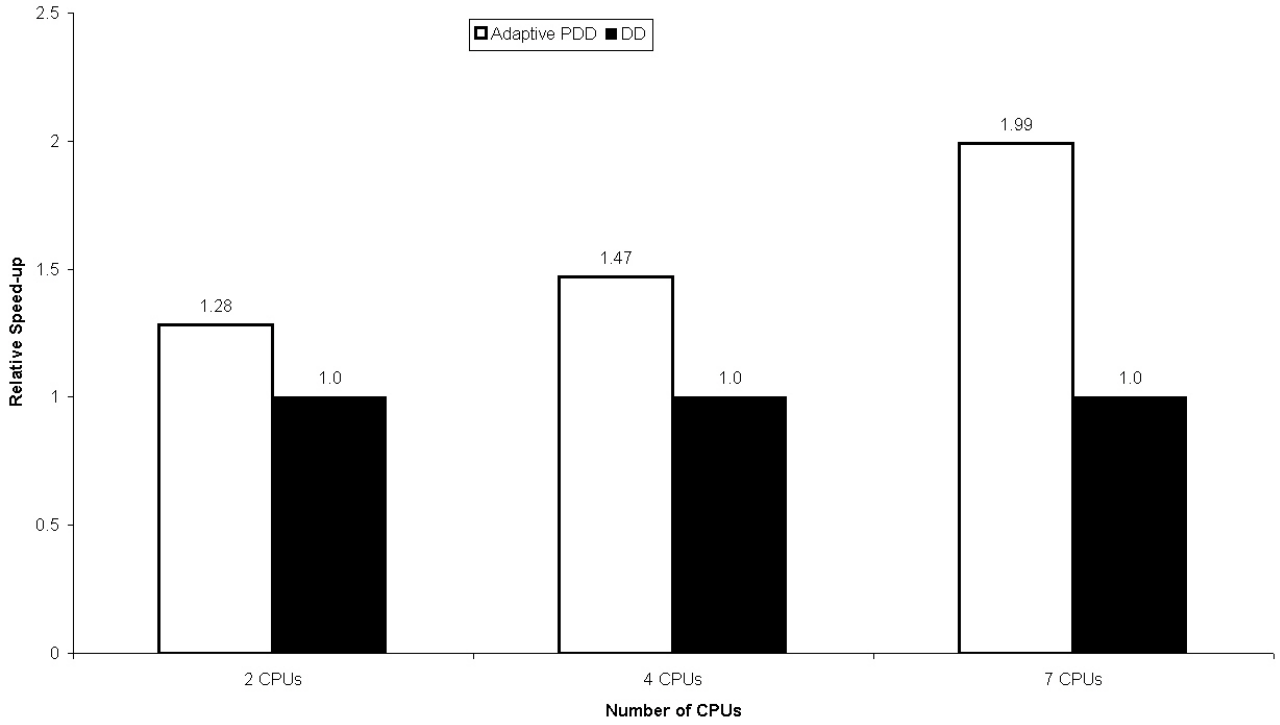


Figure 4.25: Relative Speedup of PDD over Static DD on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

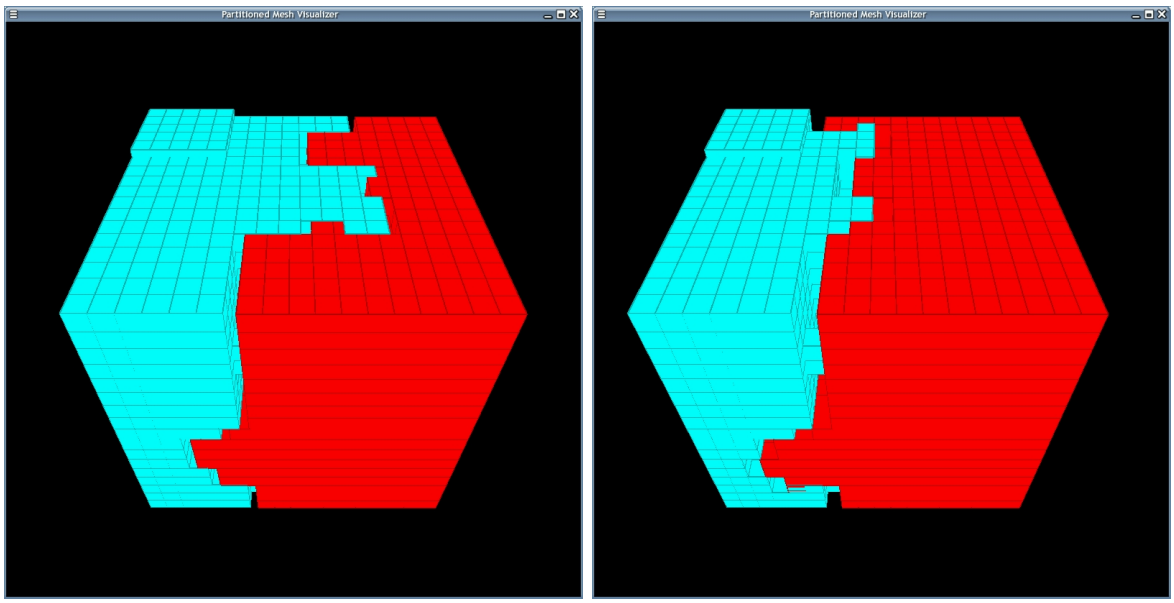


Figure 4.26: 4,938 Elements, 17,604 DOFs Model, 2 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

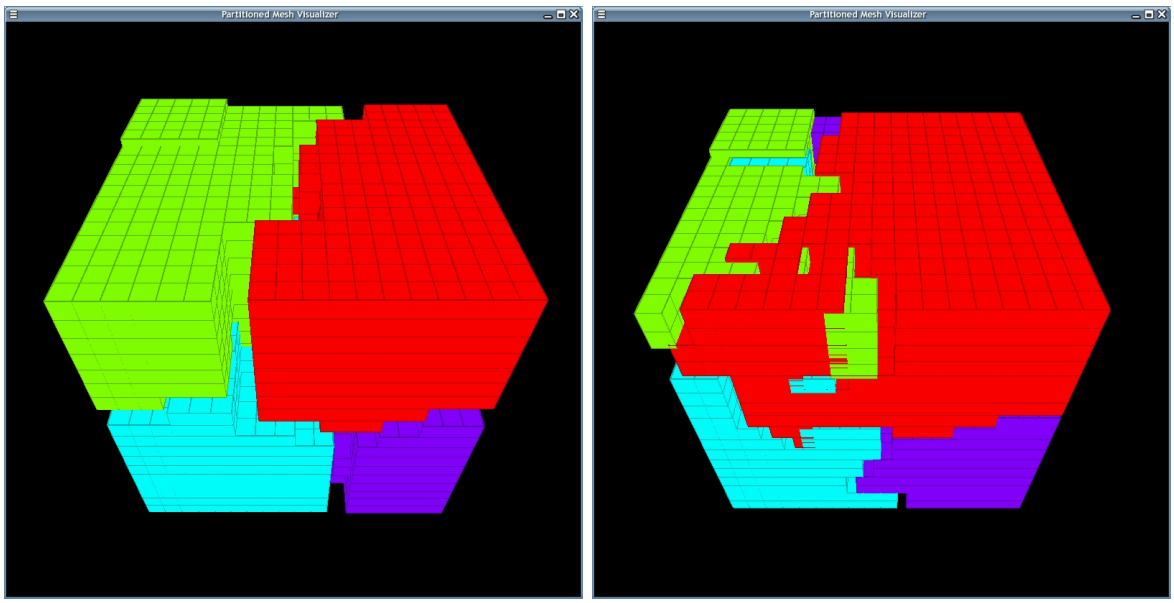


Figure 4.27: 4,938 Elements, 17,604 DOFs Model, 4 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

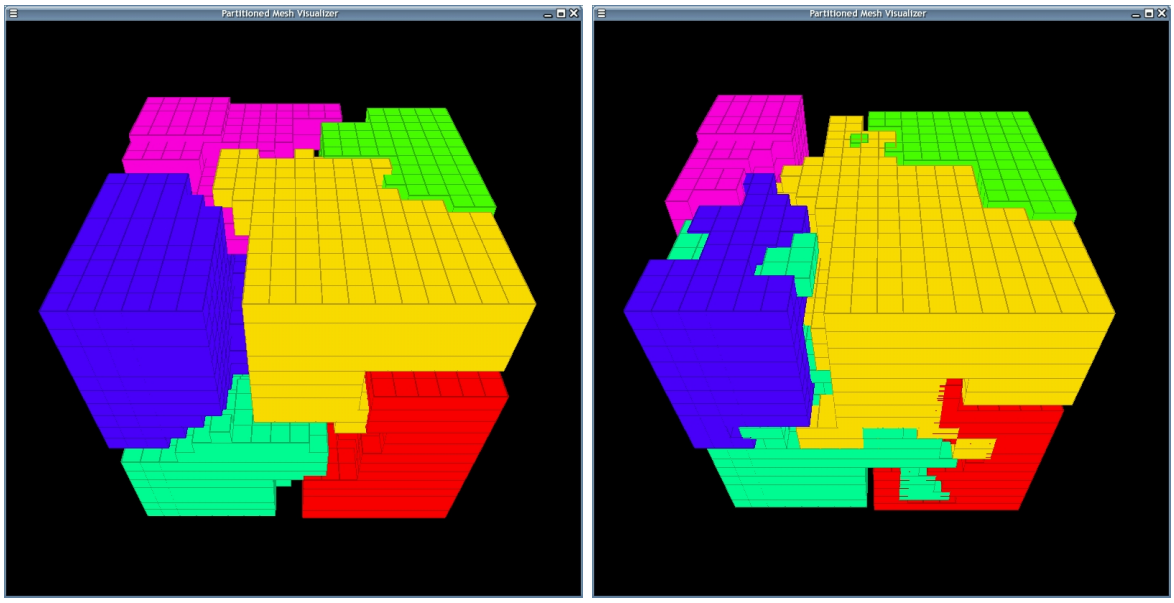


Figure 4.28: 4,938 Elements, 17,604 DOFs Model, 8 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

4.5.3 Soil-Foundation Model with 9,297 Elements, 32,091 DOFs

The mesh is shown in Figure 4.29. Speed up results are shown from Figure 4.30 to Figure 4.32. Partition and repartition figures are shown from Figure 4.33 to Figure 4.37.

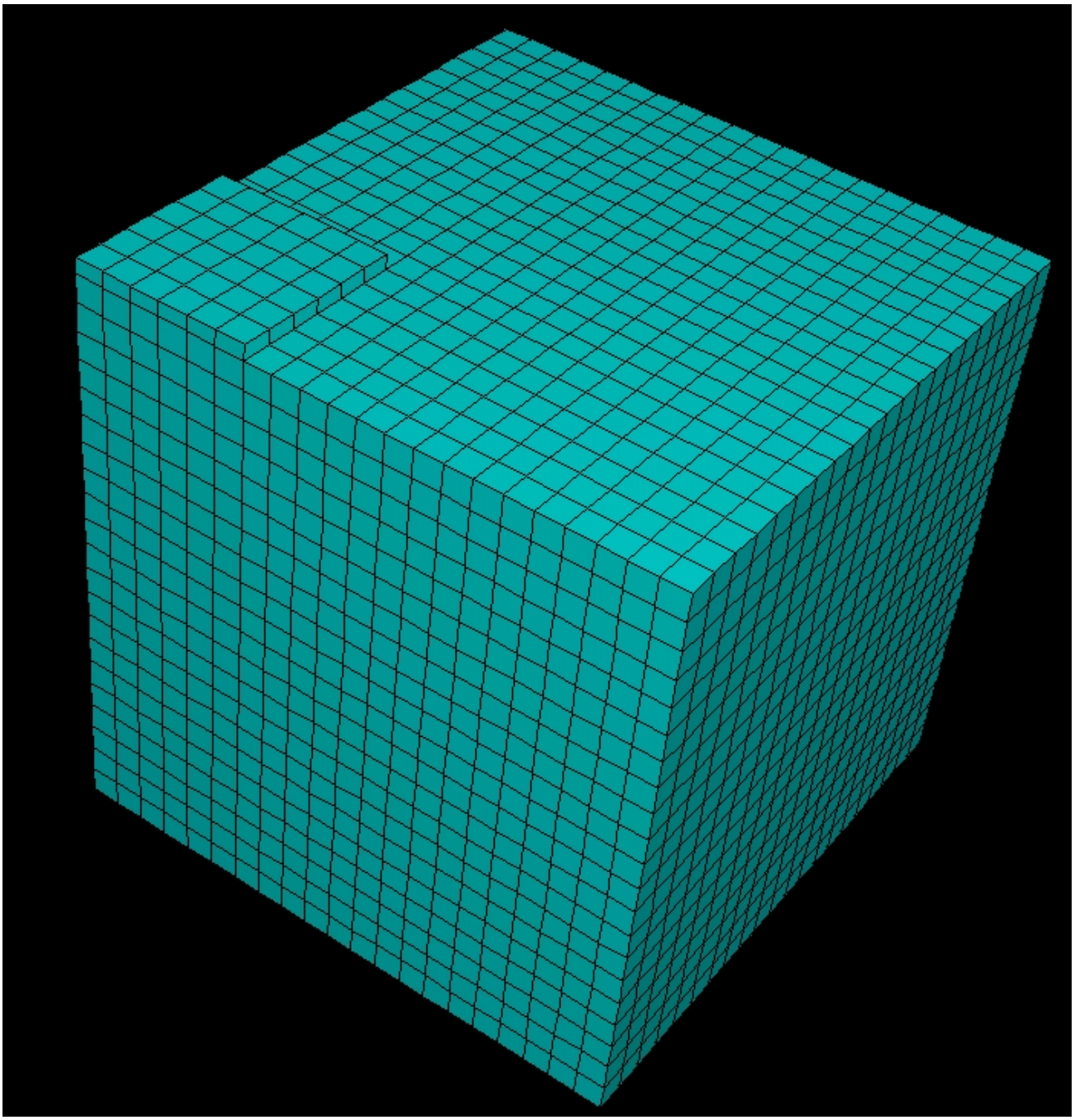


Figure 4.29: Finite Element Model of Soil-Foundation Interaction (9,297 Elements, 32,091 DOFs)

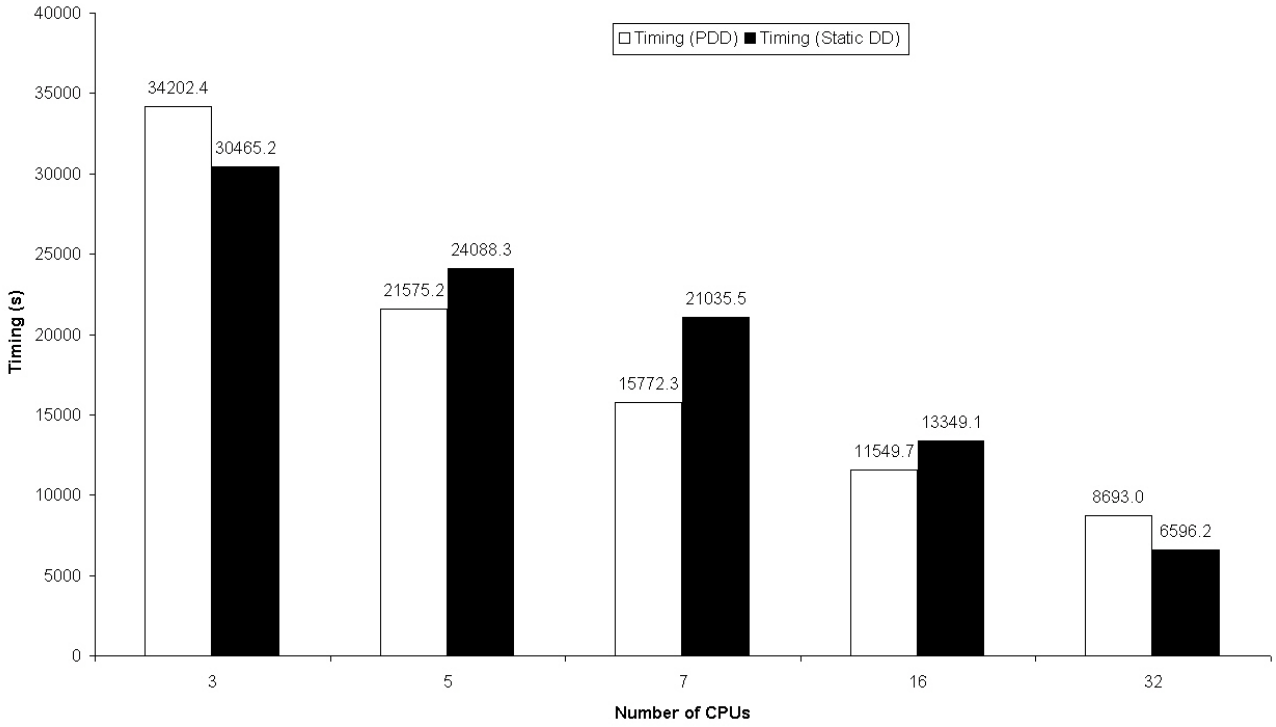


Figure 4.30: Timing Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

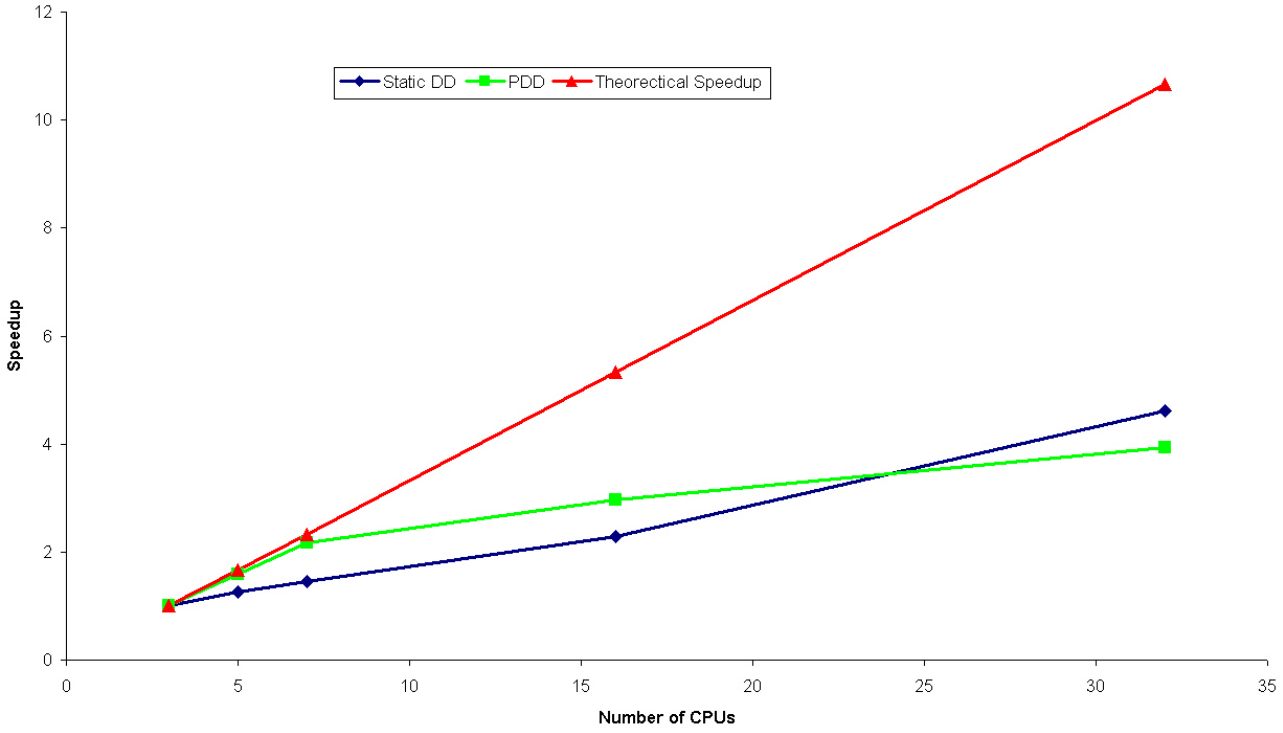


Figure 4.31: Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

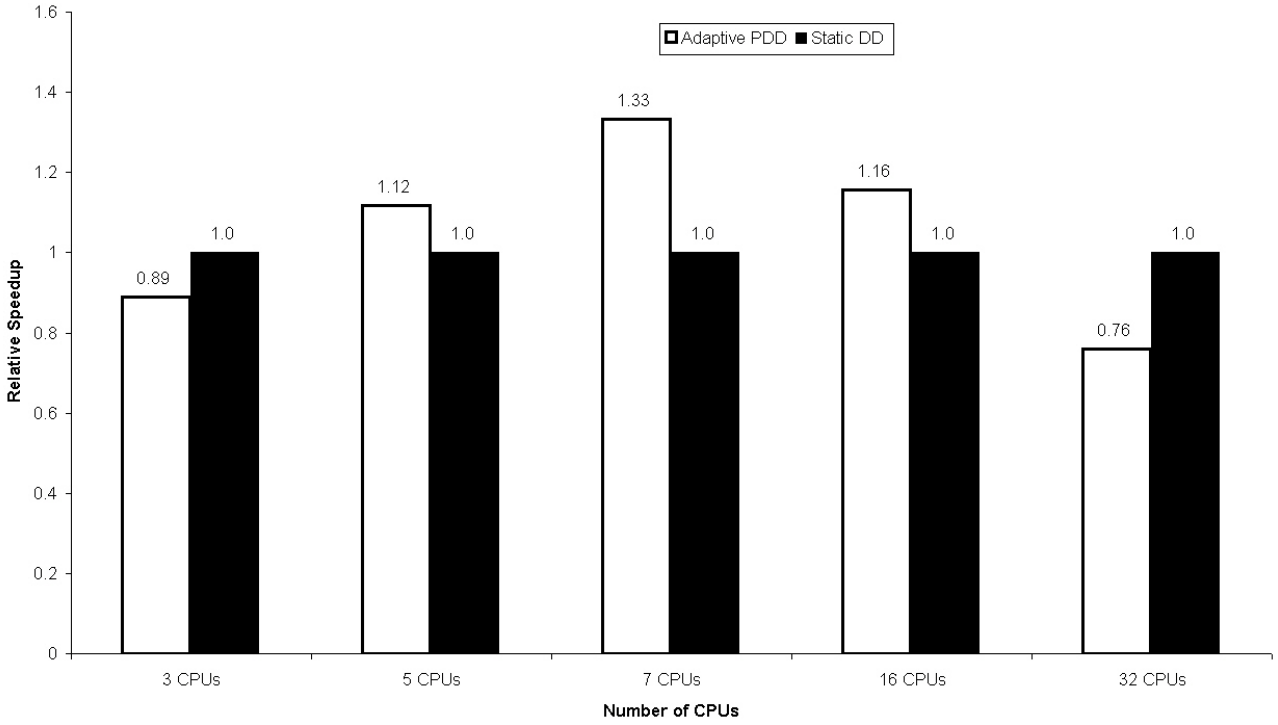


Figure 4.32: Relative Speedup of PDD over Static DD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

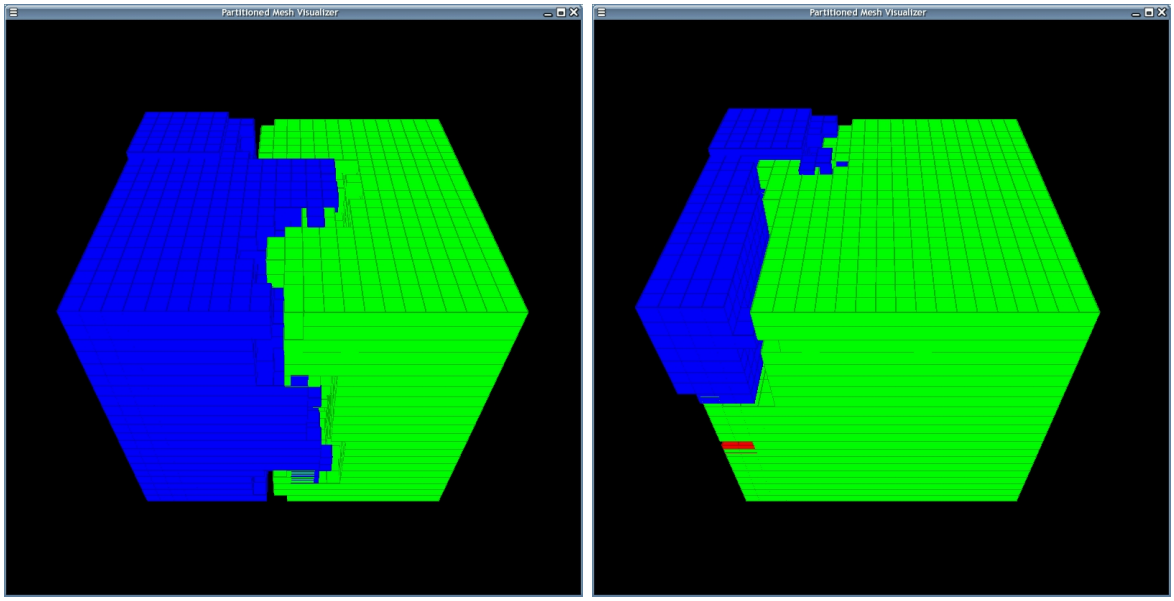


Figure 4.33: 9,297 Elements, 32,091 DOFs Model, 3 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

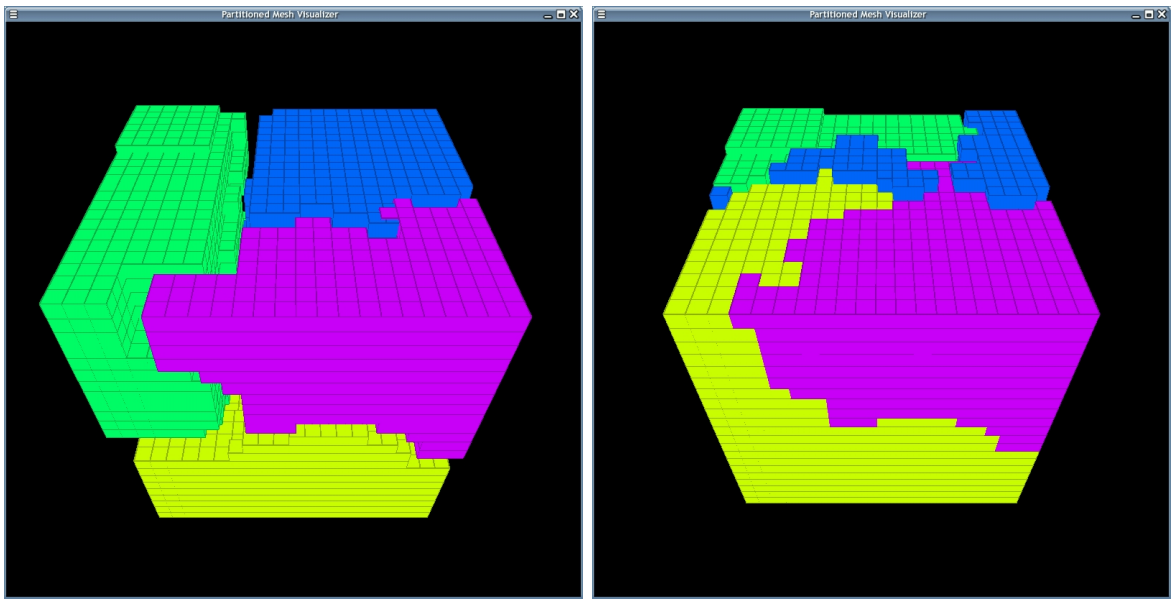


Figure 4.34: 9,297 Elements, 32,091 DOFs Model, 5 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

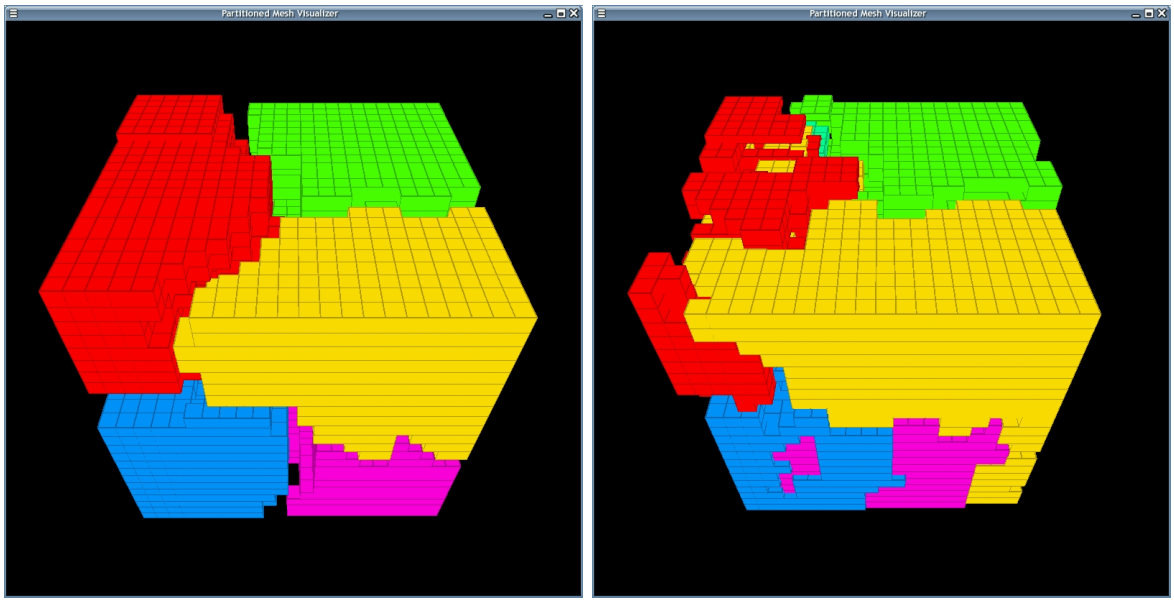


Figure 4.35: 9,297 Elements, 32,091 DOFs Model, 7 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

