UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Mecânica

GILBERTO LUIS VALENTE DA COSTA

High Order Finite Element Method on the High Performance Hybrid Architecture Applied on Structural Mechanics

Método de Elementos Finitos de Alta Ordem e Alto Desempenho em Arquiteturas Híbridas Aplicado à Mecânica Estrutural

CAMPINAS
2020

GILBERTO LUIS VALENTE DA COSTA

High Order Finite Element Method on the High Performance Hybrid Architecture Applied on Structural Mechanics

Método de Elementos Finitos de Alta Ordem e Alto Desempenho em Arquiteturas Híbridas Aplicado à Mecânica Estrutural

Thesis presented to the School of Mechanical Engineering of the University of Campinas in partial fulfillment of the requirements for the PhD degree, in the field of Solid Mechanics and Mechanical Design.

Tese apresentada à Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Doutor em Engenharia Mecânica, na Área de Mecânica dos Sólidos e Projeto Mecânico.

Orientador: Prof. Dr. Marco Lúcio Bittencourt

Co-orientador Prof. Dr. Edson Borin

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA TESE DEFENDIDA PELO ALUNO GILBERTO LUIS VALENTE DA COSTA, CO-ORIENTADA PELO PROF. DR. EDSON BORIN E ORIENTADA PELO PROF. DR. MARCO LÚCIO BITTENCOURT.

CAMPINAS

2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

V234h

Valente, Gilberto Luis, 1983-
High order finite element method on the high performance hybrid architecture applied on structural mechanics / Gilberto Luís Valente da Costa. – Campinas, SP : [s.n.], 2020.

Orientador: Marco Lúcio Bittencourt.
Coorientador: Edson Borin.
Tese (doutorado) – Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica.

1. Métodos dos elementos finitos. 2. Computação de alto desempenho. 3. C++ (Linguagem de programação de computador). 4. Programação paralela (Computação). 5. Supercomputadores. I. Bittencourt, Marco Lúcio, 1964-. II. Borin, Edson, 1979-. III. Universidade Estadual de Campinas. Faculdade de Engenharia Mecânica. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Método de elementos finitos de alta ordem e alto desempenho em arquiteturas híbridas aplicado à mecânica estrutural
Palavras-chave em inglês:
Finite element method
High Performance computing
C++ (Computer program language)
Parallel computers - Programming
Supercomputers
Área de concentração: Mecânica dos Sólidos e Projeto Mecânico
Titulação: Doutor em Engenharia Mecânica
Banca examinadora:
Marco Lúcio Bittencourt [Orientador]
Alvaro Luiz Gayoso de Azeredo Coutinho
Marcilio Alves
Philippe Remy Bernard Devloo
Josué Labaki Silva
Data de defesa: 28-08-2020
Programa de Pós-Graduação: Engenharia Mecânica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-7594-3084
- Currículo Lattes do autor: http://lattes.cnpq.br/4560031114548916

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA

TESE DE DOUTORADO ACADÊMICO

High Order Finite Element Method on the High Performance Hybrid Architecture Applied on Structural Mechanics

Método de Elementos Finitos de Alta Ordem e Alto Desempenho em Arquiteturas Híbridas Aplicado à Mecânica Estrutural

Autor: Gilberto Luis Valente da Costa
Orientador: Prof. Dr. Marco Lúcio Bittencourt
Co-orientador: Prof. Dr. Edson Borin

A Banca Examinadora composta pelos membros abaixo aprovou esta Tese:

Prof. Dr. Marco Lúcio Bittencourt, Presidente
FEM/UNICAMP

Prof. Dr. Philippe Remy Bernard Devloo
FEC/UNICAMP

Prof. Dr. Josué Labaki Silva
FEM/UNICAMP

Dr. Alvaro Luiz Gayoso de Azeredo Coutinho
Universidade Federal do Rio de Janeiro

Prof. Dr. Marcilio Alves
Universidade de São Paulo

A Ata da defesa com as respectivas assinaturas dos membros encontra-se no processo de vida acadêmica do aluno.

Campinas, 28 de agosto de 2020.

À memória de minha mãe e meus avós, Eugênia Cornélia Valente, Pedro Catarino Valente e Luiza Marçal Valente que foram os pilares para minha formação. Dedico sempre pelo apoio que me deram e pelo exemplo de respeito, simplicidade, honestidade e amor ao próximo. Saudades eternas!

ACKNOWLEDGEMENTS

À Deus, a minha maior fonte de força espiritual, fé e determinação. Agradeço sempre pelo dom da vida.

Ao meu orientador Prof. Marco Lúcio pela oportunidade deste trabalho, dedicação, paciência, amizade e pelas importantes contribuições ao longo deste período.

Ao meu coorientador Prof. Edson Borin pela sua disponibilidade e suporte sempre que necessário na elaboração deste trabalho.

Ao Prof. Renato Pavanello pela ajuda e alocação de recursos para o projeto no cluster Kahuna do CCES - Fapesp 2013/08293-7 do IQ-UNICAMP.

Estes anos de pesquisa foram marcados de desafios, construção e amadurecimento. Também houve perdas e nestas perdas não tem como deixar a minha gratidão eterna a minha família que se foi, a minha mãe Eugênia Cornélia, o meu avô Pedro Catarino, a minha avó Luíza Marçal, o meu tio Pedro. Obrigado por todo suporte aonde vocês estiverem. Esta conquista também pertence a vocês.

Agradeço também ao amor da mulher que Deus colocou na minha vida, Gabriela Thais, pelo seu carinho, compreensão e apoio em todos os momentos ao longo destes anos.

Agradeço a todos os meus familiares. A minha grande e querida família que envolve tios e tias que me tratam como filho, primos e primas que me tratam como irmão. Agradeço a todos vocês que me acompanharam mesmo distantes, buscaram dar suporte em vários momentos. Aos amigos que deram suporte a este trabalho: Jorge Susuki, Allan Dias, Fabiano Bargos, Jaime Izuka, Luan Franchini, Caio Rodrigues, Geovane Haveroth, Victor Campos, Gabriel Welfany e Luis Renato.

A todos os amigos do laboratório pela convivência e amizade: Alfredo, Mari, Cezar, Alan, Luis Teixeira, Ana Luisa, Thais, Matheus, Paola Ramos, Darla Caroline, Pedro, Guilherme.

Aos amigos conquistados nestes anos de Campinas e UNICAMP. Ao José Guilherme, Carolliny Miranda, Fabrício Luis, Andrei Braga e Alice Cordovil.

Aos amigos de república, Camilo Ariza, Maria Fernanda e Daniel Garcia. Agradeço também a todos os amigos colombianos, que conviveram constantemente comigo, Oscar Rojas, German Castañeda, Diana Martinez, German Buitrago, Joan Sebastian Chaves, Suranny Jiménez, Ramiro Chamorro, Jenny Lombo, Manuel Arcila e Camilo Gordillo e tantos outros. Agradeço pela amizade, apoio e por grandes momentos de convivência.

À Empresa Dom Rock, pelo suporte na realização deste trabalho. Aos ex-colegas

de trabalho da empresa Dom Rock, os quais se tornaram amigos e importantes no suporte da minha vida profissional.

As pessoas que contribuíram na revisão deste texto, André Almeida, Marcelo Lavor e Gabriel Welfany.

A todos os amigos e familiares de quem eu possa não ter lembrado de colocar o nome aqui, mas que de alguma forma direta ou indireta colaboraram para que eu pudesse alcançar esta conquista.

I acknowledge the computing resources provided on IBM Blue Gene/P and Q, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. Finalmente, ao Centro de Computação em Engenharia e Ciências (CCES - Fapesp 2013/08293-7) pelos recursos computacionais e suporte a esta pesquisa.

This study was financed in part by the São Paulo Research Foundation (FAPESP), grant 2012/19922-2.

“In the middle of every difficulty lies opportunity.”
(Albert Einstein)

RESUMO

Este trabalho descreve a implementação de uma arquitetura de software serial e paralela denominada $(hp)^2$FEM para o Método de Elementos Finitos de Alta Ordem (MEF-AO). O projeto do software foi realizado de forma a facilitar e promover a reutilização e a manutenção do código. A implementação baseia-se no paradigma de programação orientada a objeto em C++. É possível usar diferentes ordens de aproximação para os elementos da malha. O uso de malhas com distribuição de graus não-uniformes permite aumentar a ordem de interpolação apenas nos elementos com maior gradiente na solução aproximada. Procedimentos eficientes para o cálculo das matrizes elementares permitem ganhos expressivos em termos de tempo de processamento e memória. Um algoritmo local baseado no método de mínimos quadrados é apresentado para a obtenção dos coeficientes da aproximação. Este algoritmo requer a solução de um sistema linear de equações para cada elemento e obtém os coeficientes da aproximação local pela inversão das matrizes elementares. A solução global para os coeficientes compartilhados por dois ou mais elementos é obtida por uma média ponderada das soluções locais e das medidas dos elementos (comprimento, área e volume). Resultados para distribuição de ordens não-uniforme são apresentados para malhas de quadrados e hexaedros no problema de projeção. O perfilamento do código é analisado para quantificar os ganhos em termos de memória e processamento. Resultados de escalabilidade para paralelismo híbrido com OpenMP e MPI apresentam um bom ganho de velocidade, solucionando problemas estruturais transientes lineares e não-lineares com integração temporal explícita. Além disso, verificou-se escalabilidade forte e fraca para o modelo paralelo utilizando bibliotecas de álgebra linear otimizadas. A escalabilidade e o perfilamento foram avaliados no computador IBM Blue Gene/Q - Mira executando o solver local de projeção em 32768 nós de computação com até 840 milhões de graus de liberdade. O procedimento local explícito foi processado no computador Kahuna com processadores HT Intel Xeon E5-2670, localizado no CCES Unicamp. As análises de desempenho do algoritmo paralelo explícito foram realizadas para malhas de um virabrequim com até $1,791$ milhões de elementos e 400 milhões de graus de liberdade.

Palavras-Chave: Arquitetura de Software; Método dos Elementos Finitos; Métodos de Alta Ordem; C++; Programação Paralela; MPI; OpenMP; IBM Blue Gene.

ABSTRACT

This work describes the implementation of a serial and parallel software architecture called $(hp)^2$FEM for the high-order finite element method (HO-FEM). The software was designed to facilitate reusability and ease of maintenance. The implementation is based on the object-oriented paradigm in C++. It is possible to use different polynomial approximation orders for the mesh elements. The use of meshes with non-uniform degree distribution allows increasing the polynomial order just in elements with higher gradients in the approximate solution. Efficient procedures to calculate element matrices allow significant gains in terms of processing time and memory. A local algorithm based on the least-squares method is presented to obtain the approximation coefficients. This algorithm requires the solution of a linear system of equations for each element and obtain the local approximation coefficients by the inversion of the element matrices. The global solution for the coefficients shared by two or more elements are obtained by a weighted average of the local solutions and element measures (length, area or volume). Results for non-uniform order distribution are presented for meshes of squares and hexahedra with the projection problem. Code profile is analyzed to quantify the gains in terms of memory and processing. Scalability results for hybrid parallelism with OpenMP and MPI presented good speedup, solving linear and non-linear transient analysis problems with explicit time integration. In addition, there were weak and strong scalability for the parallel model using optimized linear algebra libraries. Scalability and profiling were evaluated in the IBM Blue Gene/Q Mira computer running the projection local solver on 32768 computer nodes with up to 840 million degrees of freedom. The explicit local solver was run in the Kahuna cluster with HT Intel Xeon E5-2670 processors, located at CCES Unicamp. Performance analyzes of the parallel solver in this cluster were performed by running crankshaft meshes with up to $1,791$ million elements and 400 million of degrees of freedom.

Keywords: Software Architecture; Finite Element Method; High Order Methods; C++; Parallel Programming; MPI; OpenMP; IBM Blue Gene.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

PDE	Partial Differential Equation
FEM	Finite Element Method
HO-FEM	High Order Finite Element Method
HPC	High Performance Computing
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
ID	Process or Partition Identification Number
DOF	Degree of Freedom
CGD	Conjugate Gradient Method with Diagonal Pre-conditioner
BLAS	Basic Linear Algebra Subprograms
LAPACK	Linear Algebra Package
PAPI	Performance Application Programming Interface
SIMD	Single Instruction, Multiple Data
FLOPS	Floating Point Operations Per Second
DDT	Distributed Debugging Tool

CONTENTS

1 INTRODUCTION

Computer simulations use numerical models to analyze physical problems, allowing the modification of parameters and initial conditions to provide relevant information about their behavior. In this context, the finite element method (FEM) is a numerical procedure that obtains approximate solutions of boundary value problems (BVPs). This method calculates the approximate solution of BVPs in the whole domain through a discretization into sub-domains called finite elements. The high-order FEM (HO-FEM), also called spectral *hp*-FEM, obtains a converged approximate solution by mesh refinement and increasing the polynomial order of the interpolation functions (KARNIADAKIS; SHERWIN, 2005; BITTENCOURT, 2014). The development of finite element software architectures that support several applications and are generic, flexible, and reusable has been considered in the literature (MACKERLE, 2004; BANGERTH *et al.*, 2007; KAWABATA *et al.*, 2009; ANZT *et al.*, 2010). The HO-FEM makes significant demands on memory and processing resources, mainly in complex applications where high-performance computing is required.

Many commercial software implementing the FEM have been developed in the last six decades, for instance Ansys, Nastran, Adina, LS-Dyna, Abaqus, HyperMesh, COMSOL, among others. Most of them implements the *h*-version of the FEM, where the approximate solution is improved by refining the size of elements, for many different problems including structures, electromagnetism, fluid mechanics, acoustics and many others. The *p*- and *hp*-versions of the FEM, later called HO-FEM, were created at the end of the 1970 decade and the main feature is the exponential convergence rate of approximations for smooth solutions. One of the main historical difficulties to the dissemination of these versions was the lack of commercial and open-source software. StressCheck was one of the first finite element analysis software based on the *p*-FEM with focus on solid mechanics. However, it was inefficient for the analysis of practical problems (ESRD, 2009).

Analogously, many open source software for the FEM have been developed such as CalculiX, deal.II, Dune, Elmer, FEniCS, FreeFEM, GetFEM++, Hermes, Libmesh, MOOSE, RANGE, Z88, PZ, among others. Many of these software packages implement the *p*- and *hp*-versions of the FEM and are based on the object-oriented paradigm using the C++ and Python languages.

Nowadays, there is a large community working on high-order numerical methods, see references herein (KARNIADAKIS; SHERWIN, 2005; BITTENCOURT, 2014). Along the last two decades, many improvements were developed including basis functions, efficient numerical integration and differential quadrature, efficient solvers and pre-conditioners, mesh generation, visualization of results and applications in many different problems.

The $(hp)^2$FEM software was developed based on previous versions of the ACD-POOP and SAT software (GUIMARÃES; FEIJÓO, 1989; BITTENCOURT *et al.*, 1998) implemented at the National Laboratory for Scientific Computing (LNCC) and by the UNICAMP group where this thesis was developed. This group developed also the $(hp)^2$FEM MATLAB version applied mainly to structural analysis (NOGUEIRA Jr., 2002; VAZQUEZ, 2004; VAZQUEZ, 2008; MIANO, 2009; BARGOS, 2009; SANTOS, 2011; FURLAN, 2011; AUGUSTO, 2012). The MATLAB architecture was designed taking into account arbitrary polynomial orders, use of nodal and modal bases, numerical integration procedures and efficient calculation of element operators.

In Valente (2012), the $(hp)^2$FEM MATLAB architecture was expanded and ported to the C++ language. The work of this thesis consisted of the hybrid parallelism implementation applied to different problems. The local solution algorithms based on the least squares method and the possibility of using meshes with non-uniform polynomial distribution on hybrid architectures make the software suitable for the next generation of exaflops computers (BUNGARTZ *et al.*, 2020).

1.1 LITERATURE REVIEW AND RELATED FINITE ELEMENT SOFTWARE

Bangerth *et al.* (2007) proposed a modular architecture and a partitioned library in C++ for the FEM without loss of performance. The goal was to organize the code in independent modules that can be arbitrarily used. In specific cases, preprocessor directives and constants are used to construct finite element meshes, avoiding the use of virtual methods and reducing the system overhead.

Cantwell *et al.* (2011) proposed to encapsulate the spectral/*hp* method in the Nektar++ software for fluid dynamics applications ("NECKTAR++..., 2013). The best combinations of h (mesh size) and p (polynomial order) refinements were studied for the Helmholtz problem.

In Kawabata *et al.* (2009), Fu (2008), authors used the FEM in parallel cluster environments and identified the bottlenecks of running the parallel software. The work in Kawabata *et al.* (2009) identified the parts of code with the highest memory demand and time consumption, and then implemented parallel processing in these pieces of code. In Fu (2008), three different meshes were considered for a two-dimensional plane strain model of a dam and its foundations. Two parameters were changed: the interpolation order for each mesh and the number of processors. The author concluded that the increase in degrees of freedom enabled better performance. However, the increased number of processors improved the performance up to a certain limit depending on the communication and synchronization overheads.

In terms of parallel strategies for finite difference schemes, a space-filling curve algorithm was used to solve workload imbalance of static partition of the domain in (WANG *et*

al., 2015a; WANG *et al.*, 2015b). The development of an adaptive mesh refinement procedure based on finite differences was proposed in (WANG *et al.*, 2015a). In Wang *et al.* (2015b), ParMetis is used for grid partitioning of a black oil discrete model. Other works involve different numerical methods such as (OGUIC *et al.*, 2015), where the 3D Navier-Stokes equations based on fourth-order compact schemes are solved, with hybrid OpenMP/MPI parallelization, presenting attractive scalability when using up to four MPI tasks. A different parallel strategy was used by (CHAN *et al.*, 2016), focusing on high order discontinuous Galerkin methods using a single GPU with an unified approach to multi-threading programming model.

The deal.ii package is a C++ library with a Lesser General Public License (LGPL) (BANGERTH *et al.*, 2007). The unified interface of deal.ii can handle problems in one, two, or three dimensions and enables adaptive mesh refinement through error estimators and local indicators. This FEM package works by adapting h, p, and hp for continuous and discontinuous Lagrange, Nedelec, and Raviart-Thomas elements for any polynomial order. deal.ii also implements parallelism, with scalable simulations of up to 16 000 processors using multithreading and Message Passing Interface (MPI) ranks.

The Libmesh package was created with the goal of supporting adaptive h-refinement. Nowadays, it also works with finite element and finite volume simulations for p and hp refinements for some element types (KIRK *et al.*, 2006). The libraries offer procedures that allow developers to perform a few calls of the principal functions, rather than many calls to smaller functions, thus avoiding the overhead caused by virtual function calls of abstract base classes. Code debugging for smaller problems in 2D can be applied immediately to large problems. Libmesh uses algorithms from METIS and PARMETIS to partition weighted graphs in serial and parallel for 1D, 2D, and 3D meshes. The software uses other external tools as iterative solvers and preconditioners in serial applications such as LASPack and parallel applications using PETSc.

The z88 freeware software project operates on Linux, Windows, and Mac OS X (RIEG, 2014). This FEM package can solve non linear problems with large displacements, linear static analysis, thermal and thermomechanical problems, and natural frequency problems. This software has three kinds of solvers: a direct Cholesky solver with Jennings storage, an iterative solver for sparse matrices with conjugate gradient (CG) preconditioners and a direct solver with sparse storage and multi-core processing.

In addition to the FEM, the Distributed and Unified Numerics Environment (DUNE) supports finite volume and finite difference methods (BLATT; BASTIAN, 2008). DUNE is offered with a GPL 2 license with runtime exception, which allows its use in proprietary software. Among the available linear algebra methods, DUNE uses BLAS1 (basic linear algebra subroutine) functions, stationary iterative methods, and parallel preconditioners for the multigrid method. The software also supports parallelization with h and p refinements. The implemented solvers can handle linear and non linear problems using time discretization methods,

e.g., Runge–Kutta and multistep methods. DUNE uses static polymorphism, allowing the compiler to apply additional optimization, e.g., inline functions. The software is modulated to allow each package to be tested independently.

Nektar++ is a finite element package that allows operations with low (*h*-version) and high (*p*-version) polynomial orders (CANTWELL *et al.*, 2015). Nektar++ is designed to operate with meshes of hybrid elements, i.e., prisms and hexahedrons, triangles, quadrilaterals, and tetrahedra. In addition, it is possible to define domains with segments, planes, volumes, curves, and surfaces. The software works with continuous and discontinuous Galerkin operators. Nektar++ has a parallel version which had strong scalability up to 2048 cores.

The Hermes software was developed as a fast development of FEM solvers (SOLIN *et al.*, 2014). It implements *hp* adaptivity, but only works on 2D applications and for polynomial orders of up to 10. Furthermore, Hermes runs with multithreading parallelization using OpenMP. This package has its own module for data visualization using OpenGL and VTK to output meshes and solutions.

The library GetFEM++ solves linear and nonlinear systems of partial differential equations (“GETFEM++. . . ,). The library is written in C++ with interfaces for Python, Matlab, and Scilab, which collaborate with GetFEM++ for post-processing operations. Among the main features, the library was designed with variables, data, and terms for the solution of classic models such as the Helmholtz problem with Dirichlet, contact and Neumann boundary conditions and elasticity in small and large deformations. There is also a module for generating regular meshes or importing meshes with GID, GMSH, and EMC2 formats.

The Multiphysics Object-Oriented Simulation Environment (MOOSE) is a finite element framework (GASTON *et al.*, 2009) with high-level interfaces and simple APIs that facilitate the implementation of real-world problems. MOOSE performs *h*, *p*, and *r* refinements. It is possible to build the domain geometry, generate meshes, and visualize the solution process and final results. The FEM module and mesh adaptivity are accessed through the libMesh library.

The object-oriented framework PZ implements one-, two- and three-dimensional finite elements with arbitrary interpolation orders, various solvers as Krylov space, direct and iterative methods. The PZ makes a substantial distinction between the generation of approximation spaces, the geometric modeling, and the definition of the variational statement (DEVLOO, 1997; DEVLOO; LONGHIN, 2002).

Unlike the packages mentioned above, the purpose of this work was the implementation of the HO-FEM software with local solvers and hybrid parallelism. In addition, the software allow, using local solvers, the simpler use of non-uniform interpolation in the elements.

1.2 RESEARCH OBJECTIVES AND CONTRIBUTIONS

The main objective of this thesis was to develop an open source software for the HO-FEM using local algorithms, based on least squares, for transient linear and non-linear structural problems with explicit temporal integration. The system of equation are solved for each element and the global solutions are obtained by the weighted average of the local solutions at the element interfaces. These local algorithms were implemented for parallel execution in clusters of heterogeneous machines with multiple nodes and cores. The $(hp)^2$FEM package was written in C ++ and is available at www.hp2fem.org. It has been tunned for the IBM Power and Intel-X86 architectures, allowing testing and profiling of the sub-packages independently.

The main contributions of this work are as follows:

- The development of a software architecture for the HO-FEM and its parallel implementation on hybrid computers using the libraries MPI and OpenMP, considering the possibility of non-uniform polynomial distribution (p-non-uniform) for the elements.
- A topology mapping for the sub-domain meshes avoiding deadlock among processors during the exchange of messages. The finite element coordinates are used to construct the mapping of the local degrees of freedom stored in each partition to send and receive solutions among neighbor sub-domain meshes.
- Scalable hybrid parallel implementation of linear and non-linear transient solvers using linear algebra packages such as BLAS and LAPACK. The high order data are generated locally in each sub-domain reducing the memory usage.

We presented the scientific works in the following conferences:

- J.L. Suzuki, G.L. Valente, C.F. Rodrigues, M.L. Bittencourt, Application of high-order minimum energy bases for transient non-linear structural problems. MECSOL-2017, Joinville, 2017.
- G.L. Valente, E. Borin. and M.L. Bittencourt. Optimization of the high performance software $(hp)^2$FEM : a C++ framework for the high-order finite element method. ICOSAHOM2016 - International Conference on Spectral and High Order Methods, Rio de Janeiro, RJ, 2016.
- G.L. Valente, M.L. Bittencourt and E. Borin. High order finite element method on the IBM power systems. High performance computing applied to structural mechanics. 5th European Conference on Computational Mechanics (ECCM V), Barcelona, Espanha, 2014.

- G.L. Valente, M.L. Bittencourt and E. Borin. Perfilamento e otimização do $(hp)^2$FEM no IBM Blue Gene/P. IV Escola Regional de Alto Desempenho de São Paulo. São Carlos, SP, 2013.

The following papers have been submitted to journals:

- G.L. Valente, M.L. Bittencourt and E. Borin. $(hp)^2$FEM - A p-non-uniform software architecture for the high-order finite element method on massively parallel computers.
- A.P.C. Dias, J.L. Suzuki, G.L. Valente and M.L. Bittencourt,. Minimum energy high-order bases applied to non-linear structural and contact problems.

1.3 LAYOUT OF THE THESIS

The thesis is presented in seven chapters. Concepts of the HO-FEM important to this work are described in Chapter 2: the tensor product procedure to calculate the element operators which were fundamental to accelerate $(hp)^2$FEM reducing the memory usage; the local solvers of the $(hp)^2$FEM ; the *p*-non-uniform approach applied to the projection problem; and the central difference local solvers applied to linear and non-linear problems.

Chapter 3 presents the primary concepts of high-performance computing (HPC) used in this thesis. In summary, the characteristics of a parallel system using distributed and shared memory, the interfaces and/or libraries employed to implement the parallel version of $(hp)^2$FEM and the main scalability measures are described.

The framework architecture is presented in Chapter 4 with the low-coupling hierarchy of classes implemented with C++ object-oriented programming. This chapter also presents a short description of the framework to implement the HO-FEM package. Subsequently, the key features of the $(hp)^2$FEM considered as contributions are also discussed. We also describe the parts of $(hp)^2$FEM and the C++ classes which were remodeled to accelerate the serial version; for instance, adapting some functions to use external optimized linear algebra libraries.

The parallel version of the $(hp)^2$FEM is discussed in Chapter 5. The implementation of partitioning of the global finite element meshes and the methodology to distribute them among processors are described. Besides, four different cases to distribute and solve a global finite element model are presented. At last, a parallel version of the central difference local method using hybrid parallelism with OpenMP and MPI is discussed.

Results are presented in Chapter 6 considering the evolution of the software acceleration of the serial version to the scalability of the parallel version. We give the configurations of the supercomputers used at the Argonne National Laboratory of the U.S Department of Energy and the cluster of Intel system provided by the Center for Computational Engineering & Sciences located at the Institute of Chemistry of Unicamp. We present the software validation

for p-non-uniform meshes to projection problem as the scalability cases to the same solver. For time integration solvers, we use the central difference local method to evaluate the scalability of the parallel version of the $(hp)^2$FEM . The final considerations and future works are then presented in Chapter 7.

2 FUNDAMENTAL CONCEPTS OF THE HIGH ORDER - FINITE ELEMENT METHOD (HO-FEM)

In traditional FEM, the most significant demand for computing resources comes from solving the systems of equations obtained from an approximation procedure. For HO-FEM, the increase in the order of shape functions results similarly in larger processing time for calculating the element matrices. In some cases, the computation time is of the same order of magnitude as for the solution of the systems of equations (KARNIADAKIS; SHERWIN, 2005; BITTENCOURT, 2014). Also, the element matrices and shape functions calculated on integration points require a large amount of memory.

This chapter describes a technique based on the tensor product of one-dimensional matrices to construct square and hexahedron matrices. Similarly, we present a technique for solving systems of equations based on element matrices and least-squares smoothing on the element boundaries. We describe the projection problem used as an application to accelerate $(hp)^2$FEM . Additionally, element wise linear and non-linear transient solvers with explicit time integration are described.

This chapter does not intend to present an overall description of the HO-FEM, but just the main aspects used in this work. More complete presentation of the HO-FEM may be found in (KARNIADAKIS; SHERWIN, 2005; BITTENCOURT, 2014)

2.1 NODAL BASIS FUNCTIONS

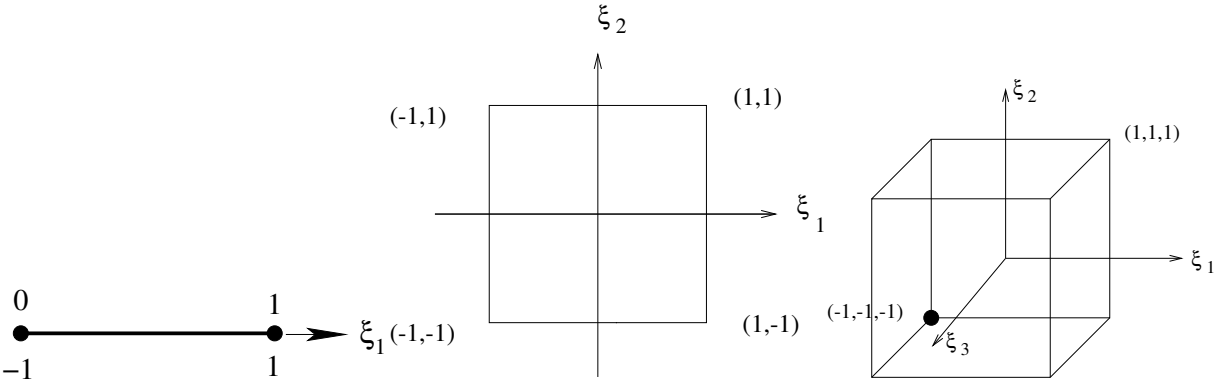
This section summarizes the construction of nodal interpolation functions for line, square, and hexahedron finite elements defined in their standard coordinate systems, illustrated in Figure 2.1, using the Lagrange polynomials according to Karniadakis and Sherwin (2005), Bittencourt (2014).

Consider a set of P_1+1 nodal or collocation points on the standard one-dimensional element in the interval $-1 \leqslant \xi_1 \leqslant 1$, as illustrated in Figure 2.2. The Lagrange polynomial of degree P_1 associated to an arbitrary node a, denoted as $L_a^P(\xi_1)$, is given by

$$L_a^{(P_1)}(\xi_1) = \frac{\prod_{b=0,a\neq b}^{P}\left(\xi_1 - \xi_{1_b}\right)}{\prod_{b=0,a\neq b}^{P}\left(\xi_{1_a} - \xi_{1_b}\right)}, \tag{2.1}$$

where $L_a^{(0)}(\xi_1) = 1$. The Lagrange polynomials have the collocation property $L_a^{(P_1)}(\xi_{1_b}) = \delta_{ab}$, where δ_{ab} is the Kronecker's delta.

The nodal shape functions of the one-dimensional elements, denoted as $N_p(\xi_1)$, are given by the Lagrange polynomials. The shape functions are commonly associated to the element topological entities. In the case of the line element, the topological entities are the



(a) Line element in the local coordinate system ξ_1

(b) Square element in the local coordinate system $\xi_1 \times \xi_2$.

(c) Hexahedron element in the local coordinate system $\xi_1 \times \xi_2 \times \xi_3$.

Figure 2.1. Line, square and hexahedron elements in their standard coordinate systems (BITTENCOURT, 2014).

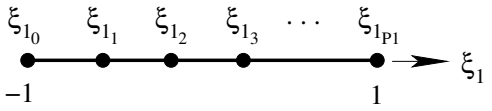


Figure 2.2. Nodal points on the standard coordinate system ξ_1 of the line element (BITTENCOURT, 2014).

vertices and body, which corresponds to the node indices ($p = 0$ and $p = P_1$) and ($0 < p < P_1$), respectively. The interpolation functions are indicated by

$$N_p(\xi_1) \mapsto \phi_p(\xi_1) = \begin{cases} L_0^{(P_1)}(\xi_1), & p = 0, \\ L_{P_1}^{(P_1)}(\xi_1), & p = P_1, \\ L_p^{(P_1)}(\xi_1), & 0 < p < P_1, \end{cases} . \tag{2.2}$$

For square and hexahedron elements, the shape functions are given by the tensor product of the one-dimensional shape functions (see Figures 2.3 and 2.4), respectively, by Bittencourt (2014)

$$N_i(\xi_1, \xi_2) = \phi_p(\xi_1)\phi_q(\xi_2), \tag{2.3}$$

$$N_i(\xi_1, \xi_2, \xi_3) = \phi_p(\xi_1)\phi_q(\xi_2)\phi_r(\xi_3), \tag{2.4}$$

where p, q and r are tensor product indices associated with the topological entities of the element such that $0 \leq p \leq P_1$, $0 \leq q \leq P_2$ and $0 \leq r \leq P_3$; P_1, P_2 and P_3 are the polynomial orders in directions ξ_1, ξ_2, and ξ_3, respectively; $i = 1, \ldots, (P+1)^2$ for square and $i = 1, \ldots, (P+1)^3$ for hexahedron. We assume here that $P = P_1 = P_2 - P_3$. It is possible to define procedures to construct the tensor indices p, q and r for any polynomial order P. Observe that as the polynomial order increases, the number of body shape functions of hexahedron increases very fast with the cubic power of P. In this way, it is very important to construct the shape functions using the tensor product of the one-dimensional functions, avoiding large memory demand.

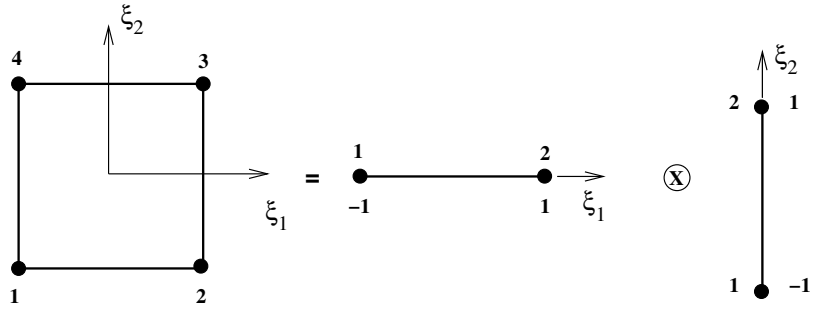


Figure 2.3. Tensor construction of square shape functions (BITTENCOURT, 2014).

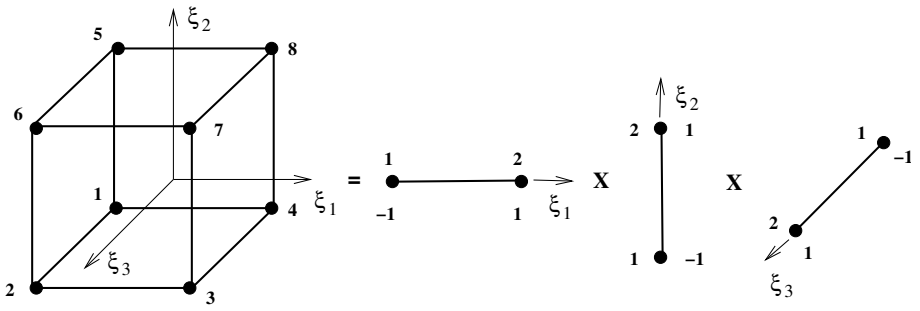
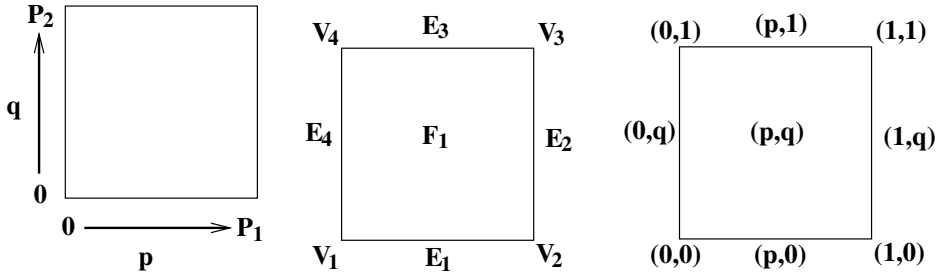


Figure 2.4. Tensor construction of shape functions for hexahedra (BITTENCOURT, 2014).

The shape functions of squares are associated with the element topological entities, which include four vertices (V_1, V_2, V_3, V_4), four edges (E_1, E_2, E_3, E_4), and one face (F_1), illustrated in Figure 2.5b. The indices p and q of Equation (2.3) are associated to the topological entities according to Figure 2.5c.

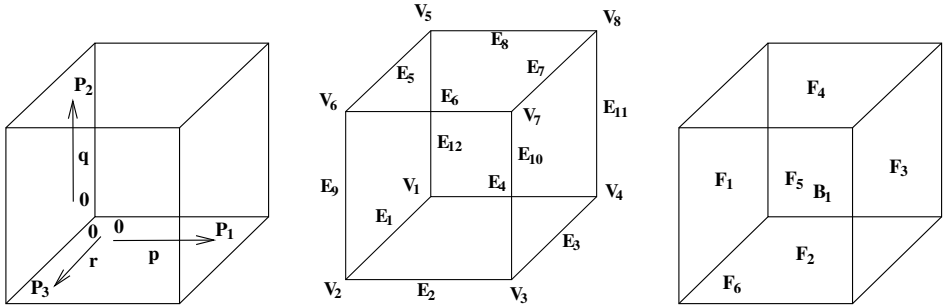


(a) Indices p and q. **(b)** Topological entities. **(c)** Entities and indices p and q.

Figure 2.5. Association between the topological entities and tensor indices p and q in the square (adapted from (BITTENCOURT *et al.*, 2007)).

Figures 2.6a illustrates the tensor indices p, q and r for the hexahedron. Figures 2.6b and 2.6c illustrate the topological entities of the hexahedron, which are constituted of eight vertices (V_1 to V_8), twelve edges (E_1 to E_{12}), six faces (F_1 to F_6), and one volume (B_1). Figure 2.7 presents the relation between indices p, q, and r and the topological entities.

When applying the FEM in the analysis of a linear elastic solid, we should interpolate, besides the displacement field, the geometry of the body by means of the coordinates of

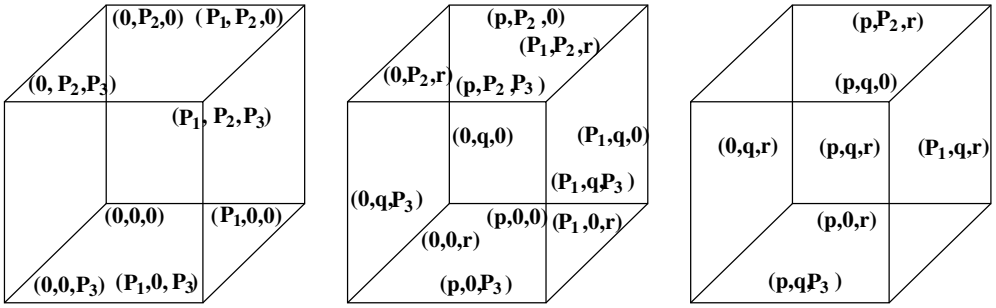


(a) Indices of p, q and r.

(b) Topological entities: vertices and edges.

(c) Topological entities: faces and volumes.

Figure 2.6. Indices p, q, and r and topological entities of the hexahedron (BITTENCOURT, 2014).



(a) Vertices.

(b) Edges.

(c) Faces and volume.

Figure 2.7. Association between indices p, q, and r and the topological entities of the hexahedron (BITTENCOURT, 2014).

the boundary points. For this purpose, we can apply the same set of shape functions used for the displacement field, defining the class of isoparametric finite elements.

Denoting x, y, and z as the coordinates of points of a finite element relative to a global reference system, we can write the following relations to interpolate the element geometry, from the nodal coordinates and interpolation functions expressed in the local system, that is,

$$\begin{aligned} x(\xi_1,\xi_2,\xi_3) &= \sum_{i=1}^{N_e} \phi_i(\xi_1,\xi_2,\xi_3)x_i, \\ y(\xi_1,\xi_2,\xi_3) &= \sum_{i=1}^{N_e} \phi_i(\xi_1,\xi_2,\xi_3)y_i, \\ z(\xi_1,\xi_2,\xi_3) &= \sum_{i=1}^{N_e} \phi_i(\xi_1,\xi_2,\xi_3)z_i, \end{aligned} \tag{2.5}$$

where N_e is the number of nodes of the element and (x_i, y_i, z_i) are the global Cartesian coordinates of node i of the element. The previous equations define the geometric mapping from the local to the global coordinate system as illustrated for a quadrangular element in Figure 2.8.

The finite element operators involve derivatives of the shape functions to the global coordinates x, y and z. As the shape functions are constructed in terms of the standard reference

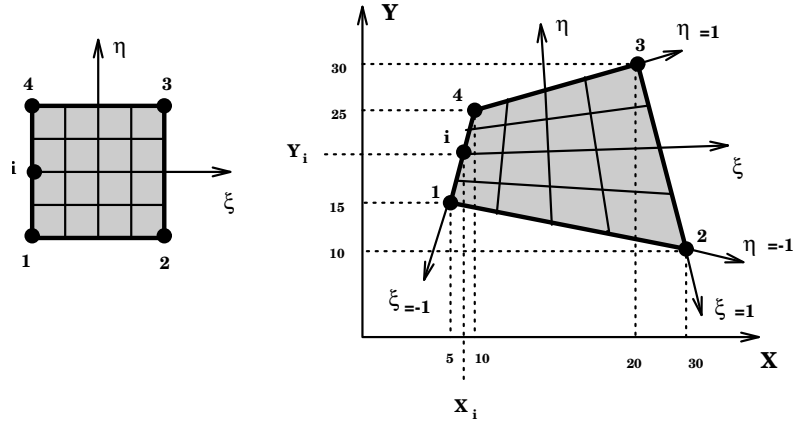


Figure 2.8. Example of transformation between the local and global reference systems using the shape functions (BITTENCOURT, 2014).

system of the element, the chain rule must be used to obtain these global derivatives.

