Pontifícia Universidade Católica
do Rio de Janeiro

**Leonardo Seperuelo Duarte**

# TopSim: A plugin-based framework for large-scale numerical analysis

**Tese de Doutorado**

Thesis presented to the Programa de Pós–Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor em Informática.

Advisor: Prof. Waldemar Celes Filho

Rio de Janeiro
September, 2016

Pontifícia Universidade Católica
do Rio de Janeiro

**Leonardo Seperuelo Duarte**

# TopSim: A plugin-based framework for large-scale numerical analysis

Thesis presented as partial fulfillment of the requirements for the degree of Doutor to the Programa de Pós–graduação em Doutorado em Informática of the Departamento de Informática da PUC-Rio.Approved by the Examining Commission signed below.

**Prof. Waldemar Celes Filho**
Advisor
Departamento de Informática – PUC-Rio

**Prof. Marcelo Gattass**
Departamento de Informática – PUC-Rio

**Prof. Ivan Fábio Mota de Menezes**
Departamento de Engenharia Mecânica – PUC-Rio

**Prof. Glaucio Hermogenes Paulino**
– Georgia Tech University

**Prof. Renato Fontoura de Gusmão Cerqueira**
IBM Research – Brazil

**Prof. Marcos de Oliveira Lage Ferreira**
– UFF

**Prof. Márcio da Silveira Carvalho**
Coordinator of the Centro Técnico Científico – PUC-Rio

Rio de Janeiro, September 9th, 2016

**Leonardo Seperuelo Duarte**

Graduated in Electrical Engineering at Pontifícia Universidade Católica do Rio de Janeiro. Obtained a Master´s degree in Computer Science at Pontifícia Universidade Católica do Rio de Janeiro, acting in the area of grains simulation in GPGPU programming. Did his PhD in Computer Science at Pontifícia Universidade Católica do Rio de Janeiro with a full CNPq scholarship, including a collaboration with the University of Illinois as a visiting researcher. While doing his Masters and PhD, he worked as a researcher at Tecgraf/PUC-Rio developing simulators to design flexible lines and anchor systems for oil and gas industry.



Bibliographic data

S. Duarte, Leonardo

TopSim: A plugin-based framework for large-scale numerical analysis / Leonardo Seperuelo Duarte; advisor: Waldemar Celes Filho. – Rio de Janeiro : PUC-Rio, Departamento de Informática, 2016.

v., 91 f: il. ; 29,7 cm

1. Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Sistema baseado em plugin. 3. Análise Numérica. 4. Otimização Topológica. 5. Solver elemento-por-elemento. 6. Análise em larga escala. 7. Método dos Elementos Finitos. 8. Computação Paralela. Computação Distribuída. I. Celes, Waldemar. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

## Acknowledgements

To my lovely wife, Tabatha Fonseca Seperuelo Duarte, for all the partnership and devotion by my side during all this journey. Without you, the experience of living in another country would had not been an adventure.

To my family, for all the support they have given throughout my life. To my father, Célio de Oliveira Duarte, my mother, Norma Seperuelo Duarte, my brothers and sisters, Ricardo, Eduardo, Roberta and Camila, and my newborns nephew and niece, Rafael and Antonia. In memoriam to my grandmother, Noemia Martins Seperuelo, whose life was dedicated to our family.

To my advisor, Waldemar Celes Filho, and my co-advisors Ivan F. M. de Menezes and Glaucio H. Paulino without whom the research would not be possible, specially during my collaboration with the University of Illinois as a visiting researcher. Thank you for motivating me throughout these whole years as my advisers, teachers and leaders.

To Rodrigo Espinha and Anderson Pereira, for all the collaboration and teamwork during this project.

To Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ), for making this research possible.

To Tecgraf/PUC-Rio laboratory, for giving me the opportunity and support to face such challenges and learning with them.

To all my friends for their support and friendship.

## Abstract

S. Duarte, Leonardo; Celes, Waldemar. **TopSim: A plugin-based framework for large-scale numerical analysis**. Rio de Janeiro, 2016. 91p. PhD Thesis – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Computational methods in engineering are used to solve physical problems that do not have analytical solution or their perfect mathematical representation is unfeasible. Numerical techniques, including the largely used finite element method, require the solution of linear systems with hundreds of thousands equations, demanding high computational resources (memory and time). In this thesis, we present a plugin-based framework for large-scale numerical analysis. The framework is used as an original tool to solve topology optimization problems using the finite element method with millions of elements. Our strategy uses an element-by-element technique to implement a highly parallel code for an iterative solver with low memory consumption. Besides, the plugin approach provides a fully flexible and easy to extend environment, where different types of applications, requiring different types of finite elements, materials, linear solvers, and formulations, can be developed and improved. The kernel of the framework is minimum with only a plugin manager module, responsible to load the desired plugins during runtime using an input configuration file. All the features required for a specific application are defined inside plugins, with no need to change the kernel. Plugins may provide or require additional specialized interfaces, where other plugins may be connected to compose a more complex and complete system. We present results for a structural linear elastic static analysis and for a structural topology optimization analysis. The simulations use elements Q4, hexahedron (Brick8), and hexagonal prism (Honeycomb), with direct and iterative solvers using sequential, parallel and distributed computing. We investigate the performance regarding the use of memory and the scalability of the solution for problems with different sizes, from small to very large examples on a single machine and on a cluster. We simulated a linear elastic static example with 500 million elements on 300 machines.

## Keywords

Plugin-based Framework; Numerical Analysis; Topology Optimization; Element-by-Element Solver; Large-scale Analysis; Finite Element Method; Parallel Computing; Distributed Computing.

## Resumo

S. Duarte, Leonardo; Celes, Waldemar. **TopSim: Um sistema baseado em plugin para análise numérica em larga escala**. Rio de Janeiro, 2016. 91p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Métodos computacionais em engenharia são usados na solução de problemas físicos que não possuem solução analítica ou sua perfeita representação matemática é inviável. Técnicas de métodos numéricos, incluindo o amplamente usado método dos elementos finitos, podem exigir a solução de sistemas lineares com centenas de milhares de equações, demandando altos recursos computacionais (memória e tempo). Nesta tese, nós apresentamos um sistema baseado em *plugins* para análise numérica em larga escala. O sistema é usado como uma ferramenta original na solução de problemas de otimização topológica usando o método dos elementos finitos com milhões de elementos. Nossa estratégia utiliza uma técnica elemento-por-elemento para implementar um código altamente paralelo para um *solver* iterativo com baixo consumo de memória. Além disso, a abordagem de *plugin* proporciona um ambiente completamente flexível e fácil de estender, onde diferentes aplicações, exigindo diferentes tipos de elementos finitos, materiais, *solvers* lineares e formulações podem ser desenvolvidos e melhorados. O *kernel* do sistema é mínimo, com apenas um módulo gerenciador de *plugin*, responsável por carregar os *plugins* desejados em tempo real usando um arquivo de configuração de entrada. Todas as funcionalidades necessárias para uma determinada aplicação são definidas dentro dos *plugins*, sem a necessidade de mudar o *kernel*. *Plugins* podem disponibilizar ou exigir interfaces adicionais especializadas, onde outros *plugins* podem ser conectados para compor um sistema mais complexo e completo. Nós apresentamos resultados para uma análise estrutural estática linear elástica e para uma análise estrutural de otimização topológica. As simulações utilizam elementos Q4, hexagonal (Brick8) e prisma hexagonal (Honeycomb), com solvers diretos e iterativos usando computação sequencial, paralela e distribuída. Nós investigamos o desempenho com relação ao uso de memória e escalabilidade da solução para problemas com diferentes tamanhos, de exemplos pequenos a muito grandes em apenas uma máquina e em um cluster. Foi simulado um exemplo de análise estática linear elástica com 500 milhões de elementos em 300 máquinas.

## Palavras-chave

Sistema baseado em plugin; Análise Numérica; Otimização Topológica; Solver elemento-por-elemento; Análise em larga escala; Método dos Elementos Finitos; Computação Paralela; Computação Distribuída.

# Contents

# List of figures

# List of tables

# 1
# Introduction

## 1.1
## Motivation

Computational methods in engineering are commonly used to solve physical problems that do not have an analytical solution or when a perfect mathematical representation of complex geometries, loads, boundary conditions, and material behavior is unfeasible. The physical nature of the problem is transformed into a mathematical abstraction, and numerical methods use arithmetic operations to compute the solution. Some analyses may consider the behavior of a continuum or the interaction of particle elements. The decision regarding the best model to be used requires a fine balance between algorithm design, efficiency, accuracy, and computational difficulties.

There are a wide variety of numerical techniques in continuum mechanics, including the largely used finite element method (FEM). This is one of the most important methods for numerical analysis, used to solve problems in different areas such as structural and fluid mechanics, thermal analysis and electromagnetism, by both academic and industry communities. Basically, the displacement-based FEM consists of subdividing the domain into small and convex regions, called elements, where a displacement field is prescribed. Then, the governing differential equation is satisfied in an approximated way and the error between the exact and approximated solutions is minimized within the domain. The original differential equation is then transformed into a system of linear algebraic equations whose solution corresponds to the displacements at the nodes of each element.

The level of accuracy required in a finite element analysis (FEA) depends basically on the number of elements, element type, and shape functions used within the elements. Regarding the size of the problem, it is easy to find models with hundreds of thousands elements. The computational cost to solve such large systems can be extremely high and it usually turns out to be the main performance and memory bottlenecks. Nowadays, with the development of modern machines and supercomputers, the challenge has become to design algorithms with an efficient use of memory and all the computer power

available, in general by means of parallel computing. Researchers have been working to push the limit further to millions and billions elements in order to solve real life problems.

Considering a more specific and complex engineering application that uses FEM as its main numerical tool, structural topology optimization has even more computational challenges. The method is used to find the optimum material distribution within a given domain, subject to loading, boundary conditions and design constraints, in such a way to minimize some performance measure. In the case of compliance minimization, the optimization process usually requires a finite element analysis which means that a linear system of equations must be solved in each iteration of the optimization process.

Accurate formulations of real life topology optimization problems require solving extremely large systems, as shown in Table 1.1. Furthermore, an important issue when implementing a topology optimization analysis is the ability to extend, develop and modify the code to solve more complex and large-scale problems. For example, by using a modular code it is easier to replace the current analysis method with a more suitable analysis package for solving a different problem. The formalism of this modular approach is crucial when one seeks to improve the analysis routines, change the objective function, modify the sensitivities, without changing the topology optimization formulation, including the material interpolation and regularization schemes (e.g. filters and other manufacturing constraints).

| Problem | No. of unknows |
|---|---|
| Very Large | 90 000 000 [1] |
| Largest | 343 800 000 [2] |
| Our Goal | 1 000 000 000 |

Table 1.1: Example of large-scale problems in topology optimization.

The combination of flexibility and scalability to solve large-scale problems in numerical analysis is very challenging. Using a very general solution, a well organized modular code may focus more in attending a variety of applications and lose its efficiency to solve specific large problems. On the other hand, a fast but hard coded implementation is also not desired because it would be useful only for specific types of applications.

## 1.2 Objectives

This work presents a plugin based framework for large-scale numerical analysis. The framework is designed to perform numerical analysis using the finite element method for engineering applications. One of the goals is to present an original tool capable of solving topology optimization problems with up to a billion of finite elements. The strategy consists of using an assembly-free technique to implement a highly parallel code for an iterative solver with low memory consumption. Moreover, the plugin approach provides a flexible and easy to extend environment, where different types of applications, requiring different types of finite elements, materials, linear solvers, and formulations, can be developed and improved.

We propose the architecture and the interface required to create and connect a plugin into the framework. Plugins are created to implement specialized algorithms to solve specific problems efficiently. Using a configuration input file, the framework is able to load at runtime a set of desired plugins to perform a specific numerical analysis. To achieve a good performance in large-scale simulations, we propose an element-by-element version of the PCG solver [3]. The global stiffness matrix is never assembled, and our matrix-vector multiplication is done in parallel for each node in the mesh, without race condition and no need of mesh coloring. The framework also includes a distributed implementation using domain decomposition and MPI (Message Passing Interface) to simulate problems on large clusters.

We present results for a structural static linear elastic analysis and for a structural topology optimization analysis. The simulations use three different types of elements, Q4, hexahedron (Brick), and hexagonal prism (Honeycomb), with direct and iterative solvers using sequential and parallel computing. We investigate the performance of problems with different sizes, from small to large examples in a single machine and in a cluster, considering the use of memory and computational resources. We compare our results with Abaqus [1], a well-known commercial tool, to check if the plugin approach could affect and degrade the performance scalability. We simulated a few iterations of a topology optimization example obtained with a billion of elements using 3000 machines in the Blue Waters Supercomputer [2].

The main contributions of this thesis are:

– A fully flexible and easy to extend numerical analysis framework, based on plugins;

[1] http://abaqus.software.polimi.it/v6.14/index.html
[2] http://bluewaters.ncsa.illinois.edu

– An assemble-free iterative solver for the linear system of equations, with low memory consumption and a scalable performance;
– The capability to run extremely large-scale problems (e.g. a billion of elements) using supercomputers.
– A functionality that allows the user to configure the plugins dynamically in order to simulate different problems and try to achieve the best performance with both small (using direct solvers) and large (using iterative solvers) problems.

## 1.3 Related Work

### 1.3.1 Finite Element Analysis

Tools for finite element analysis have been developed for many years by a wide range of different research groups around the world. We can find some common features and concerns in many publications, such as modularity, flexibility to extend the code to different applications, and parallel solutions to speedup the simulation. The FEMOOP code [4,5], for example, was developed in the beginning of 90s as a standard C object-oriented implementation of the finite element method, originally created for numerical analysis of structures considering both linear and nonlinear behaviors. Couple of years later, the code was completely reformulated in C++, increasing the object-oriented capability and allowing an easy collaboration among different researchers during the development and expansion of the code. Today, there is also a parallel version [6] developed with MPI and PVM (Parallel Virtual Machine). The publications describe the class organization, geometry representation, material nonlinearities, and details about nonlinear solution schemes. The parallel version focus both on mesh generation and FEM solution, using domain decomposition techniques and an element-by-element solver. They present results with up to 9,000 tetrahedral elements running in 6 processors.

In 2002, Lee Margetts [7] presented a parallel finite element code in his PhD Thesis. Today, he is one of the leading developers of the ParaFEM project [3], which is an open source platform to provide a foundation for teaching and researching. The code is an extension of the standard Fortran finite element software presented in the text book by Smith, Griffiths and Margetts [8]. As a main feature, the parallel version of the program looks identical to their serial counterparts, packing the parallel processing in self-contained subroutines with

[3] http://parafem.org.uk/

MPI. The authors claim that anyone with the ability to modify a sequential finite element code can easily modify its parallel version. The system of equations is solved using an iterative technique with an element-by-element approach. It has been used to solve problems with more than a billion degrees of freedom, scaling on up to 32,000 cores using modern supercomputers.

The Trilinos Project [9] aims to facilitate the development and integration of mathematical software libraries within an object-oriented framework at Sandia Laboratory [4]. The goal is the solution of large-scale, complex multi-physics engineering and scientific problems. Two fundamental issues are addressed for these problems: to provide a streamlined process and a set of tools for development of new algorithmic implementations, and to promote interoperability of independently developed software. A two-level software structure was designed based on the idea of collections of packages. A package is an unit usually developed by a small team of experts in a particular area, such as algebraic preconditioners and nonlinear solvers. The packages are created underneath a top level, which provides an uniform interface. It was motivated by a recognition that a modest degree of coordination across these efforts could have a large positive impact on the quality and usability of the software produced, intended to enhance research, development and integration of new algorithms into applications. The fundamental idea of the project is that the effort required to develop new parallel solvers can be substantially reduced, because the common infrastructure provides a good starting point for new developments. Furthermore, interoperability of packages supports a broad set of new solvers for coupled multi-physics applications that are a critical requirement for advanced high-fidelity simulations.

The library deal.II [10] is a general purpose finite element code written in C++. It uses object-oriented and data encapsulation techniques to break the analysis into smaller blocks that can be arranged to fit the user requirements. The library supports different applications covering a range of scientific areas and specific algorithms without imposing a rigid framework. The authors invested a lot of effort to avoid the computational costs associated with an abstract object-oriented architecture. The paper presents a detailed description of the abstractions chosen for defining the mesh, the numbering of degrees of freedom, the linear system solver, the input and output features, and the interfaces to other softwares, like visualization tools. The work presents many publications showing results obtained with the library in different fields such as mathematical error analysis for finite element discretization, incompressible and compressible fluid flow, fluid-structure interaction, magnetohydrody-

[4] http://www.sandia.gov/

namics, biomedical inverse imaging problems, fuel cell modeling, simulation of crystal growth, and nuclear reactor simulations. They present results of a compressible flow and a biomedical imaging application with up to 68,000 lines of code on top of deal.II.

ViennaX is a plugin execution framework for scientific simulations presented by Weinbub et al. [11]. The framework functionality is based on the notion of a task, which is implemented as a plugin. It is an open source code designed for modularization and parallelization. The plugin system facilitates the utilization of existing functionalities as well as new implementations. A task can be defined with arbitrary data dependencies which are used by ViennaX to build a task graph. By using MPI, the framework executes the dependence graph based on either a serial or a parallel computing. An input configuration file in XML (Extensible Markup Language) [5] indicates the plugins to be used during the course of execution. Results obtained by this work include applications based on the Mandelbrot set and partial differential equations with up to 67 million degrees of freedom simulated on a cluster.

The GeMA [12] (Geo Modeling Analysis) framework, presented by Carlos Augusto in his DSc dissertation in April of 2016, was designed to support the development of new multiphysics simulators, and the integration with pre-existing simulators. The main goal is to speed up code development, with support functions for a extensible and modular environment, allowing engineers to focus on the physical problem formulation.



### 1.3.2 Topology Optimization Analysis

The theory about topology optimization has recently received much attention when compared to practical challenges in developing efficient and modular codes, especially when the goal is to solve complex and large-scale problems with millions of degrees of freedom (dofs). Most computationally oriented papers in topology optimization, which rely on the finite element method, employ either triangular or quadrilateral elements with linear interpolation of the displacement field and constant density field. However, it has been shown that these elements suffer from numerical instabilities such as checkerboard patterns [13–15]. Although regularization schemes such as filtering may be used to suppress the numerical instabilities, these techniques often involve heuristic parameters that can augment the optimization problem and can lead to significant cost increases [16, 17]. Recently, polygonal discretization have been proposed to achieve stable topology optimization formulations using low or-

[5] https://www.w3.org/TR/REC-xml/

der elements (degrees of freedom sampled at the nodes and constant design variable within the element) as reported in references [16, 18, 19].

Several educational codes, e.g. 99-line [20] and 88-line [21], serve as a resource to the community and were developed to solve specific topology optimization problems, with an implementation that mixes the analysis routines and the optimization formulation. The main goal of these codes was to introduce the concept of topology optimization in a compact and simple way, rather than achieving flexibility when solving more general problems. Different versions of these codes are necessary to be implemented if we need to change, for example, the finite element analysis to deal with polygonal meshes, or if we want to change the optimization formulation, e.g. from compliance minimization to compliant mechanisms [22].

The educational code `PolyTop`, presented by Talischi et al. in 2012 [18], features a modular Matlab structure to solve topology optimization problems. Its structure separates the analysis routine and the optimization algorithm from the specific choice of the topology optimization formulation. The finite element model and the topology optimization parameters are passed to the `PolyTop` kernel by a Matlab script, allowing the user to investigate different problems without changing the basic kernel. The formalism offered by this decoupling approach provides an easy way to extend, develop and modify the code to test different topology optimization formulations. Moreover, the `PolyTop` code addresses practical issues regarding the use of polygonal meshes in arbitrary design domains (rather than boxes). Numerical results are presented using polygonal meshes with up to 120 thousand elements [18]. However, when dealing with large-scale problems using the `PolyTop` code, it is difficult to have total control of memory allocation with a code in Matlab. Efficient algorithms are necessary to build the filtering sparse matrix, used to map the design variables into the analysis parameters. The process of building this matrix is a memory bottleneck of the current `PolyTop` code, while the solution of the linear system of equations within the finite element analysis module is another bottleneck related to the computational performance.

The number of publications focused on solving large-scale problems in topology optimization has considerably increased – see, for example, the review paper by Deaton and Grandhi [23]. In 2012, Suresh introduced an algorithm for large-scale 3D problems in topology optimization [1]. It is an extension of the 2D topological-sensitivity based method [24]. The 3D model explores the congruence between hexahedron/brick finite elements and modern multi-core computer architectures. Suresh presented numerical results with 700 thousand dofs, solved in 16 minutes in a CPU and, in 125 seconds in a GPU. He also

obtained results for relatively large-scale problems with 15 million dofs, solved in 19 hours using a CPU, and in 2 hours using a GPU. He solved an even greater problem, with 92 million dofs, which took 12 days of processing on the CPU - no results on the GPU were provided because of memory limitation. However, the work uses uniform grids (instead of unstructured meshes), which are susceptible to checkerboard pattern problems [16], as mentioned before. Recently, Aage and Lazarov implemented a parallel framework for topology optimization using the method of moving asymptotes [25]. They simulated fluids and solid mechanics problems with linear scalability up to approximately 800 CPUs on a Cray XT4/XT5 supercomputer. They presented results for a 3D mesh using linear hexahedron elements with almost 15M dofs spending approximately 5 minutes for each topology optimization iteration. Another recent work by Amir et al. [26] presents a computational approach to reduce the time for solving 3D structural topology optimization problems. They obtained performance improvement by exploiting specific characteristics of a multigrid preconditioned conjugate gradient (MGCG) solver.

Aage et al. [2] presented a parallel topology optimization framework using the PETSc library [6]. In this work, no mesh input file is read but the structured grid is automatically generated, and subdivided among the cluster nodes. The PETSc library is responsible for generating/partitioning the mesh, assembling the global stiffness matrix, performing the matrix-vector operations, and solving the linear system of equations. They presented results for the compliance minimization problem with more than 100 million elements, using 1800 cores.

In 2014, Yadav and Suresh [27] presented an assembly-free version of the deflated conjugate gradient (DCG) for solving large linear system of equations, where neither the stiffness matrix nor the deflation matrix is assembled. The novelty pursued in the paper is the use of assembly-free deflation. The authors claim that the solution is particularly well suited for large-scale problems and can be easily ported to multi-core CPU and GPU architectures. They solved a 50 million degree of freedom system on a single GPU card, equipped with 3 GBs of memory.

In a previous work, we presented PolyTop++ [28] that contributes for an efficient and modular code, capable of dealing with different topology optimization problems, addressing issues related to the use of polygonal meshes in arbitrary design domains, and offering a hierarchical modular structure, which is easy to be extended to different finite element solvers, different topology optimization formulations, and different physics. It consists of a

[6] https://www.mcs.anl.gov/petsc/

C++ and CUDA (a parallel computing model for GPUs) [29] alternative implementation to the serial PolyTop Matlab code, presented by Talischi et al. [18]. A relevant feature of PolyTop++ is an easy selection between the CPU or the GPU parallel solution as an input parameter. The software takes advantage of the C++ programming language and the CUDA model to design algorithms with efficient memory management, capable of solving large-scale problems, besides exploring its object-oriented flexibility in order to provide a modular scheme. The work describes the implementation of different solvers for the finite element analysis, including both direct and iterative solvers, and an iterative assembly-free solver. Numerical results for problems with 40 million degrees of freedom both in 2D and 3D are presented.