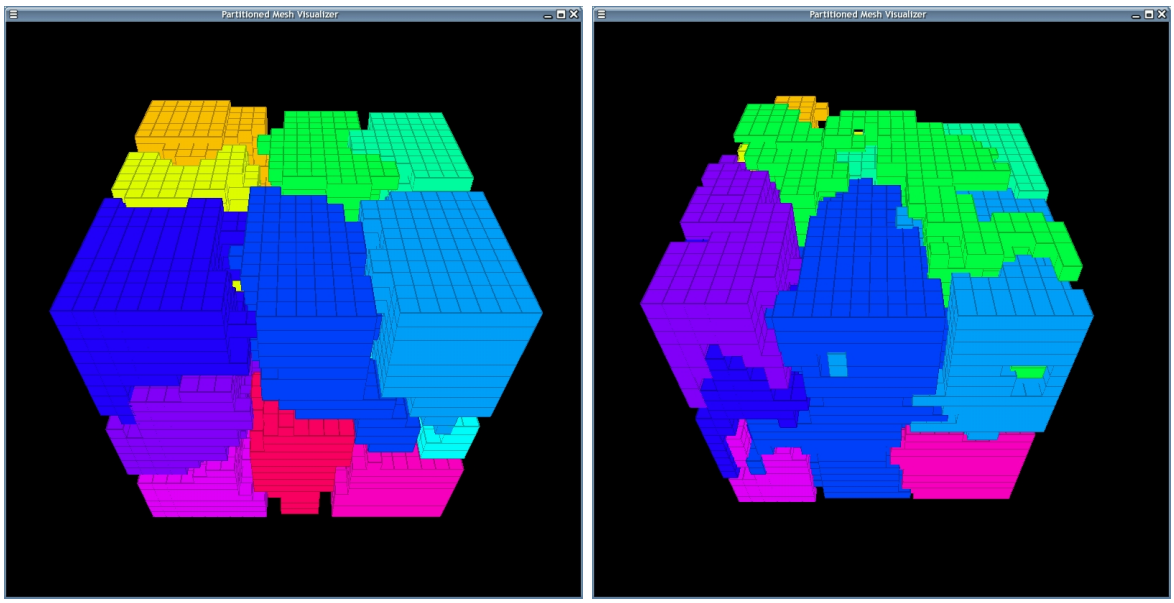


Figure 4.36: 9,297 Elements, 32,091 DOFs Model, 16 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

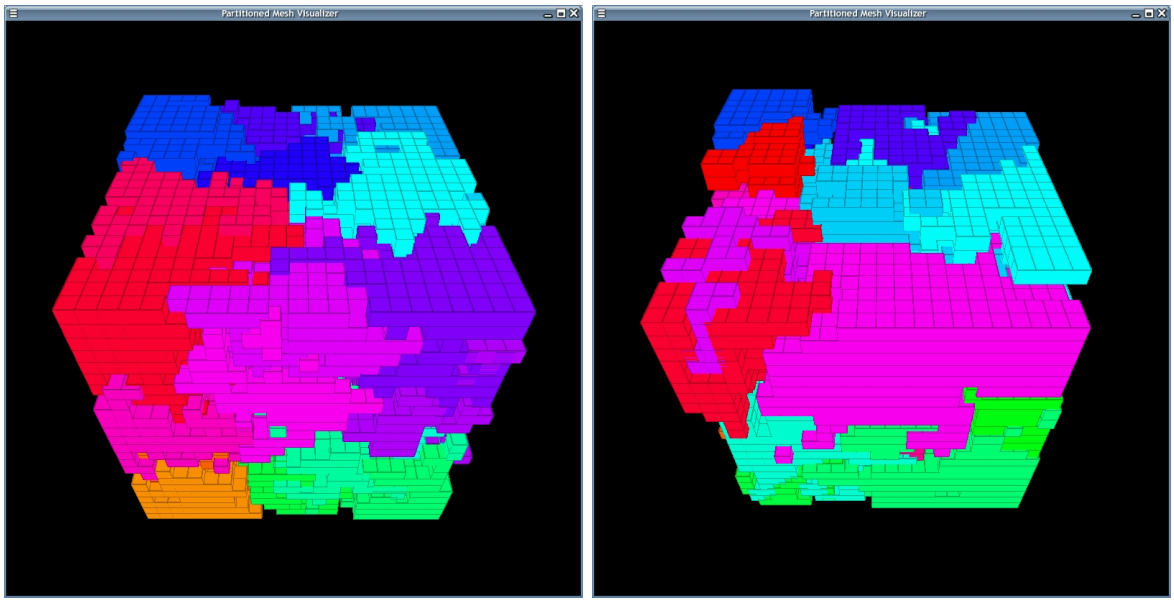


Figure 4.37: 9,297 Elements, 32,091 DOFs Model, 32 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

4.6 Algorithm Fine-Tuning

From performance analysis results in previous sections, it has been shown that adaptive graph partitioning algorithm based on element graph can improve overall load balance for nonlinear elastic-plastic finite element calculations. Speed up has been observed on example problems. While on the other hand, we can also see as the model size increases, the efficiency of proposed PDD algorithm dropped sharply as shown in Figures 4.31 and 4.32.

So the naive implementation of PDD does not work as expected. With load balancing, one expects that the performance of PDD should not be worse than the DD case. It otherwise implies that the PDD does not bring performance gain that can completely offset its own extra load balancing operations-related overheads.

In this report , more detailed algorithm fine-tuning has been performed to address the problems we had in previous sections of the naive PDD implementation.

In order to improve the overall efficiency of proposed PDD algorithm. we have to consider two levels of costs when one wishes to balance the computational load among processing units. One is the data communication cost, and the other one is finite element model regeneration overhead associated with specific application problems.

Currently the adaptive graph partitioning algorithm does not consider the fact that the network communication patterns might differ much among processing nodes. The single ITR value indicates the algorithmic approach of the graph partitioning algorithm, but the real communication performance has not been addressed in the implementation.

On the other hand, certain applications impose extra problem-dependent overhead to repartitioning operations. For example, whenever data communications happen, the finite element model has to be wiped off and regenerated. This is not inherent with the graph partitioning algorithm but still needs to be addressed in order to get the best performance. As observed in this report , model regeneration overhead increases when the finite element model becomes bigger.

In order to improve the overall performance of our application, we hope to consider both data communication and model regeneration cost and create a new strategy through which we can adaptively monitor the extra overheads to assure that load balancing operation can offset both costs.

This chapter will first investigate the effect of load balance tolerance on performance and then a new globally adaptive strategy will be proposed to handle both communication and model regeneration overhead. Speedup analysis have been done to show performance gains.

4.7 Fine Tuning on Load Imbalance Tolerance

If one finds out that the application-associated overhead (say, model regeneration cost) overwhelms when repartitioning happens, the most natural way to improve performance is to increase the load imbalance

tolerance of the adaptive repartition routine. In this way, one hopes to increase the critical load imbalance that can trigger the balancing routine and so that the repartition counts can be reduced. As a result, model regeneration cost can do less harm to the overall performance.

This should rather viewed as a work-around and has not been effective in our application.

The tuning approach aims at improving efficiency of previous runs that failed showing speedup over static domain decomposition method. Shallow foundation model with 9,297 Elements, 32,091 DOFs has been chosen to study the effect of imbalance tolerance on parallel performance. Model setup has been the same as in previous sections.

Speedup analysis results have been shown in Figures 4.38, 4.39 and 4.40.

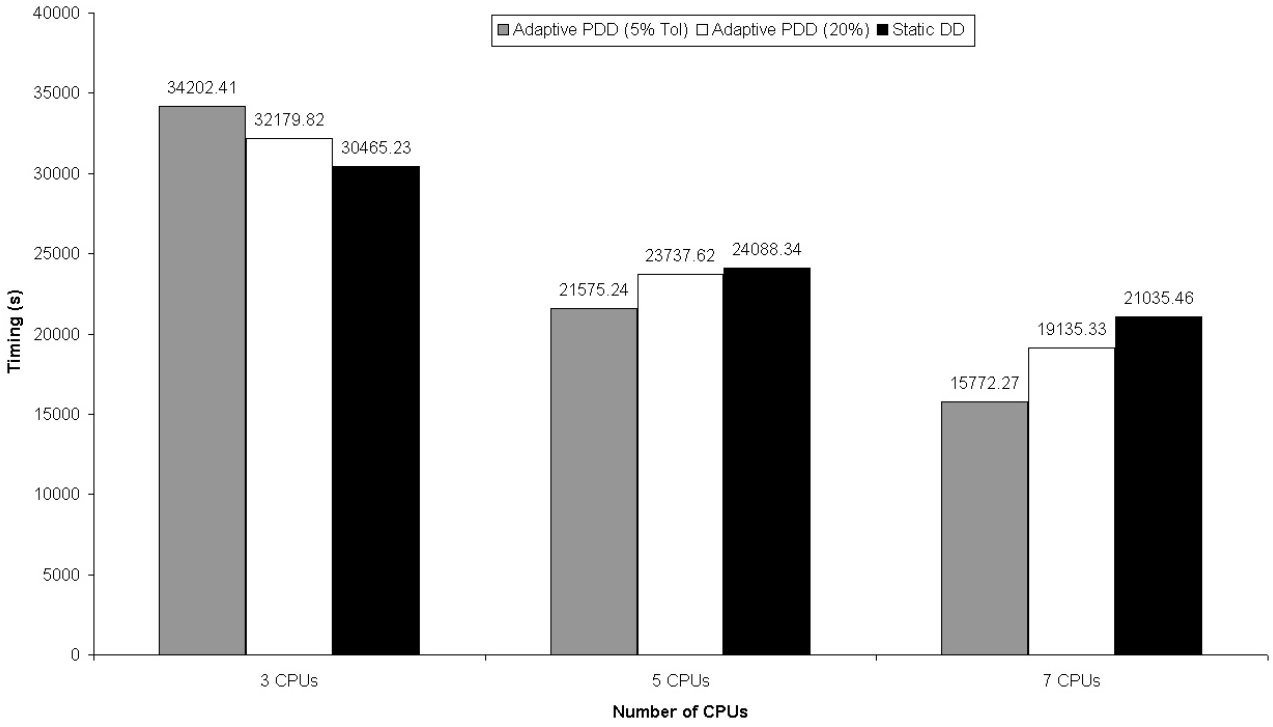


Figure 4.38: Timing Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20%

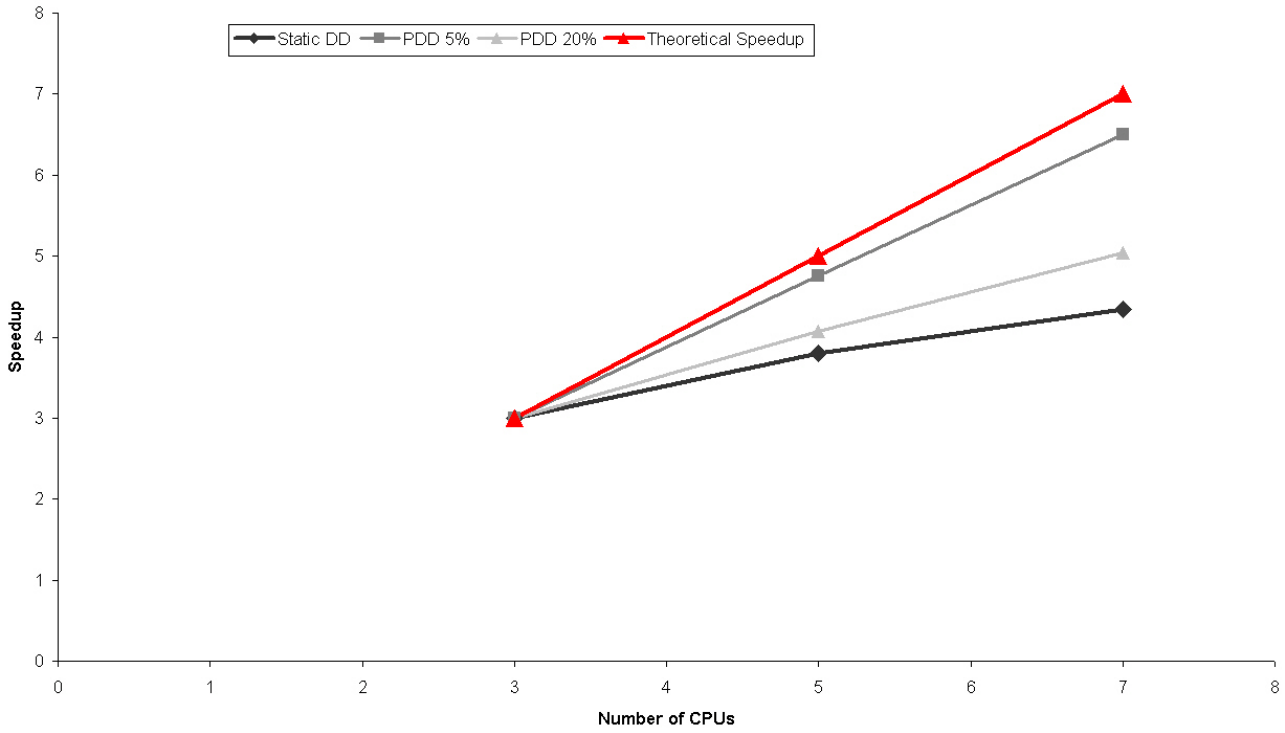


Figure 4.39: Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20%

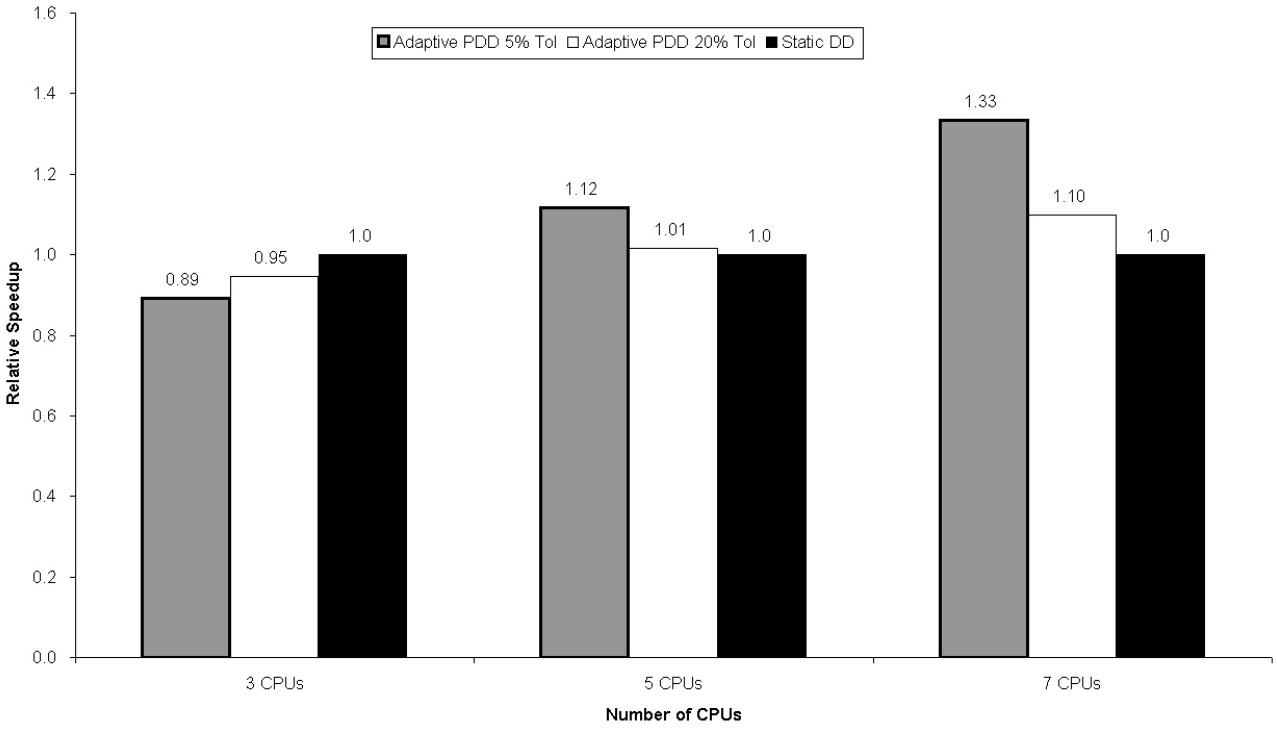


Figure 4.40: Relative Speedup of PDD over Static DD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20%

From the performance results, we can see that increasing load imbalance tolerance does not lead to efficiency for our application. As the number of processing units increases, the whole performance of application codes deteriorates. It is also important to note that the adaptive graph partitioning/repartitioning kernel in ParMETIS has not been capable of producing adequate partitions for finite element calculations when the load imbalance tolerance is larger than the recommended 5% Karypis et al. (2003). The application crushed with 20% imbalance tolerance for same models tested in previous sections.

The conclusion reached for the application in this report is that load imbalance tolerance larger than 5% has not been proved more efficient. This can also be explained in more details.

In the implementation of ParMETIS, load imbalance tolerance is one of the most important parameters in the sense that this value determines whether repartition will be switched on. The other equally significant implication of this value comes from the fact that it also establishes target load imbalance residual to be achieved after adaptive load balancing. That means for each repartition, the ParMETIS will only reduce the load imbalance to the provided tolerance.

In current implementation, the load imbalance tolerance is set to be the same for both switch-on and target values, which is not capable of bringing the best performance into our application due to the fact that aside from data redistribution cost, analysis model reconstruction is equally expensive. The dilemma is described by numerical example as shown in Table 4.5.

Table 4.5: Observation on Load Imbalance Tolerance %5

Model	20,476 Elements, 68,451 DOFs
CPUs	32
Imbalance Before	7.018%
Imbalance After	4.9%
Model Regeneration	57.2934 seconds
Total Step Time	140.961 seconds

We can easily see that tiny portion of data movement to balance out $7.018 - 4.9 = 2.228\%$ loads still invoked analysis model regeneration, which accounts for extra overhead that is about 40.6% of total step time.

Because the load balance tolerance is also the target value that the repartitioning operation hopes to achieve. The implication is that after repartitioning, the load distribution among processing units is barely under this acceptable tolerance. The performance study conducted so far showed that continuous plastification can easily creates load imbalance over this tolerance so another round of repartitioning would be launched again. It greatly brings down the performance of the whole application when the huge data redistribution overhead is taken just to overcome a tiny imbalance. This explains why changing the tolerance was not able to bring better performance in our application.

In order to improve performance while still minimizing load imbalance, we hope to maximize the efficiency of model regeneration routine in our application. This is a two-fold statement, firstly, we don't want to blindly increase the load imbalance because it basically claims we fail our adaptive PDD algorithm by not switching on repartitioning (5% is suggested by the author of ParMETIS Karypis et al. (2003) and has been proved to be the most stable value in this report), secondly, with each repartitioning, we hope to achieve "perfect balance" as much as possible and in this way, the huge model regeneration cost can be offset by performance gain. What was proposed as future extension of this report is the idea of *dual load imbalance tolerances*. Load balancing triggering tolerance and the target tolerance can be defined separately. We can set higher triggering tolerance to reduce the number of repartition counts, while on the other hand a strict target tolerance can be set close to 1.0 to get better load distribution out of the balancing routine. With proposed approach, our application in this report will be able to fully take advantage of the repartition routines without sacrificing too much on model regenerations.

4.8 Globally Adaptive PDD Algorithm

One significant drawback of current implementation is that neither network communication nor model regeneration cost has been considered in element-graph-based type domain decomposition algorithm. Element graph only records computational load carried by each element. Only one ITR factor characterizes algorithmic approach of the load balancing operation and this is apparently too crude for complicated network/hardware configurations. The ignorance of the repartitioning-associated overheads inherent with application codes can lead to serious performance drop of the proposed PDD algorithm as shown in Figure 4.41.

This drawback can harm the overall performance of the whole application code more seriously when the simulation is to be run on heterogeneous networks, which means we can have different network connections and nodes with varied computational power. The dilemma is, without exact monitoring of network communication and local model regeneration costs, we can easily sacrifice the performance gain by load balancing operations.

A second approach proposed in this report was the idea of modified *Globally Adaptive PDD* algorithm. The novelty comes from the fact that both data redistribution and analysis model regeneration costs will be monitored during execution. Load balancing will be triggered only when the performance gain necessarily offset the extra cost associated with the whole program. Domain graph structures will be kept intact till successful repartitioning happens. Meanwhile all elemental calculations will be timed to provide graph vertex weights. Data will be accumulated till algorithm restart happens, when all analysis model and vertex weights will be nullified.

This improvement aims at handling network communication and any specific application-associated overheads automatically at the global level in order to remedy the drawback that the element graph repartitioning kernel currently supported by ParMETIS is not capable of directly reflecting this application

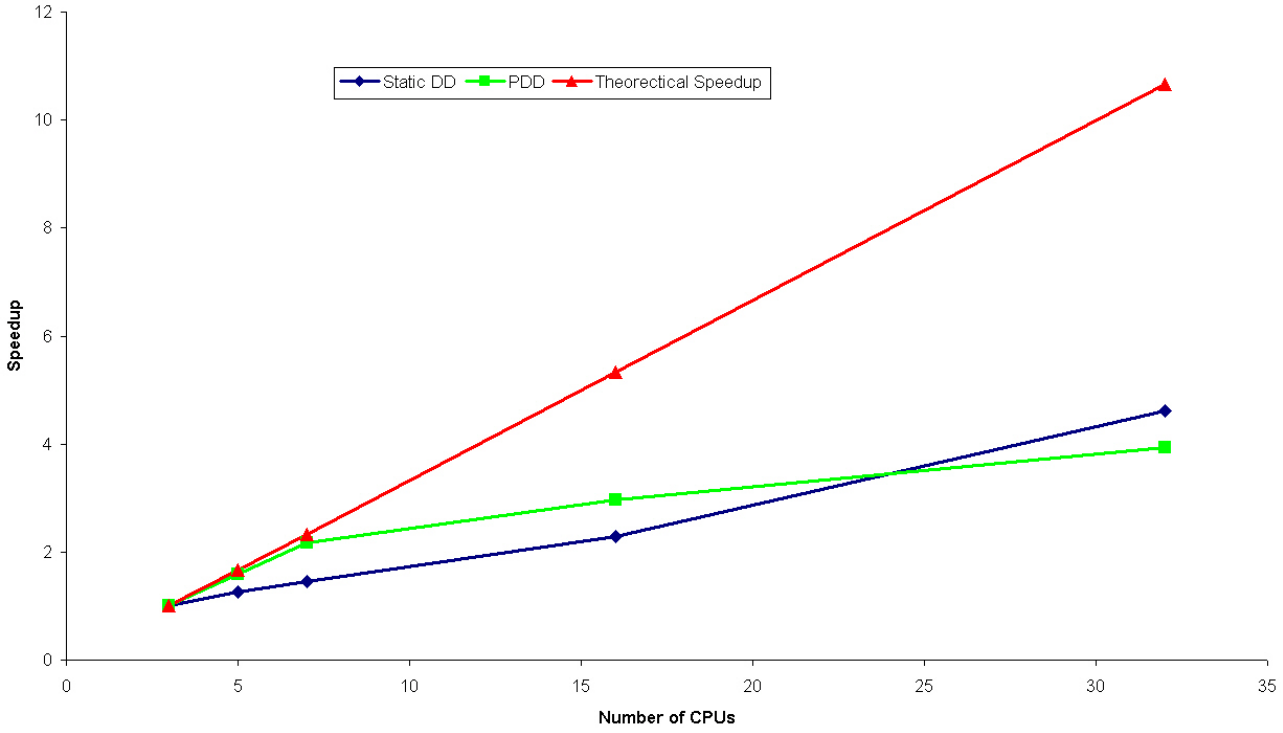


Figure 4.41: Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

level overheads. The new strategy is to automatically monitor network communication and local model regeneration timings which will be integrated to the entry of load balancing routines to act as additional triggers of the operation along with the load imbalance tolerance.

Performance study shows that PDD algorithm with the new additions significantly improve performance even when the number of processing units is large. This modification fixes the drawback shown in previous sections that the performance of PDD was beaten by static domain decomposition when the number of processors increases.

This strategy is called to be *globally adaptive* because both data communication and model regeneration costs are monitored at the application level, which tells best how the real application performs on all kinds of networks. Whatever the network/hardware configurations might be, real application runs always deliver the most accurate performance counters. This information can be applied on top of graph partitioning algorithm as a supplement to account for the drawback that the algorithm kernel is not capable of integrating global data communication costs.

4.8.1 Implementations

We can define the global overhead associated with load balancing operation as two parts, data communication cost T_{comm} and finite element model regeneration cost T_{regen},

$$T_{overhead} := T_{comm} + T_{regen} \tag{4.1}$$

Performance counters have been setup to study both.

- T_{comm}
  Data communication patterns characterizing the network configuration can be readily measured as the program runs the initial partitioning. As described in previous sections, initial domain decomposition needs to be done to send elements over to processing nodes. This step is necessary for parallel finite element processing and it provides perfect initial estimate how the communication pattern of the application performs on specific networks. Timing routines have been added to automatically measure the communication cost. This cost is inherently changing as the network condition might vary as simulation progresses, so whenever data redistribution happens, this metric will be automatically updated to reflect the network conditions.

- T_{regen}
  Model regeneration cost basically comes from the fact that if data redistribution happens, the analysis model needs to be regenerated to reflect changes of nodes and elements inside the domain. Detailed operations include renumbering DOFs and rehandling constraints. This part of cost is application-dependent. In current implementation of PDD, efforts have been made to set up timing stop at the entry and exit of model regeneration routines to get the accurate data for the extra overhead. It is also

important to note that model regeneration happens when the initial data distribution finishes, again the initial domain decomposition phase provides perfect initial estimate of the model regeneration cost on any specific hardware configurations.

Naturally, for the load balancing operations to pay off, the $T_{overhead}$ has to be offset by the performance gain T_{gain}. This report also creates a strategy to estimate the performance gain T_{gain} even before the load balancing operation happens and this metric provides global control on top of the existing graph repartitioning algorithm.

As implemented in previous sections, the computational load on each element is represented by the associated vertex weight $vwgt[i]$. If the SUM operation is applied on every single processing node, the exact computational distribution among processors can be obtained as total wall clock time for each CPU as shown in Equation 4.2,

$$T_j := \sum_{i=1}^{n} vwgt[i],\ j = 1, 2, \ldots, np \tag{4.2}$$

in which n is the number of elements on each processing domain and np is the number of CPUs.

If we define,

$$T_{sum} := sum(T_j),\ T_{max} := max(T_j), \text{ and } T_{min} := min(T_j),\ j = 1, 2, \ldots, np \tag{4.3}$$

one always hope to minimize T_{max} because in parallel processing, T_{max} controls the total wall clock time. By load balancing operations, we mean to deliver evenly distributed computational loads among processors. So theoretically, the best execution time is,

$$T_{best} := T_{sum}/np, \text{ and } T_j \equiv T_{best},\ j = 1, 2, \ldots, np \tag{4.4}$$

if the perfect load balance is to be achieved.

Based on definitions above, the best performance gain T_{gain} one can obtain from load balancing operations can be calculated as,

$$T_{gain} := T_{max} - T_{best} \tag{4.5}$$

Finally, the load balancing operation will be beneficial **IF AND ONLY IF**

$$T_{gain} \geq T_{overhead} = T_{comm} + T_{regen} \tag{4.6}$$

4.8.2 Performance Results

The newly improved design has been compared to the old design to see the effectiveness of the globally adaptive switch of PDD algorithm.

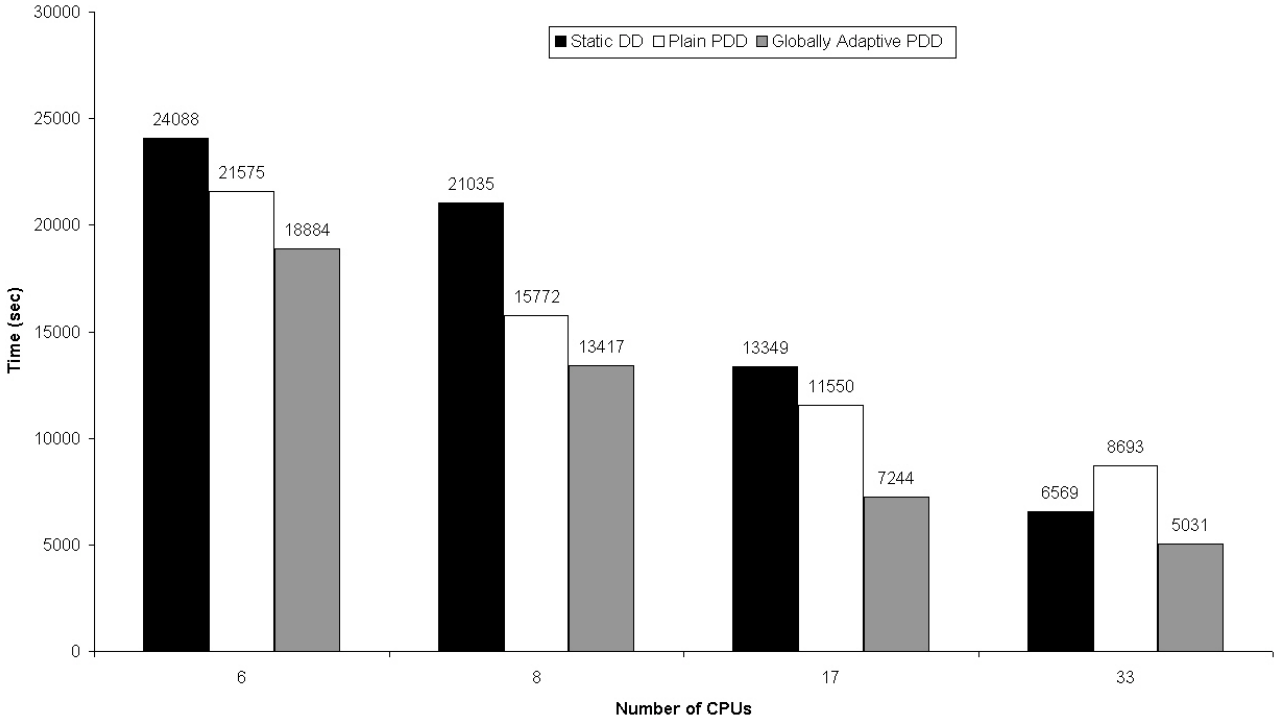


Figure 4.42: Performance of Globally Adaptive PDD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

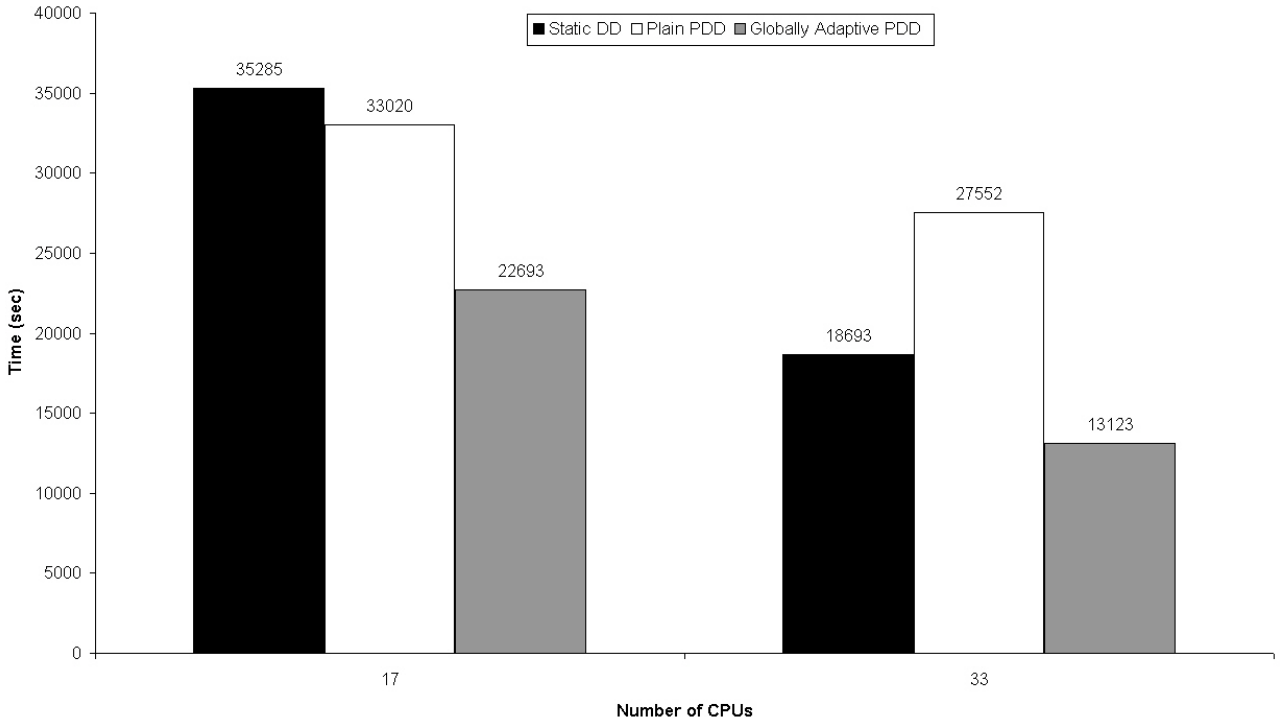


Figure 4.43: Performance of Globally Adaptive PDD on 20,476 Elements, 68,451 DOFs Model, ITR=1e-3, Imbal Tol 5%

From Figures 4.42 and 4.43, advantage of the improved globally adaptive PDD algorithm have clearly been shown. After considering the effect of both data communication and model regeneration costs, the adaptive PPD algorithm necessarily outperforms the static Domain Decomposition approach as expected. This new design also significantly improves the overall scalability of the proposed PDD algorithm as shown in Figure 4.44 and 4.45.