For example, consider the component $u_x(x,y,z)$ of the displacement vector field $\mathbf{u}(x,y,z)$ expressed in terms of the local coordinates by expressions (2.5), that is,

$$u_x(x,y,z) = u_x\left(x(\xi_1,\xi_2,\xi_3), y(\xi_1,\xi_2,\xi_3), z(\xi_1,\xi_2,\xi_3)\right).$$

Using the chain rule, we can calculate the partial derivatives of the scalar function u_x relative to the local coordinates ξ_1, ξ_2 and ξ_3 as

$$\begin{aligned} u_{x,\xi_1} &= u_{x,x}x,_{\xi_1} + u_{x,y}y,_{\xi_1} + u_{x,z}z,_{\xi_1}, \\ u_{x,\xi_2} &= u_{x,x}x,_{\xi_2} + u_{x,y}y,_{\xi_2} + u_{x,z}z,_{\xi_2}, \\ u_{x,\xi_3} &= u_{x,x}x,_{\xi_3} + u_{x,y}y,_{\xi_3} + u_{x,z}z,_{\xi_3}. \end{aligned} \tag{2.6}$$

We have, in matrix notation,

$$\left\{ \begin{array}{c} u_{x,\xi_1} \\ u_{x,\xi_2} \\ u_{x,\xi_3} \end{array} \right\} = \left[\begin{array}{ccc} x,_{\xi_1} & y,_{\xi_1} & z,_{\xi_1} \\ x,_{\xi_2} & y,_{\xi_2} & z,_{\xi_2} \\ x,_{\xi_3} & y,_{\xi_3} & z,_{\xi_3} \end{array} \right] \left\{ \begin{array}{c} u_{x,x} \\ u_{x,y} \\ u_{x,z} \end{array} \right\} = [\mathbf{J}] \left\{ \begin{array}{c} u_{x,x} \\ u_{x,y} \\ u_{x,z} \end{array} \right\}, \tag{2.7}$$

where $[\mathbf{J}]$ is the Jacobian matrix of the transformation between the local and global reference systems. Inverting the Jacobian matrix, we obtain the partial derivatives of u_x relative to the global coordinates x, y, and z, that is,

$$\left\{ \begin{array}{c} u_{x,x} \\ u_{x,y} \\ u_{x,z} \end{array} \right\} = [\mathbf{J}]^{-1} \left\{ \begin{array}{c} u_{x,\xi_1} \\ u_{x,\xi_2} \\ u_{x,\xi_3} \end{array} \right\}. \tag{2.8}$$

This same procedure can be employed to obtain the partial derivatives of the element shape function ϕ_i relative to the global variables x, y, and z, as required in the expression of the stiffness matrix of elements. Thus,

$$\left\{ \begin{array}{c} \phi_{i,x} \\ \phi_{i,y} \\ \phi_{i,z} \end{array} \right\} = [\mathbf{J}]^{-1} \left\{ \begin{array}{c} \phi_{i,\xi_1} \\ \phi_{i,\xi_2} \\ \phi_{i,\xi_3} \end{array} \right\}. \tag{2.9}$$

2.2 TENSOR PRODUCT OF ONE-DIMENSIONAL MATRICES

As presented in the last section, the shape functions for square and hexahedron are constructed by the tensor product of one-dimensional shape functions. Similarly, this work implements a procedure to construct mass and stiffness matrices for two and three-dimensional elements using the tensor product of the one-dimensional elements matrices. This procedure, denominated here $D1-MATRICES$, is similar to the sum factorization presented in the literature (KARNIADAKIS; SHERWIN, 2005) and will be illustrated to the mass matrix.

The coefficients of the one-dimensional standard mass matrix of line elements, denoted as M_{ij}^{1D}, are given by

$$M_{ij}^{1D} = \int_{-1}^{1} \phi_i(\xi_1)\phi_j(\xi_1)d\xi_1. \tag{2.10}$$

The coefficients of the mass matrix for a quadrilateral are

$$M_{ij}^{2D} = \int_{-1}^{1}\int_{-1}^{1} N_i(\xi_1,\xi_2)N_j(\xi_1,\xi_2)|J|d\xi_1 d\xi_2, \tag{2.11}$$

where $|J|$ is the determinant of the Jacobian matrix of Equation (2.7).

Substituting Equation (2.3) into Equation (2.11), we obtain the coefficients of the quadrilateral mass matrices in terms of the coefficients of the one-dimensional mass matrices calculated on integration points (k,l) as

$$\begin{aligned} M_{ij}^{2D} &= \int_{-1}^{1} \phi_a(\xi_1)\phi_p(\xi_1)\left(\int_{-1}^{1}\phi_b(\xi_2)\phi_q(\xi_2)|J|d\xi_2\right)d\xi_1 \\ &= \Sigma_{k=1}^{n_1}\Sigma_{l=1}^{n_2} M_{ap}^{1D}(\xi_{1_k})M_{bq}^{1D}(\xi_{2_l})W_l W_k|J_{kl}|, \end{aligned} \tag{2.12}$$

where (n_1, n_2), (W_k, W_l) and (ξ_{1_k}, ξ_{2_l}) are the number of integration points, weights and coordinates in local directions ξ_1 and ξ_2, respectively.

For undistorted elements, the Jacobian is constant and can be factored from the integral operation. Thus, the Equation (2.12) can be rewritten as

$$\begin{aligned} M_{ij}^{2D} &= |J|\left[(\Sigma_{k=1}^{n_1} M_{ap}^{1D}(\xi_{1_k})W_k)(\Sigma_{l=1}^{n_2} M_{bq}^{1D}(\xi_{2_l})W_l)\right] \\ &= |J|M_{ap}^{1D}M_{bq}^{1D}. \end{aligned} \tag{2.13}$$

Similarly, the coefficients of the mass matrix for the hexahedron can be obtained by multiplying the coefficients of the one-dimensional matrices as

$$M_{ij}^{3D} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} N_i(\cdot)N_j(\cdot)|J|d\xi_1 d\xi_2 d\xi_3. \tag{2.14}$$

Substituting Equation (2.4) into Equation(2.14), the three-dimensional mass matrix can be written in terms of the one-dimensional mass matrices calculated on integration points

(k,l,m) as

$$\begin{aligned} M_{ij}^{3D} &= \int_{-1}^{1} \phi_a(\xi_1)\phi_p(\xi_1) \int_{-1}^{1} \phi_b(\xi_2)\phi_q(\xi_2) \\ &\quad \int_{-1}^{1} \phi_c(\xi_3)\phi_r(\xi_3)|J| d\xi_3 d\xi_2 d\xi_1 \\ &= \Sigma_{k=1}^{n_1}\Sigma_{l=1}^{n_2}\Sigma_{m=1}^{n_3} M_{ap}^{1D}(\xi_{1_k}) M_{bq}^{1D}(\xi_{2_l}) M_{cr}^{1D}(\xi_{3_m}) \\ &\quad |J_{klm}| W_m W_l W_k, \end{aligned} \tag{2.15}$$

where n_1, n_2, n_3, W_k, W_l, and W_m are the number of integration points and weights in local directions ξ_1, ξ_2, and ξ_3, respectively.

The terms of the Jacobian matrix for undistorted elements are again constant and can be factored from the integral operation. Therefore,

$$M_{ij}^{3D} = |J| \left[M_{ap}^{1D} M_{bq}^{1D} M_{cr}^{1D} \right]. \tag{2.16}$$

2.3 PROJECTION PROBLEM

The projection problem determines the approximate solution $\bar{u}$ of a function u defined in the domain Ω through a linear combination of the global basis functions $\{\Phi_i\}_{i=1}^{n}$. Therefore,

$$u \approx \bar{u} = \sum_{i=1}^{n} u_i \Phi_i, \tag{2.17}$$

where u_i are the approximation coefficients. In matrix notation,

$$\bar{u} = [N]\{u\}. \tag{2.18}$$

$[N] = [\Phi_1\ \Phi_2\ \ldots\ \Phi_n]$ is the matrix of the global shape functions.

The approximation error function is given by

$$e = u - \bar{u}. \tag{2.19}$$

Figure 2.9 gives the interpretation of the error function e as the minimum distance between the solution u and the hyperplane of dimension n defined by the interpolation functions $\{\Phi_i\}_{i=1}^{n}$.

The best approximation $\bar{u}$ for u is obtained when the error function e is orthogonal to each interpolation function ϕ_i. Therefore,

$$\int_{\Omega} e\Phi_i d\Omega = 0 \quad i = 1,\ldots,n. \tag{2.20}$$

Substituting Equation (2.19) into Equation (2.20), we have

$$\int_{\Omega} \bar{u}\Phi_i d\Omega = \int_{\Omega} u\Phi_i d\Omega. \tag{2.21}$$

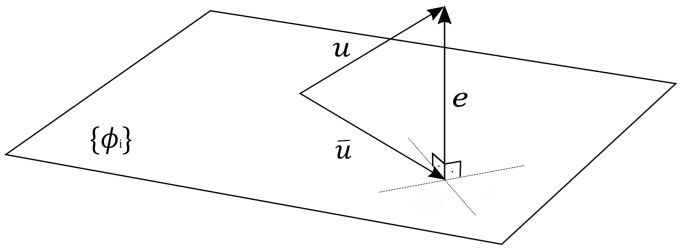


Figure 2.9. Interpretation of the error function in the projection problem.

Substituting now Equation (2.17) into Equation (2.21) results in the following expression:

$$\sum_{i=1}^{n} u_i \int_{\Omega} \Phi_i \Phi_j d\Omega = \int_{\Omega} u \Phi_j d\Omega \quad j = 1, ..., n. \tag{2.22}$$

The previous expression can be rewritten as

$$\sum_{i,j=1}^{n} M_{ij} u_j = b_j, \tag{2.23}$$

where M_{ij} are the coefficients of the mass or projection matrix given by

$$M_{ij} = \int_{\Omega} \Phi_i \Phi_j d\Omega, \tag{2.24}$$

and b_j denotes the coefficients of the body load vector

$$b_j = \int_{\Omega} u \Phi_j d\Omega. \tag{2.25}$$

The system of equations (2.23) may be expressed in matrix format as

$$[M]\{u\} = \{f\}, \tag{2.26}$$

where $[M]$, $\{u\}$ and $\{f\}$ are the global mass matrix, the global vector of unknown coefficients and the global body load vector. They are obtained by the assembling of the element contributions. For a mesh of N_{el} elements, we have

$$[M] = \bigcup_{e=1}^{N_{el}} [M_e], \tag{2.27}$$

$$\{u\} = \bigcup_{e=1}^{N_{el}} \{u_e\}, \tag{2.28}$$

$$\{f\} = \bigcup_{e=1}^{N_{el}} \{f_e\}, \tag{2.29}$$

and $[M_e]$, $\{u_e\}$ and $\{f_e\}$ are the mass matrix, the vector of unknown coefficients and the body load vector of each element e. The assembling procedure is illustrated in Figure 2.10. The

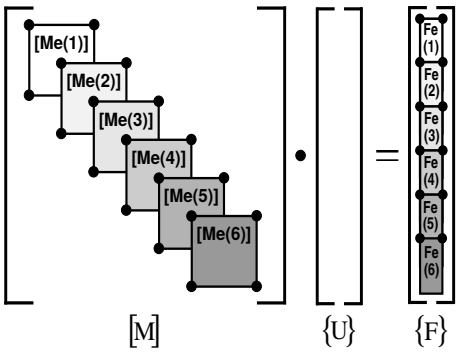


Figure 2.10. The global solution representation for the projection problem.

unknown vector $\{u\}$ is calculated using direct or iterative methods for the solution of systems of equations.

Matrix $[M_e]$ and vector $\{f_e\}$ are given in terms of the element shape functions $\{\phi_i\}_{i=1}^{N_e}$ for hexahedra as

$$[M_e] = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \rho [N_e]^T [N_e] |J_e| d\xi_1 d\xi_2 d\xi_3, \tag{2.30}$$

$$\{f_e\} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} [N_e]^T u |J_e| d\xi_1 d\xi_2 d\xi_3, \tag{2.31}$$

where ρ is the density and $[N_e] = [\phi_i\ \phi_2\ \ldots\ \phi_{N_e}]$ is the matrix of the shape functions for element e.

The error is given in terms of the L_2-error norm as

$$\|e\|_{L_2} = \sqrt{\int_\Omega (u-\bar{u})^2 d\Omega}. \tag{2.32}$$

For a mesh of N_{el} hexahedra, we have

$$\|e\|_{L_2} = \sqrt{\sum_{e=1}^{N_{el}} \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} (u|_e - \bar{u}_e)^2 |J_e| d\xi_1 d\xi_2 d\xi_3}, \tag{2.33}$$

where $u|_e$ is the restriction of the function u to element e.

2.4 ELEMENT WISE SOLVER

The element wise solution strategy is obtained from each element independently. Thus, the global matrix and body load vector are not assembled. The system of linear equations is solved locally for each element as illustrated in Figure 2.11. Next, the global solution vector $\{u\}$ is calculated through a weighted average of the local element solutions and the element measures as

$$u_i = \frac{\sum_{j=1}^{N_{ep}} u_{i,j} S_j}{\sum_{j=1}^{N_{ep}} S_j}, \tag{2.34}$$

where u_i is the global solution for degree of freedom i, $u_{i,j}$ is the local solution for degree of freedom i computed for each element j, S_j is the measure for element j and N_{ep} is the number of elements sharing degree of freedom i (BITTENCOURT; FURLAN, 2011; Y.YU *et al.*, 2014). The denominator in the previous expressions represents the sum of the measures of the elements which share the degree of freedom i. This set of elements is called patch and its measure is denoted by

$$S_p = \sum_{j=1}^{N_{ep}} S_j. \tag{2.35}$$

Therefore, Equation (2.34) may be rewritten as

$$u_i = \frac{\sum_{j=1}^{N_{ep}} u_{i,j} S_j}{S_p}, \tag{2.36}$$

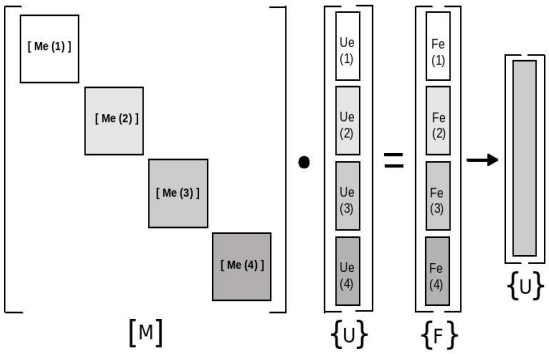
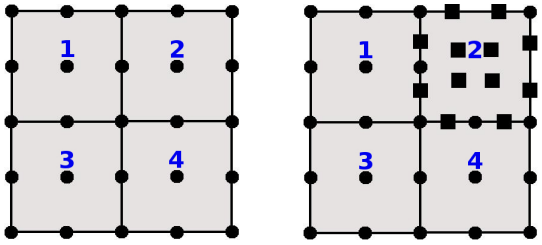


Figure 2.11. Element wise solution scheme for the projection problem.

2.5 *p*-NON-UNIFORM PROCEDURE

In this section, we propose a *p*-non-uniform strategy for the HO-FEM using the local method presented in the previous section. This approach allows different polynomial orders for the mesh elements, as shown in Figure 2.12 for a square mesh with four elements.

Figure 2.12(a) illustrates a *p*-uniform mesh with interpolation order two for all elements. Figure 2.12(b) presents the same mesh with the non-uniform polynomial orders 2, 3, 2, 2



(a) *p*-Uniform mesh **(b)** *p*-Non-uniform mesh

Figure 2.12. Mesh of quadratic elements with uniform and non-uniform polynomial order distribution.

for elements 1, 2, 3, and 4, respectively. The solution on edges with different orders are mapped to a common order from the element solutions. For example, the local solution of element 2 on the shared edge with element 1 is interpolated to order 2 before the calculation of the global coefficients given by Equation (2.36).

Higher interpolation orders can be used only to those elements with high solution gradients identified by an error estimator, not use in this work. Consequently, the total number of variables in the system of equations is reduced. In this study, the L_2-error norm calculated using the p-uniform and p-non-uniform strategies will be considered for projection solver in Section 6.2.

The local methods were implemented using different strategies to solve the linear system of equations. The strategies are chosen according to mesh type and polynomial order distribution. For example, for uniform mesh and p-uniform distribution, the mass and Jacobian matrices are the same for all elements and need to be calculated just for one element. Consenquently, memory demand is reduced as shown in Section 6.3.

These strategies also defined the manner to parallelize the linear and non-linear transient solvers discussed in Sections 2.6 and 2.7. These solvers use two-different approaches with multi-threaded parallelism based on the construction of lumped or consistent mass matrices discussed in Section 2.6.

2.6 EXPLICIT FINITE ELEMENTS IN LINEAR TRANSIENT ANALYSIS

The discrete linear dynamic equilibrium equation for a mechanical component is given by

$$[M]\{\ddot{u}^t\} + [C]\{\dot{u}^t\} + [K]\{u^t\} = \{f^t\}, \tag{2.37}$$

where $[M]$, $[C]$ and $[K]$ are the global mass, damping and stiffness matrices; $\{\ddot{u}^t\}$, $\{\dot{u}^t\}$ and $\{u^t\}$ are the global acceleration, velocity and displacement vectors of the degrees of freedoms at time t; $\{f^t\}$ is the global load vector at time t applied to the degrees of freedom of the finite element model.

Given initial conditions for the displacement and velocity vectors at t_0, the solution of Equation (2.37) consists in calculating the displacement vector $\{u^t\}$ for specific time instants using explicit and implicit time integration methods. In this work, we will consider the explicit central difference method (BATHE, 1996).

Consider the time interval $[t_0, t_f]$ discretized in N time steps of length Δt, i.e., t_0, $t_0 + \Delta t$, $t_0 + 2\Delta t, \ldots, t_0 + N\Delta t$. The following central difference approximation is applied to the acceleration vector at the time step $t_n = t_0 + n\Delta t$:

$$\{\ddot{u}^{(n)}\} = \frac{1}{\Delta t^2}(\{u^{(n-1)}\} - 2\{u^{(n)}\} + \{u^{(n+1)}\}). \tag{2.38}$$

The velocity vector also is approximated by the central difference scheme as

$$\{\dot{u}^{(n)}\} = \frac{1}{2\Delta t}(\{u^{(n+1)}\} - \{u^{(n-1)}\}). \tag{2.39}$$

Replacing Equations (2.38) and (2.39) in (2.37), we obtain the system of equations that gives the displacement vector at time step t_{n+1} as follow:

$$[\hat{M}]\{u^{(n+1)}\} = \{\hat{f}^{(n)}\}, \tag{2.40}$$

where,

$$[\hat{M}] = a_0[M] + a_1[C], \tag{2.41}$$

$$[\hat{f}^{(n)}] = \{f^{(n)}\} - ([K] - a_2[M])\{u^{(n)}\} - (a_0[M] - a_1[C])\{u^{(n-1)}\}. \tag{2.42}$$

The constants in the previous equations are $a_0 = \frac{1}{\Delta t^2}$, $a_1 = \frac{1}{2\Delta t}$ and $a_2 = 2a_0$. Note that the displacement vector $\{u^{(n+1)}\}$ depends only on the terms calculated at the previous time steps. For this reason, the method is known as explicit. Neglecting the damping effect, Equations (2.41) and (2.42) reduces to

$$[\hat{M}] = a_0[M], \tag{2.43}$$

$$[\hat{f}^{(n)}] = \{f^{(n)}\} - ([K] - a_2[M])\{u^{(n)}\} - a_0[M]\{u^{(n-1)}\}. \tag{2.44}$$

Given the initial conditions at time step t_0 for the displacement $\{u^{(0)}\}$ and velocity $\{\dot{u}^{(0)}\}$ vectors, the initial acceleration $\{\ddot{u}^{(0)}\}$ is calculated from Equation (2.37) as

$$\{\ddot{u}^{(0)}\} = [M]^{-1}(\{f^{(0)}\} - [K]\{u^{(0)}\}). \tag{2.45}$$

Equation (2.44) requires the vector $\{u^{(-1)}\}$ at time step t_0 ($n = 0$), which is calculated combining Equations (2.38) and (2.39), obtaining

$$\{u^{(-1)}\} = \{u^{(0)}\} - \Delta t\{\dot{u}^{(0)}\} + \frac{\Delta t^2}{2}\{\ddot{u}^{(0)}\}. \tag{2.46}$$

The element wise version for the explicit central difference method solves the system of equations given in (2.40) for each element e of the finite element mesh. Therefore,

$$[\hat{M}_e]\{u_e^{(n+1)}\} = \{\hat{f}_e^{(n)}\}. \tag{2.47}$$

Neglecting the damping effect, we obtain the following equations:

$$[\hat{M}_e] = a_0[M_e], \tag{2.48}$$

$$\{\hat{f}_e{}^{(n)}\} = \{f_e^{(n)}\} - [K_e]\{u_e^{(n)}\} + (a_2 - a_0)[M_e](\{u_e^{(n)}\} - \{u_e^{(n-1)}\}). \tag{2.49}$$

The velocity and acceleration vectors are calculated for each element using the expressions

$$\{\ddot{u}_e^{(n)}\} = a_0(\{u_e^{(n-1)}\} - 2\{u_e^{(n)}\} + \{u_e^{(n+1)}\}), \tag{2.50}$$

$$\{\dot{u}_e^{(n)}\} = a_1(\{u_e^{(n+1)}\} - \{u_e^{(n-1)}\}). \tag{2.51}$$

The global displacement vector at time step t_{n+1} and the velocity and acceleration vectors at time step t_n are calculated from Equations (2.50) and (2.51) similarly to ((2.36) as follows:

$$\{u^{(n+1)}\} = \bigcup_{e=1}^{Nel} \{u_e^{(n+1)}\}\frac{S_e}{S_p}, \tag{2.52}$$

$$\{\dot{u}^{(n)}\} = \bigcup_{e=1}^{Nel} \{\dot{u}_e^{(n)}\}\frac{S_e}{S_p}, \tag{2.53}$$

$$\{\ddot{u}^{(n)}\} = \bigcup_{e=1}^{Nel} \{\ddot{u}_e^{(n)}\}\frac{S_e}{S_p}. \tag{2.54}$$

The vector of equivalent forces on the element e, $\{f_e^{(n)}\}$, in Equation (2.49) is obtained from the respective global vector $\{f^{(n)}\}$ using

$$\{f_e^{(n)}\} \leftarrow \{f^{(n)}\}\frac{S_e}{S_p}. \tag{2.55}$$

Considering that the global vectors are obtained from Equations (2.52), (2.53) and (2.54), the corresponding element vectors are updated using the numbering of the degrees of freedom for each element such that

$$\{u_e^{(n+1)}\} \leftarrow \{u^{(n+1)}\}, \tag{2.56}$$

$$\{\dot{u}_e^{(n)}\} \leftarrow \{\dot{u}^{(n)}\}, \tag{2.57}$$

$$\{\ddot{u}_e^{(n)}\} \leftarrow \{\ddot{u}^{(n)}\}. \tag{2.58}$$

The internal load vector obtained from the global vectors calculated in (2.52) to (2.50) does not reach the equilibrium with the applied external loads and the difference will generate a residue vector. The internal load vector for element e is calculated as

$$\{f_{i,e}^{(n+1)}\} = [M_e]\{\ddot{u}_e^{(n)}\} + [K_e]\{u_e^{(n)}\}. \tag{2.59}$$

The following global vector of internal loads is obtained using the assembling procedure:

$$\{f_i^{(n+1)}\} = \bigcup_{e=1}^{N_{el}} \{f_{i,e}^{(n+1)}\}. \tag{2.60}$$

However, this vector does not balance the external force vector $\{f^{(n+1)}\}$ for the next time step t_{n+1}. In this way, the following residue vector is calculated:

$$\{r\} = \{f^{(n+1)}\} - \{f_i^{(n+1)}\}, \tag{2.61}$$

and assigned to the element load vectors to the subsequent time step as

$$\{f_e^{(n+1)}\} \leftarrow \{r\}\frac{S_e}{S_p}. \tag{2.62}$$

The procedure presented here is repeated for each time step until reaching the final time t_f.

When using nodal basis, the consistent element and global mass matrices may be replaced by the respective spectral or lumped mass matrices which are diagonal. This makes the solution of systems of equations (2.40) and (2.47) trivial.

2.7 EXPLICIT FINITE ELEMENTS FOR NON-LINEAR TRANSIENT ANALYSIS

Consider the discrete equation of motion for a nonlinear structural problem, involving large displacements and deformations and neglecting damping, given by

$$[M]\{\ddot{u}^t\} = \{f^t\} - \{f_i^t\}, \tag{2.63}$$

where $\{f_i^t\}$ is the vector of the internal nodal loads at time t, which contains all the nonlinearities of the problem.

In the explicit nonlinear algorithm, the acceleration $\{\ddot{u}^{(n)}\}$ and velocity $\{\dot{u}^{(n)}\}$ vectors at time step t_n are approximated by (2.38) and (2.39), respectively. Replacing these expressions in Equation (2.63), we obtain the same expression as for the linear explicit method given in (2.40). However, the vector $\{\hat{f}^{(n)}\}$ depends now on the vector $\{f_i^{(n)}\}$ such that

$$\{\hat{f}^{(n)}\} = \{f^{(n)}\} - \{f_i^{(n)}\} + [M]\left(a_2\{u^{(n)}\} - a_0\{u^{(n-1)}\}\right). \tag{2.64}$$

The displacement vector $\{u^{(n+1)}\}$ is obtained by solving the system of Equations (2.40) stated here as

$$\{u^{(n+1)}\} = [\hat{M}]^{-1}\{\hat{f}^{(n)}\}. \tag{2.65}$$

The constants used in the previous expressions are the same of the linear explicit method.

Given the initial conditions at $t = t_0$ $(n = 0)$ for the displacement $\{u^0\}$ and velocity $\{\dot{u}^0\}$ vectors, the initial acceleration is calculated from (2.63) as

$$\{\ddot{u}^{(0)}\} = [M]^{-1}(\{f^{(0)}\} - \{f_i^{(0)}\}). \tag{2.66}$$

Note that Equation (2.64) also requires of the displacement vector $\{u^{-(1)}\}$ for $n = 0$, which is obtained in the same way as for the linear explicit method using Equation (2.46).

The local version of the previous procedure is discussed below. For any time step n, the external load vector $\{f_e^n\}$ for element e is obtained from the external global load vector $\{f^n\}$ minus the vector of the internal forces $\{f_i^n\}$ using the smoothing procedure. Therefore.

$$\{f_e^t\} \leftarrow (\{f^t\} - \{f_i^t\})\frac{S_e}{S_p}. \tag{2.67}$$

The previous equation means that the coefficients of the global loading vector $\{f^t\} - \{f_i^t\}$, which correspond to the degrees of freedom of the element, are multiplied by $\frac{S_e}{S_p}$ and assigned to the element load vector $\{f_e^t\}$.

The global vectors of the initial conditions can be assigned to the vectors of the element again using the numbering of degrees of freedom. The displacement vector $\{u_e^{(-1)}\}$ can be calculated for each element using an expression similar to (2.46), but using the element vectors. Using the smoothing procedure, the global vector is determined.

For each time step, Equation (2.40) is solved for each element. The effective loading vector for each element e is given by

$$\{\hat{f}_e^t\} = \{f_e^t\} - \{f_{i,e}^t\} + [M_e](a_2\{u_e^t\} - a_0\{u_e^{(n-1)}\}). \tag{2.68}$$

From the element solution vectors, the displacement, velocity and acceleration global vectors are obtained by the smoothing procedure using the element measurements, respectively, by (2.52), (2.53) and (2.54).

As in the linear method, the coefficients of the previous global vectors for homogeneous boundary conditions are assigned zero value. Once that the global vectors are obtained, the element vector of displacement $\{u_e^{(n+1)}\}$, velocity $\{\dot{u}_e^{(n)}\}$ and acceleration $\{\ddot{u}_e^{(n)}\}$ are obtained using (2.56), (2.57) and (2.58), respectively.

The equivalent nodal inertia force vector for each element is calculated as

$$\{f_{inertia,e}^{(n)}\} = [M_e]\{\ddot{u}_e^{(n)}\}. \tag{2.69}$$

It can be seen that the acceleration vector $\{\ddot{u}_e^{(n)}\}$ is available at time step n. Thus, the global inertia load vector is assembled from the element loading vector $\{f_{inertia,e}^{(n)}\}$ as

$$\{f_{inertia}^{(n)}\} = \bigcup_{e=1}^{Nel} \{f_{inertia,e}^{(n)}\}. \tag{2.70}$$

The external $\{f^{(n+1)}\}$ and internal $\{f_i^{(n+1)}\}$ global load vectors are calculated at time t_{n+1} and the following residue vector is obtained

$$\{r\} = \{f^{(n+1)}\} - \{f_i^{(n+1)}\} - \{f_{inertia}^{(n)}\}. \tag{2.71}$$

Analogously to the linear method, the global residue vector is assigned back to the degrees of freedom of each element and used in the next time step t_{n+1}. Therefore,

$$\{f_e^{(n+1)}\} \leftarrow \{r\}\frac{S_e}{S_p}. \tag{2.72}$$

This procedure is repeated until the final time t_f.

The element wise methods were implemented in $(hp)^2$FEM with distributed and shared-memory parallelism. These strategies have the advantage of solving local systems of equations with order given by the number of element unknowns which are much smaller than the number of global unknowns. These features allowed for flexibility to implement the hybrid parallel methods using multi-threads to construct the mass and stiffness matrices and distributed computing to solve different mesh subdomains.

3 BASIC CONCEPTS ON HIGH PERFORMANCE COMPUTING

The term High-Performance Computing can be used to describe computers with a high level of processing power that works in parallel to solve complex applications (QUINTERO, 2014). However, there are many aspects that do not strictly address this definition of high-performance computing. For example, the use of clusters, distributed memory programming, shared memory, optimized libraries as BLAS and LAPACK to linear algebra, and solution of the linear system of equations are also important. Another aspect is the hardware architecture knowledge, e.g., the proper use of memory hierarchy to improve access to available resources, avoiding cache misses and page faults.

A parallel computing environment is made up of hardware and software resources, as well as their interactions. Hardware resources are made up of personal computers, workstations, and clusters. Clusters must use fast communication devices to achieve high-performance parallel computing since there may be high latency on the network, causing low-performance (GUEDES, 2009).

The main goal of parallel application development is to reduce job execution time by dividing a problem into many parts. The division creates tasks that will be executed in many processors, aiming to speedup and making the best use of memory resources. In general terms, the problem's size, solution approaches, algorithm, and application features determine how the resources provided, and the allocated processes should be used in a parallel computing environment.

The performance gain in parallel applications comes with difficulties by combining computer architecture and algorithms. Examples are the deterministic loss of the solution precision between the sequential and parallel programs and the reordering of resources due to network behavior. Besides that, there could be deadlock events, mainly in scenarios of synchronized communication. Deadlock happens when one or more processes wait for another process resource to be available, and that process is also waiting simultaneously for the availability of resources allocated to another process that invoked it.

It is important to develop high-performance applications that minimize the cost of algorithm coding. Consequently, right after bug fixing, distribution, and communication aspects of parallel software, the next step is to analyze and improve program performance. Besides, the type of communication among processes and the main metrics of parallel programs are also relevant, as shown in Chapter 6.

3.1 FUNDAMENTALS OF PARALLEL COMPUTING AND STRATEGIES

The implementation of parallel software may follow the explicit and implicit methodologies.

In the explicit parallelism methodology, the programmer accounts for the following aspects:

- identify the tasks that can be executed in parallel,
- assign the tasks to the processors,
- control the flow of execution, indicating the set of synchronizing points, and
- knowledge about the hardware architecture to develop dedicated code to improve performance (for example, increase local computing load and decrease message exchanging among processors).

In the implicit parallelism methodology, the parallel program's main functionalities, as controlling and synchronizing, could be inferred by the compiler. It also detects the potential of parallelism and assignment of the respective tasks to parallel threads for execution.

The explicit and implicit implementation are outlined following the steps of the Foster's methodology to develop a parallel program (SCHMIDT *et al.*, 2017). This methodology establishes a problem's parallelization in the following steps: partitioning or decomposition, communication or synchronization, agglomeration, and mapping that will be detailed below.

Explicit and implicit parallelism methodologies may be combined. This combination should use the best advantages of these methodologies, such as defining which parts of the parallel program should have more automatic or programmer control. In one of the combined cases, the programmer can identify the tasks parallelized, but the decomposition, communication, and mapping are automatic. Another way is to have the decomposition defined explicitly, but implicit communication and mapping. We can also have only automatic communication, with the task decomposition and assignment to processors explicitly. Finally, the programmer may have more control of communication among processes, but the data synchronization or updating processes would be automatic.

The parallel decomposition algorithms are classified as domain or functional. For the domain decomposition, data are divided and distributed to processors that perform the same task. Functional decomposition occurs when tasks are divided before data. Consequently, the size of the tasks on the sub-domain defines the granularity of parallelism, which could be fine-grained, medium-grained, or coarse-grained (HAGER; WELLEIN, 2010).

In parallel applications, the communication by processors topology has some specific characteristics and may be structured and non-structured. Structured communication may

use a tree or network, while non-structured communication uses, for instance, an arbitrary graph. Besides that, communication is static when a set of processors has a pre-established transmission topology. Dynamic communication occurs when data transfer in a set of processors changes during the parallel execution. Other characteristics of parallel communication are:

- ▷ Scope:
  - Global - communication among all processors.
  - Local - communication just established among neighbors' processors.
- ▷ Synchronism:
  - Synchronous - the tasks are executed in a coordinated and/or synchronized manner.
  - Asynchronous - the tasks are executed independently. Each processor does not need to wait for each other for data transferring.

Agglomeration is defined by grouping two or more tasks into larger ones allowing to decrease the communication cost by increasing the computational granularity, thus obtaining better reuse of parts of the serial program. Agglomeration also allows decreasing the number of communications performed due to grouping tasks with small communication tasks.

Mapping is the last step of the Foster's methodology and where task attribution to processors occurs, aiming to increasing their usage and decreasing communication among them. Optimum percentage use of processors is obtained through load balancing, such that processors will end execution simultaneously for the same number of tasks. In summary, concurrent tasks should be allocated on different processors and more frequent communication tasks in the same processor.

Based on these characteristics, some aspects should be considered to avoid any performance degradation in parallel programs:

- The granularity of decomposition - the number and size of tasks determine the cost of parallel execution.
- Load balance - all processors should keep busy during the parallel execution.
- Synchronization delay - data exchange among several processors, could cause problems with memory contention.
- Communication delay - use of distributed memory makes communication naturally slower.

3.2 DISTRIBUTED MEMORY MULTIPROCESSOR SYSTEMS

Distributed computation is when a program runs on machines consisting of multiple computers, and each computer has exclusive access to its local memory. Programs must exchange messages for communication and synchronization among processes. Also, data granularity is coarse when using domain decomposition in multicomputer systems.

In this context, one software that enables this type of programming is the Message Passing Interface (MPI). The MPI is a message passing library standard designed by academics, software library developers, researchers, and users to function on various parallel computing architectures. This library specifies names, calling sequences, and results of subroutines or functions for many users writing portable message-passing programs in C/C++ and Fortran (GROPP EWING LUSK, 2014). As a result, any application using MPI is considered a set of processes in the computing environment. One of the main advantages of MPI is portability across different machines. However, parallel programming development is explicit; that is, the programmer needs to identify parallelism regions to use MPI routines.

One type of communication used with MPI is peer-to-peer, i.e., message exchanging only between two processes. The respective routines can be blockers, which only return from the call after some events occur, such as waiting for a given data set to be delivered. Examples of blocking routines used in this work are MPI_Send and MPI_Recv for sending and receiving messages, respectively. On the other hand, non-blocking communication allows the execution to continue without waiting for any events that determine the routine execution completion.

Another form of communication is collective, which involves all processes within the same process group. This is often used to handle common data set by the processes of a group. Collective communication routines are defined by blocking peer-to-peer communication routines. The main communication routines used in this work were MPI_Allgather, which allows all processes to send information to and collect information from each process; MPI_Allgatherv, a variation of the previous routine, allowing both sent and collected information of different sizes; MPI_Allreduce, which combines the results of all processes according to the defined MPI operation type (MPI_Op) and returns the result to all processes; MPI_Scan, which performs partial reductions of data from a set of processes; and MPI_Barrier, which performs synchronization of all processes in a group (GROPP EWING LUSK, 2014).

3.3 SHARED MEMORY AND MULTITHREADED PROCESSORS

The concept of parallelism can be given in terms of threads or processes. A thread can be defined as a program control line flow. Threads share the same memory allocation and may have their own resources during program execution, such as program counter or execution stack to allocate local variables. Threads can also communicate with each other through access to the same memory allocation and synchronization through mutual exclusion mechanisms.

Processes have the same characteristics as threads, but they use a private memory allocation. Inter-process communication is accomplished through message exchange, as mentioned in the previous section by using MPI (GROPP EWING LUSK, 2014). Figure 3.1 presents a comparison between threads and processes, where one or more threads can be within the same process.

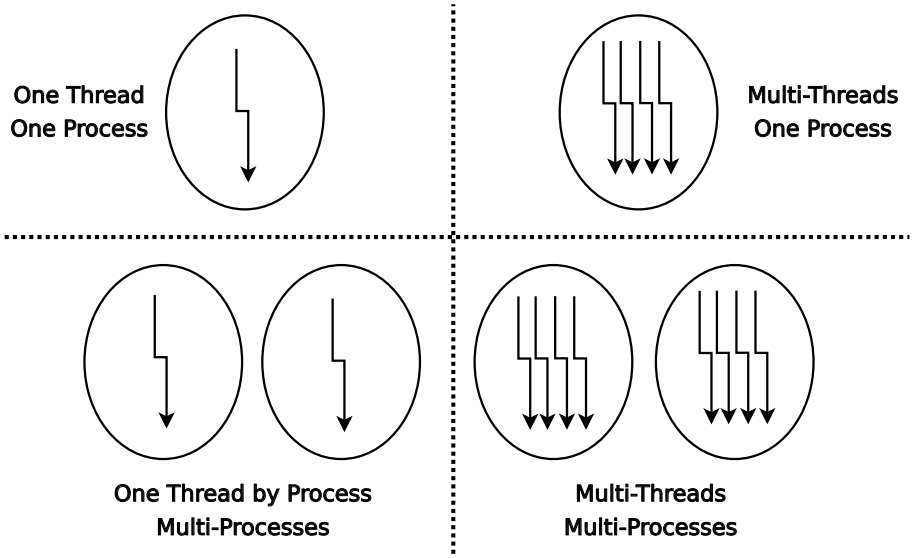


Figure 3.1. Composition of threads and processes.

The parallelism implementation in the shared-memory systems may be applied by coordinating two or more threads. Among the interfaces for multi-thread programming, we chose OpenMP (BOARD., 2013), which is portable and has three main elements: environment variables, execution libraries, and building directives. OpenMP is maintained by a group of hardware and software companies and available for C/C++ and Fortran languages.

OpenMP parallel programming is explicit and allows full control of the code. The directives always start with "#pragma omp" in C/C++, as presented in Chapter 5. These directives allow defining parallel regions within the code, set the permissions of variables in the parallel region, define threads' behavior, and apply operations on them.

3.4 PARALLEL SCALABILITY PARAMETERS AND MEASURES

Consider a parallel application running with p processes, which can be a combination of cores and/or nodes in a system, such as the IBM Blue Gene described in Chapter 6. This application's expected optimal performance is p times faster than the time of the respective serial application. This result is known as linear speedup. For example, if the execution times of the parallel and serial programs are, respectively, T_p and T_s, the ideal execution time of the parallel program is $T_p = T_s/p$ (PACHECO, 2011).

Due to several factors, linear speedup is not easy to achieve. One of the possible overheads is implementing mechanisms such as semaphores to avoid concurrency problems

among processes (KLEIN *et al.*, 2003). Such problems are common in parallel programs that share the same memory allocation, also known as the critical section. Another factor is the overhead time for data transmission by processes, which can be much longer than local memory access time. In general, the greater the number of processes p, the larger the communication overhead increase. Besides, the tasks of partitioning and data distribution should also be considered.

The Speedup S is defined by the ratio of the run times of the serial (T_s) and parallel (T_p with p processors). Therefore,

$$S = \frac{T_s}{T_p}. \tag{3.1}$$

T_s can be determined by executing: a) the parallel program in one core of the multi-core machine, b) the serial program in a parallel machine process, c) the best serial program algorithm in a parallel machine process, or, d) the best serial program algorithm in a standard machine. In this work, we used option c), thus avoiding the overheads of communication, partitioning, and data distribution. Overheads in the parallel applications may occur even considering the execution with one process. The speedup measured in this case is called absolute speedup. Option a) is known as the relative speedup.

Another cost metric used is the efficiency of E obtained by the ratio of speedup to the number of processors. Therefore,

$$E = \frac{T_s}{pT_p} = \frac{S}{p}. \tag{3.2}$$

This metric is the normalized speedup measure representing the percentage value of theoretical or ideal speed achieved (MASUERO, 2009; FERREIRA, 2006). In summary, E, S and T_p depend on the number of processes p. On the other hand, problem's size influences the results obtained for E, S, T_p, and T_s.

Efficiency achieves the maximum value of 1 and tends to 0 as the number of processes increases, considering a fixed-size problem (DUNN, 2003). The primary metrics S and E define how much faster the parallel code is compared to serial code, considering the ideal speedup (PACHECO, 2011).

3.4.1 Strong scalability

Since some processing must be done synchronously, not all parts of the code can be parallelized. The total execution time of a program is the sum of serial and parallel portions. In this context, the parallel run-time can be defined according to the Amdahl law as (AMDAHL, 1967)

$$T_p = (1-P)\,T_s + \left(\frac{P}{N}\right) T_s, \tag{3.3}$$

where T_p is the parallel time for a specific number of processors N, P, and $(1-P)$ are the parallel and serial percentages of the code, and T_s is the serial time. If $N = 1$, then the serial and

parallel times are the same. The speedup can also be obtained as

$$S_p = \frac{1}{(1-P) + \frac{P}{N}} \tag{3.4}$$

Amdahl's work found an upper limit of speedup in parallel programs from Equation (3.4). The parallel part (P/N) will be zero by increasing the number of processors to infinity. Thus, the maximum speedup can be calculated by $1/(1-P)$. If the serial portion of the algorithm is too large, then the speedup will be 1. However, if this part is too small, then the speedup will be constrained by the serial percentage $(1-P)$ (ROSÁRIO, 2012).

Therefore, Amdahl's law shows that the performance range depends on limiting the serial part of the program because the parallel portion can only be optimized by increasing the number of parallel machines by multi-threaded or multi-process, i.e., it is not possible to obtain an infinite gain just by increasing the number of processors.

Strong scalability emerged through Amdahl's law. A parallel program is scalable if the calculated efficiency is maintained, given a fixed problem size and increasing the number of processing units. Thus, scaling is defined as strong if performance increases as the number of processors also increase proportionally.

3.4.2 Weak scalability

Contrary to Amdahl's law, Gustafson (1988) found out that parallelism can also increase according to the problem size for many engineering problems. Also, it was observed that a rather small increase occurs in the serial piece compared to the parallel portion. In this context, weak scalability, also known as Gustafson's law, states that a correct way to evaluate scalability is by increasing problem size and number of processors simultaneously.

In this way, poor scalability is evaluated by increasing the problem's size in proportion to the number of processors or threads. Consequently, parallel programs will be considered weakly scalable when efficiency is maintained by increasing these two factors (problem size and number of processes) in the same proportion (KAMINSKY, 2016).

Weak scalability can be measured by the quantity $Sizeup(N,P)$ similarly to the speedup of Equation 3.1, but considering the increase in the problem size and the computation ratio simultaneously. It is determined by multiplying the problem size proportion and the serial and parallel execution time ratio. For a problem size N and P processors, it is defined by the following expression:

$$Sizeup(N,P) = \frac{N(P)}{N(1)} \frac{T_s(N(1),1)}{T_p(N(P),P)}, \tag{3.5}$$

where $N(P)$ is the problem size for P parallel processes; $N(1)$ is the problem size for serial run; $T_s(N(1),1)$ and $T_p(N(P),P)$ are the serial and parallel execution times, respectively. If the problem size of serial and parallel applications are the same, that is, $N(P) = N(1)$, the *Sizeup* metric will be equal to the speedup of Equation (3.1).

The efficiency metric will determine how close Sizeup is to the ideal value. In this sense, the ideal will be linear and occur when parallel and serial executions are performed simultaneously. In that case, the parallel program running in P processors must have the problem size P times the problem size with 1 process. Efficiency in weak scalability is defined by the ratio between Sizeup and the number of processors, that is,

$$E(N,P) = \frac{Sizeup(N,P)}{P}, \tag{3.6}$$

Analogously to the Equation 3.2, efficiency may have values in the range $[0,1]$. For the ideal case, efficiency will be 1; otherwise, the efficiency will be less than 1 (KAMINSKY, 2016).