The combination of flexibility and high performance has always been challenging. Many recent publications try to develop a modular code, suitable to simulate different applications and capable of handling millions, or even billions, of degrees of freedom. This thesis contributes exactly in this direction. Despite previous publications of plugin based frameworks for numerical analysis [11], this is a new and unique approach to solve engineering problems. In our opinion, this is the best way to guarantee a modular and fully flexible framework. In our case, the overhead cost paid for this generalization is very low, since the classic hierarchical structure of an usual objected-oriented C++ code is replaced by small and specific plugins defining the behavior of the element type instead of each element itself. Furthermore, using the Tops [30] library to represent the geometric model, we can manage large mesh data efficiently and provide a unified communication protocol between the plugins. Combined with a parallel and distributed assembly-free iterative solver, we can achieve good results for both small and extremely large-scale problems with a billion of elements.



## 1.4 Document Organization

The remainder of this work is organized as follows. Chapter 2 explains the theory and formulations of the finite element method and the topology optimization analysis. Chapter 3 describes the TopSim framework with its functionality to configure and load the plugins dynamically, all the plugins created to simulate a linear elastic static and a topology optimization analysis, using different finite element types and solvers for the linear system of equations. In Chapter 3, we also present results and a performance analysis of the framework with tests using only 1 machine with shared memory cores, investigating the efficiency of the solution with both small and large mesh sizes.

Chapter 4 describes the extension of the TopSim framework for a distributed parallel computing. We present the plugins created for domain decomposition and communication between subdomains using MPI, while OpenMP is used for a parallel computing inside each subdomain. In Chapter 4 we also present the results and a performance analysis for simulation of hundreds of millions and even a billion of finite elements on the Blue Waters Supercomputer.

# 2
# Theory and Formulations

## 2.1
## Finite Element Method

The finite element method is one of the most popular numerical methods for solving physical problems in continuum mechanics. The method approximates the physical governing equations, usually partial-differential-equations (PDEs), as a linearized system of algebraic equations. For example, considering the static linear elastic problem illustrated in Figure 2.1, the domain is discretized into a finite element mesh with the displacement field $\mathbf{U}$ approximated by $\mathbf{U^e}$ as follows:

$$\mathbf{U^e}(\mathbf{x}) = \sum_{i=1}^{n} U_i N_i(\mathbf{x}) = \mathbf{N}\hat{\mathbf{U}}^e \tag{2-1}$$

where $n$ is the number of nodes in the element, $N_i$ is the shape function associated to node $i$, and $\hat{\mathbf{U}}^e$ is the nodal displacement of element $e$.

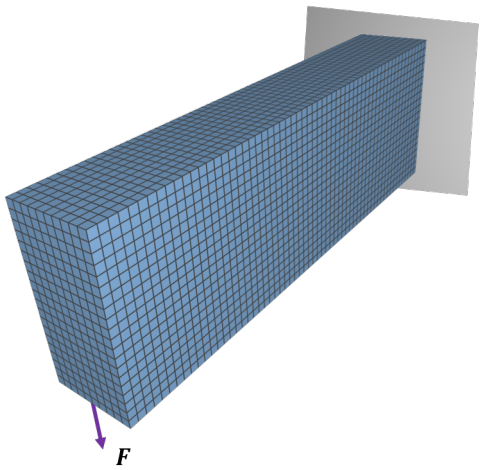



Figure 2.1: Finite element model of a cantilever beam with linear elastic material.

The element stiffness matrix is evaluated by performing a numerical integration over the element volume $V^e$ at the Gauss points, as below:

$$\mathbf{K^e} = \int_{V^e} \mathbf{B^T C B} dV \tag{2-2}$$

with $\mathbf{B}$ representing the strain-displacement matrix defined as:

$$\begin{gathered} \varepsilon = \nabla \mathbf{U}^e \quad \Rightarrow \quad \varepsilon = \nabla \mathbf{N} \hat{\mathbf{U}}^e \\ \therefore \quad \mathbf{B} = \nabla \mathbf{N}; \end{gathered} \tag{2-3}$$

where $\nabla$ is a differential operator. For the hexahedron element (also known as brick element), used in many examples in this work, the stresses can be written as:

$$\begin{gathered} \sigma = \mathbf{C}\varepsilon \\ \mathbf{C} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \end{gathered} \tag{2-4}$$

where $\mathbf{C}$ is the material constitutive matrix, $E$ is the Young modulus, and $\nu$ is the Poisson ratio. For the sake of completeness, considering the specific brick element, the following matrices are defined:

$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ \varepsilon_{xy} \\ \varepsilon_{xz} \\ \varepsilon_{yz} \end{bmatrix}; \qquad \mathbf{B}_i = \begin{bmatrix} \partial N_i/\partial x & 0 & 0 \\ 0 & \partial N_i/\partial y & 0 \\ 0 & 0 & \partial N_i/\partial z \\ \partial N_i/\partial y & \partial N_i/\partial x & 0 \\ 0 & \partial N_i/\partial z & \partial N_i/\partial y \\ \partial N_i/\partial z & 0 & \partial N_i/\partial x \end{bmatrix}. \tag{2-5}$$

The global stiffness matrix is obtained by assembling all the element stiffness matrices. Each global degree of freedom related to a node in the mesh will receive the contribution from every element connected to it. Together with the global forces **F** applied to the domain (Figure 2.1) and the global nodal displacement vector **U**, we arrive at the well-known linear system of equations:

$$\mathbf{KU} = \mathbf{F}. \tag{2-6}$$

for linear problems.

## 2.2 Topology Optimization

Topology optimization is an area of the structural analysis that usually requires the use of the finite element method. The main goal of topology optimization is to find the most efficient material distribution inside a domain $\Omega \subseteq \mathbb{R}^N$ subject to loads and boundary conditions and minimizing some performance measure. Many topology optimization formulations can be defined in the form:

$$\begin{cases} \min\limits_{p} \quad f(\rho, \mathbf{u}(\rho)) \\ \\ s.t.: \quad g_i(\rho, \mathbf{u}(\rho)) \leq 0, \qquad i = 1, ..., N_c \end{cases} \tag{2-7}$$

where $f$ and $g_i$ are, respectively, the objective and constraint functions, $N_c$ is the number of constraints, and $\rho$ represents the design variables (e.g. densities).

Here, we consider linear elasticity as the governing state equation, which is typical in continuum structural optimization. The solution satisfies the variational problem:

$$\int_\Omega m_E(\rho)\mathbf{C}\nabla\mathbf{u} : \nabla\mathbf{v}d\mathbf{x} = \int_{\Gamma_N} \mathbf{t} \cdot \mathbf{v}ds, \qquad \forall \mathbf{v} \in \mathcal{V} \tag{2-8}$$

$$\mathcal{V} = \{\mathbf{v} \in \mathbf{H}^1(\Omega; \mathbb{R}^d) : \mathbf{v}|\Gamma_D = 0\} \tag{2-9}$$

where $m_E$ is a material interpolation function, $\mathbf{C}$ is the stiffness tensor, $\Gamma_N$ and $\Gamma_D$ are portions of $\partial\Omega$ where, the non-zero traction $\mathbf{t}$ and the displacements are specified, respectively, and $\mathcal{V}$ is the space of candidate functions.

When the objective function is the structural compliance [1], the optimization problem can be expressed in its discrete form as:

$$\begin{cases} \min\limits_{z} \quad \mathbf{F}^T\mathbf{U} \\ \\ s.t.: \quad \frac{\mathbf{A}^T m_V(\mathbf{Pz})}{\mathbf{A}^T\mathbf{1}} - \overline{v} \leq 0 \end{cases} \tag{2-10}$$

where $\mathbf{F}$ is the nodal load vector, which is independent of the design variables $\mathbf{z}$, $\mathbf{U}$ is the nodal displacement vector that arise from the solution of the system of equilibrium equations $\mathbf{K}(m_E(\mathbf{z})))\mathbf{U} = \mathbf{F}$, $\mathbf{K}$ is the global stiffness matrix, $\mathbf{A}$ is the vector of element volumes, $m_V$ is the volume interpolation function, $\mathbf{P}$ is the matrix that maps the design variables $\mathbf{z}$ into the element densities $\mathbf{y}$ by means of $\mathbf{y} = \mathbf{Pz}$, $\mathbf{1}$ denotes an array of unit entries, and $\overline{v}$ is an input parameter which defines the design volume fraction.

In the so-called SIMP (Solid Isotropic Material with Penalization) approach [16, 18, 20, 21, 23, 31, 32], the element material properties are considered constant and the element densities are penalized as follows:

$$m_E(\rho) = \varepsilon + (1-\varepsilon)\rho^p, \qquad m_V(\rho) = \rho, \qquad \varepsilon = \rho_{min} \tag{2-11}$$

where $p$ is the penalty parameter, and $\varepsilon$ the Ersatz parameter. The SIMP method is very popular and has great acceptance in the solution of various types of problems. However, alternative well-known methods can also be found in the literature, such as level-set [23, 33–35], phase-field [23, 33, 36], and topological-sensitivity Pareto-optimal [1, 24] based methods.

Figure 2.2 describes the structure of the topology optimization method implemented in this work to solve the compliance minimization problem. The boxes highlighted in yellow represent the main steps of the optimization algorithm and the convergence is achieved when the change between the element densities in subsequent iterations lies within the tolerance level and the final topology is obtained.

The flowchart in Figure 2.2 also shows the decoupling approach between the FE analysis stage, the constraint stage, and the interpolations functions chosen for mapping the sensitivities to the design variables. The structure of

[1] Compliance minimization is the objective function presented in the original code `PolyTop` [18].

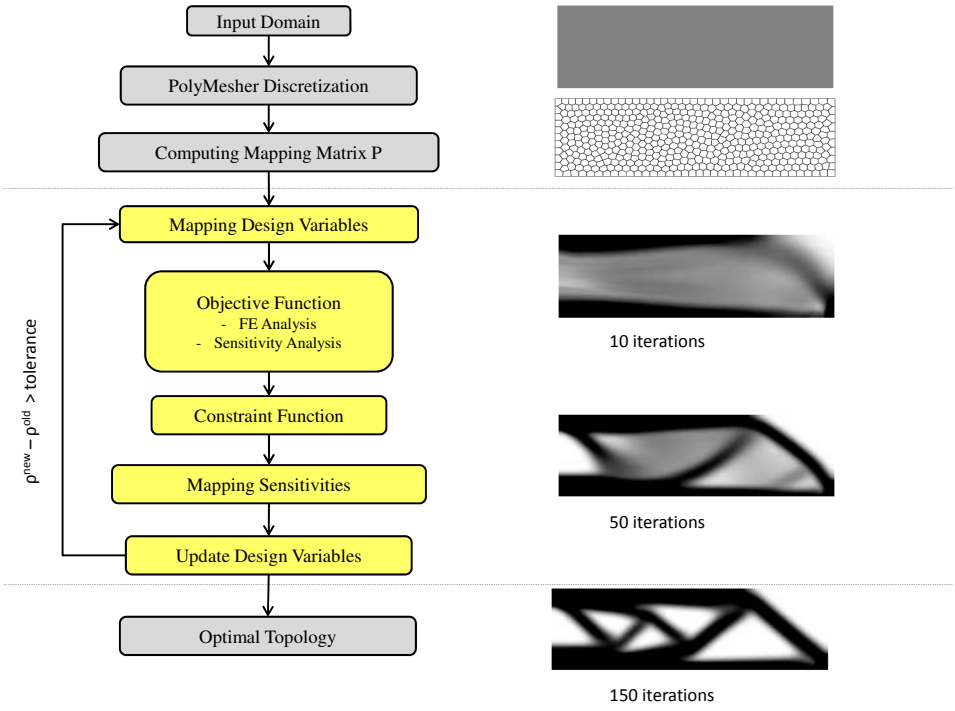



Figure 2.2: Topology optimization steps for the compliance minimization problem.

the discrete optimization problem allows separation of the analysis routine from the particular topology optimization formulation. The analysis functions do not need to know about the choice of interpolation functions which corresponds to the choice of sizing parametrization or the mapping $\mathbf{P}$ that places constraints on the design space. Therefore, a general implementation of topology optimization in the context of this discussion must be structured in such way that the finite element routines contain no information related to the specific topology optimization formulation. Because of this, we define the analysis module as a collection of functions that compute the objective and constraint functions. This module communicates with the finite element module to access the mesh information. An advantage of this approach is that the analysis functions can be extended, developed and modified independently of any modification in the topology optimization formulation.

In addition, certain quantities used in the analysis module, such as element areas $\mathbf{A}$ and local stiffness matrices $\mathbf{K}_e$, as well as connectivity of the global stiffness matrix $\mathbf{K}$, need to be computed only once in the course of the optimization algorithm, as well as the mapping matrix $\mathbf{P}$. To use a gradient-based optimization algorithm for solving the discrete problem, we

must compute the derivatives of the cost functions with respect to the design variables **z**. The sensitivity analysis can be separated along the same lines discussed above. The analysis functions compute the sensitivities of the cost functions with respect to their own internal parameters. Note that the analysis functions only compute the sensitivities of a certain function $g_i$ with respect to its internal parameters **E** and **V**, i.e.:

$$\frac{\partial g_i}{\partial \mathbf{z}} = \frac{\partial \mathbf{E}}{\partial \mathbf{z}} \frac{\partial g_i}{\partial \mathbf{E}} + \frac{\partial \mathbf{V}}{\partial \mathbf{z}} \frac{\partial g_i}{\partial \mathbf{V}}. \tag{2-12}$$

For the compliance problem, $f = \mathbf{F}^T\mathbf{U}$, therefore we can write:

$$f = \mathbf{F}^T\mathbf{U} - \lambda^T(\mathbf{K}\mathbf{U} - \mathbf{F}) \tag{2-13}$$

where the second term in the right hand side is equal to zero (i.e. $\mathbf{K}\mathbf{U} = \mathbf{F}$) and $\lambda$ is an adjoint vector. Differentiating the compliance, we have:

$$\begin{gathered}
\tfrac{\partial f}{\partial x} = \mathbf{F}^T \tfrac{\partial \mathbf{U}}{\partial x} - \lambda^T(\tfrac{\partial \mathbf{K}}{\partial x}\mathbf{U} + \mathbf{K}\tfrac{\partial \mathbf{U}}{\partial x}) \\
\tfrac{\partial f}{\partial x} = (\mathbf{F}^T - \lambda^T\mathbf{K})\tfrac{\partial \mathbf{U}}{\partial x} - \lambda^T(\tfrac{\partial \mathbf{K}}{\partial x}\mathbf{U}) \\
\text{choosing } \lambda \text{ such that} \quad \mathbf{F}^T - \lambda^T\mathbf{K} = 0, \quad \text{we obtain:} \\
\mathbf{F} - \mathbf{K}\lambda = 0 \\
\mathbf{F} = \mathbf{K}\lambda \Rightarrow \lambda = \mathbf{U} \\
\therefore \tfrac{\partial f}{\partial x} = -\mathbf{U}^T \tfrac{\partial \mathbf{K}}{\partial x}\mathbf{U}.
\end{gathered} \tag{2-14}$$

This means that the FE function that computes the objective function also returns the negative of the element strain energies as the vector of sensitivities $\partial f/\partial \mathbf{E}$ (for more details, please refer to [37]):

$$\frac{\partial f}{\partial \mathbf{E}_e} = -\mathbf{U}^T \frac{\partial \mathbf{K}}{\partial \mathbf{E}_e}\mathbf{U} = -\mathbf{U}^T\mathbf{K}_e\mathbf{U}, \qquad \frac{\partial f}{\partial \mathbf{V}_e} = 0. \tag{2-15}$$

The remaining terms in Equation(2-12) depend on the formulation, i.e., how the design variables $\mathbf{z}$ are related to the analysis parameters. For example, if $\mathbf{E} = m_E(\mathbf{Pz})$ and $\mathbf{V} = m_V(\mathbf{Pz})$ we can write:

$$\frac{\partial \mathbf{E}}{\partial \mathbf{z}} = \mathbf{P}^T \mathbf{J}_{m_E}(\mathbf{Pz}), \qquad \frac{\partial \mathbf{V}}{\partial \mathbf{z}} = \mathbf{P}^T \mathbf{J}_{m_V}(\mathbf{Pz}) \tag{2-16}$$

where $\mathbf{J}_{m_E}(y) := diag(m_E^{'}(y_1), \ldots, m_E^{'}(y_N))$ is the Jacobian matrix of map $m_E$, and $\mathbf{J}_{m_V}(y) := diag(m_V^{'}(y_1), \ldots, m_V^{'}(y_N))$ is the Jacobian matrix of map $m_V$. The evaluation of expression (2-12) is carried out outside the analysis routine and the result, $\partial g_i / \partial \mathbf{z}$, is passed to the optimizer to update the values of the design variables.

# 3
# Plugin-based Framework

## 3.1
## Overview

The TopSim framework is a tool to perform large-scale numerical analysis. Its plugin approach provides a fully flexible and easy to extend environment, capable of solving different types of numerical problems. The kernel of the framework has a very simple and basic infrastructure with only a plugin manager module, responsible for loading the desired plugins at runtime using an input configuration file. All the features required for a specific application must be defined inside plugins, with no need to change the kernel. Plugins may provide or require additional specialized interfaces where others plugins may be connected to compose a more complex and complete system.

The model representation is the only framework component that cannot be defined as a plugin. We decided to use the TopS library [30] to handle all the information about the model. TopS is a compact topological data structure developed for finite element mesh representation. It is capable of reducing the required storage space while being able to retrieve all the topological adjacency relationships in a constant time or in a time proportional to the number of retrieved entities. By using TopS as the only model representation, we can manage large mesh data efficiently and provide a unified communication protocol between the plugins. TopS can store all important model data shared among different plugins during the analysis. The plugins handle only specialized algorithms and their private data, while TopS is used as a bridge to update or retrieve any model data modified by any plugin.

In Figure 3.1, we show how plugins and TopS are combined to solve a static linear elastic problem. The Host application calls the Static analysis plugin. This plugin handles the input and output processes during the entire simulation by using reading and writing plugins. The Reader plugin loads an input file with the model, populates TopS with all the mesh information, and creates nodes, elements, and materials. The Static plugin calls a linear behavior plugin, which requires an integrator plugin to compute the external forces and a solver plugin to assemble and solve the linear system of equations,

exemplified here by the Load Control and the Umfpack plugins, respectively. The model attributes, e.g. nodes and the elements of the mesh, dof indices, node displacements, and right hand side of the linear system of equations, are interchanged among plugins through the TopS library.

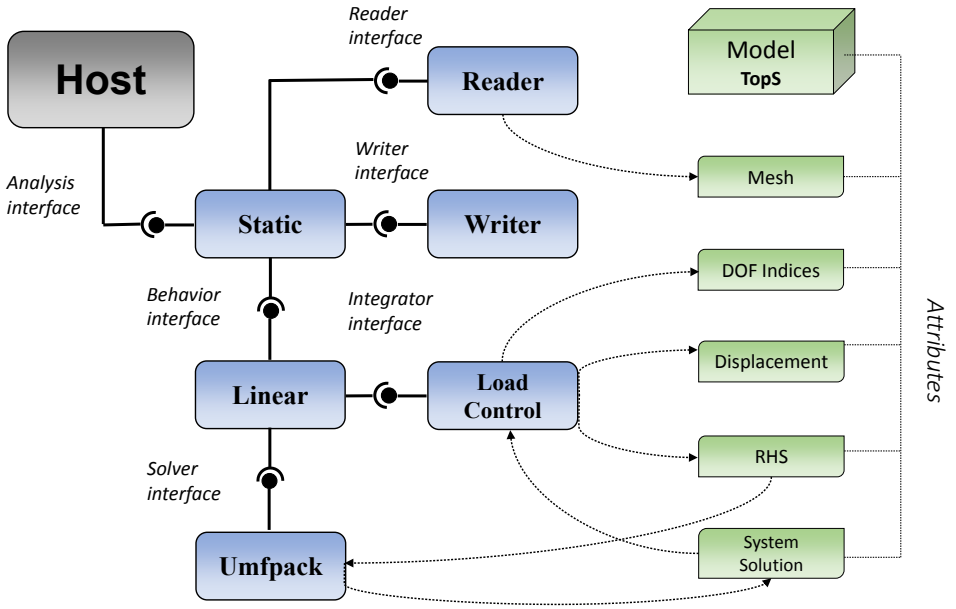



Figure 3.1: Example of a framework configuration to solve a static linear elastic problem.

Advantages of using a single model representation:

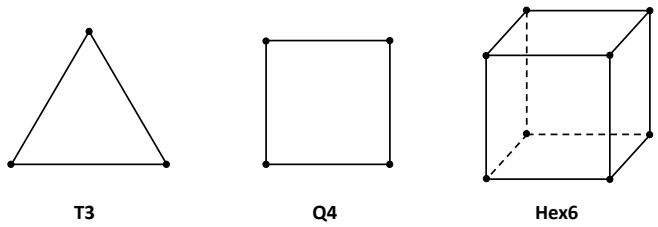
- uniform unity;
- efficiency;
- easy of sharing info (the flexibility must be ensured by the chosen model representation).

## 3.2 Model Representation

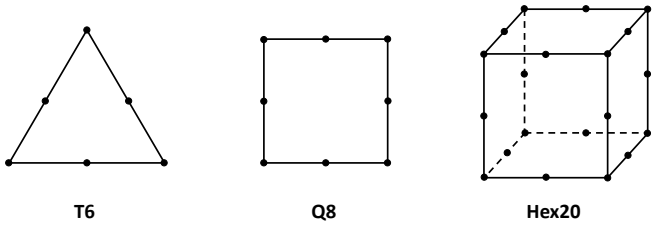
The topological data structure TopS was proposed by Celes et al. [30] in 2005 to represent finite element meshes. It provides a compact structure for manifold meshes, where elements and nodes are the only topological entities explicitly stored in memory. At the same time, the library is considered complete in the sense that all topological adjacencies can be retrieved in a time proportional to the entities consulted.

The element entity represents any finite element type with an ordered set of nodes (template). The main element types used in a typical finite element analysis are defined in TopS, such as triangles

(TOP_ELEMENT_T3), tetrahedrons (TOP_ELEMENT_TET4), and hexahedrons (TOP_ELEMENT_HEX8). Linear, quadratic, and higher order elements are considered (Figure 3.2). The node entity represents any finite element node and may be related to corner nodes, edge nodes, and interior nodes (Figure 3.2(b)). Beyond the traditional element representation, with its nodal connectivity, an element also has references to all its neighboring elements. The same happens with a node, which has reference to one element connected to it. Any other adjacency relationship in TopS can be derived from these previous ones.




3.2(a):




3.2(b):

Figure 3.2: Finite element types available in TopS. (a) Linear elements. (b) Quadratic elements.

TopS also supports the creation and association of a set of attributes to the nodes, the elements and the model. The attributes can be associated to all the entities of a same type, e.g. physical quantities related to each node or element in a numerical simulation. In this work we associate the displacement attribute to the nodes, the density attribute to the elements, and some global attributes, such as the simulation step, nodal support and nodal forces, to the model (Figure 3.3). This functionality in TopS is very useful to manage the communication of simulation data between mesh subdomains distributed in a cluster, as will be addressed in Chapter 4.

