PARALLEL SOLUTION OF SOIL-STRUCTURE INTERACTION PROBLEMS ON PC CLUSTERS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TUNÇ BAHÇECİOĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CIVIL ENGINEERING

FEBRUARY 2011

Approval of the thesis:

PARALLEL SOLUTION OF SOIL-STRUCTURE INTERACTION PROBLEMS ON PC CLUSTERS

submitted by **TUNÇ BAHÇECİOĞLU** in partial fulfillment of the requirements for the degree of **Master of Science in Civil Engineering Department, Middle East Technical University** by,

Prof. Dr. CANAN ÖZGEN _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Güney Özcebe _____
Head of Department, **Civil Engineering**

Prof. Dr. Kemal Önder Çetin _____
Supervisor, **Civil Engineering Dept., METU**

Assist. Prof. Dr. Özgür Kurç _____
Co-supervisor, **Civil Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Kemal Önder Çetin
Civil Engineering Dept., METU _____

Assist. Prof. Dr. Özgür Kurç
Civil Engineering Dept., METU _____

Prof. Dr. Yener Özkan
Civil Engineering Dept., METU _____

Assist. Prof. Dr. Yalın Arıcı
Civil Engineering Dept., METU _____

Dr. H. Tolga Bilge
Turkish Military Academy _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: TUNÇ BAHÇECİOĞLU

Signature :

ABSTRACT

PARALLEL SOLUTION OF SOIL-STRUCTURE INTERACTION PROBLEMS ON PC CLUSTERS

Bahçecioğlu, Tunç

M.Sc., Department of Civil Engineering

Supervisor : Prof. Dr. Kemal Önder Çetin

Co-Supervisor : Assist. Prof. Dr. Özgür Kurç

February 2011, 85 pages

Numerical assessment of soil structure interaction problems require heavy computational efforts because of the dynamic and iterative (nonlinear) nature of the problems. Furthermore, modeling soil-structure interaction may require finer meshes in order to get reliable results. Latest computing technologies must be utilized to achieve results in reasonable run times.

This study focuses on development and implantation of a parallel dynamic finite element analysis method for numerical solution of soil-structure interaction problems. For this purpose first, an extensible parallel finite element analysis library was developed. Then this library was extended with algorithms that implement the parallel dynamic solution method. Parallel dynamic solution algorithm is based on Implicit Newmark integration algorithm. This algorithm was parallelized using MPI (Message Passing Interface). For numerical modeling of soil material an equivalent linear material model was used. Additional numerical verification of the implemented equivalent linear material model was shown by comparisons with EduShake software. Several tests were done to benchmark and demonstrate parallel performance of implemented algorithms.

Keywords: Linear Dynamic Analysis, Equivalent Linear Method, High Performance Computing, Parallel Computation, Soil-Structure Interaction

ÖZ

ZEMİN-YAPI ETKİLEŞİMİ PROBLEMLERİNİN BİLGİSAYAR KÜMELERİNDE PARALEL ÇÖZÜMLENMESİ

Bahçecioğlu, Tunç
Yüksek Lisans, İnşaat Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Kemal Önder Çetin
Ortak Tez Yöneticisi : Yrd. Doç. Dr. Özgür Kurç

Şubat 2011, 85 sayfa

Zemin yapı etkileşimi problemlerinin sayısal değerlendirilmesi problemlerin dinamik ve yinelemeli (doğrusal olmayan) yapısı nedeniyle ağır hesaplama yükü gerektirir. Ayrıca, zemin-yapı etkileşiminin modellenmesi güvenilir sonuçlar için daha sıkı ağ yapılarını gerektirebilir. Makul yürütme zamanı içinde sonuçlar elde etmek için en son bilgisayar teknolojileri kullanılmalıdır.

Bu çalışma zemin yapı etkileşimi problemlerinin sayısal çözümü için paralel dinamik sonlu elemanlar analizi yönteminin geliştirilmesi ve uygulanmasını hedefler. Bu amaçla öncelikle, genişletilebilir bir paralel sonlu elemanlar analizi kütüphanesi geliştirilmiştir. Daha sonra bu kütüphaneye paralel dinamik çözüm yöntemini uygulayan algoritmalar eklenmiştir. Paralel dinamik çözüm algoritması örtük Newmark integral algoritması üzerine kuruludur. Bu algoritma MPI (Message Passing Interface - Mesaj Geçme Arayüzü) kullanılarak paralel hale getirilmiştir. Toprak malzemesinin sayısal modellenmesi için doğrusala eşdeğer malzeme modeli kullanılmıştır. Uygulanan doğrusala eşdeğer malzeme modelinin ilave doğrulama testleri EduShake programı ile yapılan karşılaştırılmalarla gösterilmiştir. Çeşitli testler yapılarak uygulanan algoritmalar değerlendirilmiş ve paralel performansı gösterilmiştir.

Anahtar Kelimeler: Doğrusal Dinamik Analiz, Doğrusala Eşdeğer Yöntem, Yüksek Performanslı Hesaplama, Paralel Hesaplama, Zemin-Yapı Etkileşimi

To my family.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisors Assist. Prof. Dr. Özgur Kurç and Prof. Dr. Kemal Önder Çetin. They always pushed me towards perfection, opened the paths of success and tolerated my unusual working habits. Without their support, confidence and guidance this study could not be completed.

I also would like to thank my dear friends and colleagues Semih Özmen, Andaç Lüleç, Erdem Efek and Güray Erteği for their encouragements and enlightening discussions during researching and writing this work. It was and will be a pleasure to work with them.

I wish to thank my family for their love and support without this research as well as everything in my life would not come to life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF SYMBOLS

$\dot{}$	Time Differentiation.
[]	Matrix.
$[]^T$	Matrix Transpose.
{}	Vector.
A	Area; Acceleration; Amplitude.
α	Rayleigh Damping Constant.
β	Implicit Newmark Algorithm Constant; Rayleigh Damping Constant.
D	Damping Ratio.
Δt	Time Step.
E	Young's Modulus.
ϵ	Strain.
f	Frequency.
G	Shear Modulus.
g	Gravity Acceleration.
γ	Strain; Implicit Newmark Algorithm Constant.
G_{sec}	Secant Shear Modulus.
G_{tan}	Tangent Shear Modulus.
H	Height.
λ	Wavelength; Lagrange Multiplier; Damping Ratio.
$[B]$	Spatial Derivatives of Field Variables.
$[C]$	Damping Matrix.
$[E]$	Constitutive Matrix.
$[K]$	Stiffness Matrix.
$[K^{eff}]$	Effective Stiffness Matrix.
$[M]$	Mass Matrix.
$[N]$	Shape Function Matrix.
M_w	Earthquake Magnitude.
N	Shape Function.
π	Ratio of Circumference of Circle to Its Diameter.
ρ	Mass Density.
σ	Stress.
τ	Value of Time Between a Typical Time Step; Shear Stress.
u	Displacement.
$\dot{u}$	Velocity.
$\ddot{u}$	Acceleration.
$\dot{u}_n$	Normal Velocity.
$\dot{u}_t$	Tangential Velocity.
V	Velocity.
v	Poisson's Ratio.
$\{D\}$	Displacement Vector.

$\{\dot{D}\}$	Velocity Vector.
$\{\ddot{D}\}$	Acceleration Vector.
$\{F\}$	Force Vector.
$\{R^{ext}\}$	External Force Vector.
$\{R^{int}\}$	Internal Force Vector.
V_p	Pressure Wave Velocity.
V_r	Rayleigh Wave Velocity.
V_s	Shear Wave Velocity.
W	Energy.
w	Circular Frequency.
x, y, z	Cartesian Coordinates.
ζ, η, ξ	Reference Coordinates of Isoparametric Elements.

LIST OF ACRONYMS

BEM	Boundary Element Method.
CPU	Central Processing Units.
CUDA	Compute Unified Device Architecture.
DOF	Degree Of Freedom.
FDM	Finite Difference Method.
FEM	Finite Element Method.
GFLOP	Giga Floating Point Operations.
GPGPU	General Purpose computation on Graphics Processing Units.
GPU	Graphics Processing Units.
LEMON	Library for Efficient Modeling and Optimization in Networks.
MPI	Message Passing Interface.
MPICH2	Message Passing Interface Chameleon 2.
MSDN	Microsoft Developer Network.
MS-MPI	Microsoft Message Passing Interface.
MUMPS	MUltifrontal Massively Parallel Sparse direct Solver.
OpenCL	Open Computing Language.
OpenMP	Open Multi Processing.
PI	Plasticity Index.
PVM	Parallel Virtual Machine.
RAM	Random Access Memory.
SM	Symmetric Multiprocessor.
SMP	Symmetric Multi Processing.
SPU	Symmetric Processor Unit.
SSI	Soil-Structure Interaction.

CHAPTER 1

Introduction

1.1 Statement of the Problem

Numerical solution of problems soil-structure interaction (SSI) problems carry an important role in geotechnical engineering. Modeling soil media and simulating seismic wave propagation has its own difficulties. Mesh requirements of modeling wave propagation through an infinite soil layer can strain the memory limits of computers. Nonlinear and dynamic nature of the problem requires solving a system of linear equations repeatedly which can result in unbearable analysis times.

Today's computers are equipped with processors composed of multiple cores which are actually individual processors. For effective usage of modern processors applications must be developed by the help of parallel programming techniques. Although utilizing all cores of a processor is a big step in developing effective applications, for some problems it might not be enough. For applications that require more computing power utilizing computers connected to each other becomes the next step.

Numerical solution of SSI problems require the latest computing technologies to be utilized. In order to achieve solutions in reasonable times and solve problems with larger number of unknowns parallel computing technologies must be applied to SSI problems.

1.2 Research Statement

This study aims to develop a parallel dynamic finite element analysis algorithm that can be used to solve dynamic SSI problems. For this purpose first a general extensible parallel finite

element library was developed. Then, the library was extended with a parallel dynamic solution algorithm and an equivalent linear material model which enable solution of SSI problems. Several verification tests were performed to verify and benchmark the parallel performance of the implemented software.

1.3 Thesis Outline

In Chapter 2, a literature survey about parallel computing and SSI is given. In Section 2.1, parallel computing hardware and software implementations to use this hardware are classified. Methods for numerical analysis of SSI problems and parallel implementations of these methods are presented in Section 2.2.

Chapter 3 presents the implemented general purposed extensible parallel finite element library: Panthalassa. Class architecture of the library is detailed.

Implemented parallel algorithms for parallel linear dynamic and equivalent linear analysis are given in Chapter 4. Both theory behind these algorithms and details of the implementations are presented in this chapter.

In Chapter 5, results of a series of verification problems that benchmark the dynamic linear and equivalent linear analysis methods are given.

Chapter 6 presents results of a series of tests that verifies and benchmarks the performance of parallel dynamic linear and dynamic equivalent linear implementations. In addition to the results a discussion on the performance of implementations is given.

Finally, Chapter 7 gives a brief summary of this study and outlines the conclusions that can be made from the study.

CHAPTER 2

Literature Review

2.1 Parallel Computing

2.1.1 Introduction

Parallel computing is a term indicating two or more computations executed in the same time. The idea of parallel computing was proposed by researchers around mid-1950s (Wilson, G.V. [1]) but the idea became a reality when Burroughs Corp. introduced D825, a four-processor computer that accessed up to 16 memory modules via a crossbar switch (Anthes, G. [2]). Parallel computing continued to develop and today parallel computers are being utilized frequently as almost every computer comes with processors composed of more than one core.

Parallel computation is implemented by different architectures, with unique advantages and disadvantages. Every architecture aims to increase speed-up (increase in speed versus a sequential architecture) and scalability (keeping constant speed with growing problem size) (Trobec, R. [3]). Parallel architectures can be categorized according to different properties. According to the type of memory access of processors parallel architectures are divided in two divisions: Shared memory and distributed memory architectures. Another specialized architecture that is recently put into use is the GPGPU (General-Purpose Computing on Graphics Processing Units) architecture. GPGPU architecture utilizes GPUs (Graphics Processing Units) for computations. In the next sections these architectures are discussed in detail.

2.1.2 Shared Memory Architecture

In the shared memory architecture processors use a single RAM (Random Access Memory) space. This architecture type is widely used in today's laptop and desktop systems. Figure 2.1 presents INTEL quad core architecture as an example to shared memory architectures. In this system processing units are named as cores and four cores are combined to create the processor. Every core has its own working memory space called L1 cache. In addition to the the L1 cache there is a separate memory space shared by every two cores called the L2 cache. RAM can be accessed by all cores. Cache space is very small compared to RAM, however it is much faster. Caches are internally used by processors for storing data from RAM before computation and connecting to each other and generally not available to programmers. This type of architecture which consists of cores is also known as the multicore architecture.

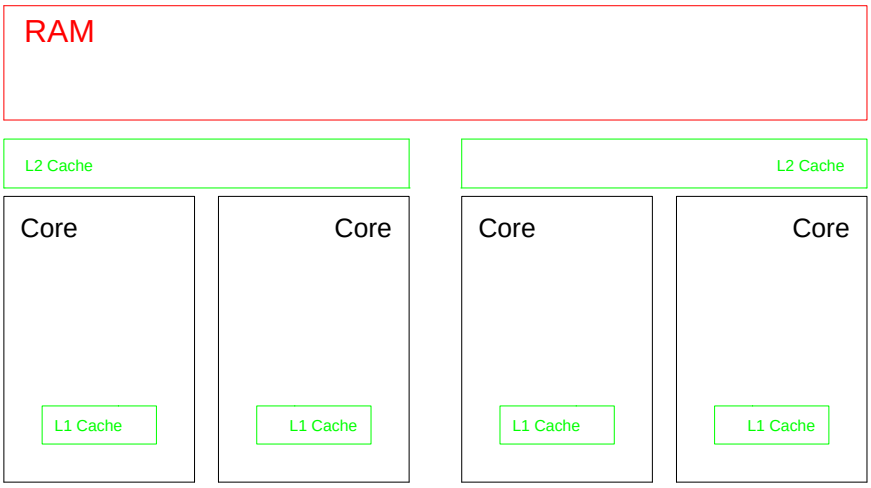


Figure 2.1: Intel Quad-Core Processor Architecture

Another example to the shared memory architecture is the SMP (Symmetric Multi Processing) architecture. SMP architecture uses processors instead of cores and processors are connected via a bus instead of cache memory.

Programming for shared memory architecture involves an application creating execution units called threads. Threads are controlled by the operating system and executed by cores or processors depending on the architecture. Applications initiate parallel execution by creating a number of threads, usually equal to the number of processing units. Ideally, every thread com-

putes data from different parts of the RAM and writes the result to a different part of RAM. If threads need to access the same location of the RAM, reading or writing from RAM becomes sequential since RAM cannot be accessed by more than one processing unit. Problems that occur from un-enforced dependence of threads similar to this situation are known as race conditions (Sottile et al. Chapter 3.1.1 [4]).

Every programming language comes with different methods to execute and control threads. OpenMP (Open Multi Processing [5]) is the most common method for C++ and Fortran languages. OpenMP introduces new keywords to C++ and Fortran languages to support threads. Other methods usually rely on operating system level functions to control threading. Libraries are developed to ease the use of these functions. C++0x which is the latest standard of the C++ language adds a threading library to the standard libraries ([6]). Other languages like C# and JAVA have already threading support in their standard libraries.

Beside using threads another way of programming the shared memory architecture is to use message passing. In this method more than one process for an application is executed. Processes communicate to each other by using message passing libraries. If libraries that are specially designed for shared memory systems are used, overhead of passing messages between processes is minimized. Several implementations of the MPI (Message Passing Interface [7]) interface can be used for this purpose like MS-MPI (Microsoft MPI [8]) or MPICH2 (Message Passing Interface Chameleon 2 Library [9]). Message passing between processes for parallelism is also used for programming distributed memory architectures. This flexibility is a major advantage if applications need to be run in both architectures.

2.1.3 GPGPU Architecture

Computer graphics and animations require heavy usage of floating point operations. With the boom of game industry the prices of GPUs fell down and they became gradually more advanced. It is suddenly realized that they can be used for general purpose applications.

GPUs differ vastly from CPUs (Central Processing Units) in design. Figure 2.2 presents the Nvidia Fermi architecture. Main execution unit of Fermi architecture is the CUDA (Compute Unified Device Architecture) cores (dark green rectangles in Figure 2.2). Every 32 CUDA cores are grouped as SMs (Streaming Multiprocessors). In Figure 2.2 32 SMs combine for

512 CUDA cores. CUDA cores execute instructions in groups of 32 which are called warps. CUDA cores have access to a small memory local memory called registers. Every SMs host a bigger memory block called shared memory. This shared memory can be accessed by every CUDA core in the SM. ALL SMs have access to one big but slow memory block called the global memory. Since both shared memory and global memory can only be accessed sequentially race conditions are a big platform in GPU architecture. ATI GPU design is somewhat similar to the Nvidia design however the main processing unit is called SPUs (Streaming Processor Units).

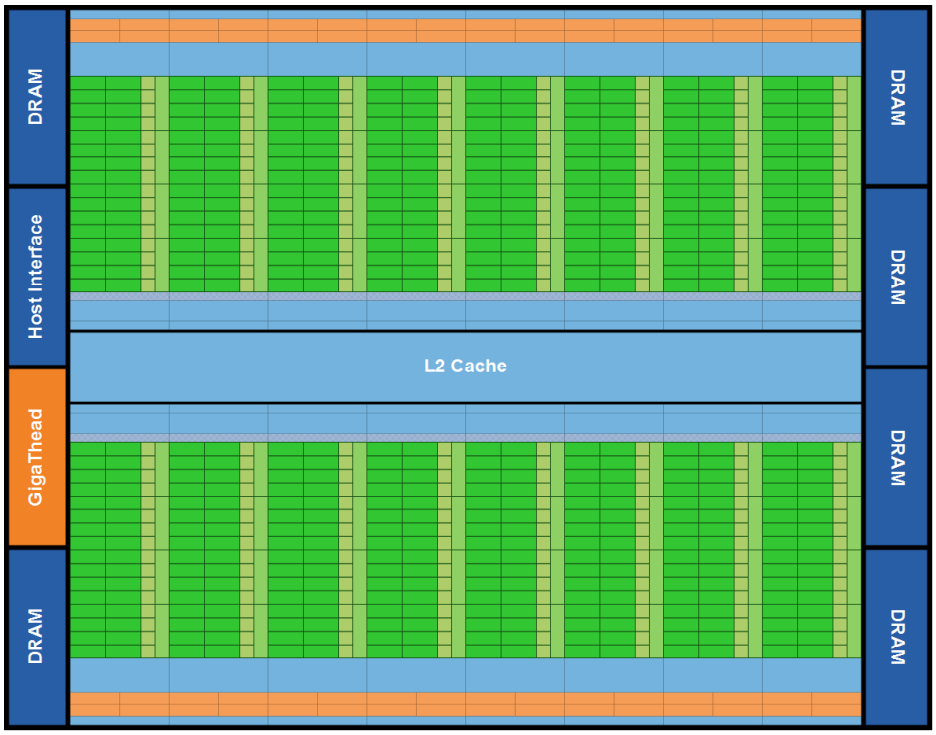


Figure 2.2: Nvidia Fermi Architecture (Nvidia Fermi Architecture Whitepaper [10])

Unique design of GPUs allow for incredibly fast floating point operations. For example Nvidia GTX 460 GPU can compute 900 GFLOP (Giga Floating Point Operations) a second whereas a similarly priced INTEL CPU, the Intel Core i5 760 can only compute around 50 GFLOP a second. However using all the processing power of a GPU is harder than a CPU. Since GPU has a different global memory than CPU problem and solution data has to be transfered from main RAM to the global memory of GPU. Also programming the GPU has its own unique challenges because of the special design of processing units and internal

memory. Another issue is although GPUs are very fast at computing single precision floating point arithmetic, they are much slower at computing double precision floating point arithmetic which is a must for some of the scientific applications.

Currently there are three methods for programming GPUs for general purpose applications: CUDA (Compute Unified Device Architecture [11]), OpenCL (Open Computing Language [12]) and DirectCompute (Microsoft DirectCompute [13]). All methods are composed of special libraries and a software language similar to C (Language is different for every method). CUDA can only be used with Nvidia GPUs whereas OpenCL and DirectCompute can be used for programming both Nvidia and ATI GPUs. OpenCL can also be used for programming CPUs. Discussions on these methods can be found in the following papers: Kindratenko et al. [14] and Karimi et al. [15].

2.1.4 Distributed Memory Architecture

Figure 2.3 presents an example for the distributed memory architecture. In this architecture type processors that have access to their own memory space are connected by a network. If the processors are regular computer devices, the system is called a computer cluster. Clusters with computers connected through the Internet are called grids (Sterling, T. L. [16]). If processors are specially designed to be a one large computer the system is called a massively parallel processor (Potter, J. L. [17]). Processors of a distributed memory architecture can also be shared memory or GPGPU devices. As of 24-12-2010 fastest super computer in the world is a distributed memory system that connects Nvidia GPUs (Top500 [18]).

Since every processor in a distributed memory architecture has its own memory space programming this type of architecture needs message passing through a network. There are two standards for message passing libraries today: PVM (Parallel Virtual Machine [19]) and MPI. A comparison of theses standards can be found in Geist, G. A. and Kohl, J.A. [20]. Freely available and more advanced message passing libraries that use the MPI standard made MPI the popular interface in the last ten years.

Message passing libraries have special applications that can start processes at every processor of a distributed memory architecture. In this way an application can be executed at every processor and solve a problem in parallel by passing messages with each other.

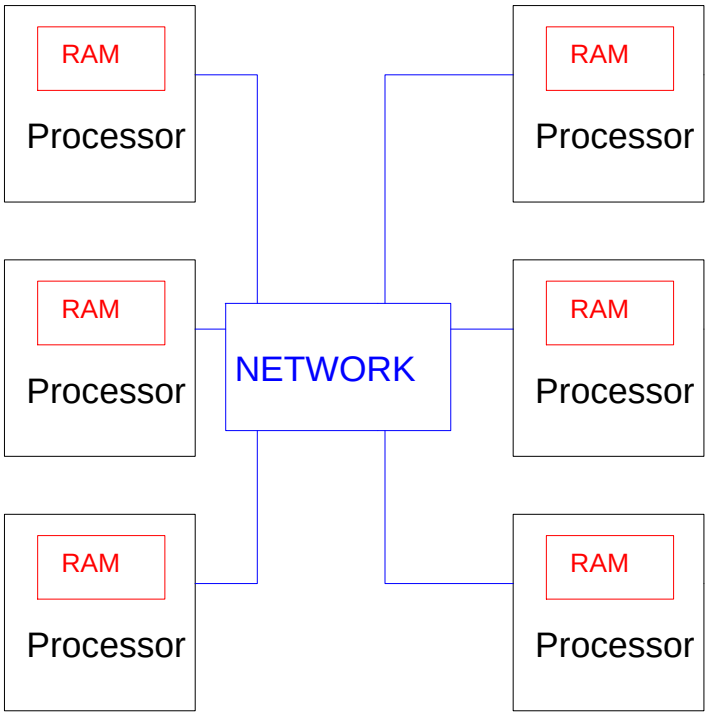


Figure 2.3: Distributed Memory Architecture

2.2 Soil-Structure Interaction

2.2.1 Numerical Methods

For numerical solution of static and dynamic SSI problems three methods are used: Finite element method (FEM), finite difference method (FDM) and boundary element method (BEM). All these methods are actually used for solution of differential equations. They are applied to mechanical problems usually through a discretized continuum.