4 $(HP)^2$FEM ARCHITECTURE MODEL

The architecture of the $(hp)^2$FEM software was designed to allow flexibility and generalization, facilitating the use, maintenance, and creation of new classes.

A software architecture defines a set of structures needed to describe the system and collaborate to make high-level decisions. It performs the role of linking the requirement and process phases (CLEMENTS *et al.*, 2010; GARLAN; SHAW, 1994).

A software architecture can be developed using top-down and/or bottom-up strategies. In the former, the system is recursively decomposed in modules or functions, until they are recognized as being easily implementable. The top-down strategy describes the state of a centralized and shared system with active functions. We developed the $(hp)^2$FEM software using the bottom-up approach; the system is viewed as a set of blocks and its state is decentralized among the objects. In this way, each object operates on its own state (JALOTE, 2012).

4.1 INFRASTRUCTURE AND INTERFACES TO OTHER LIBRARIES

In addition to the development strategies, we used tools to manage each software module independently in the phases of modeling, implementation, error handling, memory leak checking and tuning.

4.1.1 Modeling and documentation

The Unified Modeling Language (UML) diagrams and the program documentation were developed using the Metamill tool, a multiplatform software for UML 2.3 (FERNANDES, 2011). The class documentation was generated by the Doxygen software. These tools, as well as UML, supported the development process, simplifying the generation and visualization of the source code.

The modeling, implementation, and documentation steps were developed concurrently using the bottom-up approach. All documentation is available at www.hp2fem.org.

4.1.2 Symbolic-numeric analyzer

We developed a symbolic-numeric analyzer that allows users to describe boundary conditions and loads and other model attributes using symbolic expressions or functions in terms of spatial coordinates (X, Y, Z) and time T. This feature allows the validation of solvers in the $(hp)^2$FEM architecture using manufactured solutions. The symbolic-numeric analyzer was implemented using lex and yacc, which are open-source libraries for creating lexical and syntactic analyzers (LEVINE, 2009).

4.2 KEY FEATURES

$(hp)^2$FEM software was originally developed for structural mechanics, but it is also possible to use the basic infrastructure to implement solvers for other problems. It has the following features:

- Nodal and modal tensor bases for low- and high-order approximations (BITTENCOURT *et al.*, 2007; KARNIADAKIS; SHERWIN, 2005).
- Tensor-based Gauss–Jacobi, Gauss–Radau–Jacobi and Gauss–Lobato–Jacobi quadrature rules (BITTENCOURT; VAZQUEZ, 2009; KARNIADAKIS; SHERWIN, 2005).
- Different data structures for symmetric and sparse matrices. These structures allocate memory and operate on one-dimensional arrays.
- Storage of the relation between the geometry represented by non-uniform rational basis spline (NURBS) and the finite element mesh.
- Different material models through generalization, which makes simpler to add specific new material models when required.
- Builds the nodal incidences and coordinates for the high-order elements with planar and curved faces.
- *p*-non-uniform polynomial order distribution, i.e., different polynomial orders for finite elements.
- Use of element groups for the global mesh, allowing different element shapes in the same mesh.
- Use of tensor product of one-dimensional operators to calculate the shape functions and operators for two and three dimensions.
- Converts symbolic functions to numeric values using a symbolic-numeric analyzer.
- Particular efficient solvers for distorted and non-distorted meshes.
- Different meshes for the input data, solution, mapping, and post-processing with different polynomial orders. The input mesh is for elements with straight edges and planar faces. The boundary elements may have curved edges and faces. Using the definition of curves and surfaces represented by NURBS, it is possible to generate high-order nodes for the boundary elements. Generally, the mapping mesh has an interpolation order that is smaller than the solution mesh. In addition, the mapping mesh uses shape functions based on Lagrange polynomials. The post-processing mesh contains a fixed polynomial order to generate the results.

4.3 CLASS MODEL AND DATA FLOW EXECUTION

The main packages of the $(hp)^2$FEM architecture are depicted in Figure 4.1 and described in the next sections.

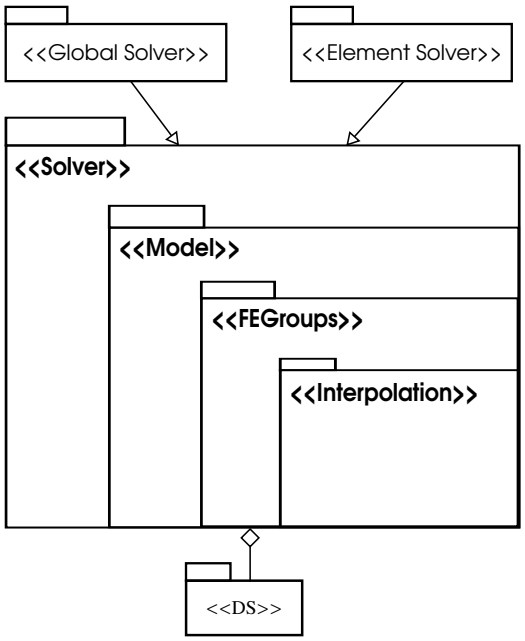


Figure 4.1. Main packages of the $(hp)^2$FEM architecture.

4.3.1 DS - Data Structure package

The classes of the DS package contain various matrix types, such as symmetric and sparse matrices (BITTENCOURT, 2000). The package performs matrix and vector operations and implements direct and iterative methods for solving linear system of equations. The data structures of the matrices use one-dimensional arrays allocated using aligned memory for improved performance.

DS uses C++ templates to implement classes of arrays and tables (BITTENCOURT, 2000). This C++ feature allows the class methods and parameters to work with many different data types (VANDEVOORDE; JOSUTTIS, 2003). The most frequently used classes with templates in $(hp)^2$FEM are **OneIndexTable** and **TwoIndexTable**.

These data structures are important because of their flexibility to store data of finite element models. The **OneIndexTable** and **TwoIndexTable** classes are used to store multiple data as the interpolation function values on the integration points for all different polynomial orders given in the input files.

The data structure of the **TwoIndexTable** class is illustrated in Figure 4.2. This class is organized by three one-dimensional arrays (one for data and the other two for indices), which are used for indexing ranges of elements of the data array and dividing them into groups and subgroups of values. For instance, consider the storage of shape function values on the integra-

tion points. The shape functions are stored in the data array of the **TwoIndexTable** instance and separated in subgroups of functions according to the polynomial and integration orders. Consequently, the shape functions for all polynomial orders used in a given problem can be accessed for all integration orders, or accessed only for a given polynomial and integration order.

The DS package classes are used extensively in other packages. They implement methods for the allocation and deallocation of data, memory management, and linear algebra with BLAS and LAPACK (DONGARRA, 2003). These characteristics greatly simplify the code development for the finite element classes and solvers (BITTENCOURT, 2000).

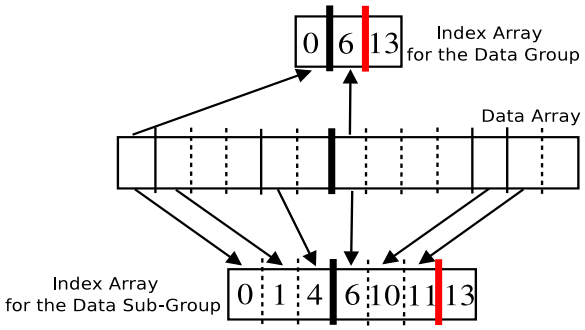


Figure 4.2. Illustration of the data structure of **TwoIndexTable** class.

4.3.2 Interpolation package

The *Interpolation* package is used to generate the nodal and modal one-dimensional interpolation functions based on Lagrange, Jacobi, Legendre, Hermite, and Lobatto polynomial bases (BITTENCOURT *et al.*, 2007; BITTENCOURT, 2014). The polynomial orders and the integration and collocation coordinates are the main input data for calculating the values of shape functions and their derivatives for each polynomial basis implemented.

The classes for the one-dimensional polynomial bases are encapsulated by the class **ShapeFunctions**, which is responsible for calculating the interpolation functions and derivatives on the integration and collocation points. Information about the integration and collocation points is encapsulated by another set of classes, specifically **IntegrationPoints** and **CollocationPoints**. The calculation and storage of interpolation functions occur independently for lines, squares, triangles, hexahedra, and tetrahedra. However, triangles and tetrahedra elements were not used in this work. An important aspect in the construction of the interpolation functions for two and three dimensions is the use of the tensor product of the one-dimensional bases (BITTENCOURT *et al.*, 2007).

4.3.3 FEGroups package

The FEGroups package represents the finite element groups. Each group has finite elements with the same characteristics in terms of material, shape, and interpolation functions.

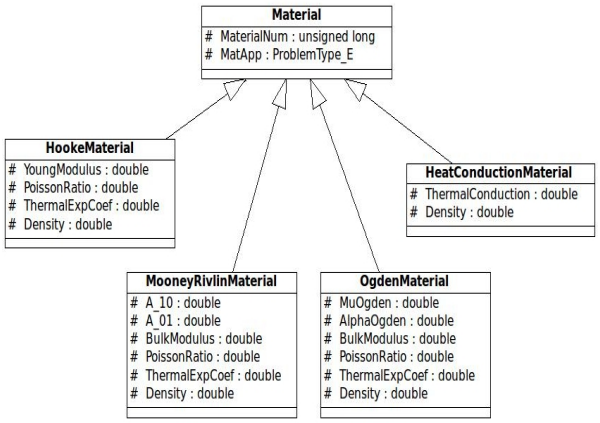


Figure 4.3. Material class diagram.

This package is also responsible for calculating the element operators, such as mass and stiffness matrices and load vectors implemented by the **FiniteElement** class.

The **Mapping** class is included in this package and is responsible for calculating the Jacobian matrix and the element measures (length, area, and volume).

The **Mesh** class stores the coordinates and polynomial orders of each element, as well as, the indices of their degrees of freedom and incidence. $(hp)^2$FEM software allocates instances of this class to four mesh types (input, solution, mapping, and post-processing) that can have different polynomial orders. The mapping and solution classes may have p-non-uniform distributions for the elements.

The hierarchy of classes shown in Figure 4.3 is used in the **FEGroups** package. The **Material** class defines different material models used by the applications implemented in the **Solver** package. The use of inheritance and dynamic polymorphism concepts allows flexibility when defining new material types.

Model package

The **Model** package manages all finite element groups stored in the instance of **FEGroups** class. This package builds and manages the main characteristics of the discrete model, i.e., numbering of degrees of freedom, incidences, coordinates, boundary conditions, and load sets. The diagram of this class is illustrated in Figure 4.4.

Furthermore, the package stores the relations between mesh and geometry that are used as input parameters of the finite element model. In this case, it is possible to obtain all nodes, element edges, and faces of geometric surfaces with a distributed load. The geometry is stored using the NURBS representation, with data obtained during the preprocessing stage (SEVILLA *et al.*, 2008).

The mesh topology is built and managed in the **Model** package through the **MeshTopology** class. The topology stores the mesh entities in tables as node–element and element–

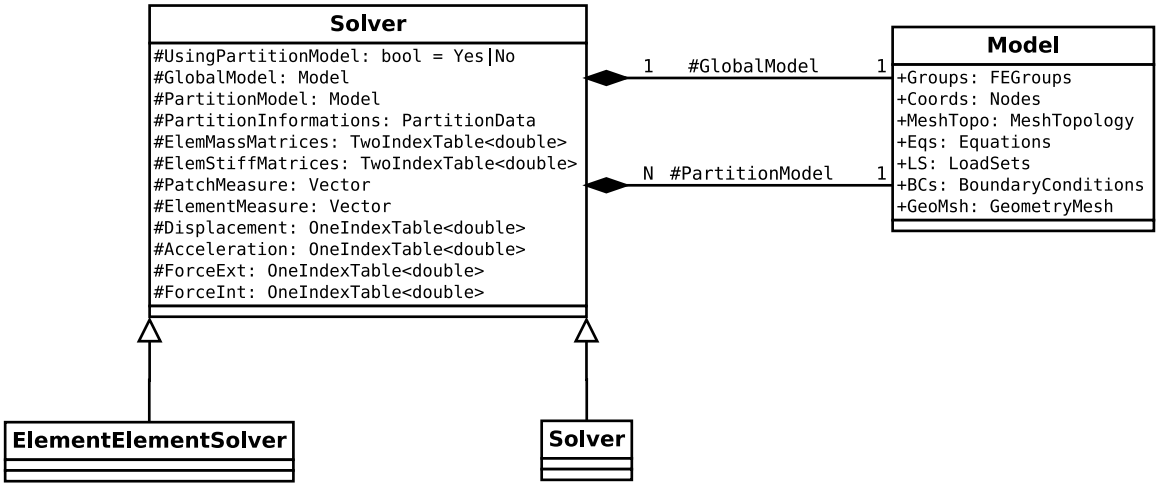


Figure 4.4. Diagrams of **Model** and **Solver** classes.

element relations. The **HighOrder** class is responsible for generating the new incidences and coordinates from the input mesh and given mapping polynomial orders. The **MeshTopology** package is shown in Figure 4.5.

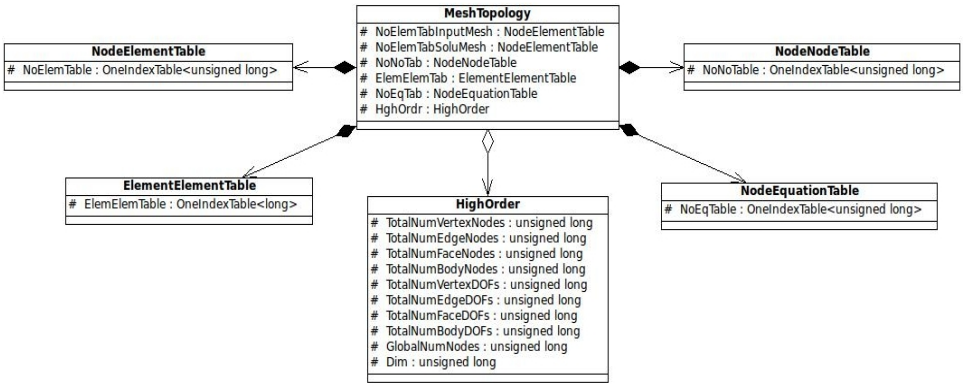


Figure 4.5. Classes of the **MeshTopology** package.

Solver package

The solver procedures are implemented in the solver package. There are two main solution strategies for the finite element mesh. In the first one, the solution is calculated globally for the element operators assembled in global sparse matrices. In the second approach, the solutions are calculated locally, i.e., element by element, as described in Section 2.4. The software contains solvers for permanent (static) problems, Newton's method for non linear elasticity, explicit and implicit methods for transient analysis and phase field models for damage and fracture. The diagram of this class is illustrated in Figure 4.4.

4.4 REMODELING AND OPTIMIZATIONS OF THE $(hp)^2$FEM SOFTWARE

One of the main goals of this work was to tune the $(hp)^2$FEM software on high performance computing architectures. Chapter 5 discusses the code optimization applied to

speedup the execution time of $(hp)^2$FEM .

As a first step in accelerating the code, we applied profiling tools and APIs from the Argonne Leadership Computing Facility (ARGONNE, 2015) and Center for Computing in Engineering & Sciences (CCES, 2020). As expected, code profiling revealed that a significant portion of the execution time was associated with the solver. Because solvers rely on matrix–matrix and matrix–vector operations, we first identified and replaced segments of the code by calls to BLAS. The machine vendors provide highly optimized versions of these routines in libraries, which can be easily linked to the code. We also used functions from the LAPACK library (DONGARRA, 2003) to solve the linear systems.

The assembly code profile revealed that the compiler was unable to automatically hoist some loop-invariant computations out of their respective loops. Hence, we manually modified the code to hoist the loop-invariant computations. We also used special flags of the compiler (-qhot (GILGE, 2014a)) to perform more aggressive loop unrolling, which provided significant performance benefits.

We calculate shape functions for squares and hexahedra using the tensor product of the one-dimensional matrices (D1-Matrices procedure); the equations are described in Section 2.2. The element matrices are denser for high-order shape functions. Hence, this procedure reduced the time and memory consumption when using high-order polynomials, as will be discussed in Chapter 6.

5 PARALLELIZATION OF THE $(hp)^2$FEM SOFTWARE

This chapter discusses the parallelization of the elementwise solvers of the $(hp)^2$FEM for projection problems and explicit transient analyses discussed in Chapter 2. The main changes considered in the parallel implementation are in the upper layer of the $(hp)^2$FEM architecture. Sections 5.1 to 5.4 describe how the finite element model is partitioned and the parallel procedures considered to renumber the nodes on the interfaces of partitions/sub-domains. In Section 5.5, the solutions to avoid deadlock during communication are presented. Finally, Section 5.6 describes the implementation of the central difference elementwise method, used to linear and non-linear structural problems. The parallel versions are implemented using MPI and OpenMP, respectively, for execution on distributed and shared memory based high-performance computing systems.

5.1 MESH PARTITIONING

The finite element mesh is partitioned using the METIS library (KARYPIS, 2011), which uses procedures based on graph theory (NETTO, 2012). Different processes manage each partition. Initially, the element mesh is converted to a weighted graph that guides the METIS library in the partitioning procedure, looking for minimizing the interfaces (formed by element faces, edges, and nodes) of partitions and reducing communication costs among the processes. The local information about each partition is stored in attributes of the **PartitionData** class, illustrated in Figure 5.1, of the Solver Package. Each process exchanges information about its partition boundary using MPI routines. Subsequently, each process constructs data structures to map the partition boundary elements and nodes to synchronize partial results to neighboring processes/partitions.

PartitionData is instantiated as an attribute of the **Solver** class, replacing the use of attributes of the GlobalModel instance that stores all the parameters of the serial finite element model. In summary, each process executes the following steps to construct information for its partition:

1) The input mesh, which in most cases is composed of linear elements, is read and converted into a graph;

2) The METIS library is invoked to partition the graph;

3) Each partition is associated with one process labeled with the MPI rank (process ID). A data structure is created to store the internal and boundary elements, the list of neighboring partitions, and the mapping between the global and local element and node numbers.

4) The incidences, degrees of freedom, and boundary conditions in the local partition are renumbered, and data structures are created to map the local and global incidences of the partition boundary elements.

The parallel methods for the local solvers were implemented in the **ElementElementSolver** class described in Section 4.3. In the **ElementElementSolver** class, the parallel algorithms were designed using the local numbering of incidences and degrees of freedom of the partition elements. Besides that, the parallel elementwise solvers allocate the vector *PatchMeasure* to store the element measures for weighting the local solutions.

MPI is used to exchange the local *PatchMeasure* and the solution of the boundary elements among processes. In the projection solver, the system of equations for the internal elements is calculated separately from the boundary elements. For the transient analysis solvers, the local solutions are calculated for all partition elements and sent to neighbors. The local vector *PatchMeasure* is exchanged one time among partitions, before the time step loop. Next, the local solution of the system of equations is received by the neighboring partitions. After that, the local solution on the boundary elements is updated and weighted using the *PatchMeasure* vector. As a result, each partition computes the global solution for its degrees of freedom. The energy norm of the approximated solution is also calculated in parallel.

5.2 PARTITION DATA CLASS

A summary view of the **PartitionData** class with its key attributes is illustrated in Figure 5.1. This class belongs to the **Solver** package of Figure 4.1 and stores information of the mesh partitions used in the parallel distributed memory strategies to be discussed later. The class has data structures for the local incidences and auxiliary arrays for the boundary degrees of freedom of the partition. This class implements the methods to build the required structures for different parallel strategies used to exchange solutions of the system of linear equations among partitions. Additionally, the class has the parameters for loading and balancing used by the METIS library for partitioning the global mesh.

The key attributes of the **PartitionData** class illustrated in Figure 5.1 are built from the output variables of the METIS library. These attributes are instances of classes of the **DS** package (Section 4.3.1) and store, for example, the mapping of the local (*LocalElements*) and boundary (*BoundaryElements*) elements to global numbers as well the incidences (*LocalIncidences*). Furthermore, some attributes store the mesh topology for each sub-domain, e.g., an array with the numbering and the total number of neighbors (*NeighborsPartitions*). The boundary solutions are updated using the degrees of freedom indices. The variables *BoudaryIndicesRecv* and *BoudaryIndicesSend* are used to access the boundary solution when sending to or receiving from solutions of other partitions.

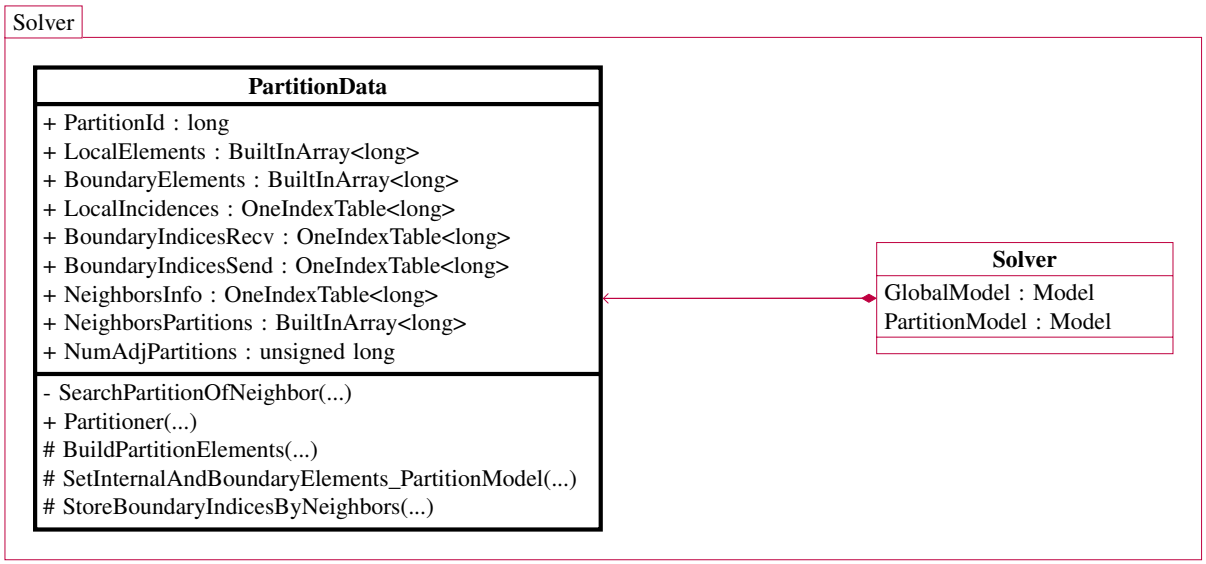


Figure 5.1. PartitionData class diagram.

The method *Partitioner(...)* handles the algorithm for partitioning of the finite element meshes. It prepares the input of variables for METIS whose results are used by *BuildPartitionElements* and *SetInternalAndBoundaryElements_PartitionModel(...)* methods to generate the local information in a partition. The *StoreBoundaryIndicesByNeighbors(...)* variable stores the values for the boundary elements indicating if the neighborhood is in terms of a node, edge or face as will be explained later. This information is used to set up the variables *BoudaryIndicesRecv* and *BoudaryIndicesSend*.

5.3 METIS FUNCTIONS FOR MESH PARTITIONING

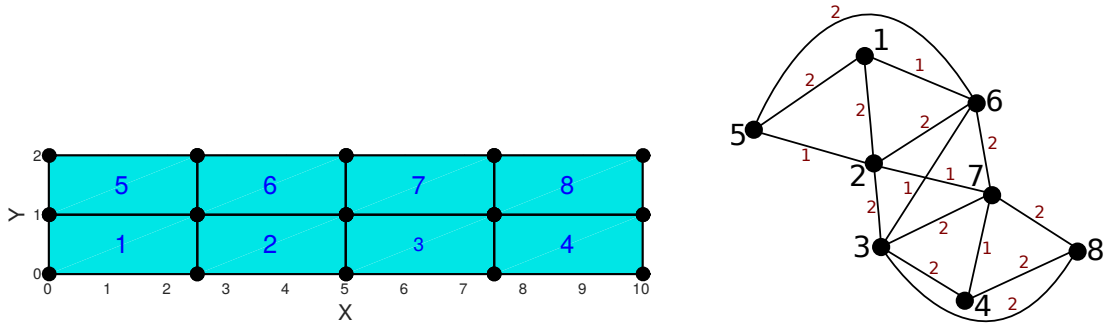
The manner the mesh is partitioned for the processes and the mapping of nodes and degrees of freedom among the neighboring partitions is important to increase the performance when exchanging messages. The reduction of the communication time between processes is an NP-Hard problem (non-deterministic polynomial acceptable problem) due to the exponential growth of the number of options to distribute the finite elements in distributed computers (FERREIRA, 2012).

The algorithms for partitioning finite element meshes were implemented using the METIS library, a software package for partitioning unstructured graphs and meshes developed at the University of Minnesota (GUEDES, 2009).

Function *PartGraphRecursive* of METIS was used to set loading and balancing to the finite element mesh. Firstly, the mesh was converted to a graph. Then, we wrote the code to store the input parameters for the *PartGraphRecursive* function.

Figure 5.2 shows how a mesh is converted to a graph: the elements are the vertices, and the neighborhood among them are the edges of the graph. Each edge has a weight that

corresponds to the number of shared nodes between two neighbor elements. For instance, the eight-node hexahedron can share one, two, or four nodes. Here, each of these possibilities will correspond to different weights for the graph edge, consequently changing how the graph will be partitioned. Furthermore, the number of shared nodes are increased by high-order polynomials used for each finite element.



(a) Finite element mesh.

(b) Mesh converted to a weighted graph.

Figure 5.2. Example of conversion of finite element mesh to weighted graph.

The signature for the *PartGraphRecursive* function is **int METIS_PartGraphRecursive** (idx_t *nvtxs*, idx_t *xadj*, idx_t *adjncy*, idx_t *vsize*, idx_t *adjwgt*, idx_t *nparts*, idx_t *options*, idx_t *part*). The main parameters used to input ([in]) and output ([out]) data are as follows:

- *nvtxs*[in]: number of vertices of the graph; in this case, the number of mesh elements.
- *xadj*[in]: array to map adjacency list *adjncy*. The *xadj[i]* variable is used to access the first neighbor of element i into the *adjncy* array. The number of neighbors can be obtained by $xadj[i+1]-xadj[i]$.
- *adjncy*[in]: adjacency list which stores neighbors of each vertex.
- *adjwgt*[in]: an array with the weights of the edges. The weights are the number of shared nodes between elements. The size of *adjwgt* is $2 \times m$, where m is the number of graph edges.
- *nparts*[in]: number of partitions to divide the graph.
- *options*[in]: an array that allows configuring the characteristics or behavior of the METIS algorithm.
- *edgecut*[out]: returns the sum of the edge path's weights where the graph was partitioned. If there are no weights on the edges, the number of graph edges will be returned. Figure 5.3 shows an example of a partitioned graph. The value of *edgecut* will be the sum of weights of edges $(2,3)$, $(18,19)$, $(19,16)$, $(16,17)$ and $(10,9)$. The algorithm's goal is to obtain the smallest value of *edgedcut* minimizing the communication between partitions.

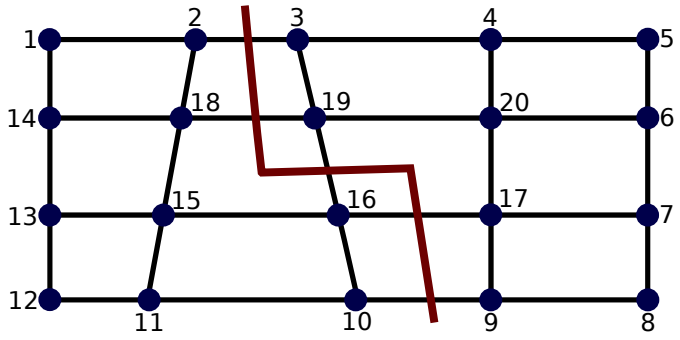


Figure 5.3. Partitioned graph with the edgecut parameter.

- *partitions*[out]: the result of the graph partitioning. The array size is the number of mesh elements or graph vertices defined by *nvtxs*, and each position represents a finite element *i*. The value stored in the array index *i* is the partition number assigned to the element *i*.

For meshes associated with complex geometry and distorted elements, as for the crankshaft used in Chapter 6, we employ a multilevel *k*-way graph partitioning (**METIS_PartGraphKway**). The algorithm has the same outputs as the *PartGraphRecursive* or multilevel bisection method explained earlier. The multilevel *k*-way algorithm has three phases: coarsening, initial partitioning, and uncoarsening. The coarsening step groups vertices, scaling down the size of the graph to a few hundred vertices. The initial partitioning computes the coarse graph in *k*-way partitions. At last, the smallest graphs are sequentially extrapolated in larger graphs until the original graph size is reached (GUEDES, 2009). Results with this algorithm will be shown in Section 6.8.

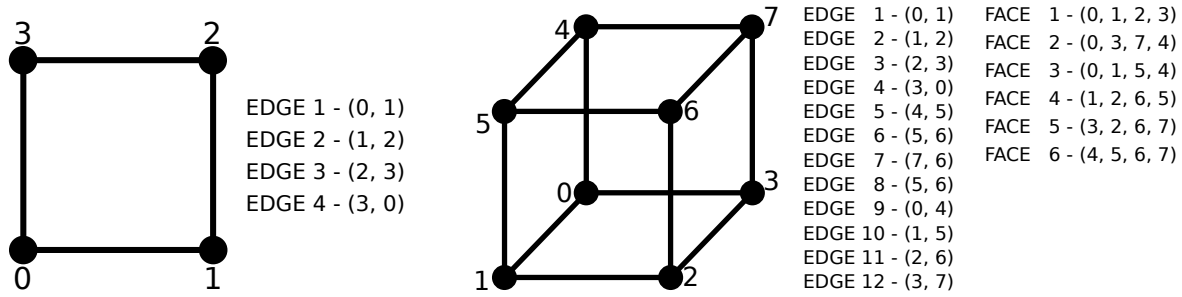
5.4 PARALLEL NUMBERING ALGORITHMS TO INTERFACES OF PARTITIONS

The algorithms described in this section are applied to the MPI communication among the processes. These algorithms support the message exchange for parallel solvers using different numbering of incidences on the partition boundary. Consequently, the exchanging of solutions among partitions is modified. Each partition stores the local incidences and node numbering to support the assembly of the global solution of solvers. Some procedures use a mapping structure to convert from global to local or local to global numbering.

The algorithms considered in this section use the element topological entities. They are the vertices and edges for squares and vertices, edges, and faces for hexahedra. Figure 5.4 shows the topological entities and their labels or indices for square and hexahedron considered in $(hp)^2$FEM .

5.4.1 Overlapping algorithm

The overlapping procedure was implemented to avoid communication overhead when increasing the number of mesh elements and their polynomial orders. Figure 5.5 shows the



(a) Square. **(b)** Hexahedron.

Figure 5.4. Topological entities and their indices for square and hexahedron in $(hp)^2$FEM .

partition of a domain into four sub-domains $P1$, $P2$, $P3$, and $P4$. Each sub-domain is allocated to one process and solves the elementwise problem.

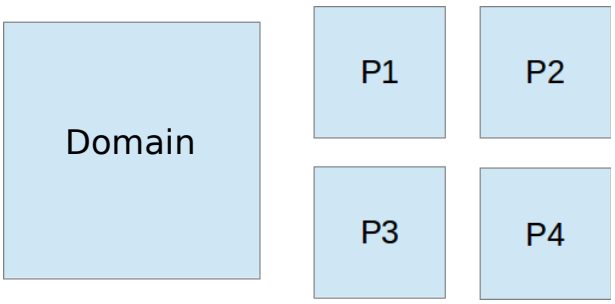


Figure 5.5. Domain decomposition in four sub-domains.

The global domain result is computed by the assembly of the local solutions from the four partitions or sub-domains. Generally, updating the global solution on the sub-domain boundaries is done by exchanging messages among the processes. To avoid this cost of communication, the boundary information for neighbors of each sub-domain is duplicated.

Figure 5.6 shows the overlapping algorithm implemented in $(hp)^2$FEM . The sub-domains $P1$ and $P2$ store data from their neighbors. For instance, data for elements on the boundary of the adjacent partitions are copied. Based on that, the global solution for the degrees of freedom is updated for each partition's original boundary, avoiding communication among processes.

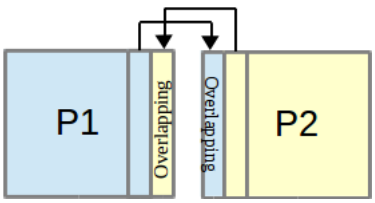


Figure 5.6. *Overlapping* algorithm and duplicated regions.

The method *BuildOverlappingData* of the **PartitionData** class (Figure 5.1) implements Algorithm 5.1 and builds three storage structures. This algorithm saves the numbering of

elements adjacent to the boundaries for a specific partition ID, the incidence, and the element topological entities (node, edge, or face) of these adjacent elements.

Algorithm 5.1 Algorithm for the *BuildOverlappingData* method.

Input: Global Solution Mesh, Partition Boundary Elements, Adjacent Partitions.
Output: Partition Boundary Elements, Element Incidences, Element Topological Indices.
begin

- ▷ Allocates the local incidences in the ID partition;
- ▷ Builds the mapping array to map global-local incidences;
- ▷ Numbers the local partition incidences;
- ▷ Assigns the incidences to adjacent elements of the partition (or boundaries elements).

Build Data of Overlapping Region
begin

- Calculates and allocates the size of adjacent elements to the ID partition;
- Calculates and allocates the size of the vector to store the partition number for each adjacent element in the boundary region.
- Saves the adjacent elements of the boundary region and the partition ID number to which the element belongs.
- Computes the numbering of incidences and topology indices to adjacent elements of the partition.

The overlapping algorithm is used in the local projection solver of Section 2.3 implemented in the **ElementElementSolver** class. The solution of the system of linear equation is performed in two stages. First, the systems of equations are solved for elements in the overlapping regions. In the second stage, the systems are solved locally for the internal elements, and their solutions are saved into the same vector used by the elements in the overlapping regions. The error norm is also calculated in parallel. The performance of this algorithm will be presented in Chapter 6.

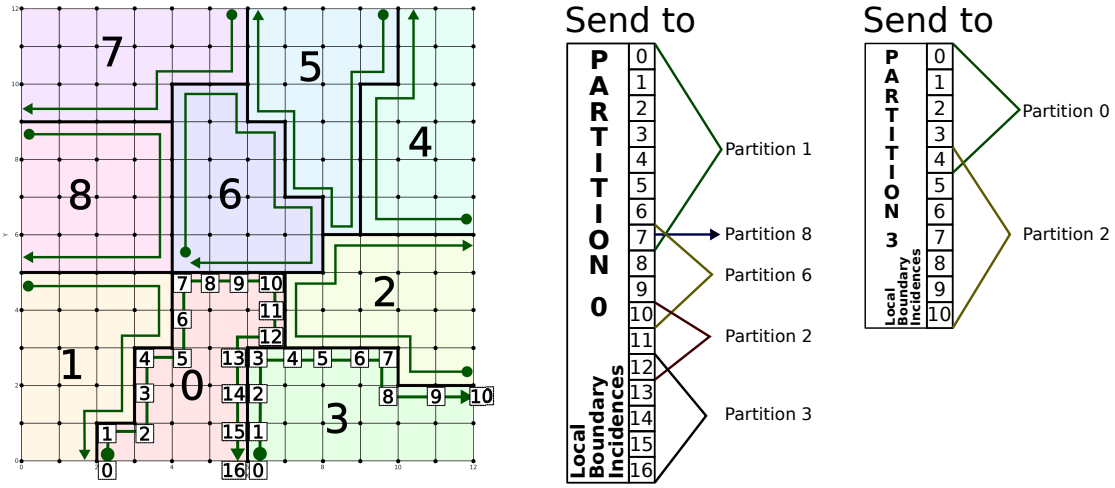
5.4.2 Sequential numbering algorithm for boundary nodes

This algorithm aims to number first the boundary nodes of each partition sequentially, avoiding the exchange of the node numbering arrays when communicating with neighboring partitions.

This algorithm starts by numbering the boundary nodes of partitions' interfaces in a clockwise direction, sequential, and synchronized, using each partition label. Figure 5.7 illustrates the partitions, the node numbering for partitions 0 and 3, and the way they communicate with other sub-domains.

Partition 0 has boundaries with partitions 1, 8, 6, 2 and 3; partition 3 sends and receives data to partitions 0 and 2. The partitions need only to send the first node number and the offset of the local solution vectors. Based on that, the partition receiving the solution vector

knows how to address the correct positions to update its boundary solution. Consequently, it will not be necessary to exchange all node numbers.



(a) Partitions for a square mesh.

(b) Partitions 0 and 3 sending information to other partitions.

Figure 5.7. Numbering of boundary incidences by sequential algorithm.

The starting partition is that one with the minimum partition ID found by scanning the boundary elements. The ID used depends on the load balancing given by the **METIS_PartGraphKway** function of METIS because some MPI ranks maybe not assigned. For example, suppose we are running five processes, but METIS partitioned the mesh with three partitions and assigned the MPI ranks $[4,2,3]$. In this case, the first partition to start the sequential renumbering algorithm is partition 2.

This algorithm is applied to square meshes. The main idea is to reduce the memory usage for each partition and size of messages with the MPI communication for the local solvers. The steps of the algorithm are described as follow:

1. Search the first valid partition and determines the first element to start, which must share an edge with another partition.

2. Renumber the interface nodes for each neighboring partition. The method scans all shared edges with other partitions. The renumbering of nodes is generated using the incidence ordering generated by $(hp)^2$FEM . This stage will save the node number of the first renumbered node and the number of nodes for each neighboring partition.

3. Get the next partition using the last renumbered edge. Subsequently, the first global element and edge will be sent to the next process to renumber its local nodes. The interface elements from a partition are known by their neighboring partition.

4. The next process receives the first element and the topological element index, which was used to start the numbering of the interface local nodes. Given that, three conditions may happen:

a) If the process has not yet performed the renumbering of nodes and received the first element and edge to start the numbering, the algorithm returns to step two.

b) If the process already did the local nodal renumbering, searches for the next partition to be renumbered. Following, go to step two.

c) If the process has not yet performed the local renumbering and not received the first edge and element, the algorithm will search the first edge and element using the topological element indices and its neighboring partitions. The element found must be the first shared with the neighbor partition, considering clockwise direction through the elements. Subsequently, the algorithm will repeat step two.

5. Renumber other local incidences that are not on the boundary. The internal renumbering is made following the sequence of the topological element indices of the global mesh. Therefore, the renumbering will not be sequential.

6. The data structure to save the local incidence numbering is split into the internal and boundary incidences.

5.4.3 Non sequential numbering algorithm for boundary nodes

The non sequential algorithm is an independent manner to renumber the interface nodes of partitions when compared to the sequential procedure of the previous section. However, the global interface nodes must be exchanged among partitions to map the correct local boundary node number of another partition.

The algorithm starts with each process finding and numbering the global boundary nodes locally into the partition and sending them to its neighbor sub-domains. After receiving this information, the neighboring partition uses global-local mapping to indicate the local nodes' interface. Consequently, each partition will store the local interface nodes in the same position that the global nodes were received. This ordering is essential to update solution values in the correct positions during the solver processing. The partitions will exchange only a part of the solution arrays. The solution arrays' indices will not be sent because they already have pre-processed to update the boundary nodes' solution. This algorithm has the following steps:

1. Set a table that stores the numbering of global nodes shared with each neighboring partition;

2. Renumber the local nodes for each boundary element of partition;

3. Renumbers the local nodes for each internal element.

4. Construct the data structures to store the local internal and boundary nodes of the partition.

5. Allocate index arrays to map the local incidences of the partition and its neighbors. This array is filled using the shared global nodes. Thus, an auxiliary map will be used to convert the indices from global to local.

6. Sort and send the shared global nodes for each neighboring partition. Sending and receiving the local solution arrays among the partitions will follow this ordering, allowing the correct update of the shared interface nodes' solutions.

The solvers will use the data structure to map local incidences among partitions to exchanging the local solutions.

5.4.4 Non sequential numbering for boundary nodes using the PartitionModel object

As mentioned before, each process reads the global finite element model and stores the mesh parameters (nodal coordinates and element incidences) and other attributes (material properties, loads, boundary conditions, polynomial order, solver directives, and many others) in the attributes of the **Model** class presented in Section 4.3.3. After the METIS library returns with the information on the mesh partitioning; each process will manage one sub-domain. For the given polynomial order, the **HighOrder** class generates the high-order nodal coordinates and element incidence but still using the global model stored in the instance of the **Model** class for each partition. As the high-order data may demand much memory, depending on the polynomial order and the number of elements, the advantages of partitioning the finite element model may be lost. This occurs because, in the serial version of $(hp)^2$FEM , the high-order information is generated for all global model elements.

To overcome this limitation, the *PartitionModel* instance of the **Model** class is allocated for each partition and the *GlobalModel* object reads only the required parameters from the input files. The high-order incidences and coordinates are generated only for each partition element, avoiding the higher computational cost and especially memory space.

Also, it was necessary to add auxiliary structures of type **OneIndexTable**. These tables are used to map global and local incidences since the common boundary incidences between the partitions are identified using the global numbering. Consequently, there is an increase in the use of memory for high order elements and communication time of MPI messages during the creation of these auxiliary structures. On the other hand, the considered algorithm now uses only local information of high-order incidences and nodal coordinates, reducing the computational cost for memory and synchronization time using MPI functions. The creation of high-order information in each sub-domain allowed increasing the number of processors, as described in Chapter 6.

The algorithm proposed here builds two auxiliary tables of type *OneIndexTable*, *BoundaryIncidRecv*, and *BoundaryIndicesSend*, which will be used in the solver for updating the solution of degrees of freedom on the partition boundary. These tables are created from the

topological indexes (node, edge, and face) of the neighbor's elements, as we will see below. The tables store the local incidence for each neighbor element. The *BoundaryIncidRecv* table stores the incidences using the element topological entities of $(hp)^2$FEM and exemplified in Figure 5.4. The organization of the table *BoundaryIncidSend* follows the order of incidence numbering of the neighbor element of the current partition. In this way, the *BoundaryIncidSend* table will be used to access the boundary solution sent to neighboring partitions and the table *BoundaryIncidRecv* to update the boundary solutions of the current partition.

There are other auxiliary structures created from the METIS partitioning that will be needed before creating the *BoundaryIncidRecv* and *BoundaryIndicesSend* tables. These structures store information about the boundary of partitions. They allowed to optimize and simplify data access when creating the mapping of boundary incidences between neighboring partitions, as they only consider the local incidence numbering.

The **PartitionData** class associates these auxiliary structures with the boundary elements from the element topological indexes. The local incidences and nodal coordinates are accessed from the topological indexes of the element. The main auxiliary structures are computed in the following order after the METIS partitioning:

- Array of boundary elements;
- Array of internal elements;
- Array of neighboring partitions of each partition;
- Mapping table between global and local element numbers;
- Table with information about each boundary element.

The last variable is *BoundaryElementInfo*, an instance of **OneIndexTable**. Data stored in this table have information associated with each boundary element of each subdomain. This table stores for each boundary element: the adjacent element, the shared topologic entity (node, face, edge), the partition ID, the partition ID of the adjacent element, and the number of shared vertices.