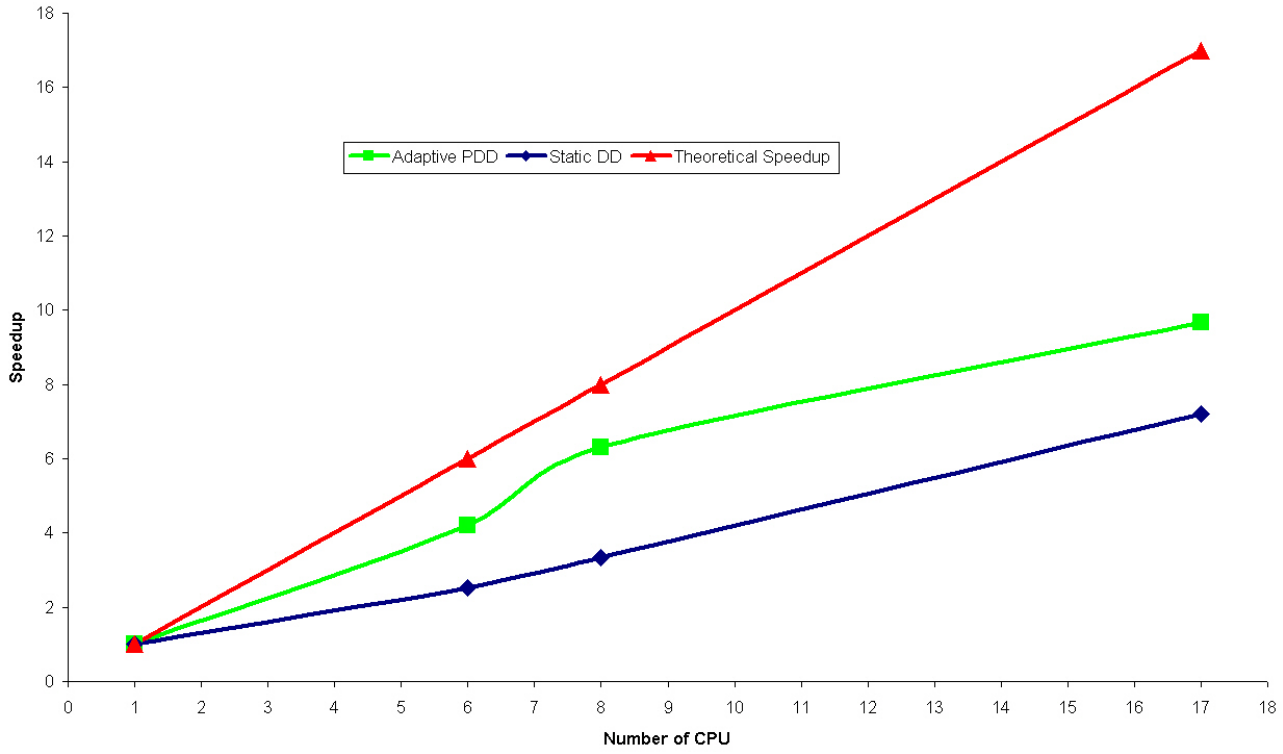


Figure 4.44: Scalability Study on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

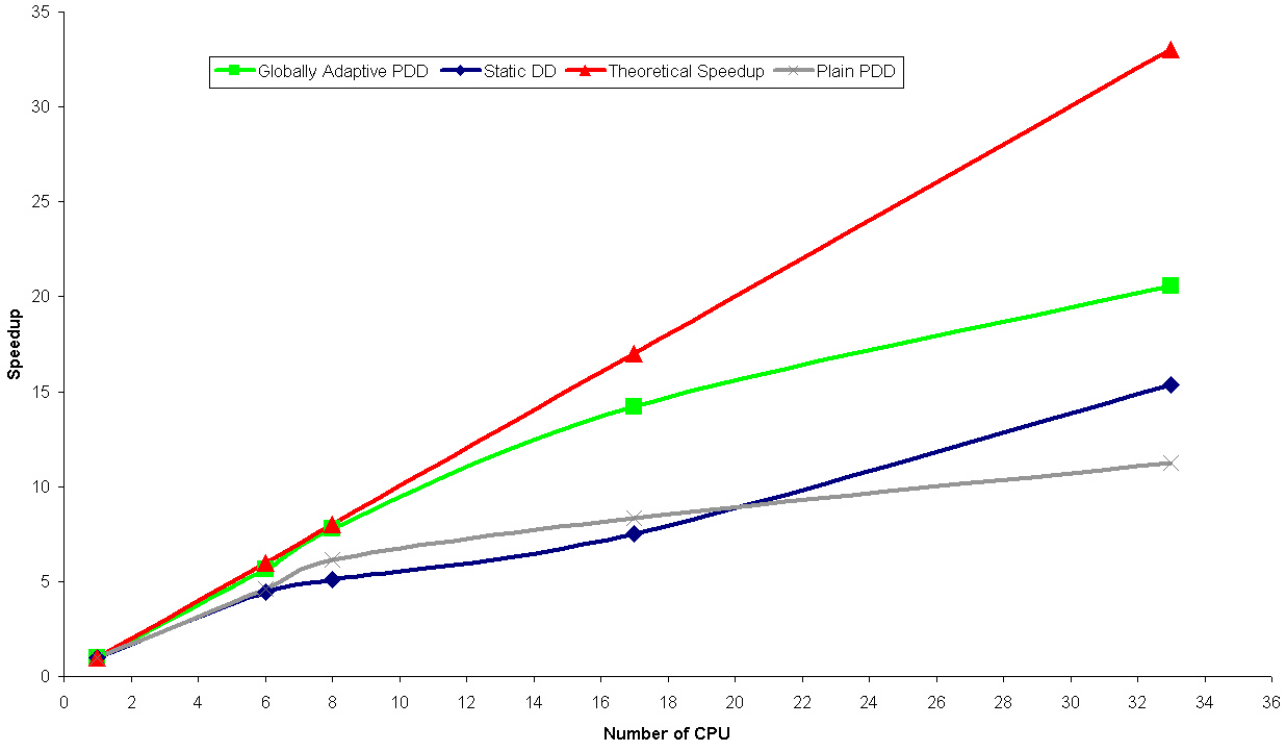


Figure 4.45: Scalability Study on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

4.9 Scalability Study on Prototype Model

The ultimate purpose of this report is to develop an efficient parallel simulation tool for large scale earthquake analysis on prototype SFSI system. After in-depth development-refining process conducted in previous sections, real 3-bent production models have been set up to study the parallel performance of the proposed PDD algorithm using real world earthquake ground motions.

4.9.1 3 Bent SFSI Finite Element Models

As described in later sections, various sizes of a 3 bent bridge SFSI system has been developed to study dynamic behaviors of the whole system in different frequency domain. These models provide perfect test cases for parallel scalability study of our proposed PDD algorithm.

Detailed model description will be presented in later chapters of this report and only model size and mesh pictures are shown here to indicate the range of model sizes we have covered.

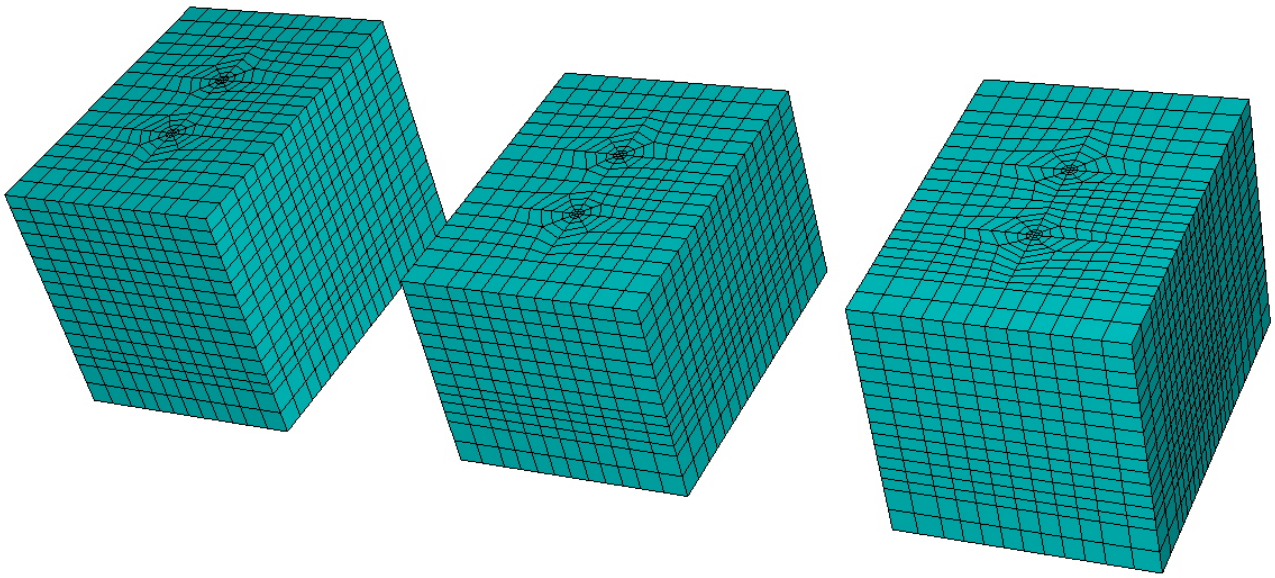


Figure 4.46: Finite Element Model - 3 Bent SFSI, 56,481 DOFs, 13,220 Elements, Frequency Cutoff $>$ 3Hz, Element Size 0.9m, Minimum G/G_{max} 0.08, Maximum Shear Strain γ 1%

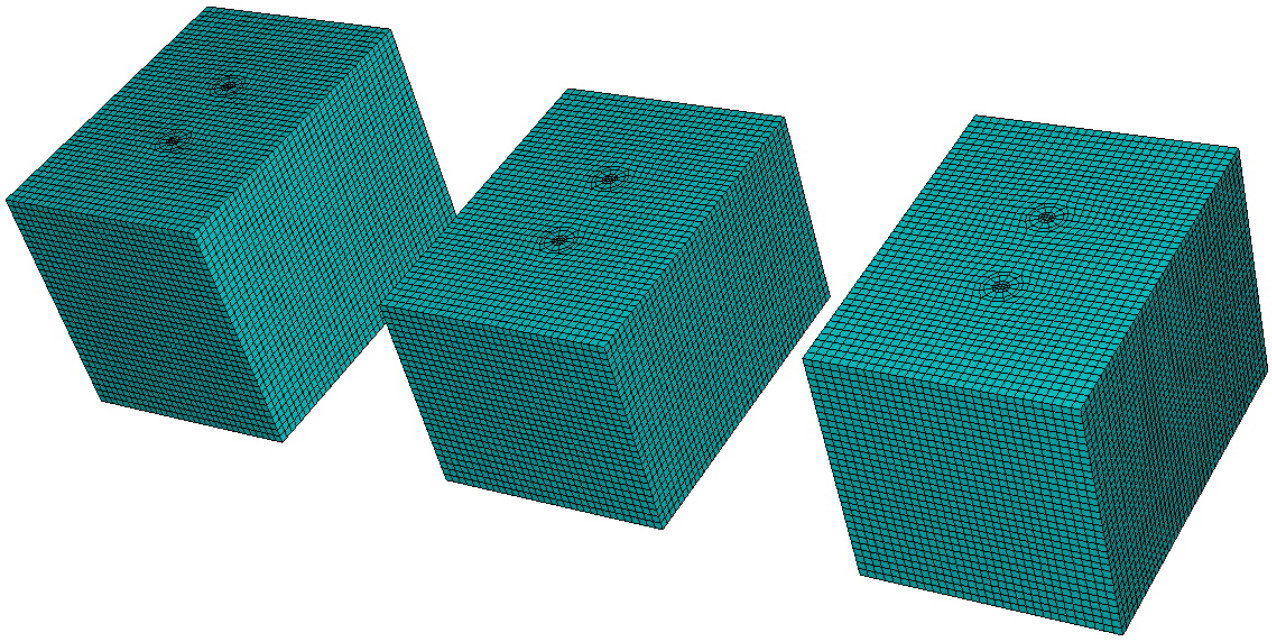


Figure 4.47: Finite Element Model - 3 Bent SFSI, 484,104 DOFs, 151,264 Elements, Frequency Cutoff 10Hz, Element Size 0.3m, Minimum G/G_{max} 0.08, Maximum Shear Strain γ 1%

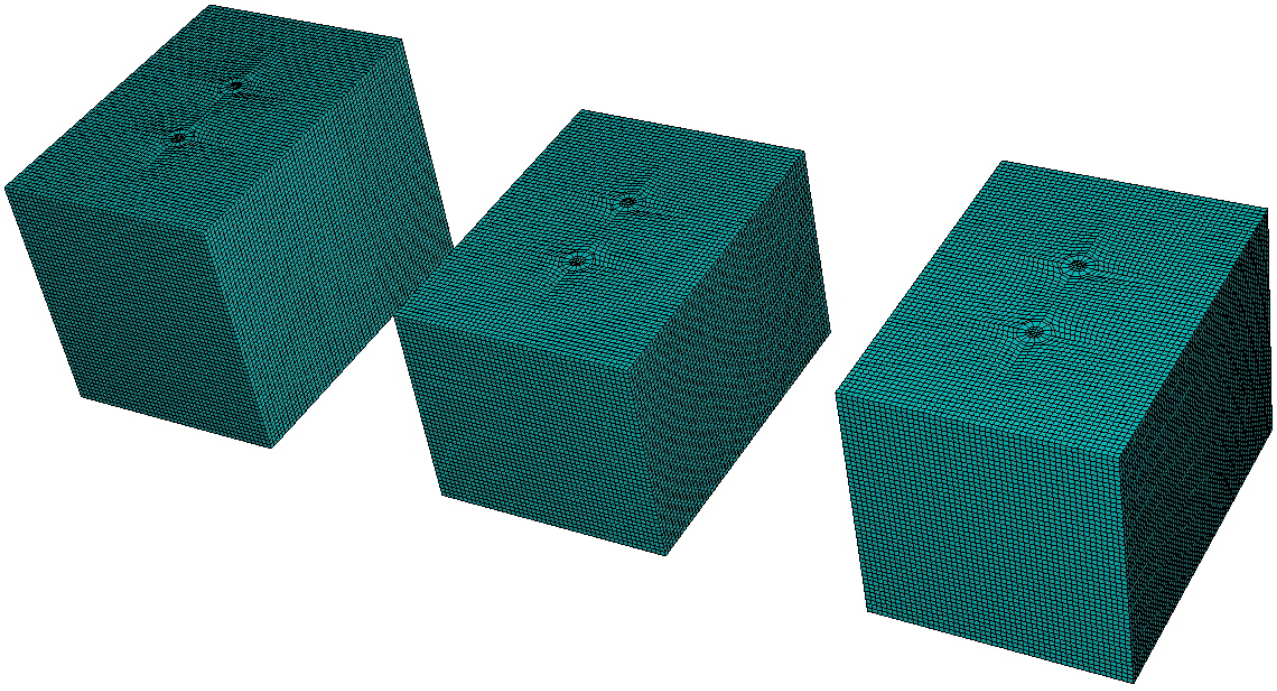


Figure 4.48: Finite Element Model - 3 Bent SFSI, 1,655,559 DOFs, 528,799 Elements, Frequency Cutoff 10Hz, Element Size 0.15m, Minimum G/G_{max} 0.02, Maximum Shear Strain γ 5%

4.9.2 Scalability Runs

The models with different detail levels have been subject to 1997 Northridge earthquake respectively for certain time steps and total wall clock time has been recorded to analyze the parallel scalability of our proposed PDD. The result is presented in Figure 4.49.

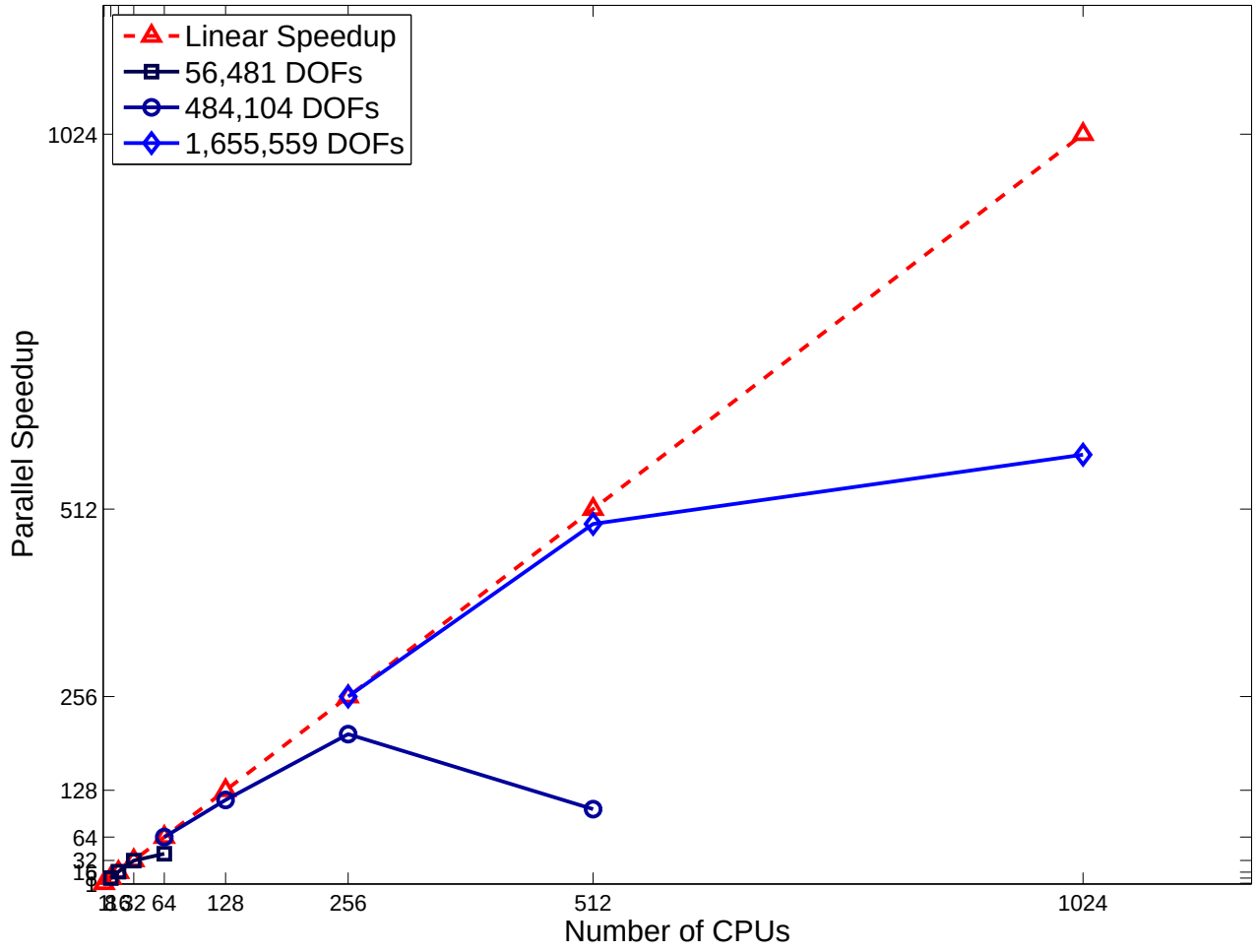


Figure 4.49: Scalability Study on 3 Bent SFSI Models, DRM Earthquake Loading, Transient Analysis, ITR=1e-3, Imbal Tol 5%, Performance Downgrade Due to Increasing Network Overhead

4.10 Conclusions

Through detailed performance studies as presented in previous sections, some conclusions can be drawn and future directions can be noted.

- Plastic Domain Decomposition (PDD) algorithm based on adaptive multilevel graph partitioning kernels has been shown to be effective for elastic-plastic parallel finite element calculations. PDD algorithm consistently outperforms classical Domain Decomposition method for models tested so far in this report as shown in Figures 4.50 and 4.52.

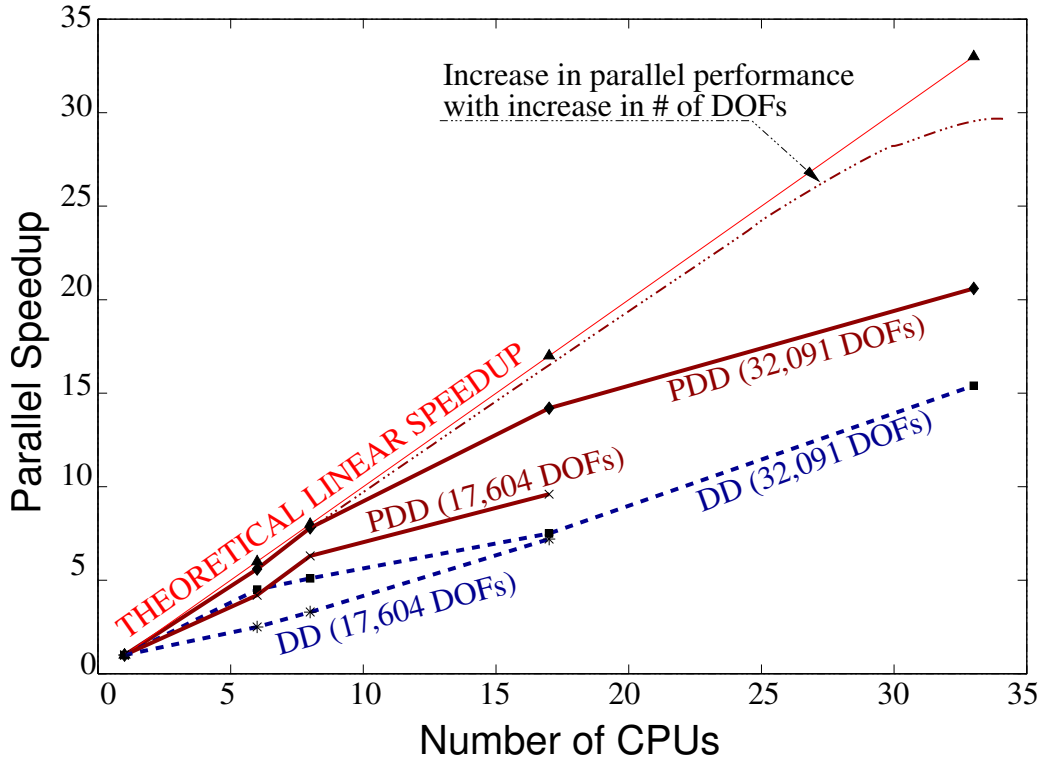


Figure 4.50: Relative Performance of PDD over DD, Shallow Foundation Model, Static Loading, ITR=1e-3, Imbal Tol 5%

- There are some parameters that can be calibrated in the current implementation. As indicated by results of thorough numerical tests, ITR=0.001 and load imbalance tolerance ubvec=1.05 (5%) should be adopted and studies on our application in this report have shown they are adequate and able to bring performance not worse than the commonly used domain decomposition method in parallel finite element analysis.

- For the parameters suggested in the report , we can see a general trend that the efficiency of PDD will drop as the number of processors increases. This can be explained. The implication of increasing processing units is that the subdomain problem size will decrease. It is naturally evident that the repartition load balancing won't be able to recover the overhead by balancing off small size local calculations. The improved design of globally adaptive PDD algorithm has been implemented in this report and both data communication and model regeneration costs associated with graph repartitioning have been integrated into the new globally adaptive strategy. With the new design, it has also been shown that the PDD algorithm consistently outperforms classic one step domain decomposition algorithm and better scalability can be obtained as shown in Figure 4.51. It has been shown that even for large number of processors, the current implementation can always guarantee that the performance of PDD is not worse than static DD method as shown in Figure 4.52. (the repartition routine has less than 5% overhead of the total wall clock time).