In the finite element method mechanical problem is discretized using geometrical structures called finite elements. Connection points of finite elements are called nodes (Figure 2.4). In each finite element, variation of variables that describe the nature of the problem (displacement, velocity, temperature etc.) are expressed as simple spatial variations commonly described by polynomial terms. Numerical unknowns, which are values of variables described by finite elements, at nodes are calculated using spatial variations.(Cook et al. [21])

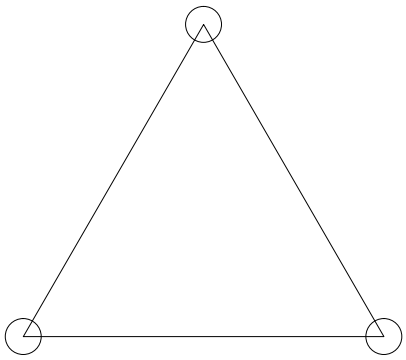


Figure 2.4: Triangle Finite Element With Three Nodes

In the finite difference method derivatives in a function that describes a quantity in a specific problem is replaced by its approximations. Then continuum can be discretized as points and values of the approximated variable at points can be calculated.

Boundary element method is based on discretization of boundaries of the problem. Problem definition is reduced to its boundaries which is one dimension less than the problem. For example Figure 2.5 presents an area mesh reduced to its boundaries: a line mesh. This property of BEM is a major advantage as it greatly simplifies the solution procedure. However BEM cannot be used to solve all types of differential equations. Green functions are used to move the problem to its boundary, thus for differential equations that are solvable with BEM Green function of the problem must be calculatable.

2.2.2 Absorbent Boundaries

In order to model infinite soil media absorbent boundaries that absorb waves are used. Boundary conditions for normal and shear stresses at an absorbent boundary are given in the follow-

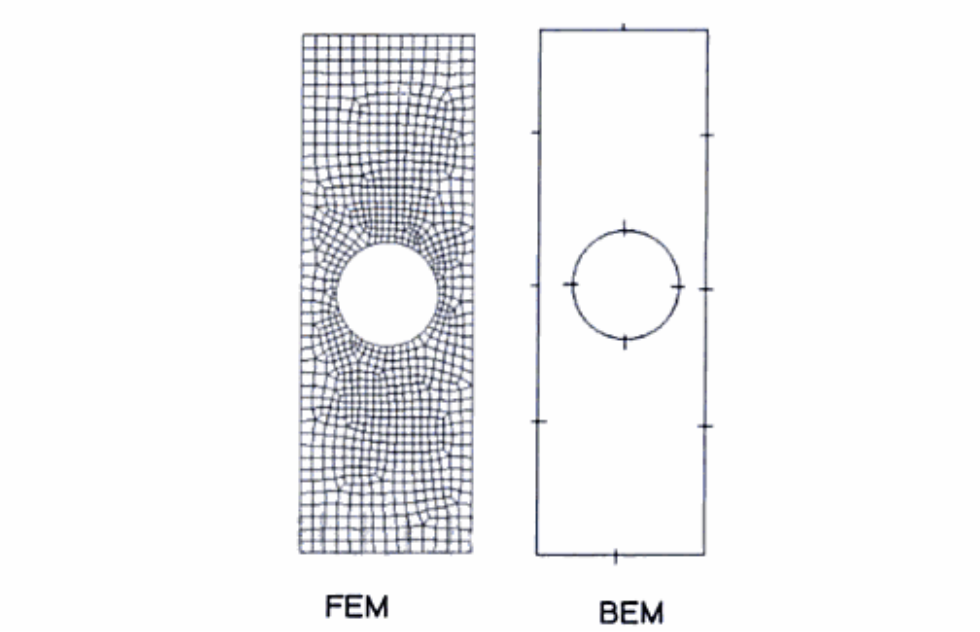


Figure 2.5: A Finite Element Mesh and Its BEM Representation (Aliabadi [22])

ing equations (Lysmer and Kuhlemeyer [23]):

$$\sigma = a\rho V_p \dot{u}_n \tag{2.1}$$

$$\tau = b\rho V_s \dot{u}_t \tag{2.2}$$

In these equations a and b are constant parameters, ρ is density, V_p and V_s are velocities of pressure and shear waves, $\dot{u}_n$ and $\dot{u}_t$ are normal and tangential velocities at the boundary. Equation 2.1 give perfect absorption in one dimension when only pressure waves are considered with $a = 1$. A detailed discussion on parameters a and b can be found in White et al. [24].

Absorbent boundary conditions can be applied in numerical calculations either by lumped formulation at boundaries or by using consistent matrices calculated at finite elements. Lumped formulation is easy to implement in software, however causes significant errors in calculations (Chow [25]).

2.2.3 Wave Propagation in Finite Element Mesh

Largest finite element size that transmits a wave with wavelength λ is given by Equation 2.3 (Lysmer and Kuhlemeyer [23]).

$$\Delta l = \frac{\lambda}{10} \tag{2.3}$$

In terms of frequency of the wave:

$$f = \frac{V}{\lambda} \tag{2.4}$$

$$\Delta l = \frac{V}{10f} \tag{2.5}$$

In Equations 2.4 and 2.5, V is the velocity of wave and f is the largest frequency of wave that is transmitted by a finite element with size Δl. Finite element meshes should be created according to Equation 2.5 for accurate results.

2.2.4 Parallel Soil-Structure Interaction Applications

Several studies for parallel solution of SSI problems exist in literature. Some of them are specifically developed for SSI problems, others are developed as general numerical solution procedures for mechanical problems and can be utilized for SSI problems as well.

In Yerli et al. [26] SSI system is divided into individual parts and solved by multiple processors using the finite element method (Figure 2.6). Substructures create a separate system of linear equations for the interface known as Schur Complement equation. Both substructure and interface equations are solved in parallel using PVM.

Several parallel implementations of the explicit Newmark integration method can also be given as examples to parallel SSI applications. Explicit Newmark integration method implemented with finite element method enables element by element solution of SSI problems (Hughes and Liu [27]). Finite elements that make up the problem's mesh are partitioned to processors and solved in parallel. Solutions are then combined using parallel computing techniques. The first examples used PVM for parallelization of the algorithm. For example Krysl and Belytschko [28] came up with an object oriented parallelization algorithm that used Nonlinear Explicit Integration and PVM to solve structural dynamics problems. As MPI replaced PVM, researchers used different algorithms using MPI to parallelize the dynamic integration. Krysl and Bittnar [29] used MPI with different decomposition techniques for the solution of

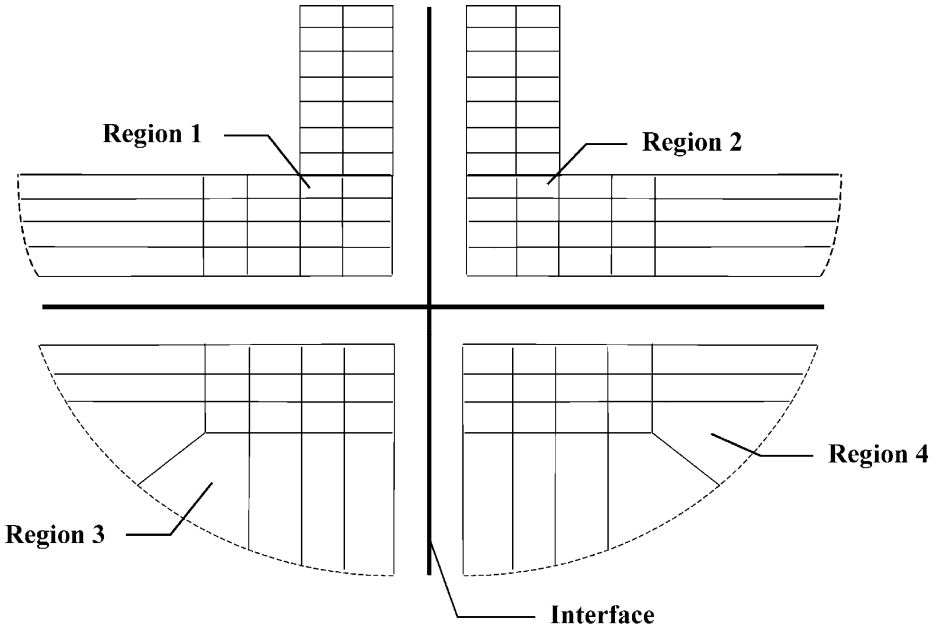


Figure 2.6: Disconnected Substructuring Representation of an SSI System (Yerli et al. [26]).

dynamic finite element problems. Some GPGPU implementations of the explicit Newmark integration method are also available. Noe and Sorensen [30] presented a real time simulation of nonlinear elastic material properties using Total Lagrangian Explicit Dynamic finite element method running on GPU. Komatitsch et. al. [31] used second order Newmark dynamic integration equations to model seismic wave propagation on a large GPU cluster. In this study MPI was used to parallelize the algorithm on computer networks.

CHAPTER 3

An Extensible Parallel Finite Element Analysis Environment: Panthalassa

3.1 Introduction

Panthalassa[1] is a computer library, intended to solve general finite element problems. Although Panthalassa can be used for solving any type of finite element problem, current implementation is focused on structural and geotechnical ones. Panthalassa was developed in c++ language, with state of the art object oriented design techniques. Panthalassa was built upon the idea of using more than one processor for computation, thus it provides data structures and a base foundation for parallel computing. Panthalassa was developed as a core finite element library; in other words, it provides necessary data structures and methods for the numerical solution of finite element problems, on the other hand it does not provide any implementation of necessary algorithms. Finite elements, material models, solution algorithms are added on the core system, by the help of plug-ins [2]. In this way, any type of modeling and solution algorithm, can be implemented without modifying the core library.

Panthalassa was solely based on object oriented design. All components of the library were programmed as classes and techniques like inheritance and polymorphism were used in the design of the library. Section 3.2 explains details of the object oriented data structure of Panthalassa.

Object oriented design of Panthalassa was also used in the plug-in architecture of the library. Base classes provided by Panthalassa can be inherited and extended for specific algorithms

[1] Vast global ocean that surrounded the super-continent Pangaea, during the late Paleozoic and the early Mesozoic years (Wikipedia [32])

[2] An accessory program designed to be used in conjunction with an existing application program to extend its capabilities or provide additional functions (Houghton Mifflin Harcourt [33]).

as plug-ins. Plug-ins are developed separate from Panthalassa to extend the functionality of the core library. To add plug-in functionality to Panthalassa, a computer library called Pugg [3] was developed. Details of Pugg library and plug-in architecture of Panthalassa are discussed in section 3.2.7.

Panthalassa follows the MPI standards for parallel programming. MPI standards provide a set of function definitions for interprocess communication, along processes from computers connected by special networking hardware. Panthalassa uses the MPICH2 implementation of MPI standards. MPICH2 is a widely known open source implementation of MPI standards, focusing on homogeneous hardware. Since applications based on MPI standards can be used sequentially, Panthalassa can be used in single processor systems without any modifications.

3.2 Object Oriented Design

Object oriented design of Panthalassa, can be assorted into several subgroups as presented in Figure 3.1.

The first subgroup is solely composed of the *Domain* class. *Domain* class hosts and maintains objects in memory and directs the execution of Panthalassa. Second subgroup, Finite Element Model Classes, represent components of a finite element model. Entities like finite elements and nodes of a finite element model are programmed as classes in this group. The third subgroup, Analysis Classes, include classes that represent solution procedures for the finite element method. Subgroup, Input-Output classes, is responsible for reading user input and outputting solution results to disk. Lifeline of classes from Finite Element Model Classes, Analysis Classes and Input-Output Classes are maintained by the *Domain* class. Subgroups, Utility classes and Pugg Library, aid the implementation of data structures defined in Panthalassa. Utility Classes are used by the classes that belong to above defined three subgroups for service purposes. Pugg Library was developed to implement plug-in system into Panthalassa. Plug-ins developed bu users are loaded and maintained by classes from the Pugg Library.

[3] Named after the dog breed pug.

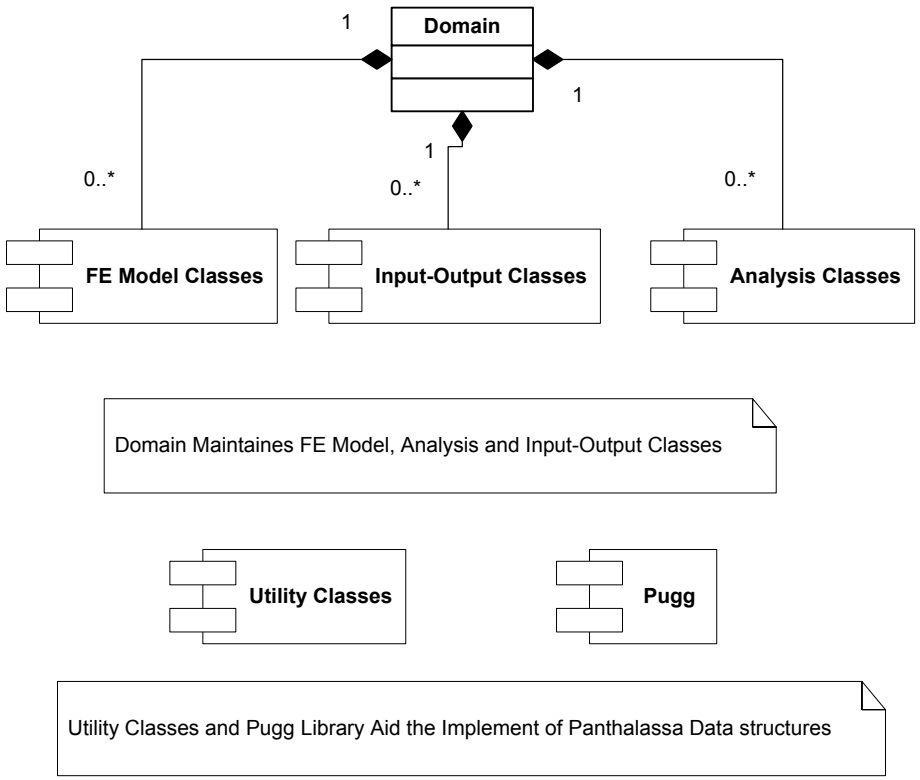


Figure 3.1: Panthalassa Components

3.2.1 Domain Class

Domain class is the backbone of Panthalassa. It is responsible for creation and maintenance of all Finite Element Model, Analysis and Input-Output Classes as presented in Figure 3.2. Objects from Analysis Classes are held in special containers discussed in Section 3.2.3.2. Objects from Finite Element Model Classes are held in a *Structure* object. This *Structure* object holds and maintains other Finite Element Model Classes as explained in Section 3.2.2.7.

In addition to maintenance of objects, *Domain* class directs the execution of Panthalassa. *Domain* object loads plug-ins available to the system, using the Pugg Library, to initiate the execution. Next, user input is read from a text file. This text file consists of statements described in a special language called Ptl. Every statement indicates either the creation of an object or a value to define an already created one. Dictated by these statements *Domain* object creates and initializes objects of classes from Panthalassa Library and loaded plug-ins.

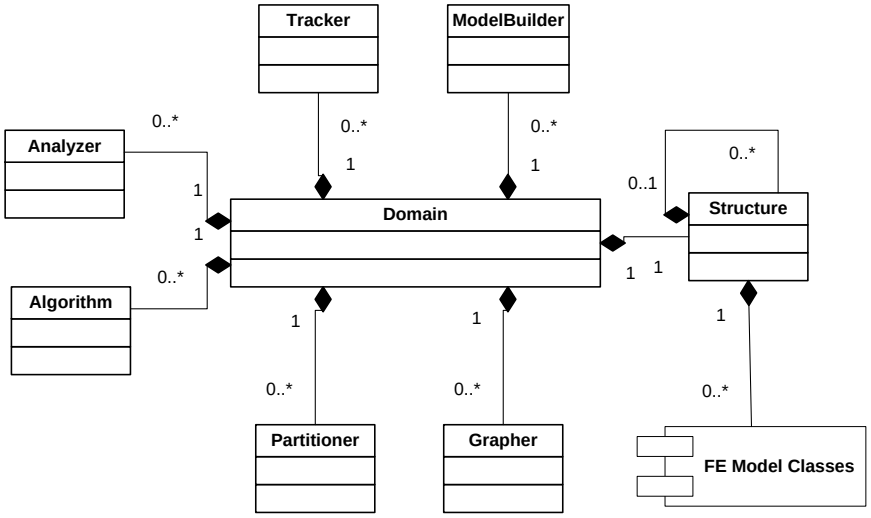


Figure 3.2: Domain Class Diagram

At last, control of the system is given to the *Analyzer* object (section 3.2.4.1) created by the user in order to start the solution process.

3.2.2 Finite Element Model Classes

3.2.2.1 FEMObject Class

FEMObject is the superclass for all finite element model classes. It is implemented in order to group common properties of finite element model classes (Figure 3.3).

FEMObject class has two important attributes: *id* and *partition*. Attribute *id* is an unsigned integer,determined by the user to distinguish different objects from one another. Panthalassa includes special container classes (section 3.2.3.2) that have unique functions for objects that have the *id* attribute. Attribute *partition* is used in order distinguish the partition that the object belongs in case a domain decomposition process is applied to the system (Section 3.2.5).

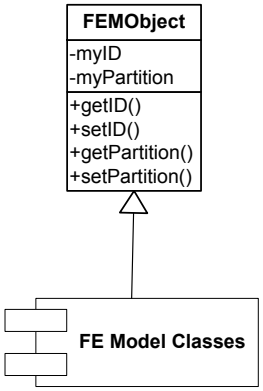


Figure 3.3: *FEMObject* Class Diagram

3.2.2.2 Node Class

Node class represents a node of a finite element model. Nodes are used to define the coordinate geometry of finite elements. Figure 3.4 presents attributes of the *Node* class and its relationship with the *Element* class.

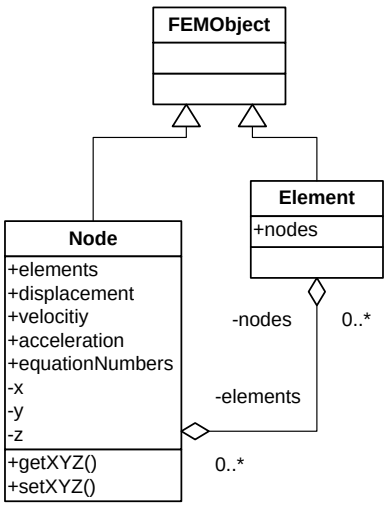


Figure 3.4: Node Class Diagram

Node class has pointers to its connected *Element* objects (*Element* objects have a similar

attribute, nodes, for its connected *Node* objects), that can be used in different algorithms such as DOF numbering or partitioning. *Node* class stores the coordinates, displacements, velocities and accelerations of the node corresponding to the current step in analysis. Equation numbers of DOFs are also stored in the Node object.

3.2.2.3 Element Class

Element class represents a finite element of a finite element model. *Element* objects are defined by a number of connected nodes, a *Property* object and a *MaterialModel* object. *Element* class is an abstract class. It is inherited to implement different types of finite elements (Figure 3.5).

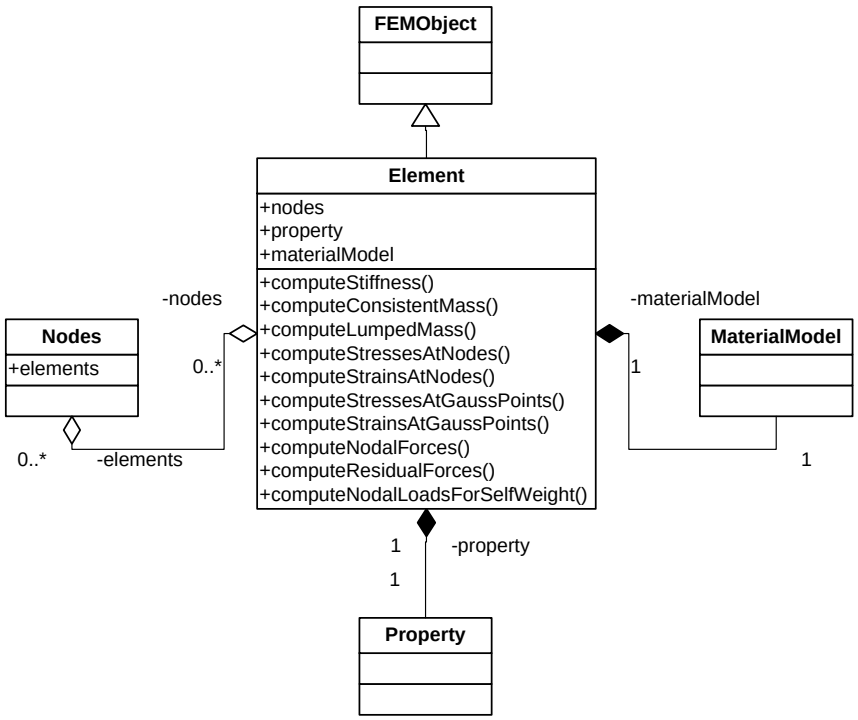


Figure 3.5: Element Class Diagram

Element geometry is defined by a series of Node objects connected to the element by the user. Geometric properties of an element (thickness, area etc.) are defined by connecting

a Property object to it. Property is a user defined object that can store a number of double values. The meaning of these numbers are specific to each element type. *Element* gets its material properties, density and the constitute relationship, from a *MaterialModel* object. *MaterialModel* class is discussed in section 3.2.2.4.

The main purpose of a finite element is to define its stiffness, mass, and unique relationships between displacement, strain and stress states. *Element* class has a number of virtual functions returning different matrices defining these relationships (Table 3.1). Inheritors of the *Element* class override these functions to implement the new finite element type. Another responsibility of an *Element* object is to compute the equivalent nodal forces to its connected Elemental forces as Panthalassa can only work with loads connected to nodes.