Table *NeighborsInfo* of **PartitionData** is also allocated as an instance of the **OneIndexTable** class. This table is computed from the auxiliary table *BoundaryElementInfo*. The information stored in both tables are the same. However, *NeighborsInfo*'s data is associated by neighboring partitions and not for boundary elements or the partition as in the *BoundaryElementInfo* table. The tables *BoundaryIndicesSend* and *BoundaryIndicesRecv* are built from the storage order of the *NeighborsInfo* table.

An example of the data stored in *NeighborsInfo* variable is given in Tables 5.1 to 5.2 for a mesh of 15 quadratic elements partitioned into 3 sub-domains as shown in Figure 5.8.

The partitions 1, 2, and 3 are represented by the colors red, green, and yellow, respectively. Tables 5.1 to 5.3 store, for each neighbor partition, information about the boundary elements and their adjacent ones in neighboring partitions. In this way, each row of these tables are organized in the following order:

1. The number of the boundary element of the current partition.
2. The number of the adjacent elements of the neighboring partition.
3. The topological entity is shared by the element and one of its adjacents, which have values 0, 1, and 2 for the node, edge, and face, respectively.
4. The local number of the topological entity is shared by the boundary element and its adjacent. For a quadratic element the values are $[0,3]$ for node, $[0,3]$ for edge and $[0,5]$ for the face.
5. The number of vertices shared between the boundary element and its adjacent.
6. The number of the topological entity of the adjacent element.

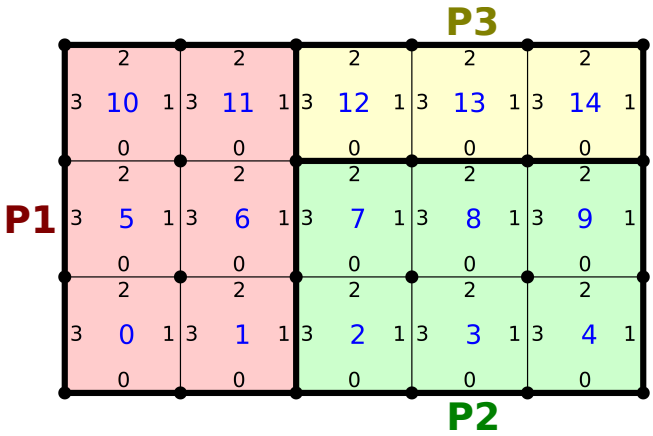


Figure 5.8. Mesh of square partitioned in three sub-domains using the METIS library.

Data in Tables 5.1 to 5.3 were organized in this way so that, based on the incidences of elements at the interfaces of neighboring partitions, the incidences for the same topological entity of the neighbor elements are obtained. The algorithm works according to the following conditions:

- If the current partition *ID* is greater than the neighboring partition *ID*, data will be ordered by the numbering of the adjacent elements and then by the numbering of the boundary elements of the partition;

Table 5.1 – NeighborsInfo table for partition 1.

Partition 1					
Number of neighboring partitions: 2					
Neighboring ID: 2					
Boundary	Adjacent	Entity	Entity # (Boundary)	# Vertices	Entity # (Adjacent)
1	2	1	1	2	3
1	7	0	2	1	0
6	2	0	1	1	3
6	7	1	1	2	3
11	7	0	1	1	3
Neighbor ID: 1					
Boundary	Adjacent	Entity	Entity # (Boundary)	# Vertices	Entity # (Adjacent)
6	12	0	2	1	0
11	12	1	1	2	3

Table 5.2 – NeighborsInfo table for partition 2.

Partition 2					
Number of neighboring partitions: 2					
Neighbor ID: 1					
Boundary	Adjacent	Entity	Entity # (Boundary)	# Vertices	Entity # (Adjacent)
2	1	1	3	2	1
7	1	0	0	1	2
2	6	0	3	1	1
7	6	1	3	2	1
7	11	0	3	1	1
Neighbor ID: 3					
Boundary	Adjacent	Entity	Entity # (Boundary)	# Vertices	Entity # (Adjacent)
7	12	1	2	2	0
7	13	0	2	1	0
8	12	0	3	1	1
8	13	1	2	2	0
8	14	0	2	1	0
9	13	0	3	1	1
9	14	1	2	2	0

- otherwise, the current partition *ID* will be less than the neighboring partition *ID* and the

Table 5.3 – NeighborsInfo table for partition 3.

Partition 3					
Number of neighboring partitions: 2					
Neighbor ID: 1					
Boundary	Adjacent	Entity	Entity # (Boundary)	# Vertices	Entity # (Adjacent)
12	6	0	0	1	2
12	11	1	3	2	1
Neighbor ID: 2					
Boundary	Adjacent	Entity	Entity # (Boundary)	# Vertices	Entity # (Adjacent)
12	7	1	0	2	2
13	7	0	0	1	2
12	8	0	1	1	3
13	8	1	0	2	2
14	8	0	0	1	2
13	9	0	1	1	3
14	9	1	0	2	2

ordering will be established first by the numbering of the boundary elements and then by the adjacent boundary elements.

This organization guarantees an ordering between neighboring elements of neighboring partitions by topological indices, which allowed gains in performance and ease of preparation of the auxiliary tables *BoundaryIncidRecv* and *BoundaryIncidSend*.

In the following step, most of the mesh data stored in the *GlobalModel* object of **Model** class is deleted to save memory space. The *PartitionModel* object of **Model** class is constructed calling methods of $(hp)^2$FEM . These methods map the boundary conditions and load sets of the global into each partition using local numbers. Then, groups, incidences, and coordinates are renumbered locally in each partition, and the high-order incidences and coordinates are generated. Figure 5.9 shows the main methods used to create the data structure for the $(hp)^2$FEM parallel version. The high-order coordinates and incidences will be set up according to the polynomial orders of the solution, mapping, and post-processing meshes, which may use different polynomial orders.

Finally, the tables *BoundaryIndicesRecv* and *BoundaryIndicesSend* are generated for nodal basis as follows:

1. Calculate the size of each row of tables that will be used to update and send boundary incidences between neighboring partitions. This size is obtained by using data of table

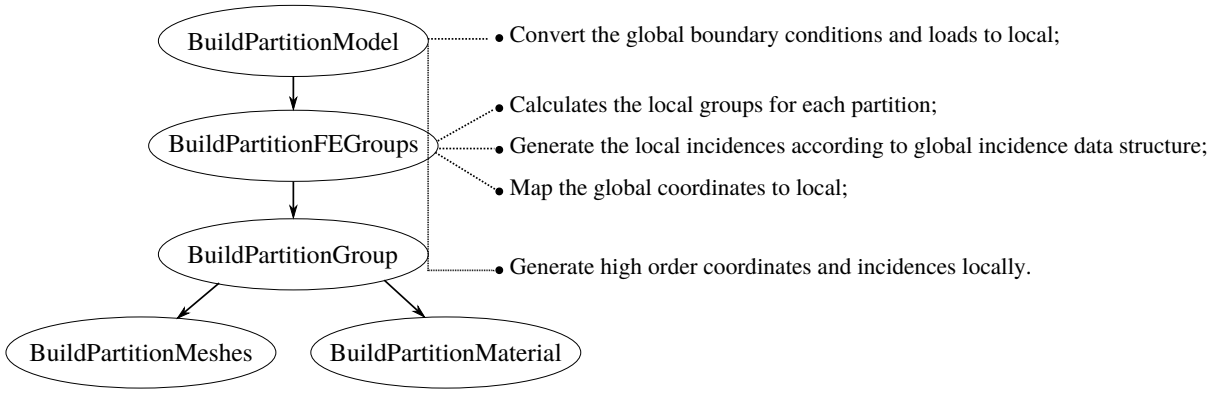


Figure 5.9. *PartitionModel* package of the parallel $(hp)^2$FEM .

NeighborsInfo. The maximum number of boundary incidences for each neighboring partition is obtained by means of the topological entity (node, face, edge) of the boundary element and the polynomial order.

2. Calculate local incidence and nodal coordinates for each boundary element of the partition using the order given in table *NeighborsInfo* for topological indexes of the boundary elements. In this step, the *BoundaryIndicesRecv* table is built.

3. Exchange of boundary coordinates between neighboring partitions using the Algorithm 5.4 to obtain the correct order that the neighboring partition accesses the incidence for the same topological entity. The use of coordinates is necessary, as the same topological entity may differ between boundary elements and their adjacents.

4. Finally, for each neighboring partition and each element in table *NeighborsInfo*, the *BoundaryIndicesSend* table is created with the incidences already ordered for the solution array be accessed by the neighboring partition. Data of *BoundaryIndicesSend* will be filled using table *BoundaryIndicesRecv* and the topological entity coordinates of each boundary element.

BoundaryIndicesRecv and *BoudaryIndicesSend* tables are used in the elementwise solvers of Sections 5.6 and 2.7. The same structures are also used for the global solvers of $(hp)^2$FEM .

5.5 THE DISTRIBUTED ALGORITHM FOR AVOIDING DEADLOCK PROBLEMS.

Due to concurrency, deadlock problems are common in parallel applications. It occurs when one or more processes wait for data or resources from another process. Some approaches were developed in $(hp)^2$FEM to use more nodes and cores as possible in clusters, for example, the IBM Blue Gene/Q described in Section 6.1. Before implementing the deadlock algorithm, it was not possible to run meshes with more than one thousand hexahedra and polynomial orders up to four for the local projection solver. The overlapping algorithm, described in

Section 5.4.1, improved this limitation, allowing to run meshes with more elements and nodes of the IBM Blue Gene/Q. In this case, meshes with one million hexahedra and polynomial order up to four using 16 thousand processes were used. However, the performance was not good, as will be discussed in Section 6. The deadlock problem was identified in the IBM Blue Gene/Q computer only when using more than $1,024$ processes/nodes.

The earlier solver version using point-to-point communication with MPI caused deadlock among the processes. Larger finite element meshes increase MPI communication messages' size and, consequently, the wait time to receive a message from neighboring processes. As a result, the code may experience deadlock and processes wait for tasks that will never be completed.

Algorithm 5.2 is a classic example of the deadlock problem. Processes A and B try to send data to each other, and both processes keep waiting to begin a new task. The situation can be avoided by choosing one process to send and the other to receive the data first, as illustrated in

Algorithm 5.2 Deadlock between two processes.

```
if Process A then
    // Process A
    Send_Data_To(Process B);
    Receive_Data_From(Process B);
else
    // Process B
    Send_Data_To(Process A);
    Receive_Data_From(Process A);
```

Algorithm 5.3 An algorithm to solve the deadlock between two processes.

```
if Process A then
    // Process A
    Receive_Data_From(Process B);
    Send_Data_To(Process B);
else
    // Process B
    Send_Data_To(Process A);
    Receive_Data_From(Process A);
```

A similar strategy can be used for more than two processes. In this case, the algorithms classify the processes into two categories: those who send and those who receive the message first. Some point-to-point communication cases classify sending or receiving using the processes IDs; for example, processes with odd IDs for sending and processes with even IDs for receiving data first. In another case, communication uses the topology of the processes: those on the right send and those on the left receive data as the first task (PACHECO, 2011; HAGER; WELLEIN, 2010).

Based on point-to-point communication and Algorithm 5.3, we propose in this work a mapping algorithm for the partitions with the same characteristics. The algorithm was proposed according to the four-color theorem. This theorem, proved by Appel and Haken in (Appel; Haken, 1977), stated that a plane divides a domain into continuous regions generating a map. Subsequently, no more than four colors are assigned to the regions, considering that neighboring regions can not have the same color (Appel; Haken, 1977; CHARTRAND; ZHANG, 2008). This map can be characterized by a two-dimensional finite element mesh, as illustrated in Figure 5.10.

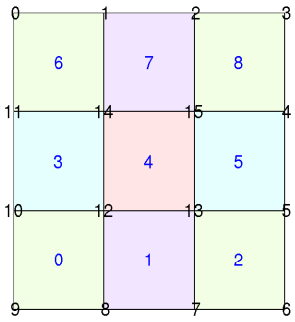


Figure 5.10. Color map for partitions for square finite element mesh.

Figure 5.10 presents the partitions of a global finite element mesh. The worst-case is when each partition has only one element, as shown in Figure 5.10. We assign colors to a $3D$ partitioned finite element mesh avoiding the same color between neighboring partitions, requiring a maximum of twenty-six colors to paint the global mesh regions.

Similarly to Algorithm 5.3, the procedure for color mapping for all partitions is a pre-processing step and avoids deadlocks for finite element meshes. The general method is shown in Algorithm 5.4. The starting color label is 0 and goes to the maximum number of colors to paint a partitioned mesh. The partitions with the same color label i will send first the solution vectors to the neighboring partition with a larger label $i+1$. Meanwhile, the partitions that do not have the same color i receive the solution and send back the solution vectors computed for the degrees of freedom.

Algorithm 5.4 Point-to-point communication using the color mapping algorithm for the $(hp)^2$FEM partitions (*ExchangeBoundariesSolutions*).

Input: Color map of the mesh partitions, neighboring partitions array, data to exchange on point-to-point MPI communication

```
/* Synchronization phase */
 foreach i = 0; i < number_of_colors; i++ do
    if my_partition.color == i then
       /* Color time. Send, then receive */
        /* Send to neighbors. */
        foreach n = 0; n < my_partition.number_of_neighbors; n++ do
           if neighbors[n].color > i then
              send_to (neighbors[n]);
       /* Receive from neighbors. */
        foreach n = 0; n < my_partition.number_of_neighbors; n++ do
           if neighbor[n].color > i then
              receive_from (neighbor[n]);
    else if my_partition.color > i then
       /* Receive from neighbors with color time. */
        foreach n = 0; n < my_partition.number_of_neighbors; n++ do
           if neighbor[n].color == i then
              receive_from (neighbor[n]);
       /* Send to neighbors. */
        foreach n = 0; n < my_partition.number_of_neighbors; n++ do
           if neighbor[n].color == i then
              send_to (neighbor[n]);
    /* my_partition.color < i : sync. already done. */
```

The procedure to assign color IDs to the global mesh partitions was implemented in the **PartitionData** class. The implemented algorithm allowed to execute of the projection solver with three-dimensional meshes with more than one million elements and increase the polynomial order, as will be shown in Chapter 6.

5.6 PARALLEL TRANSIENT EXPLICIT ELEMENTWISE SOLVERS

This section presents the key steps for the parallel implementation of the transient explicit elementwise solvers discussed in Sections 2.6 and 2.7 for linear and non-linear problems, respectively. The description below considers the linear version, and the non-linear version is similar.

As explained in Section 2.4, the central difference local solvers computes the systems of linear equations operating in an elementwise fashion. The parallel implementation used shared and distributed memory with MPI and OpenMP described in Chapter 3. The solutions for the displacement $\{u^{(n+1)}\}$ and acceleration $\{\ddot{u}^{(n+1)}\}$ vectors at time steps of each partition is exchanged to the neighboring partitions using Algorithm 5.4.

Two instances of the **OneIndexTable** class are used to store the element stiffness and mass matrices for the transient analyses. The rows store the matrices for each element of the sub-domain. Some auxiliary variables were also allocated for each element, for example the element load vector $\{f_e\}$ and displacement vector $\{u_e\}$. In addition, the vector for the displacement, velocity, and acceleration of the partition are also required.

The considered problem may have different load conditions, which are input using the concept of load cases. Each load case is a set of different loads applied to the edges, faces, or body of elements. Each element load generates a vector of nodal equivalent loads. The *ComputeLoadVector* method of the **Solver** class calculates the nodal equivalent loads for each load case of the partition.

In many parts of the $(hp)^2$FEM , OpenMP multithreading solutions were used. For loads, an OpenMP thread is used to calculate the equivalent load for the elements. Consider the example of Figure 5.11, where three partitions are used. Each edge load is computed by one thread using local attributes of the partition, and the thread performs the numerical integration of the load intensity using the shape functions obtained from the **Interpolation** package. For the symbolic load intensities, the symbolic analyzer procedure is used to evaluate the variables X, Y, and Z on the integration points.

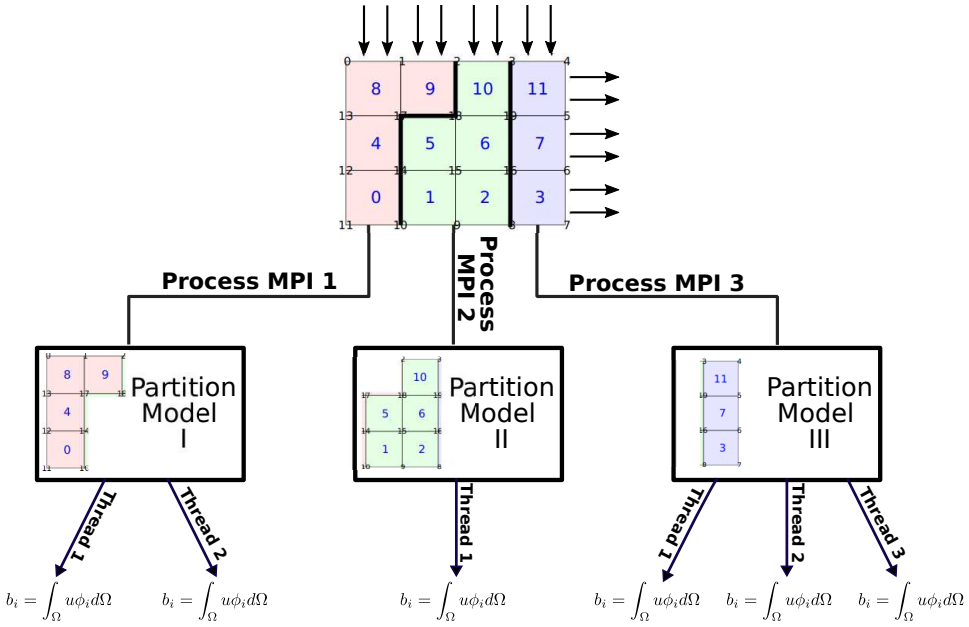


Figure 5.11. Use of MPI and OpenMP in $(hp)^2$FEM to calculate the load vectors.

The initial acceleration vector $\left\{u^{(0)}\right\}$ must be calculated before the main loop of the explicit method. The product of the stiffness matrix $[K]$ by the initial displacement $\left\{u^{(0)}\right\}$ or $\left\{u^{(n-1)}\right\}$ is done and used in Equation (2.45). Subsequently, the equation $\{f\} - [K]\{u^{(0)}\}$ is computed.

The main parallel regions implemented before the iteration loop of the transient solver are in constructing element matrices and calculating the initial acceleration and displacement.

When building the tables for the element matrices, parallel regions are defined so that each thread is responsible for calculating the stiffness and mass matrix coefficients for each element. The parallel region was implemented in the class **FEGroups** described in Chapter 4. Methods of the **FiniteElement** class, called by **FEGroups**, have been rewritten because the class attributes used in parallel regions could not be shared between threads. In general, compilers do not allow private members or class attributes used within parallel OpenMP regions, except for static members and attributes (OPENMP, 2018). In this way, some class attributes have been replaced by local variables in methods of the **FiniteElement** class used to calculate the coefficients of element matrices.

For the calculation of the initial acceleration, a parallel region was implemented for the matrix-vector product in each element, as we will see below. These operations can also be found in the element-by-element conjugated gradient method (CGD) implemented in (SUZUKI, 2017).

It is important to highlight that parallel regions are defined with explicit rules, that is, all variables used in a parallel region were explicitly defined as shared or private. The Code 5.1 illustrates the explicit case and how the parallel regions were defined for the solvers in $(hp)^2$FEM to calculate element solutions. On the other hand, the implicit way of declaring variables in the parallel regions is implemented when variables are not evidenced within the scope of *shared()* or *private*, as seen in Code 5.2.

Code 5.1 – Example of OpenMP - parallel region with explicit rules.

```
. . .
#pragma omp parallel default(none) \
    shared(NumberElements, ElementMassMatrices, ElementStiffMatrices) \
    private(ElemNum, Ke_P, Me_P, NumberLocalDOFs)
{
. . .
#pragma omp for
  for (ElemNum = 0; ElemNum < NumberElements; ElemNum++) {
    // Access the mass and stiffness matrix for all elements
    Ke_P = ElementStiffMatrices->GetRowData(ElemNum, NumberLocalDOFs);
    Me_P = ElementMassMatrices->GetRowData(ElemNum, NumberLocalDOFs);
    . . .
  }
  . . .
} // End of parallel region
. . .
```

Code 5.2 – Example of OpenMP - parallel region with implicit rules.

```
1     unsigned long NumberElements = GetNumberElements();
2     OneIndexTable<double> *ElementMassMatrices = GetElementMatrices(MASS
          );
3     OneIndexTable<double> *ElementStiffMatrices = GetElementMatrices(
          STIFFNESS);
4  . . .
5  #pragma omp parallel for
6    for (int ElemNum = 0; ElemNum < NumberElements; ElemNum++) {
7      int NumberLocalDOFs;
8      double *Ke_P = ElementStiffMatrices->GetRowData(ElemNum,
           NumberLocalDOFs);
9      double *Me_P = ElementMassMatrices->GetRowData(ElemNum,
           NumberLocalDOFs);
10     . . .
11   }
12   . . .
```

Before the solver iteration, synchronization among processes is also used to update the initial displacement. In this way, the displacement solutions $\{u^{(-1)}\}$ (see Equation (2.46)) for the degrees of freedom on the partition boundary will be exchanged via MPI point-to-point communication using Algorithm 5.4. In addition, the vector of element measurements (length, area, and volume) are also updated in the neighboring partitions. Then, the displacement solution's weighting in the partition boundary will be computed according to Equation (2.36).

Since the solver is iterative, it was not possible to implement OpenMP directives in such a way that each thread takes care of each time step. In this way, the application of multithreading parallelism was carried out by dividing the main iteration loop into stages, where each one starts and ends a parallel region through OpenMP directives.

These parallel regions are defined in each of the following solver stages: calculation of displacement and local acceleration (Equations (2.50) and (2.51)); exchange the boundary solutions among partitions; displacement weighting and global acceleration calculation (Equations (2.52) and (2.54)); global-local mapping; computation of internal loading (Equation (2.59)); loading interpolation and residue calculation (Equation (2.61)); solution of system equations (2.47) for the residue using the conjugated gradient method with diagonal preconditioner (CGD); update of local loading array to the next time step; and finally, calculation of the energy norm of the approximated solution.

In the first step, each OpenMP thread is responsible for matrix-vector and vector-vector multiplication of Equation (2.49). The implementation of the parallel section consists of executing one thread per element. Thus, the variables related to finite elements are defined as private; for example, the number of equations (NumberLocalDOFs) used in Code 5.1. On the other hand, the variables with a single memory address in the partition are considered shared;

for example, the element mass matrix table (ElementMassMatrices), used in Code 5.1.

The LAPACK and BLAS optimized linear algebra libraries are used in this parallel region. The LAPACK functions *dpptrf* and *dpptrs* solve the system of equation (2.47) by the Cholesky method. BLAS is used in the expressions $[K_e]\{u_e^{(n)}\}$ and $[M_e](\{u_e^{(n)}\} - \{u_e^{(n-1)}\})$. Finally, the local values for displacement and acceleration are mapped to the global solution vectors of the partition.

In this mapping, a concurrency problem was observed when assigning local solutions to the partition's global displacement and acceleration vectors since the global vectors are shared among threads. The *#pragma omp atomic* directive was used in the mapping operation from element to the global DOF vector. In this way, the global DOF vectors are updated atomically, avoiding the possibility of simultaneous writing by threads (OPENMP, 2018).

Exchanging of boundary solutions among neighboring partitions is performed only by the master thread. Synchronization is performed to send and receive data among neighboring partitions using the Algorithm 5.4.

Some parallel regions have been created so that each thread operates on degrees of freedom and not on elements. For example, in the step for weighting displacement and acceleration vectors by element measures, each thread calculates and stores each DOF's weighted value in the partition solution vectors.

The next parallel region is to map from the global solution to update the local solution vectors. At this stage, each thread operates by the element, as shown in Code 5.3. Analogously, each thread runs in parallel by element in the calculation of the internal load of Equation (2.59) using the BLAS library to perform matrix-vector multiplication according to Equation (2.59). At the end of the parallel region, the use of the *# pragma omp atomic* directive is necessary since the expression will be mapped again to the global internal load vector, which is shared among threads.

Code 5.3 – Mapping global to local after update solution among partitions.

```
1  . . .
2  //  For each degree of freedom of the partition ID
3  for (local_dof = 0; local_dof < NumberEquations; local_dof++) {
4      //  Gets the global dof.
5      global_dof = EquationsArray[local_dof];
6      //  Than the total number of free DOFs.
7      if (global_dof < NumberFreeDOFs) {
8          U^{t-Δt}_{e, local_dof} = U^t_{global_dof}
9          U^t_{e, local_dof} = U^{t+Δt}_{global_dof}
10         U̇^t_{e, local_dof} = U̇^{t+Δt}_{global_dof}
11         Ü^t_{e, local_dof} = Ü^{t+Δt}_{global_dof}
12     } // End if
13 }
```

```
14 . . .
```

In the next step, each thread is responsible for operations for each degree of freedom, calculating the residue vector given by the difference of the external and internal loads, see Equation (2.61). In the same operation, there is a linear interpolation between two external loading steps.

The CGD method is called by the master thread to solve Equation (2.47). In this method, parallel regions were added, associating each thread to a degree of freedom in the partition to compute matrix-vector and vector-vector products. In addition, another parallel region was created associating the threads for each element in the method MultiplicateElemByElem (...) of the **Solver** class. This method calculates the matrix-vector multiplication for each element in the CGD iterations. The threads for this method are also used in three steps: mapping between global to local DOFs of the partition; matrix-vector product executed with BLAS function *cblas_dspmv (...)*; and the sum in the global partition vector from the element product. In the latter, the directive *#pragma omp atomic*, because the elements associated with the threads may share the same degree of freedom of the global result vector. Although the master thread invokes the CGD method, good scalability results were obtained as presented in Section 6.7.2.

In the next parallel region, the element external load vector is updated for the next time step. First, each thread maps the residue computed by the CGD to an element vector of the partition. Then, the thread calculates the product between the element mass matrix and the residue mapped to the element with the BLAS *cblas_dspmv (...)* function. At the end of the parallel region, the resulting value per thread will be added to the element external load updating the load for the next time step. For this update, we use the directive "#pragma omp atomic" in each degree of freedom of the element.

In the step of calculating the energy norm for each loading step, parallelism is used with OpenMP and communication with MPI. The energy norm is calculated as

$$||u||_E = \sqrt{\{u^T\}[K]\{u\}}. \tag{5.1}$$

The parallelism with OpenMP is used to calculate the multiplication $[K]\{u\}$ using the method MultiplicateElemByElem(...). The *MPI_AllReduce()* function sums the results from each partition.

6 RESULTS AND DISCUSSION

This chapter presents the validation of the $(hp)^2$FEM parallel architecture and analysis of its performance for different cases. First, the computational environments used and their main features are described. Next, the validation of the projection element wise solver for uniform and non-uniform p distributions are considered. The main optimization aspects of the serial version are then stated. The results of the overlapping and other interface renumbering algorithms are presented considering weak and strong scalability for the projection solver. The last two sections consider the performance of the explicit element wise solver for beam and crankshaft meshes with the hybrid parallel implementation.

6.1 COMPUTATIONAL ENVIRONMENTS

This section presents the computational resources of the IBM Blue Gene/P and Q supercomputer architectures used in this work, as well as a comparison of the two environments. The computer systems are located at the Argonne National Laboratory (ANL), of the Department of Energy of the United States (ALCF.ANL.GOV, 2014). The IBM Blue Gene/Q is an evolution of the Blue Gene/P. This section also describes the tools used to optimize the $(hp)^2$FEM serial architecture described in Chapter 4 and evaluate the scalability of the parallel version presented in Chapter 5.

Another environment used was the Kahuna cluster located at the Center for Computational Engineering & Sciences (CCES) of the Institute of Chemistry at UNICAMP (CCES, 2020). CCES is one of the Research, Innovation, and Diffusion centers funded by the São Paulo Research Foundation (FAPESP).

6.1.1 Surveyor - IBM Blue Gene/P Solution

IBM Blue Gene/P Solution system (2007) has a total of 163 840 cores with a theoretical peak performance of 551 TFlops. In 2008, this supercomputer was the 4-th ranked in the Top 500 list with the 500 fastest computers in the world (MEUER *et al.*,). The IBM Blue Gene/P was split into 3 computational systems: Intrepid, Challenger, and Surveyor, each one with different purposes according to applications.

In the default configuration, the hardware architecture of the computational system has the following hierarchy:

- 1 Chip with 4 processors - 13.6 GF/s.
- 1 Computer card with 1 chip - 13.6 GF/s and 2.0 GB DDR memory.

- 1 Node card with 32 chips - 435 GF/s and 64.0 GB DDR memory.
- 1 Rack with 32 node cards adding up to 4 096 processors - 14 TF/s.

The Blue Gene/P has 72 racks. During the first phase of the $(hp)^2$FEM system profiling, the Surveyor computational system with 1 rack was used with just one node or computer card. The initial objective was to know the $(hp)^2$FEM performance and do some optimization.

The GNU and IBM compilers are available in the Blue Gene/P computer (STALLMAN, 2003). We employed the XL C/C++ Advanced Edition compiler for Blue Gene/P, V9.0 (WALKUP, 2011) with mpixlcxx for MPI. Other libraries used were METIS 5.0 for mesh partitioning, Lex 2.5.35 and Yacc 1.9 for the symbolic analyzer, Valgrind 3.8.1 for debugging, PAPI 3.9.0 and Gprof 2.20.51 for profiling, BLAS 3.0 and LAPACK 3.0 for linear algebra and solution of systems of equations.

6.1.2 Mira - IBM Blue Gene/Q, Power BQC Systems

Blue Gene/Q is part of the third generation of the IBM Blue Gene supercomputer family with a configuration of 786 432 cores, 768 memory terabytes, and a performance peak of 10 petaflops. The BG/Q was the 3-rd top-ranked in the Top 500 list (MEUER *et al.*,) and is currently the 22-th.

The BG/Q is also split in 3 computational systems: Mira, Cetus and Vesta, being Mira the system with greater performance. The main features of these systems are as follows:

- Mira (Production)
  - 49 152 computational nodes / 786 432 cores
  - 768 TB of memory
  - Peak performance of 10 petaFLOPS
- Cetus (Tests and Development)
  - 4 096 computational nodes with 65 536 cores
  - Peak performance of 838 teraFLOPS
- Vesta (Tests and Development)
  - 2 048 computational nodes with 32 768 cores
  - 32 TB of memory
  - Peak performance of 419 teraFLOPS.

Table 6.1 – Comparison of the nodes of the IBM Blue Gene/P and Q systems.

Hardware	**Blue Gene/Q**	**Blue Gene/P**
Processor	64 Bits Power A2	32 Bits PowerPC 450d
Floating point operations	4-way SIMD (QPX)	2-way SIMD
Interconnection topology	5D Torus	3D Torus + Tree
Clock speed	1600 MHz	850 MHz
Cores per node	16	4
Memory per node	16 GB	2 GB

Vesta was initially used and had the same architecture as Mira, but with two computational racks. Each rack is composed of 1 024 computational nodes; each node has a 1600 MHz PowerPC A2 processor with 16 cores and 16 GB of RAM, 1 GB of RAM for each core. The total number of cores in Vesta is 32 768. The communication between nodes is based on IBM 5D Torus interconnections with traffic capacity of up to 40 gigabits (ARGONNE, 2015).

The main tools utilized in the Blue Gene/Q were the IBM XL C/C++ Blue Gene/Q compiler V12.0 (IBM, 2012), Allinea DDT, and Coprocessor for debugging. Allinea DDT is a tool for large-scale parallel multi-thread applications used to enable debugging in clusters (ALLINEA, 2014). The Coprocessor is also useful when working with many cores since it indicates the cores dumped. It offers the opportunity of code debugging in all levels of hardware, kernel, and application.

Table 6.1 compares the computational nodes of the Blue Gene/P used in the optimization described in Section 6.3 and the Blue Gene/Q utilized to implement the parallel $(hp)^2$FEM .

The hardware architecture of a computational node in the IBM Blue Gene/Q is illustrated in Figure 6.1. As described previously, each computational node has 16 cores, with one program counter, Previous Program Counter (PPC), and one for redundant use. All processors are symmetric, and each one can simulate up to 4 threads using simultaneous multithreading (MARR, 2002).

Each core has one L1 cache for instructions and data, with 16 kB for instructions and 16 kB for data. From another cache, known as the L1 prefetch engine, comes an interface with another A2 Core processing unit (see (GILGE, 2014b)). There is also one L2 cache with 32 MB of memory. Besides 16GB of RAM, each computational node has a dual memory controller.

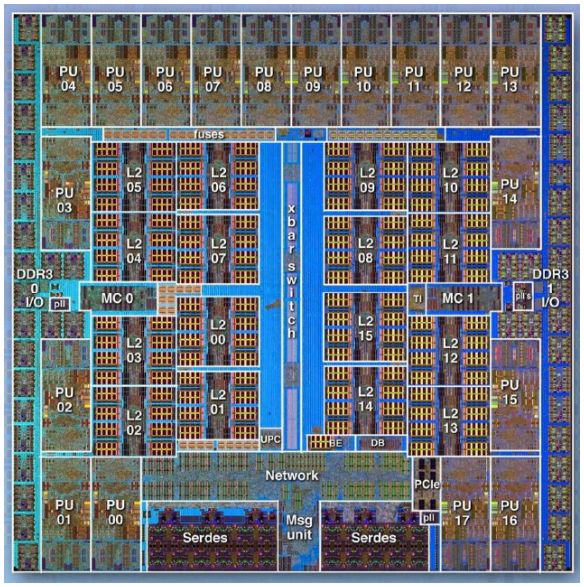


Figure 6.1. Compute node of the Blue Gene/Q architecture (Haring *et al.*, 2012).

6.1.3 CCES - Unicamp Kahuna computer cluster

The last performance tests of the $(hp)^2$FEM software were executed in the Kahuna cluster with Intel Xeon processors. The most important systems here utilized were

- 32 Graphic nodes with 20 cores HT Intel Xeon E5-2670 v2 - 2.50GHz, 64G of memory,
- 28 Graphic nodes with 24 cores HT Intel Xeon E5-2670 v3 - 2.30GHz, 64G of memory,
- 20 Compute nodes with 24 cores HT Intel Xeon E5-2670 v3 - 2.30GHz, 64G of memory.

All nodes have 2 threads per core. The first two sets have compute nodes with NVidia Tesla K20M and K40M boards, respectively. $(hp)^2$FEM software was compiled with the Intel icpc 19.0.4 compiler. In addition, the libraries PAPI 5.5.1 and Intel(R) VTune(TM) Amplifier XE 2015 were used to profile the serial and parallel code, METIS 5.1.0 for partitioning the finite element mesh, OPENMPI-1.8.3 as MPI version, MKL 11.2.0, BLAS and LAPACK to optimize the linear algebra operations.

6.2 VALIDATION OF THE ELEMENT WISE PROJECTION SOLVER

The local projection solver described in Chapter 2 was considered to validate the p-non-uniform architecture of the $(hp)^2$FEM software. The L_2-error norm of the approximated solution was calculated for different polynomial orders p for square and hexahedron meshes.

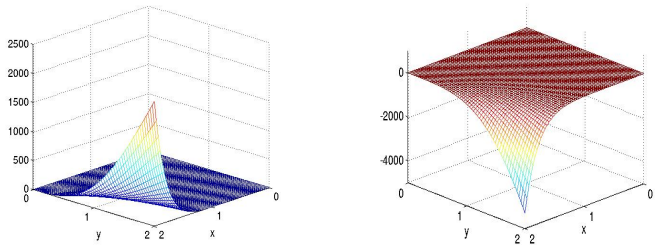
The analytical solutions used for square and hexahedron meshes are plotted in Figures 6.2(a) and 6.2(b), respectively, and given by

$$u(x,y) = \exp(\pi x)\sin\left(\frac{\pi y}{4}\right)(x-1)^2 y^2, \tag{6.1}$$

$$u(x,y,z) = \exp(\pi x)\sin\left(\frac{\pi y}{4}\right)(x-1)^2 y^2 (z-xy). \tag{6.2}$$

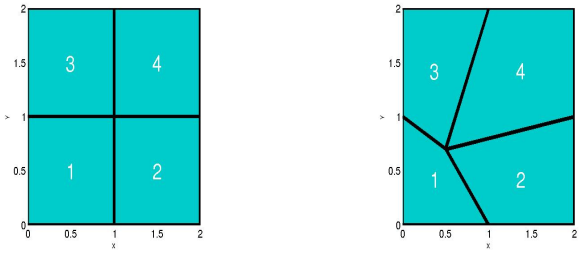
The domains of these functions are $0 \leqslant x,y \leqslant 2$ and $0 \leqslant z \leqslant 1$.

The meshes are illustrated in Figures 6.3 and 6.4. The p-non-uniform strategy described in Section 2.4 was applied using Lagrange polynomial basis, Gauss–Lobatto collocation points, and Gauss–Legendre integration points (KARNIADAKIS; SHERWIN, 2005; BITTENCOURT, 2014).



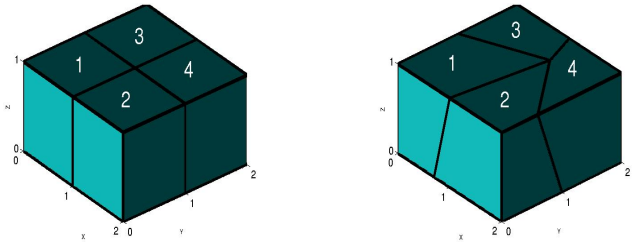
(a) 2D function. **(b)** 3D function for $z = 0$.

Figure 6.2. Functions to be approximated by the element wise projection solver.



(a) Non-distorted mesh. **(b)** Distorted mesh.

Figure 6.3. Square meshes for the validation of the element wise projection solver.

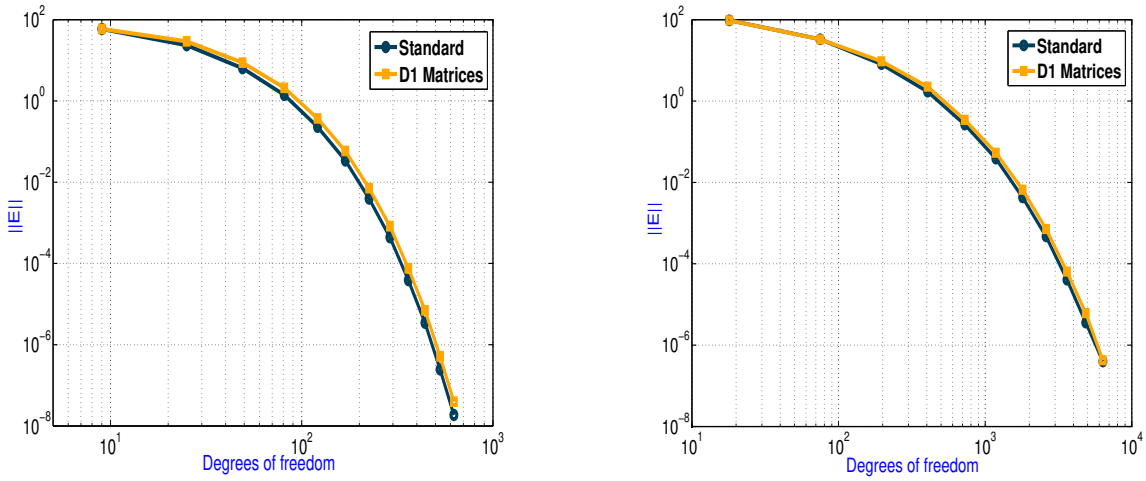


(a) Non-distorted mesh **(b)** Distorted mesh.

Figure 6.4. Hexahedron meshes for the validation of the element wise projection solver.

Figure 6.5a shows the convergence of the L_2-error norm for the square mesh according to the number of degrees of freedom for polynomial orders from 1–12. The convergence rate

improves exponentially concerning the polynomial order, as expected for the HO-FEM (KARNIADAKIS; SHERWIN, 2005; BITTENCOURT, 2014). Note that the errors are larger for distorted meshes. The hexahedron meshes gave an error norm similar to that of square meshes for polynomial orders from 1 to 11, as shown in Figure 6.5b.



(a) L_2-error norm for square meshes.

(b) L_2-error norm for hexahedron meshes.

Figure 6.5. L_2-error norm for the square and hexahedron meshes to validate the element-wise projection solvers. *STANDARD* uses the two- or three-dimensional mass matrices and $D1-MATRICES$ is a tensor product of one-dimensional mass matrices, as described in Section 2.2.

The p-non-uniform distribution allows us to use different polynomial orders for the element shape functions. The behavior of the p-non-uniform approximation was analyzed with the functions given in Equations (6.1) and (6.2). In the first case, the polynomial order that obtains the best convergence of the p-uniform solution was used as the reference result for the p-non-uniform analyses. The error norms obtained for the p-uniform cases were 1.8846×10^{-8} and 3.7999×10^{-7} for the 2D and 3D meshes, respectively.

Tables 6.2 and 6.3 present the calculated errors for eight p-non-uniform distributions on non-distorted square and hexahedron meshes, (Figures 6.3(a) and 6.4(a), respectively) and the polynomial orders used for each element. The error results are similar for these combinations, which indicates the possibility of decreasing the polynomial order for certain mesh elements.

The polynomial order combinations of Tables 6.2 and 6.3 were selected according to the function behavior shown in Figures 6.2(a) and 6.2(b), respectively. These functions exhibit high gradient variation near the domain boundary. These boundaries, as shown in Figures 6.2(a) and 6.2(b), are in the intervals $1 \leqslant x, y \leqslant 2$. Thus, the largest polynomial order is used on the boundary with the highest gradient (in this case, on element 4). Consequently, lower polynomial orders were considered for the other elements. The convergence results are close to those from the p-uniform solution.

Table 6.2 – Square mesh with p-non-uniform polynomial distribution.

	Elem. 1	Elem. 2	Elem. 3	Elem. 4	$\|e\|_{L_2}$
1	13	13	13	13	1.8846e-08
2	12	12	13	13	1.9661e-08
3	11	11	13	13	8.2871e-08
4	12	12	12	13	1.9693e-08
5	11	11	11	13	8.4874e-08
6	11	11	12	13	8.2876e-08
7	11	12	12	13	1.9863e-08
8	10	11	12	13	8.4268e-08

Table 6.3 – Hexahedron mesh with p-non-uniform polynomial distribution.