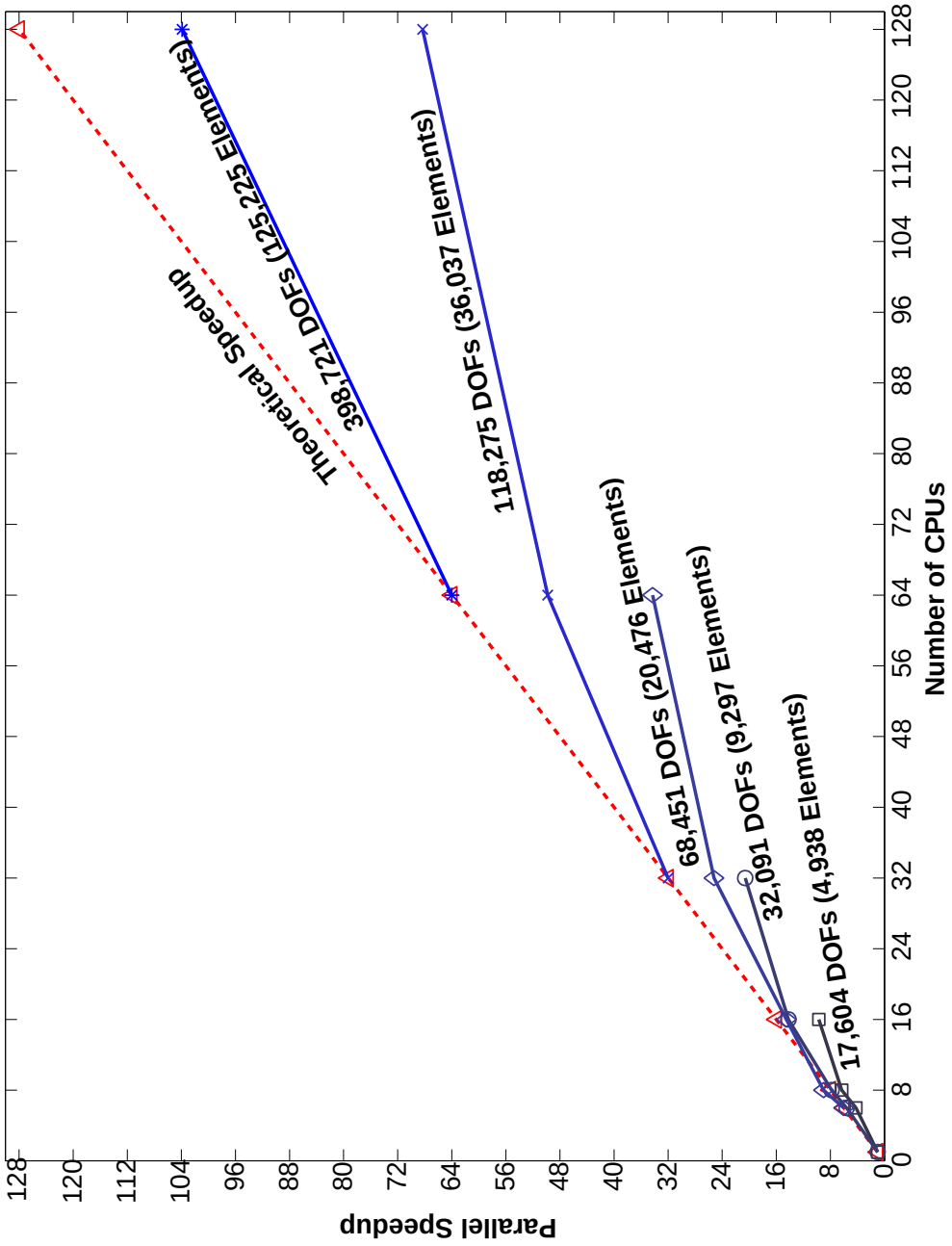


Figure 4.51: Scalability of PDD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5%

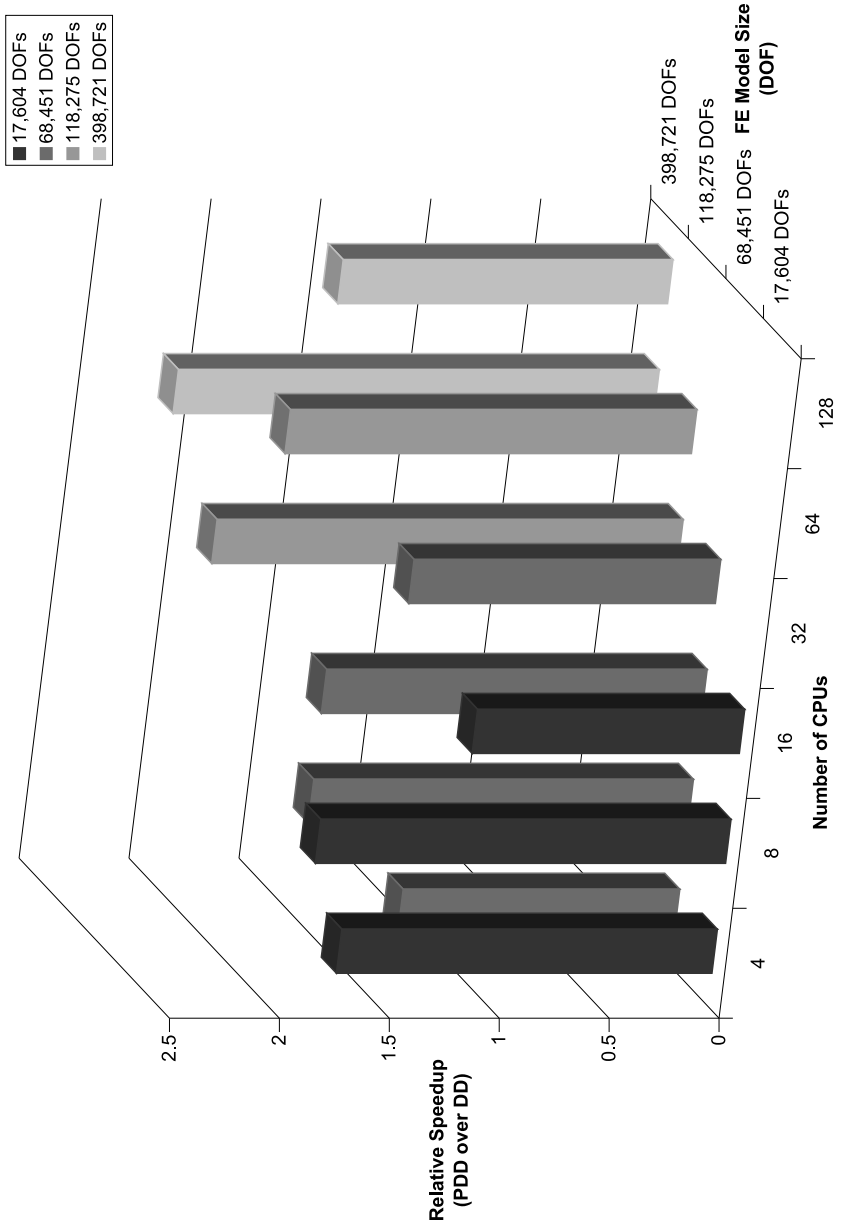


Figure 4.52: Relative Speedup of PDD over DD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5%

- If the problem size is fixed, there exists an optimum number of processors that can bring the best performance of the proposed load balancing algorithm. As the number of processing units increases after this number, the efficiency of proposed algorithm drops, which is understandable because the local load imbalance is so small overall that balancing gain won't offset the extra cost associated with repartitioning. But still the bottom line of proposed adaptive PDD algorithm is that it can run as fast as static one-step domain decomposition approach with less than 5% overhead of repartitioning routine calls. On the other hand, if the number of processing units is fixed, bigger finite element model will exhibit better performance. The conclusion is shown clearly in 3D in Figure 4.52.
- It is also worthwhile to point out that even without comparing with classical DD, PDD itself exhibits deteriorating performance as the number of processing units increases. Here the reproduction of Figure 4.51 is presented with some downside performance noted as shown in Figure 4.53.

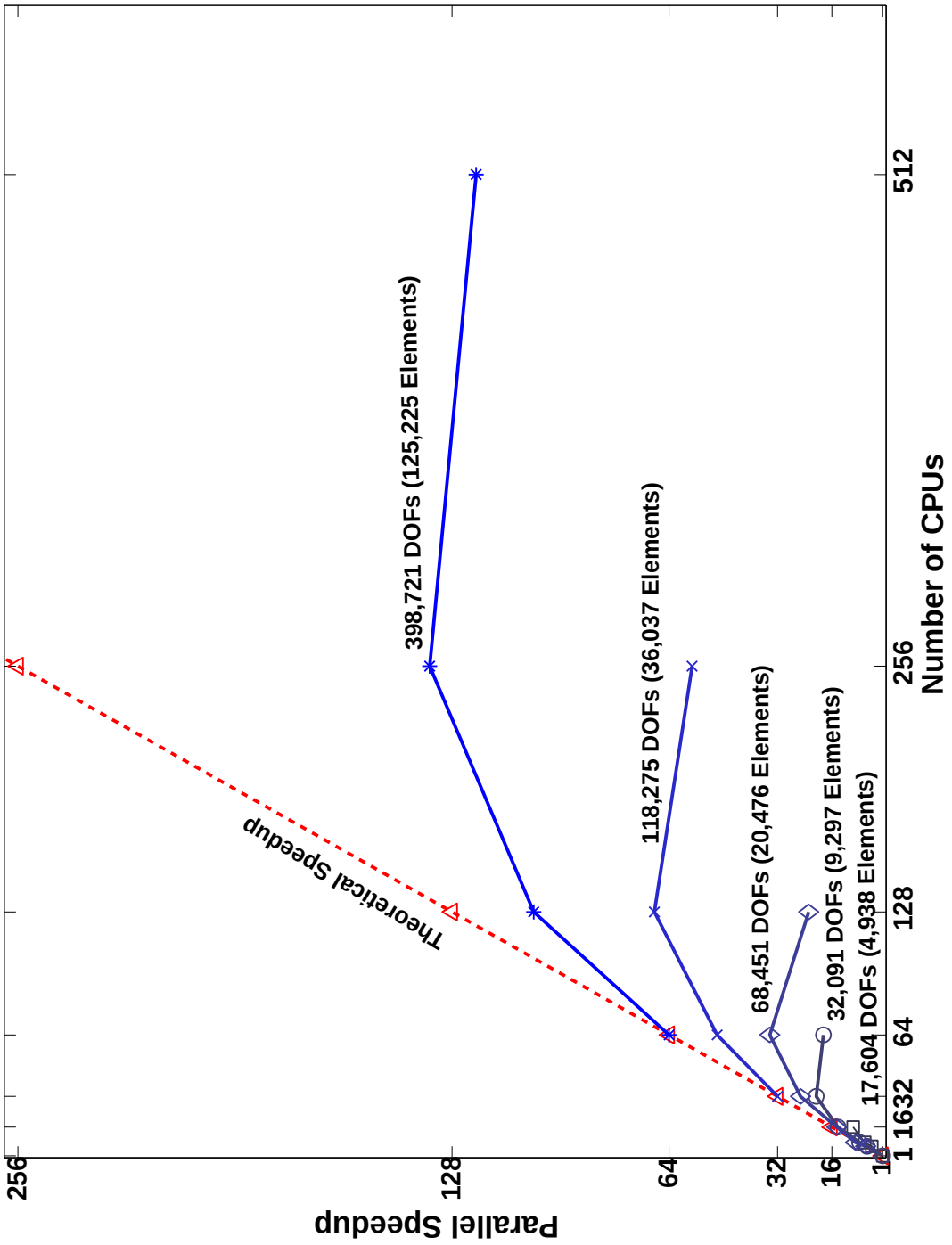


Figure 4.53: Full Range Scalability of PDD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5%, Performance Downgrade Due to Increasing Network Overhead

The implication is explained as follows:

- The performance drop partly is due to the communication overhead gets bigger and bigger so parallel processing will not be able to offset the communication loss.
- It is also noted that as the number of processing units increases, the elemental level calculation drops very scalably with the number of CPUs. This is inherently advantage of the proposed PDD algorithm. PDD through domain decomposition is very scalable for local level calculations because inherently local comp is element-based. when elements are distributed, loads are spread out evenly (during initial and redistribution). So as the number of CPU increases, the equation solving becomes more expensive.

  For the case of 56,481 DOFs prototype model with DRM earthquake loading, it has been observed that for sequential case (1 CPU), elemental computation takes 70% of time. As for parallel case (8 CPUs), we optimized parallel elemental computations through PDD, elemental computation only accounts for about 40%. As the number of CPU increases, parallel case (32 CPUs), the local level computation will only take less than 10% of total wall clock time.

  In other words, as the number of CPUs increases, PDD loses scalability because of the equation solving now dominates. As being discussed in Chapter 5, the parallel direct solver itself is not scalable up to large number of CPUs Demmel et al. (1999a). Parallel iterative solver is much more scalable but difficult to guarantee convergence. This is now also the most important topic in the whole scientific computing community.

For one set of fixed algorithm parameters, such as ITR and load imbalance tolerance, basic conclusion is there exists an *optimal number* of processors that can bring best performance and as finite element model size increases, this number increases as listed in Table 4.6.

Table 4.6: Best Performance Observed for ITR=0.001, Load Imbalance Tolerance %5

# of DOFs	**Speedup**	**# of CPUs**
4,035	1.553	4
17,604	1.992	7
32,091	1.334	7
68,451	1.068	16

The second point is related to the implementation of the multilevel graph partitioning algorithm. In current implementation of ParMETIS used in this report , vertex weight can only be specified as an *int*. That means in order to get timing data from local level calculation for each element, *double* data returned by MPI timing routine has to be converted to *int*. Significant digit loss can happen depending on what accuracy the system clock can carry. We can also adjust the vertex weight by amplifying the timing by

scale factors in order to save effective digits. 10 millisecond has been used in this report to represent the effective timing digits when converting from *double* to *int*.

Part II

Parallel Equation Solving in Finite Element Calculations

Chapter 5

Application of Project-Based Iterative Methods in SFSI Problems

5.1 Introduction

Finite element method has been the most extensively used numerical method in computational mechanics. Equation solver is the numerical kernel of any finite element package. Gauss elimination type direct solver has dominated due to its robustness and predictability in performance.

As modern computer becomes more and more powerful, more advanced and detailed models need to be analyzed by numerical simulation. Direct solver is not the favorite choice for large scale finite element calculations because of high memory requirements and the inherent lack of parallelism of the method itself.

The motivation of this research on iterative solvers results from the fact that there is no robust parallel iterative solvers now available to be used with OpenSees framework. In order to expand the tool for large scale simulation problems such as soil-structure interaction study on prototype bridge systems, it is necessary to introduce a powerful and robust parallel solver into the system.

In this report , the effectiveness of Krylov iterative methods has been tested in solving soil-structure interaction problems. Preconditioning techniques have been introduced. Robustness of iterative solvers has been investigated on equation systems from real soil-structure interaction problems. Several popular parallel algorithms and tools have been collected and implemented on PETSc platform to solve the SFSI problems. Performance study has been carried out using IA64 super computers at San Diego Supercomputing Center. A complete implementation has also been implemented in our computational system, which is based on PDD method, parts of OpenSees framework, ParMETIS, and other material and numerical libraries.

5.2 Projection-Based Iterative Methods

Projection techniques are defined as methods to find approximate solutions $\hat{x}$ for $Ax = b$ ($A \in \mathcal{R}^{n \times n}$) in a subspace $\mathcal{W}$ of dimension m. Then in order to determine $\hat{x}$, we need m independent conditions. One way to obtain these is by requiring the residual $b - A\hat{x}$ is orthogonal to a subspace $\mathcal{V}$ of dimension m, i.e.,

$$\hat{x} \in \mathcal{W}, b - A\hat{x} \perp \mathcal{V} \tag{5.1}$$

The conditions shown in Equation 5.1 are known as Petrov-Galerkin conditions (Bai).

There are two key questions to answer if one wants to use projection techniques in solving large scale linear systems. Different answers lead to many variants of the projection method.

- **Choice of Subspaces**
  Krylov subspaces have been the favorite of most researchers and a large family of methods have been developed based on Krylov subspaces. Typically people choose either $\mathcal{V} = \mathcal{W}$ or $\mathcal{V} = A\mathcal{W}$ with $\mathcal{V}$ and $\mathcal{W}$ both Krylov subspaces.

- **Enforcement of Petrov-Galerkin Conditions**
  Arnoldi's procedure and Lanczos algorithm are two choices for building orthogonal or biorthogonal sequence to enforce the projection conditions.

The iterative methods discussed in this report are generally split into two categories, one based on Arnoldi's procedure and the other on Lanczos biorthoganalization. The most popular for the first family are Conjugate Gradient and General Minimum Residual methods, while Bi-Conjugate Gradient and Quasi-Minimum Residual methods represent the Lanczos family.

5.2.1 Conjugate Gradient Algorithm

The conjugate gradient (CG) algorithm is one of the best known iterative techniques for solving sparse symmetric positive definite (SPD) linear systems. This method is a realization of an orthogonal projection technique onto the Krylov subspace $\mathcal{K}_m(A, r_0)$, where r_0 is the initial residual. Because A is symmetry, some simplifications resulting from the three-term Lanczos recurrence will lead to more elegant algorithms (Demmel, 1997).

ALGORITHM CG (Saad, 2003)

1. *Compute* $r_0 := b - Ax_0$, $p_0 := r_0$
2. *For* $j = 0, 1, \cdots$, *until convergence, Do*
3. $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4. $x_{j+1} := x_j + \alpha_j p_j$
5. $r_{j+1} := r_j - \alpha_j A p_j$
6. $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7. $p_{j+1} := r_{j+1} + \beta_j p_j$
8. *EndDo*

- **Applicability**

  Matrix A is SPD.

- **Subspaces**

  Choose $\mathcal{W} = \mathcal{V} = \mathcal{K}_m(A, r_0)$, in which initial residual $r_0 = b - Ax_0$.

- **Symmetric Lanczos Procedure**

  This procedure can be viewed as a simplification of the Arnoldi's procedure when A is symmetric. Great three-term Lanczos recurrence is discovered when the symmetry of A is considered (Demmel, 1997).

- **Optimality**

  If A is SPD and one chooses $\mathcal{W} = \mathcal{V}$, enforcing Petrov-Galerkin conditions minimizes the A-norm of the error over all vectors $x \in \mathcal{W}$, i.e., $\hat{x}$ solves the problem,

  $$\min_{x \in \mathcal{W}} \|x - x^*\|_A, x^* = A^{-1}b \tag{5.2}$$

  From the lemma above, one can derive global minimization property of the Conjugate Gradient method. The vector x_k in the Conjugate Gradient method solves the minimization problem

  $$\min_x \phi(x) = \frac{1}{2}\|x - x^*\|_A^2, x - x_0 \in \mathcal{K}_k(A, r_0) \tag{5.3}$$

- **Convergence**

  In exact arithmetic, the Conjugate Gradient method will produce the exact solution to the linear system $Ax = b$ in at most n steps and it owns the superlinear convergence rate. The behavior of Conjugate Gradient algorithm in finite precision is much more complex. Due to rounding errors, orthogonality is lost quickly and finite termination does not hold anymore. What is more meaningful in application problems would be to use CG method for solving large, sparse, well-conditioned linear systems in far fewer than n iterations.

5.2.2 GMRES

The Generalized Minimum Residual method is able to deal with more general type of matrices.

ALGORITHM GMRES (Saad, 2003)

1. *Compute* $r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, *and* $v_1 := r_0/\beta$
2. *For* $j = 1, 2, \cdots, m$, *Do*
3. *Compute* $\omega_j := Av_j$
4. *For* $i = 1, \cdots, j$, *Do*
5. $h_{ij} := (\omega_j, v_i)$
6. $\omega_j := \omega_j - h_{ij}v_i$
7. *EndDo*
8. $h_{j+1,j} = \|\omega_j\|_2$. *If* $h_{j+1,j} = 0$ *set* $m := j$ *and go to 11*
9. $v_{j+1} = \omega_j/h_{j+1,j}$
10. *EndDo*
11. *Define the* $(m+1) \times m$ *Hessenberg matrix* $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$
12. *Compute* y_m, *the minimizer of* $\|\beta_i e_1 - \bar{H}_m y\|_2$, *and* $x_m = x_0 + V_m y_m$

- **Applicability**

  Matrix A is nonsingular.

- **Subspaces**

  Choose $\mathcal{W} = \mathcal{K}_m(A, r_0)$ and $\mathcal{V} = A\mathcal{W} = A\mathcal{K}_m(A, r_0)$, in which initial residual $r_0 = b - Ax_0$.

- **Arnoldi's Procedure**

  Classic Arnoldi's procedure (modified Gram-Schmidt) is followed in GMRES (Bai).

- **Optimality**

  If one chooses $\mathcal{V} = A\mathcal{W}$, enforcing Petrov-Galerkin conditions solves the least square problem

$$\|b - A\tilde{x}\|_2 = \min_{x \in \mathcal{W}} \|b - Ax\|_2 \tag{5.4}$$

- **Convergence**

  It has been shown that in exact arithmetic, GMRES can not breakdown and will give exact solutions in at most n steps. In practice, the maximum steps GMRES can run depends on the memory due to the fact it needs to store all Arnoldi vectors. Restarting schemes have been proposed for a fixed m, which is denoted by GMRES(m). Typical value for m can be $m \in [5, 20]$. GMRES(m) can not breakdown in exact arithmetic before the exact solution has been reached. But it may never converge for $m < n$ (Bai).

5.2.3 BiCGStab and QMR

These two methods are based on nonsymmetric Lanczos procedure, which is quite different from Arnoldi's in the sense that it formulates biorthogonal instead of orthogonal sequence. They are counterparts of CG and GMRES method, which follows similar derivation procedure except the Lanczos biorthogonalization is used instead of Arnoldi's procedure (Bai).

5.3 Preconditioning Techniques

Lack of robustness is a widely recognized weakness of iterative solvers relative to direct solvers. Using preconditioning techniques can greatly improve the efficiency and robustness of iterative methods. Preconditioning is simply a means of transforming the original linear system into one with the same solution but easier to solve with an iterative solver. Generally speaking, the reliability of iterative techniques, when dealing with various applications, depends much more on the quality of the preconditioner than on the particular Krylov subspace accelerator used.

The first step in preconditioning is to find a preconditioning matrix M. The matrix M can be defined in many different ways but there are a few minimal requirements the M is supposed to satisfy (Benzi, 2002).

1. From practical point of view, the most important requirement of M is that it should be inexpensive to solve linear system $Mx = b$. This is because the preconditioned algorithm will all require a linear system solution with the matrix M at each step.

2. The matrix M should be somehow close to A and it should not be singular. We can see that actually most powerful preconditioners are constructed directly from A.

3. The preconditioned $M^{-1}A$ should be well-conditioned or has very few extreme eigenvalues thus M can accelerate convergence dramatically.

Once a preconditioner M is available, there are three ways to apply it.

1. Left Preconditioning

$$M^{-1}Ax = M^{-1}b \tag{5.5}$$

2. Right Preconditioning

$$AM^{-1}u = b, x \equiv M^{-1}u \tag{5.6}$$

3. Split Preconditioning

   It is a very common situation that M is available in factored form $M = M_L M_R$, in which, typically,

M_L and M_R are triangular matrices. Then the preconditioning can be split,

$$M_L^{-1} A M_R^{-1} u = b, x \equiv M_R^{-1} u \tag{5.7}$$

It is imperative to preserve symmetry when the original matrix A is symmetric, so the split preconditioner seems mandatory in this case.

Consider that a matrix A that is symmetric and positive definite and assume that a preconditioner M is available. The preconditioner M is a matrix that approximates A in some yet-undefined sense. We normally require that the M is also symmetric positive definite.

In order to preserve the nice SPD property, in the case when M is available in the form of an incomplete Cholesky factorization, $M = LL^T$, people can simply just use the split preconditioning, which yields the SPD matrix

$$L^{-1} A L^{-T} u = L^{-1} b, x \equiv L^{-T} u \tag{5.8}$$

However, it is not necessary to split the preconditioner in this manner in order to preserve symmetry. Observe that $M^{-1}A$ is self-adjoint for the M inner product

$$(x, y)_M \equiv (Mx, y) = (x, My) \tag{5.9}$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M \tag{5.10}$$

Therefore, an alternative is to replace the usual Euclidean inner product in the CG algorithm with the M inner product (Saad, 2003).

If the CG algorithm is rewritten for this new inner product, denoting by $r_j = b - Ax_j$ the original residual and by $z_j = M^{-1} r_j$ the residual for the preconditioned system, the following sequence of operations is obtained, ignoring the initial step:

1. $\alpha_j := (z_j, z_j)_M / (M^{-1} A p_j, p_j)_M$,
2. $x_{j+1} := x_j + \alpha_j p_j$,
3. $r_{j+1} := r_j - \alpha_j A p_j$ and $z_{j+1} := M^{-1} r_{j+1}$,
4. $\beta_j := (z_{j+1}, z_{j+1})_M / (z_j, z_j)_M$,
5. $p_{j+1} := z_{j+1} + \beta_j p_j$.

Since $(z_j, z_j)_M = (r_j, z_j)$ and $(M^{-1} A p_j, p_j)_M = (A p_j, p_j)$, the M inner products do not have to be computed explicitly. With this observation, the following algorithm is obtained.

ALGORITHM Preconditioned CG (Saad, 2003)

1. *Compute* $r_0 := b - Ax_0$, $z_0 := M^{-1}r_0$, $p_0 := z_0$
2. *For* $j = 0, 1, \cdots$, *until convergence, Do*
3. $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
4. $x_{j+1} := x_j + \alpha_j p_j$
5. $r_{j+1} := r_j - \alpha_j Ap_j$
6. $z_{j+1} := M^{-1}r_{j+1}$
7. $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
8. $p_{j+1} := z_{j+1} + \beta_j p_j$
9. *EndDo*

5.4 Preconditioners

Finding a good preconditioner to solve a given sparse linear system is often viewed as a combination of art and science. Theoretical results are rare and some methods work surprisingly well, often despite expectations. As it is mentioned before, the preconditioner M is always close to A in some undefined-yet sense. Some popular preconditioners will be introduced in this chapter.

5.4.1 Jacobi Preconditioner

This might be the simplest preconditioner people can think of. If A has widely varying diagonal entries, we may just use diagonal preconditioner $M = \text{diag}(a_{11}, \cdots, a_{nn})$. One can show that among all possible diagonal preconditoners, this choice reduces the condition number of $M^{-1}A$ to within a factor of n of its minimum value.

5.4.2 Incomplete Cholesky Preconditioner

Another simple way of defining a preconditioner that is close to A is to perform an incomplete Cholesky factorization of A. Incomplete factorization formulates an approximation of $A \approx \hat{L}\hat{L}^T$, but with less or no fill-ins relative to the complete factorization $A = LL^T$ (Demmel, 1997).

ALGORITHM Incomplete Cholesky Factorization (Saad, 2003)

1. For $j = 1, 2, \cdots, n$, *Do*
3. $\quad l_{jj} := \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$
4. $\quad$ *For* $i = j+1, \cdots, n$, *Do*
5. $\quad\quad l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk})/l_{jj}$
6. $\quad\quad$ *Apply dropping rule to* l_{ij}
7. $\quad$ *EndDo*
8. EndDo

There are many ways to control the number of fill-ins in IC factorization. No fill-in version of incomplete Cholesky factorization IC(0) is rather easy and inexpensive to compute. On the other hand, it often leads to a very crude approximation of A, which may result in the Krylov subspace accelerator requiring too many iterations to converge. To remedy this, several alternative incomplete factorizations have been developed by researchers by allowing more fill-in in L, such as incomplete Cholesky factorization with dropping threshold IC(ϵ). In general, more accurate IC factorizations require fewer iterations to converge, but the preprocessing cost to compute the factors is higher.

5.4.3 Robust Incomplete Factorization

Incomplete factorization preconditioners are quite effective for many application problems but special care must be taken in order to avoid breakdowns due to the occurrence of non-positive pivots during the incomplete factorization process.

The existence of an incomplete factorization $A \approx \hat{L}\hat{L}^T$ has been established for certain classes of matrices. For the class of M-matrices, the existence of incomplete Cholesky factorization was proved for arbitrary choices of the sparsity pattern (Meijerink and van der Vorst, 1977). The existence result was extended shortly thereafter to a somewhat larger class (that of H-matrices with positive diagonal entries) (Manteuffel, 1980; Varga et al., 1980; Robert, 1982). Benzi and Tůma (2003) presents reviews on the topic of searching for robust incomplete factorization algorithms and an robust algorithm based on A-Orthogonalization has been proposed.

In order to construct triangular factorization of A, the well-known is not the only choice. Benzi and Tůma (2003) shows how the factorization $A = LDL^T$ (root-free factorization) can be obtained by means of an A-orthogonalization process applied to the unit basis vectors $e_1, e_2, \cdots, e_n$. This is simply the Gram-Schmidt process with respect to the inner product generated by the SPD matrix A. This idea is not new and as a matter of fact, it was originally proposed at as early as 1940's in Fox et al. (1948). It has been observed in Hestenes and Stiefel (1952) that A-orthogonalization of the unit basis vectors is closely related to Gaussian elimination but this algorithm costs twice as much as the Cholesky factorization in the dense case.

Factored Approximate Inverse Preconditioner

In reference Benzi et al. (1996) A-orthogonalization has been exploited to construct factored sparse approximate inverse preconditioners noting the fact that A-orthogonalization also produces the inverse factorization $A^{-1} = ZD^{-1}Z^T$ (with Z unit upper triangular and D diagonal). Because the A-orthogonaliza-tion, even when performed incompletely, is not subject to pivot breakdowns, these preconditioners are reliable (Benzi et al., 2000). However, they are often less effective than incomplete Cholesky preconditioning at reducing the number of PCG iterations and their main interest stems from the fact that the preconditioning operation can be applied easily in parallel because triangular solve is not necessary in approximate inverse preconditioning.

Reference Benzi and Tůma (2003) investigates the use of A-orthogonalization as a way to compute an incomplete factorization of A rather than A^{-1} thus a reliable preconditioning algorithm can be developed. The basic A-orthogonalization procedure can be written as follows (Benzi et al., 2000).

ALGORITHM Incomplete Factored Approximate Inverse (Benzi et al., 1996)

1. *Let* $z_i^{(0)} = e_i$, *for* $i = 1, 2, \cdots, n$
2. *For* $i = 1, 2, \cdots, n$, *Do*
3. *For* $j = i, i+1, \cdots, n$, *Do*
4. $p_j^{(i-1)} := a_i^T z_j^{(i-1)}$
5. *EndDo*
6. *For* $j = i+1, \cdots, n$, *Do*
7. $z_j^{(i)} := z_j^{(i-1)} - (\frac{p_j^{(i-1)}}{p_i^{(i-1)}})$
8. *Apply dropping to* $z_j^{(i)}$
9. *EndDo*
10. *EndDo*
11. *Let* $z_i := z_i^{(i-1)}$ *and* $p_i := p_i^{(i-1)}$, *for* $i = 1, 2, \cdots, n$.
12. *Return* $Z = [z_1, z_2, \cdots, z_n]$ *and* $D = diag(p_1, p_2, \cdots, p_n)$.

The basic algorithm described above can suffer a breakdown when a negative or zero value of a pivot p_i. When no dropping is applied, $p_i = z_i^T A z_i > 0$. The incomplete procedure is well defined, i.e., no breakdown can occur, if A is an H-matrix (in the absence of round-off). In the general case, breakdowns can occur. Breakdowns have a crippling effect on the quality of the preconditioner. A negative p_i would result in an approximate inverse which is not positive definite; a zero pivot would force termination of the procedure, since step (7) cannot be carried out.

The way proposed to avoid non-positive pivots is simply to recall that in the exact A-orthogonalization

process, the p_i's are the diagonal entries of matrix D which satisfies the matrix equation

$$Z^T A Z = D \tag{5.11}$$

hence for $1 < i < n$

$$p_i = z_i^T A z_i > 0 \tag{5.12}$$

since A is SPD and $z_i \neq 0$. In the exact process, the following equality holds

$$p_i = z_i^T A z_i = a_i^T z_i \quad \text{and} \quad p_j = z_i^T A z_j = a_i^T z_j \tag{5.13}$$

Clearly it is more economical to compute the pivots using just inner product $a_i^T z_i$ rather than the middle expression involving matrix-vector multiply. However, because of dropping and the resulting loss of A-orthogonality in the approximate $\bar{z}$-vectors, such identities no longer hold in the inexact process and for some matrices one can have

$$a_i^T \bar{z}_i \ll \bar{z}_i^T A \bar{z}_i \tag{5.14}$$

The robust algorithm requires that the incomplete pivots $\bar{p}_i$'s be computed using the quadratic form $\bar{z}_i^T A \bar{z}_i$ throughout the AINV process, for $i = 1, 2, \cdots, n$.

ALGORITHM Stabilized Incomplete Approximate Inverse (Benzi et al., 2000)

1. *Let* $z_i^{(0)} = e_i$, *for* $i = 1, 2, \cdots, n$
2. *For* $i = 1, 2, \cdots, n$, *Do*
3. $\quad v_i := A z_i^{(i-1)}$
4. $\quad$ *For* $j = i, i+1, \cdots, n$, *Do*
5. $\quad\quad p_j^{(i-1)} := v_i^T z_j^{(i-1)}$
6. $\quad$ *EndDo*
7. $\quad$ *For* $j = i+1, \cdots, n$, *Do*
8. $\quad\quad z_j^{(i)} := z_j^{(i-1)} - (\frac{p_j^{(i-1)}}{p_i^{(i-1)}})$
9. $\quad\quad$ *Apply dropping to* $z_j^{(i)}$
10. $\quad$ *EndDo*
11. *EndDo*
12. *Let* $z_i := z_i^{(i-1)}$ *and* $p_i := p_i^{(i-1)}$, *for* $i = 1, 2, \cdots, n$.
13. *Return* $Z = [z_1, z_2, \cdots, z_n]$ *and* $D = diag(p_1, p_2, \cdots, p_n)$.