Table 3.1: Virtual Functions Defining Specific Element Behavior

Function	Explanation
computeStiffness	Returns the Stiffness Matrix
computeConsistentMass	Computes the Consistent Mass Matrix
computeLumpedMass	Computes the Lumped Mass Matrix
computeStressesAtNodes	Computes Stresses of the Element at its Nodes
computeStrainsAtNodes	Computes Stresses of the Element at its Gauss Points
computeStressesAtGaussPoints	Computes Strains of the Element at its Nodes
computeStrainsAtGaussPoints	Computes Strains of the Element at its Gauss Points
computeNodalForces	Computes the Total Forces of the element at its Nodes
computeResidualForces	Computes the Residual Forces of the Elements at its Nodes
computeNodalLoadsForSelfWeight	Computes Equal Nodal Forces to Elemental Forces

New finite element types can be added on Panthalassa using the plug-in system. Finite element types that are added in this way are listed in Table 3.2.

3.2.2.4 MaterialModel Class

In the finite element method, constitutive properties of a finite element is represented by the constitutive matrix. In simple terms, it is the relationship between the stress and the strain state of an integration point of a finite element and expressed as (Potts and Zdravkovic [34]):

$$\{\sigma\} = [E]\{\epsilon\} \tag{3.1}$$

Table 3.2: Implemented Elements

Element Type	Class Name
8 Node Solid	Brick
4 Node Quadrilateral	Quad4
6 Node Wedge	Wedge
Timoshenko Beam	Beam
Truss	Truss
NonLinear Green Truss	GreensTruss
NonLinear Corotational Truss	CorotationalTruss
Rectangular Thick Shell	ThickShell
Rectangular Thin Shell	ThinShell
Triangular Thick Shell	ThickShellT
Triangular Thin Shell	ThinShellT

In this equation $\{\sigma\}$ is the stress vector, $\{\epsilon\}$ is the strain vector and $[E]$ is the constitutive matrix. In the object oriented structure of Panthalassa, constitutive properties of an element is represented by the *MaterialModel* class.

MaterialModel class is an abstract class, different material models are implemented by inheriting this class and overriding its virtual functions (Section 4.3.2). An implementation of the equivalent linear material model will be discussed in section 4.3.2.

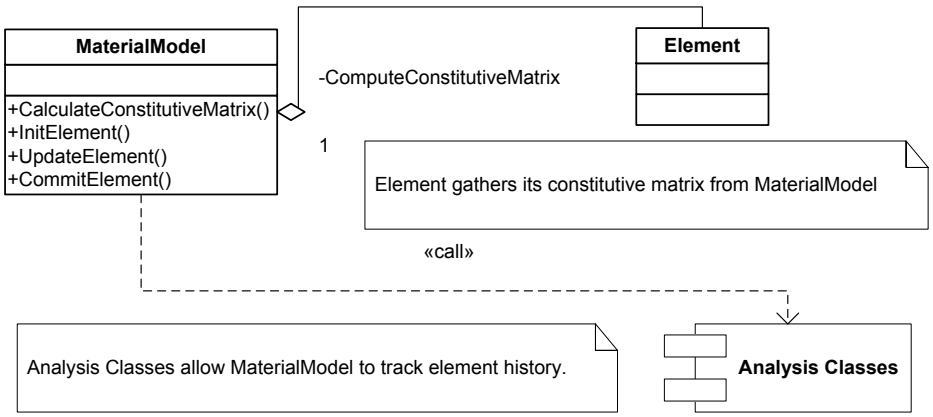


Figure 3.6: MaterialModel Class Diagram

Every *Element* object is connected to a *MaterialModel* object by the user. Elements call the *calculateConstitutiveMatrix* function in order to gather their constitutive matrix specific to

their stress and strain states. Some material models not only consider the current state of elements but take into account the stress or strain history of the elements. Three functions: *initElement*, *updateElement* and *commitElement* are called by the analyzer classes to allow the *MaterialModel* object track the history of *Element* objects. *initElement* function is called in the beginning of the analysis, whereas *updateElement* function is called at every step of the analysis. In some cases where the analysis process is applied more than one time to the model *commmitElement* function is called at the end of every analysis cycle. *MaterialModel* tracks the necessary element output to compute the constitutive matrix at the next step of the analysis.

3.2.2.5 Damper Class

Dampers are useful tools for modeling infinite surfaces in finite element models. *Damper* class is implemented to represent viscous dampers (Lysmer and Kuhlemeyer [23]).

Damper class gets three values from user to identify damping constants in three directions: x,y,z. These values are directly assigned to the viscous damping matrix in the analysis at places identified by the connected nodes of the damper.

3.2.2.6 Load Classes

Load classes represent different types of excitations, applied to the model. A combination of excitations is represented by the Loading class. *Loading* class holds different types of *Load* classes and computes the combined impact of these during the analysis phase.

Panthalassa employs three load classes that identifies with different load types: *NodalLoad*, *ElementLoad* and *DynamicLoad* as presented in figures 3.7 and 3.8. The first of these, the *NodalLoad* class, represents a static force applied on a specific node of the model. *Element-Load* is similar to *NodalLoad* except it defines a surface or a body force acting on a finite element. In contrast to first two load types, *DynamicLoad* represent an excitation applied dynamically on the model. Excitation represented by the *DynamicLoad* class can be force, displacement, velocity or acceleration. Since magnitudes of the excitation change over time they are defined by the user with discretized points of a function $f(time)$.

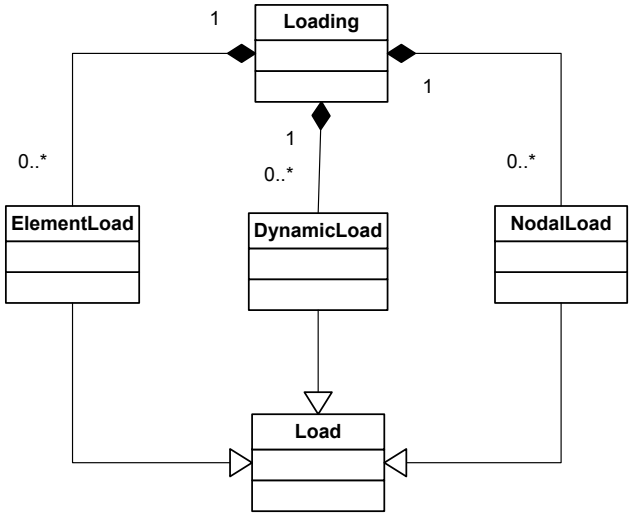


Figure 3.7: Load Classes Class Diagram

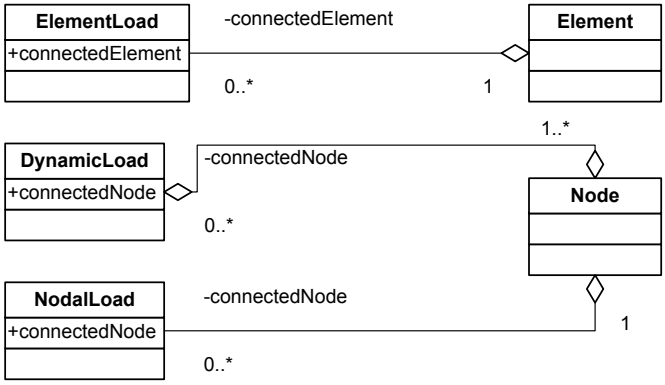


Figure 3.8: Load Classes Connections

3.2.2.7 Structure Class

Structure class represents a finite element model. Depending on how the finite element problem is modeled, one *structure* object or a series of them are used to represent the model in memory. *Structure* hosts any type of object from the finite element model classes, including other *Structure* objects, to describe the model (Figure 3.9). Algorithms using domain decomposition or substructuring methods (Kurç, Ö. [35]) can use this feature of *Structure* class to

model a hierarchy of *Structure* objects.

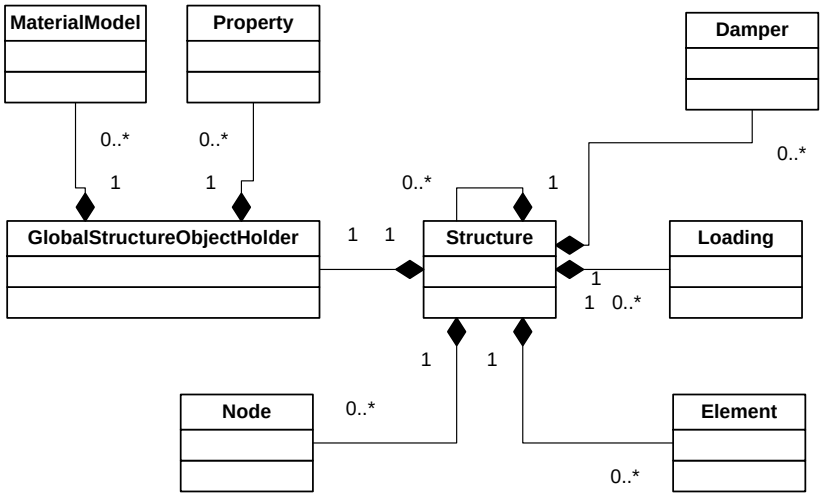


Figure 3.9: Structure Class Diagram

Property and *MaterialModel* objects created by the user are shared by every *Structure* defined in the system. These objects are stored in a storage class called *GlobalStructureObjectHolder*. Every structure has a pointer to a unique *GlobalStructureObjectHolder* object. This sharing allows the user to define elements having the same properties and/or material models that belong to separate *Structure* objects.

3.2.3 Utility Classes

3.2.3.1 FEMObjectWithOptions Class

FEMObjectWithOptions class is an abstract class inherited from *FEMObject* class that adds the ability to gather extra information from user to its sub-classes. Statements used for creating objects in the Ptl language includes only general information about the properties of the soon to be created object. Specific information related to the type of object is not included. Objects gather the required extra information from user with user options.

For example, a 2D membrane *Element* object can get information about its geometric behavior (plane stress, plane strain or axisymmetric) from user. Statements that create an instance of

this element type are given below:

```
// Create a hypothetical element with id 1,  connected to property with id 2,
// MaterialModel with id 4, connected to Nodes with ids: 22 23 24 25
Create.Element "2DMembrane" 1 2 4 22 23 24 25
// Set Behavior option to axisymmetric
Set.Option  element 1 // set options for the element with id 1
{
"Behavior" = "axisymmetric"
}
```

Classes that user can assign options are shown in figure 3.10.

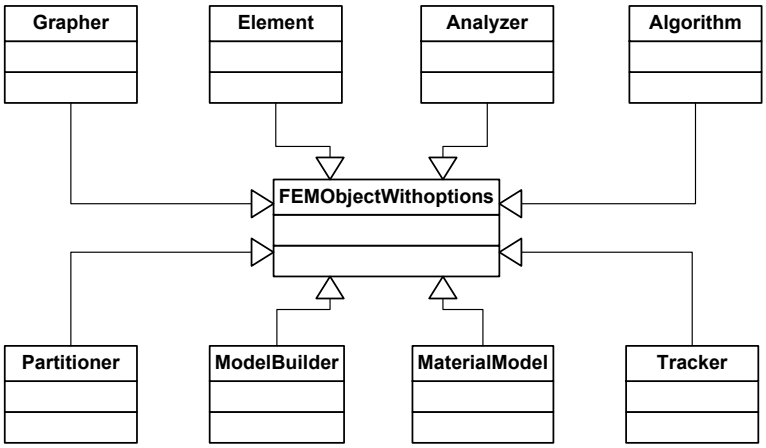


Figure 3.10: Classes that can Use User Options

3.2.3.2 Container Classes

Container classes are generic classes that hold and manipulate subclasses of the *FEMObject* class in memory. They are implemented in order to collect objects of the same type from Finite Element Model classes.

There are two different container classes defined in Panthalassa : *IDObjectVector* and *IDObjectMap*.

The *IDObjectMap* class maps an unsigned integer (*id* of the *FemObject* class) to every stored object. This storage type achieves fast random access but relatively slow sequential access to

the stored objects. Objects that need random access during analysis like objects from analyzer Classes and Node objects are stored by using *IDObjectMap* objects.

IDObjectVector class implements a vector of objects, stored as an array. Array type storage, increases the speed of accessing objects in a sequential manner.Objects that need sequential access during analysis like *Element* objects are stored with *IDObjectVector* objects.

3.2.3.3 Mathematical Data Structures

Finite element method is based on a mathematical foundation. Mathematical structures like vectors and matrices, have to be created and manipulated through the solution. Panthalassa handles these data structures on two levels: element and global.

At the element level, small matrices and vectors that hold elemental information are required. To hold such data, Panthalassa uses the Matrix and Vector structures from uBLAS library [36]. These structures holds data in memory in full matrices. Results of elemental operations like stiffness matrix of a finite element are stored in memory by the help of these structures.

At the global level large matrices and vectors have to be used. These matrices and vectors are usually a combination of elemental matrices and vectors (see Cook et al. [21] section 2.5). They are used in the analysis procedure to implement algorithms dealing with the whole finite element model, not just one element. Special data storage models might be needed for different types solution methods. Panthalassa provides this required flexibility by the help of the GlobalMatrix class.

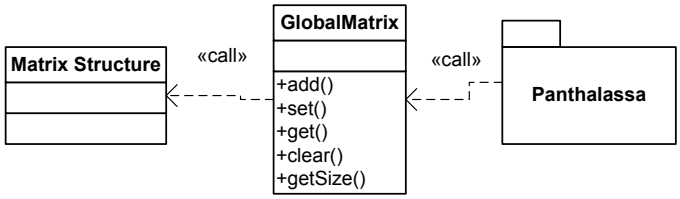


Figure 3.11: GlobalMatrix Class Diagram

GlobalMatrix class is an abstract class, that provides virtual functions for the implementation

of basic interactions with a general matrix structure and Panthalassa library (Figure 3.11). These interactions are, collecting information about the dimensions of a matrix or vector and manipulating the matrix data. Users can bind the matrix and vector from libraries to a global matrix structure by writing a sub-class of the *GlobalMatrix* class. Subclasses override the related methods of *GlobalMatrix* class to bind the functionality of the real implemented matrix structure.

3.2.3.4 ParallelInfo Class

ParallelInfo class is a static class, that provides unique information about parallel state of processes and the analysis phase. Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class (MSDN [37]). In this way, information stored in the *ParallelInfo* class can be used by any object without any instantiation.

In addition to data that define the parallel state of Panthalassa like the total number of processes or the rank of the current process, general information about the analysis like the type of analysis (linear or nonlinear) is stored in the *ParallelInfo* class.

3.2.4 Analysis Classes

3.2.4.1 Analyzer and Algorithm Classes

Panthalassa includes two classes to represent a solution algorithm for a finite element problem: *Analyzer* and *Algorithm* (Figure 3.12). Users can program different solution algorithms to a finite element problem by inheriting these classes. Both of these classes have access to the whole data structure of the finite element model. This means, subclasses of both are able to implement any type of solution algorithm. The difference in two classes appear in their application.

Algorithm class represent the mathematical base of a finite element model solution, whereas *Analyzer* class is used to implement similar functionality needed for different mathematical algorithms. For example, let's have two linear static solution algorithms that differ in only the

factorization process during the solution of equation:

$$[K]\{D\} = \{F\} \tag{3.2}$$

Both algorithms include common processes like the assembly of stiffness matrix $[K]$ and force vector $\{F\}$. These processes are implemented in an Analyzer class to prevent repetitive coding. The dissimilarities between two algorithms are implemented in separate algorithm classes.

Analyzer class executes common tasks between algorithms. These common tasks include but not limited to processes like matrix and vector assembly, time and solution control, output of results to hard disk. Analyzer class has access to the selected Algorithm object and its members to control these actions as well as the finite element model.

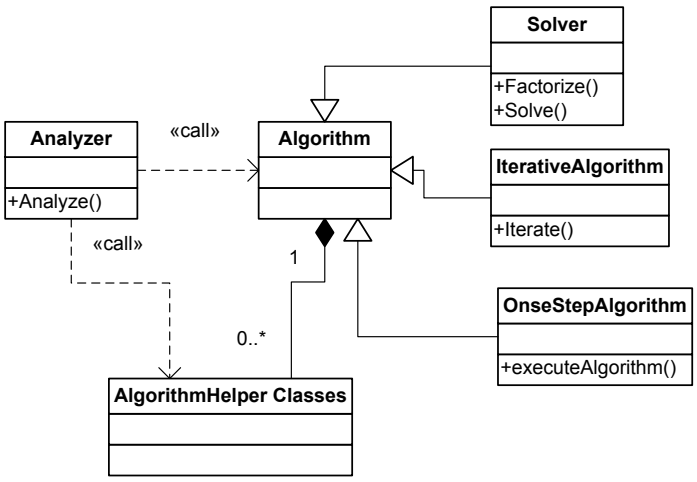


Figure 3.12: Analyzer Algorithm System

Both of the *Analyzer* and the *Algorithm* classes are abstract ones. Inheritors of the *Analyzer* class override the analyze function which is called by the *Domain* object to initiate the analysis. Panthalassa library contains three different subclasses of the *Algorithm* class. These subclasses are abstract classes like the *Algorithm* class, but they have additional virtual functions that can be usable for different algorithm types. Users choose one of these sub-classes or the original *Algorithm* class to inherit and implement their algorithm.

First of these subclasses is the *Solver* class which provides two virtual functions: *Factorize*

and *Solve*. It was implemented specifically for solution of static problems. *Factorize* function is overridden to implement the factorization phase and *Solve* functions is overridden to implement the back substitution phase of a static solution. Second subclass, *IterativeAlgorithm* was implemented for algorithms that are based on an iteration cycle. *Iterate* function of the *IterativeAlgorithm* class is overridden in order to implement a step from the iteration cycle. Last of the subclasses of *Algorithm* class is the *OneStepAlgorithm* class. It was implemented for algorithms that can finish the analysis with a single step. Inheritors of the *OneStepAlgorithm* class override the *executeAlgorithm* function to implement the single step computation.

Algorithm Helper Classes *Algorithm* and *Analyzer* classes are linked through a series of classes called Algorithm Helper classes. These abstract classes represent common tasks required in a solution procedure. Common tasks implemented with these classes, are executed by the Analyzer class. Algorithm class defines which implementation of these classes to be used.

Algorithm Helper classes consist of two classes: *DOFNumberer* and *GlobalMatrixAssembler*.

DOFNumberer class is used to order the system of equations of a finite element model in order to reduce the number operation for solution. A subclass called *NodalDOFNumberer* which, numbers degree of freedoms of the model according to node numbers, is discussed in section 4.3.2.

GlobalMatrixAssembler class is an abstract class used for the assembly of global matrices. Different type of assembly methods can be represent by inheriting this class.

3.2.4.2 TimeTable Class

Timetable class provides methods and members for discretization of time during the solution process of a finite element problem. *TimeTable* class discretized durations of time with structures called *timelines* as presented in Figure 3.13. Every *timeline* holds two variables: *sithe* and *delta*. *Sithe* is the duration of *timeline* and delta is the time difference between steps of discretization. *TimeTable* includes functions that allow to travel through the discretized time.

Algorithm class and *Tracker* class uses *TimeTable* objects to define points in the solution process to execute their implementation.

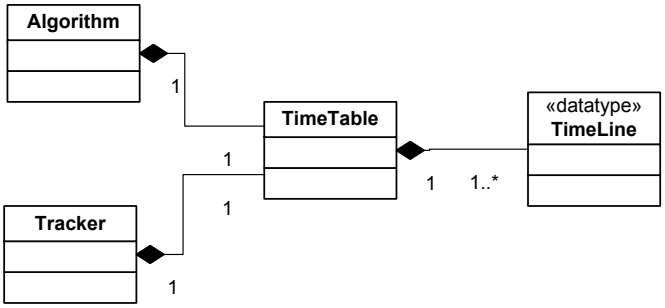


Figure 3.13: TimeTable Class Diagram

3.2.5 Partitioning Classes

3.2.5.1 Domain Decomposition

Domain decomposition is the process of partitioning the components of a finite element model, to be analyzed separately. There are numerous different solution algorithms depending on domain decomposition in literature. Domain decomposition process is generally executed in three steps:

1. Create a graph from finite element model.

2. Partition the graph.

3. Create sub-domains from the partitioned graph.

Panthalassa provides a data structure composed of three classes: *PtlGraph*, *Grapher* and *Partitioner* for the implementation of these steps (Figure 3.14). *PtlGraph* class represents the mathematical structure, graph. *Grapher* is responsible for creating finite element models from graphs and vice verse. *Partitioner* class partitions the graphs represented by *PtlGraph* objects. In addition to these classes a class named *Syncer*, responsible for information transfer between processes is provided.

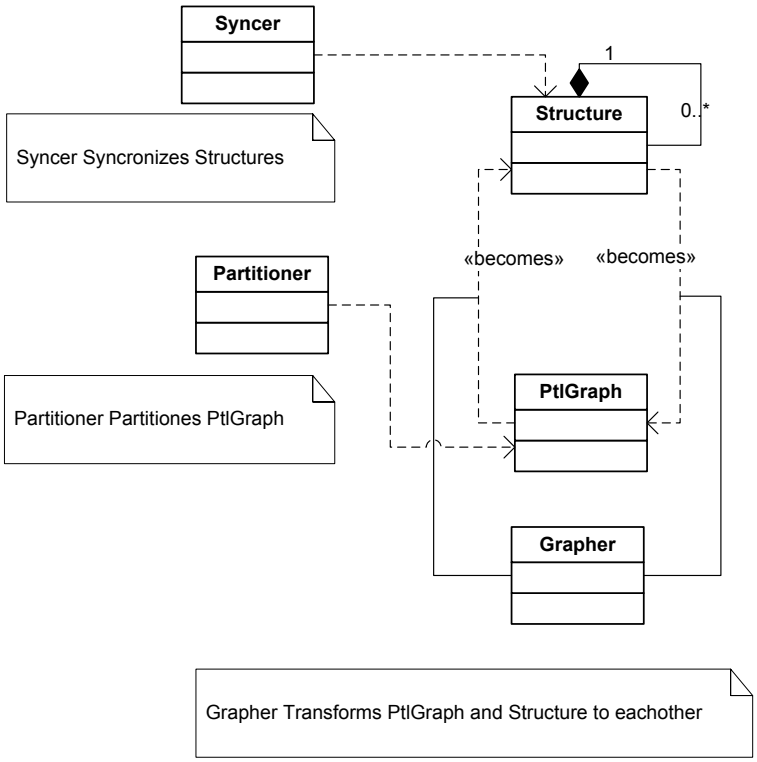


Figure 3.14: Partitioning Classes

3.2.5.2 PtlGraph Class

Any object involving points and connections between them may be called graphs (Gross and Yellen [38]). Figure 3.15 presents an example graph. Intuitively, a graph is a diagram consisting of dots and lines, where each line joins some pair of dots, and two dots may be joined by no lines or any number of lines. Formally dots are called vertices and lines are called edges (Meng et. al. [39]).

