Many-Approach Load-Balancing for Parallel Multi-Physics Simulations with Block-Structured Adaptive Mesh

Johnatas Teixeira de Freitas

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2025

Johnatas Teixeira de Freitas

Many-Approach Load-Balancing for Parallel Multi-Physics Simulations with Block-Structured Adaptive Mesh

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Prof. Flávio de Oliveira Silva, Ph.D.
Coorientador: Prof. João Marcelo Vedovotto

Uberlândia
2025

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

F866 2025

Freitas, Johnatas Teixeira de, 1988-
Many-Approach Load-Balancing for Parallel Multi-Physics Simulations with Block-Structured Adaptive Mesh [recurso eletrônico] / Johnatas Teixeira de Freitas. - 2025.

Orientador: Flávio de Oliveira Silva.
Coorientador: João Marcelo Vedovotto.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2025.172
Inclui bibliografia.

1. Computação. I. Silva, Flávio de Oliveira,1970-, (Orient.). II. Vedovotto, João Marcelo,1981-, (Coorient.). III. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. IV. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Computação
Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação, 03/2025, PPGCO				
Data:	28 de fevereiro de 2025	Hora de início:	13:04	Hora de encerramento:	14:46
Matrícula do Discente:	12212CCP013				
Nome do Discente:	Johnatas Teixeira de Freitas				
Título do Trabalho:	Many-Approach load-balancing for parallel multi-physics simulations with block-structured adaptive mesh				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	----------				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: João Marcelo Vedovotto - FEMEC/UFU (Coorientador), Pedro Frosi Rosa - FACOM/UFU, Antonio Castelo Filho - ICMC/USP e Flávio de Oliveira Silva - Universidade do Minho, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Flávio de Oliveira Silva - Braga/Portugal, Antonio Castelo Filho - São Carlos/SP e Pedro Frosi Rosa - Ivrea/ Itália. Os outros membros da banca e o aluno participaram da cidade de Uberlândia.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Flávio de Oliveira Silva, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação da Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir ao candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos,

conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Flávio de Oliveira Silva**, **Presidente**, em 07/03/2025, às 13:04, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Antonio Castelo Filho**, **Usuário Externo**, em 07/03/2025, às 15:02, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **João Marcelo Vedovotto**, **Professor(a) do Magistério Superior**, em 14/03/2025, às 13:29, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Pedro Frosi Rosa**, **Usuário Externo**, em 19/03/2025, às 10:00, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **6099954** e o código CRC **D4525F1C**.

Referência: Processo nº 23117.009375/2025-90 SEI nº 6099954

I dedicate this work to the Eternal Lord, who made an amazing work creating this beautiful universe and everything within

Agradecimentos

I thank the Eternal Lord. He creates everyone and everything we love to investigate by science. Without his work, none of this would be possible. I thank the family He gave me. My mother and father for raising me with love and discipline. My brother and sister for, more often than necessary, being a pain and teaching me resilience. I thank my friends and colleagues at the MFLab for their companionship, especially Prof. João Vedovotto, Millena, and Ricardo Serfaty, for all their teachings about multi-physics simulations and MFSim working. Also, thanks to Pedro and Freddy for your help running the cases and rendering beautiful images about them. I thank Prof. Flávio for your help, instructions, and patience while working with me for the last few years. I thank the PPGCO, MFLab, and Petrobrás for the opportunity to develop this research and the vital access to people and resources for this work. None of this would be possible without your support.

"Whether therefore ye eat, or drink, or whatsoever ye do, do all to the glory of God."
1 Corinthians 10:31

Resumo

Simulações multifísicas são uma importante ferramenta tanto para a academia como para a indústria, ajudando ambos a compreenderem como vários fenômenos físicos se comportam em um dado contexto. Com o crescimento do mercado de Computação de Alto Desempenho (HPC) e o consequente maior acesso a hardware mais avançado e potente, o uso extensivo de paralelização se tornou cena comum nas simulações multi-físicas. Isso cria um problema de distribuição de carga, o que nesse caso, dado a natureza já complexa das simulações multi-físicas, é um problema nada trivial. Por isso, nós propomos nesse trabalho um balanceamento de carga usando várias abordagens ao invés da tradicional abordagem única. Essa proposta foi implementada no software de simulação fluido-dinâmica (CFD) MFSim, que faz uso de malha adaptativa bloco-estruturada, e validada usando 3 casos industriais e um caso acadêmico.