Algorithm 1 shows the API required by TopS to create attributes in the mesh. In this example, we create the DISPLACEMENT attribute for each

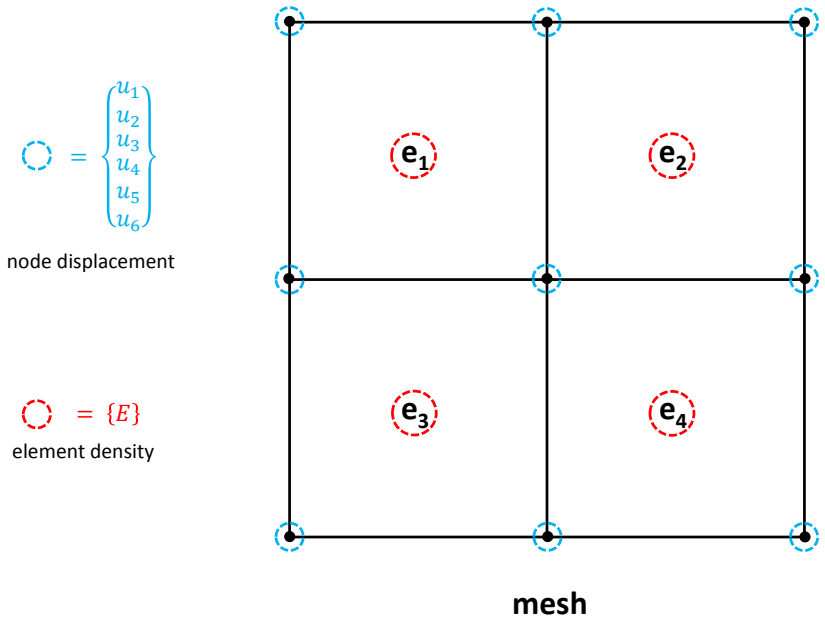



Figure 3.3: Example of nodes and elements attributes created in TopS.



node with 6 values to represent all 3 translational and 3 rotational directions. Each element has 1 value for the DENSITY attribute and the model holds some global attributes that are not related to every node or element. The NODAL_SUPPORT attribute is created with 7 values for each node that is not completely free ($n\_bc$), 1 value for the node id and 6 for the boundary condition of each node direction. In the same way, the NODAL_FORCE attribute is created for the nodes with any force applied to it($n\_f$), with 1 value for the node id and 3 for the force vector.

**Algorithm 1** TopS' API to create node, element and model attributes.

$CreateNodeAttrib(\text{"}DISPLACEMENT\text{"}, 6);$

$CreateElemAttrib(\text{"}DENSITY\text{"}, 1);$

$CreateModelAttrib(\text{"}NODAL\_SUPPORT\text{"}, n\_bc * 7);$
$CreateModelAttrib(\text{"}NODAL\_FORCE\text{"}, n\_f * 4);$

## 3.3 Plugin Manager

The kernel of the framework is composed by the Plugin Manager. It is responsible to load and register the plugins in the host system, aside from providing basic functions for querying plugin objects and interfaces. An input

configuration file in Lua [1] format is used to tell the host system which plugins must be load at runtime for the simulation. In Figure 3.4, we show the configuration file used to run the layout described in Figure 3.1.

```
Analysis =
{
  libname = "static"
}
Element = -- Element types
{
  {
    name = "Q4PLSTRESS",
    libname = "q4plstress",
  }
}
Material = -- Material types
{
  {
    name = "ISOTROPIC",
    libname = "isotropic",
  }
}
Reader =
{
  libname = "nfreader"
}
Writer =
{
  libname = "nfwriter"
}
Behavior =
{
  libname = "linear"
}
Integrator =
{
  libname = "loadcontrol"
}
LinearSystem =
{
  libname = "umfpack"
}
```

Figure 3.4: Configuration Lua file with the plugins selected by the user to run the simulation.

Every plugin must provide a public initialization function called *PluginOpen*, with the signature presented in Algorithm 2, which will be called right after the plugin is loaded. The function receives as parameters the type of service the plugin will be connected with, as shown in Figure 3.5 and represented in the configuration file by the blue color (*Analysis, Element, Material, etc*), the name of the plugin (e.g. *libname = static*, in the config file), and the TopS model. The *PluginOpen* function creates an instance (object) of the plugin class and register it in the Plugin Manager using the function AddInstance (Algorithm 2). The procedure makes all loaded plugins available to be used by the host system or by other plugins.

[1] https://www.lua.org/

**Algorithm 2** Basic functionality of the Plugin Manager.

$PluginOpen(type, name, model)$;

$AddInstance(type, name, obj)$;

$GetInstance(type, name, \&obj)$;

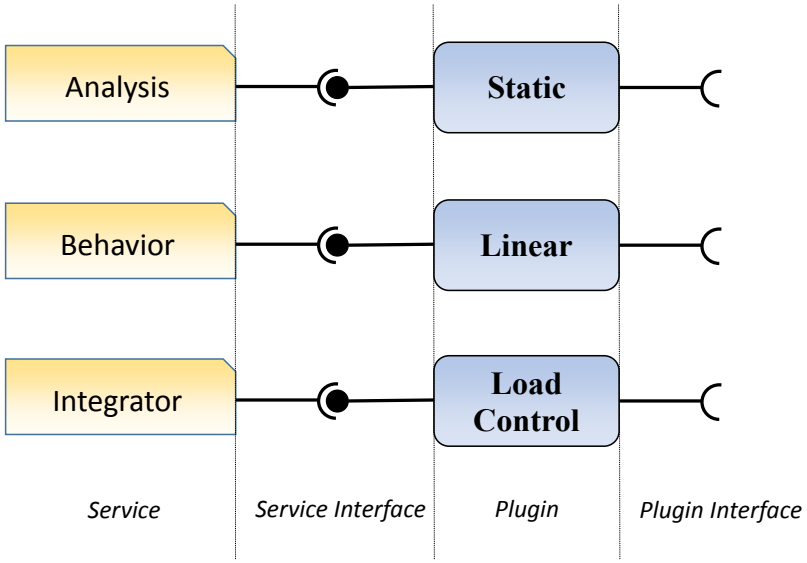



Figure 3.5: Examples of types of services and their interfaces to connect plugins.

Figure 3.5 shows the separation of services, service interfaces, plugins, and plugin interfaces. A service interface is the minimum implementation a plugin must define to be used. Such as an attempt to have an uniform API, all services must require at least the following basic functions: *Initialize, Finalize, Release, and QueryInterface*. The other important and crucial functions may vary depending on the main goal of each plugin, such as the function *Run* defined in the analysis service, and the function *Solve* in the linear system service. Each plugin may define an interface and require the connection of another plugin to perform the computation of a specific feature. Figure 3.6 depicts an example of the PCG [3] solver for the Linear System service, which requires the connection of a sparse matrix service and a preconditioner service, implemented in this case by the CSRMatrix and Jacobi plugins, respectively.

A plugin is responsible for defining if a required interface is crucial for its proper functioning or if it is just an option for a specific feature. In other words, a plugin must define the need of a connection or any other dependent plugin. In the case of the PCG, in Figure 3.6, a sparse matrix plugin must always be connected to make it work, however a preconditioner plugin may be

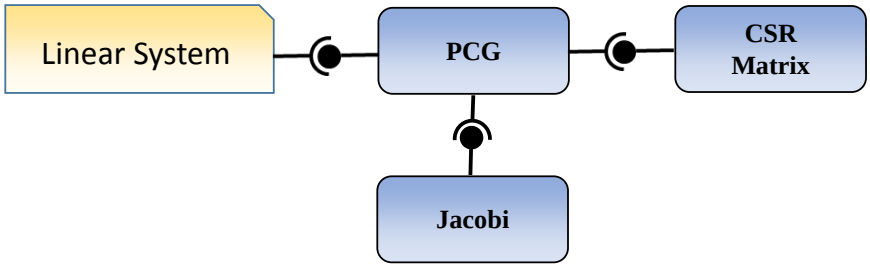



Figure 3.6: The PCG plugin connected to the Linear System service, and requiring the connection of a sparse matrix and a preconditioner plugin.

avoided and the PCG algorithm will still work. Figure 3.7 shows an example of how the PCG plugin uses a basic function of the Plugin Manager to require the dependent services.

```
if(!PluginManager::GetInstance("RowSparseMatrix", "pcg_csrmatrix", &m_A))
{
  topsim::SetError("PCG: Missing required plugin: \"RowSparseMatrix\".");
  throw false;
}

if(!PluginManager::GetInstance("RowSparsePreconditioner", "precond", &m_M))
{
  topsim::SetWarning("PCG: Missing plugin: \"RowSparsePreconditioner\".");
}
```

Figure 3.7: PCG plugins requiring the connection of dependent services.

The framework has also some useful tools to make easier the creation and connection of a new plugin, as shown in Algorithm 3. The $INTERFACE\_MEMBER$ macro helps the user with the declaration of some common functions that every plugin interface must have. As the *PluginOpen* function is always required, every plugin must also define which interfaces are to be connect with. The $INTERFACE\_CODE\#$ are compact macros that help the user to specify which interfaces the plugin need to be connected.

Figure 3.8 shows how the PCG plugin is created and loaded by the Plugin Manager, with the combination of the interface macro and the *PluginOpen* function.

By using this approach, we can reach a desirable level of flexibility to extend and specialize the code without defining long and fixed APIs or the need to change the kernel for every new desired feature. The interpretation of the configuration file is done by a code in Lua, and the dynamic libraries are loaded by functions from the current operational system, making the code portable for different platforms.

**Algorithm 3** Plugin Manager useful functions to define and connect a plugin.

$TOPSIM\_PLUGIN\_INTERFACE\_MEMBER$

$TOPSIM\_PLUGIN\_INTERFACE\_CODE1(P, T1)$
$TOPSIM\_PLUGIN\_INTERFACE\_CODE2(P, T1, T2)$
$TOPSIM\_PLUGIN\_INTERFACE\_CODE3(P, T1, T2, T3)$
$TOPSIM\_PLUGIN\_INTERFACE\_CODE4(P, T1, T2, T3, T4)$
$TOPSIM\_PLUGIN\_INTERFACE\_CODE5(P, T1, T2, T3, T4, T5)$

```
/* PLUG-IN INTERFACE */

TOPSIM_PLUGIN_INTERFACE_CODE1(PCG, LinearSystem)

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
bool PluginOpen(const char* type, const char* name, TopModel* model)
{
  topsim::LinearSystem* obj = new PCG(model);
  if(!topsim::PluginManager::AddInstance(type, name, obj))
    return false;

  return true;
}
```

Figure 3.8: Creating the PCG plugin using functions from the Plugin Manager.

## 3.4 Plugins

### 3.4.1 Analysis, Behavior, Integrator, and Numbering

The framework has three important service categories where the user can connect plugins to implement the type of analysis, the behavior of the problem, and the integration between the external environment and the mesh domain (Figure 3.9).

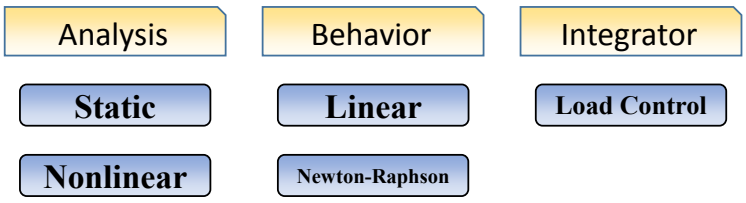



Figure 3.9: Three main services of the framework with their plugin definitions.

Figure 3.10 gives an idea of how these three plugins interact with each other to run the simulations. The Static plugin checks if the required plugins, i.e. reader, writer, and behavior, are available to be used. After reading the

input model, the initialization of the Linear behavior plugin is called to process all the data necessary to solve the analysis step. In this initialization, the Linear plugin also checks if an integrator plugin and a linear solver plugin are available. We also developed a numbering plugin responsible to traverse the entire model and designate an equation number for each free degree of freedom present in the model, considering the boundary condition information. This service category, called here as Numbering, can be very useful if, for some reason, the user needs to use a different number scheme. Once the dof numbers and the external forces are computed, the behavior plugin calls the initialization and the assembling process from the linear system service. Because there are different ways to make the assembly of the stiffness matrix and each type of solver may use a different storage scheme, this stage is defined in the solver plugin. After each simulation step, the nodal displacements are updated in the model, and depending on the framework configuration, it can be written in an output file by the writer plugin.



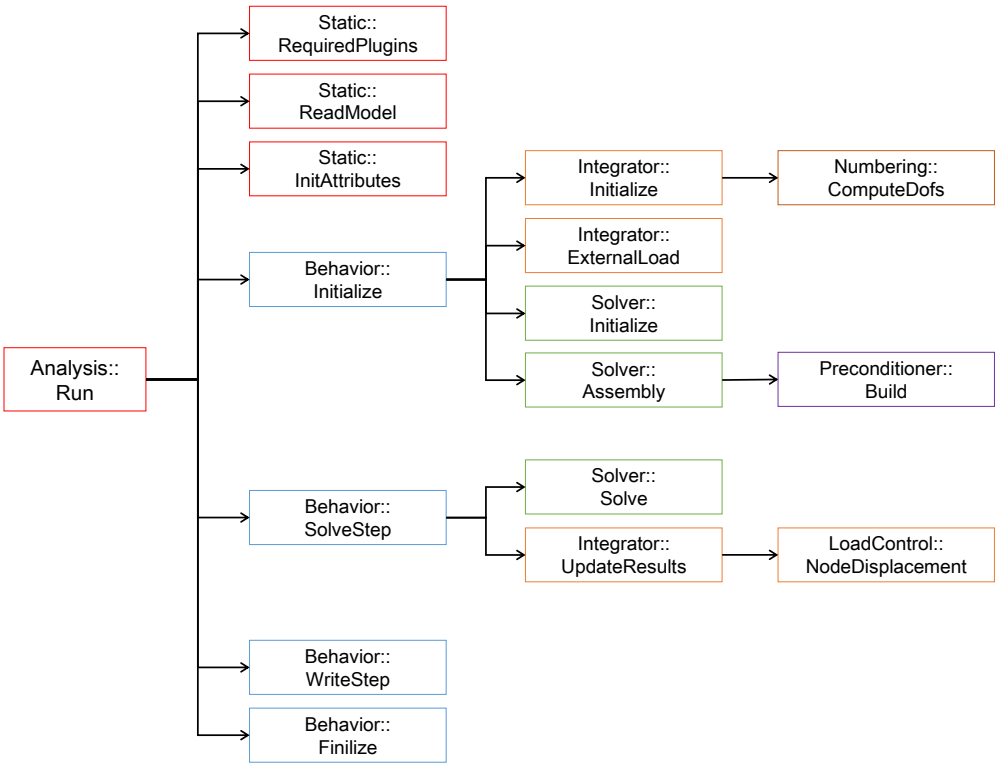



Figure 3.10: Framework main stages, showing how the plugins interact with each other to run the simulation.

The Nonlinear plugin was also developed to simulate structural problems involving geometrical nonlinearities [38, 39]. The system of equations can be

solved by means of an incremental-iterative procedure, based on the following general equation:

$$\mathbf{K}_j^i \delta \mathbf{u}_{j+1}^i = \delta \lambda_{j+1}^i \mathbf{p} + \mathbf{r}_j^i \tag{3-1}$$

where superscripts and subscripts refer to step and iteration numbers, respectively, $\mathbf{K}_j^i$ is the tangent stiffness matrix, $\mathbf{p}$ is the reference load vector, $\mathbf{r}_j^i$ is the unbalance load vector, $\delta \mathbf{u}_{j+1}^i$ is the unknown incremental displacement vector, and $\delta \lambda_{j+1}^i$ is the unknown incremental load factor.

The Newton-Raphson behavior plugin was also created to be used together with the Nonlinear analysis plugin and the Load Control integrator plugin. The solution scheme, in the Newton-Raphson plugin, computes the external loads at the first iteration of each incremental step and keep it constant throughout the remaining iterations within the step, as illustrated in Figure 3.11. Hence, the load factor is

$$\delta \lambda_j^i = \begin{cases} \text{prescribed value} & \text{for} j = 1 \\ 0 & \text{for} j \geq 2. \end{cases} \tag{3-2}$$

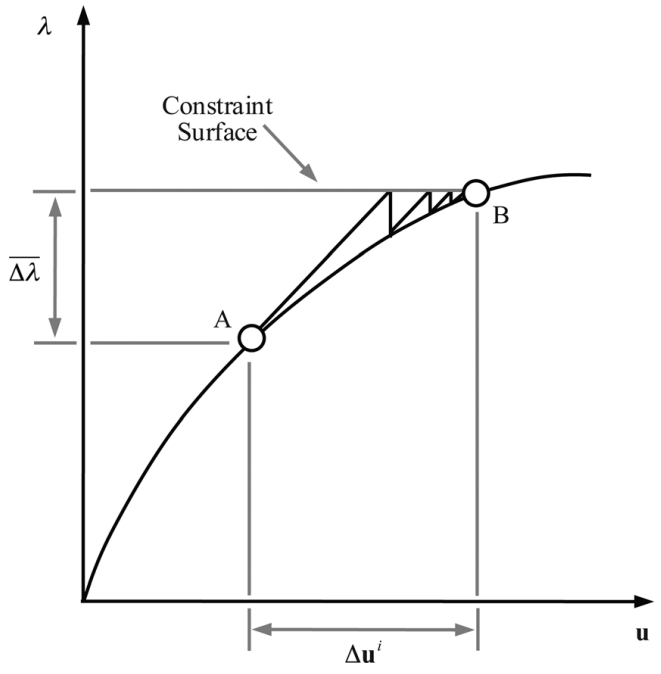



Figure 3.11: Load control method.

### 3.4.2 Reader and Writer

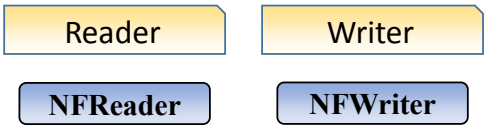



Figure 3.12: Reader and writer services with their plugin definitions.

The reader and writer services have a very simple interface. Basically, a reader plugin must define a function to set the name of the input file and a function to read the file to memory. We implemented the NFReader plugin to read files in the neutral file format, generated by the `PolyMesher` [40]. A new plugin can be developed and used in the framework to support different file formats. The most important thing in the reading process with plugins is the mechanism to query a reading section interface of a certain service. For example, to read the material info, the NFReader plugin first requests the instance related to the specific material, here called "ISOTROPIC", and then the reader plugin can query the reading section interface from this object, which knows exactly how to read the specific information about the isotropic material (Figure 3.13). The same mechanism works for other blocks of the neutral file, such as the types of elements and types of analyses. Each one can be read by its own plugin with no need to change the reader plugin.

```
Material* material;
if(!PluginManager::GetInstance("Material", name, &material))
  throw false;

INfSectionReader* reader;
if(!material->QueryInterface("INfSectionReader", &reader))
  throw false;

if(!reader->ReadSection(m_fp, name))
  throw false;
```

Figure 3.13: Piece of code to query and call the read section interface from the Isotropic plugin during the model reading process.

The NFWriter plugin follows the same idea of the reader plugin. It was developed to write the model and the results of a simulation in the neutral file format. We use this format so we can visualize the results in a tool for post-processing finite element models, called Pos3D. Again, the flexibility of

the plugin approach allows the framework to be extended to write the results in a different file format, supported by a different post-processor. We plan, as a future work, to write the results also in the ParaView file format [2], since the visualization of extremely large-scale problems may require, in some cases, a tool capable to post-process the results distributed in a cluster.

### 3.4.3 Sparse Matrix, Linear System, and Preconditioner

The coefficient matrix associated to the linear system of equations that arises when a numerical method is employed for solving a structural problem is usually sparse, i.e. most of its elements are zero. It is almost mandatory the use of techniques to take advantage of this property to reduce memory requirements, even for problems with just a few thousands of elements. In a sparse matrix format, only the non-zero entries are stored. Depending on the number and distribution of the non-zero entries, different data structures can be used. The trade-off is between the complexity of accessing an individual element and the need of additional data structures to recover the original matrix unambiguously.

We implemented plugins for the sparse matrix service with two different formats: Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) [41]. These formats were chosen because of their support for efficient access and matrix operations. The CSR format stores a sparse $(m \times n)\mathbf{M}$ matrix by means of three arrays: RowPtr, ColInd, and Values. The Values array holds all the non-zero entries from left to right and from top to bottom order. The RowPtr array is of length $(m + 1)$ and each entry has the information where the corresponding line begins in the Values array. The last entry represents the total number of non-zeros present in the matrix and is used to show where the Values array ends. The ColInd array has the same length of the Values array and holds the column index in $\mathbf{M}$ of each non-zero value. For example:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 7 & 0 \\ 0 & 4 & 0 & 0 \\ 7 & 0 & 2 & 0 \\ 0 & 0 & 0 & 9 \end{pmatrix} \tag{3-3}$$

[2] http://www.paraview.org/

$$
\begin{aligned}
RowPtr &= [\ 0 \quad 2 \quad 3 \quad 5 \quad 6\ ] \\
ColInd &= [\ 0 \quad 2 \quad 1 \quad 0 \quad 2 \quad 3\ ] \\
Values &= [\ 1 \quad 7 \quad 4 \quad 7 \quad 2 \quad 9\ ]
\end{aligned}
\tag{3-4}
$$

The CSC format is analog to the CSR format. The difference is that the RowPtr array becomes the ColPtr array, with the information of where the column begins in the Values array, and the ColInd array becomes the RowInd array, with the row index in **M** of each non-zero value. These two plugins are used with the linear system plugins, and they play a very important role in iterative solvers, where the matrix-vector multiplication is the main bottleneck.

Another sparse matrix storage scheme very popular in finite element codes for structural mechanics is the Skyline scheme (SKS). Its popularity comes from the fact that the skyline allocation is preserved during the process of solving the linear system of equations by a Gauss elimination type of method [42]. In order to improve the computational efficiency, the system of equations from finite elements must has a relatively small skyline where all fill-in entries fall inside it. We implemented a plugin with the skyline scheme together with the Crout linear solver, described later.

The linear system solver is very important in FEM and topology optimization, usually being the performance bottleneck of the simulation. Accordingly, choosing a solver type that is more suitable for a specific problem or improving the solver performance will result in a major impact on the system overall performance. Considering that we are solving elasticity problems, the corresponding coefficient matrix of the linear system is usually symmetric, positive-definite, and sparse. We implemented modules to handle direct and iterative solvers, assembling the global stiffness matrix and also an iterative element-by-element (assemble-free) solver, in which only the local stiffness matrix of each finite element is computed. When the global stiffness matrix needs to be computed, the process consists in first traversing all mesh elements and then computing their contributions in the triplets format. Then, the list of triplets, composed by the global degrees of freedom from the entire mesh, is used to assemble the global stiffness matrix in the sparse format. Figure 3.14 shows all the implemented services and plugins necessary to solve a linear system of equations.

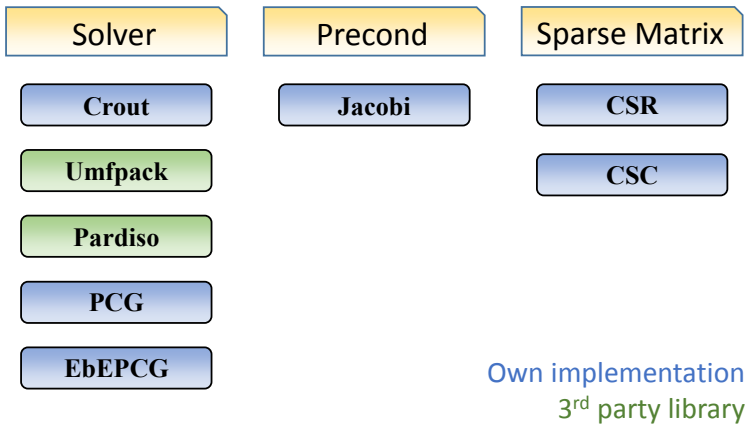



Figure 3.14: Services and plugins created in the framework to solve a linear system of equations.

### Direct Solvers

We implemented the Crout matrix decomposition as one of the direct solvers available in the framework. Crout is an algorithm that decomposes the matrix into a lower triangular matrix (L), an upper triangular matrix (U) and, although not always needed, a permutation matrix (P). The algorithm was designed for factorizing large sparse matrices, since the computational cost of sparse solvers is determined by the number of non-zero entries. The skyline sparse scheme (SKS) is used exclusively with this solver in the framework, since the solver algorithm takes advantage of the skyline allocation during the fill-in stage. However, a general sparse method, which stores only the non-zero entries rather than a skyline, becomes more efficient for very large problems (over hundred of thousands of equations), such as the CSR and CSC, described before.