Graphs are generally used for representing relations between objects from a collection. In the finite element method, graphs are used to represent relations between elements and nodes of finite element models.

In Panthalassa graphs are represented by the *PtlGraph* class (Figure 3.16). To implement general capabilities of a graph diagram LEMON (Library for Efficient Modeling and Opti-

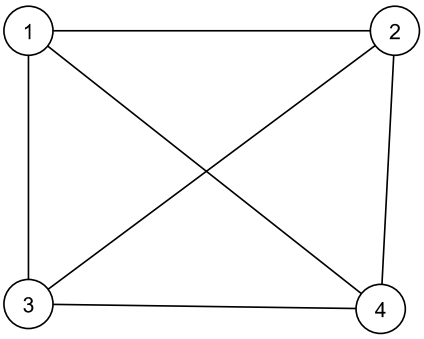


Figure 3.15: Example Graph with Four Vertices and Six Edges

mization in Networks [40]) library was used. LEMON is a general graph library developed in C++ language. Panthalassa uses a specific graph type called *smart_graph* that does not allow manipulation of graph components once the graph is created. Manipulation of graphs are done by creating new graphs. The advantage of *smart_graph* type is, very fast access to the graph components.

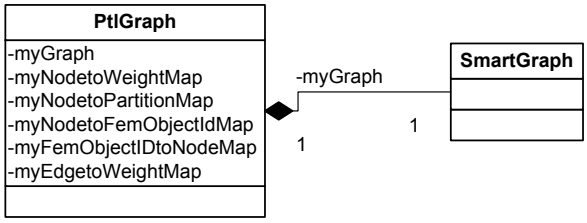


Figure 3.16: PtlGraph Class Diagram

PtlGraph class provides special containers that connect the ids of *FEMObject* classes to vertices and edges of a graph. In this way, any sub-class of the *FEMObject* class can be connected to a vertex or an edge of a graph. Furthermore, vertices and edges can hold integer weights that emphasize the importance of the vertex or edge related to the other vertices or edges of the graph.

3.2.5.3 Grapher Class

Grapher class is responsible for transformations between *PtlGraph* and *Structure* classes. *Grapher* class is an abstract class and should be inherited to provide functionality (Figure 3.17). *Grapher* class provides two virtual functions: *createGraphfromStructure* and *createStructurefromGraph* to handle transformations. Inheritors initialize the attached *Syncer* object by sending the ids of shared vertices (vertices from different *PtlGraph* objects that are connected to the same *FEMObject* object).

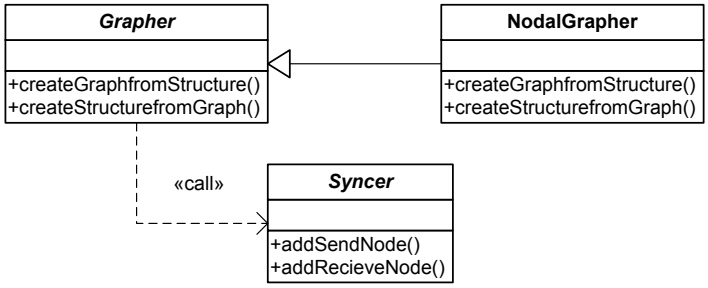


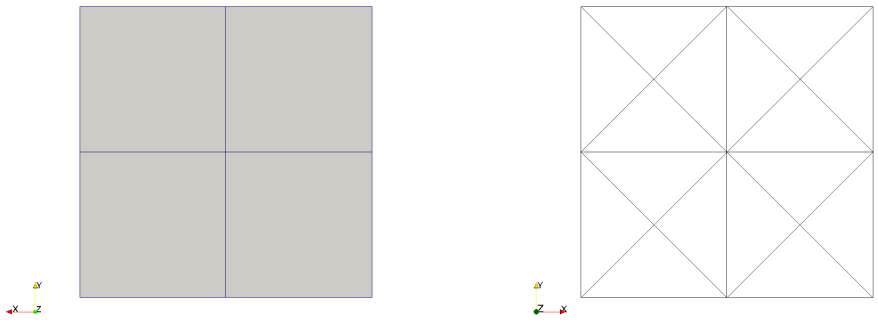
Figure 3.17: Grapher Class Diagram

Panthalassa includes a *Grapher* implementation, *NodalGrapher* class, that utilizes nodal graphs to represent finite element models. A nodal graph represents the nodes of a finite element model with vertices and the finite elements with edges (Figure 3.18).

3.2.5.4 Partitioner Class

Partitioner class is responsible for partitioning of graphs. *Partitioner* class is an abstract class. Inheritor classes take the graph to be partitioned and the number of partitions as parameters, and partitions the graph accordingly by overriding the virtual *partition* function (Figure 3.19). At the end of the partitioning process partition numbers are assigned to the vertices of the graph.

Panthalassa includes a partitioner class called *ParMetisPartitioner* that uses the ParMetis library (Parallel Graph Partitioning and Sparse Matrix Ordering Library [41]) for partitioning



(a) Structure (b) Nodal Graph

Figure 3.18: An Arbitrary Structure and Its Nodal Graph

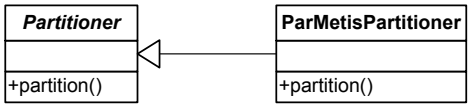


Figure 3.19: Partitioner Class Diagram

of graphs. ParMetis is an MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs. The algorithms in ParMetis library are based on the multilevel partitioning and fill-reducing ordering algorithms that are implemented in the serial graph partitioning library METIS (Karypis and Kumar [42]). The algorithms in METIS are based on multilevel graph partitioning described in Karypis and Kumar [43], [44] and [45] (Karypis et. al. [46]).

3.2.5.5 Syncer Class

Syncer class holds the shared vertices between *PtlGraph* objects and provides ways to transfer necessary information between processors. *Syncer* is provided as an abstract class that only holds shared vertices. Inheritors provide the functionality of transferring information.

Panthalassa provides a subclass called *AdditiveSyncer* to be used by Algorithm objects (Syncer

3.20). *AdditiveSyncer* class synchronizes information between the nodes of different sub-structures by adding up the elements of a *GlobalMatrix* object. *Algorithms* can use this class to synchronize displacements, velocities or accelerations between sub-structures.

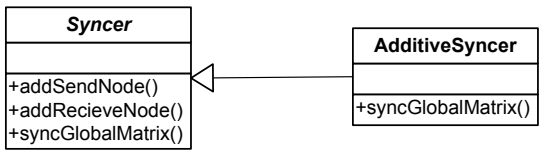


Figure 3.20: Syncer Class Diagram

3.2.6 Input and Output Classes

Panthalassa uses hard disk files to read the information of a finite element model and write the results of the finite element solution. *Tracker* and *ModelBuilder* classes are utilized for this purpose.

3.2.6.1 Tracker Class

Tracker class is an abstract class used to write output to files. *Track* method of the class is called by the *Analyzer* object at every step of the analysis phase. *Tracker* class has access to every component of the model and analysis, thus can gather every piece of information in the memory. Low level interaction with files is left to the implementors.

Panthalassa does not save information between steps of the analysis, so relevant information to the user must be written to hard disk at every step. *Tracker* class has access to a *TimeTable* object. By winding this *TimeTable* object, user can choose at which steps information will be written.

Panthalassa includes several implementations of the *Tracker* class that track elemental and nodal information during analysis and users can implement their unique tracker implementation by inheriting the *Tracker* class.

3.2.6.2 ModelBuilder Class

As default, Panthalassa reads user input based on the Ptl language, but input files of any format can be read and used to create the finite element problem. *ModelBuilder* class is an abstract class that is inherited to implement this feature.

Subclasses of *ModelBuilder* class override the *BuildModel* function, in order to create the finite element problem using the data structures explained in this section.

3.2.7 Plug-in Architecture: Pugg Library

3.2.7.1 Introduction

Pugg is a c++ library for automatic loading and management of plug-ins. Pugg was developed exclusively for Panthalassa, however in time it became an open-source project and has been used by many developers around the world. In this section main concepts behind Pugg framework are explained. Detailed examples about usage of the library can be found on Pugg website [47].

Pugg enforces a certain procedure for implementing plug-ins. Plug-ins developed according to this procedure can be automatically loaded at run time by Pugg. Pugg looks for libraries including a special function called *registerPlugin* and loads the library according to the special initialization code defined in this function.

In Pugg's framework system, a plug-in consists of classes that inherit from superclasses defined in the main application. These sub-classes are mapped to string identifiers. Main application uses these identifiers to create objects of the subclasses and add functionality to itself. Panthalassa uses this string-class mapping system to load specific classes that are defined in the user input file.

Panthalassa allow many super-classes to be inherited with plug-ins. List of classes that has plug-in support is given in Table 3.3.

Table 3.3: List of classes that has plug-in support in Panthalassa

Class Type
Element
Analyzer
Tracker
Grapher
Algorithm
MaterialModel
Partitioner
ModelBuilder

3.2.7.2 Server Driver System

Pugg is capable of loading more than one type of class from a single plug-in library. To manage different types of classes Pugg uses classes called *Server*. Main application creates a *Server* object for every superclass type that supports a plug-in system.

Subclasses defined in plug-ins are hosted by *servers* through object factories. An object factory is a class that instantiates another class at run time. An extensive discussion about object factories can be found in Alexandrescu [48]. Pugg uses the early mentioned string mapping system to gather the intended *Driver* object from server and uses this *Driver* object to create an instance of the associated sub-class.

Figure 3.21 presents an example to the server driver system described above. In this example, *2DMembrane* class is binded a *Server* object through *2DMembraneDriver* class. In order to gather a pointer to a new instance of the *2DMembrane* class, first *getDriver* function of the *ElementServer* class is called and a pointer to the *2DMembraneDriver* object is gathered. In this process, the name associated with the *2DMembraneDriver* class is used as parameter. Then, *createElement* function of the *2dMembraneDriver* is called to gather a pointer to a new instance of the *2DMembrane* object.

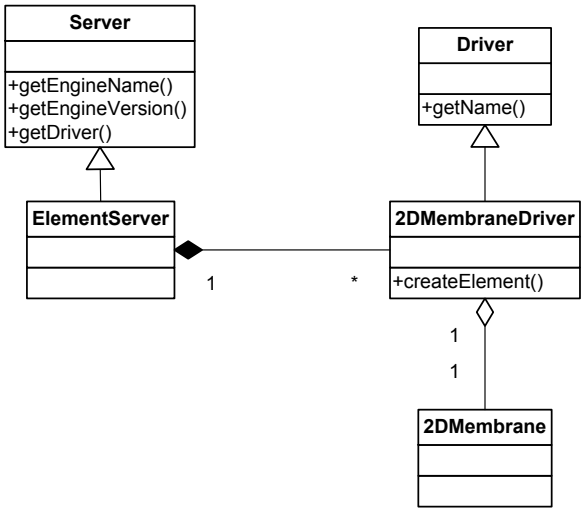


Figure 3.21: Server Driver System Example

3.2.7.3 Object Oriented Design of Pugg Library

Pugg consists of four classes: *Kernel*, *Plugin*, *Server* and *Driver* (Figure 3.22). *Kernel* class is the management class of the library. It stores instances of the Plug-in and *Server* classes. It has functions to automatically load subclasses from plug-in files. Main application creates an instance of *Kernel* class and uses it to load and control plug-in libraries.

Plugin class represents a plug-in file. It is responsible for loading and initialization of the plug-in libraries. *Plugin* class uses low level Windows functions (*LoadLibrary* and *FreeLibrary* functions MSDN [49]) for this purpose.

Remaining two classes: *Server* and *Driver* are used to implement the server driver system described in Section 3.2.7.2.

3.3 Parallel Execution

Execution timeline of a parallel program differs significantly from a sequential one. A parallel program has to run with more than one processor at a time. This necessity requires special

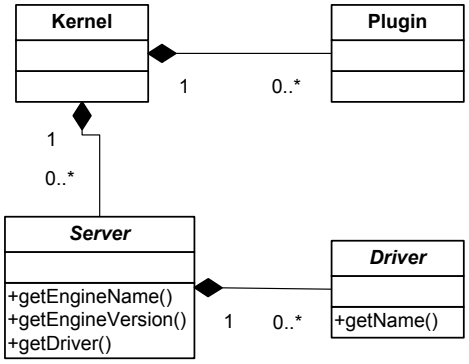


Figure 3.22: Pugg Class Diagram

initialization and finalization code inserted into program code. In addition, information stored in memory have to be shared, or transferred between processes during execution.

As a parallel program itself, lifeline of Panthalassa was designed to deal with above mentioned problems and can be studied in four phases: initialization, model creation, analysis and finalization (Figure 3.23).

Initialization of Panthalassa starts with a step called process spawning. Process spawning is the creation of several processes from an application. MPICH2 has a special application called MPIEXEC for this purpose. MPIEXEC is a command line application that gets the properties of the spawning process (name of the application's executable file, number of processes to spawn etc.) as parameters.

MPICH2 not only creates and starts the processes, but it opens a channel between them as well. A channel is a software technology, used for interprocess communication. Processes, executed by different processors, exchange information using channels.

For every processor that is to be used in calculations, a copy of the Panthalassa process must be spawned. Panthalassa executes special code to initialize the channel between processes. Every process gathers an id called rank from MPICH2. Ranks are used by processes while communicating with each other. Process with rank zero is called the master and other processes are called slaves. Master process of Panthalassa ends the initialization phase by reading the user input from a file and sending it to slaves for execution.

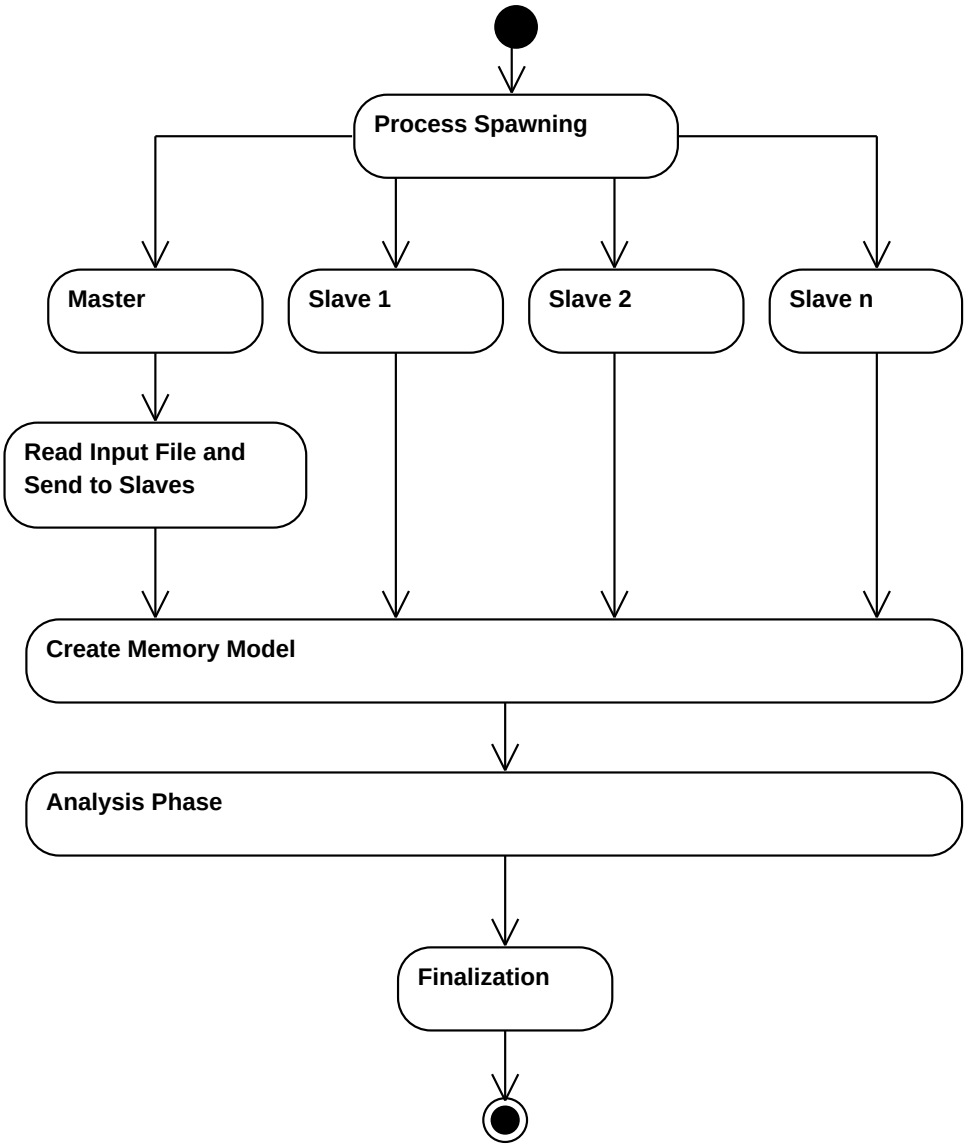


Figure 3.23: Panthalassa Lifeline

After the initialization phase, Panthalassa starts creating data structures that represents the finite element problem in memory. Two approaches can be used for this purpose; either the whole data model is created in memory on all processes, or every process creates a small part of the model. Panthalassa uses the first approach. Creating whole model in memory on every process eliminates any communication requirement before solution phase. Every process has a copy of every data structure, thus no data transferring is necessary along processes. Implementing algorithms is vastly simplified with this type of data storage model. Disadvantage of

this approach is that it increases memory requirements. If more than one process is spawned on a single computer, memory of the system is filled with copies of the same data.

After the finite element problem is created in memory, Panthalassa starts the analysis phase. In the analysis phase algorithms created by the user through plug-ins are executed. These algorithms are implemented through several classes, examined in section 3.2.4. Parallelization of these algorithms are left to the implementor.

Finally Panthalassa closes the communication channel between processes and frees the memory occupied by the internal data structure in the finalization phase.

CHAPTER 4

Parallel Implementation of Linear Dynamic Analysis for Soil-Structure Interaction

4.1 Introduction

Solution of SSI problems using the finite element method, has to overcome two main difficulties: Dynamic solution of a large domain and the mathematical representation of the soil material. Dynamic nature of the SSI problems requires the fundamental dynamic equilibrium equation (Equation 4.1) to be solved. This second order differential equation can be solved by numeric integration methods such as central difference, trapezoidal rule, Newmark methods. In this study implicit Newmark method (Newmark [50] and Wilson [51]) was employed for the solution of Equation 4.1.

$$[M]\{\ddot{D}\} + [C]\{\dot{D}\} + \{R^{int}\} = \{R^{ext}\} \tag{4.1}$$

In the process of solving SSI problems, modeling soil material behavior under cyclic loading conditions possesses great importance. Nonlinear material behavior of soil must be approximated to a reasonable degree in order to attain realistic and accurate solutions. In this study the equivalent linear model (Seed and Idriss [52]), which approximates nonlinear material behavior parameters with linear approximations, was utilized.

Modeling of soil often results in large models because large geographies must be modeled and restrictions to the mesh size must be enforced for successful propagation of waves through soil material (Lysmer and Kuhlemeyer [23]). In order to solve large-scale SSI problems parallel algorithms that utilizes multi-processor technologies offer advantages in terms of speed and memory capacity. Because of this reason, the parallel versions of the linear dynamic solution

for the SSI problems was implemented in this study.

4.2 Theory

4.2.1 Implicit Newmark Method

In the fundamental dynamic equilibrium equation (Equation 4.1) $\{\ddot{D}\}$ and $\{\dot{D}\}$ vectors represent acceleration and velocity of the system, respectively; $[M]$ is the mass matrix and $[C]$ is the damping matrix of the system. Internal and external forces are represented by vectors $\{R^{int}\}$ and $\{R^{ext}\}$. Considering linear analysis, internal forces of the system can be expressed further as the multiplication of stiffness matrix $[K]$ and displacement vector $\{D\}$:

$$\{R^{int}\} = [K]\{D\} \tag{4.2}$$

Implicit Newmark algorithm solves Equation 4.1 by making an assumption for acceleration over a time step. Acceleration value for time = τ (τ is a value of time between a typical time step $\Delta t = t_{n+1} - t_n$) for average and linear acceleration assumptions are

$$\ddot{u}(\tau) = \frac{1}{2}(\ddot{u}_{n+1} + \ddot{u}_{n+1}) \tag{4.3}$$

$$\ddot{u}(\tau) = \frac{\tau}{\Delta t}(\ddot{u}_{n+1} - \ddot{u}_{n+1}) \tag{4.4}$$

Velocities and displacements are computed at time step n+1 by equating τ to Δt and integrating above equations with initial conditions $\dot{u}(\tau) = \dot{u}_n$ at $\tau = 0$ and $u(\tau) = u_n$ at $\tau = 0$ (Table 4.1).

Table 4.1: Velocity and Displacement equations for a single degree of freedom system (Implicit Newmark Method).

Average Acceleration	Linear Acceleration
$\dot{u}_{n+1} = \dot{u}_n + \frac{1}{2}\Delta t(\ddot{u}_{n+1} + \ddot{u}_n)$	$\dot{u}_{n+1} = \dot{u}_n + \frac{1}{2}\Delta t(\ddot{u}_{n+1} + \ddot{u}_n)$
$u_{n+1} = u_n + \Delta t\dot{u}_n + \frac{1}{4}\Delta t^2(\ddot{u}_{n+1} + \ddot{u}_n)$	$u_{n+1} = u_n + \Delta t\dot{u}_n + \Delta t^2(\frac{1}{6}\ddot{u}_{n+1} + \frac{1}{3}\ddot{u}_n)$

These equations are implicit as they depend on information from step n+1 ($\ddot{u}_{n+1}$). Equations presented at Table 4.1 can be generalized for MDOF systems in the following way:

$$\{\dot{D}\}_{n+1} = \{\dot{D}\}_n + \Delta t\left[\gamma\{\ddot{D}\}_{n+1} + (1-\gamma)\{\ddot{D}\}_n\right] \tag{4.5}$$

$$\{D\}_{n+1} = \{D\}_n + \Delta t\{\dot{D}\}_n + \frac{1}{2}\Delta t^2\left[2\beta\{\ddot{D}\}_{n+1} + (1-2\beta)\{\ddot{D}\}_n\right] \tag{4.6}$$

Numerical factors γ and β control the characteristics of the algorithm.Average acceleration and linear acceleration assumptions can be achieved by setting $\gamma = \frac{1}{2}$,$\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$,$\beta = \frac{1}{6}$ respectively.