Palavras-chave: Balanceamento de Carga. Simulação Multi-física. Fluido-dinâmica computacional. Malha estruturada bloco-adaptativa.

Johnatas Teixeira de Freitas

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2025

Abstract

Multi-physics simulations are essential for the academy and industry, helping to understand how multiple physical phenomena behave in a given context. With the growth of the HPC market and the consequent greater accessibility to more advanced and powerful hardware, extensive use of parallelization became a widespread scenario in multi-physics simulations. This situation creates a problem of proper load distribution, which, in this case, is complicated by the already complex nature of multi-physics simulations. This work proposes multiple approaches to the load-balancing problem compared to the more traditional single-approach solutions. This proposition was implemented for the computational fluid dynamics (CFD) code MFSim, which uses a block-structured adaptive mesh and was validated using three industrial-scale cases and one academic case.

Keywords: Load-balancing. Multi-physics simulation. Computational Fluid Dynamics. Block-Structured Adaptive Mesh..

List of Figures

List of Tables

Acronyms list

1D One Dimension

2D Two Dimension

3D Tree Dimension

AI Artificial Intelligence

AMR Adaptive Mesh Refinement

AMG Algebraic Multigrid Method

API Application Programming Interface

CFD Computational Fluid Dynamics

CPU Central Processing Unit

DBMS Data Base Management System

FDM Finite Difference Method

FEM Finite Element Method

FVM Finite Volume Method

FSI Fluid-Structure Interaction

GMG Geometric Multigrid

GMRES Generalised Minimal Residual Method

GPU Graphics Processing Unit

HPC High Performance Computing

LBM Lattice-Boltzmann Method

LES Large Eddy Simulation

MG Multigrid Method

MFLAB Fluid Mechanics Laboratory of the Federal University of Uberlândia

RCB Recursive Coordinate Bisection

SFC Service Function Chaining

SAMR Structured Adaptive Mesh Refinement

SFC Space Filling Curves

ToT Time of Timestep

UDF User Defined Function

WeCA Weight Calculation Algorithm

Contents

Chapter 1

Introduction

Multi-physics simulations are an essential tool with both scientific and industrial applications (KO et al., 2010) (MERZARI et al., 2023) (LONGARES; GARCÍA-JIMÉNEZ; GARCÍA-POLANCO, 2023). However, direct-coupling multiple physics, each with its own governing physical laws and equations, and required space and time discretizations, many times different from each other, tend to create not only limitations on stability, accuracy, or robustness of the overall simulation (KEYES et al., 2013a), but also load-balancing problems (RETTINGER; RüDE, 2019) which adds to the already challenging task.

To mitigate the load-balancing problem, various space-time discretization methods incorporating load-balancing techniques have been developed over the years (HENDRICKSON; DEVINE, 2000) (DAS; HARVEY; BISWAS, 2001). One of those techniques, the adaptive mesh refinement, has been constantly developed and upgraded at least since 2001 (LAN; TAYLOR; BRYAN, 2001), for many types of applications (BAIGES et al., 2018) (SAKANE; AOKI; TAKAKI, 2022), discretization methods (JUDE; SITARAMAN; WISSINK, 2022) (YU; FAN, 2009), solvers (SAMPATH et al., 2008) (TEUNISSEN; KEPPENS, 2019) and hardware and software environments (SAKANE; AOKI; TAKAKI, 2022) (MATSUSHITA; AOKI, 2021).

Though many advances have been made, there is still room for improvement, especially for general-purpose multi-physics simulators with high-order, high-accuracy discretization methods that make use of block-structured adaptive mesh with dynamically sized blocks (FLúIDOS, 2022) and tackle industrial-scale problems, which tend to bring less than ideal settings, very large domains, and execution time.