The UMFPACK is a free package with a set of routines for solving sparse linear systems using the Unsymmetric MultiFrontal method [43]. We use its C interface to implement a serial direct solver in the framework, using a third party library. The solver package needs the global stiffness matrix stored in the CSC format. We convert the global list of triplets to the sparse format by means of the package routine $umfpack\_dl\_triplet\_to\_col$. The package routines $umfpack\_dl\_symbolic$, $umfpack\_dl\_numeric$, and $umfpack\_dl\_solve$ are called in this sequence to solve the linear system.

PARDISO is a shared-memory multiprocessing parallel direct solver developed by Intel Corporation [44]. The package can be used for solving large sparse symmetric and unsymmetric linear systems of equations. In this work, it has been used as an alternative solver with a powerful parallel direct approach.

The algorithm requires the global stiffness matrix stored in the CSR format. The assembling process is the same used with the UMFPACK solver, except for the fact that the package does not provides a function to convert from the triplet format to a sparse format. We implemented this conversion inside the sparse matrix plugin. The feature receives the list of triplets and mount the sparse matrix arrays. With the PARDISO solver we store only the symmetric part of the global stiffness matrix to reduce even more the required memory. The package routines can solve the linear systems in parallel using all the cores available in a very efficient way, as shown in the results.

### Iterative Solvers

We created a plugin to implement the preconditioned conjugate-gradient (PCG) solver [3]. The PCG is an iterative method for solving linear systems of equations with symmetric and positive-definite matrices. Due to the ill-conditioned nature of the linear system of equations in topology optimization problems, this method needs many iterations to converge, as pointed out by Wang [45]. Therefore, the use of good preconditioners is necessary when dealing with large-scale problems. The PCG plugin requires the connection of a preconditioner service and the connection of a sparse scheme service. We implemented the Jacobi preconditioner [3] primarily because of its simplicity and easy parallelization. The global stiffness matrix is assembled in the same way as in the PARDISO solver, using a CSR sparse scheme and storing only the symmetric part of the matrix. Algorithm 4 shows the basic steps of the PCG method [3]. We can notice that each iteration needs a matrix-vector product and some inner products in the order of the number of unknowns. The matrix-vector product becomes one of the bottlenecks of the overall simulation.

Another issue for solving large-scale problems is the computation of the global stiffness matrix ($\mathbf{K}$). Assembling $\mathbf{K}$ is the main bottleneck of the `PolyTop`, as presented by Talischi et al. [18] and shown in Table 3.1. In Duarte et.al [28], 2015, we presented a new parallel algorithm to perform 'assembly-free' PCG solver using polygonal meshes in GPU. We have chosen the 'assembly-free' PCG solver by Augarde et al. in 2006 [46] because in this method the full matrix $\mathbf{K}$ is not required at all. Instead, only the element stiffness matrices are used to solve the linear system of equations. We know that the storage for all the individual element stiffness matrices will exceed the memory taken by the global assembled matrix. However, there are issues by using a global stiffness matrix to solve large-scale problems. Using a direct solver with global stiffness matrix is impractical because of the extra memory

**Algorithm 4** Conjugate gradient with preconditioner.

$x_0$ is an initial guess, $r_0 = b - Ax_0$
**for** $i = 1, 2, ...$ **do**
  Solve $\mathbf{K}w_{i-1} = r_{i-1}$
  $\rho_{i-1} = r_{i-1}^H w_{i-1}$
  **if** $i = 1$ **then**
    $p_i = w_{i-1}$
  **else**
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p_i = w_{i-1} + \beta_{i-1} p_{i-1}$
  **end if**
  $q_i = \mathbf{A}p_i$
  $\alpha_i = \rho_{i-1}/p_i^H q_i$
  $x_i = x_{i-1} + \alpha_i p_i$
  $r_i = r_{i-1} - \alpha_i q_i$
  **if** $x_i$ accurate enough **then**
    quit
  **end if**
**end for**



needed for the factorization process (this is shown later in the results section of this work). Using an iterative solver with global stiffness matrix could be an alternative. An element-by-element solution eliminates both the assembly matrix bottleneck and the high memory consumption to store it. Combined with a parallel algorithm, this strategy is free to handle extremely high problem sizes.

In this work, we decided to improve the element-by-element solver from our previous work [28] to increase its flexibility and adequacy for an efficient parallel algorithm, to be used in a cluster of computers. It is important to mention that an ideal element-by-element solution for large problems would be to store no local stiffness matrix at all, and calculate the required member of the local matrix whenever it is needed. But for some types of elements, the process of calculating the local stiffness matrix leads to a high computational cost. Nevertheless, identical elements share identical local matrices, allowing the simulation of large-scale problems to be done with small memory footprint, as long as we can model the domain with multiple instances of a small set of elements.

Our previous element-by-element algorithm [28] suffered from race condition issues when two different parallel threads needed to write in the same memory position. In the parallel implementation of our EbEPCG, the matrix-vector product computed in two different threads may have to write results in the same dof, as shown in Figure 3.15. To address this problem, we used an approach based on graph coloring. The idea consists in computing the matrix-

| Mesh size | 2.7 K | 7.5 K | 30 K | 120 K |
|---|---|---|---|---|
| Assembling $\mathbf{K}$ | 37.8% | 41.1% | 37.2% | 28,5% |
| Solving $\mathbf{KU} = \mathbf{F}$ | 23.0% | 28.6% | 32.1% | 29.8% |
| Mapping $\mathbf{z}, \mathbf{E}$ and $\mathbf{V}$ | 6.1% | 6.8% | 6.9% | 20.1% |
| Update Design Vars. | 1.1% | 2.1% | 2.8% | 1.4% |
| Plotting | 16.0% | 7.7% | 3.3% | 1.8% |
| Others | 15.7% | 12.8% | 17.7% | 18.4% |

Table 3.1: `PolyTop` code runtime profile for different number of elements. The assembling of the global stiffness matrix $\mathbf{K}$ is the most expensive part of the code [18].

vector product of a set of elements that do have no nodes in common. By considering enough groups (colors), the matrix-vector product is computed with no race condition. The greedy coloring algorithm [47] was used there, but this includes one more step during the preprocessing stage.

On the other hand, in a strategy where each thread is assigned to a finite element node, race conditions should never appear. Furthermore, this approach fits well with our choice to use a topological structure for mesh representation. TopS provides easy access to the neighboring elements of a node to compute their contributions. The most important advantage of this solution is that all nodes can be processed in a fully parallel way without any type of serialization.

Figure 3.16 illustrates a node visiting all its neighboring elements during the matrix-vector product. Since the transverse of the neighboring elements is just for consulting data and not for writing, all nodes can be processed in parallel with no risk of race condition. However, one disadvantage of this approach is the fact that a single element will be visited many times by different nodes. In our opinion, since the number of neighbors is always limited and small (4 for Q4 elements, 8 for hexagonal elements, and 6 for hexagonal prism elements) and this cost will be shared among the many cores present in a multi-core machine or in a cluster, the slightly lower performance here is negligible and compensated by the elimination of the coloring step.

Algorithm 5 presents a pseudo-code for the matrix-vector product used in the EbEPCG solver. A parallel loop through the nodes uses the TopS functions to retrieve the node dofs and neighbors, computing each dof in sequence, visiting each neighbor element to get its own node dofs and local stiffness matrix. The corresponding element dof is used to multiply the related matrix entry by the input $x$ vector and add the result in the node dof position

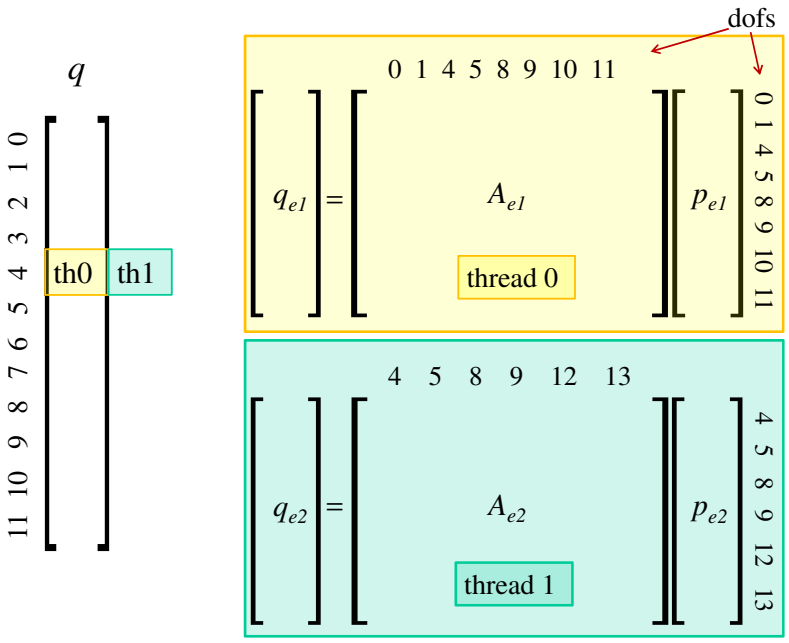


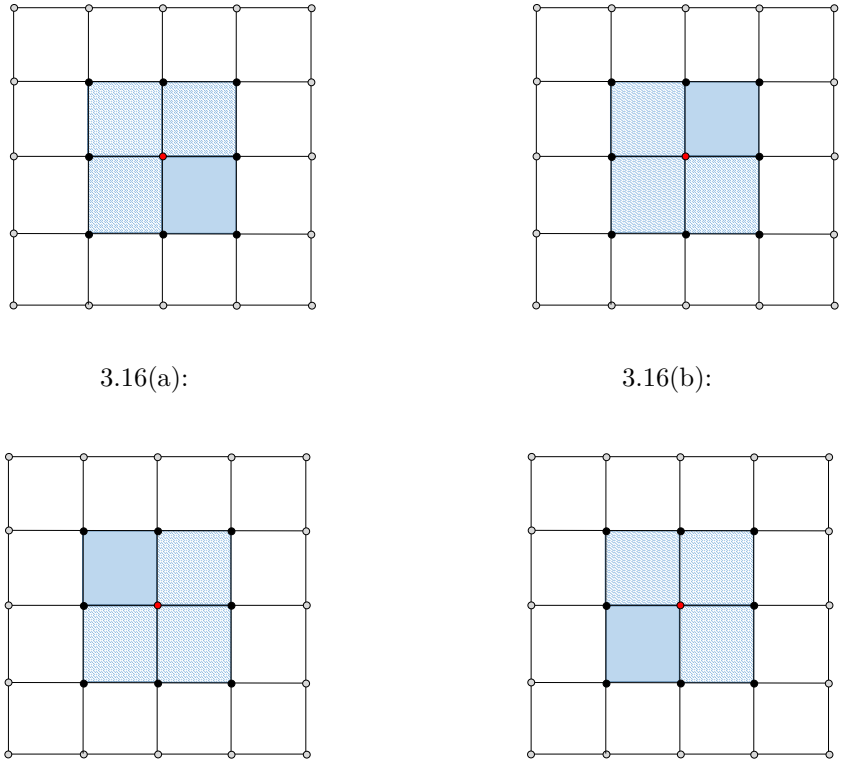

Figure 3.15: Race condition on the matrix-vector product of two different elements computed in parallel by two different threads. The elements share common dofs and may have to write in the same memory position in array $q$ [28].

of the output $y$ vector. The inner loops are not computed in parallel but they are vectorized with compiler pragma directives to extract the maximum performance of the machine.

### 3.4.4 Topology Optimization and Element Types

Topology optimization analysis is an example of a practical engineering problem with a high computational cost. Here, we focus on compliance minimization problems using the SIMP method, where the overall optimization process requires a finite element analysis inside each iteration. The linear system must be updated with the redistribution of element densities. Considering that a single finite element analysis is already computationally expensive, the process of updating the global stiffness matrix and solving a linear system during many optimization iterations is even more challenging when applied to large-scale problems.

The framework can be easily extended to include the Simp plugin modeled as a new type of analysis and replacing the Static plugin of the previous example. All other plugins presented previously to perform the FEM can be used with no changes, as shown in Figure 3.17. The issue now consists of how to update the global stiffness matrix without rebuilding it from scratch, since this is a very expensive step. Again, the flexible architecture of the



3.16(a): 3.16(b):

3.16(c): 3.16(d):

Figure 3.16: A node must visit all its neighboring elements during the matrix-vector product.

framework allows us to specialize the sparse matrix plugins, with auxiliary arrays and a new algorithm, in order to update the global stiffness matrix entries directly into the sparse format with each new element density without rebuilding it.

We implemented and used plugins with the classical finite element types Q4 and Brick8. However, it has been shown that the use of those classical elements in topology optimization suffer from numerical instabilities such as checkerboard patterns [13–15]. Indeed, one can use regularization schemes such as filtering to suppress the numerical instabilities, but these measures often involve heuristic parameters that can augment the optimization problem and can lead to significant weight increases [16, 17]. Recently, polygonal discretization has been proposed to achieve stable topology optimization formulations, using low order elements (degrees of freedom sampled at the nodes and constant design variable within the element), as reported in references [16,18,19]. Therefore, we also developed a plugin for the hexagonal prism element, also known as honeycomb (Figure 3.18), to be used specifically for topology optimization simulations.

**Algorithm 5** Matrix-vector product used in the EbEPCG solver.

```
{parallel loop}
for node = 1, 2, ... do
  TopsGet node_dofs
  TopsGet node_neighbors
  for dofi = 1, 2, ..., node_dofs do
    for node_neighbors = 1, 2, ... do
      TopsGet elem_dofs
      TopsGet elem_stiff_matrix
      for dofj = 1, 2, ..., elem_dofs do
        y[dofi] += elem_stiff_matrix[j * n + i] * x[dofj]
      end for
    end for
  end for
end for
```

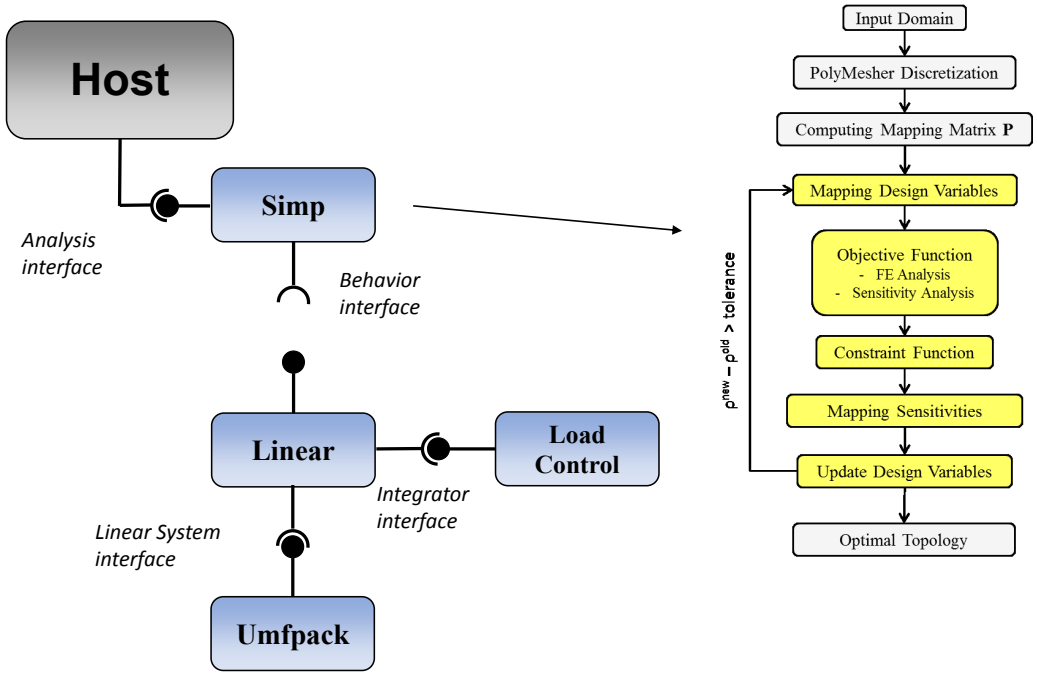



Figure 3.17: The topology optimization Simp plugin is modeled as a new type of analysis. All plugins previously used for the FEM example can be connected with no change.

The element services must provide information about the geometry of the element, e.g. area and volume, its shape functions, material behavior, number of nodes, number of active dofs per node, integration points, stress components, internal forces and the most important, the algorithm to be used to compute its local stiffness matrix. *We believe that the use of plugins to define the behavior of the finite element type, instead of defining the finite element as an object itself in the model representation, is one of the most important advantages and contribution of this work.*

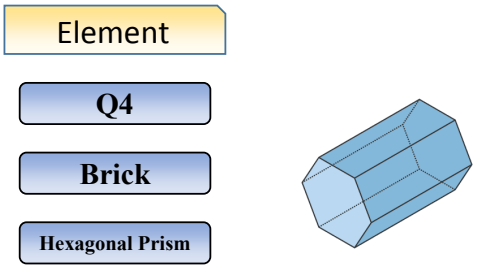



Figure 3.18: Element types implemented in the framework, including the hexagonal prism element.

In Figure 3.19 we show a summary of all plugins described up to now. The ability of the framework to deal with increasingly complex and diverse problems depends only on the creation and improvement of new services and plugins. Later in this work, we will present the plugins developed for distributed computing, with the creation of services for mesh decomposition, merging the results and generation of structured meshes distributed directly into the nodes of a cluster.



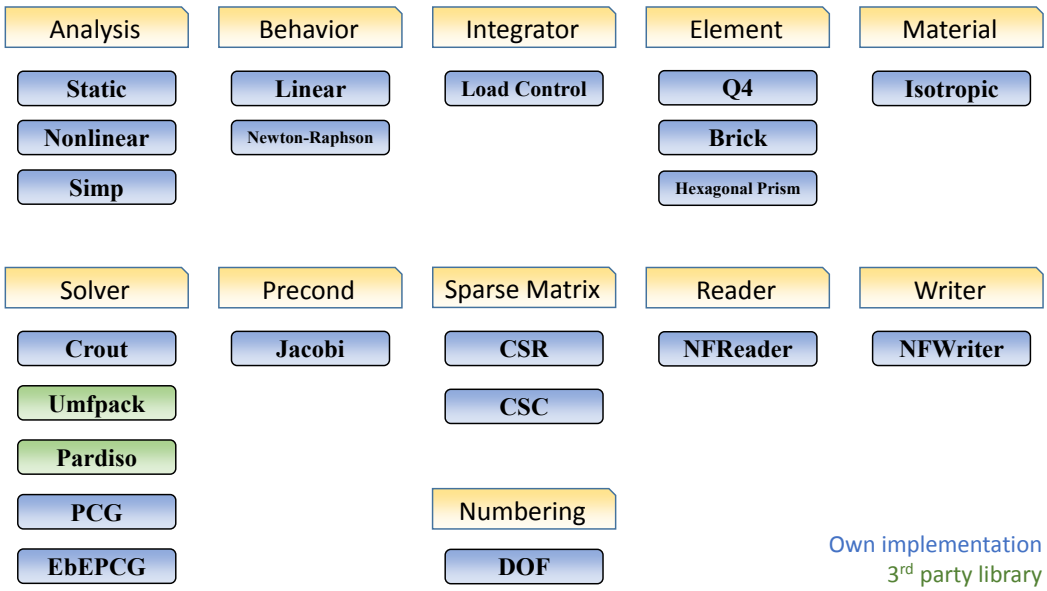



Figure 3.19: Summary of all services and plugins described up to now in the framework.

## 3.5 Results

We investigate the performance of the features implemented in the framework for solving a static linear elastic problem and a topology optimization problem. We validate our numerical results by comparing them with analytical solutions and related work from literature. The performance is analyzed regarding the use of memory, the computational performance and the scalability of the approach with the available computational resources. Because solving the linear system of equations is the most expensive computational stage of the simulation, it became our main focus in terms of performance maximization. All the examples presented in this section were simulated using the machine specification listed in Table 3.2, except when mentioned differently.

| Computing Platform | |
|---|---|
| O.S. | Ubuntu 14.04 LTS |
| Language | C++ |
| CPU | Intel Core i7-4930K |
| Clock | @3.40GHz |
| Cores | 12 |
| RAM | 64.0GBs |
| Library | OpenMP [48] |

Table 3.2: Computing platform used in the numerical simulations.



### 3.5.1 Linear Static Analysis

The first example corresponds to the Cook´s problem [49], using Q4 elements and the boundary conditions shown in Figure 3.20. We evaluated the behavior of the system with a mesh varying from 16 K to 1 M elements.

The results are presented in Figures 3.21 and 3.22. The color scale represents the displacement of the nodes in $Y$ direction. The analytical solution of the Cook´s problem consists of a $Y$ displacement of 24 mm at the C point (see Figure 3.20). In Figure 3.22, we show how the numerical solution approaches the analytical solution as we increase the number of elements of the mesh, as expected.

Figure 3.23 illustrates the time to solve the problem with different mesh sizes using the solvers implemented in the framework. We can notice that the

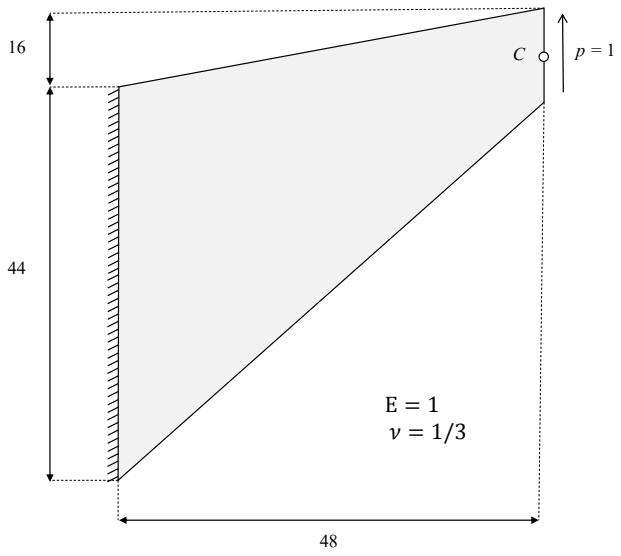



Figure 3.20: Geometry and boundary conditions of the Cook´s problem.



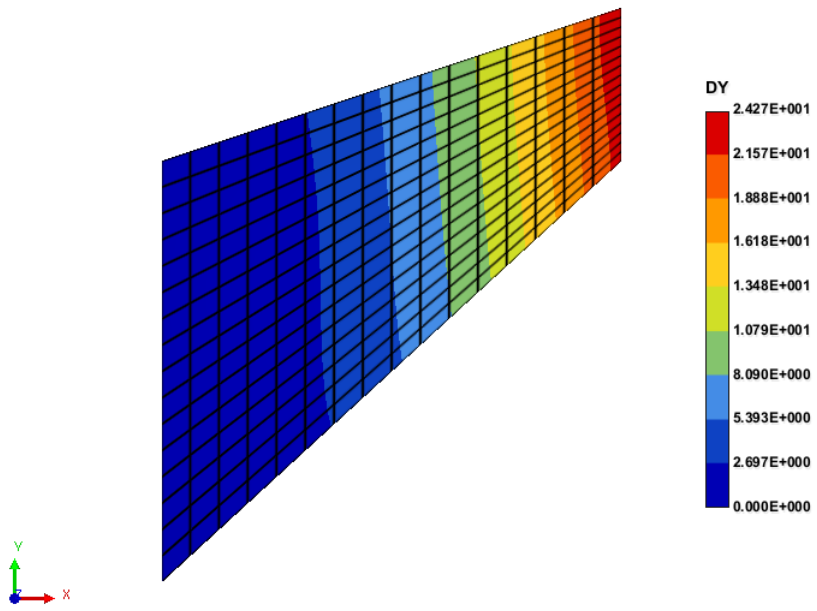



Figure 3.21: Results of the linear analysis of the Cook´s problem. The colors represent the displacement of the nodes in the $Y$ direction.

direct solvers PARDISO and UMFPACK perform better than the iterative solvers for this problem. The PARDISO solver is the fastest one with its parallel direct approach. It is almost 15 times faster than the UMFPACK solver, the second fastest one, for a mesh with 1 million elements.