Obviously, the robust (referred to as SAINV) and plain algorithm are mathematically equivalent. However, the incomplete process obtained by dropping in the z-vectors in step (9) of the robust algorithm leads to a reliable approximate inverse. This algorithm, in exact arithmetic, is applicable to any SPD matrix without breakdowns. The computational cost of SAINV is higher than basic AINV and special care has to be taken to do sparse-sparse matrix-vector multiply.

Incomplete Factorization by SAINV

Consider now the exact algorithm (with no dropping) and write $A = LDL^T$ with L unit lower triangular and D diagonal. Observe that L in the LDL^T factorization of A and the inverse factor satisfy

$$AZ = LD \quad \text{or} \quad L = AZD^{-1} \tag{5.15}$$

where D is the diagonal matrix containing the pivots. This easily follows from

$$Z^T AZ = D \quad \text{and} \quad Z^T = L^{-1} \tag{5.16}$$

If we recall that pivot $d_j = p_j = z_j^T Az_j = \langle Az_j, z_j \rangle$, then by equating corresponding entries of AZD^{-1} and $L = [l_{ij}]$ we find that (Benzi and Tůma, 2003; Bollhöfer and Saad, 2001)

$$l_{ij} = \frac{\langle Az_j, z_i \rangle}{\langle Az_j, z_j \rangle} \quad i \geq j \tag{5.17}$$

Hence, the L factor of A can be obtained as a by-product of the A-orthogonal-ization, at no extra cost. In the implementation of SAINV, the quantities l_{ij} in Equation 5.17 are the multipliers that are used in updating the columns of Z. Once the update is computed, they are no longer needed and are discarded. To obtain an incomplete factorization of A, we do just the opposite; we save the multipliers l_{ij} , and discard the column vectors z_j as soon as they have been computed and operated with. Hence, the incomplete L factor is computed by columns; these columns can be stored in place of the z_j vectors, with minimal modifications to the code. Here, we are assuming that the right-looking form of SAINV is being used. If the left-looking one is being used, then L would be computed by rows. Please refer to Benzi and Tůma (2003) for more implementation details.

5.5 Numerical Experiments

Matrices from soil-structure interaction finite element analysis have been extracted from simulation system to study the performance of different preconditioning techniques on PCG method. The prototype of soil structure model has been shown in Figures 5.1 and 5.2. In order to introduce both of the nonlinear theories for soil and structures, we use continuum elements to model the soil and beam elements for the structures. Matrices from static pushover analysis and dynamic ground motion analysis have been collected for this research.

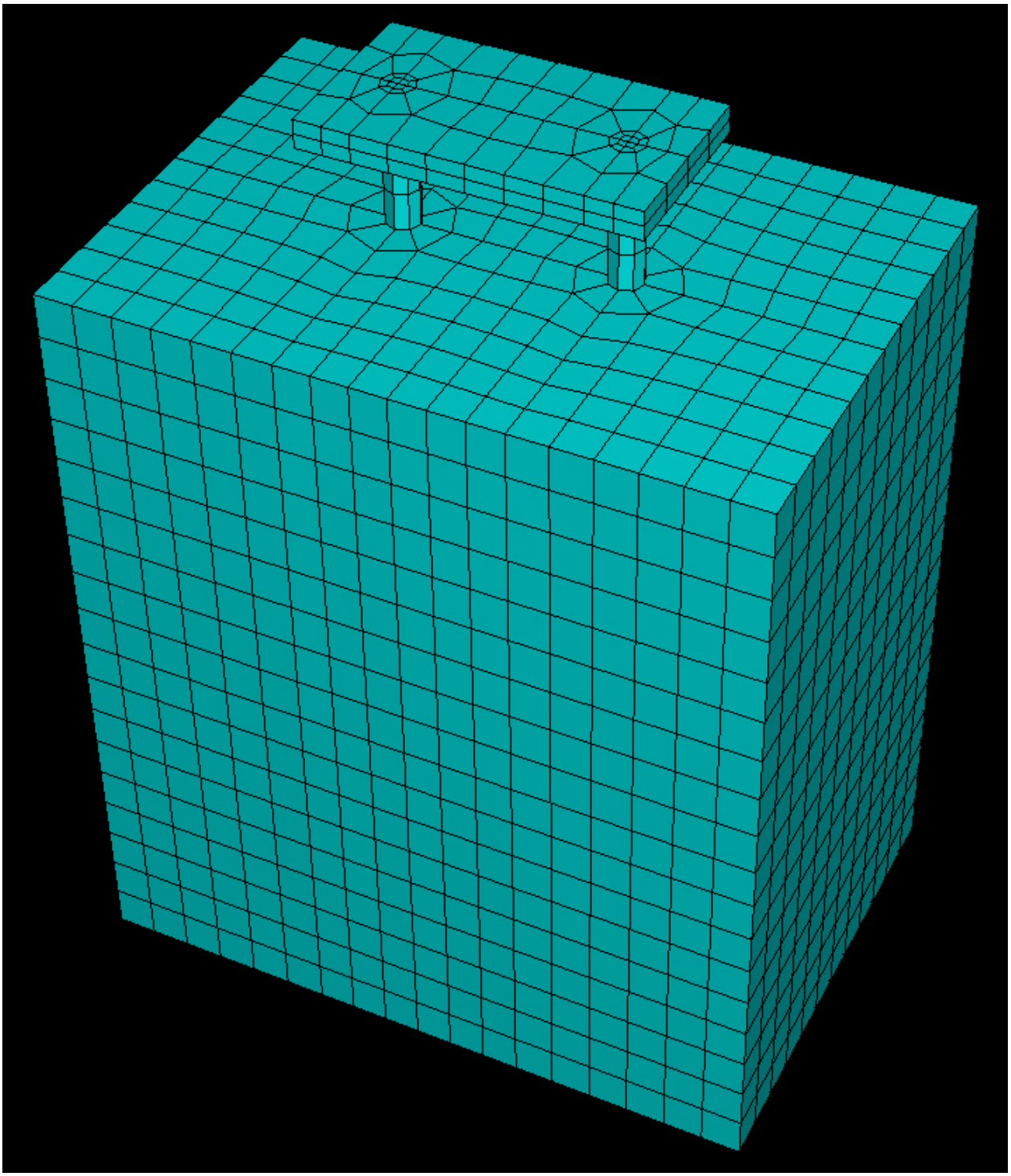


Figure 5.1: Finite Element Mesh of Soil-Structure Interaction Model

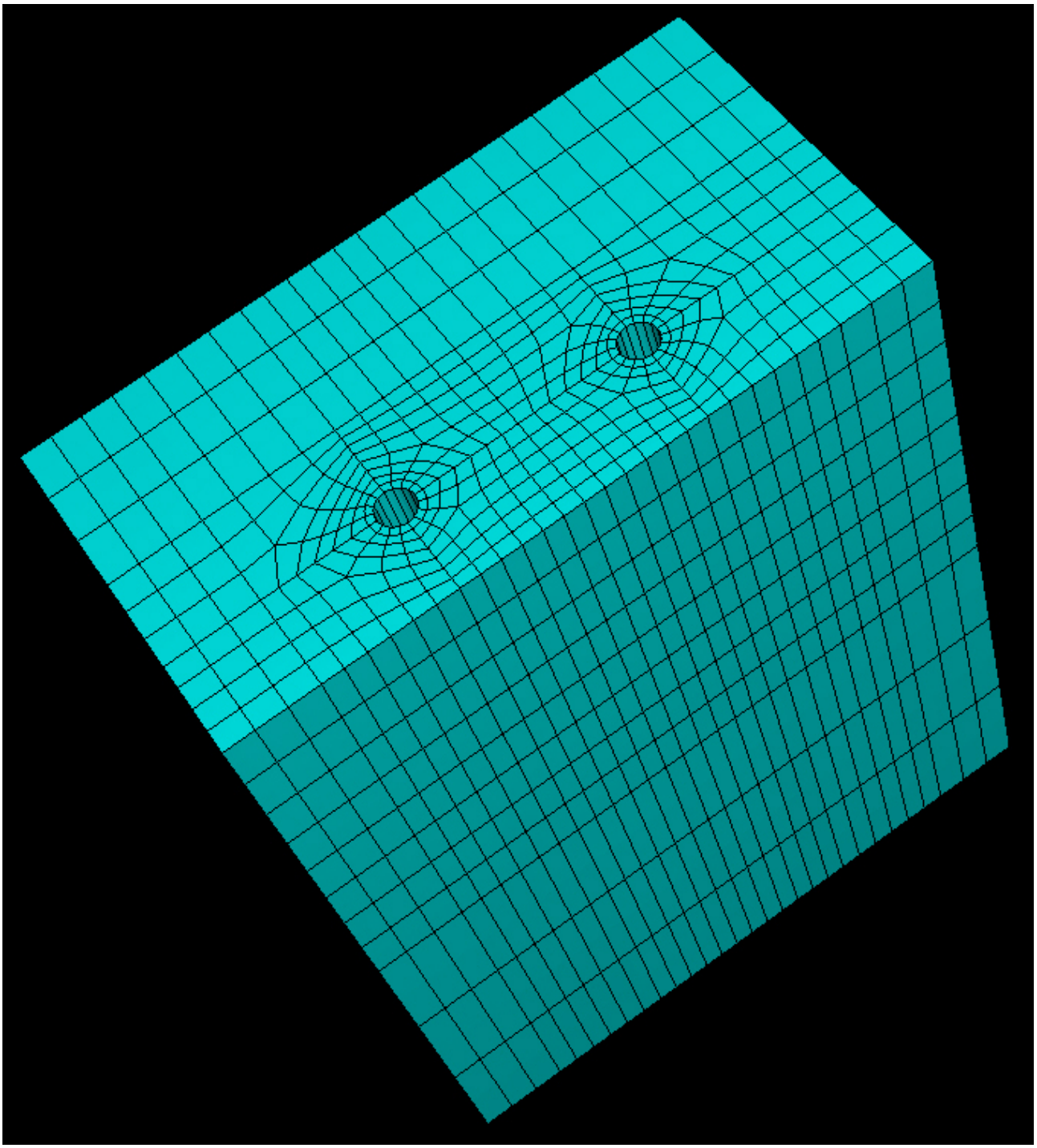


Figure 5.2: Finite Element Mesh of Soil-Structure Interaction Model

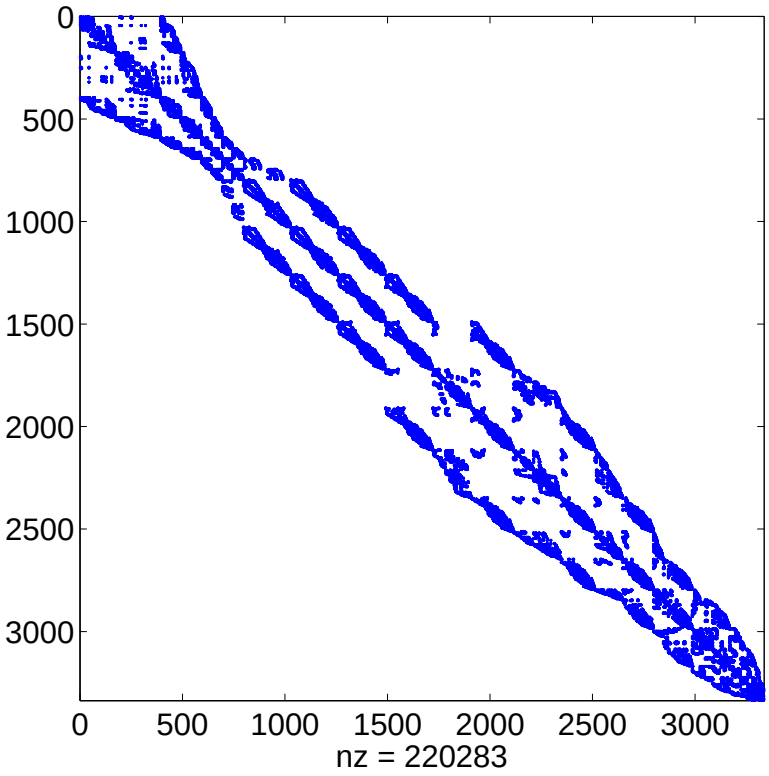


Figure 5.3: Matrices $N = 3336$ (Continuum FEM)

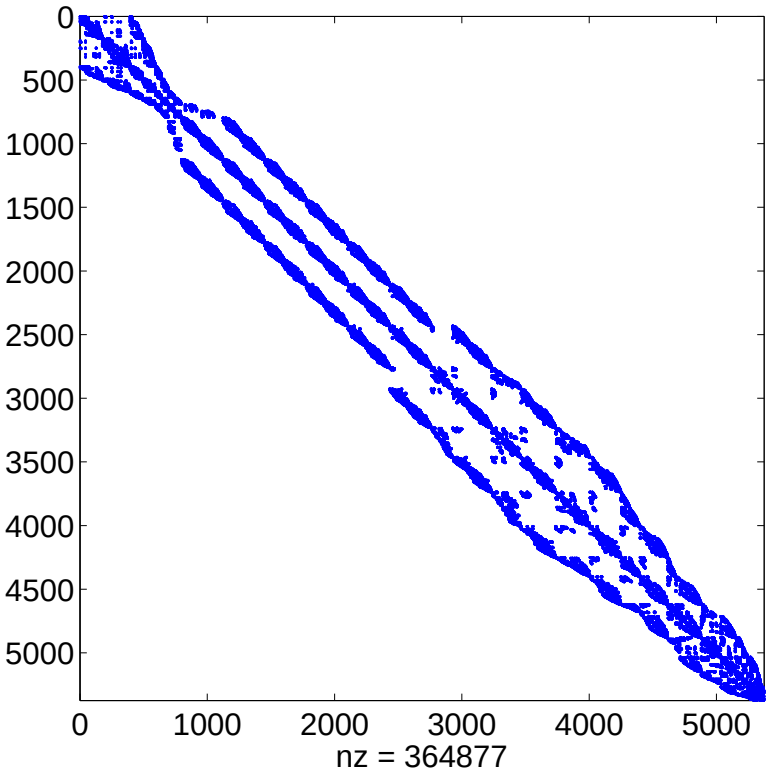


Figure 5.4: Matrices $N = 5373$ (Continuum FEM)

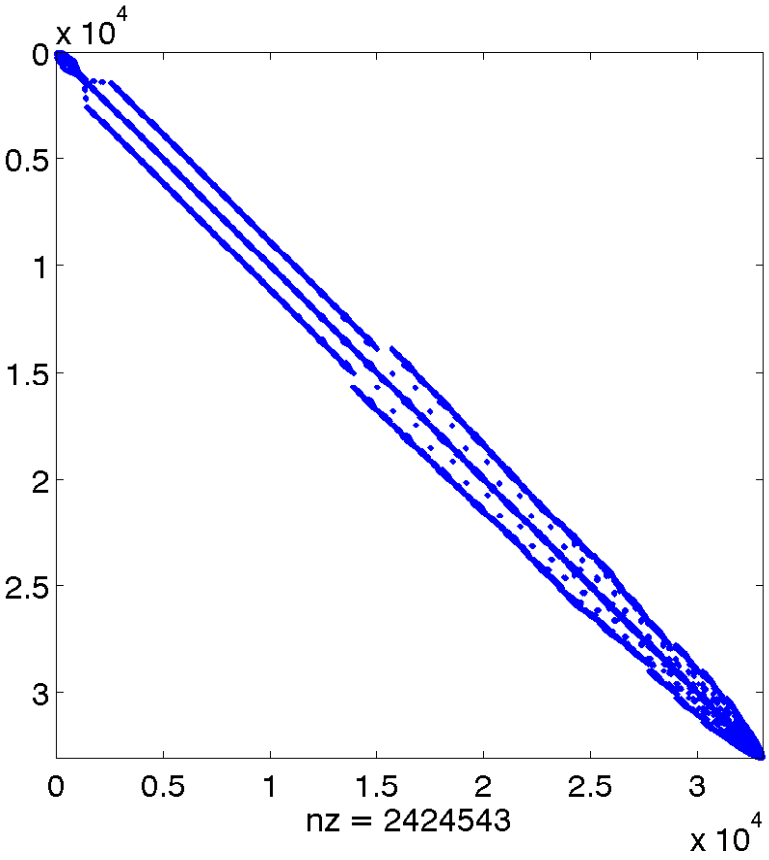


Figure 5.5: Matrices $N = 33081$ (Continuum FEM)

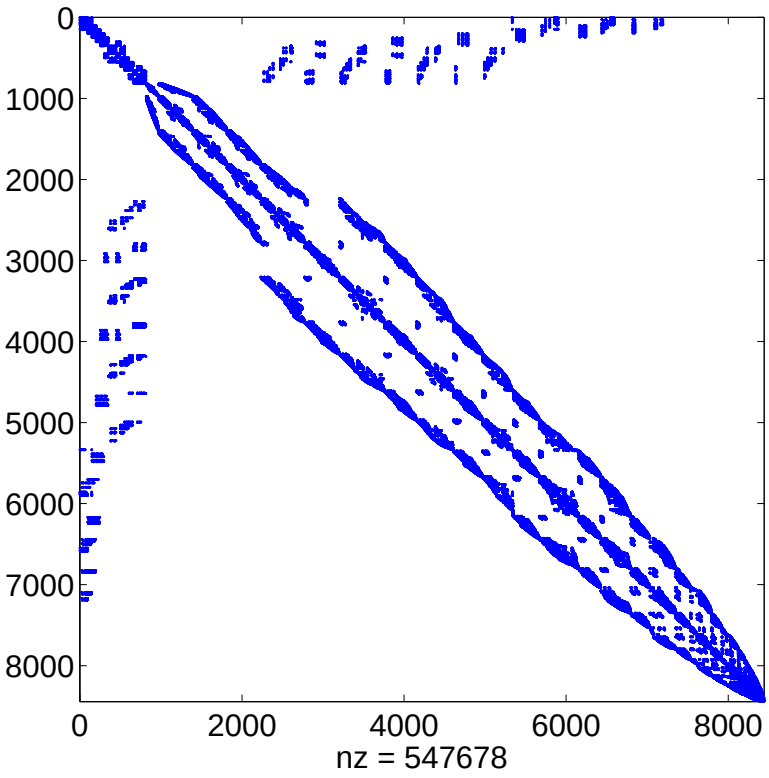


Figure 5.6: Matrices $N = 8842$ (Soil-Beam Static FEM)

Table 5.1: Matrices in FEM Models

Continuum Model (Static)			
Matrix	Property	Dimension	# Nonzeros
m1188	SPD	3336	220283
m1968	SPD	5373	364877
m11952	SPD	33081	2424543
Soil-Beam Model (Static and Dynamic)			
Matrix	Property	Dimension	# Nonzeros
SoilBeam	SPD	8442	547678
SoilBeamDyn	SPD	8442	547671

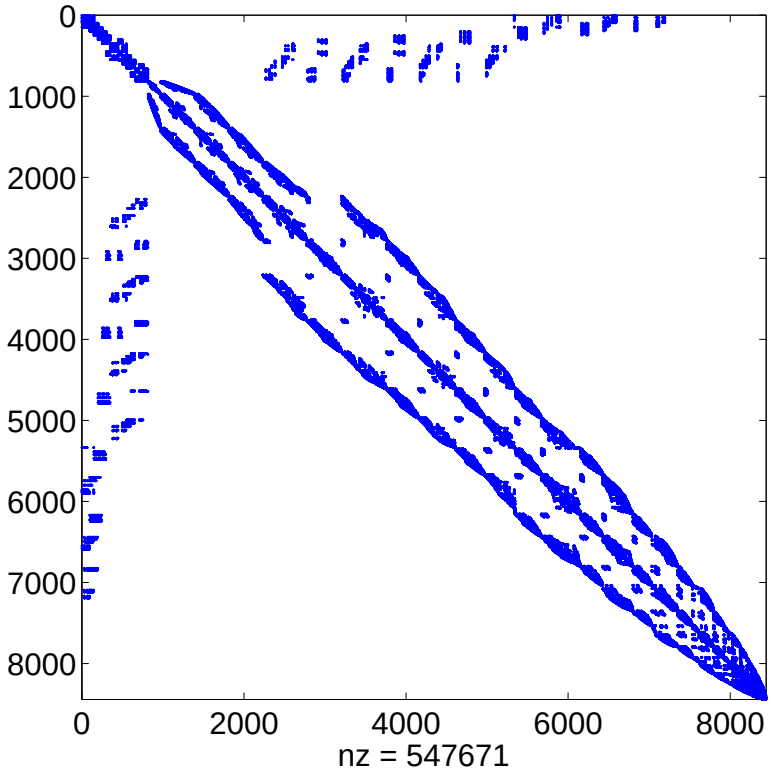


Figure 5.7: Matrices $N = 8842$ (Soil-Beam Dynamic FEM)

SPD matrices have been studied using Conjugate Gradient method with or without preconditioning. Performance has been summarized in Table 5.2.

Table 5.2: Performance of CG and PCG Method (Continuum FEM)

3336 DOFs FEM (Static)					
Preconditioner	# Iter	Pre Time(s)	Iter Time(s)	Total Time(s)	Density[1]
-	4376	-	54.82	54.82	-
Jacobi	1612	0.01	20.18	20.19	-
IC(0)	413	2.19	11.07	13.26	1.00
IC(1e-6)	5	5.90	0.47	6.37	5.88
RIF2(1e-2)	571	9.37	14.94	24.31	0.94
RIF3(1e-2)	541	6.80	14.13	20.93	0.94
5373 DOFs FEM (Static)					
Preconditioner	# Iter	Pre Time(s)	Iter Time(s)	Total Time(s)	Density
-	4941	-	103.78	103.78	-
Jacobi	1711	0.01	36.61	36.62	-
IC(0)	437	6.5	20.38	26.88	1.00
IC(1e-6)	6	19.81	1.3	21.11	8.10
RIF2(1e-2)	599	25.71	26.55	52.26	0.96
RIF3(1e-2)	566	21.31	25.23	46.54	0.96
33081 DOFs FEM (Static)					
Preconditioner	# Iter	Pre Time(s)	Iter Time(s)	Total Time(s)	Density
-	6754	-	952.53	952.53	-
Jacobi	2109	0.03	308.46	308.49	-
IC(0)	565	273.83	173.05	446.88	1.00
IC(1e-6)[2]					
RIF2(1e-2)	694	1172.7	211.88	1384.58	0.99
RIF3(1e-2)	664	1245.4	202.67	1448.07	0.99

[1]Density is defined as the number of non-zeros of the incomplete factor divided by the number of non-zeros in the lower triangular part of A.

[2]Could not continue because memory requirement larger than 1.4GB.

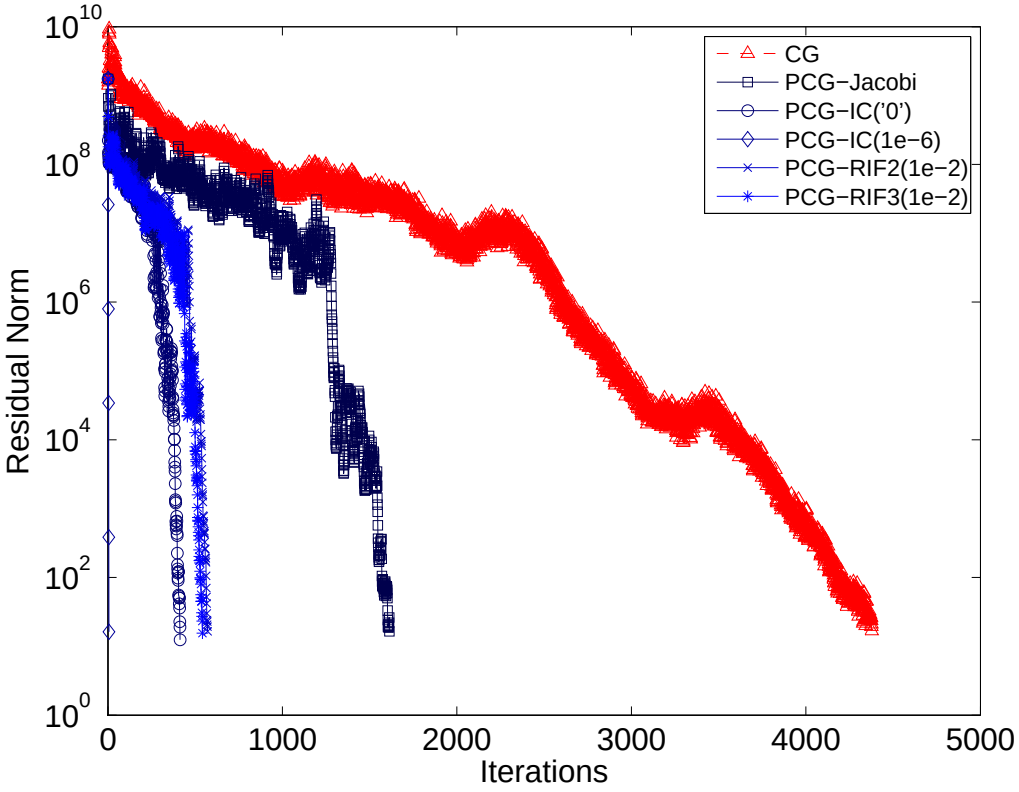


Figure 5.8: Convergence of CG and PCG Method (3336 DOFs Model)

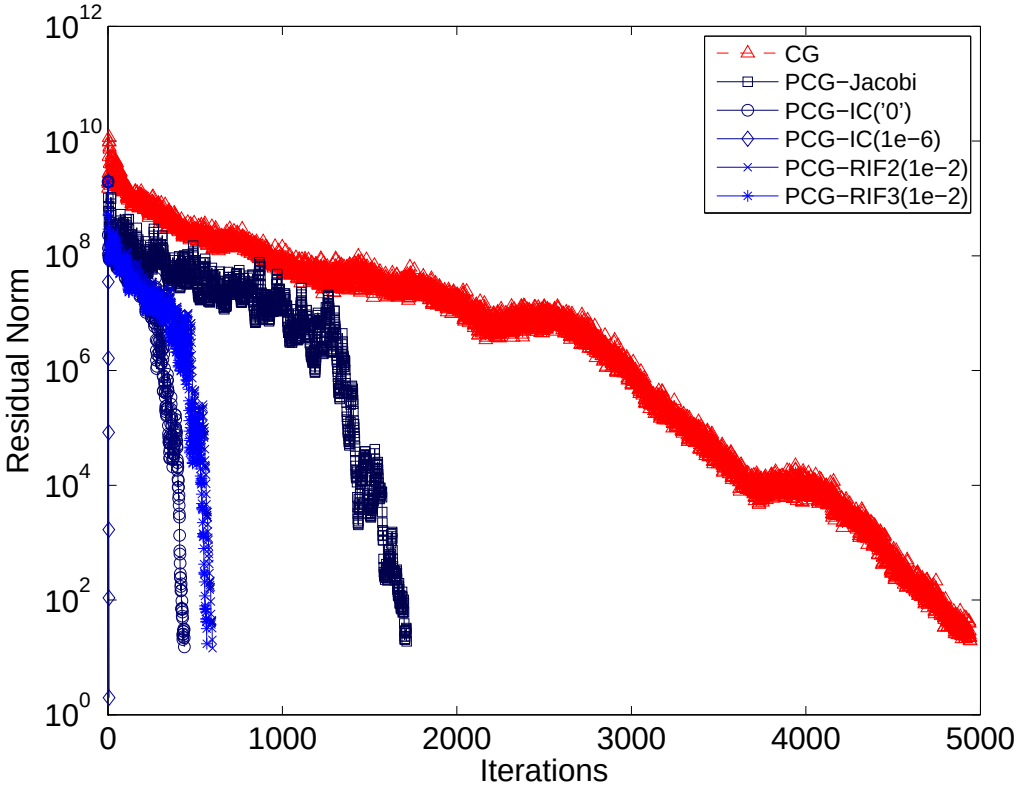


Figure 5.9: Convergence of CG and PCG Method (5373 DOFs Model)

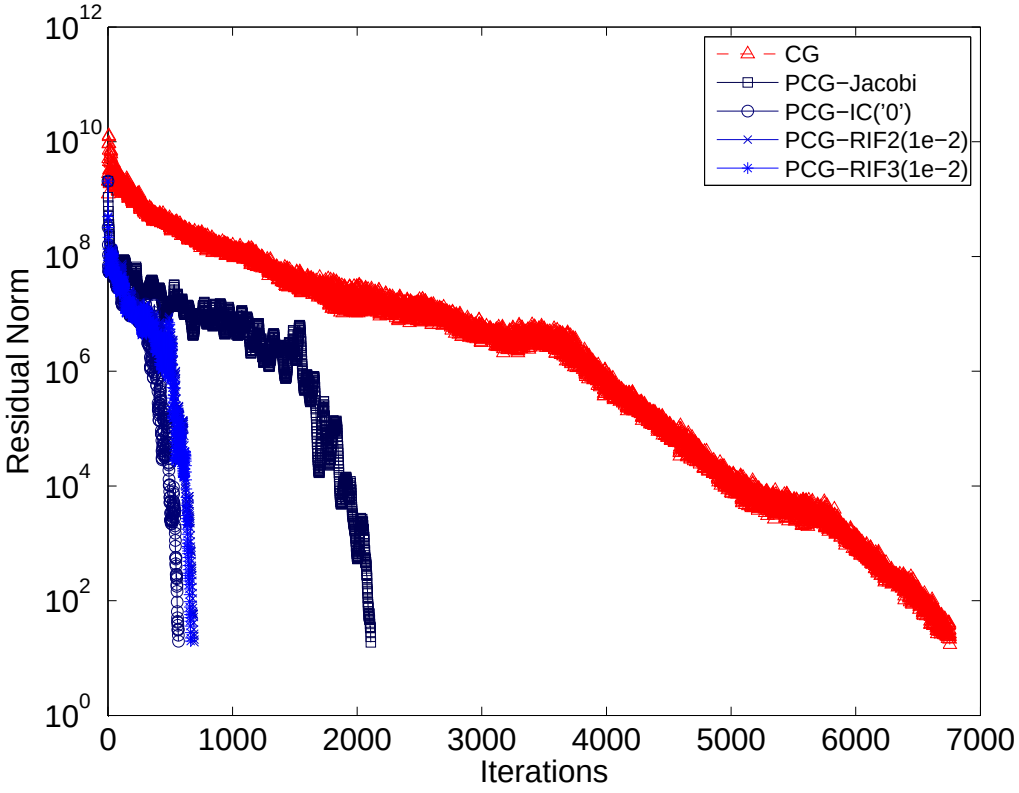


Figure 5.10: Convergence of CG and PCG Method (33081 DOFs Model)