1.1 Motivation

Simulations that handle many physical phenomena have been rather typical, both in the academy (NORDSLETTEN et al., 2011) and in the industry (BAYAT et al., 2021) in the past years. Part because of the constantly increasing computational power

(KEYES et al., 2013b), now going to the exascale (SIRCAR et al., 2023), at the same time the High Performance Computing (HPC) market grows (RESEARCH, 2022) allowing for more accessible hardware; Part because of the wide range of applications, coming from the more traditional ones as engineering and science (WILLIAMSON et al., 2012) (EWERT; KREUZINGER, 2021), passing through public health (DESAI; SAWANT; KEENE, 2021), food processing (SMAN, 2022) and even forensic investigation (GENTILI; PETRINI, 2016).

With such a wide field of applications, it is expected a plethora of codes (ONLINE, 2025) (ARC4CFD, 2025) (NASA, 2025) (MOCZ, 2025) (PROJECT, 2025), both specialized and general purposes, using different spatial and time discretization methods, with varying levels of accuracy, methods to couple the physics and various implementation of solvers. A load-balancing technique that works well for one code may not work for another. At the same time, it is also possible that, in the same code, a load-balancing technique that works well for one case may not work for another, thanks to differences in the physics involved and how they interact with each other.

Also, codes that work with industrial-scale problems have additional challenges (JANSSON et al., 2019), as these cases tend to bring less than ideal settings for coupling the physics, with many enabled physics, and very commonly, using retro-feeding mechanisms, under huge domains and, by extent, large requirements of both computational resources and execution time (LONG et al., 2021).

Adding to that, most of the load-balancing solutions are limited to proposing an algorithm that can be applied to all physics supported by the code, giving little or no freedom of customization at all to the user, which, depending on the case, can make a simulation not possible to balance, while it could be if the user could adjust some of the configurations of the load-balancing operation, according to the specifics of the studied case and used code.

This presents a situation where load balancing is necessary, but it is not easy to implement. Implementing it requires a careful understanding of the used code, its limitations, and the kind of simulations the code is built to run.

Our primary motivation for this work is to address this situation by proposing a multi-approach load-balancing solution that considers the inherent challenges of multi-physics simulations and the code used to run the simulations.

1.2 Objectives and Research Challenges

This work uses the general-purpose multi-physic simulator MFSim (more details in section 2.3), which already has load-balancing capabilities but fails to balance some industrial-scale simulations. The limitations stem from the simulation's complexity, as well as constraints on the MFSim mesh, solver, and physics coupling, and the load-

balancing technique's inability to identify where the load is being generated accurately and when to apply the load-balancing.

Our first objective is to implement the multi-approach load-balancing into MFSim's code to mitigate the limitations that prevent the proper functioning of the load-balancing. This is what section 3.2 describes.

The second objective is to validate the proposition by testing this modified version of MFSim with industrial-scale simulations that were previously incapable of applying load balancing, to verify whether the proposition can address this limitation. This is described in section 4.2.

The third objective is to evaluate the results of the second objective, both in terms of macro and micro evaluation, to see how the proposition behaves and its limits when applied to the test cases. This is described in sections 4.3 and 4.4.

The fourth objective is to compare our proposition with the state-of-the-art in load balancing of multi-physics simulations, which is what section 4.5 describes.

The final objective is to outline our future (and some not-so-future) plans to expand the multi-approach load-balancing, mitigating some of the identified limitations, and enhancing its capabilities for new Graphics Processing Unit (GPU)-enabled multi-physics simulations.

1.3 Hypothesis

The central hypothesis of this work is that balancing a multiphysics simulation is a complex task, highly context-dependent. Therefore, a multi-approach solution can produce satisfactory results, significantly impacting the time required to run a sufficiently balanced simulation.

1.4 Contributions

The main contribution of this work is the development and evaluation of the many-approach load-balancing, described in section 3.2, which provides a better tool for unbalance detection in the scope of multi-physics simulation (see section 3.2.2) and a series of approaches (hence the many-approach) for the load-balancing decision.