By solving Equation 4.6 for $\{\ddot{D}\}_{n+1}$ then substituting this expression into Equation 4.5, the following equations were obtained:

$$\{\ddot{D}\}_{n+1} = \frac{1}{\beta\Delta t^2}\left(\{D\}_{n+1} - \{D\}_n - \Delta t\{\dot{D}\}_n\right) - \left(\frac{1}{2\beta} - 1\right)\{\ddot{D}\}_n \tag{4.7}$$

$$\{\dot{D}\}_{n+1} = \frac{\gamma}{\beta\Delta t}\left(\{D\}_{n+1} - \{D\}_n\right) - \left(\frac{\gamma}{\beta} - 1\right)\{\dot{D}\}_n - \Delta t\left(\frac{\gamma}{2\beta} - 1\right)\{\ddot{D}\}_n \tag{4.8}$$

These equations are then inserted into the fundamental equation of motion and solved for $\{D\}_{n+1}$. This gives the fundamental equation for implicit Newmark methods presented below:

$$\begin{aligned}[K^{eff}]\{D\}_{n+1} = \{R^{ext}\}_{n+1} &+ [M]\left\{\frac{1}{\beta\Delta t^2}\{D\}_n + \frac{1}{\beta\Delta t}\{\dot{D}\}_n + \left(\frac{1}{2\beta} - 1\right)\{\ddot{D}\}_n\right\} \\ &+ [C]\left\{\frac{\gamma}{\beta\Delta t}\{D\}_n + \left(\frac{\gamma}{\beta} - 1\right)\{\dot{D}\}_n + \Delta t\left(\frac{\gamma}{2\beta} - 1\right)\{\ddot{D}\}_n\right\}\end{aligned} \tag{4.9}$$

where

$$[K^{eff}] = \frac{1}{\beta\Delta t^2}[M] + \frac{\gamma}{\beta\Delta t}[C] + [K] \tag{4.10}$$

$[K^{eff}]$ cannot be a diagonal matrix as it contains $[K]$. Thus, a factorization process is required to solve equation 4.9. It can be shown that implicit Newmark method is unconditionally stable (Hughes [53]) when,

$$2\beta \geq \gamma \geq \frac{1}{2} \tag{4.11}$$

Unconditionally stable algorithms do not diverge no matter how large the time step Δt is, thus allows obtaining the solution with less times steps when compared to conditionally stable algorithms. It must be noted that a larger Δt increases the error in calculations.

4.2.2 Finite Elements

There are two types of finite elements used in this study: bilinear quadrilateral and linear hexahedron. Bilinear quadrilateral is an isoparametric 2-D plane element (Figure 4.1) and

linear hexahedron (Figure 4.2) is an isoparametric 3-D element. Shape functions of these finite elements are presented in tables 4.2 and 4.3.

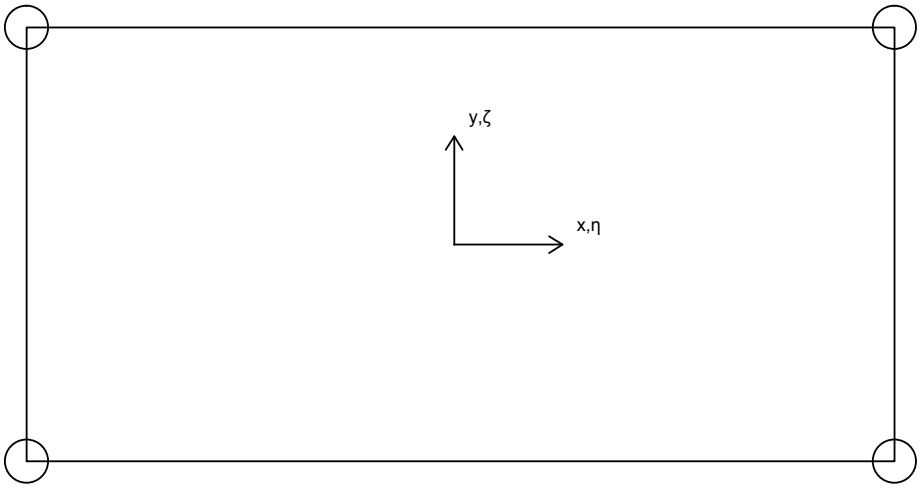


Figure 4.1: Bilinear Quadrilateral

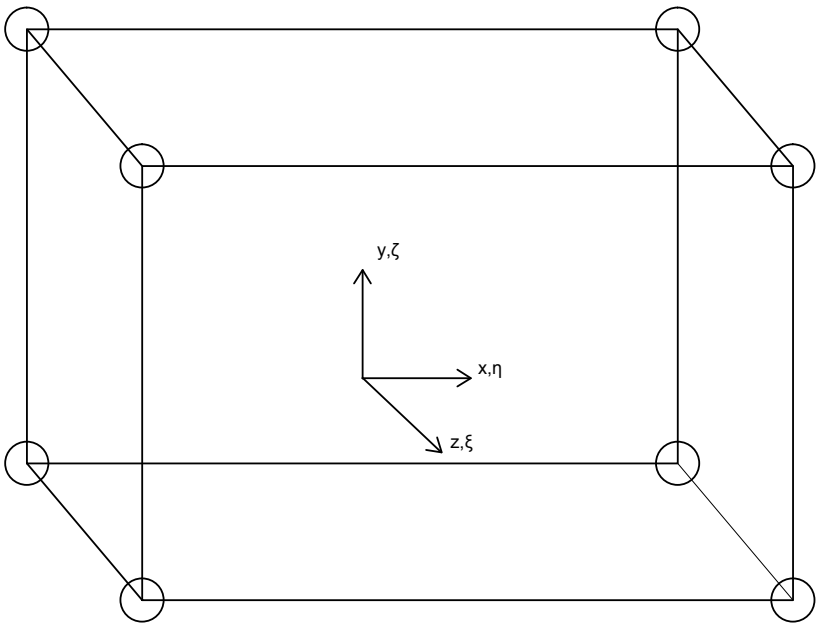


Figure 4.2: Linear Hexahedron

Table 4.2: Shape Functions for the Bilinear Quadrilateral

Shape Function
$N_1 = \frac{1}{4}(1-\zeta)(1-\eta)$
$N_2 = \frac{1}{4}(1+\zeta)(1-\eta)$
$N_3 = \frac{1}{4}(1+\zeta)(1+\eta)$
$N_4 = \frac{1}{4}(1-\zeta)(1+\eta)$

Table 4.3: Shape Functions for the Linear Hexahedron

Shape Function
$N_1 = \frac{1}{8}(1-\zeta)(1-\eta)(1-\xi)$
$N_2 = \frac{1}{8}(1+\zeta)(1-\eta)(1-\xi)$
$N_3 = \frac{1}{8}(1-\zeta)(1+\eta)(1-\xi)$
$N_4 = \frac{1}{8}(1+\zeta)(1+\eta)(1-\xi)$
$N_5 = \frac{1}{8}(1-\zeta)(1-\eta)(1+\xi)$
$N_6 = \frac{1}{8}(1+\zeta)(1-\eta)(1+\xi)$
$N_7 = \frac{1}{8}(1-\zeta)(1+\eta)(1+\xi)$
$N_8 = \frac{1}{8}(1+\zeta)(1+\eta)(1+\xi)$

Elemental stiffness matrix $[K]_e$ and elemental mass matrix $[M]_e$ are computed with the following equations:

$$[K]_e = \int [B]^T [E][B] dV \tag{4.12}$$

$$[M]_e = \int \rho [N]^T [N] dV \tag{4.13}$$

In these equations, ρ is density, $[E]$ is the constitutive matrix that defines the relationship between strains and stresses of the element; $[N]$ is the shape function matrix of the finite element and $[B]$ matrix defines the relationship between displacements and strains of the element. $[B]$ matrix is computed by differentiating the $[N]$ matrix. Bilinear quadrilateral element is a plane strain element that uses the constitutive matrix given in Equation 4.14. On the other hand, linear hexahedron element uses general 3D stress strain relationship given in Equation 4.15. In these equations E and v represent Young's Modulus and Poisson's Ratio respectively.

$$[E] = \frac{E}{(1+v)(1-2v)} \begin{bmatrix} 1-v & v & 0 \\ v & 1-v & 0 \\ 0 & 0 & \frac{1-2v}{2} \end{bmatrix} \tag{4.14}$$

$$[E] = \frac{E}{(1+v)} \begin{bmatrix} 1-v & v & v & 0 & 0 & 0 \\ v & 1-v & v & 0 & 0 & 0 \\ v & v & 1-v & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}-v & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}-v & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}-v \end{bmatrix} \tag{4.15}$$

4.2.3 Boundary Conditions

Boundary conditions are applied to Equation 4.9 by two different methods. Boundary conditions that are equal to zero are totally omitted from vectors and matrices. Boundary conditions that change with time (i.e. an earthquake) are added to the system as constraints using the Lagrange Multipliers Method.

Lagrange Multipliers Method is used to define constraints between DOFs of the system. Every constraint is defined by introducing an extra row and column to the solution system. For example, equality between the first and the second DOFs of a three DOF static system can be achieved as:

$$u_1 - u_2 = 0 \tag{4.16}$$

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 1 \\ k_{2,1} & k_{2,2} & k_{2,3} & -1 \\ k_{3,1} & k_{3,2} & k_{3,3} & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ \lambda \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \\ 0 \end{Bmatrix} \tag{4.17}$$

λ can be interpreted as the force of the applied constraint (Cook et al. [21]). In the same way, fixed displacements d_1 and d_2 can be applied to the system as:

$$\begin{aligned} u_1 &= d_1 \\ u_2 &= d_2 \end{aligned} \tag{4.18}$$

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 1 & 0 \\ k_{2,1} & k_{2,2} & k_{2,3} & 0 & 1 \\ k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ \lambda_1 \\ \lambda_2 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \\ F_3 \\ d_1 \\ d_2 \end{Bmatrix} \tag{4.19}$$

Although Lagrange Multipliers Method slows down the solution as it increases the number of equations that has to be solved, it makes up for it by speeding up the assembly of displacement constraints as only the force vector has to be assembled at every time step.

4.2.4 Equivalent Linear Soil Model

Soil exhibits nonlinear material behavior, even when subjected to small strains. This behavior escalates in large, cyclic strain situations especially on soft soils (Beresnev and Kuo-liang [54]). Different numerical models were proposed to represent this behavior of soil in literature and the equivalent linear method, which was efficient and simple, was utilized in this study.

Equivalent linear method is based on linearization of nonlinear material characteristics of soil. Under cyclic loading, stress-strain behavior of soil material can be illustrated with a hysteresis loop as presented in Figure 4.3. As seen from the figure, tangent shear modulus, G_{tan}, takes different values at every point on the shear strain axis. On the contrary, secant shear modulus, G_{sec} is constant and equal to

$$G_{sec} = \frac{\tau_c}{\gamma_c} \tag{4.20}$$

where τ_c and γ_c represent the maximum values of shear stress and strain of the loop. G_{sec} represents a linear approximation for G_{tan}.

Damping of the soil, represented as the area enclosed by the hysteresis loop, can be described by the damping ratio (Kramer [55]) in the following way:

$$\lambda = \frac{W_D}{4\pi W_S} = \frac{1}{2\pi} \frac{A_{loop}}{G_{sec}\gamma_c^2} \tag{4.21}$$

In the above equation, W_D is the dissipated energy; W_S is the maximum strain energy and A_{loop} is the area enclosed by the hysteresis loop. Parameters G_{sec} and λ, known as equivalent linear parameters, can be used to describe the behavior of soil material.

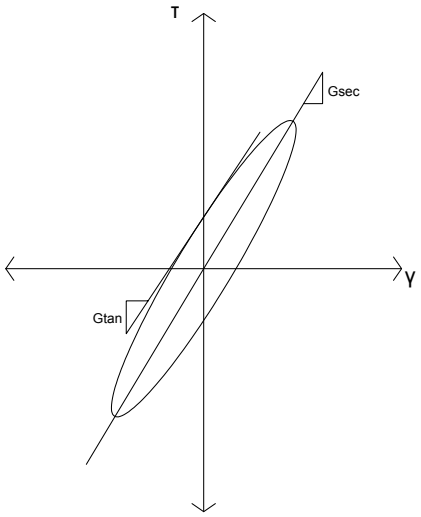


Figure 4.3: Hysteresis Loop and Secant and Tangent Shear Modulus

Equivalent linear parameters for different shear strain levels, was estimated for different types of soil by different researchers in literature (Vucetic and Dobry [56], Ishibashi and Zhang [57]). These estimations are generally depicted by two graphs, showing the variations of shear modulus and damping ratio against shear strain. Variation of shear modulus is represented by the variation of $\frac{G_{sec}}{G_{max}}$ against shear strain, and known as the modulus reduction curve. G_{max}, the maximum shear modulus, is the shear modulus of material at small strain situations. Equivalent linear parameter curves are dependent on plasticity index (PI) for clays (Vucetic and Dobry [56]) and confining shear stress for sands (Seed and Idriss [52]). Figures 4.4 and 4.5 presents equivalent linear parameter curves for soils with different PI.

It must be noted that, this soil model, only takes elastic strains into account; plastic deformations are ignored. Thus, this soil model can only be utilized for cases where plastic deformations do not hold big importance. Although some of the equivalent linear material curves in literature, took plasticity of soil material into account (Vucetic and Dobry [56]); these curves defined the effects of plasticity on equivalent linear parameters but not yield points or surfaces outlining non-recoverable strain levels.

Solution of the fundamental equation of Implicit Newmark Method (Equation 4.9) requires the assembly of $[K^{eff}]$, $[M]$ and $[C]$ matrices. These global matrices are computed from

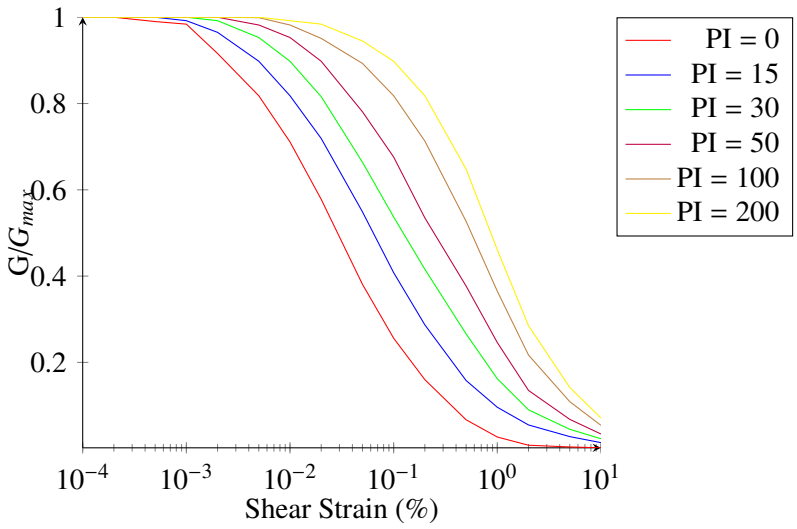


Figure 4.4: Shear Modulus Reduction Curves for Soils with Different PI (Vucetic and Dobry [56])

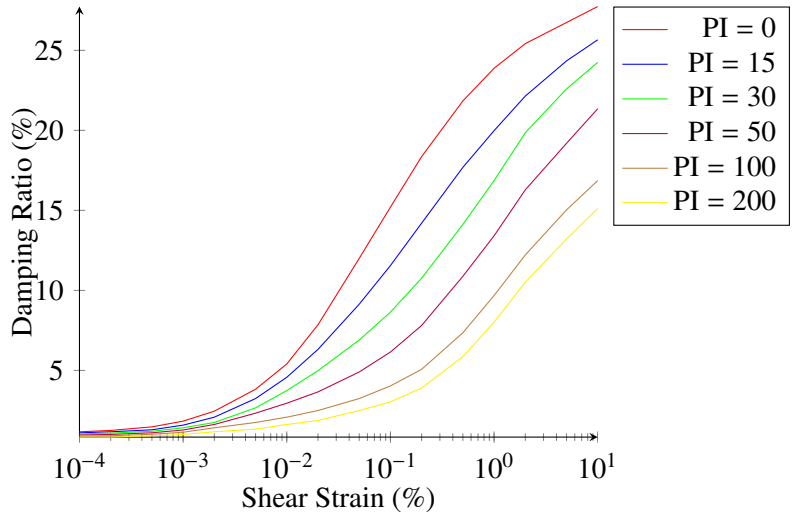


Figure 4.5: Damping Curves for Soils with Different PI (Vucetic and Dobry [56])

elemental matrices:

$$[K^{eff}] = \sum \frac{1}{\beta \Delta t^2}[M]_e + \frac{\gamma}{\beta \Delta t}[C]_e + [K]_e \tag{4.22}$$

$$[C] = \sum [C]_e = \sum \alpha [M]_e + \beta [K]_e \tag{4.23}$$

$$[M] = \sum [M]_e \tag{4.24}$$

Constants α and β of equation 4.23 are known as Rayleigh damping constants and gathered from the *MaterialModel* class associated with the element. In the Rayleigh damping formulation damping matrix is computed by combining mass and stiffness matrices. α and β coefficients determine contributions from mass and stiffness matrices respectively. Rayleigh damping coefficients are computed in order to fix the material's critical damping values at the frequencies specified by the user (Figure 4.6), using equations 4.25.

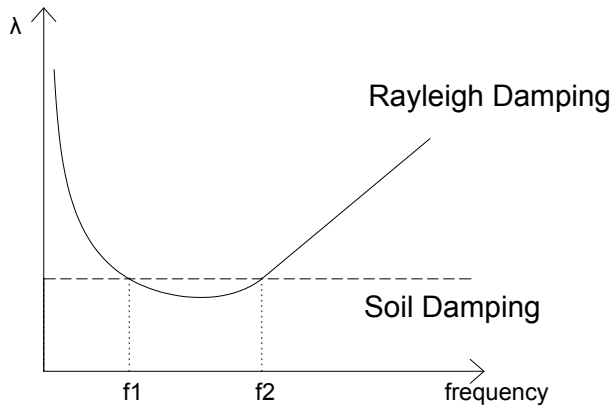


Figure 4.6: Rayleigh Damping vs Frequency Independent Damping Behavior

$$\begin{aligned} \alpha &= \frac{2\lambda w_1 w_2}{w_1 + w_2} \\ \beta &= \frac{2\lambda}{w_1 + w_2} \end{aligned} \tag{4.25}$$

Rayleigh damping formulation overdamps frequencies smaller than f1 and frequencies larger than f2 while it underdamps frequencies between two frequencies. Exact soil damping behavior is only experiences at frequencies f1 and f2. Definition of these two frequencies have extreme influence on analysis results. A discussion on determining the frequencies will be held on chapter 5.

4.3 Implementation

4.3.1 Analysis

Implementation of the Implicit Newmark method, adhered the described methodology in section 3.2.4, analysis classes. Analyzer and Algorithm classes were inherited and added on Panthalassa using the plug-in architecture. Furthermore an algorithm helper class called NodalDOFNumberer was created and binded to the algorithm class to be used in numbering DOFs in the model.

ImplicitNewmark class, an inheritor of the Algorithm class, executes the numerical computations, described in last section, in order to solve the general dynamic equilibrium equation (equation 4.1). LinearDynamicAnalyzer class, inheritor of the Analyzer class, creates a loop dictated by the user and ImplicitNewmark class repeatedly while execution other ancillary tasks.

4.3.1.1 ImplicitNewmark Class

Time stepping solution prescribed with the Implicit Newmark Method, was implemented via the ImplicitNewmark class. Panthalassa requires all iterative and time stepping algorithms, to be inherited from the IterativeAlgorithm class (section 3.2.4.1). Iterate function of the IterativeAlgorithm class, is called by the analyzer object at every step of the analysis. *ImplicitNewmark* class overrides the *Iterate* function, to compute a step from Equation 4.9. At every time step equation 4.9 is constructed in memory and solved by the parallel sparse symmetric solver library MUMPS (MUltifrontal Massively Parallel Sparse direct Solver [58]). In addition to the Iterate function, *Init* function of the *Algorithm* class is also overridden to execute initialization tasks for the implementation.

Initialization code defined in the Init function, performs two important steps. First, it instantiates the NodalDOFNumberer class, discussed later in this section, that is used to number DOFs of the finite element model. Instantiated NodalDOFNumberer object is used by the analyzer in terms with the Analyzer- Algorithm-AlgorithmHelper methodology discussed in section 3.2.4.1. Second, it creates the data structures, vectors and matrices, used to hold members of the implicit Newmark method in memory. For this reason, ImplicitNewmark

utilizes vectors and coordinate sparse matrices from uBLAS library. Sparse matrices hold only the non-zero values in memory; thus they decrease the required memory space. Global matrices $[K^{eff}]$, $[M]$ and $[C]$ are hold as sparse matrices to take advantage of this special memory schema of sparse matrices. Vectors that define forces, displacements, velocities and accelerations of the system are held by vectors.

Table 4.4: Matrices and Vectors used in the Implementation of Implicit Newmark Method

Matrix or Vector	Definition
$[K^{eff}]$	Effective Stiffness Matrix (equation 4.10)
$[M]$	Global Mass Matrix
$[C]$	Global Damping Matrix
$\{F\}$	External Load Vector
$\{D_n\}$	Displacement Vector at time step = n
$\{D_{n-1}\}$	Displacement Vector at time step = n - 1
$\{V_n\}$	Velocity Vector at time step = n
$\{V_{n-1}\}$	Velocity Vector at time step = n - 1
$\{A_n\}$	Acceleration Vector at time step = n
$\{A_{n-1}\}$	Acceleration Vector at time step = n - 1

At every time step, Analyzer object calls the Iterate function of *ImplicitNewmark* class. *ImplicitNewmark* class checks the *ParallelInfo* (section 3.2.3.4) object to determine if the assembly of matrices and vectors is necessary. If only linear material models are defined in the system, assembly process is executed only once at the beginning of analysis; whereas if equivalent linear or nonlinear material models are present in the model assembly of equations are necessary at each time step.

To take advantage of parallelism, every processor creates only a part of the global matrices, $[K^{eff}]$, $[M]$ and $[C]$. Parallel assembly of global matrices among processors is initiated by assigning equal numbers of elements to each processor. Then, each computer simultaneously assembles their assigned portions of global matrices dictated by their assigned elements. The replicated degrees of freedoms that exist at the portions of the stiffness matrix are assembled during the solution without the need of additional communication. Thus, such an assembly approach requires no communication during assembly and creates a linear speed up.