The direct solvers are not so efficient when considering the memory usage

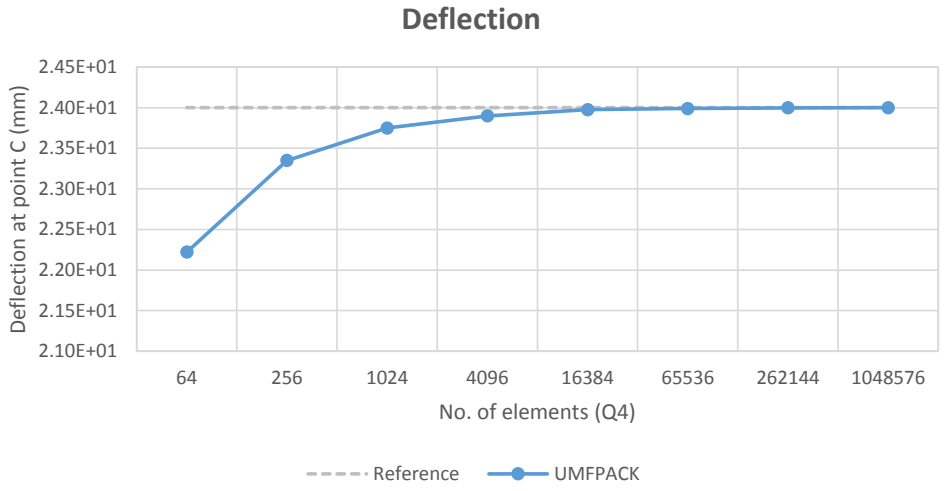



Figure 3.22: Results of the linear analysis of the Cook´s problem. The numerical solution approaches the analytical solution as the number of elements increases.

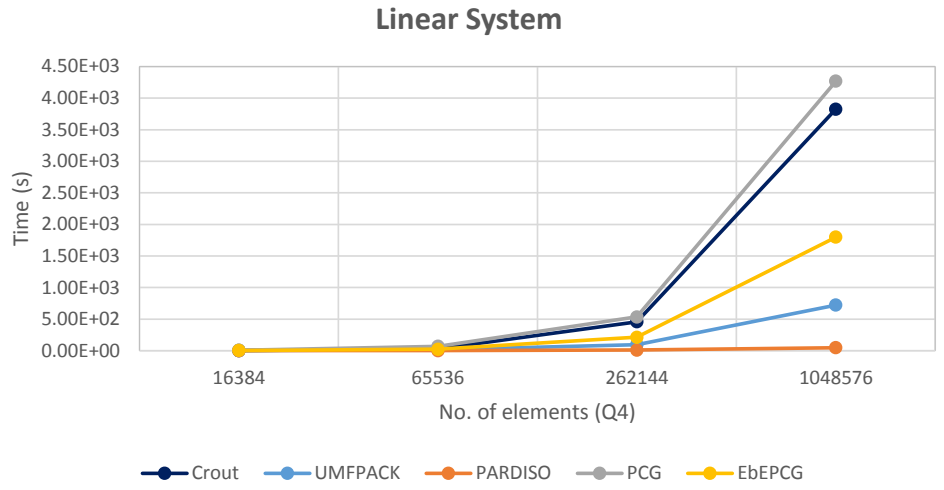



Figure 3.23: Time to solve the Cook's problem using the solvers implemented in the framework.

(Figure 3.24). The Crout solver is the most expensive, as expected, since the skyline storage scheme wastes memory allocation with zero entries, despite the fact that some of them are used during the decomposition stage. The UMFPACK and PARDISO solvers also present a high memory consumption, even using an efficient sparse matrix scheme, since a direct solver approach requires an extra memory for the decomposition step. On the other hand, the results confirm that the memory required by iterative solvers is much smaller and hence they are more suitable for large-size examples. For the curve associated to the element-by-element (EbEPCG) solver, Figure 3.24 shows that its growing rate is even smaller than a conventional PCG method, where a

global stiffness matrix is assembled and stored in memory.

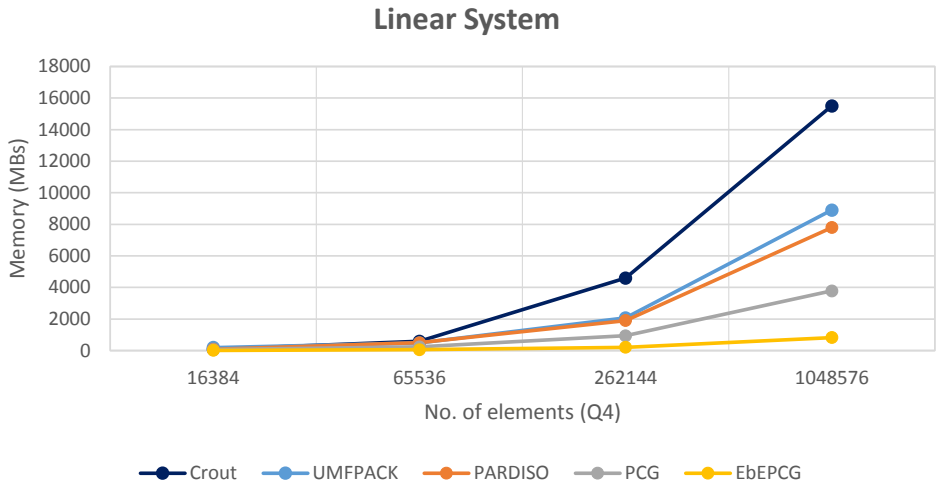
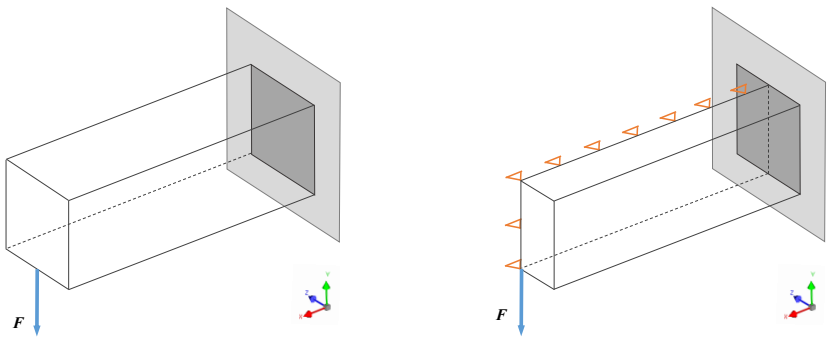



Figure 3.24: Memory used to solve the Cook's problem using the solvers implemented in the framework.

A 3D example was also tested using the Brick plugin, implemented in the framework to model the hexahedron element (Brick8). We solved the 3D Cantilever Beam example with the geometry and boundary conditions presented in Figure 3.25(a). Due to the symmetry of the problem, only half of the domain was considered, and the original problem was simulated applying the extra boundary conditions shown in Figure 3.25(b), where the displacements of all nodes on the middle face of the beam are set restrained in $Z$ direction.




3.25(a): 3.25(b):

Figure 3.25: 3D Cantilever Beam problem: (a) geometry and boundary conditions of the problem; (b) extra boundary conditions to restrain the displacement of the nodes in $Z$ direction.

The results were validated with Abaqus on a machine using the specifications listed in Table 3.3. Figure 3.26 shows an image of Abaqus software

post-processing the results. The colors represent the displacement of the nodes in $Y$ direction. The displacement of the red dot in Figure 3.26 is -5.741 mm in Abaqus and -5.743 mm in TopSim using the EbEPCG solver. Abaqus also uses a Krylov method with a preconditioner.

| Computing Platform Abaqus | |
|---|---|
| O.S. | Windows Server 64-bit |
| Language | FORTRAN |
| CPU | Intel Xeon X5650 |
| Clock | @2.67GHz |
| Cores | 24 |
| RAM | 24.0GBs |

Table 3.3: Computing platform used to compare the numerical results with Abaqus.

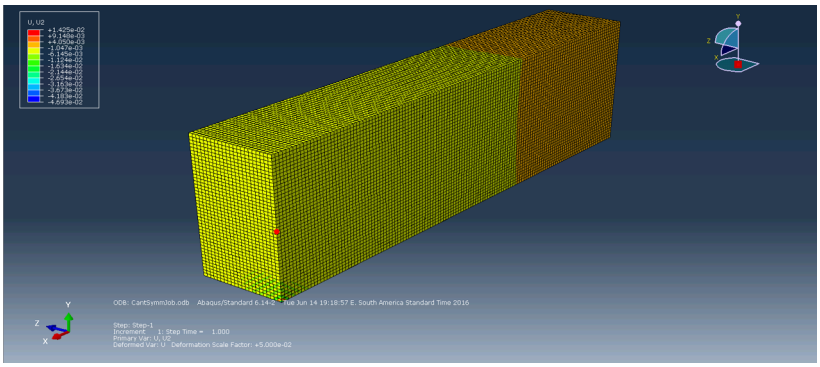


Figure 3.26: Post-processor of Abaqus with the results of the 3D Cantilever Beam problem. The colors show the displacement of the nodes in $Y$ direction.

Abaqus was also used in this work for comparison purposes, since it is a very known tool, used by many groups in academy and in industry for numerical analyses. Figure 3.27 shows the comparison between the iterative solver of Abaqus and the PCG solver implemented in TopSim, considering that both are iterative solvers assembling the global stiffness matrix in a sequential way (using just 1 core of the machine). Our main goal with this experiment is to check how efficient is the TopSim framework compared with a commercial and well accepted software. The results obtained show that TopSim is faster than Abaqus when the problem size varies from 96 K elements to 768 K elements, while Abaqus performs better for a mesh with 1.5 M elements. The PCG solver implementation is not optimized in the framework, since the EbEPCG is the main focus of our work.

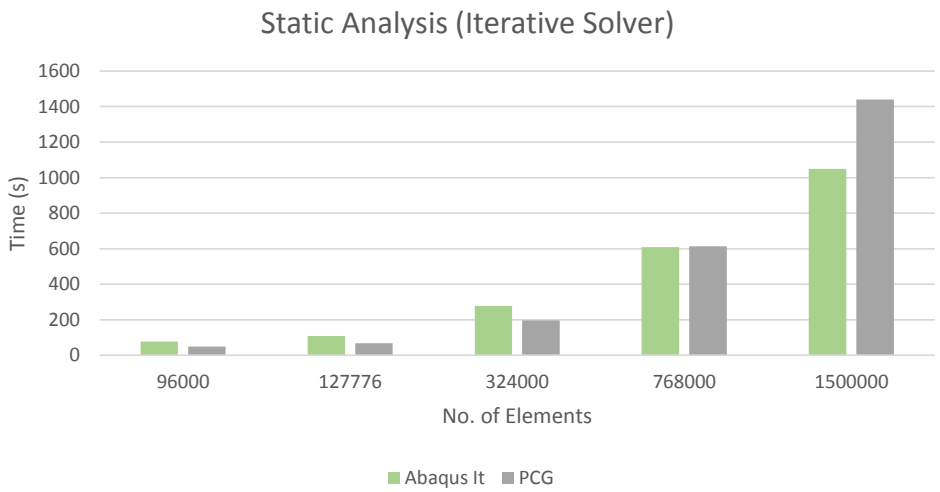



Figure 3.27: Time to solve the 3D Cantilever Beam problem using the iterative solver from Abaqus and the PCG from TopSim, both considering only 1 core of the machine.

We also compared the direct solver presented in Abaqus with the Pardiso solver used in TopSim as a third part library, still assembling the global stiffness matrix and using only 1 core of the machine (Figure 3.28). TopSim with the Pardiso library is again faster than Abaqus and is capable to solve the problem with 768 K elements, while Abaqus requires more memory than it is available on the machine. Nevertheless, both systems were unable to solve the example with 1.5 M elements, due to the high memory footprint required by their respective direct solvers.

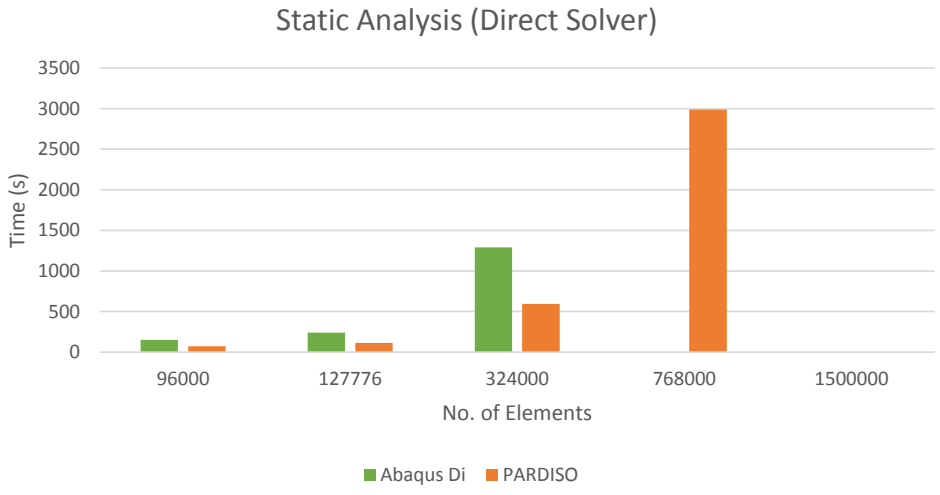



Figure 3.28: Time to solve the 3D Cantilever Beam problem using the direct solver of Abaqus and the PARDISO solver of TopSim, both using only 1 core of the machine. Abaqus was not capable to solve the examples with 768 K and 1.5 M elements due to memory limitations.

The results using only 1 core are important to evaluate if the plugin approach interferes with the performance of the TopSim system. In both examples, with iterative and direct solvers, the TopSim framework was able to handle large meshes as good as Abaqus, and even larger problems with the Pardiso solver. This result demonstrates that all the complexity of the plugin modeling does not prevents an efficient solution.

However, taking advantage of all available computing resources is essential for solving such costly numerical analysis. Therefore, we compare in Figures 3.29 and 3.30 the same solvers presented previously but now considering all the 24 cores available in the machine. The parallel solution from Abaqus is very efficient for both iterative and direct solvers. The iterative solver of Abaqus has a speedup of 3 times for 768 K elements and 1.5 time for 1.5 M elements comparing to the PCG of TopSim. In the case of the direct solvers, Abaqus is still slower than Pardiso, since this library, developed by Intel, is very efficient and takes advantage of all the features present in a parallel multi-core architecture.



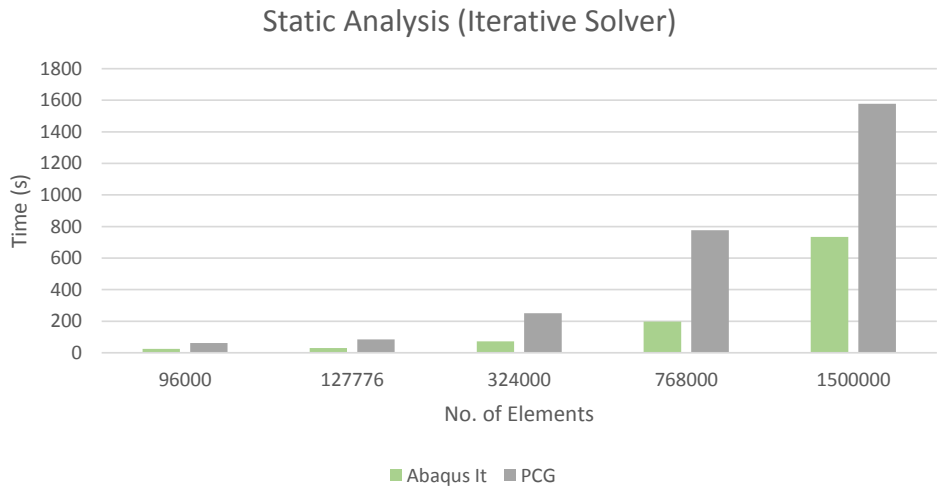



Figure 3.29: Time to solve the 3D Cantilever Beam problem using the iterative solver of Abaqus and the PCG of TopSim, both using all the 24 cores of the machine.

In Figure 3.31, we present the results of our implementation of an element-by-element PCG solver in TopSim, using all 24 cores of the machine. It is easy to notice that our parallel solution scales very well in the example with 1.5 M elements. The framework was 10% faster than Abaqus for this problem. Comparing to the direct solvers, the EbEPCG is also more efficient for a problem with 324 K elements, as we can see in Figure 3.32. The element-by-element solution is at least 3 times faster than the direct solvers and takes almost the same time as the iterative solver from Abaqus, but requiring a

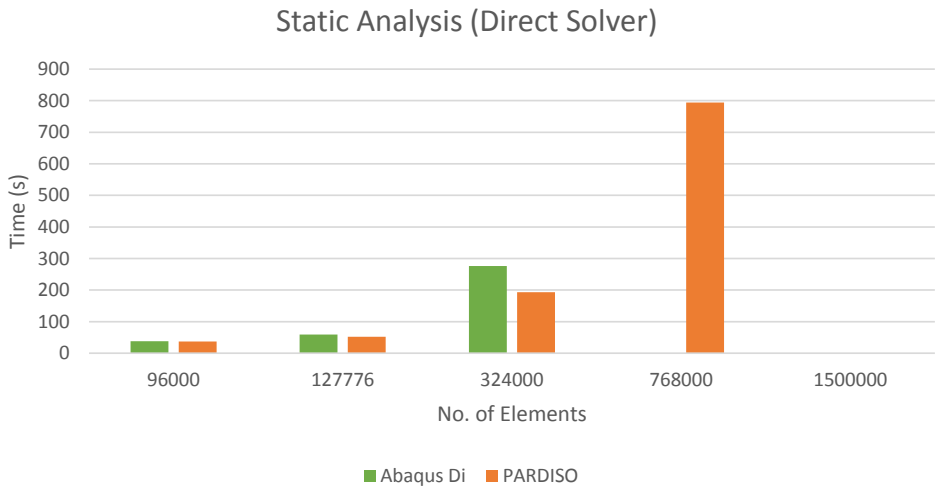



Figure 3.30: Time to solve the 3D Cantilever Beam problem using the direct solver of Abaqus and the PARDISO solver of TopSim, both using all 24 cores of the machine. Abaqus was not capable to solve the examples with 768 K and 1.5 M elements due to memory limitations.

negligible memory since the global stiffness matrix is never assembled and only one local stiffness matrix is used for all the elements in the mesh, considering that they are all identical.

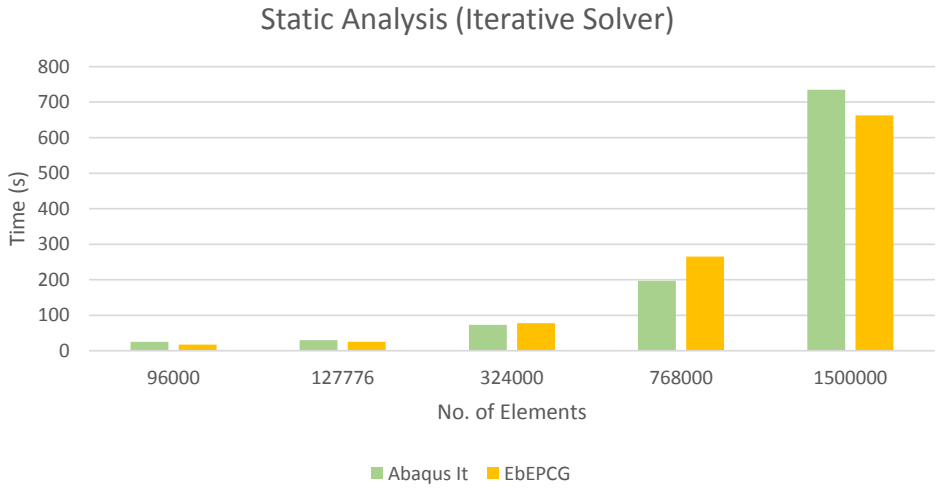



Figure 3.31: Time to solve the 3D Cantilever Beam problem using the iterative solver of Abaqus, and the EbEPCG solver of TopSim, both using all 24 cores of the machine.

### 3.5.2 Nonlinear Static Analysis

The Nonlinear analysis plugin and the Newton-Raphson behavior plugin were tested with the Lee´s problem [50], using a mesh composed by 48 K Q4

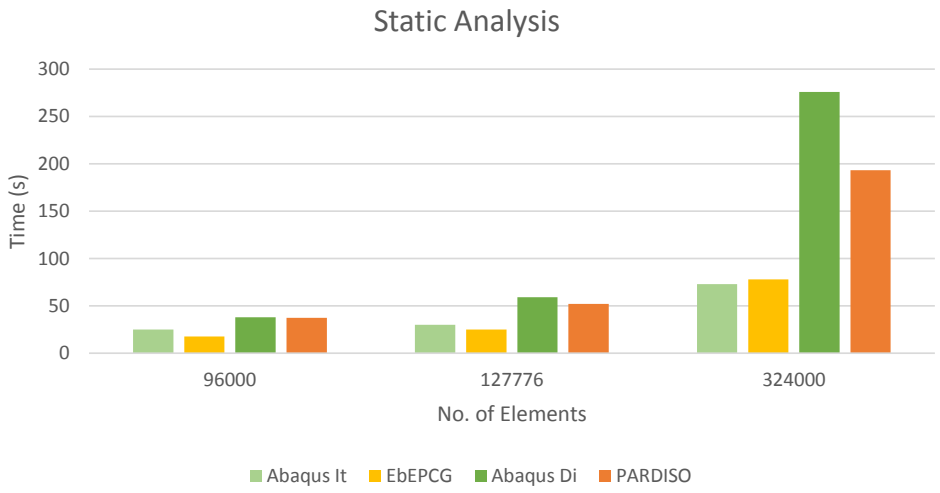



Figure 3.32: Time to solve the 3D Cantilever Beam problem using the iterative and direct solvers of Abaqus the of TopSim, using all 24 cores of the machine.

elements and the set boundary conditions as shown in Figure 3.33.



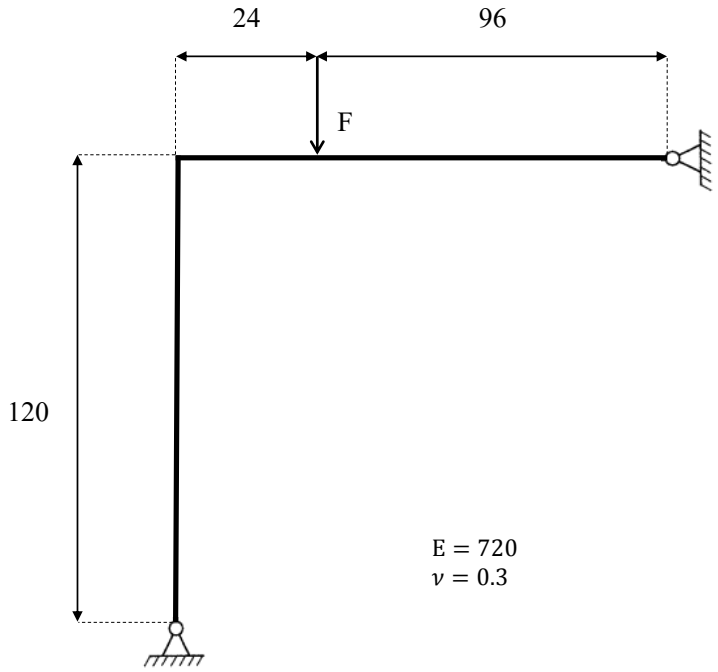
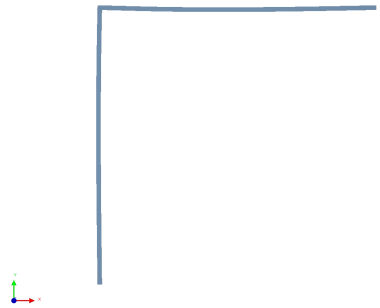


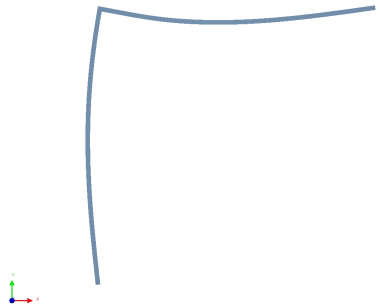

Figure 3.33: Geometry and boundary conditions of the Lee´s problem.