These approaches can both work out-of-box with a default, automatic algorithm (section 3.2.3) or can be tuned by the user (see sections 3.2.1, 3.2.3 and 3.2.4), allowing it a lot of freedom on when to use or not the load-balancing operation.

We also implemented another tool, not necessarily an approach to load balancing, but a fail-safe, inspired by the rollback functionality typically present in Data Base Management System (DBMS) systems, but tailored to the context of load-balancing multi-physics

simulations. This was necessary because of the limitations of the used code (see section 3.2.5).

The implementation was then validated using three industrial-scale cases and one academic case, which enabled the second contribution of this work: comparing the proposed many-approach load-balancing with other propositions.

The third contribution derives from the knowledge obtained during the development of this work, which is linked to our conclusion, which, in short (see section 5 for the proper conclusion), shows that there are more ways to redistribute the load of a multi-physics simulation than by only using the load-balancing operation, such as adding more computational resources or even using data from one simulations to speedup another. We investigated and published a paper on this topic, a spin-off-like contribution.

1.5 Outline

This dissertation is structured into five chapters. The current section is the introduction, which briefly presents the problem of load-balancing multi-physics simulations.

In Chapter 2, we delve into the base concepts necessary to understand the proposition, the framework used to run multi-physics simulations, and the state of the art of load-balancing multi-physics simulations.

Chapter 3 outlines the dissertation proposal and its challenges, offers insights into the implementation specifics, and presents the various components along with their respective functions and the methodologies applied, thus showing the achievement of our first objective.

Moving on to Chapter 4, we present the 4 test cases we used for experimental assessments, which underscore the achievement of the second, third, and fourth objectives. We also furnish an in-depth analysis of how this work compares to prior proposals.

Last, in Chapter 5, we discuss the conclusions drawn from the development and experimentation of the proposed solution, how they proved our hypothesis, the published papers based on the proposition or derived from it, and the future improvements of the proposition, fulfilling our last objective.

Chapter 2

Background and Related Work

This chapter will present some concepts related to Multi-physics simulations (Section 2.1). It also presents load balancing techniques (Section 2.2) and, finally, some related work (Section 2.4).

2.1 Multi-physics simulations

Simulations that handle more than one physical phenomenon, governed by their principles regarding their evolution throughout the simulation and in which conditions they achieve an equilibrium status, are considered **multi-physics simulations** (KEYES et al., 2013a). Their field of application is ample, varying from the traditional science and engineering fields (WILLIAMSON et al., 2012) (EWERT; KREUZINGER, 2021), passing through public health (DESAI; SAWANT; KEENE, 2021), food processing (SMAN, 2022), and even forensic investigation (GENTILI; PETRINI, 2016).

Solving a multi-physics simulation requires solving the partial differential (or integro-differential) equations describing the studied physical phenomena. In most cases, this can't be done analytically. So, to obtain a satisfactory resolution to the problem, numerical approximations must be used, hence the need for discretization of the equations in terms of time and space (FERZIGER; PERIĆ; STREET, 2019) (FORTUNA, 2000). Sections 2.2 and 2.1.2 will provide further insight into those themes.

Once the discretized equations are in place, a series of linear systems are mounted to solve the equations (KEYES et al., 2013a) (FERZIGER; PERIĆ; STREET, 2019) (FORTUNA, 2000). A widespread method to achieve that is to use multigrid algorithms that solve the discretized equations in a variable resolution grid to reduce errors (KEYES et al., 2013a). This is further explained in section 2.1.4.

And since we are considering multi-physics simulations, there's also the need to address how the physics are coupled, as they have different governing laws and possibly different forms of time-space discretization (ZHANG et al., 2021) (POZZETTI et al., 2019). Integrating this into a single simulation can be done directly (POZZETTI et al., 2019), with

a single code managing both physics and its particularities, or indirectly, with separate codes managing each physics and an overseer code managing the integration of the results (BESSERON; ADHAV; PETERS, 2024). Section 2.1.3 will explore these themes further.