After the assembly of global matrices, right hand side of Equation 4.9 is calculated: As $[K^{eff}]$, $[M]$ and $[D]$ are distributed among processes, calculated $\{R^{eff}\}$ vector is also dis-

tributed among processes. In order to solve Equation 4.9, MUMPS library requires the $\{R^{eff}\}$ vector to be held in master processor fully. Thus, as a next step, $\{R^{eff}\}$ vector from every process is transformed into a full $\{R^{eff}\}$ vector held in the master process using the *MPI_Reduce* function [59]. *MPI_Reduce* function gathers vectors from all processes, sums them up and send the result to the master. $[K^{eff}]$ is then factorized and solved for $\{R^{eff}\}$ to calculate displacements $\{D\}$ using the multifrontal solver of MUMPS library.

MUMPS is a software package for solving systems of linear equations by utilizing multi-frontal approach. Multi-frontal methods simultaneously perform computations on multiple independent fronts which are obtained by the sequence of partial factorizations. These fronts are named as frontal matrices and factorized by highly optimized dense matrix solvers which significantly improve the performance of multi-frontal solvers. Depending whether the matrix is symmetric or not, LU or LDL^T type solution method is utilized in MUMPS . For the positive definite symmetric matrices, the solution is performed in three main steps: analysis, factorization, and solution.

During the analysis step, first stiffness matrix equations are ordered with various ordering algorithms such as AMD (Amestoy et.al. [60]), QAMD (Amestoy [61]), AMF (an approximate minimum fill-in ordering), PORD (Schulze [62]), METIS (Karypis and Kumar [42]) and symbolic factorization is performed. It is also possible to have MUMPS choose the type of ordering method for the given matrix. Among these several ordering algorithms, the nested-dissection algorithm METIS library usually outperformed the other ordering algorithms for the matrices tested in this study and hence METIS was utilized during the solution of the structural models. As the symbolic factorization is finalized, its results are sent to other processors from the master processor and the factorization phase initiates. The computations during the analysis step are performed on a single computer.

In multi-frontal methods, the parallel factorization sequence is described by the elimination tree which is obtained during equation ordering. Based on this elimination tree, dense frontal matrices are created simultaneously and factorization of such matrices is performed by utilizing the dense matrix solvers of ScaLAPACK (Netlib [63]) library. Once the factorizations ends, solution step initiates by broadcasting of the right hand side (the force matrix) from master computer to others where the forward and back substitutions are computed utilizing the distributed factors. As the displacements are obtained, they are collected at the master

computer.

Mumps gives back the displacements only to the master processor. In order to calculate velocities and accelerations at every process displacements are distributed to the slaves from the master process. Velocities and accelerations are then calculated using Equations 4.8 and 4.7 respectively. To prepare for the next step calculated displacements, velocities and accelerations are copied to the vectors representing values for the last step:

$$\{D_{n-1}\} = \{D_n\} \tag{4.26}$$

$$\{V_{n-1}\} = \{V_n\} \tag{4.27}$$

$$\{A_{n-1}\} = \{A_n\} \tag{4.28}$$

4.3.1.2 NodalDOFNumberer Class

Task of numbering DOFs of the finite element model is accomplished by the *NodalDOFNumberer* class. *NodalDOFNumberer* class, a sub-class of the *DOFNumberer* class, gives an integer number, sequentially starting from zero, to maximum number of DOFs of the model. Moreover, *NodalDOFNumberer* class adds DOF numbers to implement the constrains on the system using the Lagrange multipliers method.

NodalDOFNumberer takes its name as it loops around all nodes of the system in the order of their ids to number the associated DOFs. The numbering process is presided by another loop around nodes of the model to define the active DOfs of the system.

In the first loop, *NodalDOFNumberer* finds the DOFs used by the connected elements. If the DOF's direction is not activated by the user, -2 is assigned to the corresponding DOF. In the absence of this condition, assigned value to the DOF retains its default value 0. In the second loop, *NodalDOFNumberer* class checks every DOF of the system and gives equation numbers starting from zero if the DOF's preassigned value is not equal to -2.

After the numeration of DOFs of the system is finished, *NodalDOFNumberer* finds the total number of Lagrange multipliers needed to represent constrains held on the system and stores it in memory to be used in the assembly process.

4.3.1.3 LinearDynamicAnalyzer class

LinearDynamicAnalyzer class implements a general time stepping structure that can be used with dynamic algorithms. In addition to the general analyzer tasks, discussed in section 3.2.4.1, LinearDynamicAnalyzer class creates a loop within the limits of timetable assigned to the connected algorithm.

Before the beginning of time steps, *LinearDynamicAnalyzer* calls the *DOFNumberer* object created by the *Algorithm* to number DOFs of the system. At every cycle of the loop, a number of tasks are executed. First *LinearDynamicAnalyzer* checks if a *GlobalMatrixAssembler* is instantiated by the algorithm; if an instance exists, it is called to assemble global matrices and vectors of the *Algorithm*. If an instance does not exist, it is assumed that either global matrix assembly is not neccesarry or is done internally by the *Algorithm*. After the assembly process Iterate function of the algorithm is called to execute a step from the time integration. Next, *LinearDynamicAnalyzer* updates the displacements, velocities and accelerations of nodes of the finite element model. *UpdateElement* function of MaterialModel class is also called for every element of the model, to update stress-strain state of material models. At last Tracker objects assigned to the analyzer are called in order to write output.

4.3.2 Material Model

Equivalent linear material model was implemented by a class called *NLElasticMaterialModel*, that inherits the *MaterialModel* class (Figure 4.7). The name NLElasticMaterialModel comes from the implementation's ability to represent elastic properties, linear or nonlinear, of a material. Change of material model characteristics are linked to the shear strain level of finite elements.

NLElasticMaterialModel class gathers values of properties that define characteristics of the material model, from user. Main properties like the Poisson's ratio υ and maximum shear modulus G_{max} of the material are gathered in the form of a vector consisting of double values; whereas names of two files, containing discretized shear modulus reduction and critical damping curves of soil material model, are gathered in the form of user options. An example of material model definition in ptl language is given below.

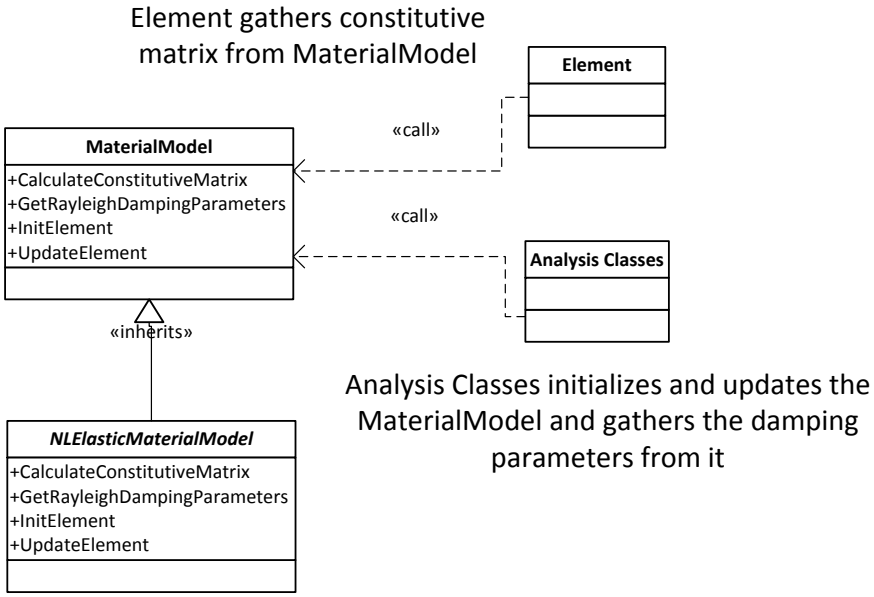


Figure 4.7: NLElasticMaterialModel Class Diagram

```
// Create a material model tagged with id 1
// Gmax = 6700, v = 0.3, density = 6.7, f1 = 0.4, f2 = 1.33
Create.MaterialModel  "NonLED"  1 6700 0.3 6.7 0.4 1.33
Set.Option  materialModel 1 // set options for material model tagged with id 1
{
"modulusFile" = "modulus_reduction.txt"
"dampingFile" = "critical_damping.txt"
}
```

Parameters, taken from user, are summarized in Table 4.5. Poisson's ratio υ and maximum shear modulus G_{max} parameters indicate the strength of the material model at small strain conditions. Density is returned to the connected elements, to be used in computation of mass matrices and self weight element loads.

Damping formulated by the Equation 4.21 is independent of frequency. This type of damping characteristic is impossible to model with time domain analysis. *NLElasticMaterialModel* class approximates this damping behavior utilizing Rayleigh damping formulation (Equation 4.23). Two frequencies are gathered from user in order to compute Rayleigh damping coefficients using equations 4.25. Computed Rayleigh damping coefficients are returned to

Analysis classes with function *GetRayleighDampingParameters*.

Table 4.5: Material Properties of NONLED

Property	Definition
G_{max}	Maximum Shear Modulus of soil material
υ	Poisson's ratio of soil material
Density	Density of Soil material
f1	frequency 1
f2	frequency 2

Curves describing the modulus reduction and critical damping behavior of the material are constructed from discrete points defined in text files prepared by user; values between the discrete points are linearly approximated. At every point of the analysis, shear modulus and critical damping of soil material can be found using these curves and the average strain level of finite elements. Since Panthalassa does not record strain levels of finite elements during analysis, they must be computed and recorded to memory at every step of the analysis.

In order to allow tracking of shear strain levels of finite elements by the *NLElasticMaterialModel* class, functions *initElement* and *updateElement* are called by Analysis classes. Function *initElement* is called once, before the beginning of the analysis for every element. *NLElasticMaterialModel* creates a map, an associative container, between the id's of elements and a double value which is used for storing the shear strain values. Function *updateElement* is called at every step of the analysis for every finite element. *NLElasticMaterialModel* calculates the average strain value for the element and stores in the map structure described above. Average shear strain of the finite elements are found by averaging the maximum shear strain calculated at nodes of the element.

The main use of an inheritor of the *MaterialModel* class is to compute constitutive matrices for connected elements. *NLElasticMaterialModel* class accomplish this task in the *CalculateConstitutiveMatrix* function. Whenever an element object calls the the *CalculateConstitutiveMatrix* function, shear strain value of the element is read from memory. This strain value is used to calculate equivalent linear parameters from user defined modulus reduction curve. Then constitutive matrix is is generated using the relationships given in Equations 4.14 and 4.15.

In addition to calculating constitutive matrices from shear strain values calculated at every time step, maximum shear strain computed up to the analysis time might be used to calculate constitute matrices as well. In this calculation type, maximum shear strain computed is multiplied with a constant value to get an average effective shear strain value. Equation 4.29 (ProShake Manual [64]) presents this relationship.

$$\gamma = \frac{M_w - 1}{10}\gamma_{max} \tag{4.29}$$

In this Equation M_w is the magnitude of the input earthquake motion and given by the user. If this value is not available Equation 4.30 is used instead of Equation 4.29.

$$\gamma = 0.65\gamma_{max} \tag{4.30}$$

0.65 is an empirical value suggested in Lysmer et al. [65].

CHAPTER 5

Verification Problems

5.1 Introduction

The developed software program, Panthalassa was tested with four different problems in order to verify the results of the linear and equivalent linear dynamic analyses. First two problems, benchmarked the dynamic behavior of bilinear quadrilateral and linear hexahedron elements. Third problem, benchmarked one dimensional earthquake wave propagation with Rayleigh damping. The last problem, benchmarked the implemented equivalent linear material model with EduShake (EduPro Civil Systems [66]) program which performs equivalent linear analysis in frequency domain.

5.2 Problem 1: 1-D Wave Propagation

In order to verify the dynamic solution procedure first, wave propagation speed in an infinite soil column was compared with a one dimensional analytical solution. To model this problem a soil column with 10 m. of height was subjected to a vertical displacement of 0.001 m. at the top. Then, the speed of pressure waves were computed at the mid point of the soil column (Figure 5.1). Two different models, one with fixed boundary at the bottom and one with absorbent boundaries surrounding the soil column, were used in the test. This problem was adopted from Plaxis Dynamics Manual Chapter 4.1 [67]. Linear hexahedron elements were used to in the finite element model (Figure 5.2). Table 5.1 summarizes the material and mesh properties of the model.

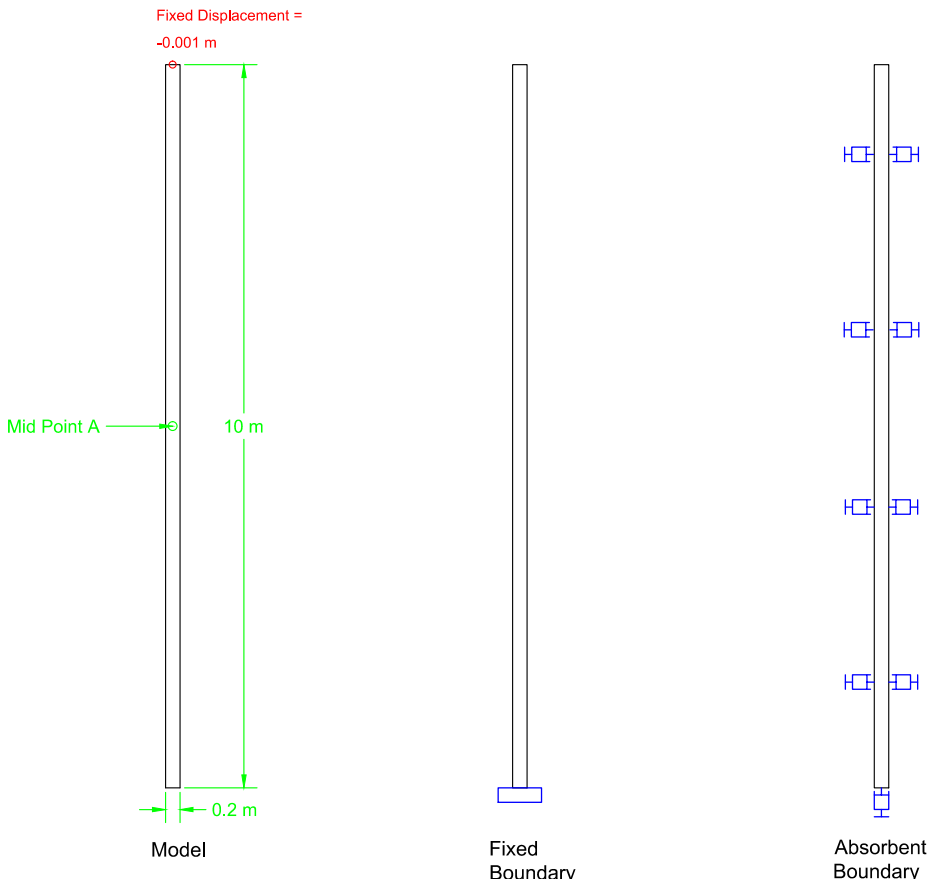


Figure 5.1: Verification Problem 1: 1-D Wave Propagation, Fixed and Absorbent Boundary Models

Pressure wave speed of a 1D soil column can be calculated using Equation 5.1.

$$V_p = \sqrt{\frac{E(1-v)}{(1+v)(1-2v)\rho}} \tag{5.1}$$

In this equation E is the elastic modulus, v is the Poisson's Ratio and ρ is the density of the soil calculated by dividing the soil weight to the gravity acceleration ($\rho = \frac{\gamma}{g}$). Using this equation, V_p traveling in this soil column was calculated to be 99 m/s^2.

Figure 5.3 presents the time-displacement curve computed at the mid-point of the soil column for two finite element models and the analytical solution. In the analytical solution, pressure waves reached to the mid point at 0.05 s. and dissipated without any reflection through the infinitely deep soil column. In both finite element models mid-point started moving around 0.05 s. Displacement curves were not as steady as the analytical one however, at average both models captured the expected behavior. Model with fixed boundaries at the bottom, showed

Table 5.1: Verification Problem 1: 1-D Wave Propagation, Model Parameters

Parameter	Value
E (Elastic Modulus)	18000 kPa
v (Poisson's Ratio)	0.2
γ (Soil Weight)	20 kN/m^2
g (Gravity Acceleration)	9.81 m/s^2
Model Size	10 m x 0.2 m x 0.2 m
Element Size	0.1 m x 0.1 m x 0.1 m

Figure 5.2: Verification Problem 1: 1-D Wave Propagation, Finite Element Mesh

the effects of wave reflection. Pressure waves reflected from the fixed boundaries, reached the mid-point at around 0.15 s. (0.1 s. to reach the fixed boundary and 0.05 s. to reach the mid-point). Pressure waves were then reflected from top and reached the mid-point again.

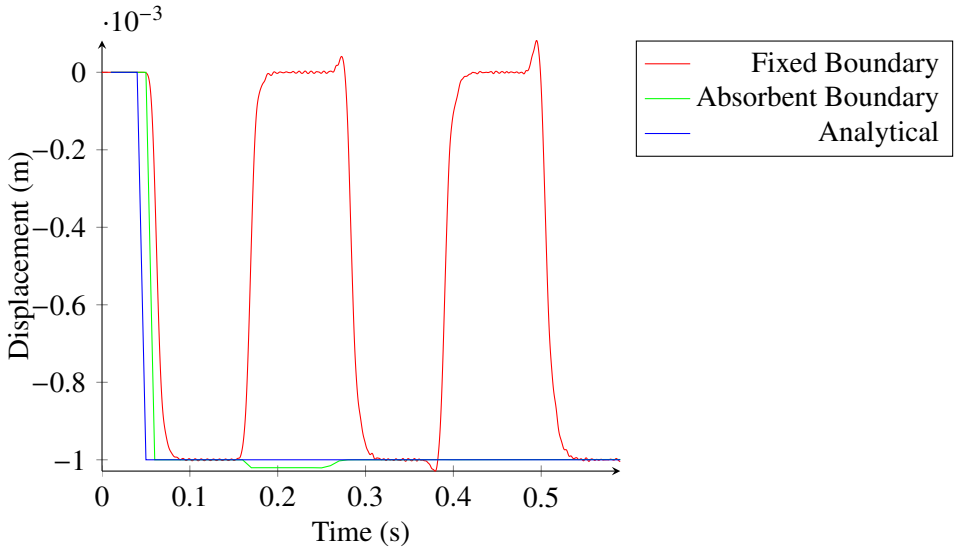


Figure 5.3: Time Displacement Curves for Mid Point A

This cycle was repeated infinitely. Absorbent boundaries solved the reflection problem as only a small part of pressure waves were reflected from them. In the first cycle almost all of pressure waves were dissipated, and in the second cycle all reflection was erased.

5.3 Problem 2: Rayleigh Wave Velocity

Figure 5.4 presents the second verification problem which compared the speed of Rayleigh waves generated by an instantaneous load at the surface of a solid body computed by a finite element model with the analytical calculation. Finite element model for the soil body has 5 m. of height and 10 m. of width and meshed with bilinear quadrilateral elements (Figure 5.5). Soil body was surrounded by absorbent boundaries. Material and mesh properties are presented in Table 5.2. This problem was also adopted from Plaxis Dynamics Manual Chapter 4.3 [67].

Ratio of speed of Rayleigh wave and pressure wave for a solid material can be calculated for different Poisson's ratios, for $v = 0.25$, $\frac{V_r}{V_p} = 0.54$ (Kramer Chapter 5.3.1.1) [55]. Pressure wave velocity, V_p, can be calculated from Equation 5.1 and using model parameters from Table 5.2 Rayleigh wave velocity, V_r was calculated as 13.23 m/s.

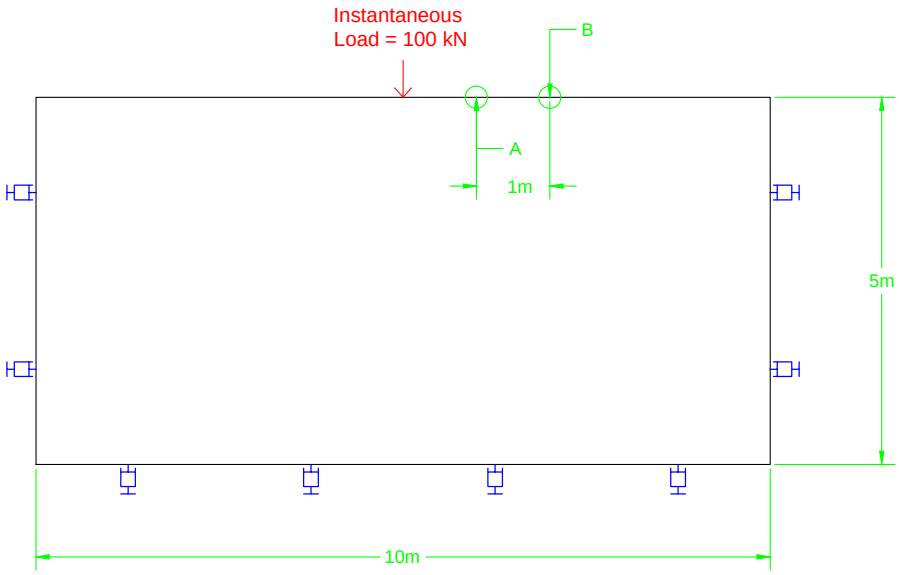


Figure 5.4: Verification Problem 2: Rayleigh Wave Propagation, Finite Element Model

Table 5.2: Verification Problem 2: Rayleigh Wave Propagation, Model Parameters

Parameter	Value
E (Elastic Modulus)	1000 kPa
ν (Poisson's Ratio)	0.25
γ (Soil Weight)	19.6 kN/m^2
g (Gravity Acceleration)	9.81 m/s^2
Model Size	0.025 m x 0.05 m
Element Size	0.1 m x 0.1 m x 0.1 m

Using the finite element model, time displacement curves for Points A and B that are 1m apart from each other were calculated. From Figure 5.6 it was observed that Rayleigh waves travel from Point A to Point B in 0.076 s. Then $V_r = \frac{distance}{time} = \frac{1m}{0.076s} = 13.16m/s$ which is in good agreement with the analytical value of 13.23 m/s.