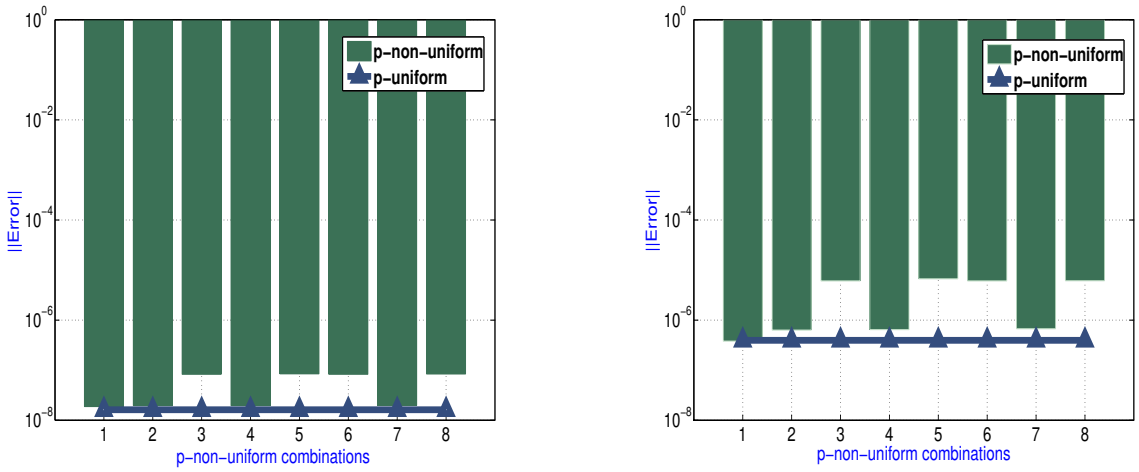
	Elem. 1	Elem. 2	Elem. 3	Elem. 4	$\|e\|_{L_2}$
1	11	11	11	11	3.7999e-07
2	10	10	11	11	6.3310e-07
3	9	9	11	11	6.0686e-06
4	10	10	10	11	6.4565e-07
5	9	9	9	11	6.6834e-06
6	9	9	10	11	6.0697e-06
7	9	10	10	11	6.7297e-07
8	8	9	10	11	6.1115e-06

The L_2-error norms for the uniform and non-uniform distributions are shown in Figures 6.6(a) and 6.6(b). The plots exhibit the errors calculated for the p-uniform case and eight combinations of polynomial orders for the p-non-uniform approach. The polynomial orders were decreased from 11 to 8 for the first element. A comparison between the p-uniform and p-non-uniform cases shows that, even with a lower polynomial order in certain elements, the error norm's magnitude remains the same as the best case with the p-uniform distribution.

An error estimator should be used to obtain an optimal distribution of element orders. The example here presented had the objective only to validate the architecture of $(hp)^2$FEM for non-uniform order distribution.

6.3 SERIAL CODE OPTIMIZATION

This section describes the performance and memory consumption improvements obtained by optimizing the $(hp)^2$FEM serial code. We performed experiments using a mesh of 16 hexahedrons and polynomial order 10, with 18 801 degrees of freedom. The software was executed on the IBM Blue Gene/P System, which runs the CNK/SLES 9 Linux operating



(a) Square mesh. **(b)** Hexahedron mesh.

Figure 6.6. Comparison between p-non-uniform and p-uniform strategies.

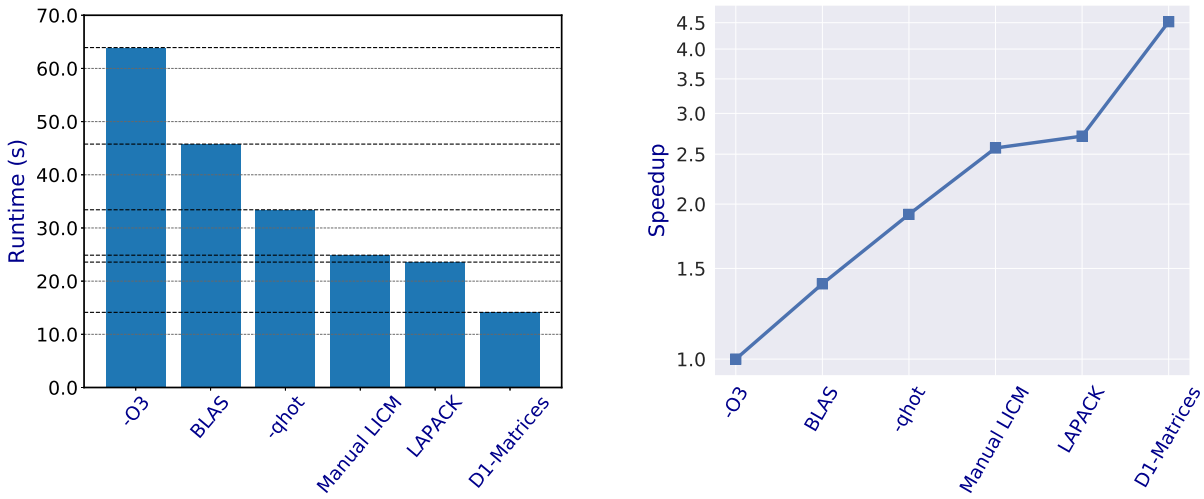
system on 860 MHz 4-core PowerPC 450d processors. Each node has 2.0 GB of memory. We used the open-source version of BLAS and LAPACK (version 3.0) from NetLib (DONGARRA, 2003). We compiled the code with XL C/C++ Advanced Edition for Blue Gene/P, V9.0, using the following flags:

- -qtmplparse=warn: to check warnings for semantic errors.
- -qlanglvl=stdc89: used for compilation, conforming the ANSI C89 standard, also known as ISO C90.
- -O3: applied to compiler optimization level 3.

$(hp)^2$FEM was executed three times, and the average runtime was used to determine performance improvement.

Figure 6.7 shows the execution times and speedup for each optimization step. The first one considered the BLAS library and reduced the runtime by 39%. The next optimization forced the IBM XLC compiler to perform more aggressive loop unrolling using the flag "-qhot" (GILGE, 2014a). This option reduced the execution time by 36% compared to the previous version optimized with BLAS. We also did modifications in the **Mapping** class of the **FEGroups** package, which is responsible for generating the Jacobian matrices. We moved loop-invariant computations out of loops and enabled loop unrolling, which accelerated the code 1.82 times. LAPACK library was used to solve a linear system of equations using Gaussian elimination, reducing the runtime by 5% only. Finally, the D1-Matrices procedure improved the performance twice over the previous optimization. In the end, the total runtime was reduced to 4.52 times.

In addition to improving the execution time, the D1-Matrices procedure reduced the memory consumption required by shape functions. For a mesh with 16 hexahedrons and polynomial order 13, the total memory consumed by **ShapeFunctions** objects were approximately



(a) Runtime. **(b)** Speedup.

Figure 6.7. Evaluation for the $(hp)^2$FEM serial code.

four times less with the D1-Matrices procedure. The overall reduction in memory consumption was from 1.63 GB to 0.41 MB.

6.4 EVALUATION OF THE OVERLAPPING ALGORITHM

This section analyzes the strong scalability of the overlapping algorithm described in Section 5.4.1 with the element wise projection solver of Section 2.3. We conducted speedup and efficiency tests using the IBM Blue Gene/Q architecture on the Mira platform.

A mesh of 10 000 hexahedron was used for polynomial orders 1, 3, 6 and 9, reaching up to 7 403 986 degrees of freedom for the approximation of function (6.1) . The solver was executed using MPI communication only for the mesh partitioning.

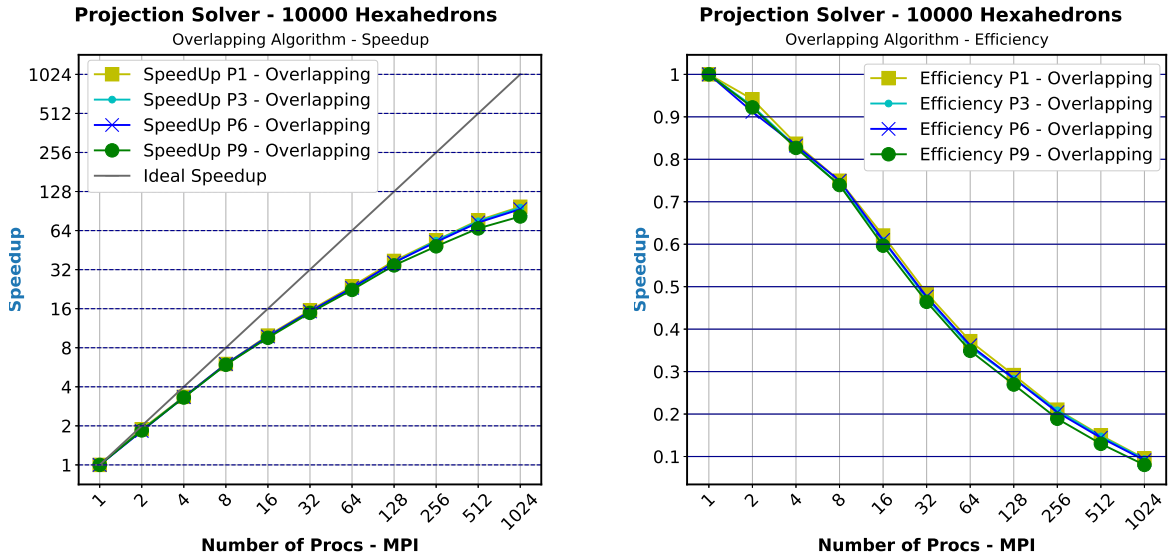
It is important to mention that the solver is executed in two phases: one for solving the finite element systems of equations for the duplicate region and the other for the internal elements of each partition. Thus, the overlapping algorithm avoided solver communication by duplicating information about interface elements of neighboring partitions.

The overlapping algorithm proved non-scalable due to the computation growth when increasing the number of partitions or neighbors of each partition. Figure 6.8(a) shows the optimal speedup and the speedups for polynomial orders 1, 3, 6 and 9. The speedup remains the same when increasing the polynomial order from 1 to 9.

In summary, the overlapping algorithm has a near-optimal speedup for up to 8 processors out of the 1024 MPI processes evaluated. The same observation is valid for Figure 6.8(b), where the efficiency up to 8 MPI processes was 75% and 74% for polynomial orders 1 and 9, respectively. Moreover, efficiency drops linearly by doubling the number of processors.

Comparing with the versions used in Section 6.6, the scalability of the overlapping

algorithm is much lower varying the polynomial order, even though there is no communication of MPI processes. Thus, the increase in computation within the partition due to the number of neighbors makes it a non-efficient parallel version.



(a) Speedup. **(b)** Efficiency.

Figure 6.8. Performance of the element wise projection solver with the overlapping algorithm and a mesh of 10000 hexahedrons.

6.5 COMPARISON OF NUMBERING ALGORITHMS OF INTERFACE NODES

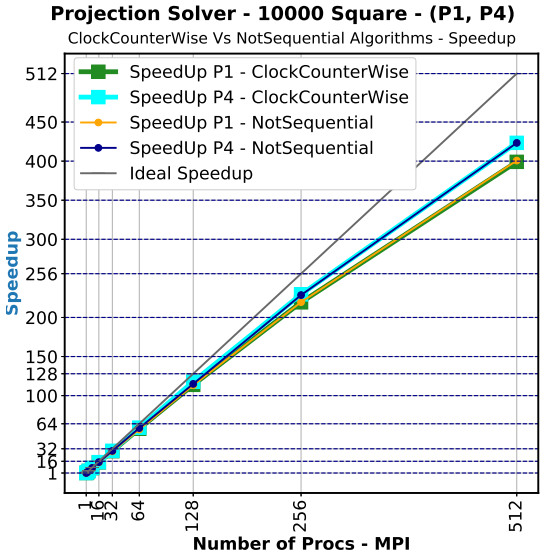
The ClockWiseCounter and NonSequential renumbering algorithms presented in Sections 5.4.2 and 5.4.3, respectively, were compared. Their performance were evaluated for the element wise projection solver with a 10000 square mesh and polynomial orders 1, 4, 6 and 9. The considered function for the projection solver approximation is given in Equation (6.1). The Blue Gene/Q computer was used for all results presented in this section.

We used the same serial code to evaluate scalability for both algorithms with one MPI process by compute node. The main differences of the algorithms are the message size exchanged by MPI point-to-point communication and the update of interface solutions. The ClockCounterWise algorithm uses the first index and size of the neighbor boundary solution array to update the interface solutions. The NonSequential algorithm exchanges all partition boundary solution array.

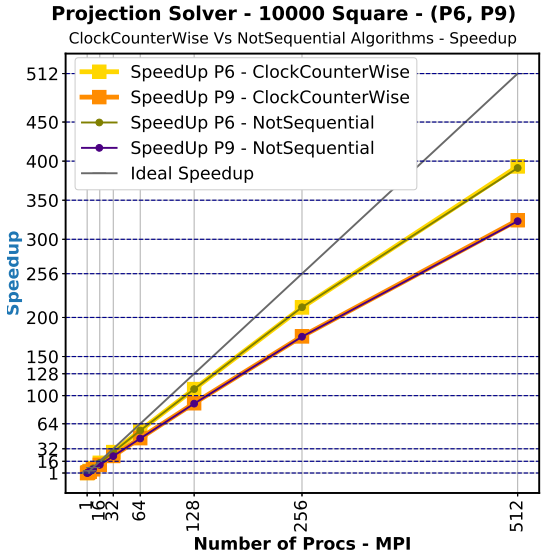
Figures 6.9(a) and 6.9(b) show the speedup of the ClockWiseCounter and NonSequential algorithms for polynomial orders 1, 4, 6 and 9. The speedup of the ClockWiseCounter algorithm for polynomial orders 1 and 4 is closer to ideal. The overall scalability for the considered polynomial orders is similar for both algorithms.

Figure 6.9(c) presents the efficiency measures for both algorithms and the considered polynomial orders. They remained almost constant up to 256 processors and polynomial

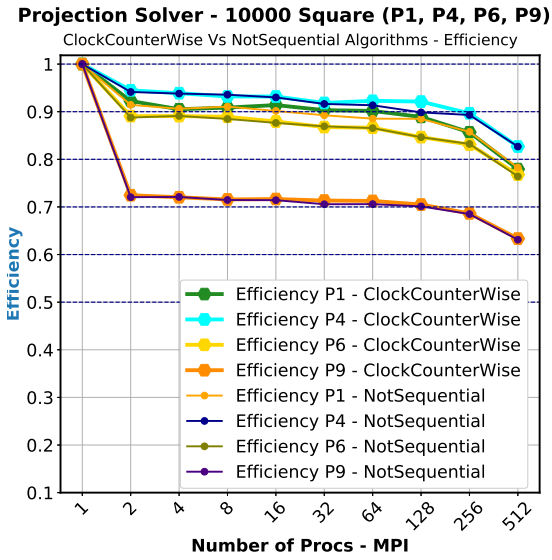
orders 1, 4 and 6. The worst value was 63% for polynomial order 9. The ClockWiseCounter algorithm exchanges less messages than the NonSequential algorithm. The overhead is higher for more than 256 partitions.



(a) Speedup ($P1$, $P4$).



(b) Speedup ($P6$, $P9$).



(c) Efficiency.

Figure 6.9. Comparison of the Non-Sequential and ClockWiseCounter renumbering algorithms for square mesh of 10000 elements.

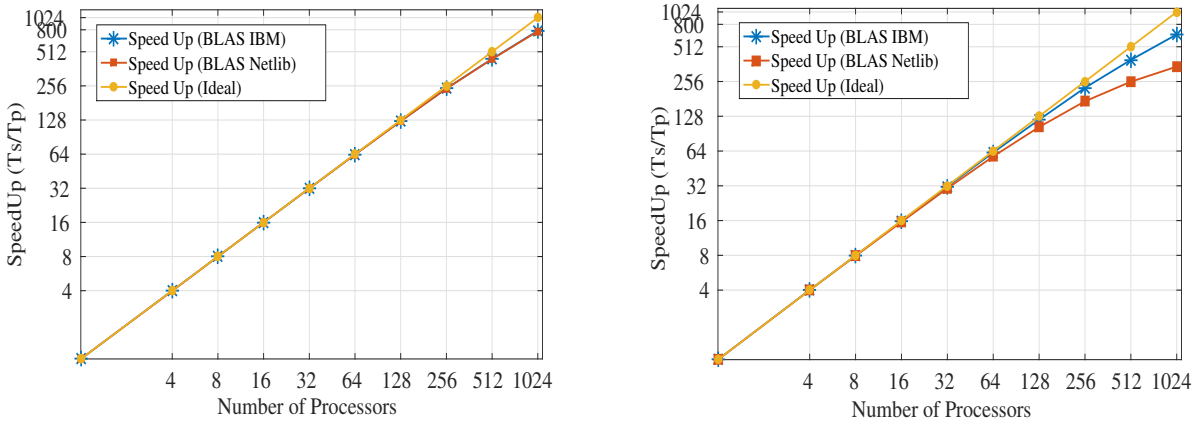
6.6 NON SEQUENTIAL NUMBERING ALGORITHM WITH HIGH-ORDER DATA GENERATION

This section considers the element wise projection solver with a mesh of 10000 hexahedrons and polynomial orders 1, 3, 6, and 9, reaching $7,403,986$ degrees of freedom for $P = 9$. We compared its performance using IBM BLAS library and open source Netlib BLAS (IBM - INTERNATIONAL BUSINESS MACHINES CORPORATION, 2012; DONGARRA, 2003). The non sequential numbering algorithm for the partition boundary nodes

with high-order data generated in the **PartitionModel** class of Section 5.4.4 is considered for all results presented here.

6.6.1 Strong scalability

Figures 6.10(a) and 6.10(b) illustrate the strong scalability with up to 1024 processors using IBM and Netlib BLAS libraries. Similar speedups were obtained for both libraries with polynomial order 1 as shown in Figure 6.10(a). However, as the polynomial order increased, the scalability decreased, as illustrated in Figure 6.10(b). The code version with IBM BLAS achieved better scalability when compared with the Netlib version.

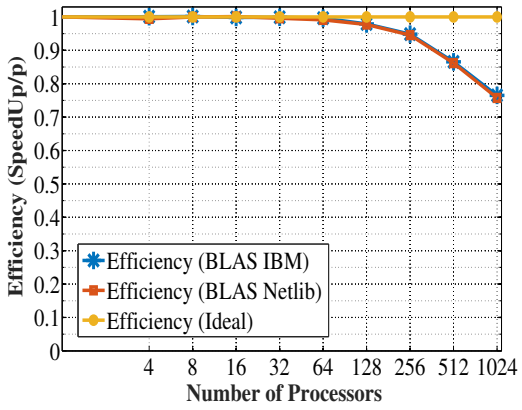


(a) Polynomial order = 1 (11466 DOFs). **(b)** Polynomial order = 9 (7403986 DOFs).

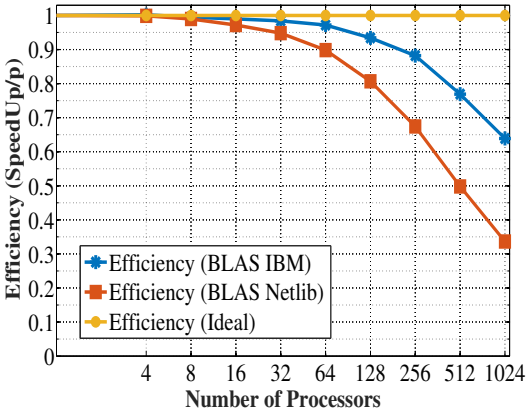
Figure 6.10. Speedup of the element wise projection solver with a mesh of 10000 hexahedrons using the IBM and Netlib BLAS libraries.

Figure 6.11 shows the efficiency measure with respect to the number of processes. Similarly to the speedup test, IBM BLAS achieved better efficiency than Netlib BLAS. The worst efficiency of 65% was obtained with polynomial order 9 and 1024 MPI processors as shown in Figure 6.11(b). Netlib obtained an efficiency of 34% for the same case. Netlib BLAS code version took 1209.59 s for one processor and 3.51 s with 1024 processors. In contrast, the IBM BLAS code version spent 1210.57 s with one processor and 1.85 s for 1024 processors. IBM version obtained better efficiency because it is optimized for the Blue Gene/Q architecture.

Figure 6.12 shows the CPU time percentage spent by MPI communication routines concerning the number of processes for polynomial orders 3, 6, and 9. It may be noted that the decrease in speedup and efficiency observed in the previous tests is directly related to point-to-point communication cost as the number of neighboring partitions and problem size increases. For $P = 9$ and 7403986 degrees of freedom, MPI communication took 8% of the total $(hp)^2$FEM runtime.



(a) Polynomial order 1 (11466 DOFs).



(b) Polynomial order 9 (7403986 DOFs).

Figure 6.11. Parallel efficiency of the element wise projection solver with a mesh of 10000 hexahedrons using IBM and Netlib BLAS libraries.

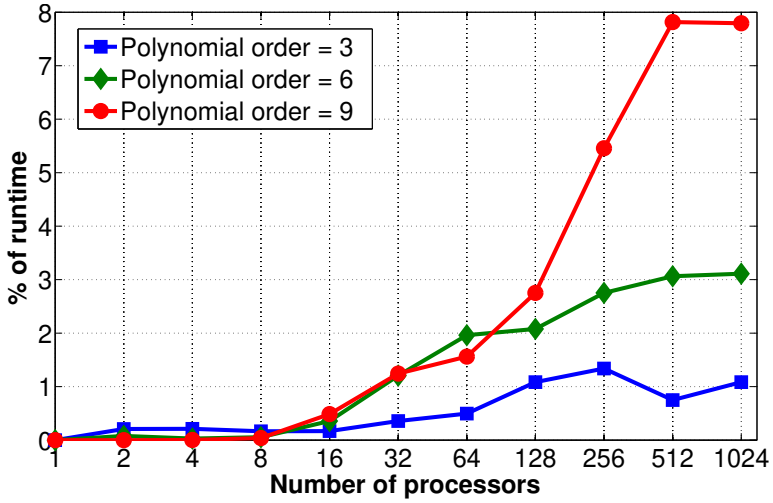


Figure 6.12. Percentage of execution time spent on MPI communication for the element wise projection solver with a mesh of 10000 hexahedrons.

6.6.2 Weak Scalibility

We used eleven meshes ranging from 1000 to 1024000 elements to evaluate the weak scalability of $(hp)^2$FEM . The number of processors was increased proportionally from 1 to 1024 nodes. The results obtained are illustrated in Figure 6.13 for polynomial orders 1, 3, 6 and 9.

As expected, the runtime increased with higher polynomial orders. However, the elapsed time measured for each polynomial order exhibited similar behavior as the number of processors and elements increased. Figure 6.14 displays the efficiency calculated based on the ideal sizeup given in Equation (3.5). The efficiency decreases as the number of processors increases. Profiling results indicate that this fact comes again from the increasing of communication time. Even though the efficiency decreases to 80% after 64 processors, it remained constant when running with more processors for all experiments. Hence, the parallel version of

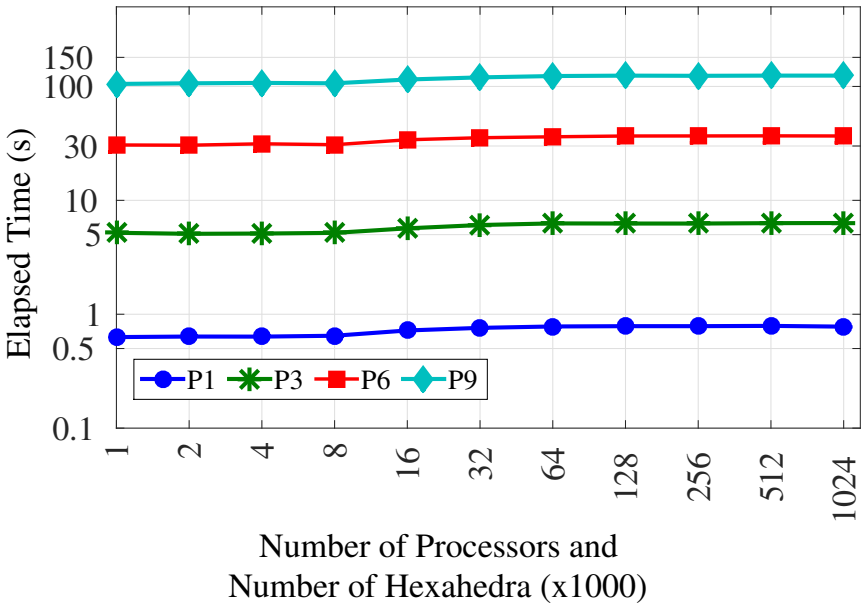


Figure 6.13. Weak scalability of the element wise projection solver.

$(hp)^2$FEM shows excellent weak scalability for the considered example.

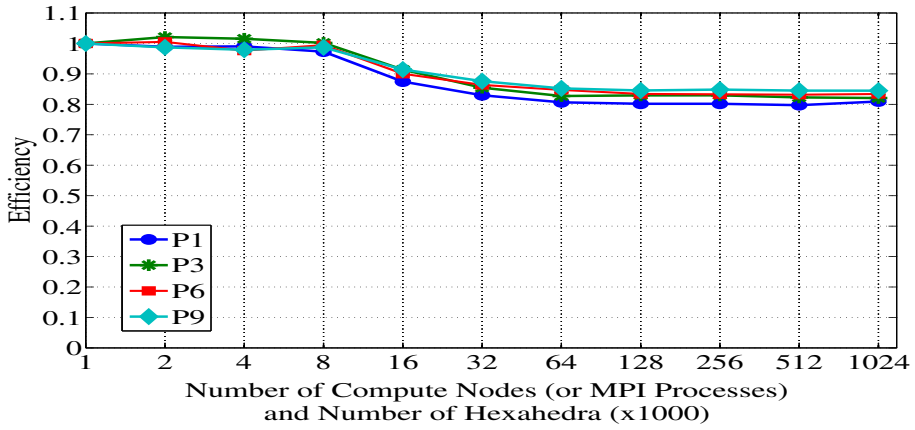


Figure 6.14. Efficiency of weak scaling analysis of the element wise projection solver.

6.6.3 Use Case with 32768 nodes on IBM Blue Gene / Q

We also evaluated the strong scalability on the production environment using 2/3 of compute nodes available in the IBM Blue Gene/Q Mira. The projection problem was solved using 1024000 hexahedrons and polynomial order 9, which corresponds to 840 millions of unknowns. Figure 6.15 illustrates the speedup and efficiency when running up to 32768 compute nodes.

We can see that the speedup was close to ideal, reaching 26436 in the worst case for 32768 processors, and an efficiency of more than 80%. The total computation time was 4.6 s. For comparison purposes, the estimated time using a single processor is approximately 35 h.

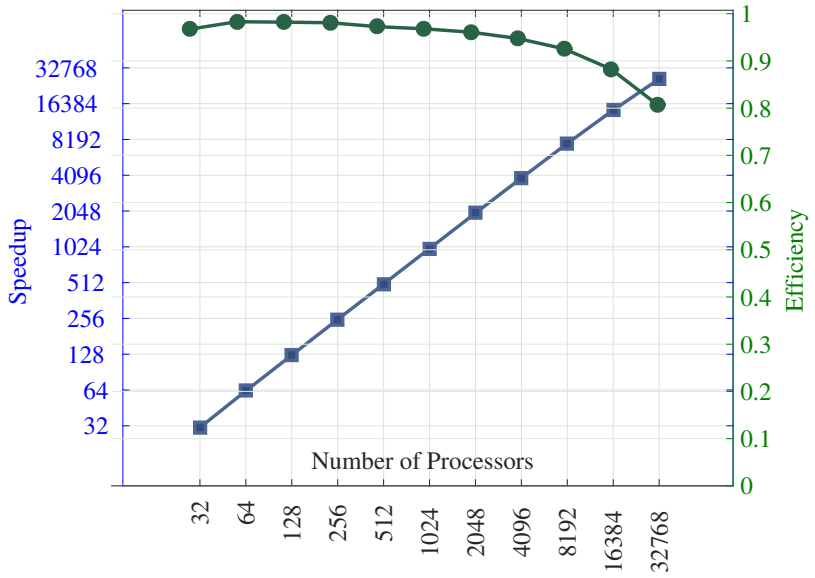


Figure 6.15. Speedup and efficiency with 32768 computing nodes of the IBM Blue Gene/Q Mira processors.

6.7 PARALLEL PERFORMANCE FOR THE ELEMENT WISE CENTRAL DIFFERENCE METHOD FOR STRUCTURED MESHES

This section presents the parallel performance for a structured mesh of a beam problem. First, we consider only OpenMP applied to the conjugate gradient method with diagonal preconditioner (CGD) of the element wise linear transient solver. Second, we consider MPI and OpenMP applied to the linear problem and show the parallel performance of the solver and the matrix-vector operations implemented in CGD. At last, we repeat the previous analysis for the non-linear large displacement problem. OpenMP is implemented only in the matrix-vector operations for each element into the CGD.

6.7.1 Evaluation of multi-threading procedures

The element wise central difference method described in Sections 2.6 and 5.6 was used to implement the multithreaded parallelism procedures of $(hp)^2$FEM software. The main OpenMP procedures are related to linear algebra operations, such as matrix-vector multiplication.

We use a beam of length 10 m in the longitudinal direction x ($0 \leq x \leq 10$) and square cross-section of 1 m ($0 \leq y,z \leq 1$). All nodes are clamped at the left end ($x = 0$) and a face load of intensity -1 in the y direction applied to elements located at the right end ($x = 10$). We considered Hookean material for the linear problem. The material properties considered are Young's modulus $E = 1000\ Pa$, Poisson ratio $\nu = 0.3$ and density $\rho = 1.0\ Kg/m^3$. The finite element mesh with 8192 hexahedrons illustrated in Figure 6.16 is used for the results presented in this section.

Table 6.4 – Runtime for the solution of the linear equation system with OpenMP version of the element by element CGD method used in the explicit transient analyses of the beam example.

Number of threads	Elapsed Time (s)		
	31 104 DOFs	221 952 DOFs	1 672 704 DOFs
1	3.4375	24.3476	505.9082
2	1.7608	12.2524	250.5481
4	0.8818	6.1521	125.0179
8	0.6472	3.2422	63.2576
16	0.4073	2.3280	36.4629
32	0.2498	1.1931	21.8032

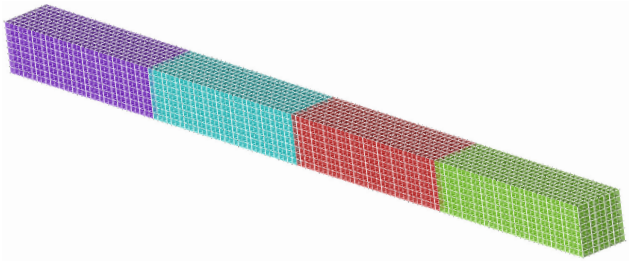


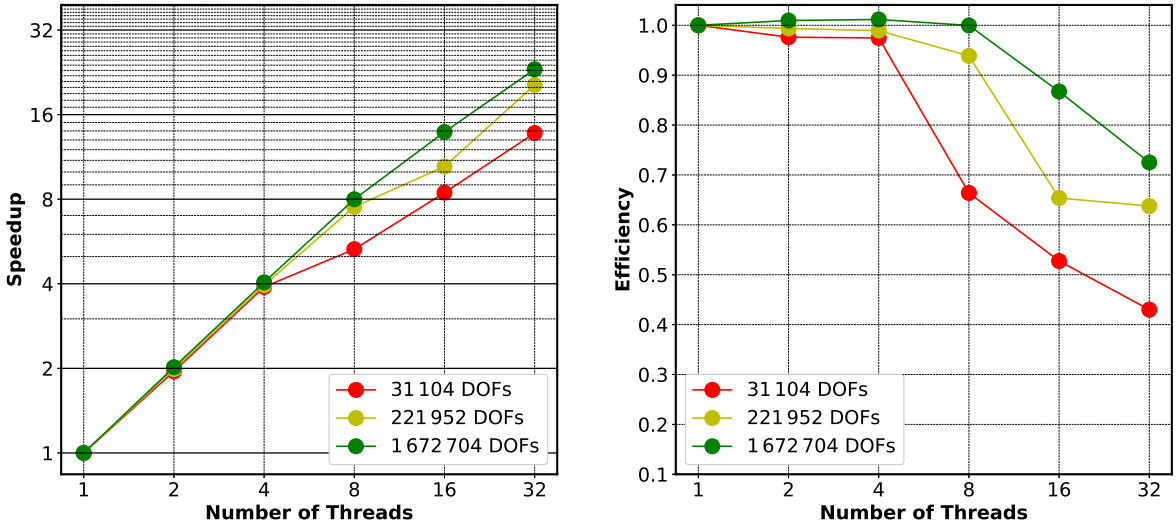
Figure 6.16. Beam mesh with 8 192 hexahedrons.

Table 6.4 presents the elapsed time in seconds for the solution of the linear equation system using the iterative conjugate gradient method with diagonal preconditioner (CGD), implemented in an element by element fashion, to solve for the residue in Equation (2.61). This method takes about 52% of execution time for all transient analyses here considered. Table 6.4 considers polynomial orders 1, 2 and 4 with 31 104, 221 952 and 1 672 704 degrees of freedom, respectively. Tolerance of 10^{-10} and maximum number of 10 000 iterations were considered for the CGD method and 10 time steps for the explicit transient analysis.

The analyses were performed in the Kahuna cluster with the mpiicpc 19.0.4 compiler and the optimization flags '-O3', '-ffastmath', and '-mavx2'. Other optimizations were applied to BLAS, MKL, LAPACK, inline methods, and loop-invariant computations not applied by the compiler similarly to Section 6.3. We also used the parallel procedure for non sequential numbering of interface nodes presented in Section 5.4.4.

Figures 6.17(a) and 6.17(b) show the speedup and efficiency for the results of Table 6.4. When the number of DOFs increased, the speedup improved due to the increasing amount of matrix-vector computation in each finite element, which is computed in a thread. The speedup with 1 672 704 DOFs was 13.9, using a maximum of 16 threads. Although each compute node has 20 cores, we use up to 32 threads to evaluate the hyper-threading performance. In this case, the hyper-threading had a gain of 30.9% for 20 cores. The result closest

to the ideal speedup occurred with 8 threads. The minimum efficiency was 72% and decreased with 8 threads because of synchronization overhead.



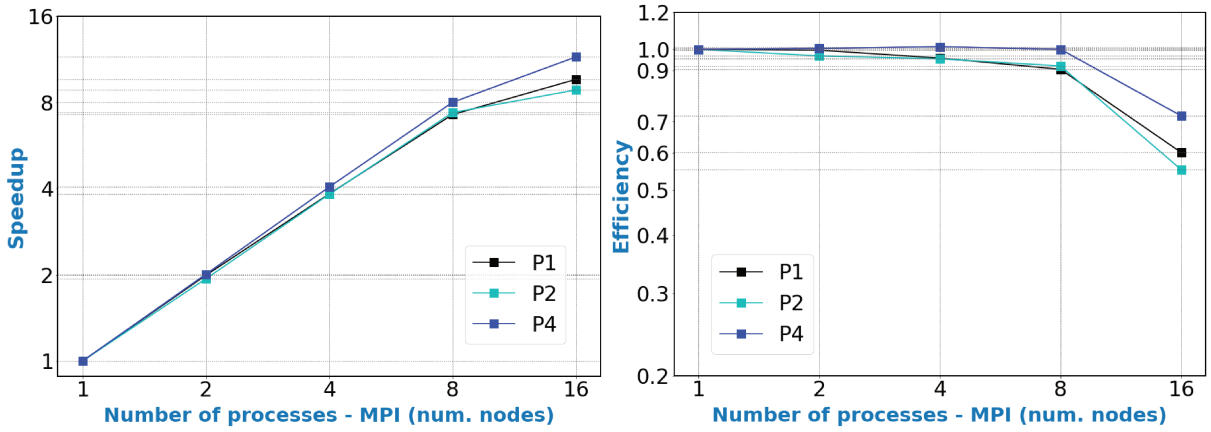
(a) Speedup. **(b)** Efficiency.

Figure 6.17. Speedup and efficiency of the multiplication methods used in the CGD method for the linear explicit transient analyses of the beam example.

6.7.2 Scalability of the hybrid element wise transient solvers applied to beam analyses

The scalability results for the MPI+OpenMP hybrid parallelism are here presented for the non-linear and linear transient analyses of the previous beam using shared and distributed memory. We considered Hookean and Neo-Hookean materials for the linear and non-linear analyses, respectively. The material properties considered are Young's modulus $E = 1000\ Pa$, Poisson ratio $\nu = 0.3$ and density $\rho = 1.0\ Kg/m^3$. The results were run in the HT Intel Xeon E5-2670 v2 - 2.50GHz machine of the Kahuna cluster described in Section 6.1.3. We used up to 16 compute nodes and 1 to 20 cores for each node; 1 MPI process per node and 1 or 2 OpenMP threads for each core. As stated in the previous section, the OpenMP implementation was applied for 52% of the running time of the profiled serial solver. There are other loops in the code where iterations cannot be executed independently.

We use all optimization options applied to the serial $(hp)^2$FEM version as discussed in Section 6.3 with the Intel compiler and optimization flags '-O3', '-ffastmath' and '-mavx2'. Figure 6.18 shows the scalability of the linear central difference element wise method for the beam analysis. The results for MPI-only are given in Figures 6.18(a) and 6.18(b) with 1 process per node. The results for MPI + OpenMP are illustrated in Figures 6.19(a) and 6.19(b) for polynomial orders 1, 2 and 4. We considered 1 MPI process for each compute node and many OpenMP processes to make better use of shared memory for the HO-FEM.



(a) Speedup MPI-only.

(b) Efficiency MPI-only.

Figure 6.18. Speedup (a) and efficiency (b) of the MPI parallelism applied to the **linear** explicit transient analysis of the beam problem using $[1, 16]$ compute nodes and 1 MPI process for each node.

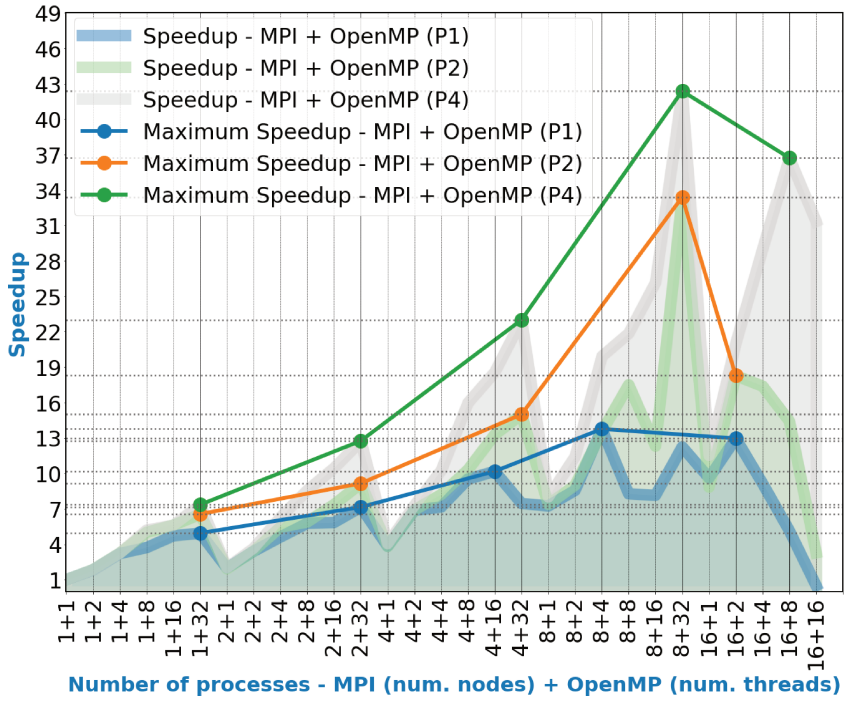
The speedup using MPI-only is closer to ideal with an efficiency of 72% for polynomial order 4, as illustrated in Figure 6.18. All parts of the local linear solver are parallelized with MPI, while OpenMP is implemented in about 52% of the running time of the central difference local solver considering the profile for one process. Figure 6.18(b) shows that higher-order approximation has better efficiency with 72%, when compared to the low order with an efficiency of 55%.

Although OpenMP was implemented in a solver portion which corresponds to only 52% of running time, the combination MPI + OpenMP reduces the elapsed time from 3.43 to 0.03 seconds, 24.34 to 0.13 seconds and 505.91 to 2.40 seconds per rime step, for polynomial orders 1, 2 and 4, respectively. The best cases of time reduction for polynomial orders 1 and 2 were with 8 MPI processes and 32 OpenMP threads. For polynomial order 4, the best execution time was for 16 MPI processes and 16 OpenMP threads. The MPI-only approach reduced the run time from 3.43, 24.34 and 505.91 to 0.30, 2.14 and 31.22 seconds for polynomial orders 1, 2 and 4, respectively, with 16 MPI processes. Consequently, adding OpenMP for each MPI process allowed reducing the running time at a rate from 10 to 16.

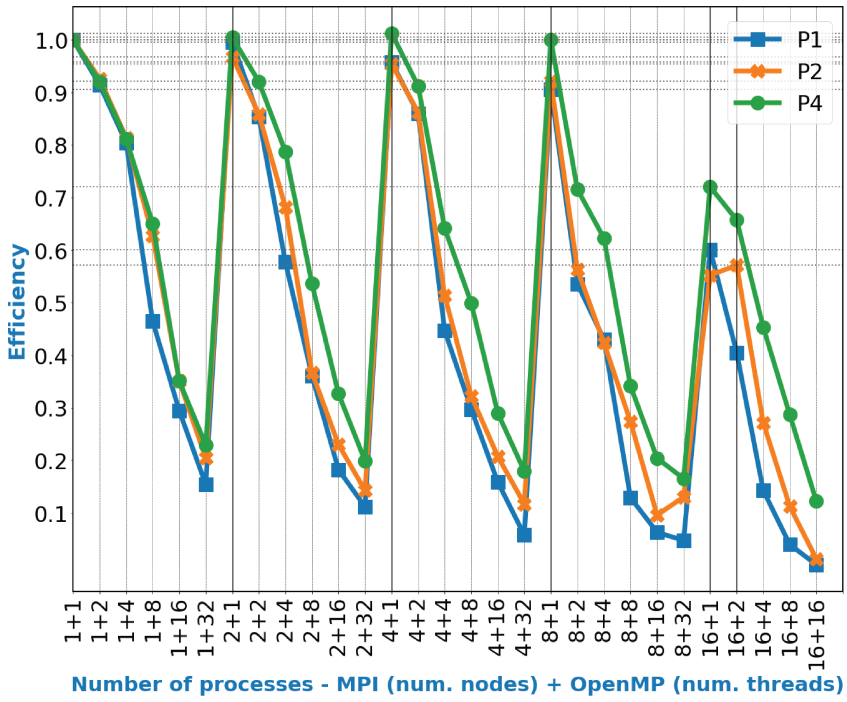
Other data that should be considered are the average number of elements in each partition. For instance, using the global mesh of Figure 6.16 and 16 MPI processes, the average number of elements per partition is 512. In this case, the scalability decreases with low-order approximation since the communication time with neighbor partitions dominates over the computation time in the subdomain.

In summary, the speedup for the linear central difference local method of the beam problem has scalability for the combination MPI + OpenMP close to ideal up to 4 OpenMP threads for each MPI process. This behavior occurs because 48% of the running time of the serial code profiled is not parallelized with OpenMP. Therefore, the number of CPU operations for

the serial portion will be higher than the parallel region as the number of processors increased.



(a) Speedup MPI + OpenMP.