Let's start with space discretization, which is a requirement for all multiphysics simulations. Then, we will proceed to time discretization, again a requirement for all multiphysics simulations, and then to more details regarding physics coupling and solver.

2.1.1 Spatial discretization and mesh

Most of the natural physical phenomena of interest for multiphysics simulations are described by partial differential or integro-differential equations. This kind of equation, usually considered a continuum of time and space, can calculate how physical phenomena behave in that region of space and time. This region is called *domain* and defines the part of space and time that is being studied in the simulation (FERZIGER; PERIĆ; STREET, 2019) (FORTUNA, 2000), and the phenomena studied are valid.

Since computers are discrete machines (RALSTON, 1986), they can't solve a continuum equation (or system of continuum equations). That creates a necessity to approximate the continuous *domain* with a discrete model, converting the infinite points of space-time contained within the *domain* into a representation with a finite number of points for space and time. Once that representation of the *domain* is in place, the continuum equation, or system of equations, is also converted from the original continuum equations (partial differential or integro-differential equations) to algebraic equations that can approximate the original ones, fit in the finite representation of the *domain* and be organized in linear systems that can be latter processed by the computer. This process is called discretization (FERZIGER; PERIĆ; STREET, 2019) (FORTUNA, 2000). In this section, we discuss only spatial discretization, with details regarding time discretization presented in section 2.1.2.

To discretize the *domain* in terms of space, we create a grid, a *mesh* (FORTUNA, 2000), composed of points (or nodes) interconnected by vertices. The space between adjacent nodes forms a cell later used by the discretized equations to obtain the physical properties in that specific region of the *domain*. The cells have a center and faces. When a face coincides with the *domain* boundaries, it is considered a boundary face. This classification is necessary to identify when boundary conditions of the *domain* should be applied to a cell. Also, some physical properties can only be obtained in the faces, like values for the velocity vectors, while others can only be obtained in the center, like pressure (FORTUNA, 2000). Figure 1 shows a graphical representation of nodes, centers, and faces of cells in Two Dimension (2D) and Tree Dimension (3D) meshes.

The *meshes* can be further classified regarding their inner structure. They can be structured, block-structured, or unstructured (FERZIGER; PERIĆ; STREET, 2019) (FORTUNA, 2000).

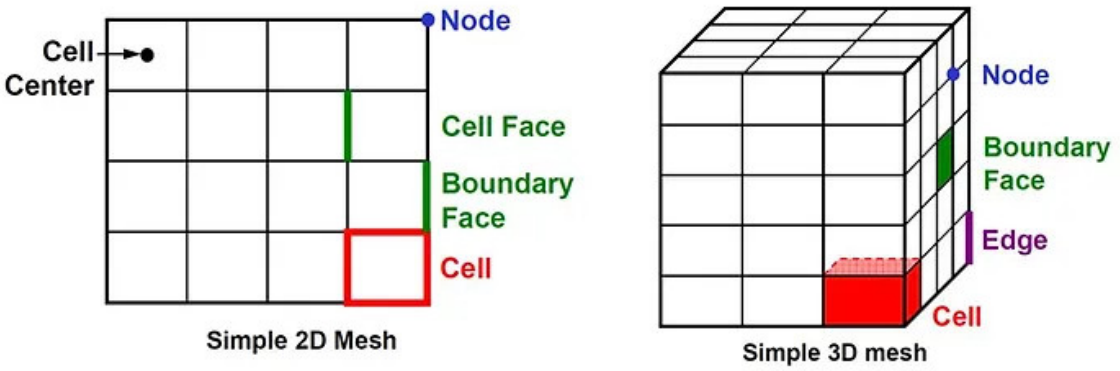


Figure 1 – Nodes and faces on 2D and 3D mesh (MANCHESTERCFD, 2024)

A **structured** or regular *mesh* is a type of grid in which each cell is numbered sequentially in reference to the axis of the grid, which allows the use of indices to access each cell. Also, all cells have a fixed number of neighbors, 4 in 2D grids and 6 in 3D grids, and the space between their nodes (deltas in each axis) can be fixed (uniform) or variable (non-uniform). They can also be Cartesian or curvilinear (non-orthogonal) (FERZIGER; PERIĆ; STREET, 2019) (MANCHESTERCFD, 2024). Figure 2 provides examples of these structured meshes.