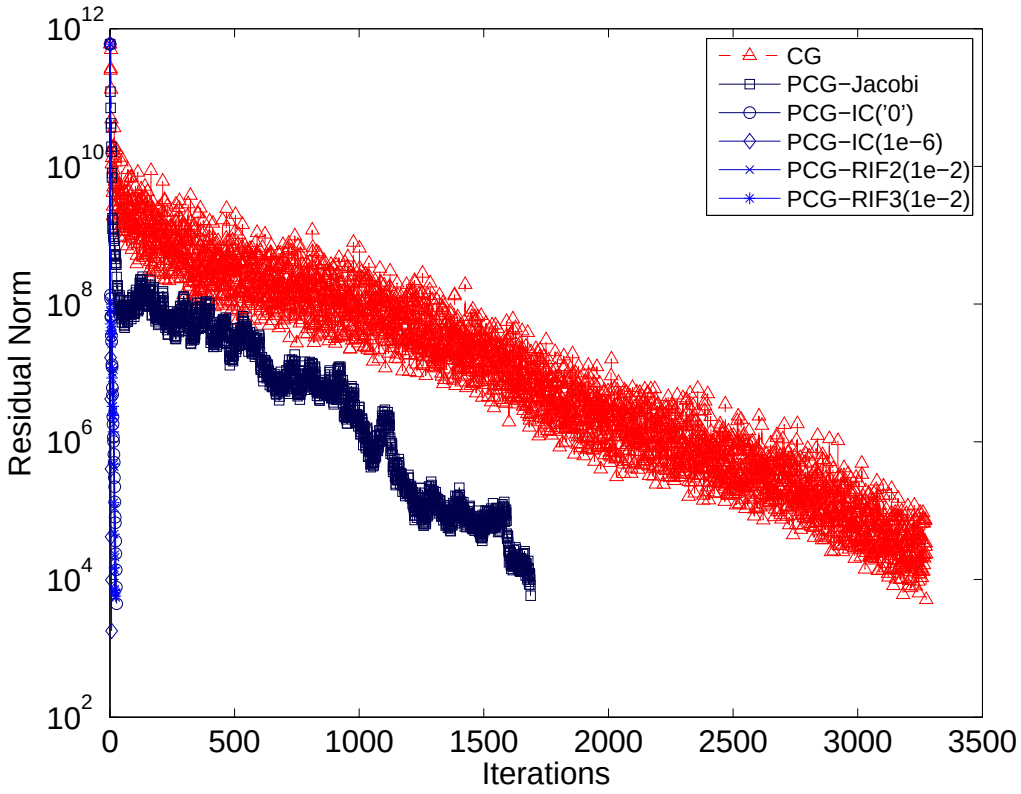


Figure 5.11: Convergence of CG and PCG Method (Soil-Beam Static Model)

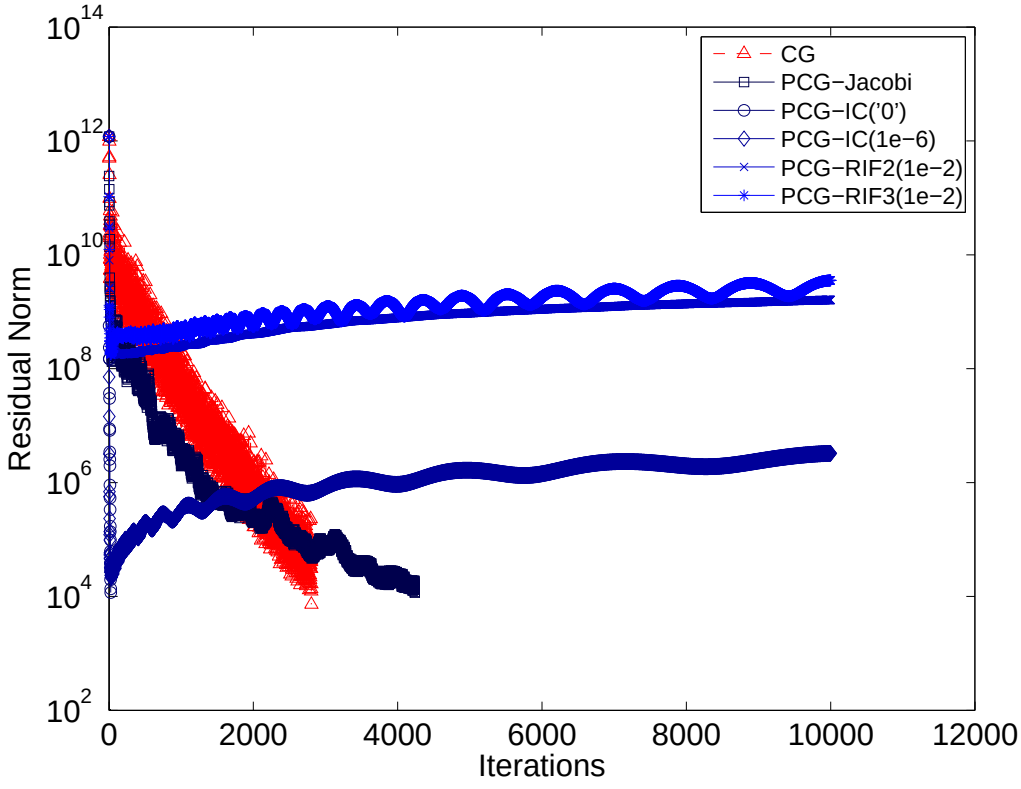


Figure 5.12: Convergence of CG and PCG Method (Soil-Beam Dynamic Model)

Table 5.3: Performance of CG and PCG Method (Soil-Beam FEM)

8842 DOFs Soil-Beam FEM (Static)					
Preconditioner	# Iter	Pre Time(s)	Iter Time(s)	Total Time(s)	Density[3]
-	3274	-	102.5	102.5	-
Jacobi	1687	0.01	54.56	54.57	-
IC(0)	26	15.77	1.95	17.72	1.00
IC(1e-6)	6	110.17	2.79	112.96	15.11
RIF2(1e-6)[4]	23	3364.8	3.44	3368.24	4.32
RIF3(1e-6)[4]	31	34541	9.26	34550.26	16.37
8842 DOFs Soil-Beam FEM (Dynamic)					
Preconditioner	# Iter	Pre Time(s)	Iter Time(s)	Total Time(s)	Density
-	3276	-	136.7	136.7	-
Jacobi	MaxIt				
IC(0)	MaxIt				
IC(1e-6)	MaxIt				
RIF2(1e-2)	MaxIt				
RIF3(1e-2)	MaxIt				

[3]Density is defined as the number of non-zeros of the incomplete factor divided by the number of non-zeros in the lower triangular part of A.

[4]Iteration with tolerance 1e-2 failed to converge.

5.6 Conclusion and Future Work

1. For the soil-structure interaction problems investigated in this report , Conjugate Gradient method works fine and the convergence is acceptable for most cases.

2. Incomplete Cholesky factorization preconditioner has been shown to be very powerful in static pushover problems.

3. Dynamic problems formulated by Newmark integration scheme have not been extensively tested. But according to the data available so far, neither IC nor RIF preconditioners performed well and further testing is necessary to reach a more persuasive conclusion. The difficulty in dynamic analysis results from the fact that consistent mass and damping matrices used in continuum finite element formulations significantly degrade the conditioning number of the final system. This situation deteriorates when penalty handler is used to apply multiple point constraints, which introduces huge off-diagonal numbers to stiffness, mass and damping matrices (Cook et al., 2002).

4. Robust incomplete factorization preconditioning based on A-orthogonalization has not been shown competitive with IC preconditioners in this research. It is also worth noting that all timings are taken in MATLAB. There are much more improvement can be achieved with a carefully coded FORTRAN program.

5. Static analysis has been extensively studied in this report . It can be safely concluded that IC(0) and Jacobi preconditioners are good choices for the nonlinear soil-beam interaction simulations.

6. Dynamic analysis has not been studied well enough to draw a detailed conclusion in this report . Generally speaking, one should be alert if iterative solver is to be used for dynamic analysis. This partially comes from the fact that mass and damping matrices undoubtedly alter the structures of the coefficient matrix. This situation becomes more complicated if penalty handler is used to introduce off-diagonal numbers when handling multi-point constraints. So direct solver would be a more stable option for solving dynamic equations.

Chapter 6

Performance Study on Parallel Direct/Iterative Solving in SFSI

The motivation of this report is to introduce a robust and efficient parallel equation solver into our parallel finite element analysis framework. Aside from sparsity, which has been well known as the result of compact support that is inherent with finite element method, there exist some other special considerations that make the equation solving in finite element simulation a more involved problem.

In nonlinear finite element simulations, handling of constraints significantly affects the condition number of assembled equation systems. In SFSI simulations, multiple-point constraint is necessary to enforce the connection between soil and pile elements. In this research, penalty handler has been adopted to impose multiple point constraints on the assembled equation systems. Transformation and Lagrange multipliers are among those popular methods as well (Belytschko et al., 2000; Cook et al., 2002). The method of Lagrange multipliers adds extra constraints to the system and the resulted coefficient matrix will lose symmetric positive definiteness. Transformation is favorable especially in the sense that it reduces the order of the equation systems by condensing out slave/constrained DOFs. But the transformation is the most difficult to code and the situation of one single master/retained node with multiple slave/constrained nodes further complicates the problem.

Penalty method is chosen in this research due to the fact that it well preserves the symmetric positive definiteness of the system if the nice property is observed. Another consideration comes from the easiness with which the penalty methods can handle the single master/retained multiple slave/constrained situations. This is proven to be extremely valuable when data redistribution is required in adaptive parallel processing because the DOF_Graph object can be clearly tracked during partition and repartition phases.

The incapability of handling constraints accurately has been long known as the weakness of penalty method. The choice of the key penalty number seems arbitrary and largely depends on experience. The dilemma is with larger penalty number, the system can handle constraints more accurately while the coefficient matrix can become very ill-conditioned. This can lead to serious convergence problem for

iterative solvers.

The majority of coefficient matrices resulted from finite element analysis are inherently symmetric positive definite, for which lots of numerical algorithms have been proposed and solving SPD, symmetric or closely symmetric systems has been relatively maturer than more common unsymmetric cases. Unfortunately, in geotechnical finite element simulations, unassociated constitutive models lead to unsymmetric stiffness matrices (Jeremić, 2004b). More general parallel solvers must be coded to solve the problem.

In this section, both iterative and direct solvers are coded using the consistent PETSc interface (Balay et al., 2001, 2004, 1997). Popular direct solvers for general unsymmetric systems such as MUMPS, SPOOLES, SuperLU, PLAPACK have been introduced and performance study has been carried out to investigate the efficiency of different solvers on large scale SFSI simulations with penalty-handled unsymmetric equation systems. GMRES is always the first choice of iterative method when general unsymmetric systems are concerned. Preconditioning techniques have been thoroughly studied in this research to explore possible advantage of preconditioned iterative solver over direct solving. Jacobi, incomplete LU decomposition and approximate inverse preconditioners represent the most popular choices for Krylov methods and they are chosen in this performance survey.

All numerical algorithms have been implemented through interface of PETSc, which provides a consistent platform on which implementation issues can be avoided to expose individual algorithmic performance.

6.1 Parallel Sparse Direct Equation Solvers

The methods that we consider for the solution of sparse linear equations can be grouped into four main categories: general techniques, frontal methods, multifrontal approaches and supernodal algorithms (Dongarra et al., 1996).

6.1.1 General Techniques – SPOOLES

The so-called general approach can be viewed as parallel versions of sparse LU decomposition. Special cares must be taken to handle the sparse data structures. Sparsity ordering is crucial in parallel sparse equation solving in order to reduce fill-in and discover large-grain parallelism (Demmel et al., 1993).

Freely available package SPOOLES provides minimum degree (multiple external minimum degree (Liu, 1985)), generalized nested dissection and multisection ordering schemes for matrix sparsity ordering. Fundamental supernode tree built on top of vertex elimination tree is used to explore granularity in parallel (Ashcraft, 1999; Ashcraft et al., 1999).

6.1.2 Frontal and Multifrontal Methods – MUMPS

Frontal methods have their origins in the solution of finite element problems from structural analysis. The usual way to describe the frontal method is to view its application to finite element problems where the

matrix A is expected as a sum of contributions from the elements of a finite element discretization (Dongarra et al., 1996). That is,

$$A = \sum_{l=1}^{m} A^{[l]}, \tag{6.1}$$

where $A^{[l]}$ is nonzero only in those rows and columns that correspond to variables in the lth element. If a_{ij} and $a_{ij}^{[l]}$ denote the (i,j)th entry of A and $A^{[l]}$, respectively, the basic assembly operation when forming A is of the form

$$a_{ij} \Leftarrow a_{ij} + a_{ij}^{[l]}. \tag{6.2}$$

It is evident that the basic operation in Gaussian elimination

$$a_{ij} \Leftarrow a_{ij} + a_{ip}[a_{pp}]^{-1}a_{pj}. \tag{6.3}$$

may be performed as soon as all the the terms in the triple product 6.3 are fully summed (that is, are involved in no more sums of the form 6.2). The assembly and Gaussian elimination processes can therefore be interleaved and the matrix A is never assembled explicitly. This allows all intermediate working to be performed in a dense matrix, termed *frontal matrix*, whose rows and columns correspond to variables that have not yet been eliminated but occur in at least one of the elements that have been assembled.

For general problems other than finite element, the rows of A (equations) are added into the frontal matrix one at a time. A variable is regarded as fully summed whenever the equation in which it last appears is assembled. The frontal matrix will, in this case, be rectangular.

The idea of multifrontal method is to couple a sparsity ordering with the efficiency of a frontal matrix kernel so allowing good exploitation of high performance computers. The basic approach is to develop separate fronts simultaneously which can be chosen using a sparsity preserving ordering such as minimum degree.

Elimination tree, again is the most important notion in the factorization process and also utilized to discover the potential of parallelism. An elimination tree defines the a precedence order within the factorization. The factorization commences at the leaves of of the tree and data is passed towards the root along the edges in the tree. To complete the work associated with a node, all the data must have been obtained from the children of the node, otherwise work at different nodes is independent.

Freely available package MUMPS (*MUltifrontal Massively Parallel sparse direct Solver*) has been used in this research to investigate the performance of multifrontal methods (http://graal.ens-lyon.fr/MUMPS/, 2006).

MUMPS is a package for solving systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric. MUMPS uses a multifrontal technique which is a direct method based on either the $\mathbf{LU}$ or the $\mathbf{LDL^T}$ factorization of the matrix. MUMPS exploits both parallelism arising from sparsity in the matrix $\mathbf{A}$ and from dense factorizations kernels.

The main features of the MUMPS package include the solution of the transposed system, input of the matrix in assembled format (distributed or centralized) or elemental format, error analysis, iterative refinement, scaling of the original matrix, and return of a Schur complement matrix. MUMPS offers several built-in ordering algorithms, a tight interface to some external ordering packages such as METIS and PORD, and the possibility for the user to input a given ordering. Finally, MUMPS is available in various arithmetics (real or complex, single or double precision).

The software is written in Fortran 90 although a C interface is available. The parallel version of MUMPS requires MPI for message passing and makes use of the BLAS, BLACS, and ScaLAPACK libraries. The sequential version only relies on BLAS.

MUMPS distributes the work tasks among the processors, but an identified processor (the host) is required to perform most of the analysis phase, to distribute the incoming matrix to the other processors (slaves) in the case where the matrix is centralized, and to collect the solution. The system $\mathbf{Ax} = \mathbf{b}$ is solved in three main steps:

1. **Analysis**. The host performs an ordering based on the symmetrized pattern $\mathbf{A} + \mathbf{A^T}$, and carries out symbolic factorization. A mapping of the multifrontal computational graph is then computed, and symbolic information is transferred from the host to the other processors. Using this information, the processors estimate the memory necessary for factorization and solution.

2. **Factorization**. The original matrix is first distributed to processors that will participate in the numerical factorization. The numerical factorization on each frontal matrix is conducted by a master processor (determined by the analysis phase) and one or more slave processors (determined dynamically). Each processor allocates an array for contribution blocks and factors; the factors must be kept for the solution phase.

3. **Solution**. The right-hand side $\mathbf{b}$ is broadcast from the host to the other processors. These processors compute the solution $\mathbf{x}$ using the (distributed) factors computed during Step 2, and the solution is either assembled on the host or kept distributed on the processors.

Each of these phases can be called separately and several instances of MUMPS can be handled simultaneously. MUMPS allows the host processor to participate in computations during the factorization and solve phases, just like any other processor.

For both the symmetric and the unsymmetric algorithms used in the code, a fully asynchronous approach with dynamic scheduling of the computational tasks has been chosen. Asynchronous communication is used to enable overlapping between communication and computation. Dynamic scheduling was initially chosen to accommodate numerical pivoting in the factorization. The other important reason for this choice was that, with dynamic scheduling, the algorithm can adapt itself at execution time to remap work and data to more appropriate processors. In fact, the main features of static and dynamic approaches have been combined and the estimation obtained during the analysis to map some of the main computational tasks

has been used; the other tasks are dynamically scheduled at execution time. The main data structures (the original matrix and the factors) are similarly partially mapped according to the analysis phase.

6.1.3 Supernodal Algorithm – SuperLU

The *left-looking* or column Cholesky algorithm can be implemented for sparse system and can be blocked by using a supernodal formulation. The idea of a supernode is to group together columns with the same nonzero structure, so they can be treated as a dense matrix for storage and computation. Supernodes were originally used for (symmetric) sparse Cholesky factorization (Demmel et al., 1999a). In the factorization $\mathbf{A} = \mathbf{LL^T}$ (or $\mathbf{A} = \mathbf{LDL^T}$), a supernode is a range $(r : s)$ of columns of $\mathbf{L}$ with the same nonzero structure below the diagonal; that is, $\mathbf{L}(r : s; r : s)$ is full lower triangular and every row of $\mathbf{L}(r : s; r : s)$ is either full or zero.

Then in left-looking Cholesky algorithm, all the updates from columns of a supernode are summed into a dense vector before the sparse update is performed. This reduces indirect addressing and allows the inner loops to be unrolled. In effect, a sequence of *col-col* updates is replaced by a supernode-column (*sup-col*) update. The *sup-col* update can be implemented using a call to a standard dense Level 2 BLAS matrix-vector multiplication kernel. This idea can be further extended to supernode-supernode (*sup-sup*) updates, which can be implemented using a Level 3 BLAS dense matrix-matrix kernel. This can reduce memory traffic by an order of magnitude, because a supernode in the cache can participate in multiple column updates (Demmel et al., 1999a). It has been reported in (Ng and Peyton, 1993) that a sparse Cholesky algorithm based on *sup-sup* updates typically runs 2.5 to 4.5 times as fast as a *col-col* algorithm. Indeed, supernodes have become a standard tool in sparse Cholesky factorization.

To sum up, supernodes as the source of updates help because of the following (Demmel et al., 1999a):

1. The inner loop (over rows) has no indirect addressing. (Sparse Level 1 BLAS is replaced by dense Level 1 BLAS.)

2. The outer loop (over columns in the supernode) can be unrolled to save memory references. (Level 1 BLAS is replaced by Level 2 BLAS.)
   Supernodes as the destination of updates help because of the following:

3. Elements of the source supernode can be reused in multiple columns of the destination supernode to reduce cache misses. (Level 2 BLAS is replaced by Level 3 BLAS.)

Supernodes in sparse Cholesky can be determined during symbolic factorization, before the numeric factorization begins. However, in sparse $\mathbf{LU}$, the nonzero structure cannot be predicted before numeric factorization, so supernodes must be defined dynamically. Furthermore, since the factors $\mathbf{L}$ and $\mathbf{U}$ are no longer transposes of each other, the definition of a supernode must be generalized.

Freely available package SuperLU proposed a couple of ways to generalize the symmetric definition of supernodes to unsymmetric factorization (Demmel et al., 1999a). It is now not possible to use Level

3 BLAS efficiently for unsymmetric systems. The implementation in SuperLU performs a dense matrix multiplication of a block of vectors and, although these can not be written as another dense matrix, it has been shown that this Level 2.5 BLAS has most of the performance characteristics of Level 3 BLAS since the repeated use of the same dense matrix allows good use of cache and memory hierarchy.

There are three versions of libraries collectively referred as SuperLU (Demmel et al., 2003),

- **Sequential SuperLU** is designed for sequential processors with one or more layers of memory hierarchy (caches).
- **Multithreaded SuperLU** (SuperLU_MT) is designed for shared memory multiprocessors (SMPs), and can effectively use up to 16 or 32 parallel processors on sufficiently large matrices in order to speed up the computation (Demmel et al., 1999b).
- **Distributed SuperLU** SuperLU_DIST is designed for distributed memory parallel processors, using MPI for interprocess communication. It can effectively use hundreds of parallel processors on sufficiently large matrices (Li and Demmel, 2003).

Parallelizing sparse direct solver for unsymmetric systems is more complicated than parallel sparse Cholesky case. The advantage of sparse Cholesky over the unsymmetric case is that pivots can be chosen in any order from the main diagonal while guaranteeing stability. This lets us perform pivot choice before numerical factorization begins, in order to minimize fill-in, maximize parallelism. precompute the nonzero structure of the Cholesky factor, and optimize the (2D) distributed data structures and communication pattern (Li and Demmel, 2003).

In contrast, for unsymmetric or indefinite systems, distributed memory codes can be much more complicated for at least two reasons. First and foremost, some kind of numerical pivoting is necessary for stability. Classical partial pivoting or the sparse variant of threshold pivoting typically cause the fill-ins and workload to be generated dynamically during factorization. Therefore, we must either design dynamic data structures and algorithms to accommodate these fill-ins, or else use static data structures which can grossly overestimate the true fill-in. The second complication is the need to handle two factored matrices $\mathbf{L}$ and $\mathbf{U}$, which are structurally different yet closely related to each other in the filled pattern. Unlike the Cholesky factor whose minimum graph representation is a tree (elimination tree), the minimum graph representations of the $\mathbf{L}$ and $\mathbf{U}$ factors are directed acyclic graphs (elimination DAGs).

In SuperLU_DIST, a static pivoting approach, called GESP (Gaussian Elimination with Static Pivoting) (Li and Demmel, 1998) is used. In order to parallelize the GESP algorithm, a 2D block-cyclic mapping of a sparse matrix to the processors is used. An efficient pipelined algorithm is also designed to perform parallel factorization. With GESP, the parallel algorithm and code are much simpler than dynamic pivoting.

The main algorithmic features of SuperLU_DIST solver are summarized as follows (Li and Demmel, 2003):

- supernodal fan-out (right-looking) based on elimination DAGs,

- static pivoting with possible half-precision perturbations on the diagonal,
- use of an iterative algorithm using the LU factors as a preconditioner, in order to guarantee stability,
- static 2D irregular block-cyclic mapping using supernodal structure, and
- loosely synchronous scheduling with pipelining.

In particular, static pivoting can be performed before numerical numerical factorization, allowing us to use all the techniques in good sparse Cholesky codes: choice of a (symmetric) permutation to minimize fill-in and maximize parallelism, precomputation of the fill pattern and optimization of 2D distributed data structures and communication patterns. Users are referred to Li and Demmel (2003) for algorithm details.

6.2 Performance Study on SFSI Systems

In this section, performance study on popular parallel direct and iterative solvers has been conducted. The purpose is to provide some guidelines on appropriate use of different solvers with the parallel finite simulation framework. Matrix systems from SFSI analysis are used as test cases. The performance investigation uses IA64 Intel-based cluster at SDSC.

6.2.1 Equation System

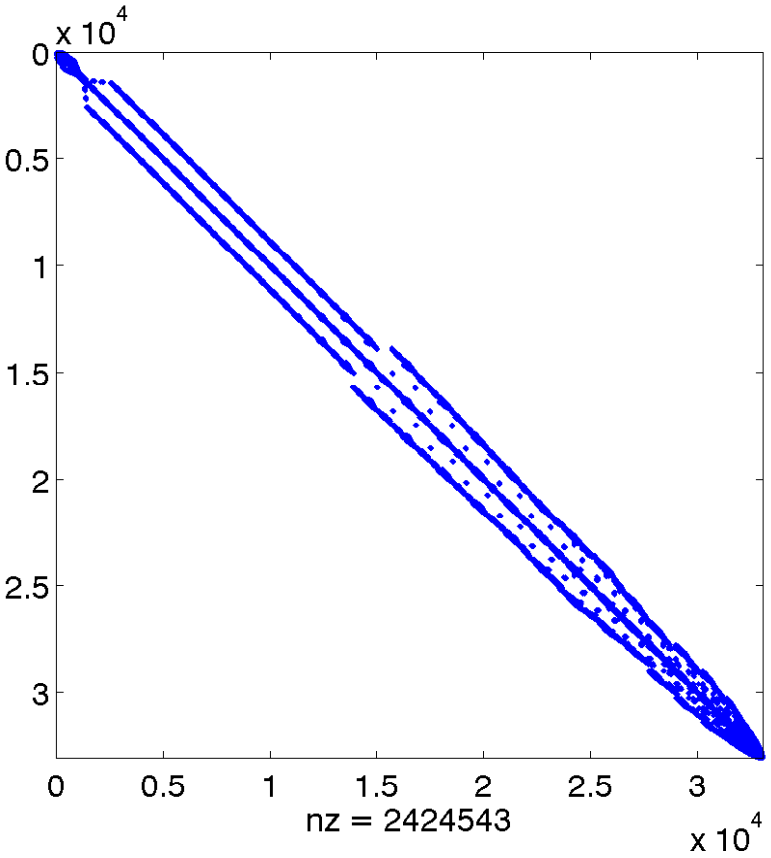


Figure 6.1: Matrices $N = 33081$ (Continuum FEM)

6.2.2 Performance Results

Table 6.1: Performance Study on SFSI Systems (N=33081)

Direct Solvers		
Solvers	Num of CPUs	Time (s)
MUMPS	4	6.0312
	8	7.0534
	16	5.3472
SuperLU_DIST	4	20.358
	8	13.803
	16	13.755
SPOOLES	4	10.696
	8	7.5338
	16	6.2448
Iterative Solvers (GMRES)		
Preconditioner	Num of CPUs	Time (s)
Jacobi	16	96.441
Parallel ILU(0)	4	277.49
	8	276.07
	16	135.78

6.3 Conclusion

This chapter presents the parallel solvers implemented in parallel finite element framework. Table 6.1, draws several conclusions about appropriate use of solvers:

- Direct solvers outperforms the iterative solver significantly for general cases. It is worthwhile to note that nonsymmetric solvers are used here due to their generality. For special cases such as SPD system, preconditioned CG will show much better performance.
- The Conjugate Gradient method applies only to Symmetric Positive Definite (SPD) system. This puts restriction on the material models we can use in our simulations. Generally speaking, elastic material will yield a SPD stiffness matrix. Plastic material with associative flow rule also satisfies this category. Plastic material with non-associative flow rule has non-symmetric element stiffness matrix and so will be the global coefficient matrix of the equation system.

- Another category of matrix that deserves attention is the stiffness matrix from softening materials, which possesses at least one negative eigenvalue so the SPD property will be broken. For advanced geo-materials subject to complicated loadings, as the material develops nonlinearity, the condition of stiffness matrix might vary greatly from SPD (elastic phase), to singular (elastic-perfectly-plastic), and non-symmetric non-positive-definite (elastic-non-associative-plastic-softening) cases. This poses another challenge when one tries to use iterative solver for production runs. The unpredictability of stiffness matrix will disable the application of powerful solvers such as Conjugate Gradient for iterative case and Cholesky for direct case.

- The reason why iterative solver exhibits poor efficiency is partly due to the problem size. We can also see from the Table 6.1 that parallel direct solver is not scalable in general. Iterative solver, on the other hand, is more scalable and it is safe to project that when the size of matrix increases, iterative solver has the advantage from the memory requirement point of view.

Parallel equation solving itself is a complicated topic in numerical computing community. In this report , the main purpose is to introduce a robust and generally efficient parallel solver for finite element simulations. So in this sense, parallel direct solvers such as MUMPS and SPOOLES are recommended.

Appendix C presents Tcl commands to invoke those solvers in parallel simulation runs.

Part III

Bibliography and Appendices

Bibliography

G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1984.

Graham Charles Archer. *Object Oriented Finite Analysis*. PhD thesis, University of California, Berkeley, 1996.

Cleve Ashcraft. Ordering sparse matrices and transforming front trees. Technical report, Boeing Shared Service Group, 1999.

Cleve Ashcraft, Daniel Pierce, David K. Wah, and Jason Wu. *The Reference Manual for SPOOLES, Release 2.2: An Object Oriented Software Library for Solving Sparse Linear Systems of Equations*. Boeing Shared Services Group, Seattle, Washington, 1999.

Zhaojun Bai. Class notes on large scale scientific computing. ECS 231, Department of Computer Science, UC Davis.

Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. `http://www.mcs.anl.gov/petsc`.

Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

H. Bavestrello, P. Avery, and C. Farhat. Incorporation of linear multipoint constraints in domain-decomposition-based iterative solvers - Part II: Blending FETI-DP and mortar methods and assembling floating substructures. *Computer Methods in Applied Mechanics and Engineering*, 196(8):1347–1368, January 2007.

Ted Belytschko, Wing Kam Liu, and Brian Moran. *Nonlinear finite elements for continua and structures*. John Wiley & Sons, 2000.

Michele Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182:418–477, 2002.

Michele Benzi and Miroslav Tůma. A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications*, 10:385–400, 2003.

Michele Benzi, Carl D. Meyer, and Miroslav Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17(5):1135–1149, 1996.

Michele Benzi, Jane K. Cullum, and Miroslav Tůma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 22(4):1318–1332, 2000.

Wendy Boggs and Michael Boggs. *Mastering UML with Rational Rose 2002*. Sybex, Alameda CA 94501, 2002.

Matthias Bollhöfer and Yousef Saad. On the relations between ILUs and factored approximate inverses. Technical Report UMSI-2001-67, Department of Computer Science and Engineering, University of Minnesota, 2001.

A Cardona, I. Klapka, and M. Geradin. Design of a new finite element programming environment. *Engineering Computations*, 11:365–381, 1994.