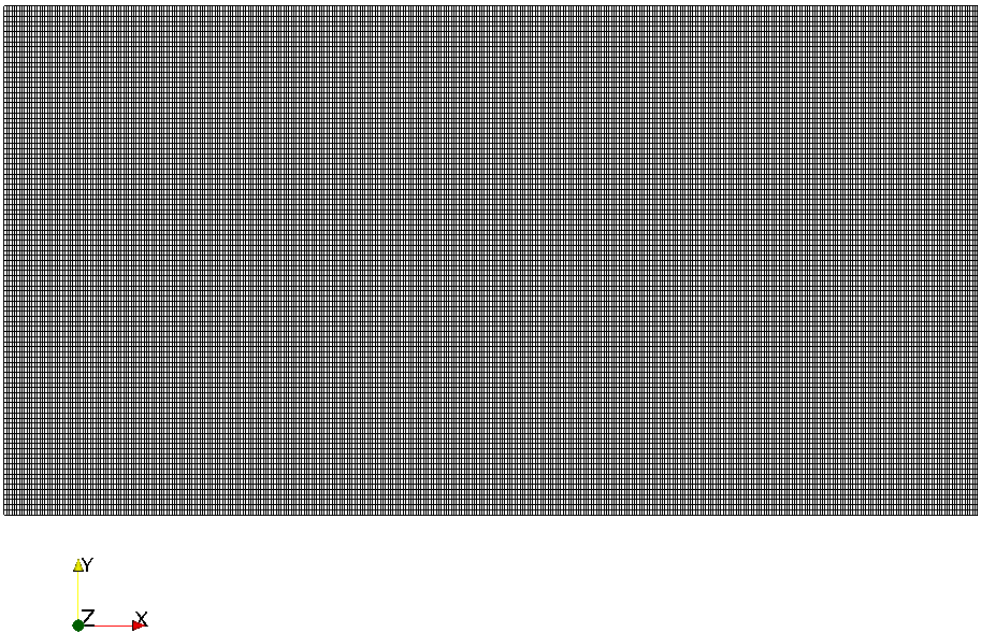


Figure 5.5: Verification Problem 2: Rayleigh Wave Propagation, Finite Element Mesh

5.4 Problem 3: 1D Translation Function

Figure 5.7 presents the third verification problem. In this problem, an earthquake motion was given to a one dimensional soil column model in one direction at the bottom and amplification of the earthquake waves were computed at the top. Measurements were compared with the analytical solution. Finite element model consists of a soil layer of 24 m. deep. Soil layer has a shear wave velocity of 200 m/s. Model parameters are presented in Table 5.3.

East-west component from Treasure Island record of Loma Prieta Earthquake was used in the model. Figures 5.8 and 5.9 present the time-acceleration and Fourier spectrum curves of the earthquake. As seen from the second graph earthquake experiences most of its acceleration in 0-5Hz frequency range and after 10Hz motion gradually vanishes. Earthquake data used in this study was composed of 2000 data points 0.02s apart that lasts for 40 seconds.

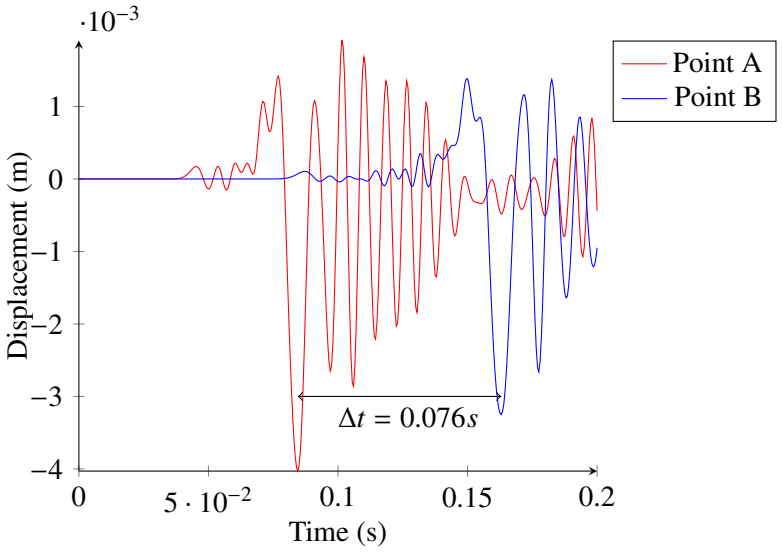


Figure 5.6: Time Displacement Curves for Points A and B

Table 5.3: Verification Problem 3: 1D Translation Function, Model Parameters

Parameter	Value
E (Elastic Modulus)	100000 kPa
v (Poisson's Ratio)	0.25
V_s (Shear Modulus)	200 m/s
γ (Soil Weight)	20.0 kN/m^2
g (Gravity Acceleration)	9.81 m/s^2
Model Size	1 m x 24 m
Element Size	0.5 m x 0.5 m

For a one dimensional soil layer on rigid bedrock amplification ratio is given by equation 5.2 (Roesset J.M. [68]).

$$A(f) = \frac{1}{\sqrt{cos^2(2\pi\frac{H}{V_s}f) + (2\pi\frac{HD}{V_s}f)^2}} \tag{5.2}$$

In this equation f is the frequency that the amplification ratio applies to, H is the depth of soil column, D is the damping ratio and V_s is the shear wave velocity which is calculated by the following equations:

$$V_s = \sqrt{\frac{G}{p}} \tag{5.3}$$

$$G = \frac{E}{2(1+v)} \tag{5.4}$$

where G is the shear modulus. For a damping ratio of 0.05 amplification ratio curve for the

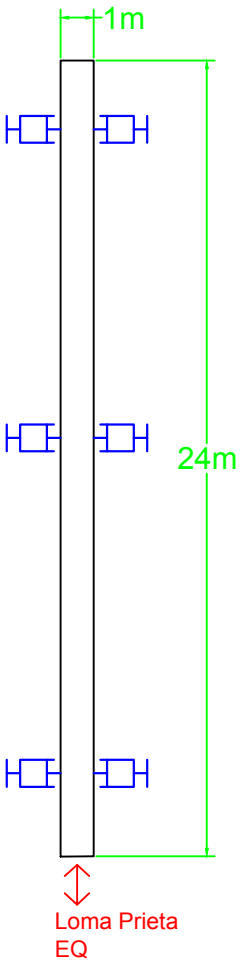


Figure 5.7: Verification Problem 3: 1D Translation Function, Model

soil site is given in Figure 5.10.

In order to include damping in the model, Rayleigh damping was used with a frequency range 1.5-5 Hz. This range covers both the important frequency range of the Loma Prieta earthquake and analytical amplification curve. Figure 5.11 presents the comparison of the analytical curve and computed amplification ratio from the finite element model. Computed amplification curve almost exactly matches with the analytical amplification curve in 0-3 Hz range, however it is overdamped for higher frequencies. This is due to the stiffness proportional part of the Rayleigh damping (Figure 4.6). This behavior is acceptable for this model as Loma Prieta earthquake does not include high amplitude motion in higher frequencies. However different

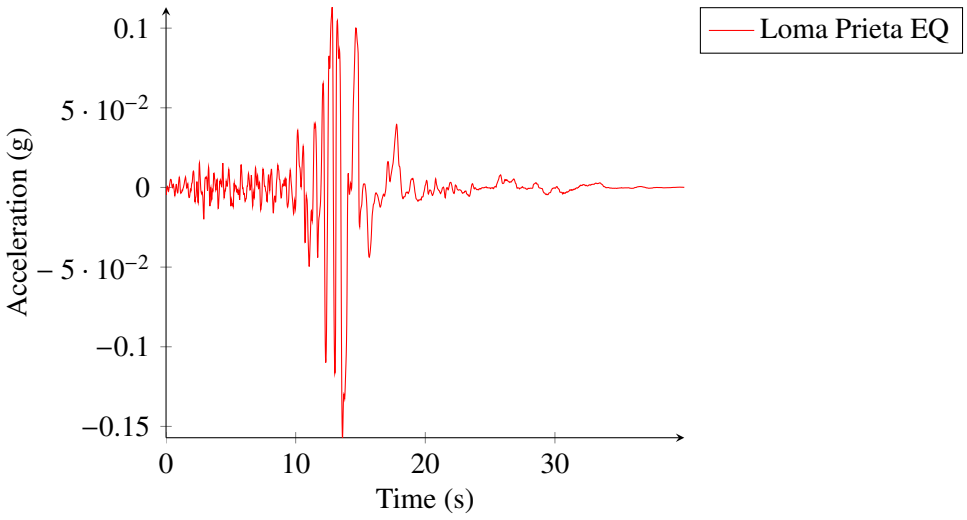


Figure 5.8: Time Acceleration Curve for Loma Prieta Earthquake

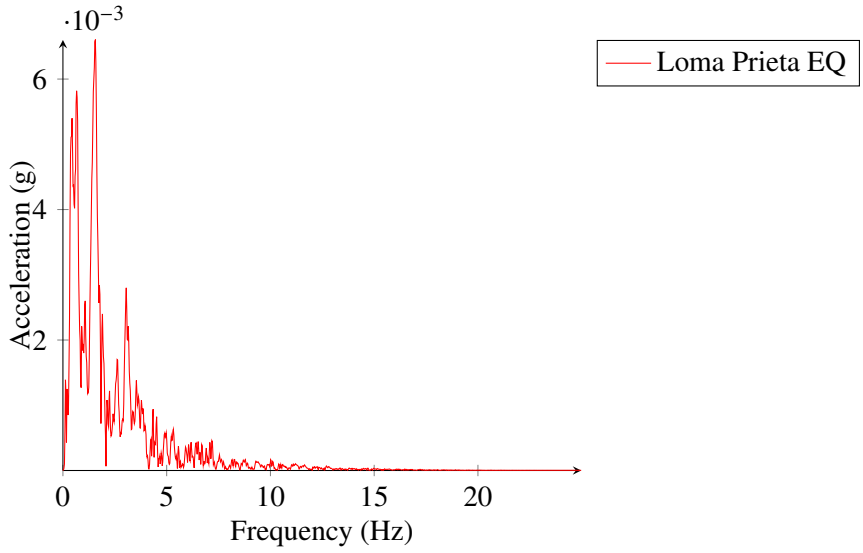


Figure 5.9: Fourier Spectrum for Loma Prieta Earthquake

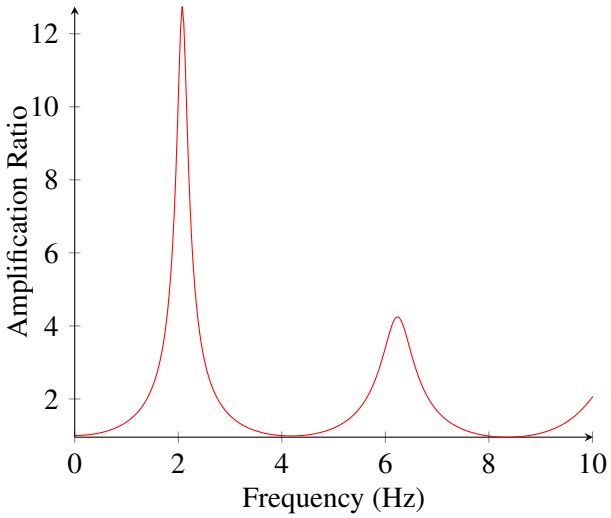


Figure 5.10: Analytical Amplification Curve D = 0.05

frequency ranges should be investigated for problems with different input motions or soil sites with different material properties. A more detailed discussion on the effect of Rayleigh damping parameters on 1D amplification can be found in Visone et.al. [69].

5.5 Problem 4: Equivalent Linear Material Model

In this Section, performance of the implemented equivalent linear material model was benchmarked. The benchmark was performed against EduShake program. EduShake implements the equivalent linear material model like Panthalassa, however solves the problem in frequency domain. In the frequency domain solution input motion is represented as the sum of a series of sine waves of different amplitudes and response of the soil profile to the input motion is calculated (ProShake User's Manual [64]). Other differences between the analysis of EduShake and Panthalassa are EduShake performs the solution in one dimension and uses frequency independent damping. Frequency independent damping used in EduShake is dependent on shear strain of soil whereas Rayleigh damping is dependent on frequency (Section 4.2.4).

Figure 5.12 presents the model used in this problem. A soil deposit with 50 m. depth was subjected to Loma Prieta earthquake that was used in the previous problem. Soil model was surrounded with absorbent boundaries. In order to avoid two dimensional wave reflections

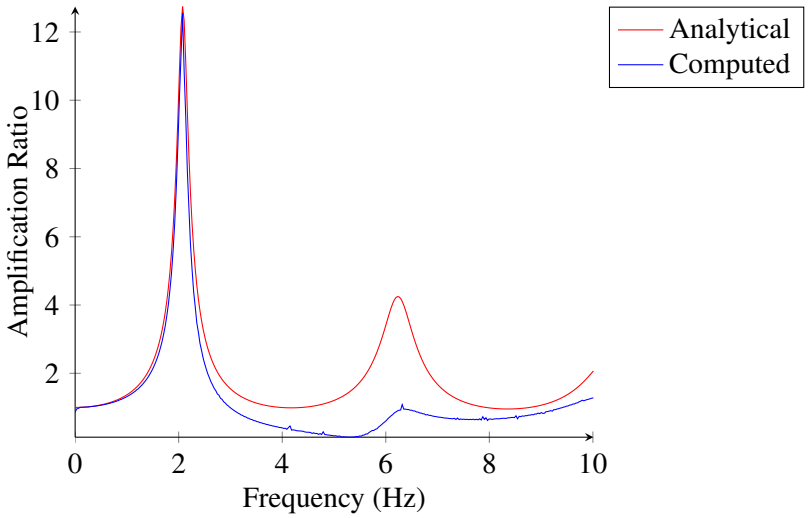


Figure 5.11: Comparison of Analytical and Computed Amplification Curves

that may not be observed by absorbent boundaries soil model was extended to 800 m. width.

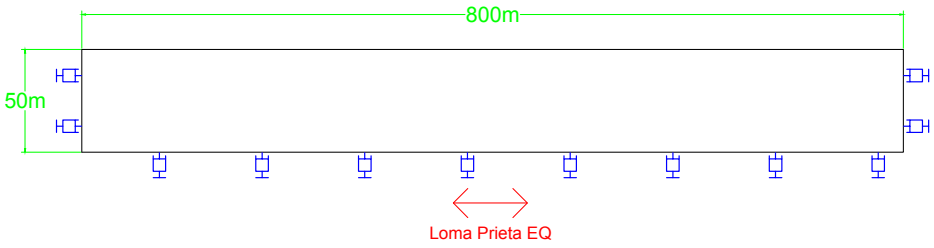


Figure 5.12: Verification Problem 4: Equivalent Linear Material Model

Properties of soil and mesh used in this problem are presented in Table 5.4. Shear modulus reduction and damping curves used in this problem were taken from from Vucetic and Dobry [56] (Figure 4.5). Mesh properties of the model were determined using Equation 2.5. Since Loma Prieta earthquake does not have significant amplitude in frequencies higher than 10 Hz. mesh length of 2 m. in vertical direction is enough to transmit earthquake waves from bottom to top. However since equivalent linear material model predicts shear modulus reduction, 0.5 m mesh length which, takes a 25x shear modulus reduction into account, was selected.

In order to test mesh properties and establish a base comparison between time domain and frequency domain solutions first a linear test with 5% damping was performed. Figure 5.13

Table 5.4: Verification Problem 4: Equivalent Linear Material Model, Model Parameters

Parameter	Value
E (Elastic Modulus)	200000 kPa
v (Poisson's Ratio)	0.25
V_s (Shear Modulus)	200 m/s
γ (Soil Weight)	19.6 kN/m^2
g (Gravity Acceleration)	9.81 m/s^2
PI (Plasticity Index)	100
Rayleigh Damping Frequency 1	1.5
Rayleigh Damping Frequency 2	4.5
Model Size	50 m x 800 m
Element Size	0.5 m x 10 m

presents absolute maximum accelerations computed at different depths by two methods. Time domain solution results in about 0.02g higher results than the frequency solution. Nonlinear curve of the time domain solution is a result of two dimensional mesh and Rayleigh damping.

After the linear analysis, an equivalent linear analysis was performed. In this analysis equivalent linear method based on both shear strain at every time step ($\gamma = \gamma_{t_\tau}$) and maximum shear strain obtained up to the time step ($\gamma = max(\gamma_{t<t_\tau})$) were used. Figure 5.14 presents absolute maximum accelerations computed at different depths by three methods for equivalent linear analysis. Similar to the results of the linear analysis, time domain solutions result in slightly larger acceleration values than the frequency domain solution. For all solution methods, equivalent linear analysis results in lower accelerations (Table 5.5) than linear analysis. This observation confirms the results of the implemented equivalent linear material model.

Table 5.5: Verification Problem 4: Accelerations Computed at the Top of the Soil Layer

	Time Domain	Frequency Domain
Linear	0.3306	0.3112
Equivalent Linear ($\gamma = \gamma_{t_\tau}$)	0.3146	0.3012
Equivalent Linear ($\gamma = max(\gamma_{t<t_\tau})$)	0.3036	

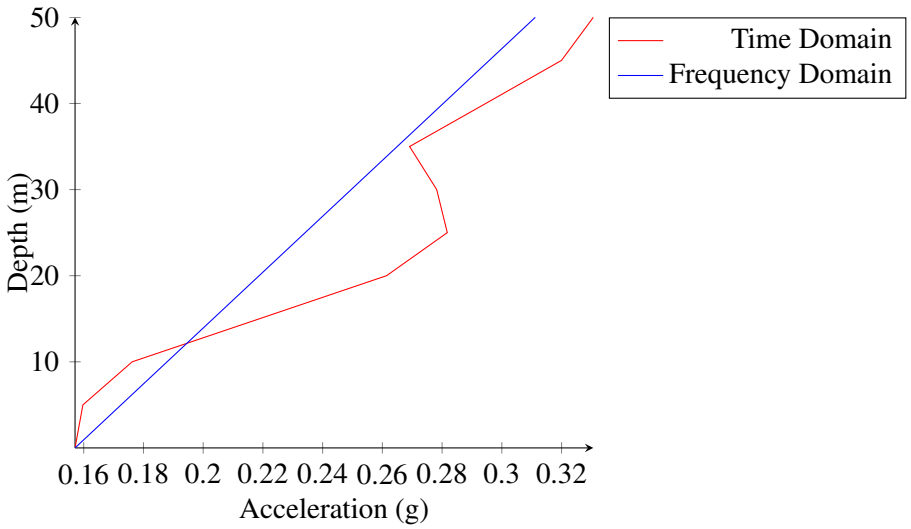


Figure 5.13: Absolute Maximum Accelerations vs Depth Curves for Linear Solution

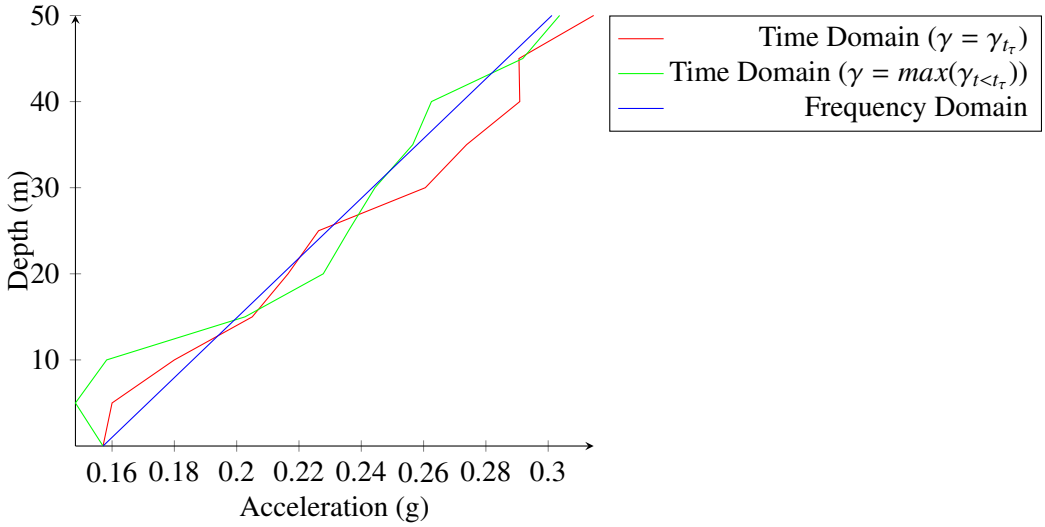


Figure 5.14: Absolute Maximum Accelerations vs Depth Curves for Equivalent Linear Solution

CHAPTER 6

Parallel Tests

6.1 Introduction

The Parallel efficiency of the developed software platform was tested with two large scale models. The first model was the fourth verification problem that is presented in Section 5.5. The second model was a three dimensional adaptation of the first model. Dimensions of the first model was changed to 240 m x 240 m x 50 m in the second model. Linear hexahedron element was used in this model. Tests were performed on a cluster of 8 computers with Intel Core2 Quad Q9300 CPUs and 3GB of RAM running on Windows XP operation system.

6.2 Case Studies

Case Study models were analyzed with both linear and equivalent linear methods using parallel processing. Output of solutions with different number of processes were compared in order to verify the parallel solution procedure. Performances of the solutions were analyzed for different meshes and number of elements. Every test was performed three times and average solution time was taken into account. Tests were performed using 1, 2, 4, 8, 16, and 32 processes. Note that, to use more than eight processes more than one core of a processor were utilized.

6.2.1 Linear Tests

Figure 6.1 presents timings and speed-ups achieved with parallel linear dynamic analyses. Model 6, which is the biggest model with 72000 elements, could not be solved using a single

Table 6.1: Mesh Sizes of Models Analyzed with Parallel Solution Procedure

Model No	Model Size	Element Size	# DOFs
1	800 m x 50	10 m x 0.50 m	16000
2	800 m x 50	10 m x 0.25 m	32000
3	800 m x 50	10 m x 0.10 m	81000
4	240 m x 240 m x 50	20 m x 20 m x 0.50 m	51200
5	240 m x 240 m x 50	20 m x 20 m x 0.25 m	101900
6	240 m x 240 m x 50	20 m x 20 m x 0.10 m	254000

processor because of memory exhaustion. In order to calculate speed-up values for Model 6, run time for the solution of Model 6 for this case was estimated by multiplying the speed-up value from Model 5 achieved using 2 processors by the run time for the solution of Model 6 achieved using 2 processors. For first three models speed-ups achieved were below one. For models 4, 5 and 6 speed-ups more than one were achieved.

Top accelerations computed with linear tests for the first three models are presented in Table 6.2. All values are the same as the sequential solution from Section 5.5 which verifies the parallel solution.