The incremental-iterative solution procedure was carried out with 20 steps, and the maximum number of iterations allowed in it step was set to 10. The convergence criterion was established in terms of the Euclidean norms of the unbalanced forces (residual) and the tolerance (TOL) was set as 0.001. Figure 3.34 shows the deformed configurations of the structure corresponding

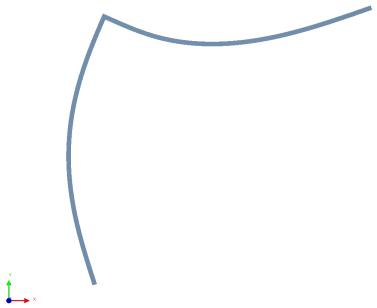
to load steps 1, 5, 10, and 17, respectively. The results obtained here are in very good agreement with the ones from the literature [38, 39, 50].



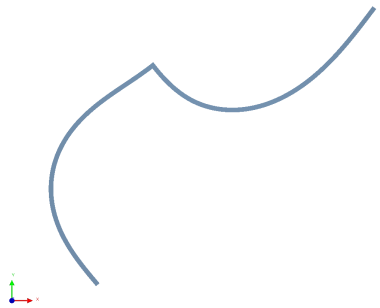
3.34(a):



3.34(b):



3.34(c):



3.34(d):

Figure 3.34: Results for the Lee´s problem: deformed configurations in steps (a) 1, (b) 5, (c) 10, and (d) 17.

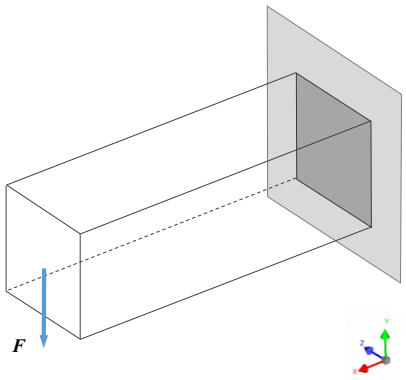
### 3.5.3 Topology Optimization Analysis

The TopSim framework was also used to simulate a topology optimization analysis. Table 3.4 presents the parameters used in the optimization process. We adopted the SIMP method with a penalty continuation from 1 to 3 with steps of 0.5, and a maximum of 50 optimization iterations in each step. The volume fraction constraint was set to 0.3 of the total volume domain.

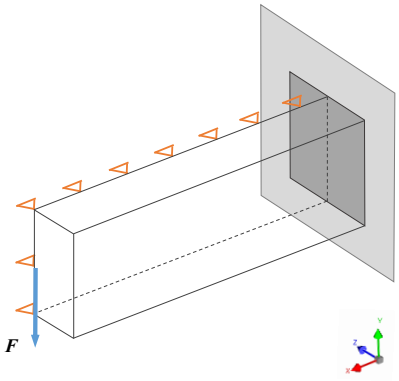
Figure 3.35(a) shows the geometry and boundary conditions of the 3D Cantilever Beam with the concentrated force applied at the center of the left face. Again, due to the symmetric nature of the problem, the original conditions are simulated with the extra boundary conditions in the symmetry plane as shown in Figure 3.35(b). The example was simulated on the same computing platform listed in Table 3.2. Figure 3.35(c) shows the results for the optimal topology using 1.5 million hexahedron elements.

| Optimization Parameters | |
|---|---|
| Method | SIMP |
| Penalty | $p \in [1, 3]; \Delta p = 0.5$ |
| Vol. Fraction | 0.3 |
| Max. Iterations | 50 |

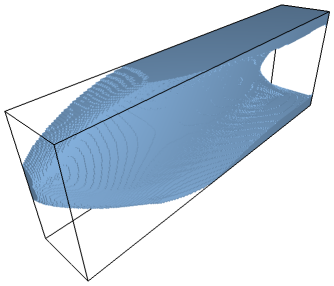
Table 3.4: Parameters used in the topology optimization simulation.




3.35(a):



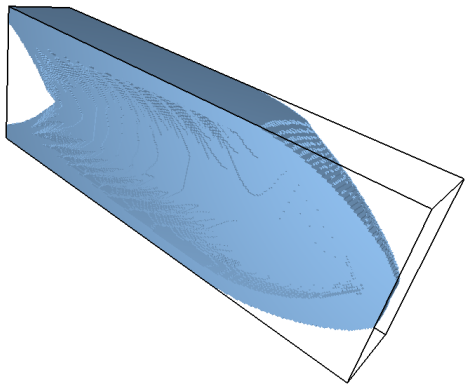

3.35(b):



3.35(c):

Figure 3.35: Topology optimization analysis of the 3D Cantilever Beam problem. (a) geometry and boundary conditions applied to the problem; (b) extra boundary conditions applied to the symmetry plane; (c) final optimal topology

Figure 3.36 illustrates different views of the optimal topology for the previous example. The density redistribution, concentrated in the center of the beam and close to the external applied force, is completely consistent with results obtained from the literature [18, 28, 51].



3.36(a):

3.36(b):

Figure 3.36: Different views of the optimal topology of previous example.

For the performance study, we start with small meshes, with 12 K elements, and only one penalization factor equals to 3, due to the complex nature of the topology optimization problem. Considering the fact that the linear system of equations becomes very ill-conditioned during the optimization process, the efficiency of the solver is crucial in order to handle large-scale problems. We can notice in Figure 3.37 that the UMFPACK solver does not perform very well compared to the other solvers, when the mesh is larger than 4000 elements. On the other hand, the PARDISO solver is the fastest one for all mesh sizes, followed by the two iterative solvers PCG and EbEPCG.

Regarding the memory used in the topology optimization problem (Figure 3.38), the PARDISO solver performs as bad as the UMFPACK solver. As expected, both direct solvers require too much extra memory. In the case of the PARDISO solver, its excellent performance to solve the system, using an efficient parallel algorithm, worth its high memory consumption. The EbEPCG is the most efficient in memory usage, since it takes the advantage of the fact that

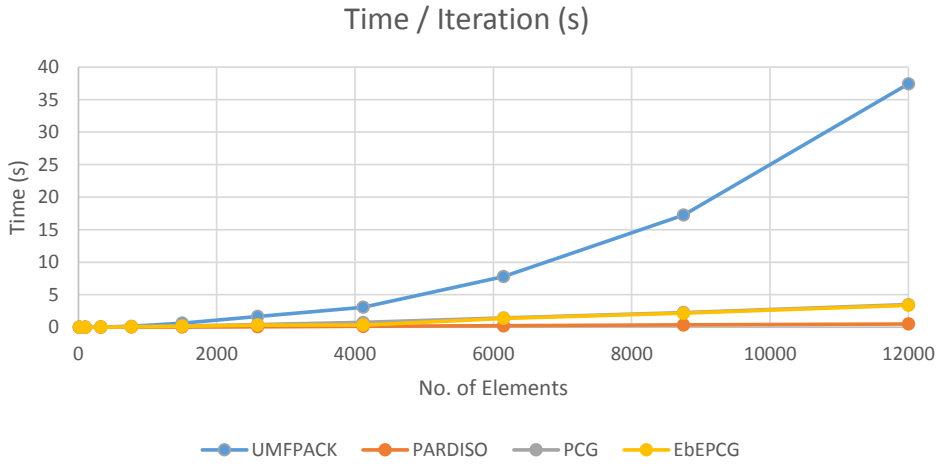



Figure 3.37: Average time to solve one iteration of the optimization process.

all elements use the same local stiffness matrix and a global stiffness matrix is never assembled.



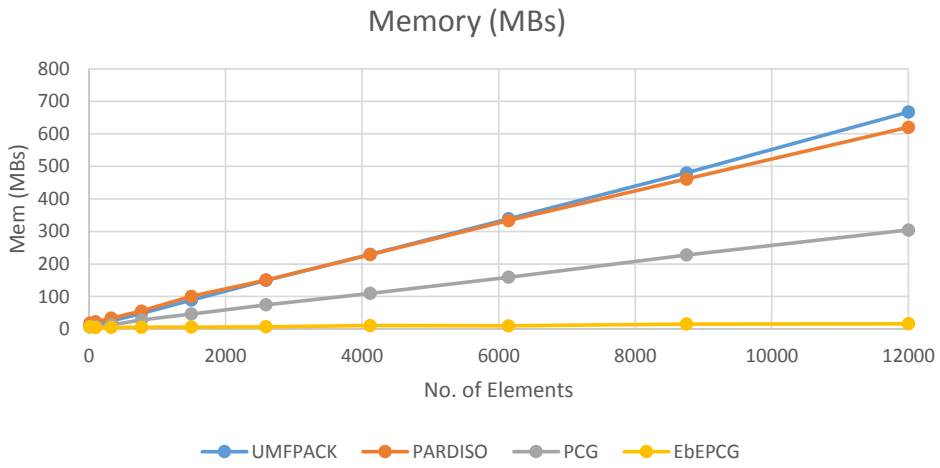
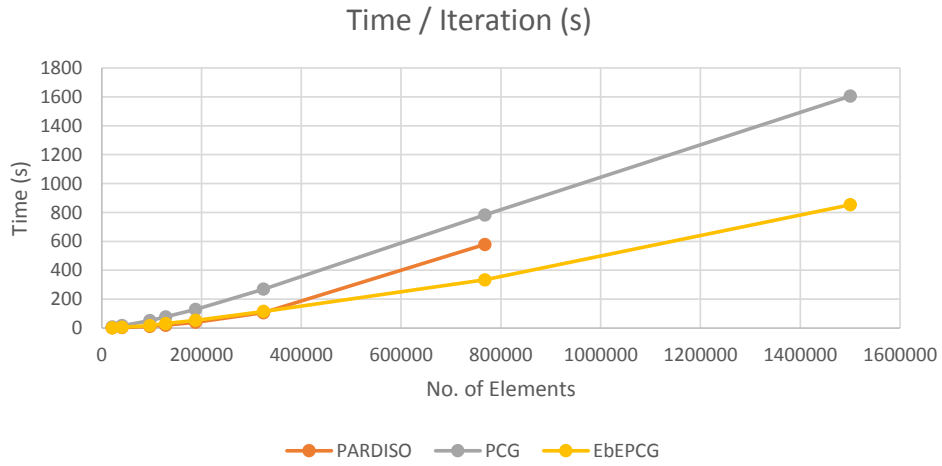


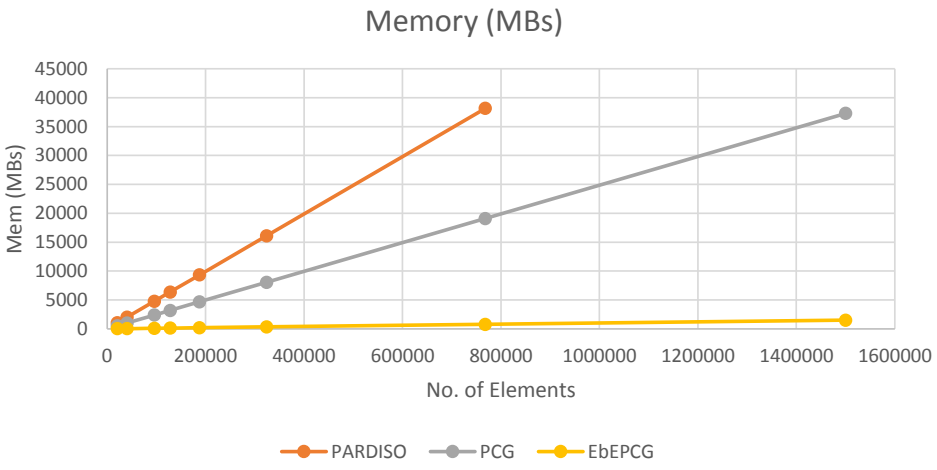

Figure 3.38: Peak memory required to solve the optimization problem.

We present results for larger meshes, with up to 1.5M elements, for the topology optimization problem presented in Figure 3.39. It can be noticed that the EbEPCG solver is faster than the PARDISO solver when the size of the mesh is larger than 300K elements. Moreover, after 700K elements the PARDISO solver is not able to solve the problem due to its high memory requirements (the required memory goes beyond the 64 GBytes of RAM, available on the machine). The memory used by each solver is presented in Figure 3.39(b). The rate of the PARDISO curve is extremely high compared to the iterative solvers. The memory required by the PCG solver is bigger than the EbEPCG because of the global stiffness matrix. The combination

of the two results presented for the EbEPCG solver shows the potential of this approach, considering that it is the fastest one with the lowest memory requirement, providing the capability to handle extremely large-scale problems even with a single machine.




3.39(a):




3.39(b):

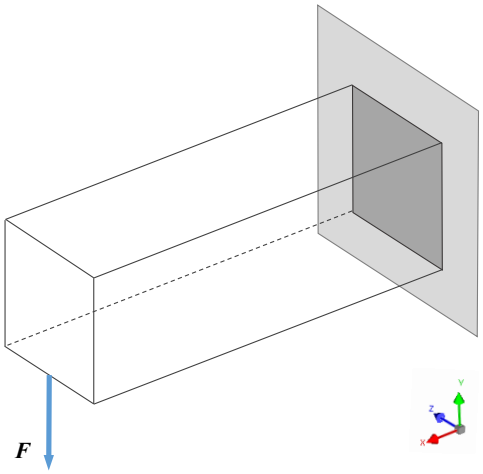
Figure 3.39: Performance analysis of the topology optimization problem for larger meshes. (a) average time to solve an iteration of the optimization process; (b) peak memory required to solve the optimization problem.

A second topology optimization example is illustrated in Figure 3.40(a), with the external applied force at the center bottom of the left face of the beam and the optimization parameters are listed in Tabel 3.5.

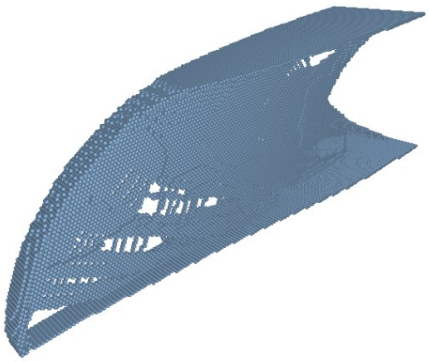
In this case, we simulated the entire beam with 1.5 million hexagonal prism elements and a volume fraction constraint of only 0.05 of the total volume domain. We can observe that the behavior of the solution in the middle

| Optimization Parameters | |
|---|---|
| Method | SIMP |
| Penalty | $p \in [1, 3]; \Delta p = 0.5$ |
| Vol. Fraction | 0.05 |
| Max. Iterations | 50 |

Table 3.5: Parameters used in the topology optimization simulation with hexagonal prism elements.



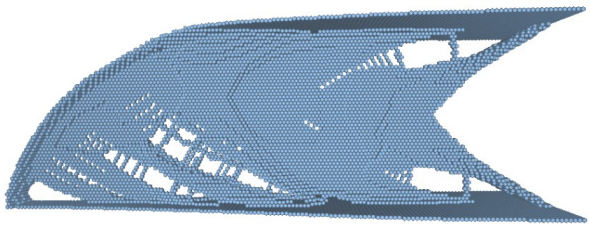

3.40(a):



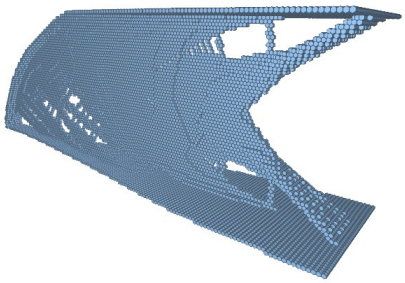
3.40(b):

Figure 3.40: Topology optimization problem simulated with the entire beam. (a) geometry and boundary conditions and applied force; (b) final optimal topology using hexagonal prism elements.

of the beam, with curved formations and no checkerboards (Figures 3.40(b) and 3.41), is consistent with 2D results using polygonal elements presented in related works [16, 18, 28].



3.41(a):



3.41(b):

Figure 3.41: Topology optimization results with different views of the final optimal topology.

### 3.5.4 Element-by-Element Approach

An efficient parallel solution for the element-by-element approach is essential for the overall performance of the TopSim framework, since the linear system solver is the main bottleneck of the simulation. An ideal result for the parallel algorithm would be the perfect use of all the cores available on the machine, with a good load balance between the cores and an efficient use of cache memory, minimizing the latency in memory access.

| Computing Platform | |
|---|---|
| O.S. | Windows 7 64-bit |
| Language | C++ |
| CPU | Intel i-7 2820QM |
| Clock | @2.20GHz |
| Cores | 8 |
| RAM | 8.0GBs |

Table 3.6: Computing platform used for the performance analysis of the EbEPCG.

We use a tool called Vtune from Intel [3] to investigate if our implementation of the EbEPCG solver is using the computational resources efficiently. We used the linear static example presented in Figure 3.25(a) with around 127K hexahedron elements and the computing platform listed in Table 3.6. Figure 3.42 presents a histogram of how long a set of cores runs simultaneously. We can see that the framework uses 7 and 8 cores in almost all the simulation, which is considered ideal by Vtune. The period using only 1 core usually happens during the reading and writing processes. The total average of the overall simulation is bigger than 6 cores simultaneously.

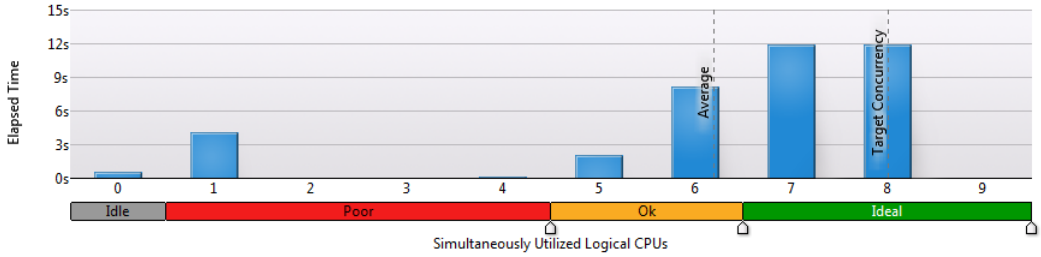
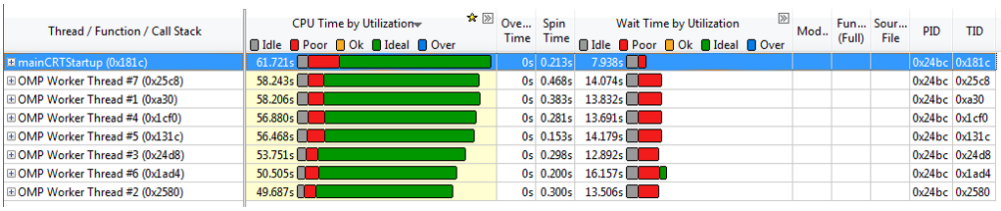



Figure 3.42: Histogram with the number of cores running simultaneously.

One of the requirements of an efficient parallel solution is a good load balance among the cores. This is important to avoid that one core finishes its work earlier and stays idle, waiting for the others to finish the computation. In Figure 3.43, it is shown the CPU time utilization of all the threads working during the simulation. Each thread has almost the same time in computation and so the idle time (bar in gray color) is very small.

Another important parameter in code efficiency is the count of cache memory misses. Every time a data in memory is consulted by the code and this data is no longer stored in cache memory, the operational system must

[3]https://software.intel.com/en-us/intel-vtune-amplifier-xe



| Thread / Function / Call Stack | CPU Time by Utilization: Idle | Poor | Ok | Ideal | Over | Ove... Time | Spin Time | Wait Time by Utilization: Idle | Poor | Ok | Ideal | Over | Mod.. | Fun... (Full) | Sour... File | PID | TID |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mainCRTStartup (0x181c) | 61.721s | | | | | 0s | 0.213s | 7.938s | | | | | | | | 0x24bc | 0x181c |
| OMP Worker Thread #7 (0x25c8) | 58.243s | | | | | 0s | 0.468s | 14.074s | | | | | | | | 0x24bc | 0x25c8 |
| OMP Worker Thread #1 (0xa30) | 58.206s | | | | | 0s | 0.383s | 13.832s | | | | | | | | 0x24bc | 0xa30 |
| OMP Worker Thread #4 (0x1cf0) | 56.880s | | | | | 0s | 0.281s | 13.691s | | | | | | | | 0x24bc | 0x1cf0 |
| OMP Worker Thread #5 (0x131c) | 56.468s | | | | | 0s | 0.153s | 14.179s | | | | | | | | 0x24bc | 0x131c |
| OMP Worker Thread #3 (0x24d8) | 53.751s | | | | | 0s | 0.298s | 12.892s | | | | | | | | 0x24bc | 0x24d8 |
| OMP Worker Thread #6 (0x1ad4) | 50.505s | | | | | 0s | 0.200s | 16.157s | | | | | | | | 0x24bc | 0x1ad4 |
| OMP Worker Thread #2 (0x2580) | 49.687s | | | | | 0s | 0.300s | 13.506s | | | | | | | | 0x24bc | 0x2580 |

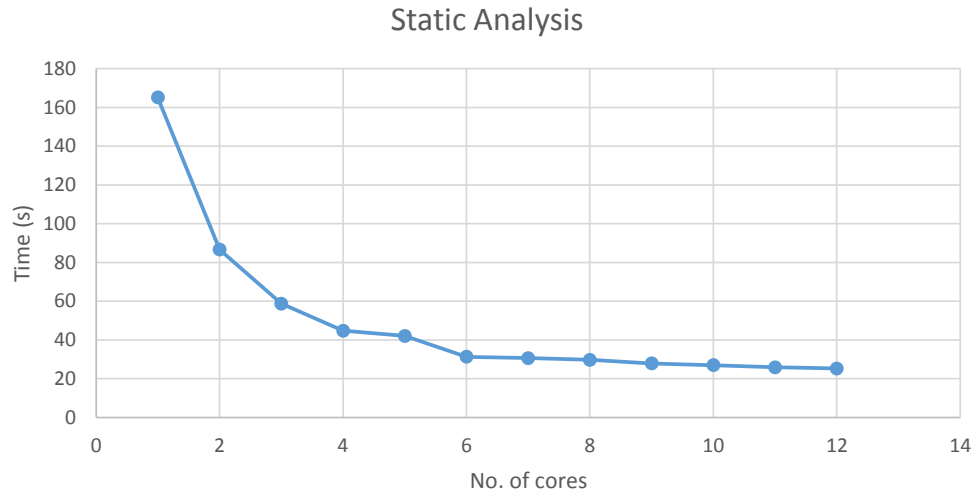
Figure 3.43: CPU time utilization by work threads, showing a good load balance during the simulation.

retrieve the data from the global memory (RAM). This process increases a lot the latency of the code since the global memory usually takes 100 clock cycles more than the cache memory. Using the Vtune tool, we show in Figure 3.44 that our code presents a rate of only 15% of cache misses. We believe that the element-by-element approach is responsible for this results because the small size of the local stiffness matrix, in this case only $24 \times 24$ entries, can be stored in cache memory during the computation, while with the global stiffness matrix the data might be retrieved from RAM more frequently.