W. T. Carter, T. L. Sham, and K. H. Law. A Parallel Finite Element Method and It's Prototype Implementation on a Hypercube. *Computers and Structures*, 31(6):921–934, 1989.

R. Chudoba and Z. Bittnar. Explicit Finite Element Computation: An Object-Oriented Approach. In *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.

Robert D. Cook, David S. Malkus, Michael E. Plesha, and Robert J. Witt. *Concepts and Applications of Finite Element Analysis*. John Wiley & Sons, 2002.

M. A. Crisfield. *Non-linear Finite Element Analysis of Solids and Structures*. John Wiley & Sons, 1997.

Luis Crivelli and Charbel Farhat. Implicit transient finite element structural computations on mimd systems: Feti v.s. direct solvers. In *34th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, La Jolla, CA, USA, April 19-22 1993.

James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel numerical linear algebra. Technical report, LAPACK Working Note 60, UT CS–93–192, 1993.

James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999a.

James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999b.

James W. Demmel, John R. Gilbert, and Xiaoye S. Li. *SuperLU Users' Guide*, 2003.

Clark R. Dohrmann. A preconditioner for substructuring based on constrained energy minimization. *SIAM Journal of Scientific Computing*, 25(1):246–258, September/October 2003.

Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Source Book of Parallel Computing*. Morgan Kaufmann Publishers, 2003.

Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High Performance Computers*. Prentice Hall, New Jersey, 1996.

Y. Dubois-Pelerin and T. Zimmermann. Object-Oriented Finite Element Programming: III. An Efficient Implementation in C++. *Computer Methods in Applied Mechanics and Engineering*, 108(1-2):165–183, 1993.

Y. Dubois-Pelerin, T. Zimmermann, and P. Bomme. Object-Oriented Finite Element Programming: II. A Prototype Program in Smalltalk. *Computer Methods in Applied Mechanics and Engineering*, 98(3): 361–397, 1992.

C. Farhat and F. X. Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32(6):1205–1227, 1991a.

C. Farhat, M. Lesoinne, P. LeTallec, K. Pierson, and D. Rixen. FETI-DP: a dual-primal unified FETI method - Part I: a faster alternative to the two-level FETI method. *International Journal for Numerical Methods in Engineering*, 50(7):1523–1544, 2001.

Charbel Farhat. *Muliprocessors in Computational Mechanics*. PhD thesis, University of California, Berkeley, 1987.

Charbel Farhat. Saddle-point principle domain decomposition method for the solution of solid mechanics problems. In *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, Norfolk, VA, USA, May 6-8 1991.

Charbel Farhat and Luis Crivelli. A transient FETI methodology for large–scale parallel implicit comutations in structural mechanics. *International Journal for Numerical Methods in Engineering*, 37:1945–1975, 1994.

Charbel Farhat and M. Geradin. Using a reduced number of lagrange multipliers for assembling parallel incomplete field finite element approximations. *Computer Methods in Applied Mechanics and Engineering*, 97(3):333–354, June 1992.

Charbel Farhat and Francois-Xavier Roux. Method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32(6):1205–1227, October 1991b.

Charbel Farhat, Michael Lesoinne, and Kendall Pierson. A scalable dual-primal domain decomposition method. *Numerical Linear Algebra with Applications*, 7(7–8):687–714, 2000.

Carlos A. Felippa. Class notes on nonlinear finite element methods. Department of Aerospace Engineering Sciences, University of Colorado at Boulder, 2004.

Bruce W. R. Forde, Ricardo O. Foschi, and Siegfried F. Steimer. Object – oriented finite element analysis. *Computers and Structures*, 34(3):355–374, 1990.

L. Fox, H. D. Huskey, and J. H. Wilkinson. Notes on the solution of algebraic linear simultaneous equations. *Quarterly Journal of Mechanics and Applied Mathematics*, 1:149–173, 1948.

R. E. Fulton and P. S. Su. Parallel substructure approach for massively parallel computers. *Computers in Engineering*, 2:75–82, 1992.

J. F. Hajjar and J. F. Abel. Parallel processing for transient nonlinear strcutraul dynamics of three–dimensional framed structures using domain decomposition. *Computers & Structures*, 30(6):1237–1254, 1988.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Stanford, CA, August 1973.

http://graal.ens-lyon.fr/MUMPS/. *MUltifrontal Massively Parallel Solver (MUMPS Version 4.6.2) Users' Guide*, 2006.

http://www.sdsc.edu/user_services/datastar/. Ibm datastar user guide. San Diego Supercomputer Center at UCSD.

Feng-Nan Hwang and Xiao-Chuan Cai. A class of parallel two-level nonlinear Schwarz preconditioned inexact Newton algorithms . *Computer Methods in Applied Mechanics and Engineering*, 196(8):1603–1611, January 2007.

Boris Jeremić. Lecture notes on computational geomechanics: Inelastic finite elements for pressure sensitive materials. Technical Report UCD-CompGeoMech–01–2004, University of California, Davis, 2004a. available online: `http://sokocalo.engr.ucdavis.edu/~jeremic/CG/LN.pdf`.

Boris Jeremić. Lectures notes on computational geomechanics: Inelastic finite elements for pressure sensitive materials. April 2004b.

Boris Jeremić and Stein Sture. Tensor data objects in finite element programming. *International Journal for Numerical Methods in Engineering*, 41:113–126, 1998.

Boris Jeremić and Stein Sture. Implicit integrations in elastoplastic geotechnics. *Mechanics of Cohesive-Frictional Materials*, 2(2):165–183, 1997.

Boris Jeremić and Christos Xenophontos. Application of the *p*-version of the finite element method to elastoplasticity with localization of deformation. *Communications in Numerical Methods in Engineering*, 15(12):867–876, December 1999.

Boris Jeremić and Zhaohui Yang. Template elastic–plastic computations in geomechanics. *International Journal for Numerical and Analytical Methods in Geomechanics*, 26(14):1407–1427, 2002.

Boris Jeremić and Zhaohui Yang. Template elastic-plastic computations in geomechanics. *International Journal for Numerical and Analytical Methods in Geomechanics*, 26(14):1407–1427, December 2002.

George Karypis and Vipin Kumar. *METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, September 1998a.

George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*. Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, September 1998b.

George Karypis, Kirk Schloegel, and Vipin Kumar. *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library*. University of Minnesota.

George Karypis, Kirk Schloegel, and Vipin Kumar. *PARMETIS Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 3.1*. Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, August 2003.

George Kaypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

O. Klaas, M. Kreienmeyer, and E. Stein. Elastoplastic finite element anslysis on a MIMD parallel–computer. *Engineering Computations*, 11:347–355, 1994.

Petr Krysl and Zdeněk Bittnar. Parallel explicit finite element solid dynamics with domain decomposition and message passing: dual partitioning scalability. *Computers and Structures*, 79:345–360, 2001.

Jing Li and Olof B. Widlund. On the use of inexact subdomain solvers for BDDC algorithms . *Computer Methods in Applied Mechanics and Engineering*, 196(8):1415–1428, January 2007.

Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of SC98: High Performance Networking and Computing Conference*, Orlando, Florida, November 7–13 1998.

Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

Joseph W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, June 1985.

R. I. Mackie. Object-Oriented Methods - Finite Element Programming and Engineering Software Design. In *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.

J. Mandel and C. R. Dohrmann. Convergence of a balancing domain decomposition by constraints and energy minimization. *Numerical Linear Algebra with Applications*, 10(7):639–659, 2003.

T. Manteuffel. An incomplete factorization technique for posititive definite linear systems. *Mathematics of Computation*, 34:473–497, 1980.

Francis Thomas McKenna. *Object Oriented Finite Element Programming: Framework for Analysis, Algorithms and Parallel Computing*. PhD thesis, University of California, Berkeley, 1997.

Francis Thomas McKenna. *Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing*. PhD thesis, University of California, Berkeley, 1997.

J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31:148–162, 1977.

Ph. Menéntrey and Th. Zimmermann. Object–oriented non–linear finite element analysis: Application to J2 plasticity. *Computers and Structures*, 49(5):767–77, 1993.

G. R. Miller and M. D. Rucki. A Program Architecture for Interactive Nonlinear Dynamic Analysis of Structures. In *Computing in Civil and Building Engineering: Proceedings of the Fifth International Conference V_ICCCBE*, Anaheim, CA, June 7–9 1993.

Esmond Ng and Barry Peyton. Block sparse cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific and Statistical Computing*, 14(5):1034–1056, September 1993.

A. K. Noor, A. Kamel, and R. E. Fulton. Substructuring techniques – status and projections. *Computers & Structures*, 8:628–632, 1978.

W. Ostermann, W. Wunderlich, and H. Cramer. Object-Oriented Tools for the Development of User Interface for Interactive Teachware. In *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.

Luca F. Pavarino. BDDC and FETI-DP preconditioners for spectral element discretizations. *Computer Methods in Applied Mechanics and Engineering*, 196(8):1380–1388, January 2007.

R. M. V. Pidaparti and A. V. Hudli. Dynamic analysis of structures using object–oriented techniques. *Computers and Structures*, 49(1):149–156, 1993.

J. S. Przemieniecki. *Theory of Matrix Structural Analysis*. McGraw Hill, New York, 1986.

Daniel Rixena and Frédéric Magoulès. Domain decomposition methods: Recent advances and new challenges in engineering. *Computer Methods in Applied Mechanics and Engineering*, 196(8):1345–1346, January 2007.

Y. Robert. Regular incomplete factorizations of real positive definite matrices. *Linear Algebra and Its Applications*, 48:105–117, 1982.

T. J. Ross, L. R. Wagner, and G. F. Luger. Object-Oriented Programming for Scientific Codes. II: Examples in C++. *Journal of Computing in Civil Engineering*, 6(4):497–514, 1992.

M. D. Rucki and G. R. Miller. An Algorithmic Framework for Flexible Finite Element-Based Structural Modeling. *Computer Methods in Applied Mechanics and Engineering*, 136(3–4):363–384, 1996.

J. Rumbaugh, M. Blaha, W. Premerhani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, second edition, 2003.

Marcus Sarkis and Daniel B. Szyld. Optimal left and right additive Schwarz preconditioning for minimal residual methods with Euclidean and energy norms. *Computer Methods in Applied Mechanics and Engineering*, 196(8):1612–1621, January 2007.

R. Sause and J. Song. Object-Oriented Structural Analysis with Substructures. In *Computing in Civil Engineering: Proceedings of the First Conference held in Conjunction with with A/E/C Systems' 94*, Washington, D.C., June 20–22 1994.

Kirk Schloegel, George Kaypis, and Vipin Kumar. Graph partitioning for high performance scientific simulations. Technical report, Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, 1999.

Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. Technical report, Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, 2000.

O. O. Storaasli and P. Bergan. Nonlinear substructuring method for concurrent processing computers. *AIAA Journal*, 25(6):871–876, 1987.

B. H. V. Topping and A. I. Khan. *Parallel Finite Element Computations*. SAXE-COBURG, Dun Eaglais, Kippen, Stirling, FK8 3DY, Scotland, 1996.

S. Utku, R. Melosh, M. Islam, and M. Salama. On nonlinear finite element analysis in single– multi– and parallel processors. *Computers & Structures*, 15(1):39–47, 1982.

R. S. Varga, E. B. Saff, and V. Mehrmann. Incomplete factorizations of matrices and connections with H-matrices. *SIAM Journal on Numerical Analysis*, 17:787–793, 1980.

C. Warshaw. *Parallel JOSTLE User Guide*. University of Greenwich, London, 1998.

W. Zahlten, P. Demmert, and W. B. Kratzig. An Object-Oriented Approach to Physically Nonlinear Problems in Computational Mechanics. In *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.

Gordon W. Zeglinski, Ray S. Han, and Peter Aitchison. Object oriented matrix classes for use in a finite element code using C++. *International Journal for Numerical Methods in Engineering*, 37:3921–3937, 1994.

Thomas Zimmermann, Yves Dubois-Pèlerin, and Patricia Bomme. Object oriented finite element programming: I. governing principles. *Computer Methods in Applied Mechanics and Engineering*, 98:291–303, 1992.

Appendix A

Compilation of Parallel Program (PDD–based) on GNU/Linux Clusters

This report developed a comprehensive package for performing parallel finite element calculations on both nonlinear structural and geotechnical models. The analysis framework of sequential implementation of OpenSees has been used in part but major new additions of partitioning and repartitioning modules have been implemented using ParMETIS (version 3.1). The parallel equation solving kernel has been developed based on PETSc (version 2.3.1-p15). In order to successfully compile the parallel implementation developed in this report , these libraries must be correctly built. This chapter introduces how to compile all the necessary libraries for producing executables of the parallel code on local clusters. The default compilers are assumed to be GNU gcc/g++ and g77 (or gfortran for Fedora Core 5 and above). All the procedures described have been successfully verified on our Linux clusters GeoWulf.

A.1 MPICH

The implementation of this report used mpich-1.2.7 for facilitating inter-process communication. SMP-based cluster now becomes more and more popular as the dual-core technology matures. By default, MPI directs the messages from the source out to switch and then to the destination. This is not efficient for SMP machines with more than one CPUs on board (or more than one cores on the chip). MPI (mpich-1.2.7) has to be compiled with the *–with-comm=shared* flag on in order to turn on shared memory communication mechanism (basically shared memory approach).

This can greatly accelerated on-board communication.

A.1.1 SMP On-Board Communication Effective Benchmark

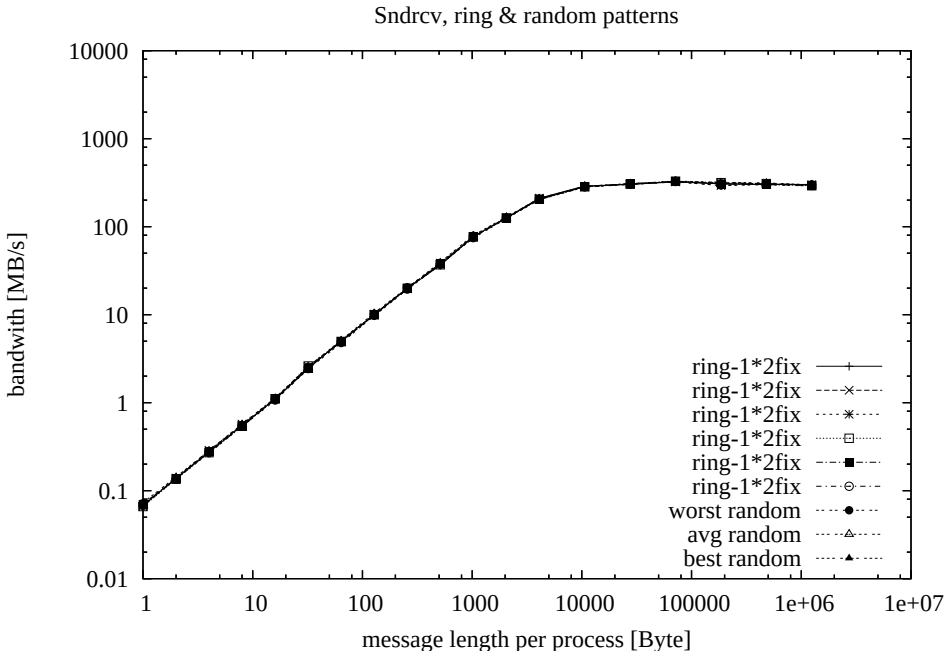
Effective Bandwith Benchmark (b_{eff}) Version 3.5

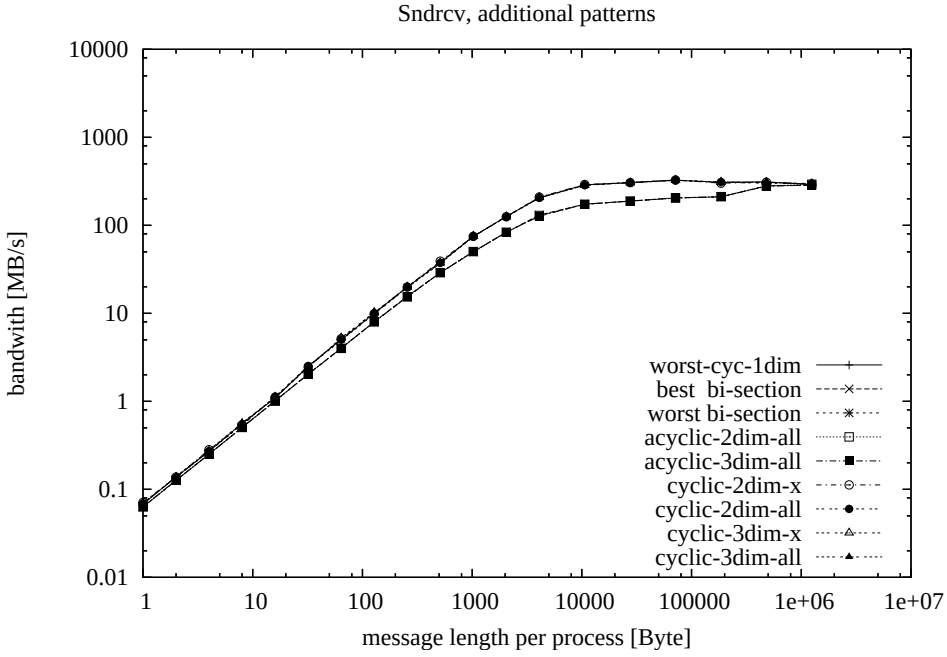
Linux geowulf 2.6.16.4 #3 SMP i686

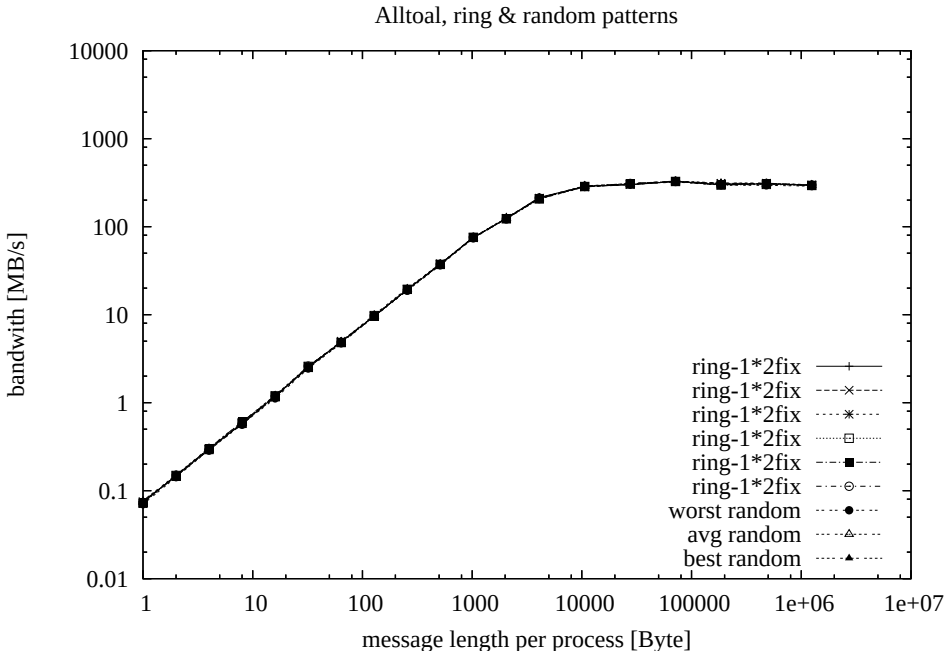
$\mathbf{b_{eff}}$ **= 278.434 MB/s**

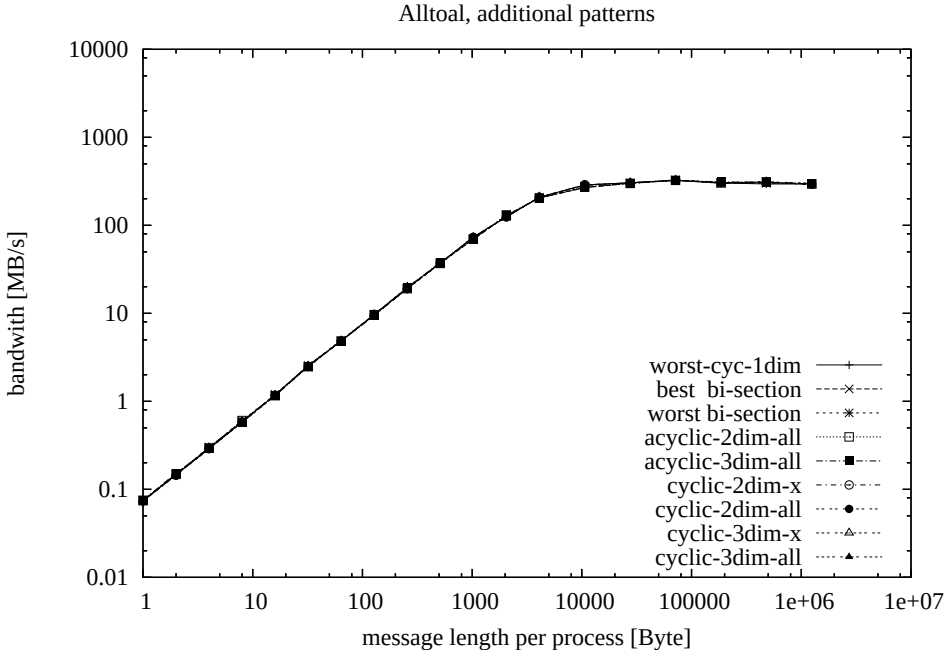
	number of processors	b_{eff} MByte/s	Lmax	b_{eff} at Lmax rings& random MByte/s	b_{eff} at Lmax rings only MByte/s
accumulated	2	278	8 MB	589	591
		Latency rings& random mircosec	Latency rings only microsec	Latency ping-pong microsec	ping-pong bandwidth MByte/s
accumulated		13.108	13.086	7.897	556

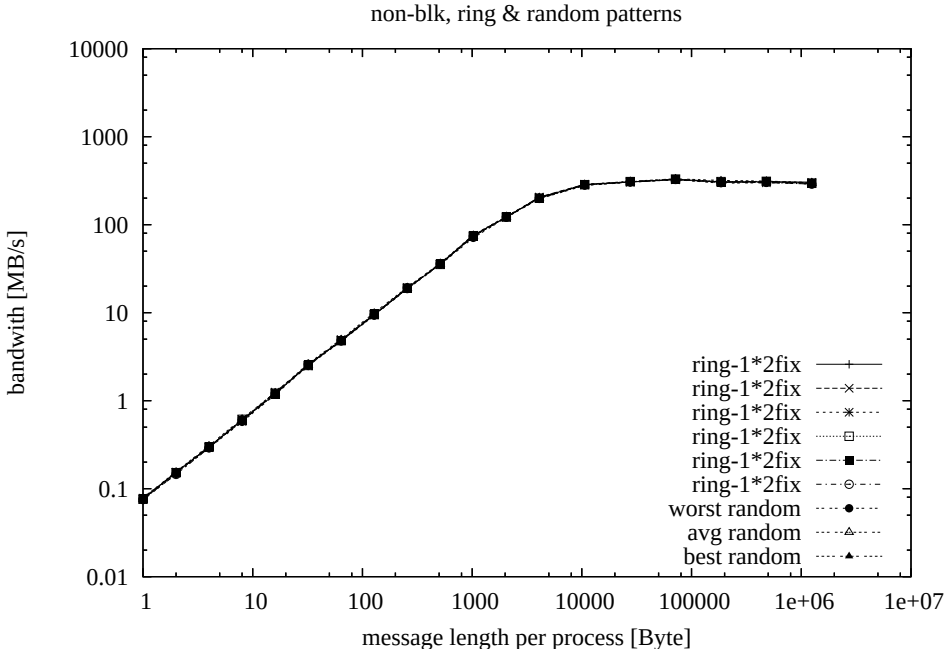
Ping-Pong result (only the processes with rank 0 and 1 in MPI_COMM_WORLD were used):
Latency: 7.897 microsec per message Bandwidth: 555.831 MB/s (with MB/s $= 10^6$ byte/s)

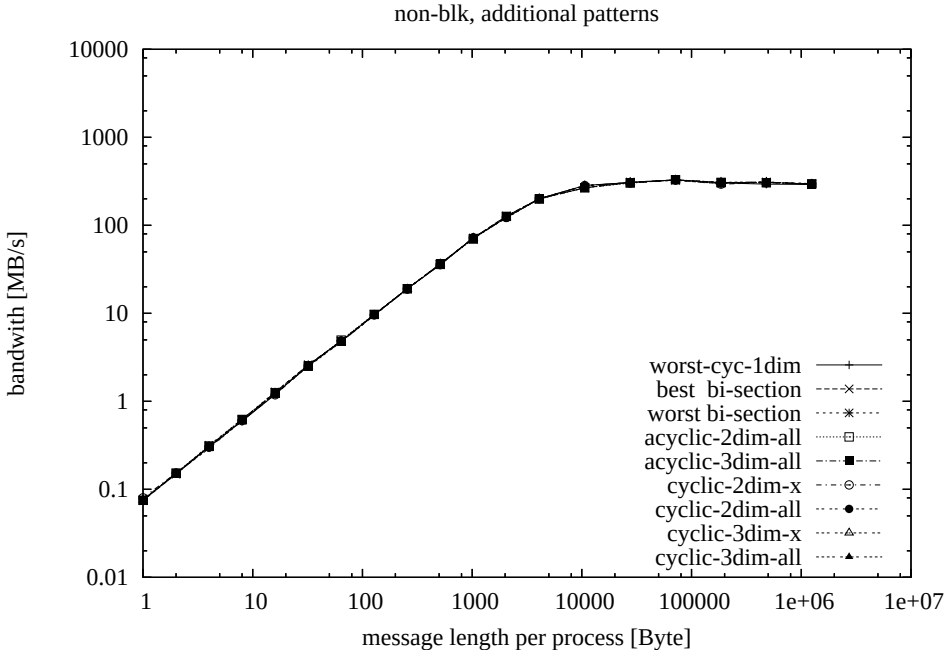

Sndrcv, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random

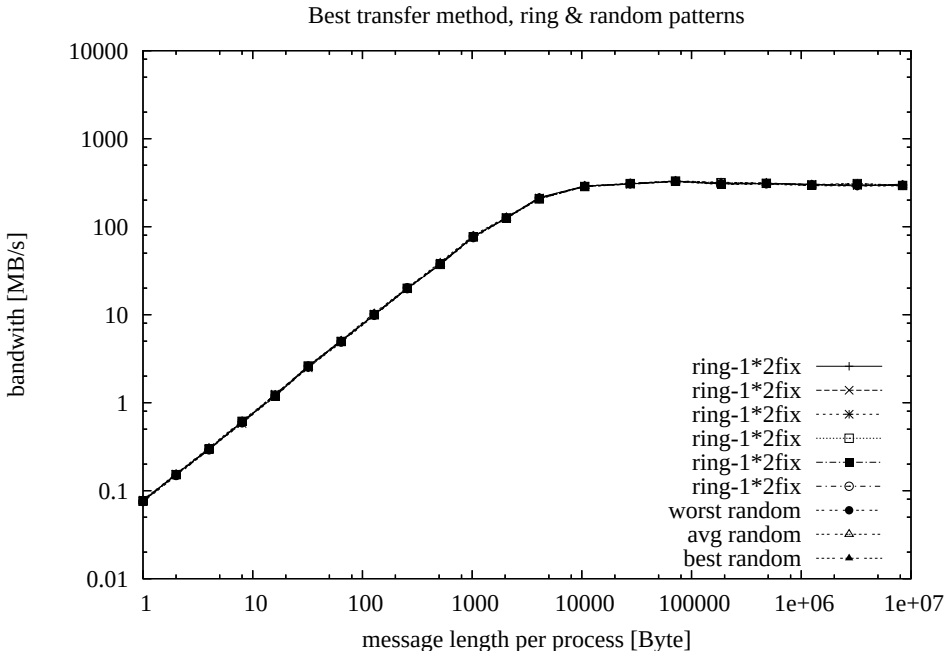

Sndrcv, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all

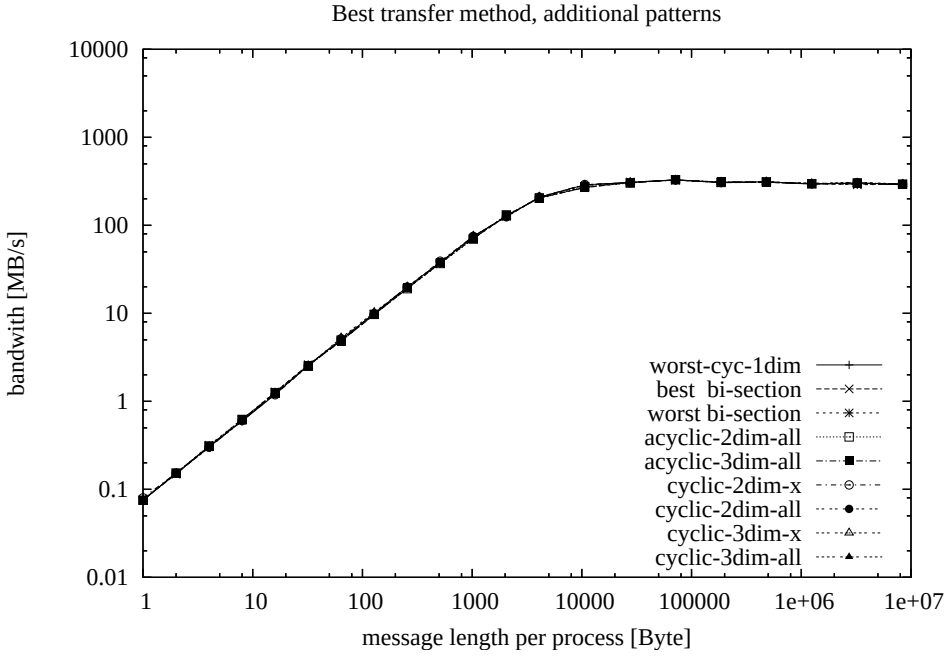

Alltoal, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random

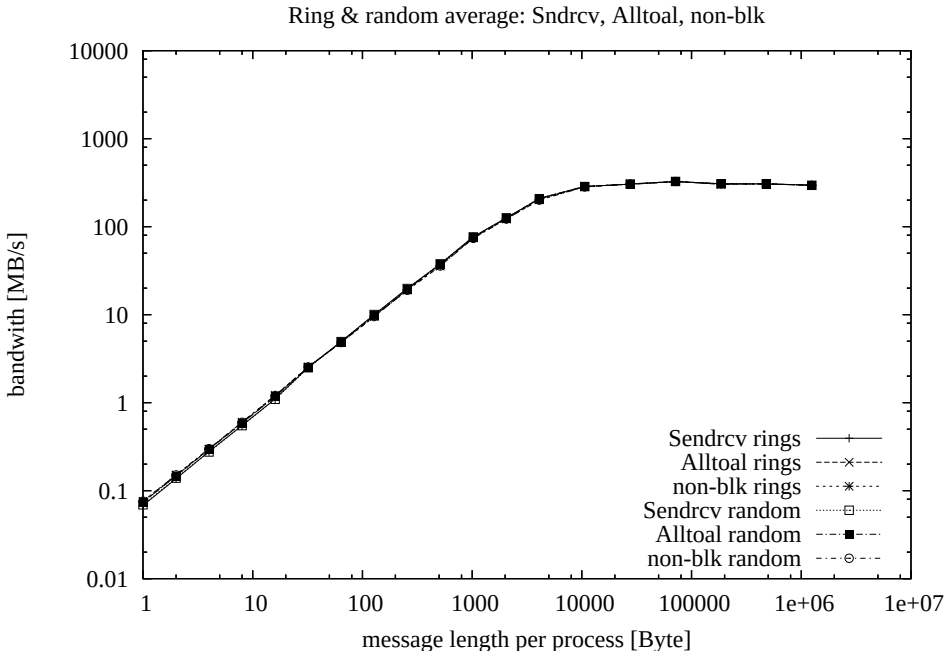

Alltoal, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all

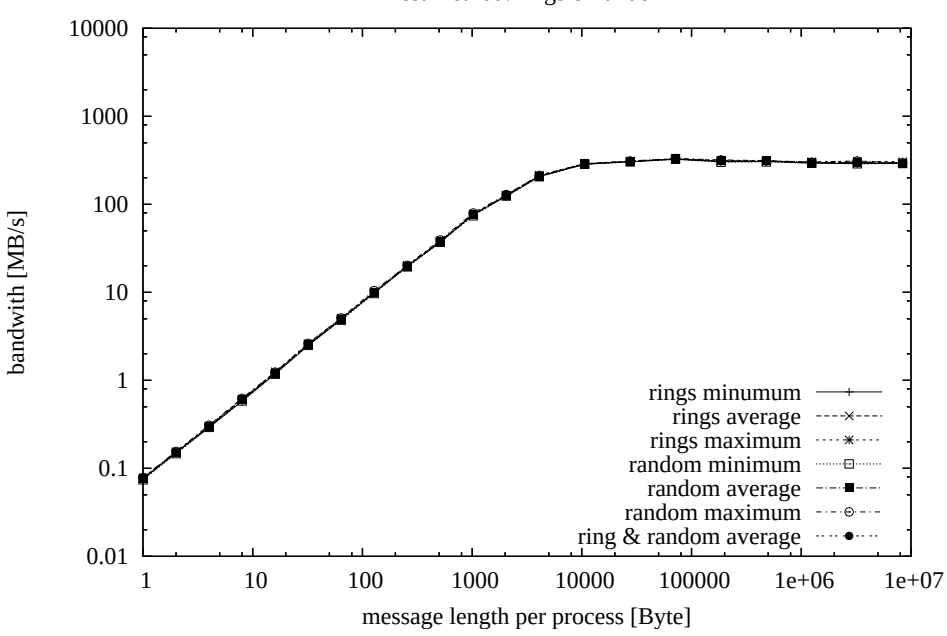

non-blk, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random


non-blk, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all


Best transfer method, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random


Best transfer method, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all


Ring & random average: Sndrcv, Alltoal, non-blk
bandwith [MB/s]
message length per process [Byte]
Sendrcv rings
Alltoal rings
non-blk rings
Sendrcv random
Alltoal random
non-blk random
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07