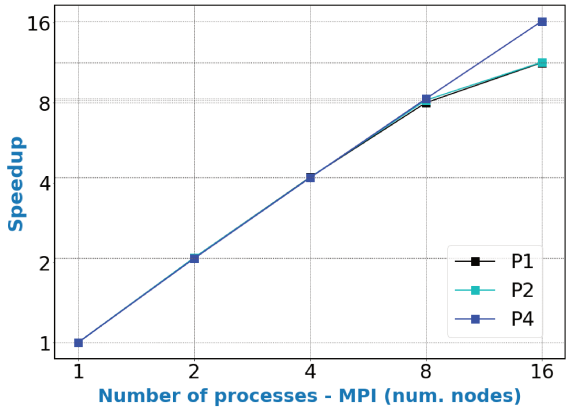


(b) Efficiency MPI + OpenMP.

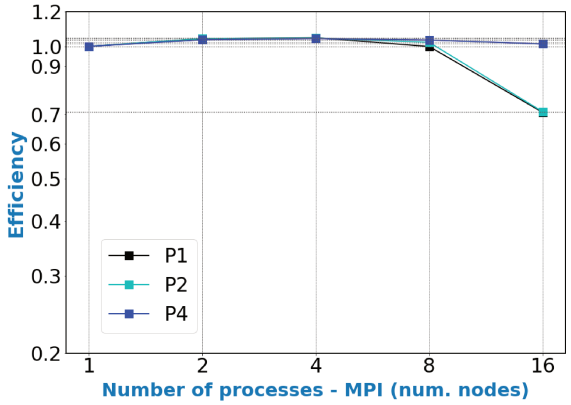
Figure 6.19. Speedup (a) and efficiency (b) of the hybrid parallelism applied to the **linear** explicit transient analysis of the beam problem for MPI + OpenMP using $[1,16]$ compute nodes with 1 MPI process for each node and $[1,20]$ cores with 1 or 2 threads per core.

Considering only the parallel region of the matrix-vector operations for each finite element implemented with OpenMP in the linear central difference local method, the speedup

is close to ideal for all number of processes considered. This occurs due to the OpenMP parallel region is useful in this code portion. The total running time of the solver is 4.74, 35.74 and 1047.90 seconds for polynomials orders 1, 2 and 4, respectively; and the running time of the matrix-vector operations code portion is 3.44, 24.35 and 505.91 seconds for the same polynomial orders. Figures 6.20(a) and 6.21(a) shows the speedup for the matrix-vector operations for polynomial order 1, 2 and 4, where each thread computes one finite element. The best speedup was obtained for polynomial order 4 with 210 for 256 processes with 16 MPI ranks and 16 OpenMP threads. Figure 6.20(b) and 6.21(b) shows the efficiency calculated based on this speedup, obtaining 82% with 256 processes (16 nodes and 16 cores).

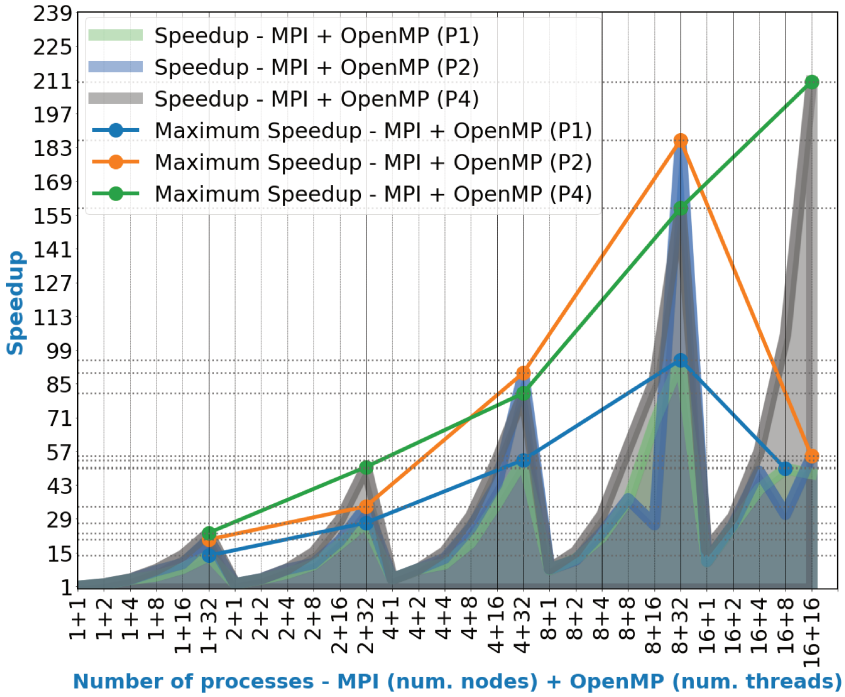


(a) Speedup MPI-only for the matrix-vector operations for each finite element.

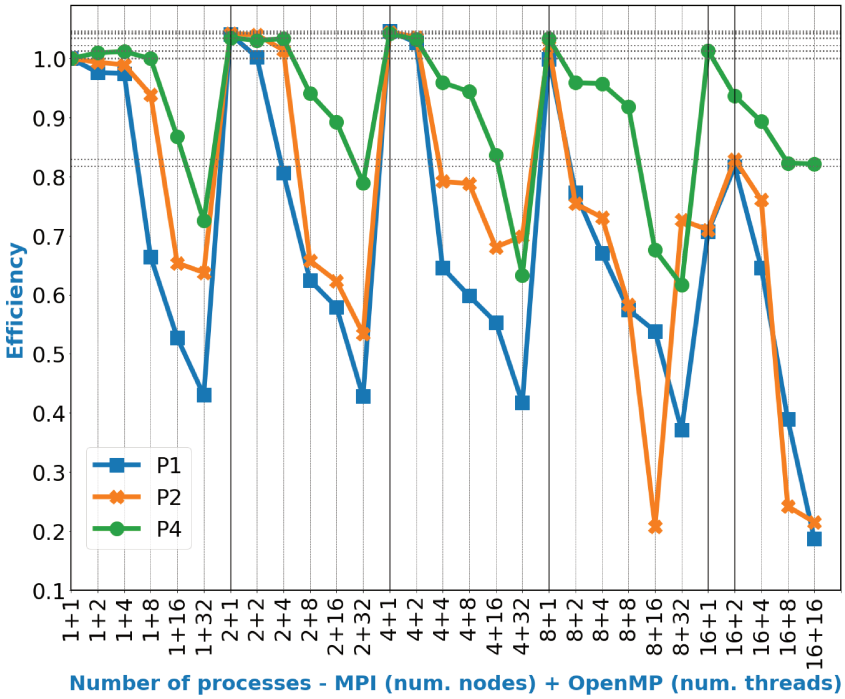


(b) Efficiency MPI-only for the matrix-vector operations for each finite element.

Figure 6.20. Speedup (a) and efficiency (b) of the **matrix-vector operations for each finite element** into the **linear** central difference local method of the beam problem for MPI with $[1, 16]$ compute nodes and 1 MPI process for each node.



(a) Speedup MPI + OpenMP.

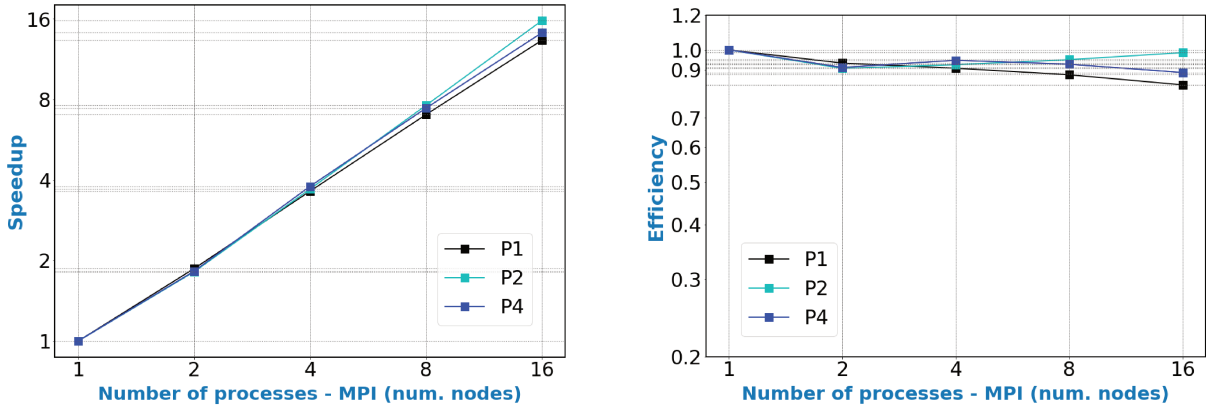


(b) Efficiency MPI + OpenMP.

Figure 6.21. Speedup (a) and efficiency (b) of the hybrid parallelism applied only to the **matrix-vector operations for each finite element** into the **linear** central difference local method of the beam problem for MPI + OpenMP using $[1,16]$ compute nodes with 1 MPI process for each node and $[1,20]$ cores with 1 or 2 threads per core.

Figures 6.22(a), 6.22(b), 6.23(a), and 6.23(b) present the scalability analysis for the non-linear central difference local method for the beam example with a superior speedup

than the linear analysis, mainly for lower-order approximation. Figure 6.22(a) with MPI-only highlights the speedup of Figure 6.23(a) with 1 MPI process for each compute node.



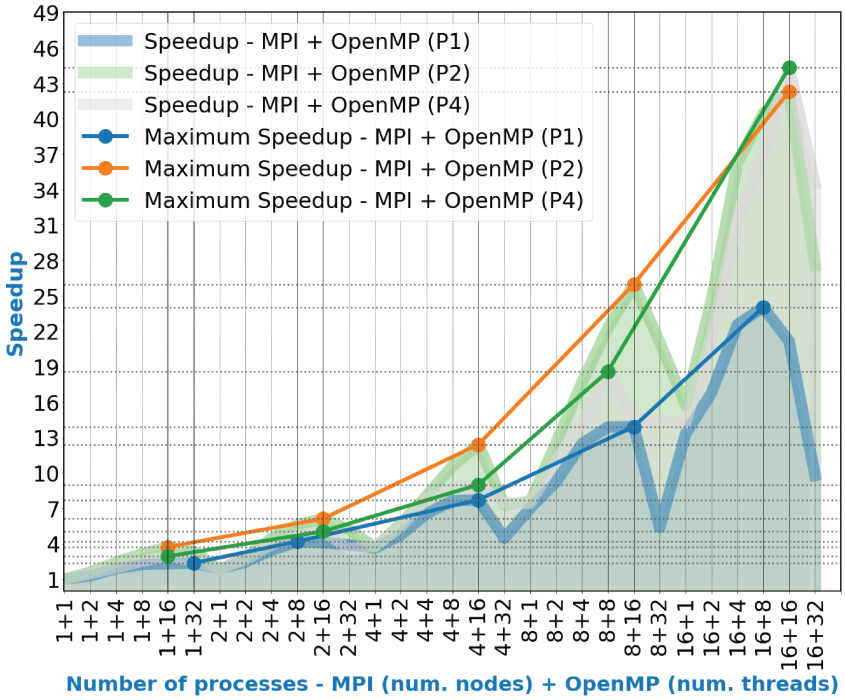
(a) Speedup MPI-only.

(b) Efficiency MPI-only.

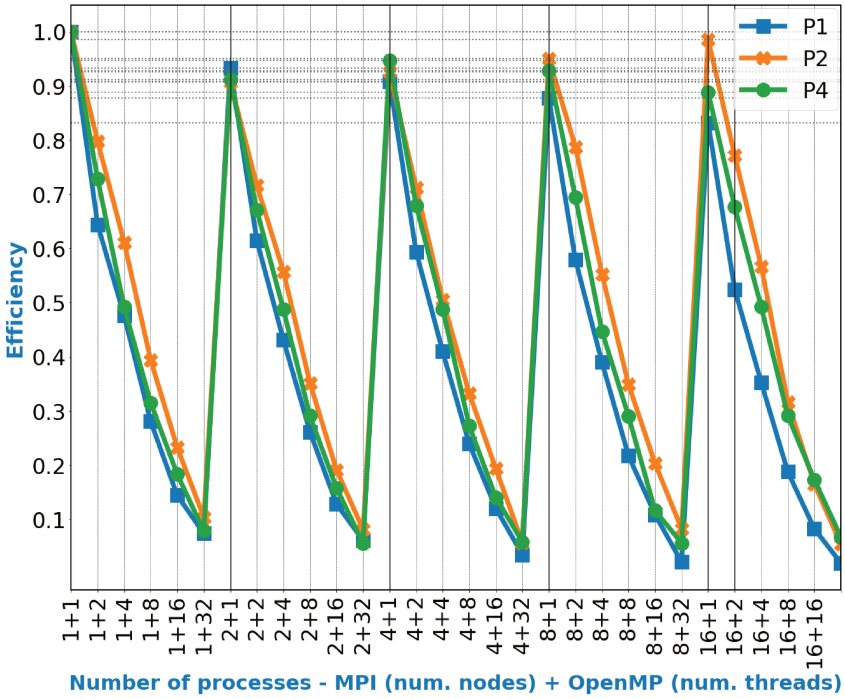
Figure 6.22. Speedup (a) and efficiency (b) of the MPI parallelism applied to the **non-linear** explicit transient analysis of the beam problem using $[1, 16]$ compute nodes and 1 MPI process for each node.

The MPI-only reached an efficiency of 88% for polynomial order 4 in the worst case. Therefore, the best reduced running time for MPI-only was from 4 to 0.17, 23.21 to 0.30, 1.47, and 29.59 seconds for the polynomials orders 1, 2 and 4, respectively, using 16 MPI processes.

Adding OpenMP in each node, the running time was reduced from 4 to 0.17, 23.21 to 0.55, and 420.40 to 9.49 seconds, for polynomial orders 1, 2 and 4, respectively. The best reduced time for the polynomial order 1 occurred using 16 MPI processes and 8 OpenMP threads, and both orders 2 and 4 occurred for 16 MPI porrocesses and 16 OpenMP threads. For the combination MPI + OpenMP in the non-linear problem, the best speedup and efficiency appear when increasing the number of DOFs, as shown in Figures 6.23(a) and 6.23(b).



(a) Speedup MPI + OpenMP.

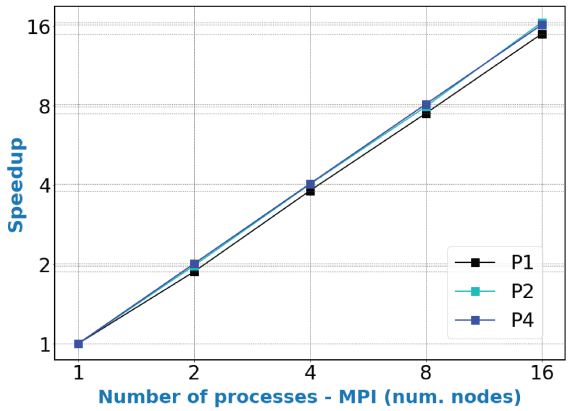


(b) Efficiency MPI + OpenMP.

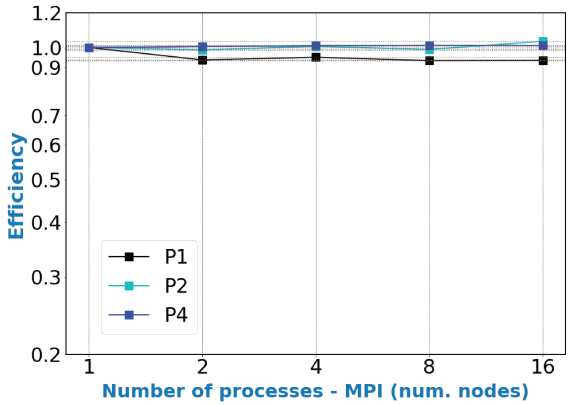
Figure 6.23. Speedup (a) and efficiency (b) of the hybrid parallelism applied to the **non-linear** explicit transient analysis of the beam problem for MPI + OpenMP using $[1,16]$ compute nodes with 1 MPI process for each node and $[1,20]$ cores with 1 or 2 threads per core.

Analyzing again only the matrix-vector operations for each finite element in the OpenMP parallel region, the speedup is superior to the linear local method for all polynomial orders, as we can see comparing Figures 6.25(a) and 6.20(a), as well 6.25(c) with 6.20(c).

The code portion analyzed was run in 2.63, 14.66, and 251.59 seconds of the total running time of the solver with 4.00, 23.21, and 420.40 seconds for the the polynomials orders 1, 2 and 4, respectively. Therefore, considering the running time, the OpenMP region was implemented for the 65.61%, 63.17%, and 59.85% of the non-linear solver for polynomials orders 1, 2 and 4. In summary, the polynomial order 4 achieved the best speedup of 200.8 and the minimum efficiency of 78% using 16 MPI processes and 16 OpenMP threads, as shown in Figures 6.25(c) and 6.25(d).

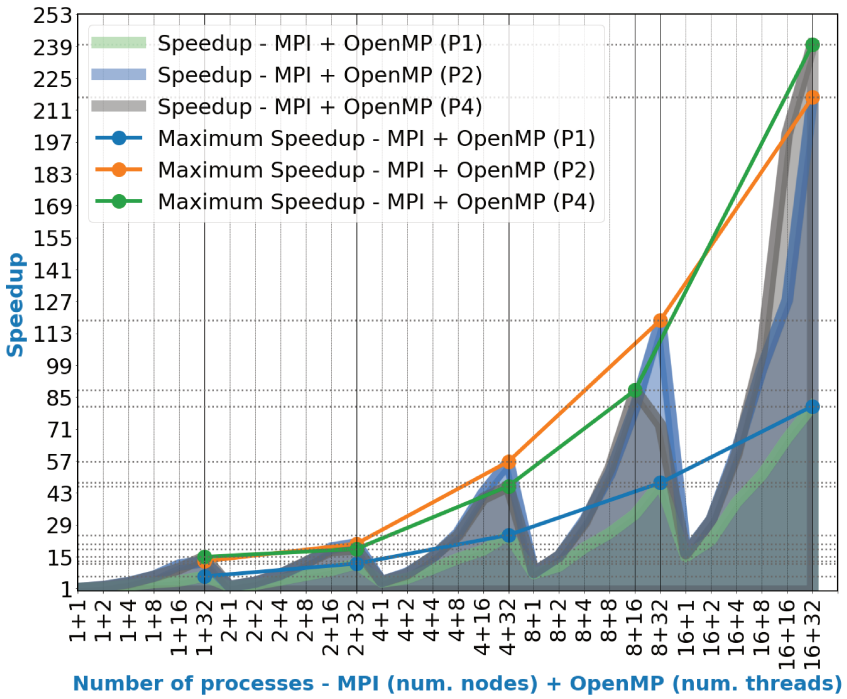


(a) Speedup MPI-only for the matrix-vector operations.

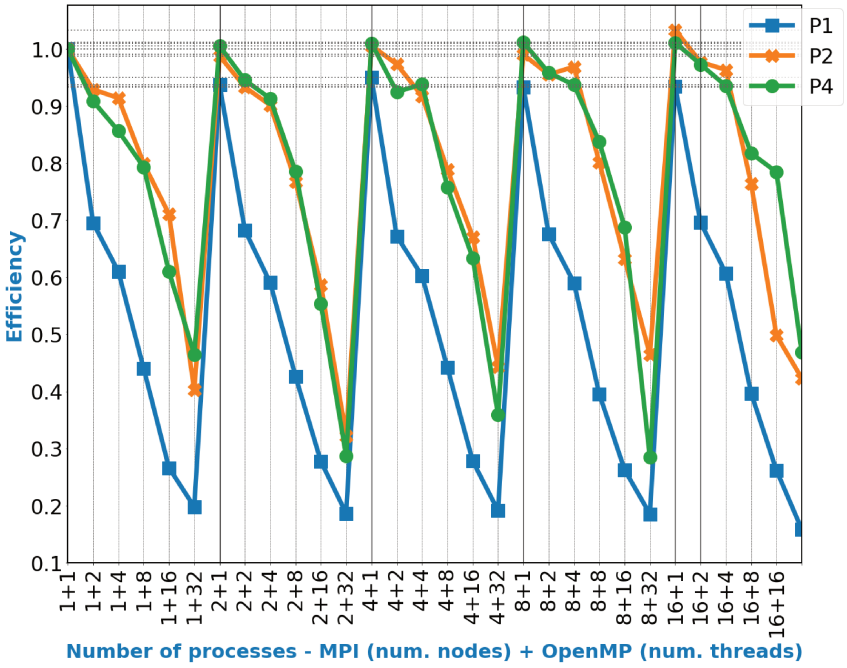


(b) Efficiency MPI-only for the matrix-vector operations.

Figure 6.24. Speedup (a) and efficiency (b) of the **matrix-vector operations for each finite element** into the **non-linear** central difference local method of the beam problem for MPI with $[1, 16]$ compute nodes and 1 MPI process for each node.



(a) Speedup MPI + OpenMP for the matrix-vector operations.



(b) Efficiency MPI + OpenMP for the matrix-vector operations.

Figure 6.25. Speedup (a) and efficiency (b) of the hybrid parallelism applied only to the **matrix-vector operations for each finite element** into the **non-linear** central difference local method of the beam problem for MPI + OpenMP using $[1,16]$ compute nodes with 1 MPI process for each node and $[1,20]$ cores with 1 or 2 threads per core.

6.8 SCALABILITY OF THE ELEMENT WISE LINEAR TRANSIENT SOLVER APPLIED TO CRANKSHAFT ANALYSES

This section presents high-performance results for linear explicit transient analyses using the element wise method for the crankshaft of a 4 cylinder and 4 strokes internal combustion engine illustrated in Figure 6.26. We considered Hookean material with Young's modulus $E = 210000\ N/mm^2$, Poisson ratio $\nu = 0.3$ and density $\rho = 7.85 \times 10^{-6}\ Kg/mm^3$.

The dynamic loads applied to the crankpins were obtained from the engine pressure curve at 2500 rpm for one engine cycle, that is, 180° of crankshaft rotation, considering 45 time instants (RODRIGUES, 2013). Figures 6.27(a) and 6.27(b) illustrate the forces in the x and y directions which define 45 load sets. They are applied to the external surfaces of each crankpin as distributed loads of intensities obtained dividing the forces by the respective surface areas for an angle of 60°. Speed of 2500 rpm corresponds to 12 milliseconds for one engine cycle. The crankpins are numbered 1 to 4 from left to right along the longitudinal axis z.

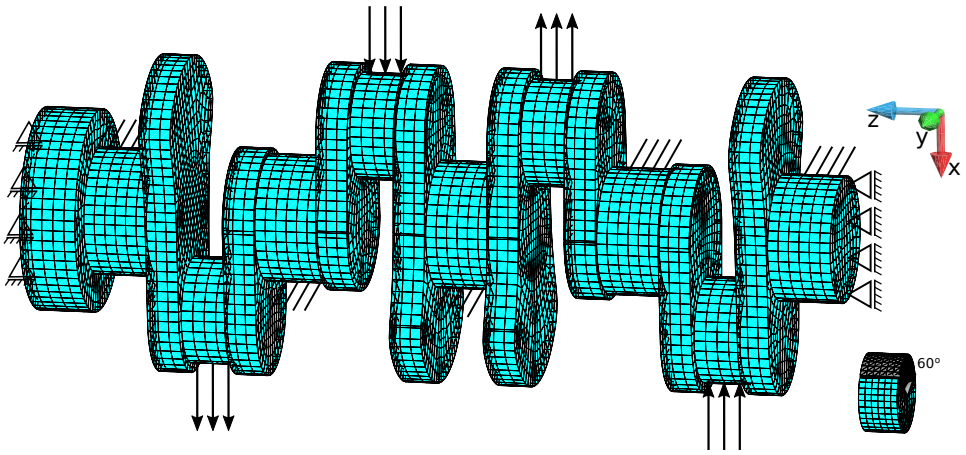
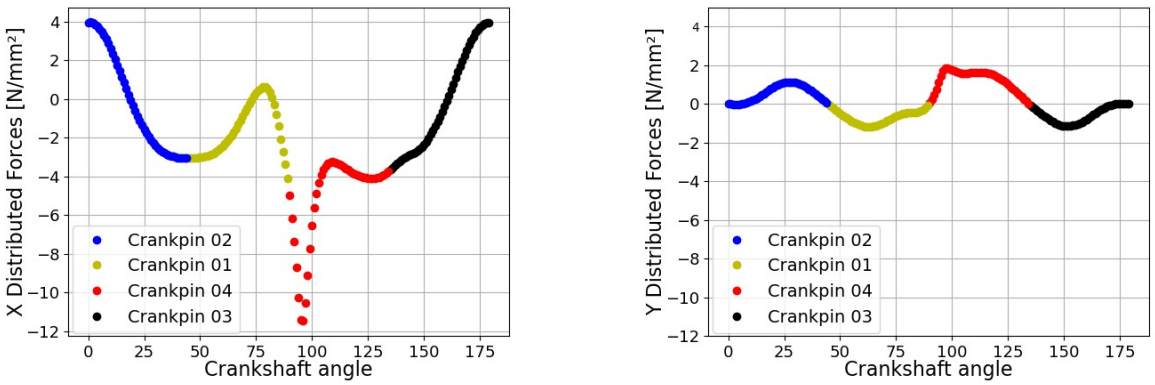


Figure 6.26. Crankshaft with the boundary conditions and loads. The crankpins are numbered 1 to 4 from the left to right of the longitudinal z-axis.



(a) Force in x-axis. **(b)** Force in y-axis.

Figure 6.27. Forces in x and y axes applied on the crank pins.

Five meshes were generated with 17 810, 73 800, 327 181, 580 781 and 1 789 811 hexahedrons and 22 423, 85 607, 358 877, 627 180 and 1 891 069 DOFs, respectively, for polynomial order 1. We also used polynomial orders 2 and 4 for all meshes and 6 for the coarsest mesh with 17 810 elements.

The transient analyses used lumped and consistent mass matrices calculated before the main timestep loop. The final time of the transient analyses was reduced as well as the number of load steps just to obtain the necessary scalability results. The tests were performed on the Kahuna cluster with 32 nodes.

6.8.1 Load balance

The multilevel k-way algorithm of the METIS library was used for partitioning of the crankshaft meshes. This algorithm has more parameters than the multilevel recursive bisection algorithm described in Section 5.3. After evaluation, the k-way procedure had better results for the load-balancing of the crankshaft meshes.

The most important parameters of the k-way algorithm were "-contig" and "-minconn". The first one avoids the discontinuity of elements in partitions. The second one reduces the size of communication messages between neighboring partitions. To reduce the *edgecut* parameter discussed in Section 5.3, we set the edge weights with larger values for neighbors sharing fewer nodes. We used the weight array $[0,4,3,2,1]$, where the index represents the number of shared nodes between elements. For example, index 1 means that elements share 1 node and the respective weight is 4; analogously, index 4 means that elements share 4 nodes as in square faces and the respective weight is 1. These options allowed to reduce the size of the communication messages between neighboring partitions.

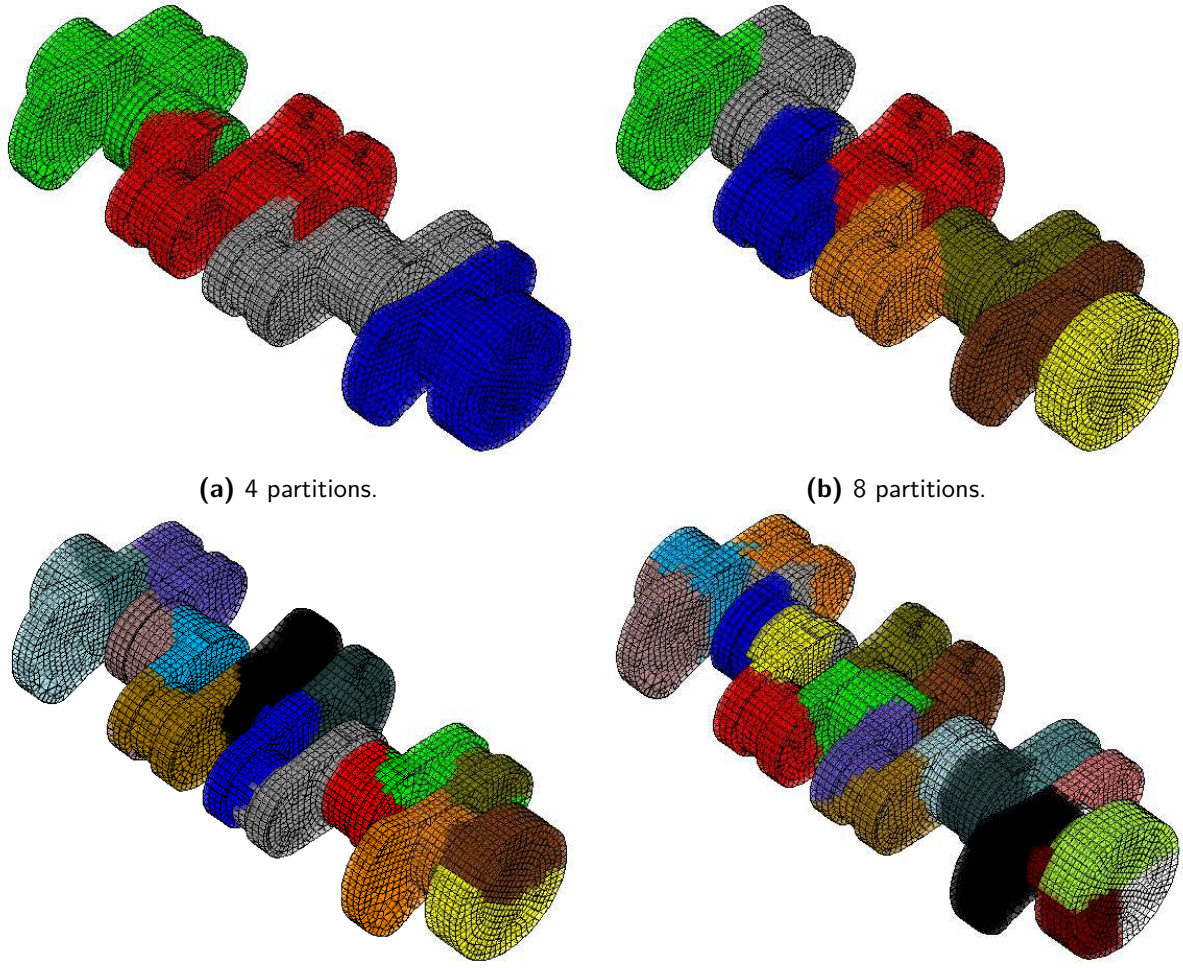
Table 6.5 presents results of partitioning for the mesh of 17 810 hexahedrons. The numbers of elements and the maximum and minimum numbers of elements of each partition are listed. The largest difference in the number of elements is for 4 partitions, where the difference between partitions 2 and 0 is 264 or 1.45% of the total number of hexahedrons. As the number of partitions increases, this difference reduces.

Figures 6.28(a) to 6.28(d) show the distribution of elements and confirm that the k-way algorithm with parameter "-contig" obtained contiguous partitions. Conversely, the multilevel recursive bisection algorithm obtained discontinuous partitions for the same meshes of Figure 6.28.

The next sections present performance analyses in terms of speedup, efficiency and execution time (runtime) for the element wise linear explicit transient method using the 5 generated meshes of the crankshaft and varying the polynomial orders, considering consistent and lumped element mass matrices. In addition, we compare the parallel performance of the algorithm for different mass matrix types and mesh sizes. Finally, we consider the analysis of taking

Table 6.5 – Partitioning of the crankshaft mesh with 17810 hexahedrons.

# Partitions	Number of elements by partition								Max	Min
4	(0) 4322	(1) 4323	(2) 4586	(3) 4579					4586	4322
8	(0) 2293	(1) 2178	(2) 2165	(3) 2293	(4) 2226	(5) 2194	(6) 2293	(7) 2168	2293	2165
16	(0) 1146	(1) 1146	(2) 1080	(3) 1111	(4) 1080	(5) 1080	(6) 1080	(7) 1080	1146	1080
	(8) 1146	(9) 1146	(10) 1146	(11) 1117	(12) 1080	(13) 1080	(14) 1146	(15) 1146		
20	(0) 917	(1) 902	(2) 876	(3) 917	(4) 906	(5) 874	(6) 864	(7) 917	917	864
	(8) 917	(9) 917	(10) 873	(11) 864	(12) 864	(13) 917	(14) 917	(15) 885		
	(16) 868	(17) 864	(18) 876	(19) 875						



(a) 4 partitions.

(b) 8 partitions.

(c) 16 partitions.

(d) 20 partitions.

Figure 6.28. Partitions for the crankshaft mesh with 17810 hexahedrons.

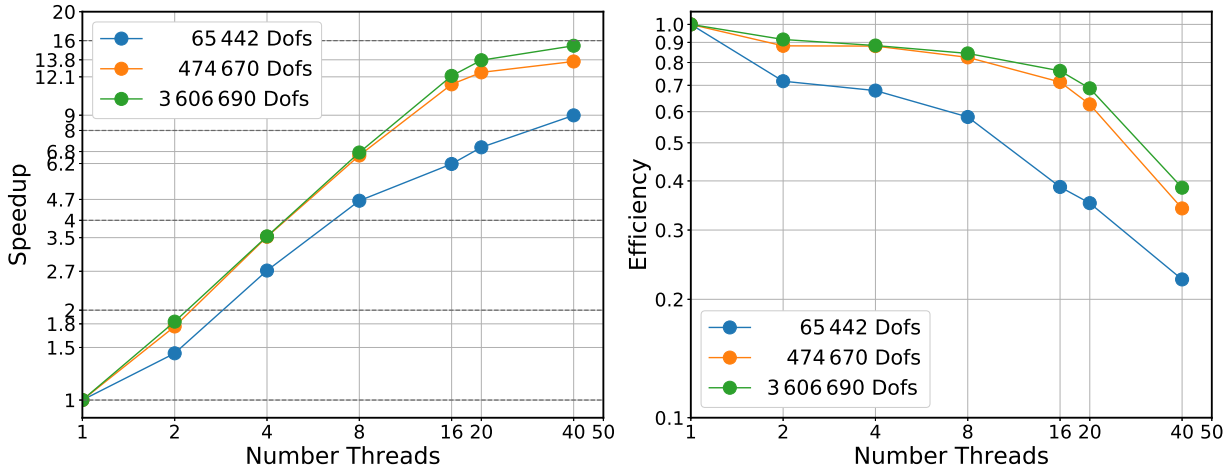
into account mesh size, polynomial order, and runtime for up to 20 computational nodes, 20 cores per node, and 2 threads per core of the Kahuna cluster.

6.8.2 Scalability analysis with consistent mass matrices

Initially, the algorithm of Section 2.6 with consistent mass matrices are considered using the Kahuna cluster with 1, 2, 4, and 8 nodes and 1 MPI rank per node, varying the number of cores from 1, 2, 4, 8, 16, and 20 and 1 and 2 threads per core, totalizing up to 8 MPI ranks and 40 OpenMP threads. The results are for the mesh of 17 810 elements, and polynomial orders 1, 2, and 4.

Figures 6.29(a) and 6.29(b) show the speedup and efficiency considering 1 computing node and varying the number of threads in order to evaluate the OpenMP behavior only. As each compute node has 20 cores, the ideal speedup should be 20 for each thread. Using hyperthreading for the Intel Xeon Processor family $E5-2670\ v2$, there may be a performance gain of up to 30% for threads in the same core.

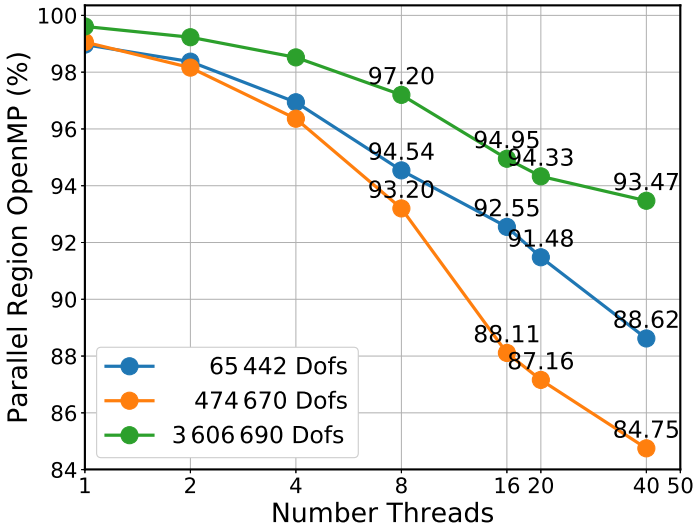
The speedup of Figure 6.29(a) shows that scalability is best for p refinement with maximum of 9.0, 13.6 and 15.4 for orders 1, 2 and 4, respectively. Using hyperthread, the performance gain was, respectively, 27.9%, 8.8% and 11.8% for the same polynomial orders. The efficiency in Figure 6.29(b) considers the average time for each OpenMP thread. Scalability for order 1 is better up to 8 threads with 58% efficiency. For polynomial orders 2 and 4, efficiency is larger than 60% with 68% for polynomial order 4 and 13.8 speedup with 20 cores.



(a) Speedup. **(b)** Efficiency.

Figure 6.29. OpenMP scalability of the element wise linear central difference algorithm using consistent mass matrices for the crankshaft mesh with 17 810 hexahedrons and polynomial orders 1 (65 442 DOFs), 2 (474 670 DOFs) and 4 (3 606 690 DOFs). (a) and (b) are speedup and efficiency results for 1 compute node with 20 cores and 2 threads per core, totalizing 40 OpenMP threads.

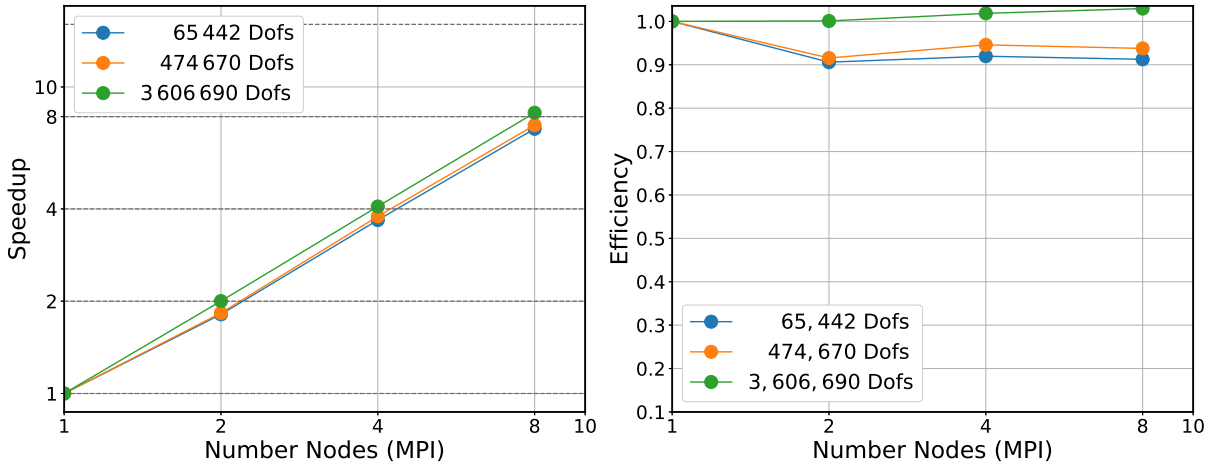
In addition, we present the cost of parallel OpenMP regions for all time steps. Figure 6.30 shows the runtime percentage of OpenMP regions over the total execution time of the algorithm. By increasing the polynomial order, the runtime is longer within the parallel regions, decreasing the percentage of thread synchronization time. The largest percentage was 93.5% for $P = 4$ and 3 606 690 DOFs. High-orders made better use of parallel regions and improved the OpenMP scalability.



(a) Parallel region (%) on total elapse time.

Figure 6.30. OpenMP parallel region percentage for the element wise linear central difference method using consistent mass matrices for the crankshaft mesh with 17 810 hexahedrons and polynomial orders 1 (65 442 DOFs), 2 (474 670 DOFs) and 4 (3 606 690 DOFs). We use 1 compute node with 20 cores and 2 threads per core, totalizing 40 OpenMP threads.

The speedup and efficiency with MPI is presented in Figures 6.31(a) and 6.31(b) for up to 8 nodes. For all polynomial orders, the speedup was close to the ideal and efficiency greater than 90%.



(a) Speedup. **(b)** Efficiency.

Figure 6.31. MPI scalability of the element wise linear central difference algorithm using consistent mass matrices for the crankshaft mesh with 17 810 hexahedrons and polynomial orders 1 (65 442 DOFs), 2 (474 670 DOFs) and 4 (3 606 690 DOFs). (a) and (b) are speedup and efficiency results for $[1,8]$ compute nodes and 1 MPI process per node.

Table 6.6 shows the runtime for each node and OpenMP thread, considering the polynomial orders 1, 2 and 4. The execution time of the serial code is highlighted in yellow and the best time with hybrid parallelism for each node in green. In summary, the execution time was reduced for the best MPI + OpenMP combination from 91.27, 551.22 and 8887.52 to 3.20, 9.07 and 139.78 seconds for polynomial orders 1, 2 and 4, respectively. The results were 28.52, 60.77 and 63.58 times faster, respectively, for MPI processes + OpenMP threads with $8+20$, $8+40$ and $8+16$.

6.8.3 Scalability analysis with lumped mass matrices

This section presents the performance analysis of the element wise linear explicit transient method using lumped mass element matrices. These matrices allowed the algorithm's execution with the more refined crankshaft meshes and higher polynomial orders in the Kahuna cluster. Table 6.7 displays the memory in GB necessary to store the mass and stiffness element matrices of the system of equation (2.49), for consistent and lumped mass matrices and h and p refinements, The memory consumption reduced up to 3 times. As described in Section 6.1.3, each node of the Kahuna cluster has 64 GB of memory RAM, requiring more than 1 computing node for running the meshes with higher polynomial orders.

Table 6.6 – Running time in seconds of the hybrid parallelism for consistent mass matrices.

MPI+OpenMP	P1-Runtime	P2-Runtime	P4-Runtime
1+1	91.27	551.22	8887.52
1+2	63.63	312.47	4853.71
1+4	33.61	156.46	2514.37
1+8	19.62	83.57	1316.58
1+16	14.77	48.27	729.14
1+20	12.99	44.02	646.08
1+40	10.15	40.47	577.93
2+1	50.37	301.01	4439.36
2+2	36.17	172.93	2754.32
2+4	19.88	94.00	1492.27
2+8	13.41	57.75	786.72
2+16	10.83	44.59	493.26
2+20	10.28	45.13	460.64
2+40	8.04	45.00	419.91
4+1	24.81	145.68	2181.40
4+2	18.51	82.88	1396.55
4+4	10.47	44.93	736.15
4+8	7.05	26.56	404.96
4+16	5.75	19.73	248.66
4+20	5.57	20.04	229.78
4+40	4.72	21.35	209.29
8+1	12.50	73.47	1079.03
8+2	9.60	41.31	682.41
8+4	5.64	22.65	354.78
8+8	3.86	13.40	199.36
8+16	3.32	9.34	139.78
8+20	3.20	9.14	146.26
8+40	3.77	9.07	155.18

Runtimes of the element wise algorithm using consistent mass matrices for many combinations of MPI processes and OpenMP threads. The serial code runtime is highlighted in yellow and the best time for each node is highlighted in green color.

Table 6.7 – Total memory in GigaBytes (GB) required for element matrices when using consistent and diagonal mass matrices.