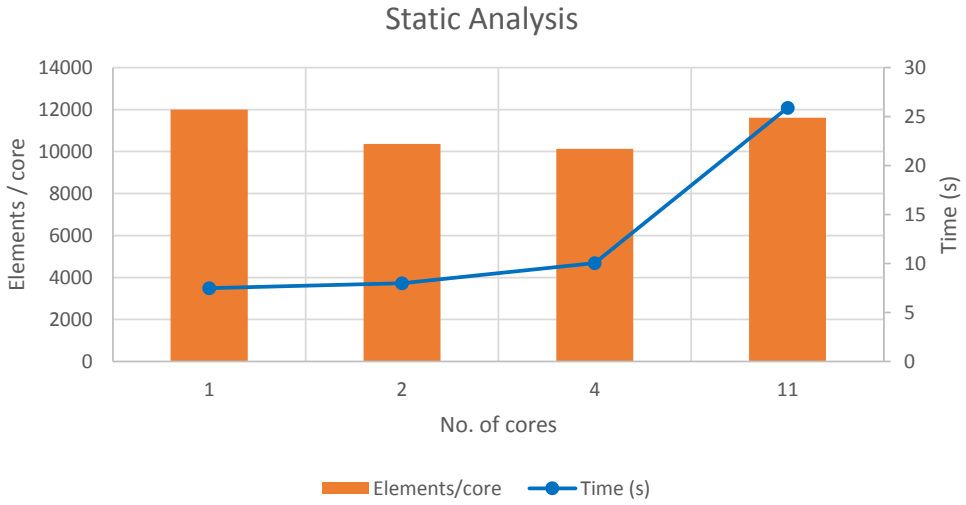
| Hardware Event Type | Hardware Event Count | Hardware Event Sample Count | Events Per Sample |
|---|---|---|---|
| INST_RETIRED.ANY | 1,293,785,125,931 | 470,711 | 2000000 |
| CPU_CLK_UNHALTED.THREAD | 1,185,221,239,644 | 278,369 | 2000000 |
| INST_RETIRED.ANY_P | 1,291,182,269,304 | 470,673 | 2000000 |
| MEM_LOAD_UOPS_RETIRED.L2_HIT | 1,650,156,802 | 12,114 | 100000 |
| MEM_LOAD_UOPS_RETIRED.L2_MISS | 281,089,813 | 3,629 | 50000 |
| Synchronization Context Switches | 11,909,671 | 6,907,055 | 0 |
| Preemption Context Switches | 2,636,719 | 18,775 | 0 |
| Wait Time | 445,644,290,992 | 6,907,055 | 0 |
| Inactive Time | 177,385,856,044 | 18,775 | 0 |
| Idle Time | 130,373,071,295 | 6,714,644 | 0 |
| Idle Wake-up | 14,238,538 | 6,714,644 | 0 |

Figure 3.44: Cache miss rate of the EbEPCG solver.

Figure 3.45 shows the performance of the framework, now using the computing platform in Table 3.2, when the size of the problem is constant and the number of cores is increased, and when we try to keep the work load constant, i.e. increasing the number of elements proportionally to the number of cores, running the simulation with the number of elements per core almost constant. The solution obtained shows a good performance when using up to 6 cores, with a speedup of 5.28× from 1 core to 6 cores (Figure 3.45(a)), and an almost constant time until 4 cores (Figure 3.45(b)). The reason for this behavior relies in the fact that every 2 cores of the machine share a floating point unit. Therefore, it is difficult to keep the performance rate with more than 6 cores, however, results show that the solution is able to take advantage of the extra computational resource to speedup the simulation.




3.45(a):




3.45(b):

Figure 3.45: Speedup results for the linear static analysis with hexahedron elements: (a) performance improvement with 127K elements and different working cores; (b) almost constant performance for a constant ratio between number of elements and cores.

# 4
# Distributed Solution on Clusters

One of the main objectives of this work is to push the mesh size limits further to billions of elements in a finite element analysis. For this purpose, an alternative is based on massive computation on supercomputers. The TopSim framework was extended with a hybrid implementation for distribute and parallel computing. The challenge here is to adapt our element-by-element solution to a domain decomposition scheme, as illustrated in Figure 4.1, using MPI for communication between subdomains and OpenMP for a parallel computing inside each subdomain, taking advantage of all available cores in each machine of the cluster.

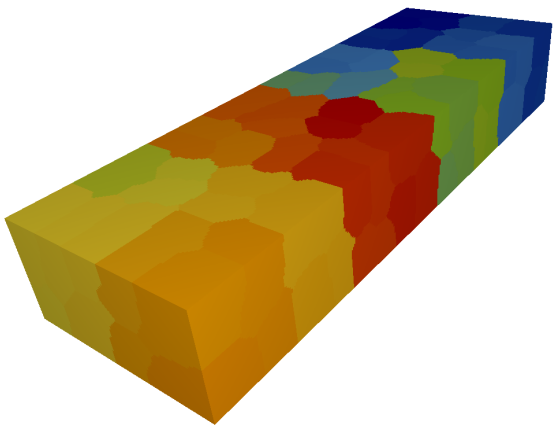


Figure 4.1: Domain decomposition for the distributed solution in clusters.

The domain decomposition was implemented as a new service and a new plugin, called Decompose and METIS respectively (Figure 4.2). The creation of a new service in the framework is important to provide the option to use different domain decomposition algorithms in the future. We decided to use the METIS library [52] because it is well known, extremely fast and produces high quality partitions.

The new domain decomposition feature requires the creation of two new plugins, responsible for reading and writing the files of the mesh partitions (Figure 4.2). In practice, with this new functionality, the framework is capable of receiving an input neutral file with the mesh domain, invoke the METIS

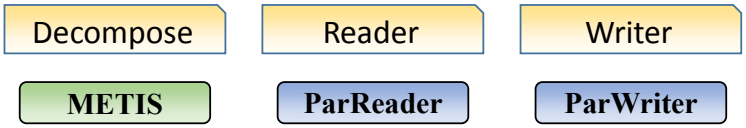



Figure 4.2: Plugins created to decompose, read, and write the mesh partitions.

plugin to decompose the mesh, and write the partition files. Inside the cluster, each machine reads the corresponding partition file and load its subdomain.

## 4.1 Distributed Approach

The model representation chosen here for the distributed computing in the framework was the ParTopS library [53], a natural extension of TopS, designed primarily to be a distributed topological data structure for finite element meshes in dynamic analysis of fracture and fragmentation.

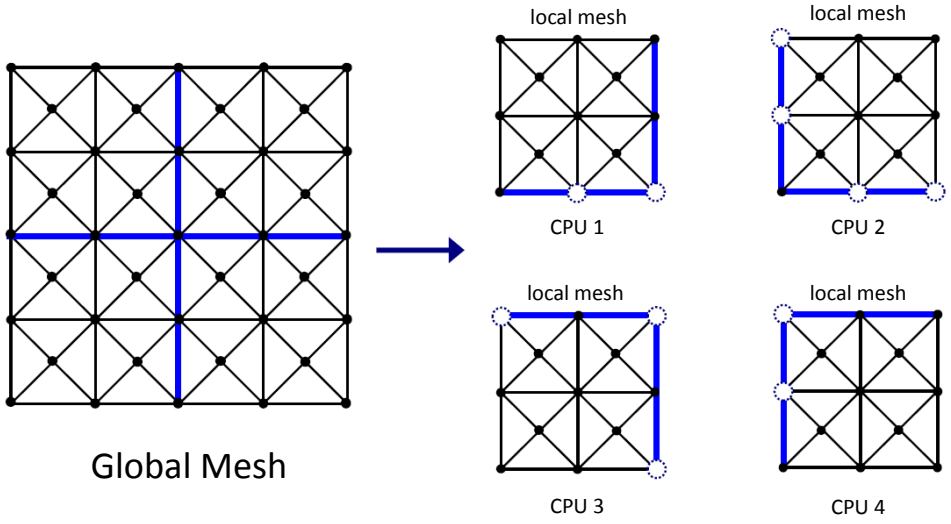



Figure 4.3: Original mesh partitioned into subdomains. Each node or element belongs to only one local mesh. [53]

The distributed mesh is composed by the union of the disjoint partitions from the original finite element mesh. Each element or node belongs to only one local mesh, as shown in Figure 4.3. However, partition boundary nodes may need to be shared by elements from different partitions. In order to keep the topological consistency of the mesh in each partition, nodes and elements are duplicated from their original partition to their neighboring partitions.

In ParTopS, the duplication of the entities is handle by the communication layer, shown in Figure 4.4. Each machine of the cluster loads its local mesh, communicates with its neighbors to create the communication layer, and synchronizes the attributes of the objects to update all subdomains and to ensure the consistency of the mesh.

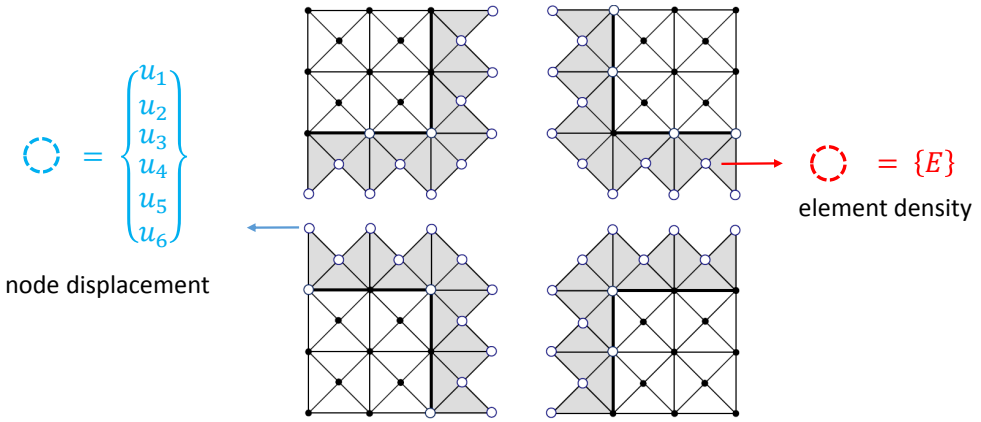



Figure 4.4: Communication layer created between the partitions. The attributes of the elements and nodes are synchronized to keep the mesh consistent. [53]

The duplication of the elements and nodes to create the communication layer does not represent a decreasing in efficiency or waste of memory, since the size of the layer is too small compared to the size of the local mesh in each partition. The examples simulated in the cluster have hundreds of thousands elements in each partition with only 1% of the elements duplicated in the communication layer. Moreover, the duplication avoids many extra communication messages during the simulation to query data from a neighboring partition.

The element-by-element algorithm described previously in Section 3.4.3 can be easily extended for a distributed environment using ParTopS. Considering that the algorithm computes the nodes of the mesh in parallel, and that each node must visit all its neighboring elements, the distributed algorithm computes only the nodes belonging to its local mesh. This way, we can guarantee that all the neighboring elements always exist, and the same node will never be computed twice, as shown in Figure 4.5. After the parallel computation of all nodes, the attributes of the elements and nodes in the communication layer are synchronized to update the mesh and make it consistent again.

Exploiting the framework flexibility to develop very specialized algorithms for a specific new feature, the new plugins presented in Figure 4.6 were created to handle the distributed solution on clusters. The ParEbEPCG and

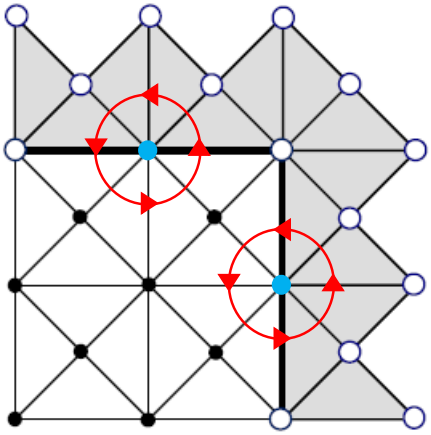


Figure 4.5: In the distributed element-by-element algorithm, only the nodes belonging to the local mesh are computed, in order to guarantee that all the neighboring elements always exist. [53]

ParJacobi plugins follow the element-by-element algorithm described before, where only the local elements and nodes of the mesh are computed with a synchronization after each iteration. The ParSimp plugin was created to handle the computation of the topology optimization analysis. The auxiliary arrays required to assign the corresponding physical quantities to the mesh were created as attributes of the elements and nodes. Each machine of the cluster performs the topology optimization analysis on its subdomain and than the attributes are synchronized to update the results for the entire mesh.

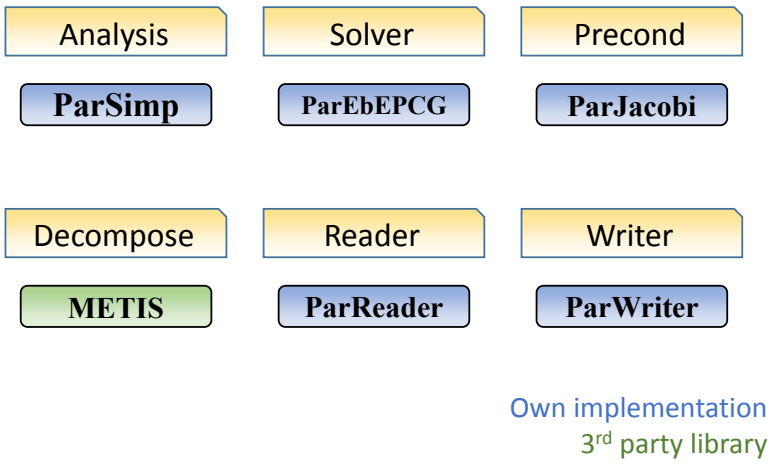



Figure 4.6: Plugins developed in the framework specifically for distributed computing in clusters.

| 3D Cantilever Beam | | | | |
|---|---|---|---|---|
| Elements | DOFs | Machines | Files size | Memory |
| 12 M | 39 M | 1024 | 3.4 GBs | 32 GBs |
| 40 M | 126 M | 4096 | 11.6 GBs | 80 GBs |

Table 4.1: Size of the files and memory required to decompose the mesh and run extremely large-scale examples on a cluster.

## 4.2 Distributed Mesh Generation

The process of generating the mesh and decomposing it can become very costly and inefficient. Table 4.1 shows two examples with the size of the mesh, number of partitions, size of all files created, and total memory required to load the input mesh file and decompose it using the METIS library.

As can be seen in Table 4.1, the decomposition of a mesh with 40 million elements already has a high computational resources cost, indicating that to decompose a mesh with a billion of elements would require an unfeasible amount of memory. The METIS library has a distributed version, where a cluster of computers can be used to decompose the mesh, but we would still have to handle extremely large files with the mesh partitions, which are used as input for simulations with the TopSim framework on clusters.

Another alternative to simulate such large example on clusters would be to generate the mesh already partitioned and distributed on the nodes of the supercomputer, considering that we are dealing with only structured meshes. Figure 4.7 shows an example of the parametric block included in the input neutral file to set the parameters to generate the distributed mesh. Each node of the cluster will generate your own local mesh and create the interface objects used in the communication layer. The boundary conditions are applied to the model also in a parametric way, from defining the position of the external force, combined with the faces or nodes that should be restrained or free in the mesh.

```
35 %MESH.NUM.ELEMENTS
36 900 450 2700
37
38 %MESH.NUM.PARTITIONS
39 18 9 18
40
41 %MESH.DOMAIN.SIZE
42 0.0 0.0 0.0 1.0 0.5 3.0
```

Figure 4.7: Parametric block included in the neutral file for the distributed mesh generation.

We implemented two plugins for the distributed mesh generation, as shown in Figure 4.8. The BrickBuilder generates a mesh in a box domain with hexahedron elements and the HexaprismBuilder generates a mesh also in a box domain with hexagonal prism elements (Figure 4.8).

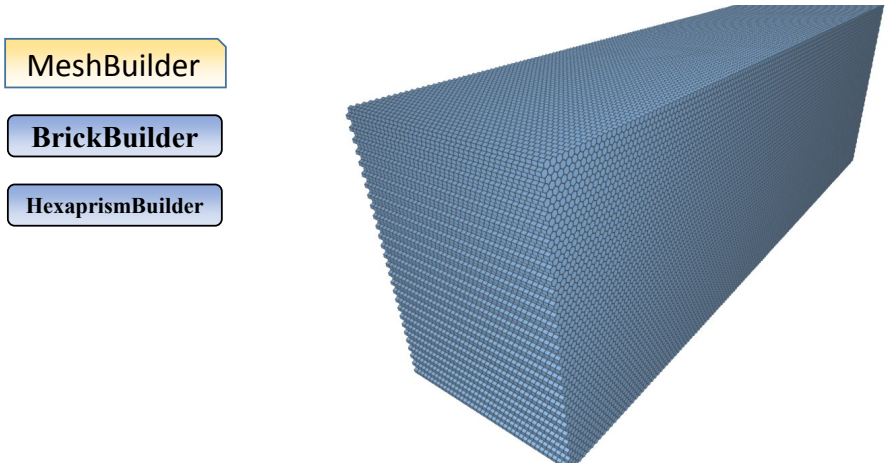



Figure 4.8: Plugins implemented for the distributed mesh generation with hexahedron and hexagonal prism elements.

By including this new feature, combined with the element-by-element approach, the framework becomes capable to provide the generation and simulation of meshes of any size, considering the computational resources available. From the moment that more machines are available in the cluster, larger meshes can be easily simulated with the TopSim framework.

## 4.3 Results

The tests of the distributed computing plugins implemented in the framework were performed on the Blue Waters Supercomputer. The cluster is one of the biggest in the world, composed by 22,640 machines, each one with 32 cores, 64 GBs of RAM and 102.4 GB/s of memory bandwidth. The machine nodes are interconnect in a 3D Torus architecture with a peak bandwidth of 9.6 GB/s. Table 4.2 presents a summary of the specifications of the supercomputer, while Figure 4.9 shows a picture of the cluster inside its dedicated building.

The primary goal of our distributed implementation is to make the framework capable of solving extremely large problems. We show results of the linear elastic static analysis, presented in Figure 3.25, with a much larger mesh on Blue Waters than the largest size of 1.5 million elements used here with only one machine.

| XE Cabinets | | XE Compute Node | |
|---|---|---|---|
| No. of Cabinets | 237 | Cores | 32 |
| Peak Performance | 7.1 PF | Mem / Core | 4 GBs |
| Compute Nodes | 22,640 | Total Mem | 64 GBs |
| Compute Cores | 362,240 | Peak Performance | 313.6 GFs |
| System Mem | 1.382 PBs | Mem Bandwidth | 102.4 GB/s |

Table 4.2: Summary with the specifications of the Blue Waters Supercomputer.

Figure 4.9: Picture of the Blue Waters Supercomputer inside its building.

Figure 4.10 shows the speedup achieved on Blue Waters when the number of machines is increased from 10 to 1200, each one with 32 cores, considering a linear static analysis of a problem with 50 million elements. The distributed solution is able to solve this example in a little more than 3 hours with 10 machines, in 22 minutes with 300 machines, and down to 13 minutes using 1200 machines.

In another example, we tested the framework in a linear static analysis with an extremely large mesh up to 500 million elements. In order to analyze the behavior of the distributed implementation, we tested different mesh sizes with different number of machines, keeping the number of elements per machine constant and equal to 500 K elements. The results are presented in Figure 4.11 and Table 4.3. The curve tends to a constant line, but the high number of machines increases the cost of communication among the partitions. However, the results show the ability of the framework to solve a numerical analysis with hundreds of millions elements.

The time required for communication between the machines may become

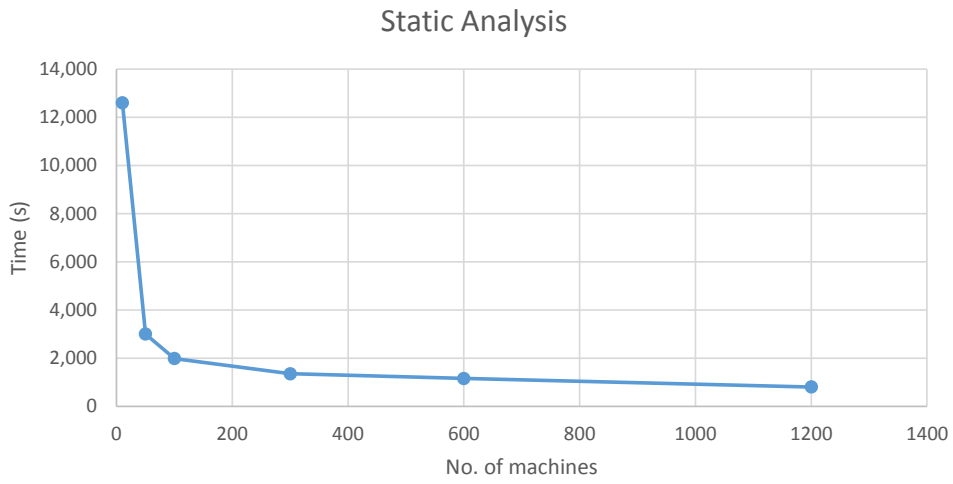



Figure 4.10: Speedup for solving a static analysis on Blue Waters with 50 M elements.

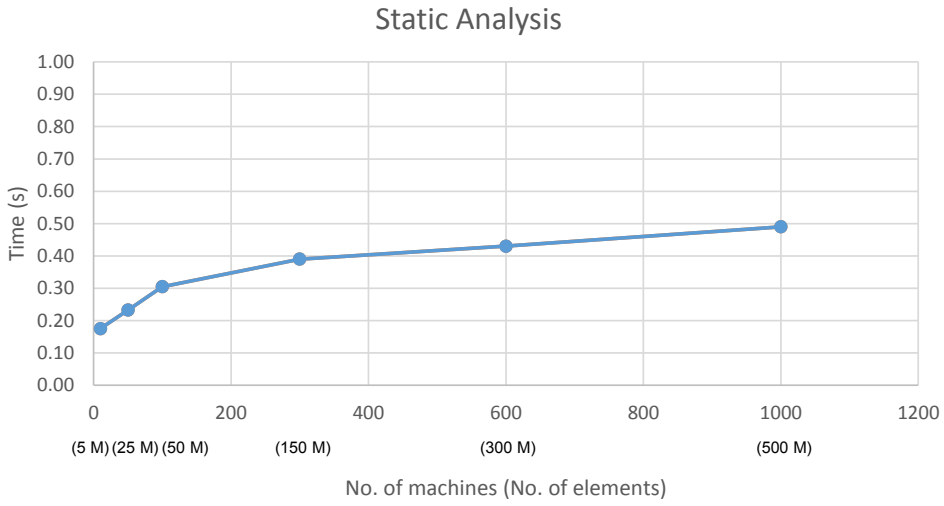



Figure 4.11: Computational performance for a constant ratio between the number of elements and machines used on Blue Waters.

a bottleneck, when the simulation is distributed among too many cluster nodes. A more detailed investigation may be required to improve the scalability of the solution. The Blue Waters system has many performance tools that can be used to map the bottlenecks and fine tune the code in order to run more efficiently on its architecture.

A topology optimization analysis was also tested on Blue Waters. We simulated the Cantilever Beam example described in Figure 4.12 with a distributed external load and the optimization parameters listed in Table 3.5, except for the volume fraction constraint, which was defined as 0.1.

We used a mesh composed by 12 million hexahedron elements with 48

| Static Analysis | | | |
|---|---|---|---|
| Elements | Machines | Solver Iters | Time/Iter (s) |
| 5 M | 10 | 2301 | 0.17 |
| 25 M | 50 | 4351 | 0.23 |
| 50 M | 100 | 6500 | 0.30 |
| 150 M | 300 | 9957 | 0.39 |
| 300 M | 600 | 14945 | 0.43 |
| 500 M | 1000 | 23255 | 0.49 |

Table 4.3: Static analysis with a constant number of elements per machine on the Blue Waters Supercomputer.