Best method: rings & random
bandwith [MB/s]
message length per process [Byte]
rings minumum
rings average
rings maximum
random minimum
random average
random maximum
ring & random average
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07

A.1.2 Cluster Inter-Switch Communication Effective Benchmark

Effective Bandwith Benchmark (b_{eff}) Version 3.5

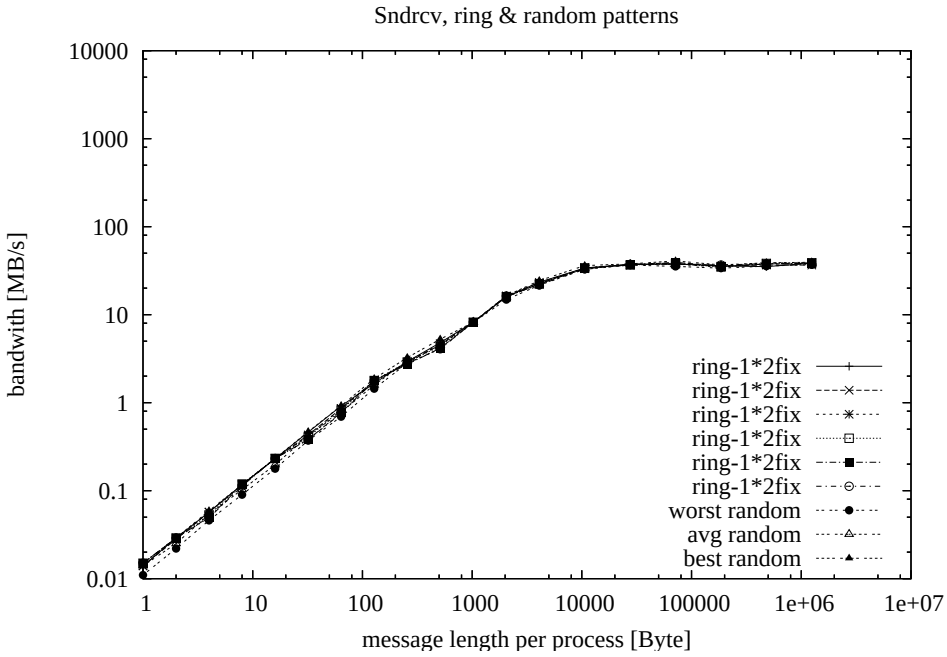
Linux geowulf to nodes 2.6.16.4 #3 i686

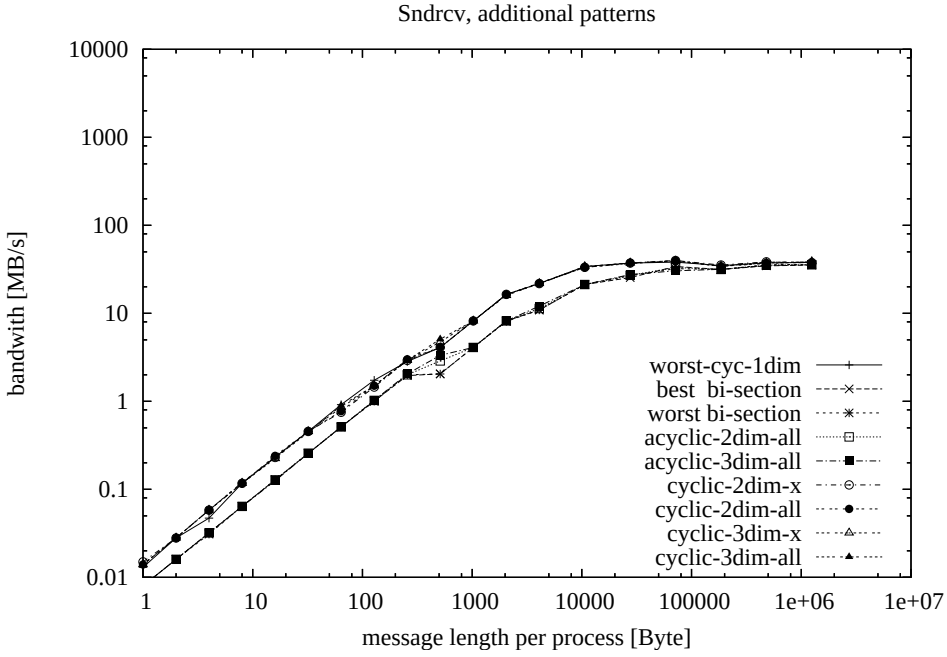
$\mathbf{b_{eff}}$ **= 34.021 MB/s**

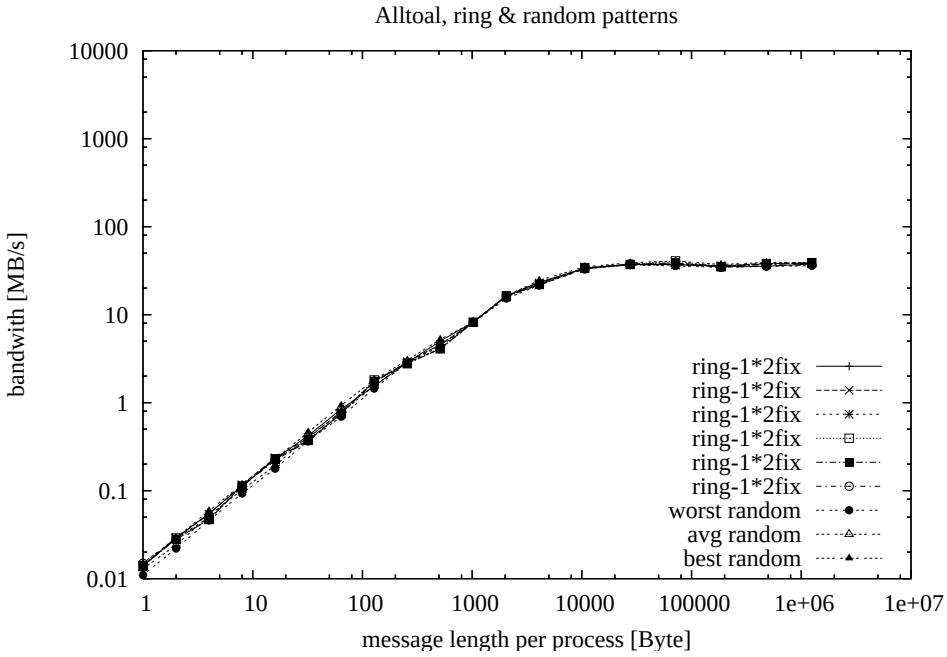
	number of processors	b_{eff} MByte/s	Lmax	b_{eff} at Lmax rings& random MByte/s	b_{eff} at Lmax rings only MByte/s
accumulated	2	34	8 MB	75	75
		Latency rings& random mircosec	Latency rings only microsec	Latency ping-pong microsec	ping-pong bandwidth MByte/s
accumulated		70.599	68.930	62.487	75

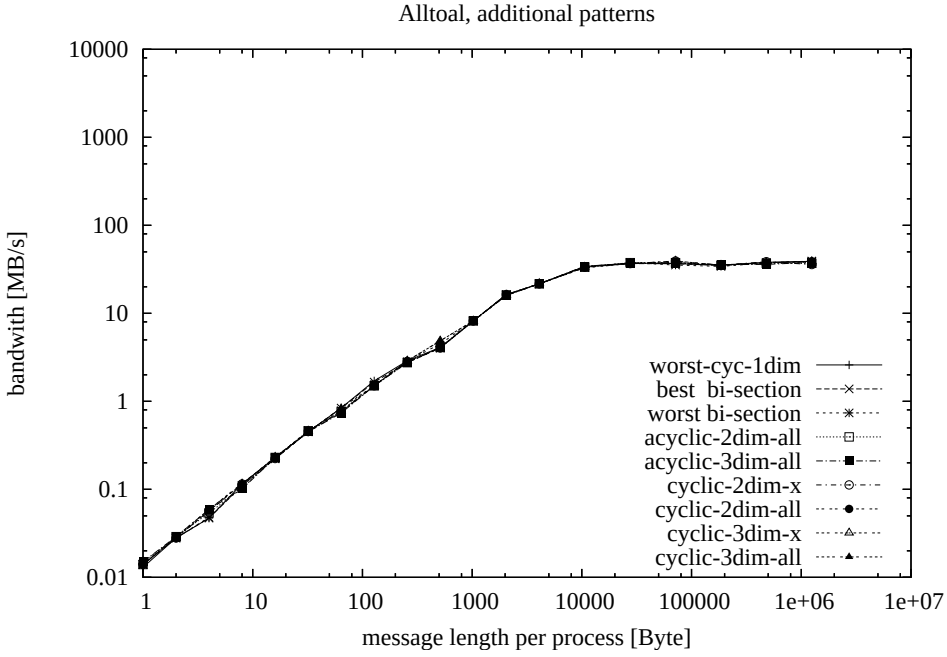
Ping-Pong result (only the processes with rank 0 and 1 in MPI_COMM_WORLD were used):

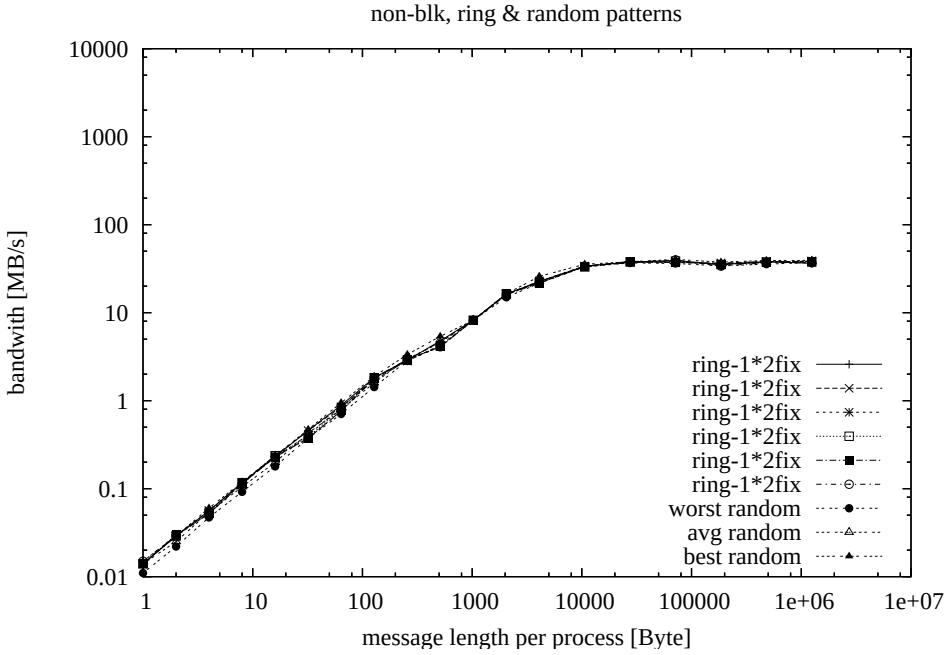
Latency: 62.487 microsec per message Bandwidth: 74.935 MB/s (with MB/s $= 10^6$ byte/s)

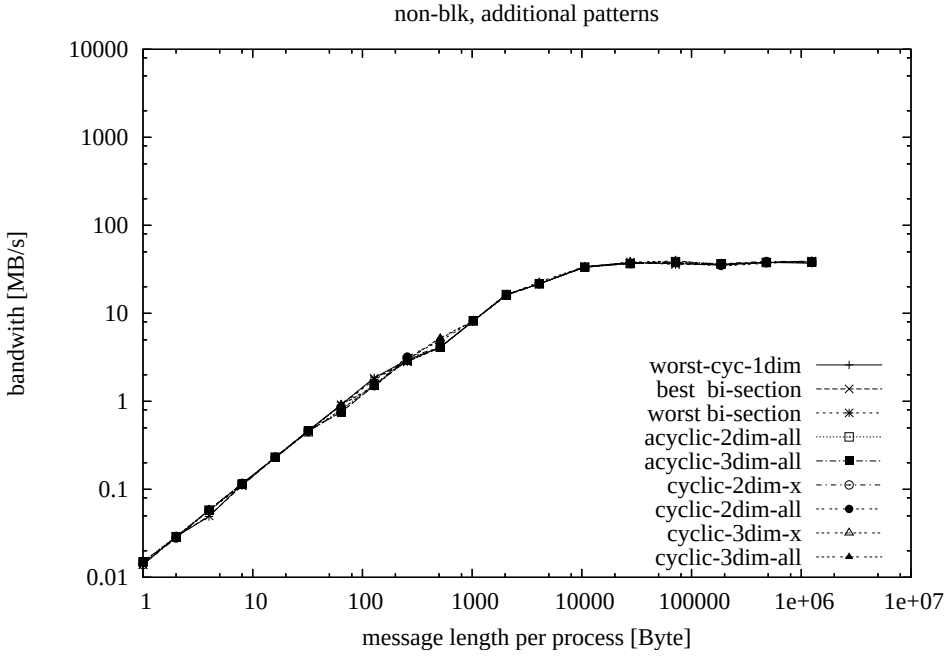

Sndrcv, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random

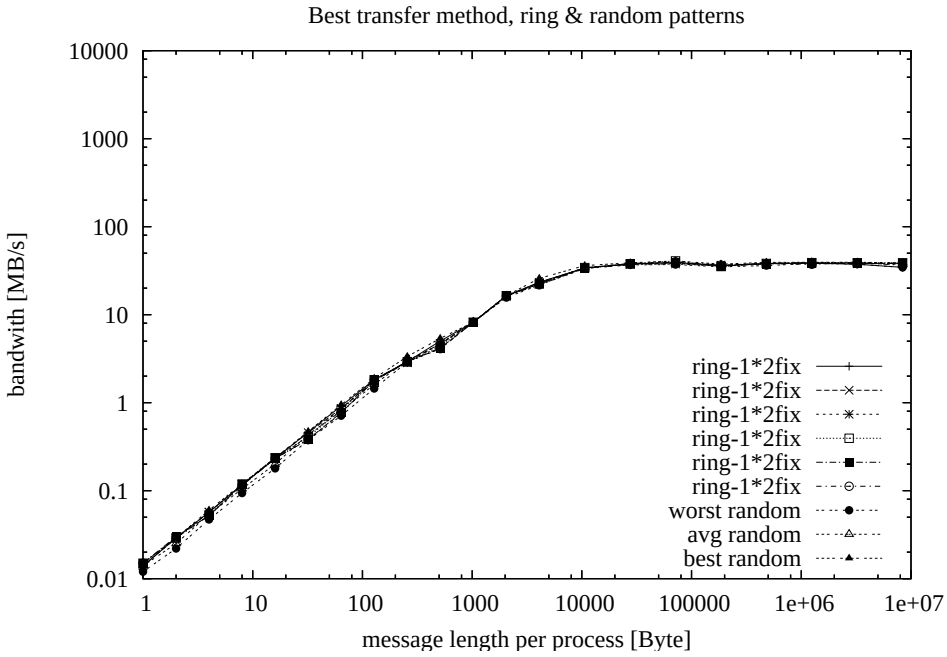

Sndrcv, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all

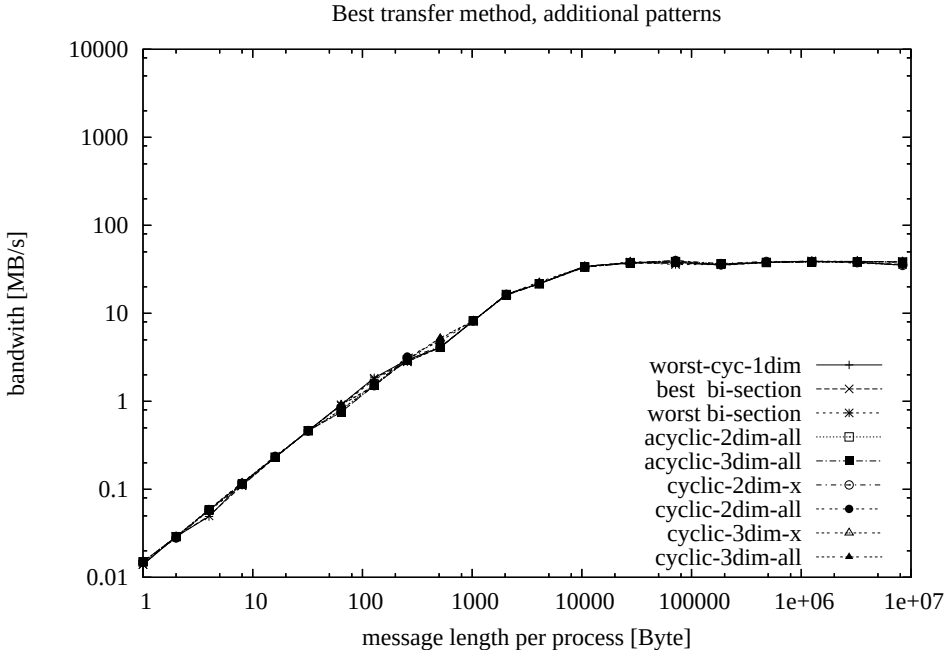

Alltoal, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random

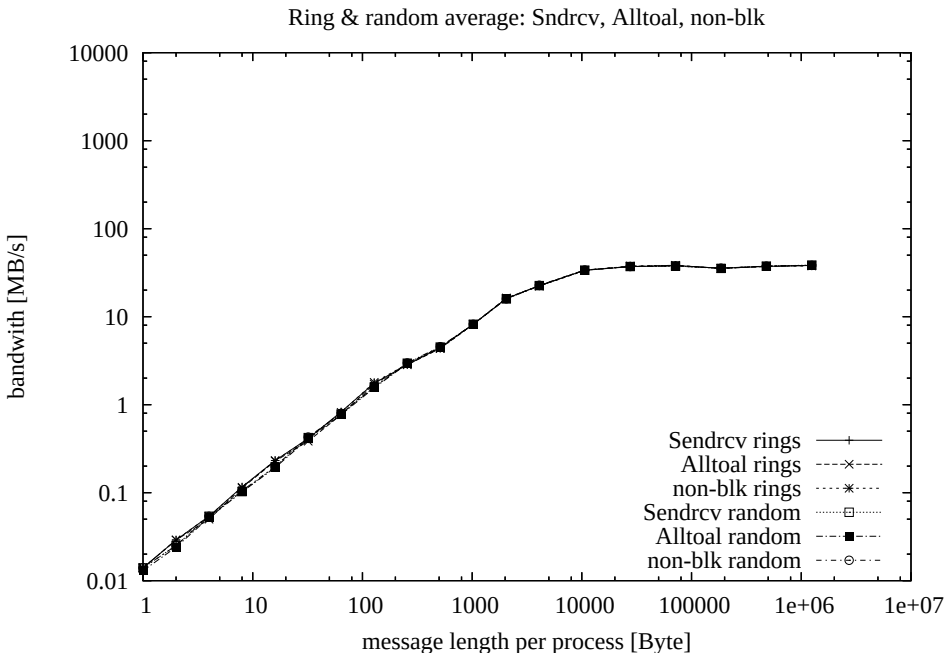

Alltoal, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all

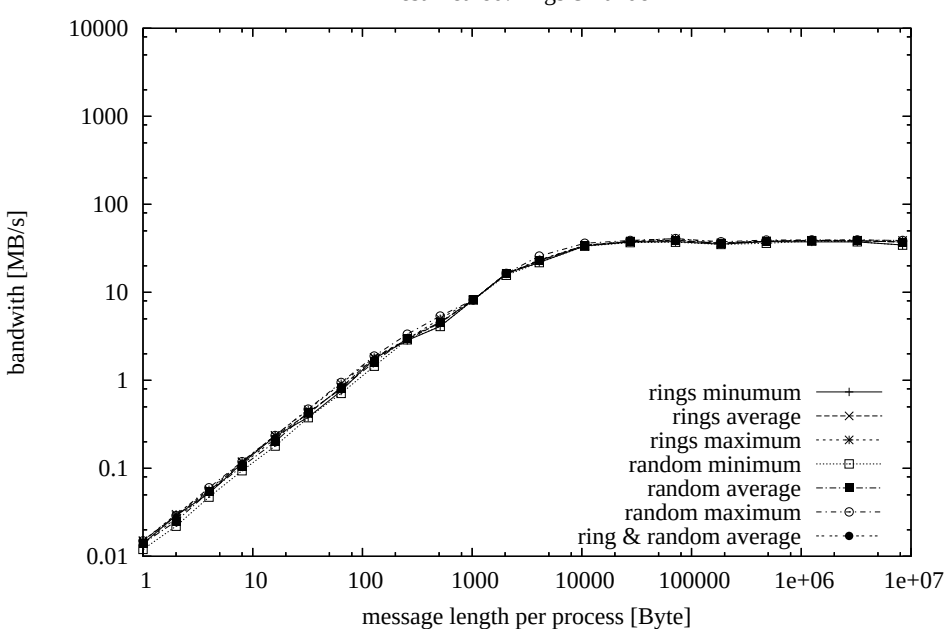

non-blk, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random


non-blk, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all


Best transfer method, ring & random patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
ring-1*2fix
worst random
avg random
best random


Best transfer method, additional patterns
bandwith [MB/s]
message length per process [Byte]
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07
worst-cyc-1dim
best bi-section
worst bi-section
acyclic-2dim-all
acyclic-3dim-all
cyclic-2dim-x
cyclic-2dim-all
cyclic-3dim-x
cyclic-3dim-all


Ring & random average: Sndrcv, Alltoal, non-blk
bandwith [MB/s]
message length per process [Byte]
Sendrcv rings
Alltoal rings
non-blk rings
Sendrcv random
Alltoal random
non-blk random
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07


Best method: rings & random
bandwith [MB/s]
message length per process [Byte]
rings minumum
rings average
rings maximum
random minimum
random average
random maximum
ring & random average
10000
1000
100
10
1
0.1
0.01
1
10
100
1000
10000
100000
1e+06
1e+07

A.2 PETSc

PETSc is a comprehensive numerical package developed by Argonne National Lab. PETSc stands for Portable Extensible Toolkit for Scientific Computation. It provides convenient parallel data structure for users to develop high performance parallel numerical algorithm. In this report , petsc-2.3.1-p15 has been used to develop a parallel equation solving kernel for the parallel computational system. Both parallel direct and iterative solving are available through a consistent interface.

Many external packages have to be compiled into PETSc in order for it to recognize the data format. Typical configuration flags look like

–with-debugging=no –with-shared=1 –with-mpi-dir=/usr/local –with-mpi=1 –useThreads=0 –with-superlu_dist=1 –download-superlu_dist=yes –with-superlu=1 –download-superlu=yes –download-f-blas-lapack=yes –with-spooles=1 –download-spooles=yes –with-blacs=1 –download-blacs=yes –with-scalapack=1 –download-scalapack=yes –with-spai=1 –download-spai=yes –with-hypre=1 –download-hypre=yes –with-plapack=1 –download-plapack=yes

The key features to include are decided by the solving algorithms user wants to use in analysis. For example, if one is interested in using SPOOLES solver, then the spooles flag must be set on and the package must be downloaded (or manually compiled) and linked with PETSc by setting relevant flags when compiling the package.

In this report , it has been shown that the spooles solving algorithm delivers best performance when pairing with penalty constraint handler. Some memory bugs have also been fixed by the author of this report . The file is available upon request.

A.3 ParMETIS

ParMETIS 3.1 provides complete multi-level graph partitioning/repartitioning algorithm for load balancing operations. The compiling is straightforward and no extra instruction is needed other than following the package README itself.

Appendix B

Import New Element/Material/Load etc. to PDD–based Parallel Program

This Appendix introduces the necessary steps to import new finite element model classes, such as **Element**, **Material**, **Load**, etc. to the PDD–based parallel program.

B.1 MovableObject

All new classes must be inherited from superclass **MovableObject** in order to use parallel framework. The **MovableObject** defines a unique class tag (in *SRC/classTags.h*) to identify every class that will be sent/received through network.

B.2 Send/RecvSelf

In the new classes, the clone functions **SendSelf/RecvSelf** must be implemented. The basic functionality of **SendSelf/RecvSelf** is to replicate the local object in remote process.

The straightforward implementation is to send/receive every private data member defined in the header files. In this sense, it should be sufficient to replicate the functionality provided by the copy constructor or the *clone()* function in Java.

B.3 Default Constructor

The new classes must also provide a default constructor which can be used by **FEM_ObjectBroker** to initially create an empty object in remote process, whose **RecvSelf** will be called to enable the remote clone process.

This default constructor, of course, can have no parameters. But from the efficiency point of view, author of the new classes should design a constructor for **FEM_ObjectBroker** that takes some default

construction list. In this way, those default parameters need not be sent through network anymore. This can save much time if the class will be instanced frequently.

B.4 FEM_ObjectBroker

The class of **FEM_ObjectBroker** is responsible for instancing the new class in the remote process. In order for the new class to be recognized by the broker, relevant *getNewYourClass*-type function must be added in *FEM_ObjectBorker*. The body of the *getNewYourClass*-type function basically returns the reference to the newly created instance by calling the default constructor introduced above.

B.5 getObjectSize

The function is used to get the exact volume size of data that will be sent through network when an object is replicated on the remote process. This metric is very important because it gives an exact measurement how many data is involved in communication. This communication overhead will be used in multi-objective repartitioning operations.

The body of this function is to sum up the size of each member that needs to be communicated in **SendSelf/RecvSelf**.

Appendix C

Commands to Invoke Parallel Equation Solvers

This Appendix presents example script to invoke parallel solvers available in PDD–based parallel program. All solvers are implemented through the interface of PETSc. In order to use these features, the PETSc package must be correctly compiled to link with corresponding libraries as explained in Appendix A. More detailed information about PETSc can be found at Balay et al. (2001, 2004, 1997).

C.1 Iterative Solvers

C.1.1 Conjugate Gradient Method

system Petsc -KSP KSPCG -PC NONE

C.1.2 Precoditioned Conjugate Gradient

Jacobi Preconditioner

system Petsc -KSP KSPCG -PC JACOBI

Incomplete Cholesky Preconditioner

system Petsc -KSP KSPCG -PC ICC -MatType MPIROWBS

C.1.3 GMRES Method

system Petsc -KSP KSPGMRES -PC NONE

C.1.4 Precoditioned GMRES

Jacobi Preconditioner

system Petsc -KSP KSPGMRES -PC JACOBI

Incomplete LU Preconditioner

system Petsc -KSP KSPGMRES -PC ILU

C.2 Direct Solvers

C.2.1 MUMPS

system Petsc -KSP KSPPREONLY -PC PCLU -MatType MPIAIJMUMPS

C.2.2 SPOOLES

system Petsc -KSP KSPPREONLY -PC PCLU -MatType SPOOLES_AIJ

C.2.3 SuperLU_DIST

system Petsc -KSP KSPPREONLY -PC PCLU -MatType SUPERLU_DIST