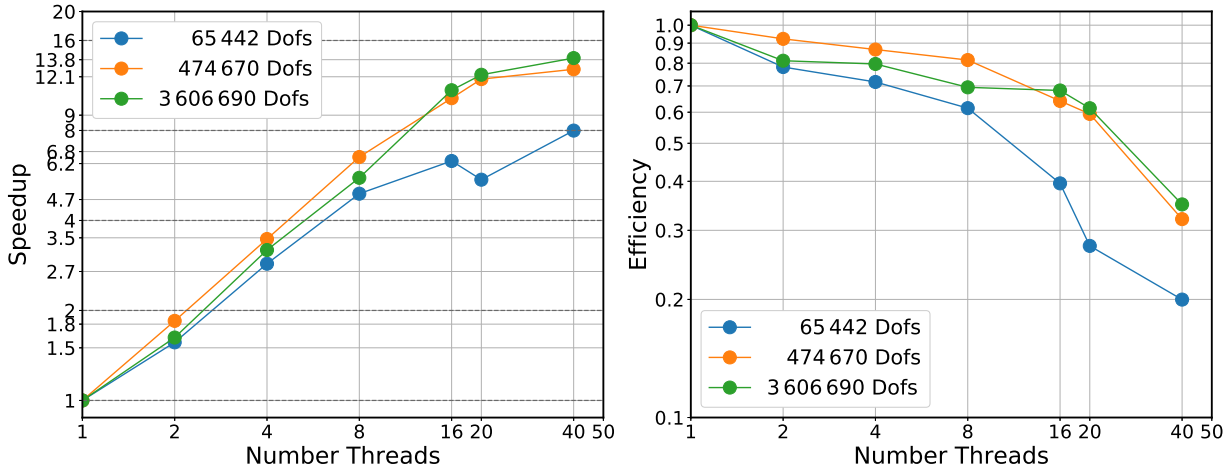
Memory (GB)						
h-refinement	Mass matrix	p-refinement				
		1	2	4	6	8
17810	CONSIST	0.1194	1.3221	28.0650	210.9589	952.4483
	LUMPED	**0.0430**	**0.4514**	**9.4048**	**70.4561**	**317.7730**
73800	CONSIST	0.4949	5.4782	116.2939	874.1588	3946.6975
	LUMPED	**0.1782**	**1.8706**	**38.9708**	**291.9521**	**1316.7683**
327181	CONSIST	2.1939	24.2867	515.5711	3875.4492	17497.0790
	LUMPED	**0.7898**	**8.2931**	**172.7711**	**1294.3248**	**5837.6909**
580781	CONSIST	3.8944	43.1115	915.1934	6879.3337	31059.1722
	LUMPED	**1.4020**	**14.7210**	**306.6872**	**2297.5638**	**10362.5209**
1789811	CONSIST	12.0016	132.8579	2820.3802	21200.2581	95716.0239
	LUMPED	**4.3205**	**45.3661**	**945.1274**	**7080.4745**	**31934.5052**

The numbers of compute nodes were 1, 2, 4, 8, 16, 20, 25 and 30 with 1 MPI process per node. In addition, 1, 2, 4, 8, 16 and 20 cores were used with 1 or 2 threads per core and a

total of 40 OpenMP threads. The coarsest crankshaft mesh of 17 810 elements was considered with polynomial orders 1, 2 and 4.

Figures 6.32(a) and 6.32(b) show the speedup and efficiency, respectively. As with the consistent mass matrix, better speedups were obtained for higher polynomial orders using lumped matrices. However, a nonlinear speedup behavior was obtained between 16 and 20 cores for $P = 1$. In this case, as the memory demand is minimal, according to Table 6.7, there was a slight time variation using 16 and 20 OpenMP threads, not enough to impact the scalability. Figure 6.33 shows the time percentage of OpenMP parallel regions. The mentioned behavior with 16 and 20 cores and $P = 1$ may also be observed. The percentage of computing time in parallel regions increases with 20 and 40 OpenMP threads.

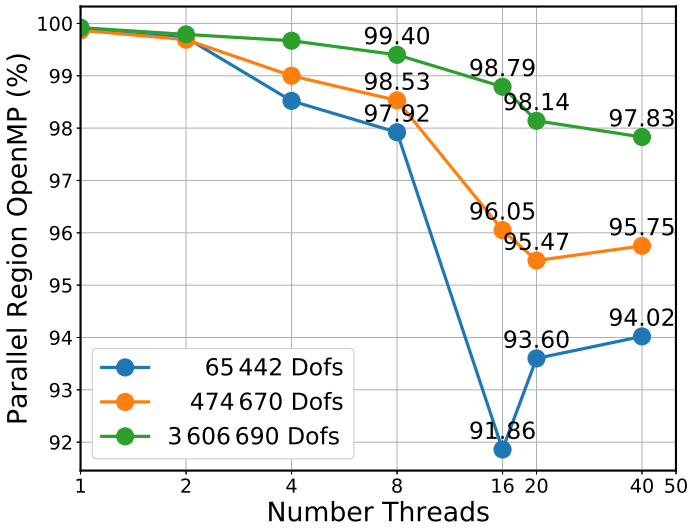
The largest speedups for polynomial orders 1, 2 and 4 were 6.3, 11.9 and 12.3 for 16, 20 and 20 nodes, respectively, using 1 thread per core. Using hyperthread, the performance gains with 40 threads were 45.0%, 7.9% and 13.8% for orders 1, 2 and 4, respectively.



(a) Speedup. **(b)** Efficiency.

Figure 6.32. OpenMP scalability for the linear central difference element wise method using lumped mass matrices for the crankshaft mesh with 17 810 hexahedrons and polynomial orders 1 (65 442 DOFs), 2 (474 670 DOFs) and 4 (3 606 690 DOFs). (a) and (b) are results for 1 compute node with 20 cores and 2 threads per core, totalizing 40 OpenMP threads.

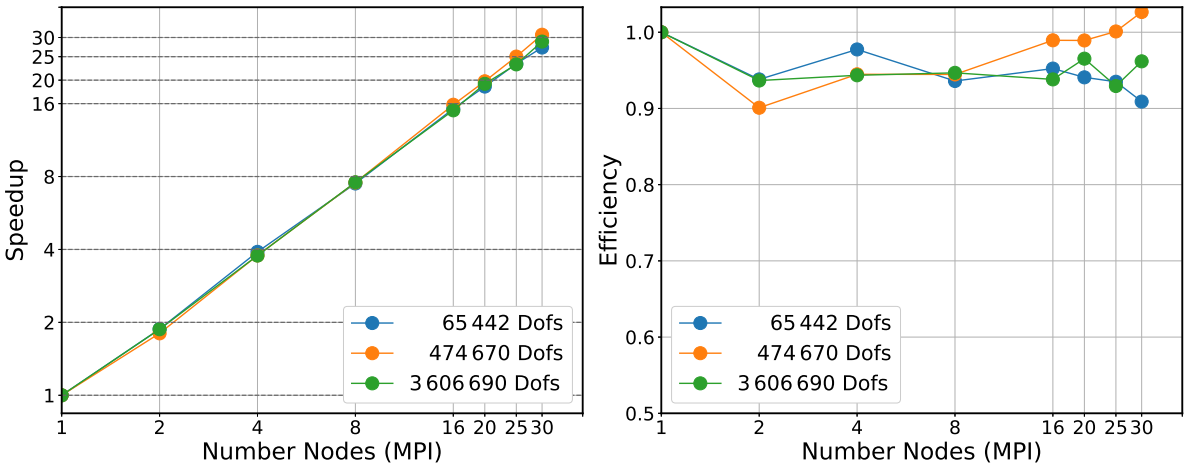
Similarly to Figure 6.30, the better percentage use of OpenMP parallel regions are related to higher polynomial orders as illustrated in Figure 6.33 for $P = 4$. Each thread in the parallel regions computes one finite element of each partition. For 20 cores and 2 OpenMP threads per core, we obtained 97.83% of runtime for all times steps and $P = 4$. This result indicates an optimal OpenMP multi-thread parallelism.



(a) Parallel region (%) on total elapse time.

Figure 6.33. OpenMP parallel region percentage for the element wise linear central difference method using lumped mass matrices for the crankshaft mesh with 17 810 hexahedrons and polynomial orders 1 (65 442 DOFs), 2 (474 670 DOFs) and 4 (3 606 690 DOFs). We use 1 compute node with 20 cores and 2 threads per core, totalizing 40 OpenMP threads.

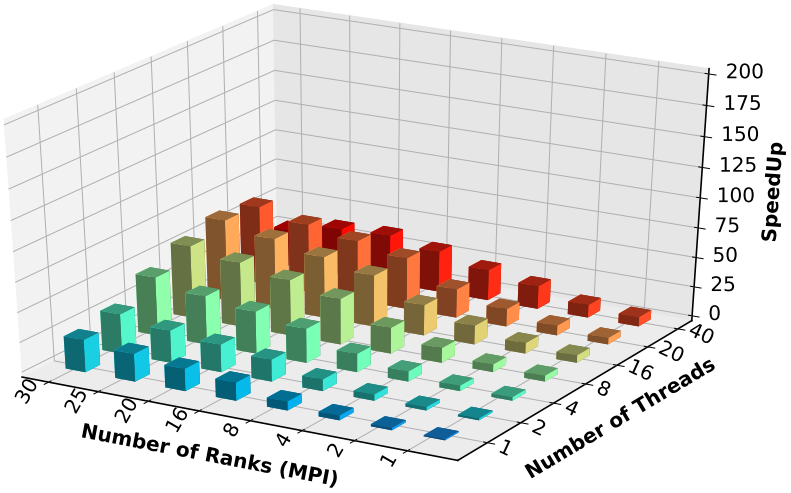
When using 1 MPI process per node, the speedup obtained was close to ideal for all polynomial orders, as shown in Figure 6.34(a). The efficiency was greater than 90% for all cases. Using up to 30 compute nodes, the maximum speedups obtained were 27.3, 30.8, and 28.9 for polynomial orders 1, 2 and 4, respectively. For $P = 2$, there was a superlinear speedup, which may be due to the decreasing of data distribution for each process and stored in cache memory.



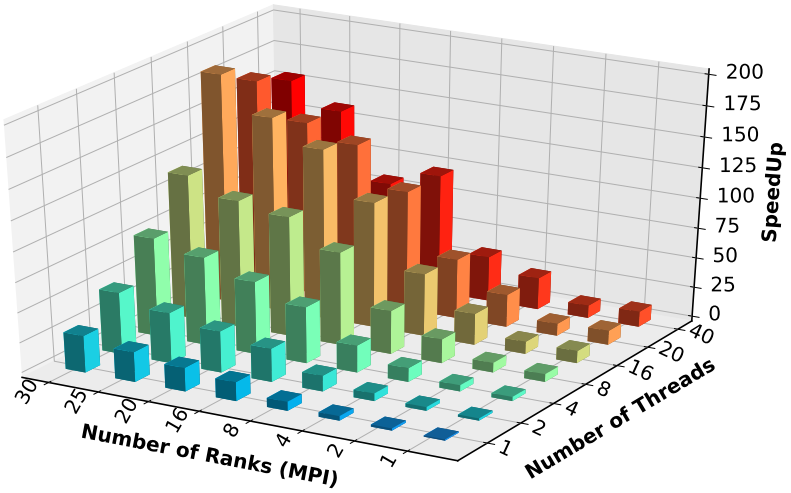
(a) Speedup. **(b)** Efficiency.

Figure 6.34. MPI scalability of the element wise linear central difference algorithm using lumped mass matrices for the crankshaft mesh with 17 810 hexahedrons and polynomial orders 1 (65 442 DOFs), 2 (474 670 DOFs) and 4 (3 606 690 DOFs). (a) and (b) are speedup and efficiency results for $[1, 30]$ compute nodes and 1 MPI rank per node.

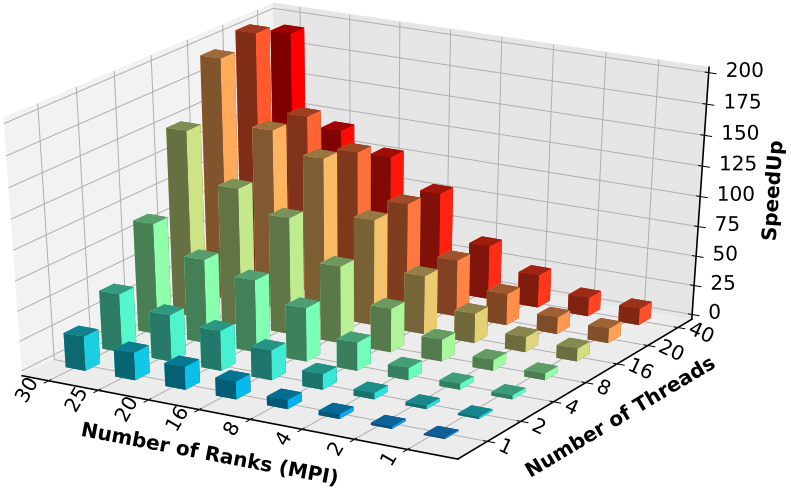
Performance of the MPI + OpenMP hybrid version is shown in Figures 6.35 and 6.36. The speedup for polynomial orders 1, 2 and 4 are presented separately in Figures 6.35(a), 6.35(b) and 6.35(c), respectively, with the number of MPI nodes, the number of OpenMP threads and the obtained speedups along the x, y and z axes. Speedup increases for higher polynomial orders.



(a) $P = 1$.



(b) $P = 2$.



(c) $P = 4$.

Figure 6.35. MPI+OpenMP scalability for the element wise linear central difference method using lumped matrices, crankshaft mesh with 17810 hexahedrons and polynomial orders=1, 2 e 4. Results are for $[1,30]$ compute nodes with 1 rank MPI per node, 20 cores per node and $[1,2]$ threads per core.

Figure 6.36 also shows MPI + OpenMP hybrid scalability for all polynomial orders considered. In addition, the maximum speedups are indicated by lines for all polynomial orders with the maximum speedup points highlighted for each number of nodes. In order to take advantage of using shared memory, we considered 1 MPI process per node and many OpenMP threads per core. Therefore, the maximum speedups for the MPI + OpenMP combinations and polynomial orders 1, 2 and 4 were 68.5 with 30 MPI ranks and 16 OpenMP threads (30 + 16), 188.5 with 30 + 16 and 207.8 with 30 + 20, respectively.

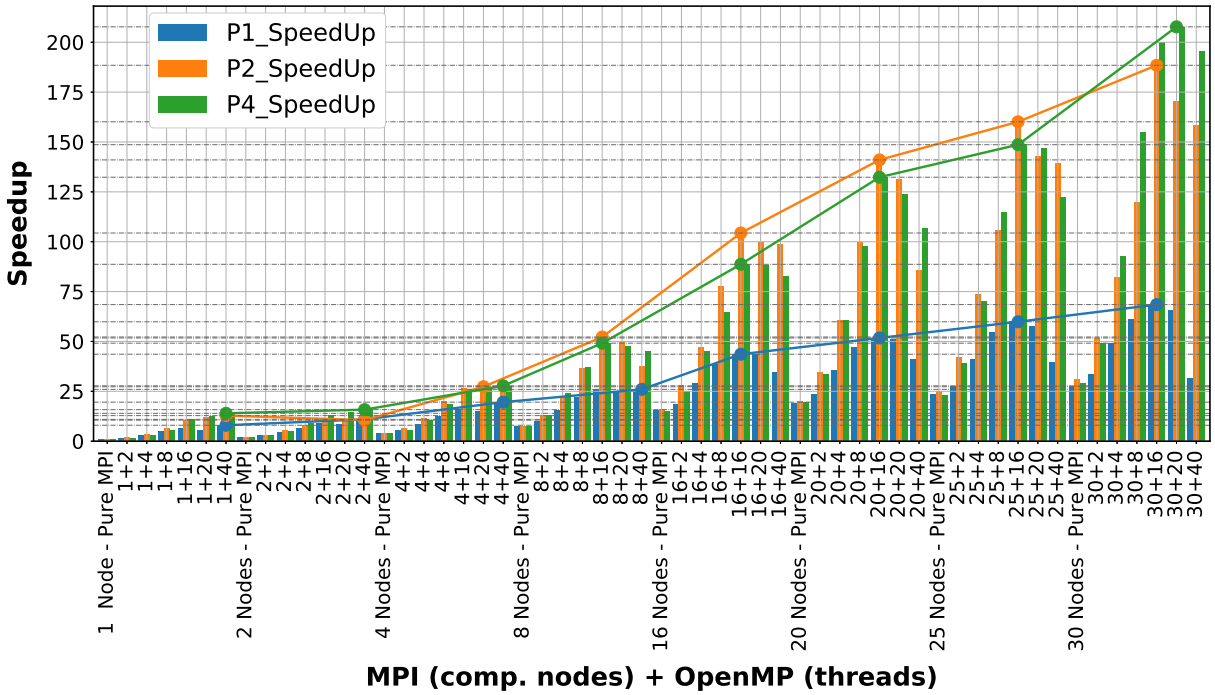


Figure 6.36. MPI+OpenMP scalability for the element wise linear central difference method using lumped mass matrices for the crankshaft mesh with 17810 hexahedrons and polynomial orders 1, 2 and 4. The results are run for $[1,30]$ compute nodes with 1 ranks MPI per node, 20 cores per node, and $[1,2]$ threads per core.

Runtime with lumped mass matrices is given in Table 6.8, including the serial and parallel runtimes using MPI processes + OpenMP threads. The best cases of runtime reduction are highlighted in green for each MPI process. These same cases can be observed by the maximum speedups shown in Figure 6.36. The runtime reduced to $P = 1$ from 9.70 seconds with 1 process to 0.14 seconds with 30 MPI processes and 16 OpenMP threads. For $P = 2$, runtime reduced from 61.56 seconds with 1 process to 0.33 seconds with 30 MPI processes and 16 OpenMP threads. Finally, for $P = 4$, runtime reduced from 890 seconds to 1 process for 4.28 seconds with 30 MPI processes and 20 OpenMP threads. In summary, the code was 69.29, 186.58 and 207.94 times faster for polynomial orders 1, 2 and 4, respectively.

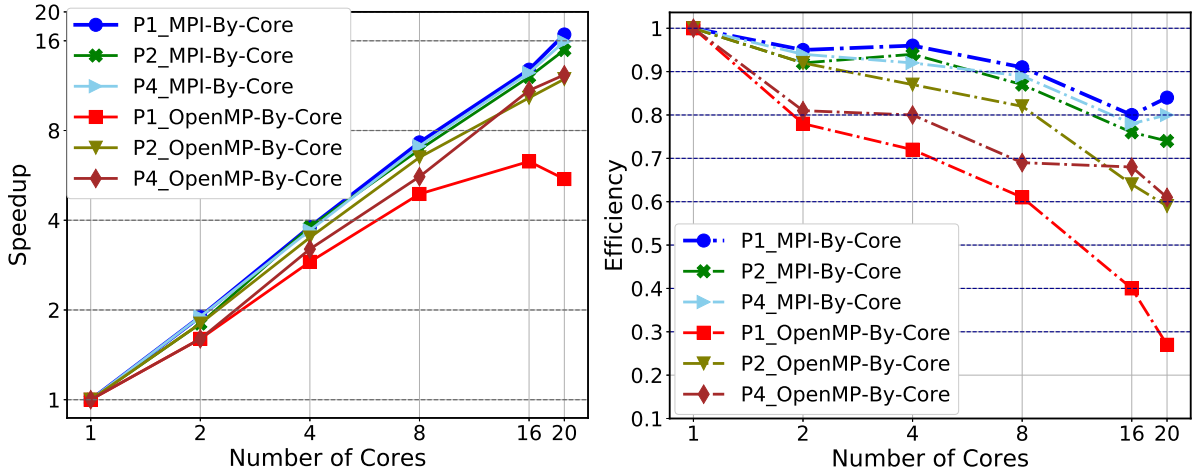
Table 6.8 – Running time of the hybrid parallelism for lumped matrices.

MPI+OpenMP	P1_ElapsedTime	P2_ElapsedTime	P4_ElapsedTime
1 + 1	9.70	61.57	890.00
1 + 2	6.20	33.35	548.19
1 + 4	3.39	17.75	279.35
1 + 8	1.97	9.44	160.20
1 + 16	1.53	6.00	81.60
1 + 20	1.77	5.18	72.44
1 + 40	1.21	4.80	63.70
2 + 1	5.17	34.17	475.07
2 + 2	3.54	19.94	323.51
2 + 4	2.14	11.79	178.05
2 + 8	1.49	7.58	98.37
2 + 16	1.08	5.97	69.28
2 + 20	1.17	6.06	62.56
2 + 40	0.89	5.83	56.29
4 + 1	2.48	16.29	235.81
4 + 2	1.86	9.65	166.77
4 + 4	1.12	5.30	86.57
4 + 8	0.77	3.05	48.13
4 + 16	0.61	2.34	35.50
4 + 20	0.65	2.26	33.59
4 + 40	0.50	2.31	32.07
8 + 1	1.30	8.15	117.51
8 + 2	0.98	4.68	68.07
8 + 4	0.63	2.71	37.21
8 + 8	0.44	1.69	24.19
8 + 16	0.38	1.18	18.11
8 + 20	0.40	1.24	18.68
8 + 40	0.37	1.65	19.68
16 + 1	0.64	3.89	59.29
16 + 2	0.52	2.21	35.38
16 + 4	0.34	1.31	19.81
16 + 8	0.25	0.79	13.76
16 + 16	0.22	0.59	10.04
16 + 20	0.22	0.62	10.11
16 + 40	0.28	0.62	10.74
20 + 1	0.52	3.11	46.10
20 + 2	0.42	1.78	26.56
20 + 4	0.27	1.02	14.68
20 + 8	0.21	0.62	9.13
20 + 16	0.19	0.44	6.73
20 + 20	0.19	0.47	7.18
20 + 40	0.24	0.72	8.34
25 + 1	0.42	2.46	38.30
25 + 2	0.36	1.46	22.75
25 + 4	0.24	0.84	12.68
25 + 8	0.18	0.58	7.76
25 + 16	0.16	0.38	5.99
25 + 20	0.17	0.43	6.07
25 + 40	0.25	0.44	7.29
30 + 1	0.36	2.00	30.84
30 + 2	0.29	1.19	18.15
30 + 4	0.20	0.75	9.60
30 + 8	0.16	0.51	5.75
30 + 16	0.14	0.33	4.45
30 + 20	0.15	0.36	4.28
30 + 40	0.31	0.39	4.55

Runtime of the element wise algorithm using lumped matrices for each compute node and each OpenMP thread. The serial runtime is highlighted in yellow, and the best time for each node is highlighted in green color.

6.8.4 Analysis of MPI processes per core

Memory reduction using lumped mass element matrices allows the partitioning of meshes into subdomains in the same compute node. We present here the performance results for the MPI process in each core, disabling OpenMP. Figure 6.37 shows the speedup and efficiency of comparing MPI and OpenMP for one compute node and assigning one MPI process or one OpenMP thread to each core. The results are for up to 20 cores and the coarsest mesh of 17 810 hexahedrons with polynomial orders 1, 2, and 4.



(a) Speedup. **(b)** Efficiency.

Figure 6.37. Scalability using one MPI process and one OpenMP thread per core.

Figure 6.37(a) shows that the best speedups occurred with MPI per core compared to OpenMP per core. The speedups are calculated using the runtimes given in Table 6.9. It can be observed that they decay according to the number of cores. When running 1 compute node with the maximum capacity of 20 cores, the speedups for polynomial orders 1, 2, and 4 were 16.8, 14.9, and 15.9, respectively. On the other hand, the speedups with OpenMP were lower at 5.5, 11.9, and 12.3. As mentioned previously, the use of OpenMP shared-memory parallelism had better gains with an increasing number of DOFs. However, for this case, MPI results had better scalability than OpenMP even for polynomial order 4.

Similarly, the efficiency of Figure 6.37(b) shows that the core processing using MPI performed better than OpenMP per core. For lower order $P = 1$, the difference in performance between the two parallel strategies is even larger with an efficiency of 84% for MPI and 27% for OpenMP using 20 cores. The results also show that the updating of solutions by OpenMP threads was slower than the communication time of MPI processes.

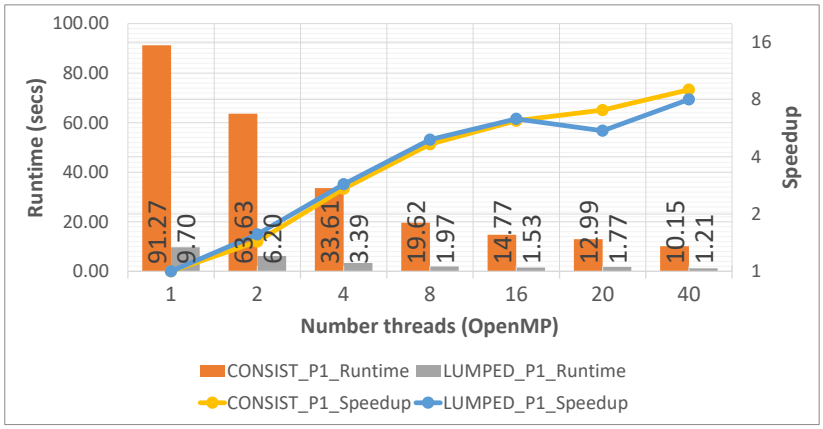
Table 6.9 – Runtime (s) of the element wise explicit transient algorithm comparing MPI and OpenMP by core for 1 compute node.

Number Cores	Poly Order 1		Poly Order 2		Poly Order 4	
	OpenMP	MPI	OpenMP	MPI	OpenMP	MPI
1	9.70	7.00	61.57	51.19	890.00	823.66
2	6.20	3.69	33.35	27.92	548.19	439.78
4	3.38	1.82	17.75	13.59	279.35	224.27
8	1.97	0.96	9.44	7.37	160.20	115.73
16	1.53	0.55	6.00	4.23	81.60	65.90
20	1.77	0.42	5.18	3.44	72.44	51.75

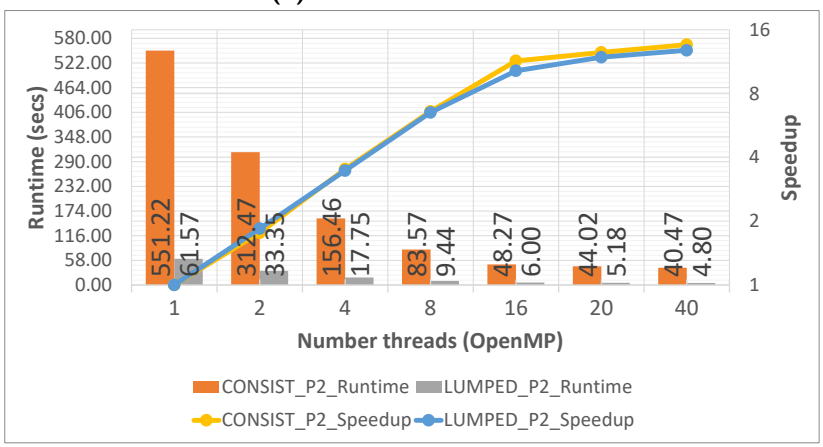
6.8.5 Comparison of hybrid scalability for consistent and lumped mass matrices

This section presents a performance comparison of the runtime of the element wise explicit algorithm with consistent and lumped element mass matrices using the mesh of 17 810 hexahedrons. Runtime and speedups with OpenMP parallelism are shown in Figures 6.38(a), 6.38(b) and 6.38(c); runtime is displayed by bars and speedup by lines. The results were obtained for 1, 2, 4 and 8 compute nodes with 1 MPI rank per node. In addition, 1, 2, 4, 8, 16 and 20 cores were considered with 1 or 2 threads per core, using up to 40 OpenMP threads.

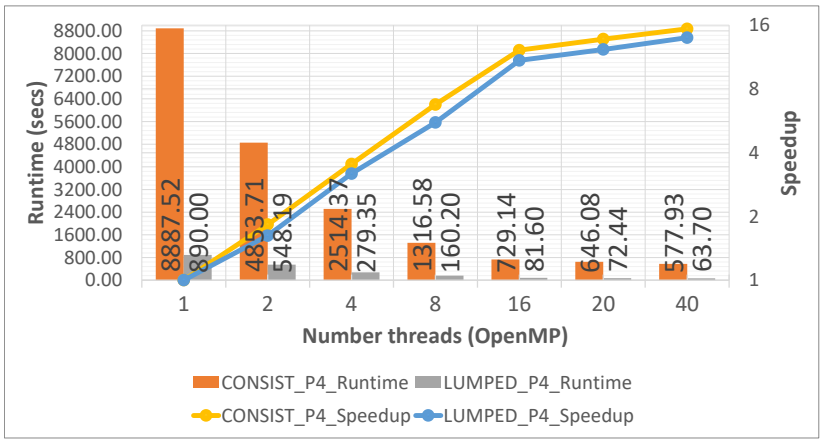
Runtime with diagonal matrices (LUMPED) was almost 10 times less than using consistent matrices (CONSIST). This proportion of runtime reduction was maintained as the number of OpenMP threads increased. Scalability with both types of matrices was similar in all cases. However, for $P = 4$, the algorithm with consistent matrices showed slightly better scalability, mainly from 8 threads. In this case, for 8, 16 and 20 OpenMP threads (1 thread per core), speedups were 6.75, 12.19 and 15.38 for consistent matrices and 5.56, 10.91 and 12.29 for diagonal matrices. This speedup behavior occurred because the amount of computation with consistent matrices was greater within the parallel regions, as observed from the runtimes.



(a) $P = 1$ - 65442 DOFs.



(b) $P = 2$ - 474670 DOFs.

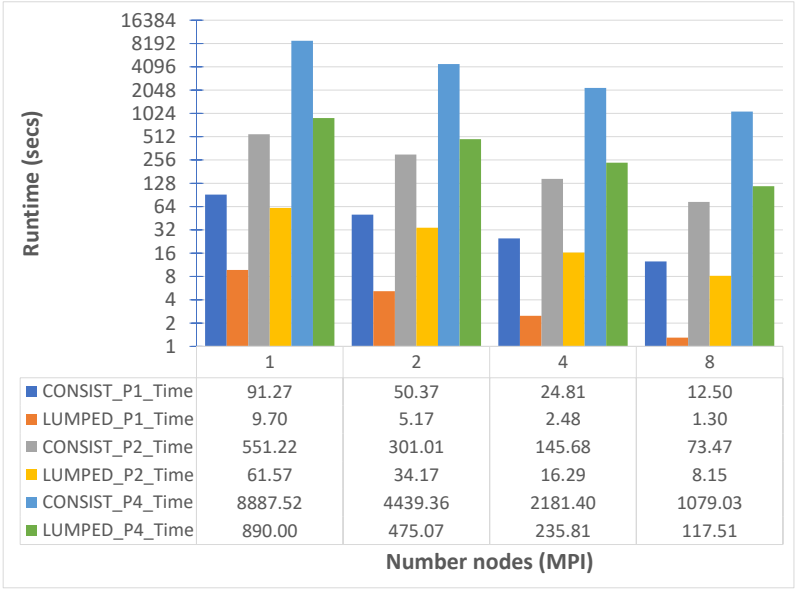


(c) $P = 4$ - 3606690 DOFs.

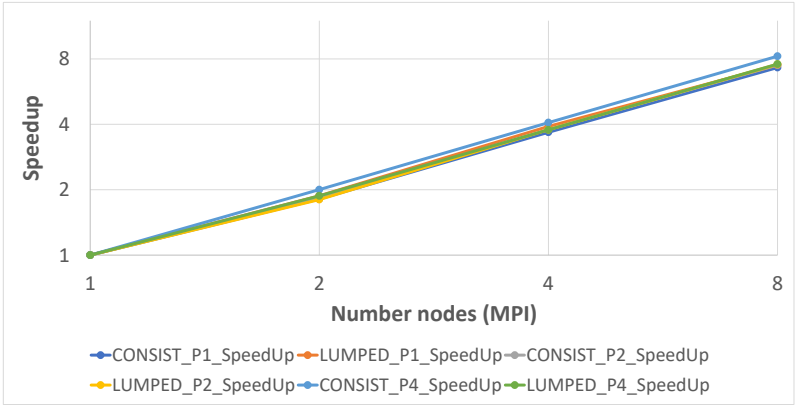
Figure 6.38. Results of runtime (in seconds) and speedup for **consistent** and **lumped** mass matrices using the crankshaft mesh with 17810 hexahedrons and polynomial orders 1, 2 and 4. Results for 1 compute node with 20 cores using 1 or 2 threads per core, totalizing 40 OpenMP threads.

The results of scalability and performance with MPI processes only are presented in Figures 6.39(a) and 6.39(b). Although the speedup results with the consistent mass matrix are slightly better for $P = 4$, the lumped matrix algorithm has reduced execution time, as shown in Figure 6.39(a), close to 10 times faster. In addition, speedups of Figure 6.39(b) shows that

both cases are scalable. In summary, lumped element mass matrices are more advantageous for reducing runtime and memory demand using both MPI and OpenMP processes.



(a) Elapsed time.



(b) Speedup.

Figure 6.39. Results of the MPI version comparing **consistent** and **lumped** mass matrices for the crankshaft mesh of 17810 hexahedrons and polynomial orders 1, 2 e 4 for up to 8 compute nodes and 1 MPI process per node.

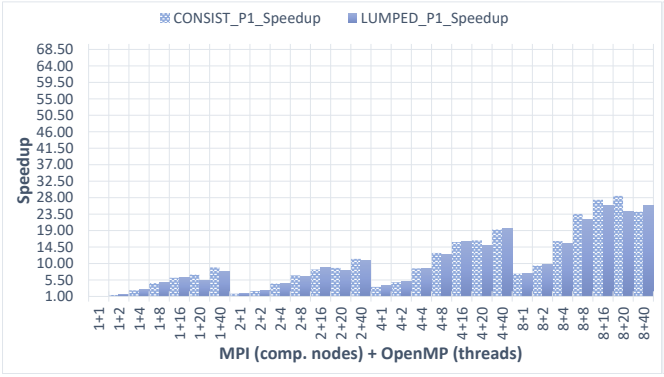
MPI + OpenMP hybrid parallel speedup for consistent and lumped mass matrices and polynomial orders 1, 2 and 4 is presented in Figures 6.40(a), 6.40(b) and 6.40(c). Figure 6.40(d), shows the maximum speedup achieved for each compute node in both cases.

As observed for the OpenMP case, larger polynomial orders, and consistent matrix also achieved greater scalability for hybrid parallelism. The OpenMP threads compute the equations by the element, and consequently, the increase in polynomial order impacts the amount of computation within the parallel region for consistent matrices. Consequently, the computation time is longer than the synchronization time for the consistent mass matrices compared to lumped ones.

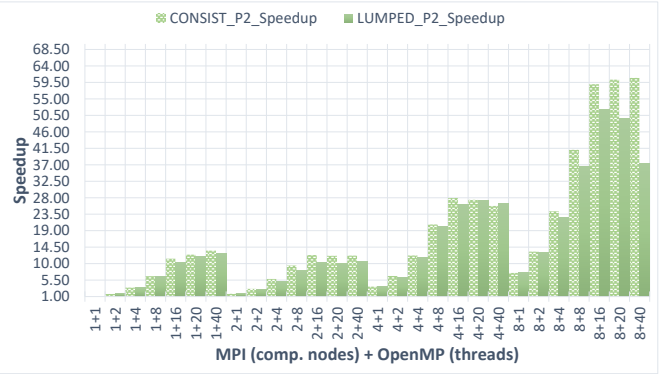
Table 6.10 – Best reduced time for lumped and consistent element mass matrices with parallel solver using MPI + OpenMP.

Runtime (secs) / Number of Processes	P1		P2		P4	
	LUMPED	**CONSIST**	**LUMPED**	**CONSIST**	**LUMPED**	**CONSIST**
Serial Runtime	9.7037	91.2671	61.5670	551.2162	889.9984	8887.5238
Best Runtime reduced	0.3735	3.1979	1.1788	9.0691	18.1117	139.7773
Combination MPI + OpenMP	8 + 16	8 + 16	8 + 16	8 + 40	8 + 16	8 + 16

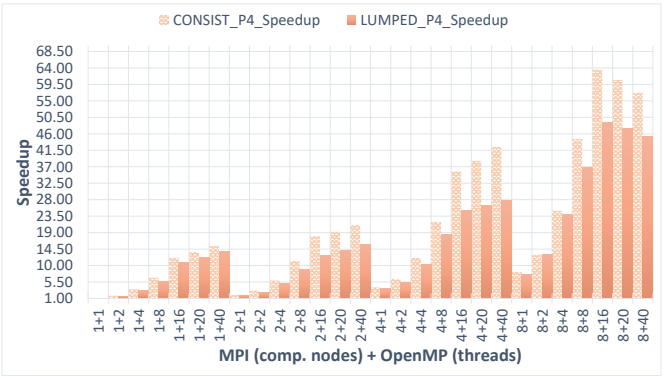
However, the diagonal matrix algorithm is faster as illustrated in Figure 6.39 and Tables 6.6 and 6.8. Table 6.10 also shows the runtime with OpenMP parallelism. Table 6.10 shows the best time reduction achieved and the number of MPI and OpenMP processes for 1, 2 and 4 polynomial orders using the 17 810 element mesh. The reduced time with hybrid parallelism using MPI + OpenMP remained advantageous with the use of lumped matrices compared to consistent ones.



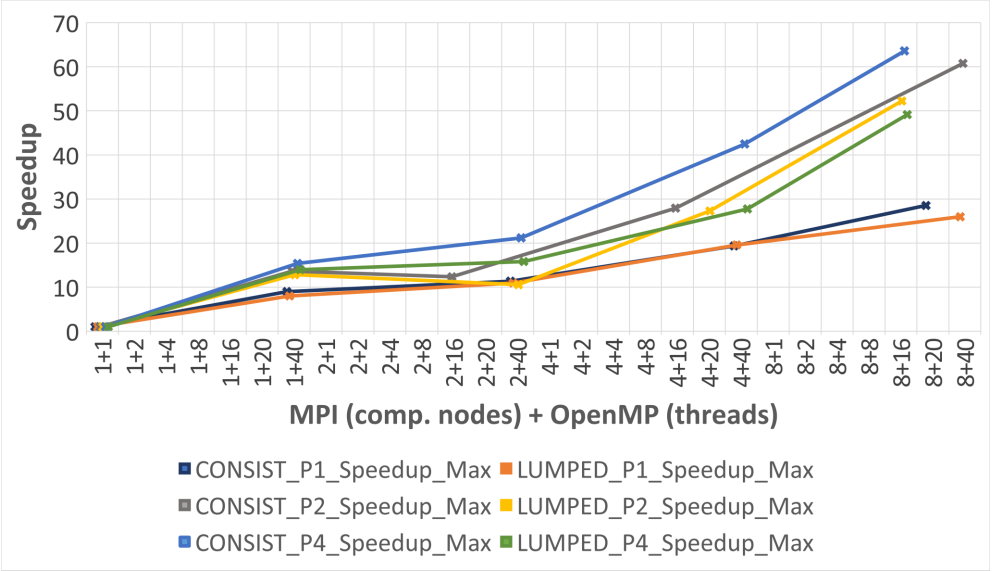
(a) $P = 1$.



(b) $P = 2$.



(c) $P = 4$.



(d) Maximum speedup - polynomials orders 1, 2 and 4.

Figure 6.40. Speedup for MPI+OpenMP with **consistent** and **lumped** mass matrices for the crankshaft mesh with 17810 hexahedrons. The results in (a), (b), (c) and (d) are for $[1,8]$ compute nodes with 1 MPI process per node, $[1,20]$ cores per node and 1 or 2 threads per core. The total of OpenMP threads is up to 40. (a), (b) and (c) show hybrid speedup for polynomial orders 1, 2 and 4, respectively. (d) shows the maximum speedup for each compute node and polynomial order.

6.8.6 Hybrid scalability analysis using h and p refinements with lumped mass matrices

Figure 6.41 presents the comparison of the OpenMP version for the crankshaft meshes with 17 810 and 73 800 elements, using 1 compute node with 20 cores and 1 or 2 threads per node, running up to 40 OpenMP threads. The mesh of 17 810 had speedup closer to the ideal for $P = 1$ and $P = 2$, while the mesh with 73 800 elements had better scalability for $P = 4$.

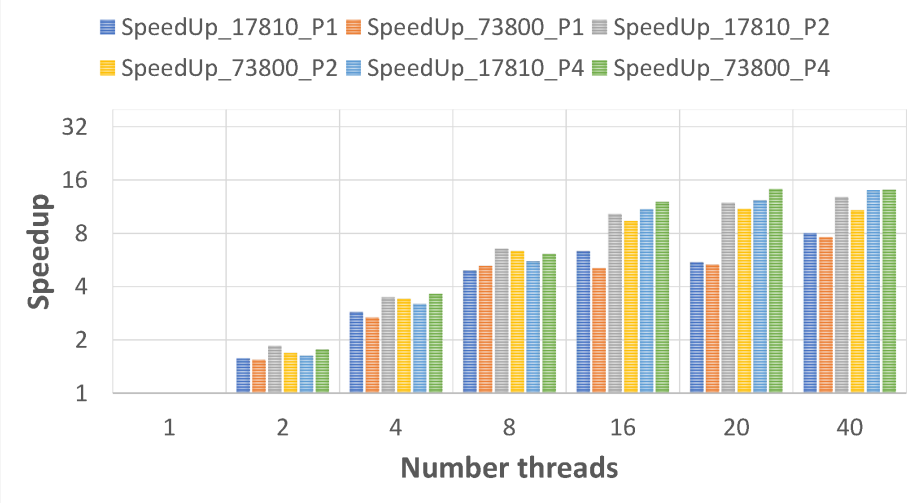


Figure 6.41. Speedup of OpenMP parallelism with meshes of 17 810 and 73 800 elements for polynomial orders 1, 2 and 4 for 1 compute node with 20 cores and 1 or 2 threads per core, totalizing 40 OpenMP threads.

Despite memory reduction with lumped mass matrices, it is important to evaluate the use of 1 MPI process per compute node to verify the distribution of elements and scalability. Figure 6.42 presents speedups with MPI processes for meshes of 17 810 and 73 800 elements and all polynomial orders. The number of compute nodes was varied in the $[1, 30]$ range with 1 MPI process per node. The speedup is higher for the mesh of 73 800 elements, and $P = 4$.