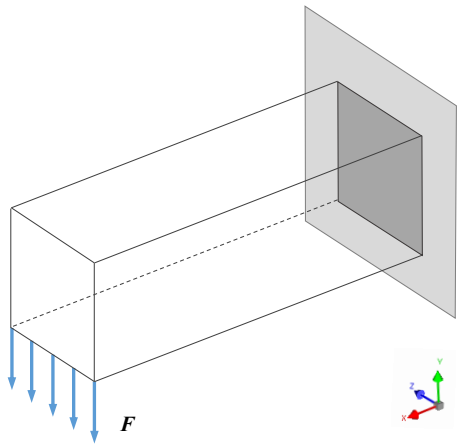


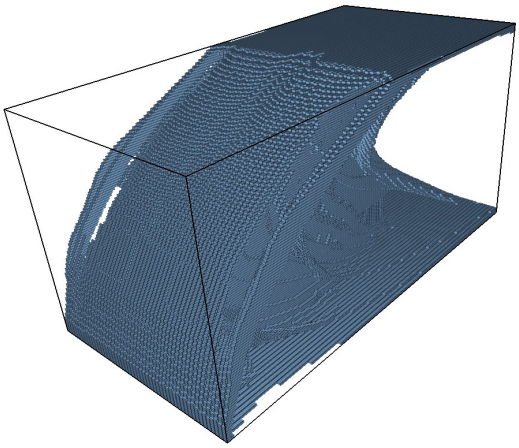

Figure 4.12: Geometry and boundary conditions and distributed external load of the topology optimization simulation on Blue Waters.

and 300 machines. The optimal topology is shown in Figure 4.13. Table 4.4 shows the total time to solve the topology optimization analysis, the time per topology optimization iteration, and the time per solver iteration.

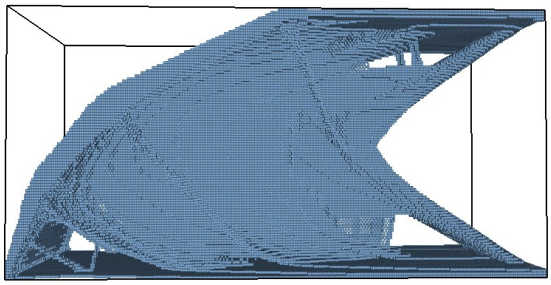
| 3D Cantilever Beam (12 M Elements) | | | |
|---|---|---|---|
| Machines | Time (h) | Time/Iter (s) | Time/Solver Iter (s) |
| 48 | 48.6 | 700 | 0.19 |
| 300 | 14.7 | 212 | 0.06 |

Table 4.4: Time to solve the example of Figure 4.12 on the Blue Waters Supercomputer.

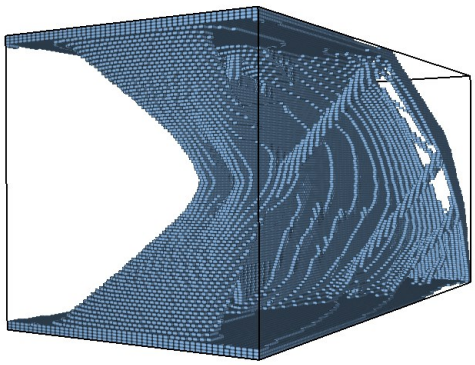
The results in Table 4.4 show how costly is a topology optimization anal-



4.13(a):



4.13(b):



4.13(c):

Figure 4.13: Final optimal topology of the 3D Cantilever Beam problem with a distributed external force, simulated with 12 million elements on 300 machines of the Blue Waters Supercomputer.

ysis. The time per solver iteration is of 0.19 seconds but the complete analysis requires 2 days, when using 48 cluster machines. The TopSim framework was able to drop the time down to 14.7 hours when the number of machines was increased to 300. In our previous work [28], a similar topology optimization example, with also 12 million elements, was solved in 6 days and 6 hours using only 1 GPU. The TopSim support for a distributed computing provides the framework the capacity to simulate computational costly analyses in a more feasible way, considering the availability of more computational resources.

Our element-by-element iterative solver provides the TopSim framework the ability to simulate an extremely large-scale example. The low memory consumption and the distributed parallel computing algorithm allow the framework to solve a linear system with billions of equations. We simulated a few iterations of the topology optimization example from Figure 4.12 with 324 million elements (975 million dofs) on 864 machines, and 1 billion elements (3 billion dofs) on 2916 machines. The results are shown in Table 4.5, where the average time per optimization iteration for the problem with almost 1 billion dofs was of 45 min and for the problem with 3 billion dofs was of 1.21 hour. The time per solver iteration was almost constant, with 0.14 and 0.16 seconds for the first and second examples, respectively.

| 3D Cantilever Beam | | | | | |
|---|---|---|---|---|---|
| Elements | DOFs | Machines | Time/Iter | Solver Iters | Time/Solver Iter |
| 324 M | 975 M | 864 | 45 min | 18700 | 0.14 |
| 1 B | 3 B | 2916 | 1.21 hr | 28000 | 0.16 |

Table 4.5: Extremely large-scale examples simulated on Blue Waters.

The results show the capacity of the framework to generate and handle a huge mesh with 3 billion dofs. Such simulation brings many challenges that are best overcome when specific approaches are applied to each of them. We believe that the TopSim framework presented efficient solutions from small to huge mesh sizes, used in different problems with different types of finite elements. We show in Figure 4.14 a summary with all the services and plugins developed in the TopSim framework.

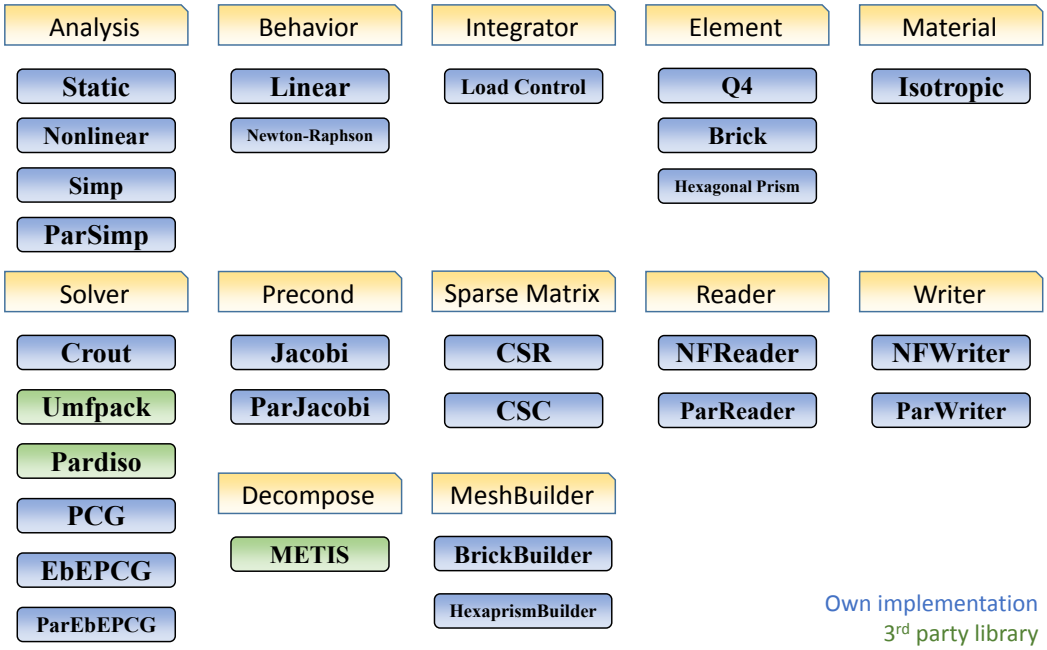



Figure 4.14: Summary of all services and plugins develop in the TopSim framework.

# 5
# Conclusion

This work presents a framework for large-scale numerical analysis based on plugins. The framework provides a fully flexible and easy to extend environment where specialized plugins are created to solve different problems, in an efficient manner. A hybrid distributed parallel code was implemented in the framework to simulate extremely large numerical analyses with a billion of finite elements on the Blue Waters Supercomputer.

We propose an element-by-element version of the PCG solver with low memory consumption using parallel computing. Our strategy is race condition free, since each thread is assigned to a finite element node. The global stiffness matrix is never assembled and only one local stiffness matrix is used for the elements in the mesh, taking advantage of the fact that they are all identical. We present the architecture of the framework and its functionality that allows the user to configure the plugins dynamically in order to achieve the best performance when simulating both small and large mesh sizes.

The flexibility of the framework demonstrated by the implementation of direct and iterative solvers for linear systems of equations, sparse matrix storage schemes and preconditioners, different finite element types (such as the Q4, Brick8 and hexagonal prism element), combined with plugins for linear elastic static and topology optimization analyses. We show that the plugin-based approach does not interfere with the performance of the TopSim framework, by comparing the results with Abaqus, a well known software for numerical analyses. The implementation proposed in TopSim, for both iterative and direct solvers, was able to handle large meshes as well as Abaqus, and even larger problems with the Pardiso and the EbEPCG solvers. Using all 24 cores of the machine, the EbEPCG solver in TopSim is 10% faster than Abaqus in the example with 1.5M elements.

The TopSim framework was extended for distributed computing on clusters. Our element-by-element solution was adapted to a domain decomposition scheme, using MPI for communication among subdomains and OpenMP for a parallel computing inside each subdomain. The domain decomposition was implemented inside the framework using the METIS library. However, to decompose a mesh with a billion of elements it would require an unfeasible

amount of memory. Therefore, we created plugins to generate the mesh already partitioned and distributed on the nodes of the cluster, considering that we are dealing with only structured meshes.

We used the Blue Waters Supercomputer to simulate a linear elastic static analysis with 500 million finite elements on 1000 machines. The solution required 23255 iterations with 0.49 seconds per iteration. The TopSim was also used to solve a topology optimization problem with 12 million elements on 300 machines, requiring 212 seconds per optimization iteration and 0.06 seconds per solver iteration. Considering that our implementation has no limitation to generate and simulate any mesh size, we tested the feasibility of the solution for an extremely large-scale problem. A few iterations of a topology optimization example were simulated with 324 million elements on 864 machines and 1 billion elements (3 billion dofs) on 2916 machines, with 0.14 seconds and 0.16 seconds per solver iteration, respectively.

The results obtained here demonstrate the feasibility and computational efficiency of the proposed plugin architecture.



## 5.1 Future Work

We present some suggestions for future work, as follows:

- **A better preconditioner for the element-by-element PCG solver.** The EbEPCG solver still needs some improvements regarding the high number of iterations for convergence. A better pre-conditioner may reduce substantially the computing time of the solver. We are currently investigating the approximate inverse preconditioner [54, 55] applied to topology optimization problems, with preliminary results. Another option is the multigrid pre-conditioned conjugate gradients solver recently presented by Amir et al. [26].
- **Improvements on MPI communications among subdomain partitions.** The distributed solution in the framework still requires a more detailed investigation about the bottlenecks in simulations of large-scale problems with too many cluster nodes. The Blue Waters system provides many performance tools that can be used to fine tune the code to run more efficiently on its architecture.
- **Distributed computing on clusters of GPUs**. The low memory consumption of the EbEPCG solver is well suited for simulations on GPUs. These massive parallel units may speedup extremely large simulations, using fewer cluster machines.

– **Distributed visualization and post-processing of extremely large simulations.** Scalable and efficient visualization methods are very important for simulating large-scale numerical analyses. The visualization of such large examples presents many challenges due to the huge volume of data, resulting from the parallel simulations.

– **Extending the framework for different types of methods and analyses.** The TopSim framework has the flexibility to be extended for solving numerical problems with different methods and analyses. Some researches are already under development to implement the Ground Structure Method and the Interior Point Methods for solving topology optimization problems [56].

# Bibliography


[1] SURESH, K.. **Efficient generation of large-scale pareto-optimal topologies**. Structural and Multidisciplinary Optimization, 47(1):49–61, may 2012.

[2] AAGE, N.; ANDREASSEN, E. ; LAZAROV, B. S.. **Topology optimization using PETSc: An easy-to-use, fully parallel, open source topology optimization framework**. Structural and Multidisciplinary Optimization, 51(3):565–572, aug 2014.

[3] SAAD, Y.. **Iterative Methods for Sparse Linear Systems**. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[4] GUIMARÃES, L. G. S.; MENEZES, I. F. M. ; MARTHA, L. F.. **Disciplina de programação orientada a objetos para sistemas de elementos finitos**. In: ANAIS DO XIII CILAMCE (CONGRESSO IBERO-LATINO AMERICANO SOBRE MéTODOS COMPUTACIONAIS PARA ENGENHARIA), volumen 1, p. 342–351, Porto Alegre, Brasil, 1992.

[5] MARTHA, L. F.; MENEZES, I. F. M.; LAGES, E. N.; PARENTE JR., E. ; PITANGUEIRA, R.. **An OOP class organization for materially nonlinear FE analysis**. In: JOINT CONFERENCE OF ITALIAN GROUP OF COMPUTATIONAL MECHANICS AND IBERO-LATIN AMERICAN ASSOCIATION OF COMPUTATIONAL METHODS IN ENGINEERING, p. 229–232, Padova, Italia, 1996.

[6] MORETTI, C. O.; CAVALCANTE, J. B.; BITTENCOURT, T. N. ; MARTHA, L. F.. **A parallel environment for three-dimensional finite element analysis**. International conference on engineering computational technology, p. 283–287, 2000.

[7] MARGETTS, L.. **Parallel Finite Element Analysis**. PhD thesis, 2002.

[8] SMITH, I. M.; GRIFFITHS, D. V. ; MARGETTS, L.. **Programming the Finite Element Method**. Wiley, 5th edition edition, 2014.

[9] BARTLETT, R. A.; HOWLE, V. E.; HOEKSTRA, R. J.; HU, J. J.; KOLDA, T. G.; LEHOUCQ, R. B.; LONG, K. R.; PAWLOWSKI, R. P.; PHIPPS, E. T.; SALINGER, A. G.; TUMINARO, R. A. Y. S.; WILLENBRING, J. M. ; STANLEY, K. S.. **An Overview of the Trilinos Project**. V(December):1–27, 2004.

[10] BANGERTH, W.; HARTMANN, R. ; KANSCHAT, G.. **deal.II—A general-purpose object-oriented finite element library**. ACM Transactions on Mathematical Software, 33(4):24–es, aug 2007.

[11] WEINBUB, J.; RUPP, K. ; SELBERHERR, S.. **ViennaX: a parallel plugin execution framework for scientific computing**. Engineering with Computers, 30(4):651–668, feb 2013.

[12] MENDES, C. A. T.; GATTASS, M. ; ROEHL, D.. **The GeMA framework – An innovative framework for the development of multiphysics and multiscale simulations**. In: M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. P., editor, VII EUROPEAN CONGRESS ON COMPUTATIONAL METHODS IN APPLIED SCIENCES AND ENGINEERING, Crete Island, Greece, 2016.

[13] TALISCHI, C.; PAULINO, G. H. ; LE, C. H.. **Honeycomb Wachspress finite elements for structural topology optimization**. Structural and Multidisciplinary Optimization, 37(6):569–583, may 2008.

[14] JOG, C. S.; HABER, R. B.. **Stability of finite element models for distributed-parameter optimization and topology design**. Computer Methods in Applied Mechanics and Engineering, 130(3-4):203–226, apr 1996.

[15] DIAZ, A.; SIGMUND, O.. **Checkerboard patterns in layout optimization**. Structural Optimization, 10(1):40–45, 1995.

[16] TALISCHI, C.; PAULINO, G. H.; PEREIRA, A. ; MENEZES, I. F. M.. **Polygonal finite elements for topology optimization : A unifying paradigm**. International Journal for Numerical Methods in Engineering, 82(December 2009):671–698, 2010.

[17] ROZVANY, G.; QUERIN, O.; GASPAR, Z. ; POMEZANSKI, V.. **Weight-increasing effect of topology simplification**. Structural and Multidisciplinary Optimization, 25(5-6):459–465, dec 2003.

[18] TALISCHI, C.; PAULINO, G. H.; PEREIRA, A. ; MENEZES, I. F. M.. **PolyTop: a Matlab implementation of a general topology optimization framework using unstructured polygonal finite element meshes**. Structural and Multidisciplinary Optimization, 45(3):329–357, jan 2012.

[19] TALISCHI, C.; PEREIRA, A.; PAULINO, G. H.; MENEZES, I. F. M. ; CARVALHO, M. S.. **Polygonal finite elements for incompressible fluid flow**. International journal for numerical methods in fluids, 74(October 2013):134–151, 2014.

[20] SIGMUND, O.. **A 99 line topology optimization code written in Matlab**. Structural and Multidisciplinary Optimization, 21(2):120–127, 2001.

[21] ANDREASSEN, E.; CLAUSEN, A.; SCHEVENELS, M.; LAZAROV, B. S. ; SIGMUND, O.. **Efficient topology optimization in MATLAB using 88 lines of code**. Structural and Multidisciplinary Optimization, 43(1):1–16, nov 2010.

[22] PEREIRA, A.; MENEZES, I. F. M.; TALISCHI, C. ; PAULINO, G. H.. **An efficient and compact Matlab implementation of topology optimization: Application to compliant mechanism**. In: PROCEEDINGS OF THE XXXII IBERIAN LATIN AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING, Ouro Preto - MG, Brazil, 2011.

[23] DEATON, J. D.; GRANDHI, R. V.. **A survey of structural and multidisciplinary continuum topology optimization: post 2000**. Structural and Multidisciplinary Optimization, 49(1):1–38, jul 2014.

[24] SURESH, K.. **A 199-line Matlab code for Pareto-optimal tracing in topology optimization**. Structural and Multidisciplinary Optimization, 42(5):665–679, jul 2010.

[25] AAGE, N.; LAZAROV, B. S.. **Parallel framework for topology optimization using the method of moving asymptotes**. Structural and Multidisciplinary Optimization, 47:493–505, jan 2013.

[26] AMIR, O.; AAGE, N. ; LAZAROV, B. S.. **On multigrid-CG for efficient topology optimization**. Structural and Multidisciplinary Optimization, 49(5):815–829, nov 2014.

[27] YADAV, P.; SURESH, K.. **Large Scale Finite Element Analysis via Assembly-Free Deflated Conjugate Gradient**. Journal of Computing and Information Science in Engineering, 14(4):041008, 2014.

[28] DUARTE, L. S.; CELES, W.; PEREIRA, A.; IVAN, I. F. ; PAULINO, G. H.. **PolyTop++: an efficient alternative for serial and parallel topology optimization on CPUs & GPUs**. Structural and Multidisciplinary Optimization, 52(5):845–859, 2015.

[29] NVIDIA. **Cuda programming guide 5.5**, 2013, https://developer.nvidia.com/cuda-downloads.

[30] CELES, W.; PAULINO, G. H. ; ESPINHA, R.. **A compact adjacency-based topological data structure for finite element mesh representation**. International Journal for Numerical Methods in Engineering, 64(11):1529–1556, nov 2005.

[31] BENDSOE, M. P.. **Optimal shape design as a material distribution problem**. Structural Optimization, 202:193–202, 1989.

[32] SCHMIDT, S.; SCHULZ, V.. **A 2589 line topology optimization code written for the graphics card**. Computing and Visualization in Science, 14(6):249–256, aug 2012.

[33] GAIN, A. L.; PAULINO, G. H.. **A critical comparative assessment of differential equation-driven methods for structural topology optimization**. Structural and Multidisciplinary Optimization, 48(4):685–710, jul 2013.

[34] OSHER, S.; FEDKIW, R. P.. **Level Set Methods: An Overview and Some Recent Results**. Journal of Computational Physics, 169(2):463–502, may 2001.

[35] OSHER, S.; SETHIAN, J. A.. **Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations**. Journal of Computational Physics, 79(1):12–49, nov 1988.

[36] BOURDIN, B.; CHAMBOLLE, A.. **Design-dependent loads in topology optimization**. 9(January):19–48, 2003.

[37] BENDSOE, M.; SIGMUND, O.. **Topology Optimization: Theory, Methods, and Applications**. Engineering online library. Springer Berlin Heidelberg, 2003.

[38] LEON, S. E.; PAULINO, G. H.; PEREIRA, A.; MENEZES, I. F. M. ; LAGES, E. N.. **A Unified Library of Nonlinear Solution Schemes**. Applied Mechanics Reviews, 64(July):040803, 2011.

[39] LAGES, E.; PAULINO, G.; MENEZES, I. ; SILVA, R.. **Nonlinear Finite Element Analysis using an Object-Oriented Philosophy – Application to Beam Elements and to the Cosserat Continuum**. Engineering with Computers, 15(1):73–89, jan 1999.

[40] TALISCHI, C.; PAULINO, G. H.; PEREIRA, A. ; MENEZES, I. F. M.. **PolyMesher: a general-purpose mesh generator for polygonal elements written in Matlab**. Structural and Multidisciplinary Optimization, 45(3):309–328, jan 2012.

[41] TEWARSON, R. P.. **Sparse Matrices**. Mathematics in Science and Engineering. Elsevier Science, 1973.

[42] GRCAR, J. F.. **Mathematicians of Gaussian elimination**. Notices of the American Mathematical Society, 58(6):782 – 792, 2011.

[43] TIMOTHY A. DAVIS; IAIN S. DUFF. **An unsymmetric-pattern multifrontal method for sparse LU factorization**. SIAM Journal on Matrix Analysis and Applications, 18(1):140–158, 1997.

[44] SCHENK, O.; GÄRTNER, K.. **Solving unsymmetric sparse systems of linear equations with PARDISO**. Future Generation Computer Systems, 20(3):475–487, apr 2004.

[45] WANG, S.; DE STURLER, E. ; PAULINO, G. H.. **Large-scale topology optimization using preconditioned Krylov subspace methods with recycling**. International journal for numerical methods in engineering, (September 2006):2441–2468, 2007.

[46] AUGARDE, C.; RAMAGE, A. ; STAUDACHER, J.. **An element-based displacement preconditioner for linear elasticity problems**. Computers & Structures, 84(31-32):2306–2315, dec 2006.

[47] GEBREMEDHIN, A. H.; MANNE, F. ; POTHEN, A.. **What Color Is Your Jacobian? Graph Coloring for Computing Derivatives**. Society for Industrial and Applied Mathematics, 47(4):629–705, jan 2005.

[48] OPENMP ARCHITECTURE REVIEW BOARD. **OpenMP application program interface version 3.1**, 2011, http://www.openmp.org/.

[49] COOK, R. D.. **Finite Element Modeling for Stress Analysis**. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[50] LEE, S.-L.; MANUEL, F. ; ROSSOW, E.. **Large deflections and stability of elastic frames**. Journal of Engineering Mechanics (ASCE), 94(EM2):521–547, 1968.

[51] SURESH, K.. **Large-Scale Matrix-Free Topology Optimization on the GPU - GTC 2012**. In: GTC 2012, 2012.

[52] KARYPIS, G.; KUMAR, V.. **A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs**. SIAM Journal on Scientific Computing, 20(1):359–392, 1998.

[53] ESPINHA, R.; CELES, W.; RODRIGUEZ, N. ; PAULINO, G. H.. **ParTopS: compact topological framework for parallel fragmentation simulations**. Engineering with Computers, 25(4):345–365, jun 2009.

[54] BENZI, M.; BERTACCINI, D.. **Approximate inverse preconditioning for shifted linear systems**. BIT Numerical Mathematics, 43(2):231–244, 2003.

[55] BENZI, M.; CULLUM, J. K. ; TUMA, M.. **Robust Approximate Inverse Preconditioning for the Conjugate Gradient Method**. SIAM Journal on Scientific Computing, 22(4):1318–1332, 2000.

[56] ZEGARD, T.; PAULINO, G. H.. **GRAND3 - Ground structure based topology optimization for arbitrary 3D domains using MATLAB**. Structural and Multidisciplinary Optimization, 52(6):1161–1184, 2015.