Table 6.2: Acceleration at Top of Soil Layer Computed with Different Number Processes, Linear Solution

# Processes	1	2	4	8	16	24	32
Top Acceleration (g)	0.3306	0.3306	0.3306	0.3306	0.3306	0.3306	0.3306

As discussed in Section 4.3.1 linear solution procedure is composed of two main parts. First part consists of assembly of solution space and factorization. The task of solution space is parallelized as each process partition a part of the solution space. Factorization is parallelized using the MUMPS library. Second part consists of forward and back substitutions for every time step of the analysis. Second part cannot be parallelized easily as it involves vector addition and multiplication with real numbers which are processes with short running times. In the second step overhead of message passing becomes large enough that little or no gain can be expected. Table 6.3 presents time spent in these two parts for linear analyses. For linear analysis first part is executed only once however second part reoccurs at every time

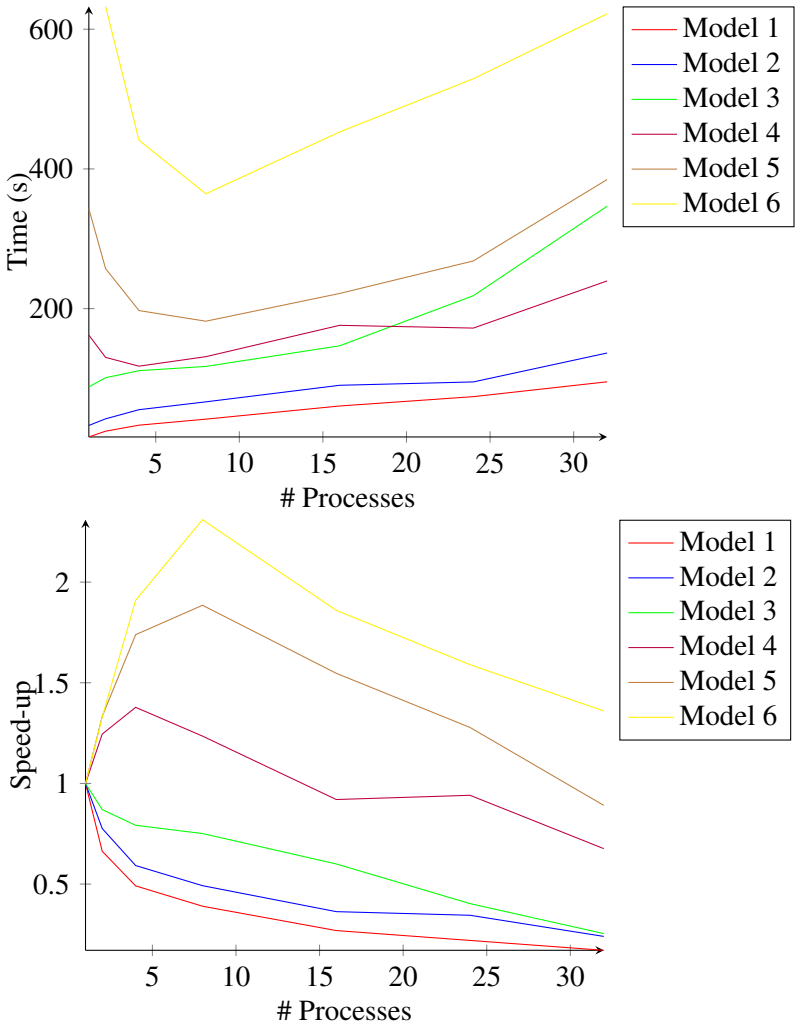


Figure 6.1: Timings and Speed-Ups, Parallel Linear Analyses

step. For dynamic analysis with many time steps (2000 for this problem) second part of the solution procedure takes up most of the analysis time. For this reason, there was no run time improvement for small two dimensional models. Although speed-ups over one were experienced for bigger three dimensional models, parallel efficiency achieved was poor and only eight processors could be used effectively.

Table 6.3: Time Spent In Solution Steps For Linear Analyses

# Processes	1	2	4	8	16	24	32
Model 1							
Part I (s)	0.607	0.350	0.223	0.187	0.352	0.627	0.857
Part II (s)	15.675	24.197	32.918	41.595	60.257	73.357	94.410
Model 2							
Part I (s)	1.231	0.691	0.424	0.302	0.284	0.499	0.757
Part II (s)	31.535	41.543	54.888	66.245	89.966	94.563	135.696
Model 3							
Part I (s)	3.128	1.803	1.071	0.752	0.640	0.785	0.933
Part II (s)	84.872	99.306	110.085	116.498	146.125	217.730	345.645
Model 4							
Part I (s)	24.617	12.203	6.094	3.000	1.578	1.062	0.875
Part II (s)	137.430	117.984	111.531	128.218	174.516	171.078	238.750
Model 5							
Part I (s)	49.985	24.813	12.328	6.078	3.046	2.078	1.656
Part II (s)	292.983	232.483	184.888	175.906	218.642	266.209	383.157
Model 6							
Part I (s)	x	62.266	30.921	15.250	7.609	5.172	4.000
Part II (s)	x	570.406	410.141	349.172	445.266	523.906	618.016

6.2.2 Equivalent Linear Tests

Figure 6.2 presents timings and speed-up values achieved with parallel equivalent linear analyses. Similar to linear analyses Model 6 could not be solved with a single processor because of memory exhaustion and run time for this case was estimated using the same procedure with the one used in linear analyses in order to calculate speed-ups for Model 6. In equivalent linear analyses speed-up values over 1 were achieved for all test cases. Table 6.4 presents the highest speed-up factors for all models. Analysis of the smallest model, Model 1, continued to get faster up to 8 processes. Medium sized models: Model 4, 5 and 6 achieved their highest speed-up with 16 processors. Model 5 continued to get faster up to 24 processors. All of 32

processors could be efficiently utilized for the solution of the largest model: Model 6. As the analyzed model got bigger speed-up values and the number of processes it was achieved with got bigger.

Table 6.4: Highest Speed-Up Values Achieved by Parallel Equivalent Linear Solution

Model	# Processes	Speed-Up
Model 1	8	3.24
Model 2	16	4.34
Model 3	16	4.88
Model 4	16	7.05
Model 5	24	8.50
Model 6	32	11.51

Top accelerations computed with equivalent linear tests for the first three models are presented in Table 6.5. All values are the same as the sequential solution from Section 5.5 which verifies the parallel solution.

Table 6.5: Acceleration at Top of Soil Layer Computed with Different Number Processes, Equivalent Linear Solution

# Processes	1	2	4	8	16	24	32
Top Acceleration (g)	0.3146	0.3146	0.3146	0.3146	0.3146	0.3146	0.3146

In the equivalent linear solution procedure, both solution steps discussed in Section 6.2.1 are performed for every time step. Since parallelization is effective for solution space assembly and factorization, more effective parallelization compared to linear analyses could be achieved.

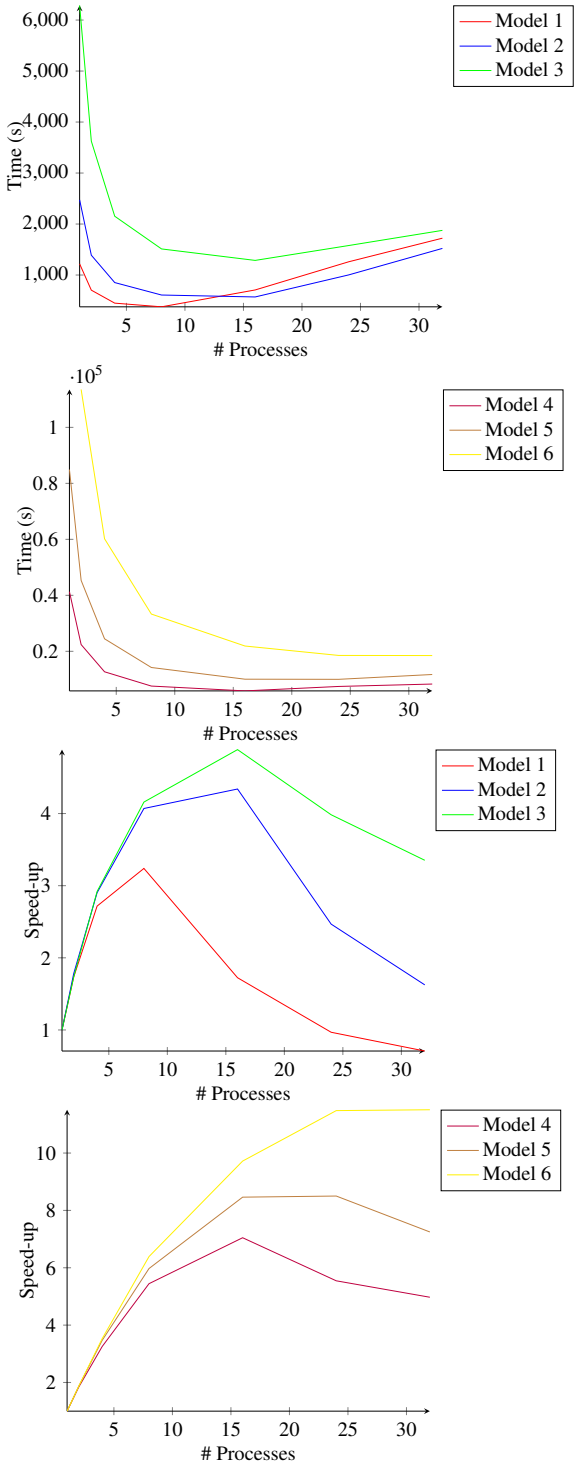


Figure 6.2: Timings and Speed-Up Values, Parallel Equivalent Linear Analyses

CHAPTER 7

Summary and Conclusion

7.1 Summary

Within the confines of this study, a parallel finite element library called Panthalassa was developed in order to analyze dynamic SSI problems using parallel computing. Panthalassa is a general purposed library which is extensible using plug-ins. A series of plug-ins were developed and added to the library in order to extend the capabilities of the library to parallel dynamic analysis with the implicit Newmark method and the use of equivalent linear material model.

Four verification problems were solved to benchmark the implemented algorithms. First three problems compared computations of the implemented dynamic analysis algorithm with analytical values. The last problem compared the implemented equivalent material model with a one dimensional frequency domain solution that uses the same material model.

In order to test the parallel implementation, the last of the verification problems was solved using linear and equivalent linear models in parallel on a cluster composed of eight computers. Verifications of the parallel solutions and performance analysis of solutions were performed. A discussion of the results were given.

7.2 Conclusion

Based on the performed parallel tests the following conclusions could be made about the implementations and their parallel performance:

- Implemented equivalent linear model results matched well with both analytical and well-based existing numerical solutions.
- Implemented parallel linear dynamic algorithm performed poorly in terms of scalability and performance gain from parallel computing.
- Implemented equivalent linear dynamic algorithm gave good results in terms of scalability and performance gain from parallel computing, achieving up to 11.51 times speed-up using 32 processes.
- Factorization and solution space assembly steps of the dynamic analysis are major parts that can be parallelized.
- Algorithms that need to execute factorization and solution space assembly steps many times, like equivalent linear analysis in time domain, nonlinear static and nonlinear dynamic analyses can be parallelized with a similar strategy to the implementation carried out in this study.
- Algorithms that need to execute factorization and solution space assembly steps only once and use the factorized matrices many times may not be good candidates to be parallelized with a similar strategy used in this study depending on time spent on parts other than factorization.
- As the number of processors used in an analysis gets bigger, the size of the problem needs to get bigger as well for good parallel performance.

Bibliography

[1] G. V. Wilson. *The History of the Development of Parallel Computing*. URL: http://ei.cs.vt.edu/~history/Parallel.html (visited on 12/23/2010).

[2] G. Anthes. *The Power of Parallelism*. URL: http://www.computerworld.com/s/article/65878/The_Power_of_Parallelism (visited on 12/23/2010).

[3] R. Trobec. *Parallel computing: Numerics, Applications and Trends*. Springer, 2009.

[4] M. J. Sottile, T. G. Mattson, and C. E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman and Hall, 2010.

[5] OpenMP. *Open Multi Processing*. URL: http://openmp.org (visited on 11/14/2010).

[6] c++0x. *The C++ Standards Committee*. URL: http://www.open-std.org/jtc1/sc22/wg21/ (visited on 11/14/2010).

[7] Message Passing Forum. *The Message Passing Interface Standard*. URL: http://www.mcs.anl.gov/research/projects/mpi (visited on 11/14/2010).

[8] MS-MPI. *Microsoft MPI*. URL: http://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx (visited on 11/14/2010).

[9] *Message Passing Interface Chameleon 2 Library*. URL: http://www.mcs.anl.gov/research/projects/mpich2 (visited on 11/14/2010).

[10] Nvidia. *Nvidia Fermi Architecture Whitepaper*. Nvidia. 2010.

[11] Nvidia. *CUDA Zone*. URL: http://www.nvidia.com/object/cuda_home_new.html (visited on 11/14/2010).

[12] Khronos. *OpenCL, The Open Standard for Parallel Programming of Heterogeneous Systems*. URL: http://www.khronos.org/opencl/ (visited on 11/14/2010).

[13] Microsoft. *Microsoft DirectX Developer Center*. URL: http://msdn.microsoft.com/en-us/directx (visited on 11/14/2010).

[14] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. Hwu. “GPU Clusters for High-Performance Computing”. In: *International Conference on Cluster Computing*. 2009.

[15] K. Karimi, N. G. Dickson, and F. Hamze. “A Performance Comparison of CUDA and OpenCL”. In: *arXiv.org* (2010).

[16] T. L. Sterling. *Beowulf cluster computing with Linux*. The MIT Press, 2002.

[17] J. L. Potter. *The Massively Parallel Processor*. The MIT Press, 1985.

[18] TOP500. *TOP500 Super Computer Sites*. URL: `http://www.top500.org/` (visited on 11/14/2010).

[19] PVM. *Parallel Virtual Machine*. URL: `http://www.csm.ornl.gov/pvm/` (visited on 11/14/2010).

[20] G. A. Geist and J.A. Kohl. “PVM and MPI: a Comparison of Features”. In: *Calculateurs Paralleles, Vol. 8, pp. 137-150* (2009).

[21] R. D. Cook, D.S. Malkus, and M.E. Plesha. *Concepts and Applications of Finite Element Analysis*. 3rd ed. John Wiley and Sons, 2007.

[22] M. H. Aliabadi. *The Boundary Element Method Volume 2 Applications in Solids and Structures*. John Wiley and Sons, 2002.

[23] J. Lysmer and R. Kuhlemeyer. “Finite Dynamic Model for Infinite Media”. In: *Journal of the Engineering Mechanics Division Proceedings of the American Society of Civil Engineers, Vol. 95, Issue 4* (1969).

[24] W. White, S. Valliappan, and I. Lee. “A Unified Boundary for Finite Dynamic Models”. In: *Journal of the Engineering Mechanics Division, Vol. 103, No. 5* (1976).

[25] Y.K. Chow. “Accuracy of Consstent and Lumped Viscous Dampers in Wave Propagation Problems”. In: *International Journal for Numerical Methods in Engineering, Vol. 21, pp. 723- 732* (1985).

[26] H.R. Yerli, S. Kacin, and S. Kocak. “A Parallel Finite-Infinite Element Model for Two Dimensional Soil-Structure Interaction Problems”. In: *Soil Dynamics and Earthquake Engineering Vol. 23, Issue 4, pp. 249-253* (2002).

[27] T. J. R. Hughes and Liu W. K. “Implicit-Explicit Finite Elements in Transient Analysis: Implementation and Numerical Examples”. In: *Journal of Applied Mechanics, Vol. 45, Issue 2, pp. 375-379* (1978).

[28] P. Krysl and T. Belytschko. “Object-Oriented Parallelization of Explicit Structural Dynamics with PVM”. In: *Computers and Structures Vol. 66, Issues 2-3* (1998).

[29] P. Krysl and Z. Bittnar. “Parallel Explicit Finite Element Solid Dynamics with Domain Decomposition and Message Passing: Dual Partitioning Scalability”. In: *Computers and Structures Vol. 79, Issues 3* (2001).

[30] K. O. Noe and T. S. Sorensen. “Solid Mesh Registration for Radiotherapy Treatment Planning”. In: *Lecture Notes in Computer Science, Vol. 5958, pp. 59-70* (2010).

[31] D. Komatitsch, G. Erlebacher, D. Goddeke, and D. Michea. “High-order Fnite-Element Seismic Wave Propagation Modeling with MPI on a Large GPU Cluster”. In: *Journal of Computational Physics Vol. 229, Issue 20* (2010).

[32] Wikipedia. *Panthalassa*. URL: `http://en.wikipedia.org/wiki/Panthalassa` (visited on 11/14/2010).

[33] Editors of the American Heritage Dictionaries, ed. *Dictionary of Computer and Internet Words: An A to Z Guide to Hardware, Software, and Cyberspace*. Houghton Mifflin Harcourt, 2001.

[34] D.M. Potts and L. Zdravkovic. *Finite Element Analysis in Geotechnical Engineering: Theory*. Thomas Telford Publishing, 1999.

[35] O. Kurc. “A Substructure Based Parallel Solution Framework for Solving Linear Systems with Multiple Loading Conditions”. PhD thesis. Georgia Institute of Technology, 2005.

[36] BOOST. *uBLAS library*. URL: `http://www.boost.org/doc/libs/1_44_0/libs/numeric/ublas/doc/index.htm` (visited on 11/14/2010).

[37] Microsoft Developer Network. *Static (C++)*. URL: `http://msdn.microsoft.com/en-us/library/s1sb61xd.aspx` (visited on 11/14/2010).

[38] L. J. Gross and J. Yellen, eds. *Handbook of Graph Theory*. CRC Press, 2003.

[39] K. K. Meng, D. Fengming, and E. T. Guan. *Introduction to Graph Theory*. World Scientific Publishing, 2007.

[40] LEMON. *Library for Efficient Modeling and Optimization in Networks*. URL: `http://lemon.cs.elte.hu/trac/lemon` (visited on 11/14/2010).

[41] ParMetis. *Parallel Graph Partitioning and Sparse Matrix Ordering Library*. URL: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview (visited on 11/14/2010).

[42] G. Karypis and V. Kumar. *METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Department of Computer Science and Engineering Army HPC Research Center. 1998.

[43] G. Karypis and V. Kumar. "Multilevel algorithms for Multi-Constraint Graph Partitioning". In: *Proceedings of Supercomputing*. 1998.

[44] G. Karypis and V. Kumar. "Multilevel K-Way Partitioning Scheme for Irregular Graphs". In: *Journal of Parallel and Distributed Computing, Vol. 48, pp. 96-129* (1998).

[45] G. Karypis and V. Kumar. "Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs". In: *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. 1996.

[46] G. Karypis, K. Schloegel, and V. Kumar. *ParMetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library*. University of Minnesota, Department of Computer Science and Engineering Army HPC Research Center. 2003.

[47] T. Bahcecioglu. *Pugg*. URL: http://pugg.sourceforge.net (visited on 11/14/2010).

[48] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Professional, 2001.

[49] Microsoft Developer Network. *Dynamic-Link Library Functions*. URL: http://msdn.microsoft.com/en-us/library/ms682599\%28v=VS.85\%29.aspx (visited on 11/14/2010).

[50] N. M. Newmark. "A Method of Computation for Structural Dynamics". In: *ASCE Journal of the Engineering Mechanics Division, Vol. 85, No. EM3.* (1959).

[51] E. L. Wilson. "Dynamic Response by Step-By-Step Matrix Analysis". In: *Proceedings, Symposium on the Use of Computers in Civil Engineering, Labortotio Nacional de Engenharia Civil, Lisbon, Portugal, October 1-5* (1962).

[52] H. B. Seed and I. M. Idriss. *Soil Moduli and Damping Factors for Dynamic Response Analysis*. Tech. rep. Earthquake Engineering Research Center, 1970.

[53] T.J.R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice Hall, Englewood Cliffs, 1987.

[54] I. A. Beresnev and K. Wen. "Nonlinear Soil Response A Reality ?" In: *Bulletin of the Seismological Society of America, Vol. 86, No. 6, pp. 1964-1978, December* (1996).

[55] S. Kramer. *Geotechnical Earthquake Engineering.* Prentice-Hall International Series in Civil Engineering and Engineering Mechanics, 1996.

[56] M. Vucetic and R. Dobry. "Effect of Soil Plasticity on Cyclic Response". In: *Journal of Geotechnical Engineering, Vol. 117, No. 1, pp. 89-107* (1991).

[57] I. Ishibashi and X. Zhang. "Unified Dynamic Shear Moduli and Damping Ratios of Sand and Clay". In: *Soils Found., Vol. 33, No. 1, pp. 182-191* (1993).

[58] P. R. Amestoy, I. S. Duff, and J.Y. L'Excellent. "Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers". In: *Comput. Methods Appl. Mech. Eng Vol. 184 pp. 501-520* (1998).

[59] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*. Message Passing Interface Forum. 2009.

[60] P.R. Amestoy, T.A. Davis, and I.S. Duff. "An Approximate Minimum Degree Ordering Algorithm". In: *SIAM Journal on Matrix Analysis and Applications, Vol. 17, no 4, pp. 886-905* (1996).

[61] P.R. Amestoy. "Recent Progress in Parallel Multifrontal Solvers for Unsymmetric Sparse Matrices". In: *Proceedings of the 15th World Congress on Scientific Computation, Modeling and Applied Mathematics, IMACS 97* (1997).

[62] J. Schulze. "Towards a Tighter Coupling of Bottom-Up and Top-Down Sparse Matrix Ordering Methods". In: *BIT Numerical Mathematics, Vol. 41, no. 4, pp. 800-841* (2001).

[63] Netlib. *ScaLAPACK Library*. URL: `http://www.netlib.org/scalapack/` (visited on 11/14/2010).

[64] *ProShake Ground Response Analysis Program Version 1.1 Users Manual*. EduPro Civil Systems, Inc. 2007.

[65] J. Lysmer, T. Udaka, C. F. Tsai, and H. B. Seed. *FLUSH: A Computer Program for Approximate 3-D Analysis of Soil-Structure Interaction Problems*. College of Engineering, Engineering University of California. 1975.

[66] EduPro Civil Systems Inc. *EduShake*. URL: http://www.proshake.com/ (visited on 11/14/2010).

[67] *PLAXIS Version 8 Dynamic Manual*. Plaxis. 2009.

[68] J. M. Roesset. "Fundamentals of Soil Amplification, in: Seismic Design for Nuclear Power Plants". In: *The MIT Press, Cambridge, pp. 183-244* (1970).

[69] C. Visone, E. Bilotta, and F. Santucci. "Remarks on Site Response Analysis by Using Plaxis Dynamic Module". In: *Plaxis Bulletin* (2008).