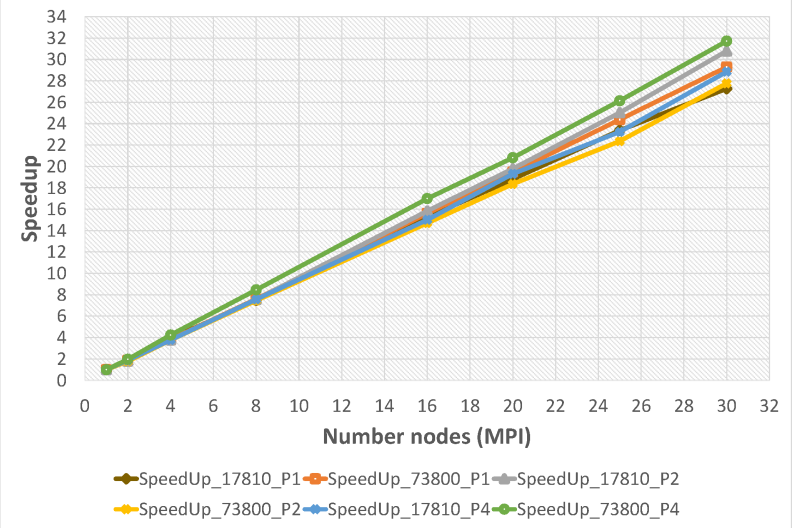
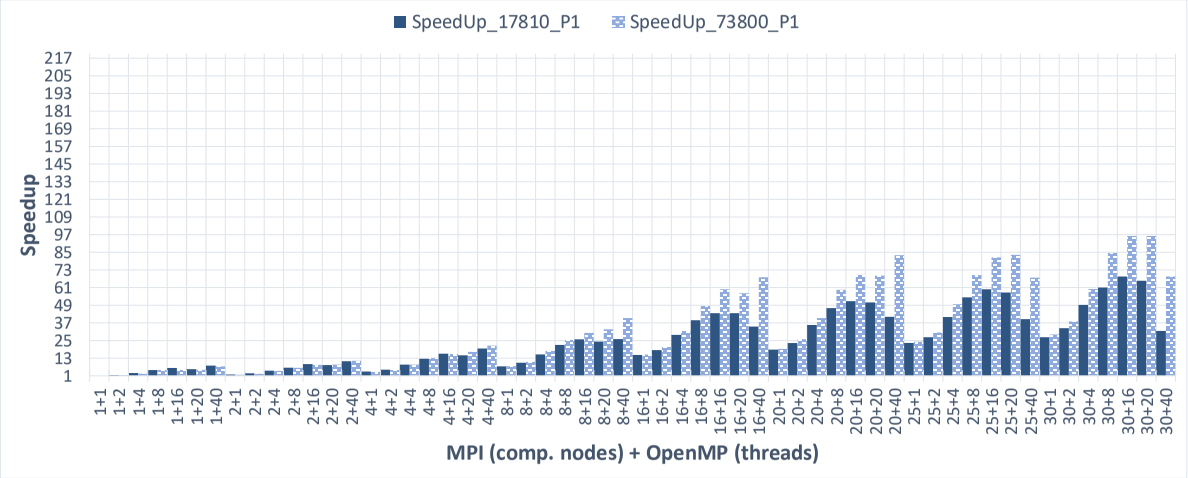
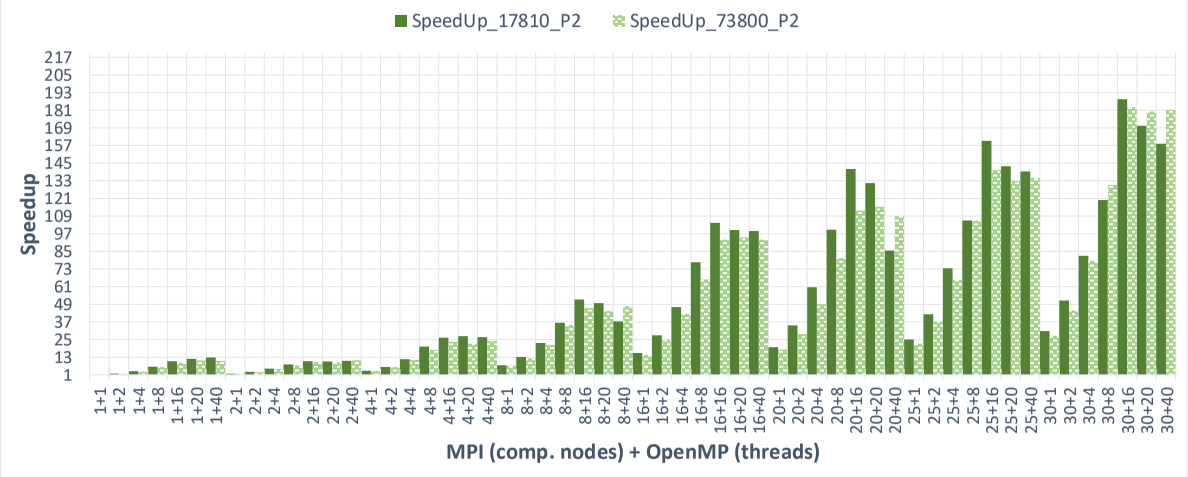


Figure 6.42. Speedup using MPI only comparing meshes of 17 810, and 73 800 elements for polynomial orders 1, 2 and 4 and $[1,30]$ compute nodes with 1 MPI process per node.

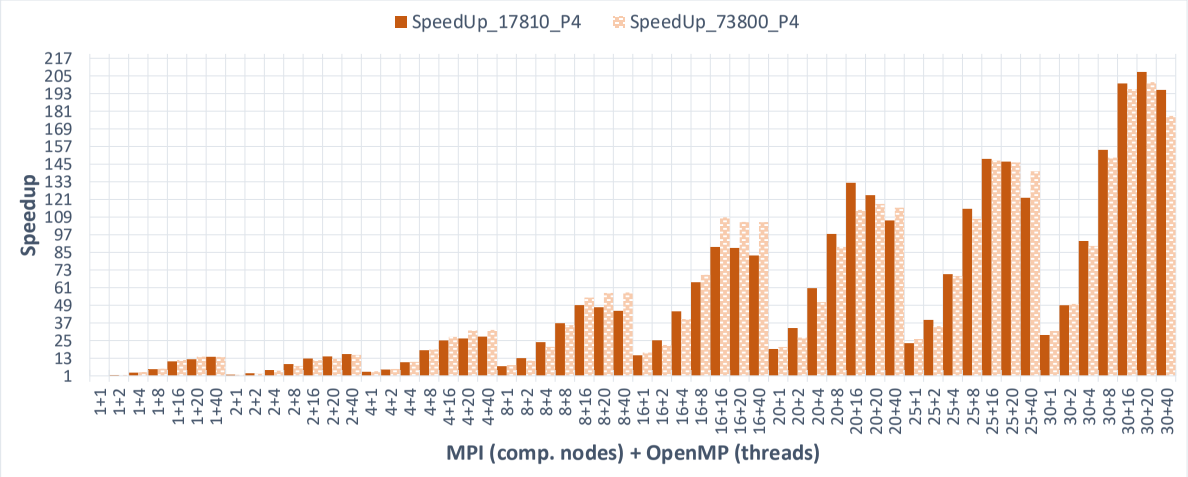
Figures 6.43(a), 6.43(b), 6.43(c) and 6.44 present the speedup of the MPI + OpenMP hybrid version for polynomial orders 1, 2 and 4 and the same meshes. Analyzing Figure 6.43, the algorithm had better speedup varying only the mesh size for $P = 1$, corresponding to the h-version of the FEM. On the other hand, by increasing the polynomial order for both meshes, similar speedups were obtained. Figure 6.44 shows the maximum speedups on each compute node (MPI process). The coarsest mesh reached the largest speedup when the polynomial order is greater than or equal to 2.



(a) $P = 1$.



(b) $P = 2$.



(c) $P = 4$.

Figure 6.43. Speedup using the MPI+OpenMP hybrid version comparing meshes with $17\,810$ and $73\,800$ elements for polynomial orders 1, 2 and 4. The results are run for $[1,30]$ computes nodes with 1 MPI process per node, $[1,20]$ cores per node and 1 or 2 threads per core. The total of OpenMP threads is up to 40.

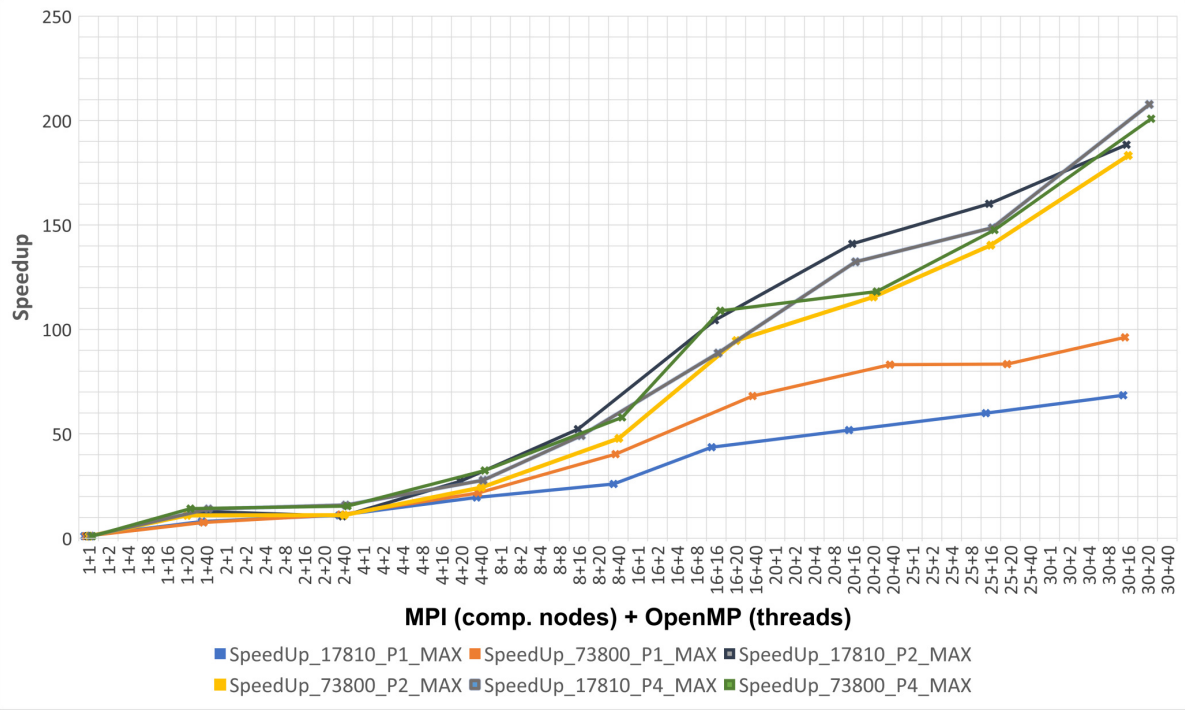


Figure 6.44. Maximum speedup using the MPI+OpenMP hybrid version comparing meshes with 17 810, and 73 800 elements for polynomial orders 1, 2 and 4. The results are run with $[1,30]$ computes nodes with 1 MPI process per node, $[1,20]$ cores per node, and 1 or 2 threads per core. The total of threads OpenMP is up to 40.

Table 6.11 complements Table 6.8 comparing the runtimes of the hybrid parallelism for the mesh of 73 800 hexahedrons. The runtimes are for each compute node number and for the different number of OpenMP threads, highlighting in green the best result for each compute node (MPI processes). The runtime decay remained for the more refined mesh. For h refinement, scalability was greater using the 73 800 mesh with a reduction of 96.2 times compared to 69.4 obtained with the 17 810 for 30 MPI processes and 16 OpenMP threads.

Table 6.11 – Runtime of the hybrid parallelism for lumped matrices using the mesh with 73 800 hexahedrons.

MPI+OpenMP	P1_ElapsedTime	P2_ElapsedTime	P4_ElapsedTime
1 + 1	41.50	256.26	4122.30
1 + 2	26.94	152.46	2338.66
1 + 4	15.55	75.61	1138.88
1 + 8	7.95	40.53	675.52
1 + 16	8.20	27.41	343.74
1 + 20	7.83	23.37	289.73
1 + 40	5.47	23.83	293.50
2 + 1	21.97	144.14	2092.21
2 + 2	16.07	89.32	1584.10
2 + 4	9.18	50.76	878.21
2 + 8	6.27	34.28	521.66
2 + 16	4.79	25.65	350.77
2 + 20	4.70	26.25	321.57
2 + 40	3.65	23.10	267.09
4 + 1	10.88	68.00	971.79
4 + 2	8.22	40.22	711.89
4 + 4	4.71	22.47	383.84
4 + 8	3.16	14.16	215.63
4 + 16	2.61	10.79	148.57
4 + 20	2.37	11.47	128.60
4 + 40	1.91	10.52	127.22
8 + 1	5.46	34.36	487.31
8 + 2	3.89	20.81	358.30
8 + 4	2.29	11.91	197.06
8 + 8	1.62	7.34	115.83
8 + 16	1.37	5.49	75.75
8 + 20	1.27	5.72	71.63
8 + 40	1.03	5.36	71.23
16 + 1	2.66	17.43	242.59
16 + 2	2.01	10.36	185.99
16 + 4	1.31	6.01	103.85
16 + 8	0.86	3.88	58.97
16 + 16	0.69	2.75	37.85
16 + 20	0.72	2.71	38.92
16 + 40	0.61	2.75	39.03
20 + 1	2.13	13.96	198.03
20 + 2	1.60	8.85	151.65
20 + 4	1.03	5.22	80.49
20 + 8	0.70	3.17	46.37
20 + 16	0.59	2.27	36.13
20 + 20	0.60	2.22	34.91
20 + 40	0.50	2.35	35.67
25 + 1	1.70	11.46	157.68
25 + 2	1.35	6.85	118.15
25 + 4	0.83	3.90	59.66
25 + 8	0.60	2.42	38.18
25 + 16	0.51	1.83	27.93
25 + 20	0.50	1.93	28.17
25 + 40	0.61	1.90	29.39
30 + 1	1.42	9.23	129.95
30 + 2	1.09	5.70	82.26
30 + 4	0.69	3.26	46.02
30 + 8	0.49	1.97	27.56
30 + 16	0.43	1.40	21.02
30 + 20	0.43	1.42	20.53
30 + 40	0.60	1.41	23.16

The runtime of the element wise explicit algorithm using lumped matrices with the mesh of 73 800 hexahedrons. The runtime is showed for each compute node (MPI processes) and each OpenMP thread. The serial code runtime is highlighted in yellow, and the best time for each node is highlighted in green color.

6.8.7 Sizeup analyzes for crankshaft meshes

This section presents an analysis taking into consideration the mesh sizes, polynomial orders, and runtimes. The element wise algorithm is run with 20 computational nodes, 20 cores, and 2 threads per core, with 1 MPI rank per node and 2 OpenMP threads per core or 40 OpenMP threads per compute node. The tests were performed with the meshes of 17 810, 73 800, 327 181, 580 781 and 1 789 811 elements with polynomial orders 1, 2, 4 and 6; order 6 was used only with the meshes of 17 810 and 73 800 elements.

Table 6.12 shows the relationship between the number of DOFs and runtime for different polynomial orders (p-refinement) and mesh sizes (h-refinement). The relationship is based on the Sizeup metric defined in Equation (3.5) and presented in the last column of the table. For this analysis, Sizeup is defined as the ratio of DOF growth and runtime rates, that is,

$$Sizeup = \frac{\frac{NDOFs(M)}{NDOFs(1)}}{\frac{T_p(NDOFs(M))}{T_p(NDOFs(1))}}, \tag{6.3}$$

where $NDOFs(1)$ is the number of DOFs for the coarsest mesh with $P = 1$, $NDOFs(M)$ is the number of DOFs for the next M cases with mesh size and polynomial orders larger than or equal to $NDOFs(1)$; $T_p(NDOFs(M))$ and $T_p(NDOFs(1))$ are the parallel runtime for $NDOFs(M)$ and $NDOFs(1)$, respectively. Table 6.12 presents also the ratios

$$\frac{NDOFs(M)}{NDOFs(1)} \tag{6.4}$$

and

$$\frac{T_p(NDOFs(M))}{T_p(NDOFs(1))} \tag{6.5}$$

in columns "DOFs Rate" e "Runtime Rate", respectively.

From Equation (6.3), it can be assumed that if $T_p(NDOFs(M)) \geqslant T_p(NDOFs(1))$, Sizeup will be on the interval $\left[0, \frac{NDOFs(M)}{NDOFs(1)}\right]$. In addition, as Sizeup goes to 1, the growth rate of the number of DOFs and runtime will be the same.

Figure 6.45 shows the Sizeup values in terms of the number of DOFs with bars. For all $P = 2$ cases, $Sizeup \geq 2$. For these cases, the DOF growth rate was greater, $[2.0, 2.63]$ larger than the runtime growth rate. There was the opposite behavior for $P = 6$, and the runtime growth rate was larger than the DOF growth rate. For $P = 4$, these rates are similar, with Sizeup approaching 1. However, the closest case between the rates was with the mesh of 1 789 811, and $P = 1$. The case with the largest number of DOFs, over 350 million, had a Sizeup of 1.4.

Table 6.12 – Sizeup analyses for the crankshaft meshes.

N. Elements	Order	N. DOFs	Runtime (s)	DOFs Rate	Runtime Rate	Sizeup
17810	P1	75559	0.2359	1.0000	1.0000	1.0000
73800	P1	276097	0.4996	3.6541	2.1178	1.7254
17810	P2	512328	0.7202	6.7805	3.0530	2.2209
327181	P1	1126440	2.0492	14.9081	8.6867	1.7162
580781	P1	1957140	3.3599	25.9021	14.2429	1.8186
73800	P2	1984444	2.3499	26.2635	9.9614	2.6365
17810	P4	3751846	8.3392	49.6545	35.3506	1.4046
1789811	P1	5860099	17.7185	77.5566	75.1102	1.0326
327181	P2	8423351	10.9673	111.4804	46.4913	2.3979
17810	P6	12283204	45.9810	162.5644	194.9173	0.8340
580781	P2	14786716	19.8754	195.6976	84.2535	2.3227
73800	P4	15010750	35.6711	198.6626	151.2128	1.3138
1789811	P2	44903317	67.4981	594.2815	286.1301	2.0770
73800	P6	49705872	254.3951	657.8418	1078.4023	0.6100
327181	P4	65083533	148.4559	861.3604	629.3171	1.3687
580781	P4	114878382	314.5824	1520.3799	1333.5413	1.1401
1789811	P4	351394243	773.4529	4650.5941	3278.7321	1.4184

Results are run with 20 compute nodes using 1 rank MPI per node, 20 cores with 2 threads per core, totalizing 40 threads OpenMP.

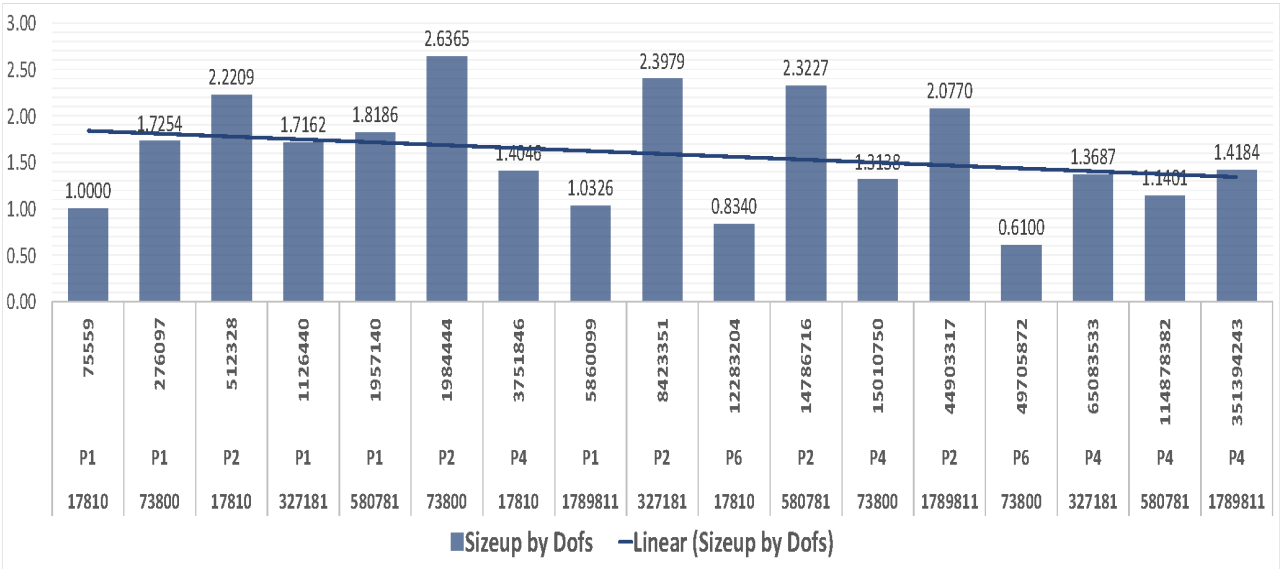


Figure 6.45. Sizeup by DOFs using the crankshaft meshes with 17810, 73800, 327181, 580781 and 1789811 elements with polynomial orders 1, 2, 4 and 6; 20 compute nodes with 1 rank MPI per node, 20 cores per node and 2 threads per core, totalizing 40 OpenMP threads.

Figure 6.46 shows the Sizeup metric where problem size of Equation (3.5) is the number of elements and not the number DOFs. Based on that the size growth rate will always be the same for polynomial orders 1, 2 and 4 and equal to 1.0, 4.1, 18.4, 32.6 and 100.5 for

meshes with 17 810, 73 800, 327 181, 580 781 and 1 789 811 elements, respectively. The runtime rate (Runtime Rate column) will be different, taking into consideration the column "Runtime (s)" in Table 6.12. As in the previous analysis, the growth rate of elements and execution time will be equal if Sizeup is 1. Figure 6.46 shows that increasing polynomial orders, the runtime rate also increases and consequently, Sizeup approaches 1 for $P \geq 2$.

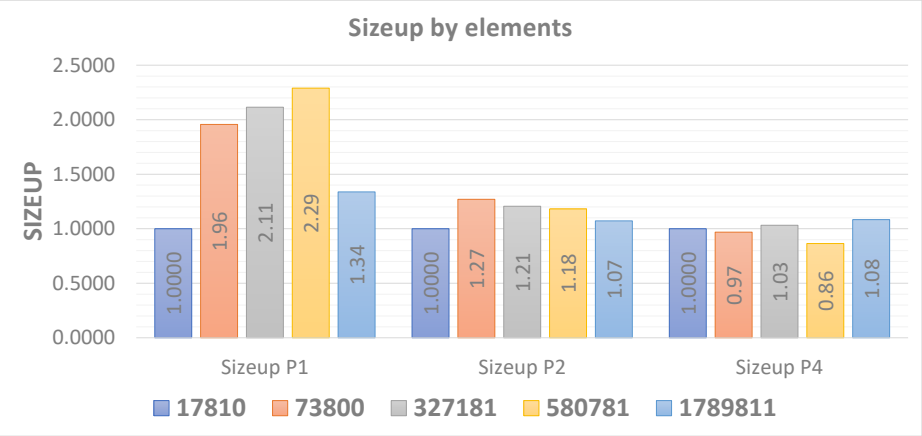


Figure 6.46. Sizeup by elements using the crankshaft meshes with 17 810, 73 800, 327 181, 580 781 and 1 789 811 elements with polynomial orders 1, 2, 4 and 6; 20 compute nodes with 1 rank MPI per node, 20 cores per node and 2 threads per core, totalizing 40 threads OpenMP.

7 CONCLUSIONS

We designed the architecture of $(hp)^2$FEM to be flexible and easy to extend because of its generic and modular classes. The concepts of encapsulation, virtual methods, and generalization are used in some components of the $(hp)^2$FEM software. Encapsulation and generalization are used in the basic modules, which implement different classes for data structures, polynomial interpolation, numerical integration and solving linear systems of equations. In the implementation of parallelism, methods and modules were added to the $(hp)^2$FEM serial architecture without requiring modifications of the architecture. The parallel version works as a conditional path to be selected according to the options indicated in the input files.

Profiling, programming techniques, and optimized linear algebra libraries such as BLAS and LAPACK were used to improve the performance of the code. In particular, the D1-Matrices procedure reduced the amount of memory required by the shape functions by a factor of 4 000, as presented in Section 6.3.

In addition, optimization of memory usage was also performed through the implementation of lumped element mass matrices in the explicit transient solver. Consequently, it was possible to process meshes with a greater number of elements and higher polynomial orders. The use of these matrices also allowed better use of compute node resources. For example, it was possible to execute more than one MPI process by distributing the mesh in the same node, associating one MPI process per core.

The p-non-uniform architecture allows polynomial orders to be varied for elements. A simple projection problem was solved to illustrate the validation of data structures. It is expected that the better quality solutions be obtained for applications with larger numbers of elements and using error estimators to determine the optimal distribution of element orders.

The work presented different parallel approaches to decomposition, storage and treatment of mesh topology data and communication in neighboring partitions. The class **PartitionData** was created to manage storage and local data communication between mesh subdomains. The class contains pre-processing data structures that helped in the communication between partitions for the solution methods. Such pre-processing structures can be used for new parallel solvers. The parallel version encapsulates the communication structure between partitions, so a new solver only needs to pass as parameter the solution vectors at the interfaces of subdomains.

Deadlock communication problems in MPI processes using more refined meshes and compute nodes were detected on the IBM Blue Gene computer when running the element-wise projection solver with more than 1 024 compute nodes. This work used the blocking communication algorithm between MPI processes, which includes constructing a color map to de-

fine the communication topology of mesh subdomains.

Scalability comparisons of $(hp)^2$FEM parallel algorithms were performed on IBM PowerPC and Intel architectures, using, respectively, the IBM Blue Gene/P and Q computers and the Kahuna cluster. Direct and iterative element wise solver were used to evaluate the different parallel structures for data storage, distribution and communication. In summary, the best results were obtained with the use of the multilevel k-way partitioning algorithm for the input mesh of degree 1 and the generation of high-order information in each subdomain. In addition, the use of pre-processed data structures in the class **PartitionData** and the mapping algorithm of subdomain topology favoured and reduced the runtime in the message exchange between partitions. Only solution vectors are exchanged without the need of index vectors to update the solutions at the interfaces between subdomains.

We also showed that the element wise parallel projection solver has weak scalability in the IBM Blue Gene/Q computer. In this case, the efficiency was approximately 80% with respect to the number of processors and mesh size. We also obtained the same efficiency when solving a problem with 840 million of DOFs using 2/3 of the compute nodes in the IBM Blue Gene/Q Mira computer.

We also evaluated the MPI + OpenMP hybrid parallelism with Intel architecture using the element wise explicit solvers for linear and non-linear problems. The scalability using these explicit solvers was close to the ideal speedup combining MPI processes and OpenMP threads, reaching a minimum efficiency of 72% for structured meshes.

Finally, the thesis also presentsd good results for distribution and partitioning of more complex and unstructured meshes of a crankshaft. Scalability is presented for the element wise linear transient solver using MPI + OpenMP hybrid parallelism. When combining MPI and OpenMP for more than one compute node, the shared memory feature reduced significantly the runtime by 200 times with 30 MPI processes and 20 OpenMP threads. However, the scalability is not close to ideal due to the solver's iterative feature and increased thread synchronization time when updating solutions. The use of only MPI processes with distribution by cores in the same compute node presented scalability close to ideal. In summary, cases of higher polynomial orders presented better scalability, including hybrid parallelism. The largest case evaluated was for the mesh of 1 789 811 elements, $P = 4$ and 351 394 243 DOFs. The serial runtime estimate is 23 hours. Using the parallel version, runtime was reduced to 12.22 minutes using 20 compute nodes and 20 cores of Kahuna cluster.

The next steps in the development of $(hp)^2$FEM focus on obtaining good performance on hybrid CPU/GPU platforms. In this case, the use of high throughput and storage density architectures such as IBM Power10+ becomes essential. Furthermore, OpenMP 5.0 or OpenCL may be used to allow for implementation on graphics cards and vectorized processing using single instruction multiple data (SIMD) parallelism.

The implementation of parallel versions of the element wise solvers with error estimators and h and p refinements is also important. In this this case, a new partitioning and redistribution of elements can be made using METIS by weighting elements that have higher polynomial order in the mesh. Other element wise implicit time integration methods have been also implemented.

REFERENCES

ALCF.ANL.GOV. **Argonne Leadership Computing Facility**. 2014. Available at: <https://www.alcf.anl.gov/>.

ALLINEA, D. **Allinea DDT and MAP User Guide**. [S.l.], 2014. 196 p.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: ACM. **Proceedings of the April 18-20, 1967, spring joint computer conference**. [S.l.], 1967. p. 483–485.

ANZT, H. *et al.* Hiflow3 – a flexible and hardware- aware parallel finite element package. **Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)**, 2010.

Appel, K.; Haken, W. The Solution of the Four-Color-Map Problem. **Scientific American**, v. 237, p. 108–121, Oct. 1977.

ARGONNE, L. C. F. **"User Guide - Performance Tools & APIs", in www.alcf.anl.gov/user-guides/performance-tools-apis, Last accessed: May, 2016**. 2015.

AUGUSTO, R. A. **Arquitetura de Software Orientada por Objetos para o Método de Elementos Finitos de Alta Ordem**. Phd Thesis (PhD Thesis) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, 2012.

BANGERTH, W.; HARTMANN, R.; KANSCHAT, G. Deal.II — a general purpose object oriented finite element library. **ACM Transactions on Mathematical Software**, v. 33, n. 4, p. 151–155, 2007.

BARGOS, F. F. **Implementação de Elementos Finitos de Alta Ordem baseado em Produto Tensorial**. Master's Thesis (Master's Thesis) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, 2009.

BATHE, K. **Finite Element Procedures**. New-Jersey: Prentice Hall, 1996.

BITTENCOURT, M. *et al.* **An Object-oriented Interactive Environment for Structural Analysis and Optimization**. Rio de Janeiro, 1998.

BITTENCOURT, M. L. Applying C++ templates to the development of finite element classes. **Engineering Computations**, v. 17, n. 6/7, p. 775–788, 2000.

BITTENCOURT, M. L. **Computational Solid Mechanics: Variational Formulation and High Order Approximation**. [S.l.]: CRC Press, 2014.

BITTENCOURT, M. L.; FURLAN, F. An elementwise least square approach for explicit integration methods applied to elasticity. In: **MecSol 2011 – International Symposium on Solid Mechanics, Solid Mechanics in Brazil 2011**. Florianópolis, Santa Catarina, Brazil: Brazilian Society of Mechanical Sciences and Engineering, 2011. p. 63–76. ISBN 978-85-85769-46-8.

BITTENCOURT, M. L.; VAZQUEZ, M.; VAZQUEZ, T. Construction of shape functions for the $h-$ and $p-$ versions of the FEM using tensorial product. **Int. J. Numer. Meth. in Eng.**, v. 71, n. 5, p. 529–563, 2007.

BITTENCOURT, M. L.; VAZQUEZ, T. Tensor-based Gauss-Jacobi numerical integration for high-order mass and stiffness matrices. **Int. J. Numer. Meth. in Eng.**, v. 79, n. 5, p. 599–638, 2009.

BLATT, M.; BASTIAN, P. On the generic parallelisation of iterative solvers for the finite element method. **International Journal of Computational Science and Engineering**, Inderscience Publishers, v. 4, n. 1, p. 56–69, 2008.

BOARD., O. A. R. **OpenMP Application Program Interface**. [S.l.], 2013. 320 p.

BUNGARTZ, H. *et al.* **Software for Exascale Computing - SPPEXA 2016-2019**. [S.l.]: Springer, 2020.

CANTWELL, C. *et al.* Nektar++: An open-source spectral/hp element framework. **Computer Physics Communications**, Elsevier, v. 192, p. 205–219, 2015.

CANTWELL, C. D. *et al.* From h to p efficiently: selecting the optimal spectral/hp discretization in three dimensions. **Mathematical Modelling of Natural Phenomena**, v. 6, p. 84–96, 2011.

CCES. **CCES Unicamp – CENTER FOR COMPUTING IN ENGINEERING & SCIENCES**. 2020. Available at: <http://cces.unicamp.br/>.

CHAN, J. *et al.* GPU-accelerated discontinuous galerkin methods on hybrid meshes. **Journal of Computational Physics**, v. 318, p. 142 – 168, 2016. ISSN 0021-9991.

CHARTRAND, G.; ZHANG, P. **Chromatic graph theory**. [S.l.]: CRC press, 2008.

CLEMENTS, P. *et al.* **Documenting software architectures : views and beyond**. 2nd. ed. [S.l.]: Pearson Education, Inc., 2010.

DEVLOO, P. Pz: An object oriented environment for scientific programming. **Computer Methods in Applied Mechanics and Engineering**, v. 150, n. 1, p. 133 – 153, 1997. Symposium on Advances in Computational Mechanics.

DEVLOO, P. R.; LONGHIN, G. C. Object oriented design philosophy for scientific computing. **ESAIM: Mathematical Modelling and Numerical Analysis**, EDP Sciences, v. 36, n. 5, p. 793–807, 2002.

DONGARRA, J. **Blas Lapack Users Guide**. 2. ed. Philadelphia, 2003.

DUNN, G. G. L. M. a. I. N. **A Parallel Algorithm Synthesis Procedure for High-Performance Computer Architectures**. 1. ed. [S.l.]: Springer US, 2003. (Series in Computer Science). ISBN 978-1-4613-4658-6,978-1-4419-8650-4.

ESRD. **StressCheck Master Guide**. 9. ed. Missiori, USA, 2009.

FERNANDES, F. G. **Um arcabouço para verificação automática de modelos UML**. Master's Thesis (Master's Thesis), Pontifícia Universidade Católica de Minas Gerais, 2011.

FERREIRA, M. K. **Mapeamento Estático de Processos MPI com Emparelhamento Perfeito de Custo Máximo em Cluster Homogêneo de Multi-cores**. 72 p. Master's Thesis (Master's Thesis) — UFGRS - Universidade Federal do Rio Grande do Sul, 2012.

FERREIRA, R. R. **Caracterização de Desempenho de uma Aplicação Paralela do Método dos Elementos Finitos em Ambientes Heterogêneos de PCs**. 137 p. Phd Thesis (PhD Thesis), 2006.

FU, C. Parallel computing for finite element structural analysis on workstation cluster. **International Conference on Computer Science and Software Engineering**, p. 291–294, 2008.

FURLAN, F. A. C. **Métodos Locais de Integração Explícito e Implícito aplicados ao Método de Elementos Finitos de Alta Ordem**. Master's Thesis (Master's Thesis) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, 2011.

GARLAN, D.; SHAW, M. **An Introduction to Software Architecture**. Pittsburgh: World Scientific Publishing Company, 1994.

GASTON, D. *et al.* Moose: A parallel computational framework for coupled systems of nonlinear equations. **Nuclear Engineering and Design**, Elsevier, v. 239, n. 10, p. 1768–1778, 2009.

"GETFEM++, an open-source finite element library", in http://download.gna.org/getfem/html/homepage/, Last accessed: May 2016.

GILGE, M. **IBM system Blue Gene solution Blue Gene/Q application development**. Armonk: IBM Redbooks, 2014.

GILGE, M. **IBM system blue gene solution blue gene/Q application development**. [S.l.]: IBM Redbooks, 2014.

GROPP EWING LUSK, A. S. W. **Using MPI: Portable Parallel Programming with the Message-Passing Interface**. third edition. [S.l.]: The MIT Press, 2014. (Scientific and Engineering Computation). ISBN 0262527391,9780262527392.

GUEDES, M. J. M. **Paralelização da Resolução de Edps pelo Método Hopscotch Utilizando Refinamento Adaptativo e Balanceamento Dinâmico de Carga**. 136 p. Phd Thesis (PhD Thesis) — UFF - Universidade Federal Fluminense, 2009.

GUIMARÃES, A.; FEIJÓO, R. **The ACDP System (in Portuguese)**. Rio de Janeiro, Brazil, 1989.

GUSTAFSON, J. L. Reevaluating amdahl's law. **Communications of the ACM**, ACM, v. 31, n. 5, p. 532–533, 1988.

HAGER, G.; WELLEIN, G. **Introduction to high performance computing for scientists and engineers**. [S.l.]: CRC Press, 2010.

Haring, R. *et al.* The ibm blue gene/q compute chip. **IEEE Micro**, v. 32, n. 2, p. 48–60, 2012.

IBM. **ESSL Guide and Reference**. 5. ed. [S.l.], 2012.

IBM - INTERNATIONAL BUSINESS MACHINES CORPORATION. **ESSL Guide and Reference**. 5. ed. Poughkeepsie, 2012.

JALOTE, P. **An integrated approach to software engineering**. Kanpur: Springer Science & Business Media, 2012.

KAMINSKY, A. **Big CPU, Big Data: Solving the World's Toughest Computational Problems with Parallel Computing**. CreateSpace Independent Publishing Platform, 2016. ISBN 9781534872288. Available at: <https://books.google.com.br/books?id=ZTY3vgAACAAJ>.

KARNIADAKIS, G. E.; SHERWIN, S. **Spectral/hp Element Methods for Computational Fluid Dynamics**. Oxford: Oxford University Press, 2005.

KARYPIS, G. **A Software Package for Partitioning Unstructured Graphs , Partitioning Meshes , and Computing Fill-Reducing Orderings of Sparse Matrices**. [S.l.], 2011. 1–34 p.

KAWABATA, C. L. O.; VENTURINI, W. S.; CODA, H. B. Desenvolvimento e implementação de um método de elementos finitos paralelo para análise não linear de estruturas. **Cadernos de Engenharia de Estruturas, São Carlos.**, v. 11, p. 151–155, 2009.

KIRK, B. S. *et al.* libmesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. **Engineering with Computers**, Springer, v. 22, n. 3-4, p. 237–254, 2006.

KLEIN, P. N.; LU, H. I.; NETZER, R. H. B. Detecting Race Conditions in Parallel Programs that Use Semaphores. **Algorithmica**, v. 35, n. 4, p. 321–345, Apr. 2003. ISSN 0178-4617. Available at: <http://link.springer.com/10.1007/s00453-002-1004-3>.

LEVINE, J. **Flex & Bison: Text Processing Tools**. Sebastopol: " O'Reilly Media, Inc.", 2009.

MACKERLE, J. Object-oriented programming in FEM and BEM: a bibliography (1990–2003). **Advances in Engineering Software**, v. 35, n. 6, p. 325–336, 2004.

MARR, D. Hyper-threading technology in the netburst® microarchitecture. **14th Hot Chips**, 2002.

MASUERO, J. **Computação paralela na análise de problemas de engenharia utilizando o Método dos Elementos Finitos**. 255 p. Phd Thesis (PhD Thesis) — UFRGS - Universidade Federal do Rio Grande do Sul., 2009. Available at: <http://www.lume.ufrgs.br/handle/10183/16874>.

MEUER, H. *et al.* **"Top 500 Supercomputer Sites", in www.top500.org/, Lasted accessed: Jun 2016**.

MIANO, M. **Tensorização de Matrizes de Rigidez para Quadrados e Hexaedros Usando o Método de Elementos Finitos de Alta Ordem**. Phd Thesis (Tese (Doutorado)) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, São Paulo, Brasil, 2009.

"NECKTAR++ - Spectral/hp Element Framework", in http://www.nektar.info/, Last accessed: May 2016. 2013.

NETTO, P. O. B. **Grafos: Teoria, Modelos, Algoritmos**. 5. ed. São Paulo: [s.n.], 2012. 314 p.

NOGUEIRA Jr., A. C. **Formulação *p* do Método de Elementos Finitos em problemas de elasticidade linear e não-linear com malhas 3D não-estruturadas e em métodos multigrid algébricos**. Phd Thesis (PhD Thesis) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, 2002.

OGUIC, R.; VIAZZO, S.; PONCET, S. A parallelized multidomain compact solver for incompressible turbulent flows in cylindrical geometries. **Journal of Computational Physics**, v. 300, p. 710 – 731, 2015. ISSN 0021-9991.

OPENMP, A. R. B. **OpenMP Application Programming Interface**. [S.l.], 2018. Available at: <http://www.openmp.org/mp-documents/openmp-4.5.pdf>.

PACHECO, P. S. **An Introduction to Parallel Programming**. San Franciso, CA, USA: Morgan Kaufmann, 2011. 391 p. ISBN 9780123814722.

QUINTERO, D. **IBM High Performance Computing Cluster Health Check**. First. Armonk, NY, U.S.A.: International Business Machines Corporation, 2014. 124 p.

RIEG, P. D. F. **Z88 The compact Finite Elements System**. [S.l.], 2014.

RODRIGUES, A. d. S. **Análise dinâmica e balanceamento de virabrequins leves de motores**. Master's Thesis (Master's Thesis) — UNICAMP - Universidade Estadual de Campinas, 2013.

ROSÁRIO, D. A. N. d. **Escalabilidade Paralela de um Algoritmo de Migração Reversa no Tempo (RTM) Pré-Empilhamento**. Master's Thesis (Master's Thesis), 2012.

SANTOS, C. F. R. dos. **Funções de Interpolação e Técnicas de Solução para Problemas de Poisson usando o Método de Elementos Finitos de Alta Ordem**. Master's Thesis (Master's Thesis) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, 2011.

SCHMIDT, B. *et al.* **Parallel programming: concepts and practice**. [S.l.]: Morgan Kaufmann, 2017.

SEVILLA, R.; FERNÁNDEZ-MÉNDEZ, S.; HUERTA, A. Nurbs-enhanced finite element method for euler equations. **International Journal for Numerical Methods in Fluids**, Wiley Online Library, v. 57, n. 9, p. 1051–1069, 2008.

SOLIN, P.; KOROUS, L.; KUS, P. Hermes2d, a C++ library for rapid development of adaptive hp-fem and hp-dg solvers. **Journal of Computational and Applied Mathematics**, Elsevier, v. 270, p. 152–165, 2014.

STALLMAN, R. M. **Using the GNU Compiler Collection**. [S.l.], 2003.

SUZUKI, J. L. **Aspectos em modelamento fracionário e bases de interpolação eficientes em simulação de materiais complexos utilizando método de elementos finitos: Aspects of fractional-order modeling and efficient bases to simulate complex materials using finite element methods**. 155 p. Phd Thesis (PhD Thesis) — Unicamp, Campinas, SP, 2017. 1 recurso online (155 p.). Tese (doutorado. Available at: <http://www.repositorio.unicamp.br/handle/REPOSIP/330675.>

VALENTE, G. L. **hp2FEM - Uma Arquitetura de *Software* p Não-Uniforme para o Método de Elementos Finitos de Alta Ordem**. Master's Thesis (Master's Thesis) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, 2012.

VANDEVOORDE, D.; JOSUTTIS, N. **C++ templates: the complete guide.** [S.l.]: Pearson Education, Inc., 2003.

VAZQUEZ, M. **Construção de funções de interpolação para as versões h e p do MEF através de produto tensorial**. Master's Thesis (Dissertação (Mestrado)) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, São Paulo, Brasil, 2004.

VAZQUEZ, T. **Funções de Base e Regras de Integração Tensorizáveis para o MEF de Alta Ordem**. Phd Thesis (Tese (Doutorado)) — Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, São Paulo, Brasil, 2008.

WALKUP, B. **Blue Gene/P System and Optimization Tips**. [S.l.], 2011.

WANG, C.; DONG, X.; SHU, C.-W. Parallel adaptive mesh refinement method based on weno finite difference scheme for the simulation of multi-dimensional detonation. **Journal of Computational Physics**, v. 298, p. 161 – 175, 2015. ISSN 0021-9991.

WANG, K.; LIU, H.; CHEN, Z. A scalable parallel black oil simulator on distributed memory parallel computers. **Journal of Computational Physics**, v. 301, p. 19 – 34, 2015. ISSN 0021-9991.

Y.YU; BITTENCOURT, M. L.; KARNIADAKIS, G. A semi-local spectral/hp element solver for linear elasticity problems. **Int. J. Numer. Meth. in Eng.**, v. 100, n. 1, p. 347–373, 2014.