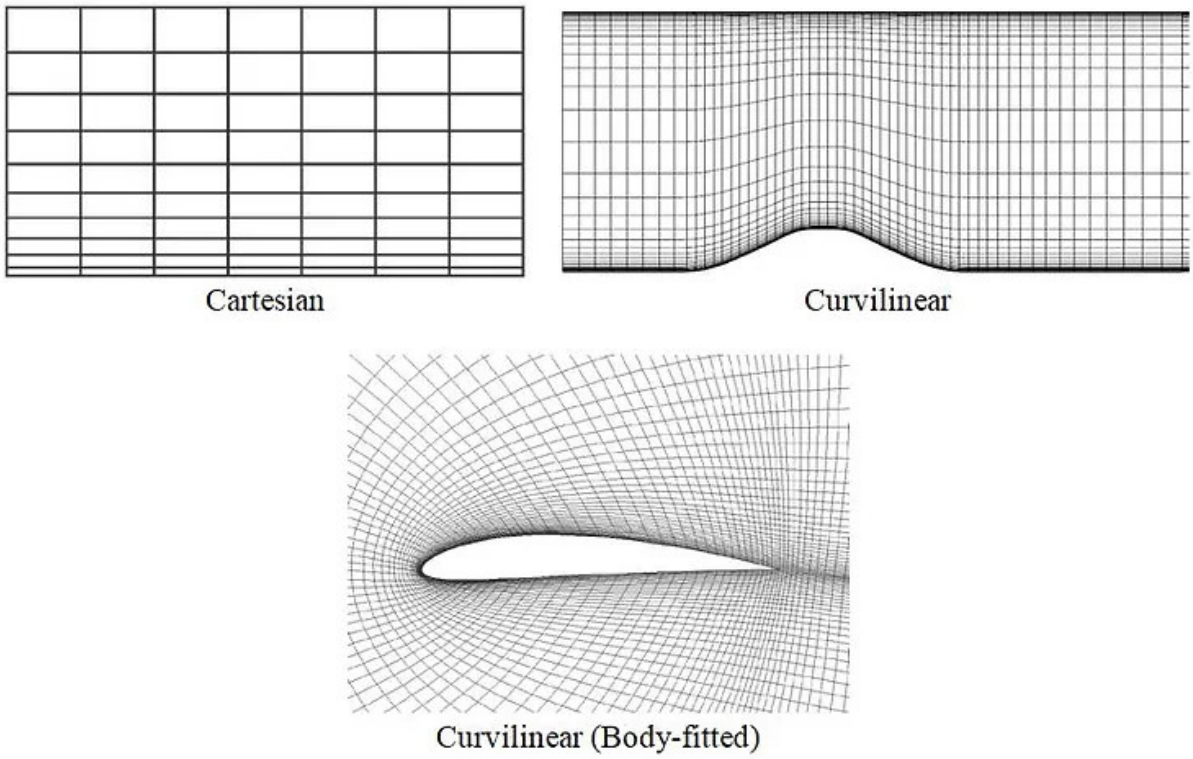


Figure 2 – Types of structured mesh (MANCHESTERCFD, 2024)

The main advantage of structured *meshes* is that the fixed neighborhood they allow simplifies the programming of the matrices later used by the algebraic equations and the linear systems built upon them. The main disadvantage is that they can only be used for

simple geometries (FERZIGER; PERIĆ; STREET, 2019).

The **block-structured** *mesh* expands the structured meshes. It provides easy access to the cells from the structured meshes, but organizes the entire mesh as a group of blocks rather than a single object. Those blocks, also called **patches** can have cells of uniform (matching) or variable (non-matching) sizes and in the case of the latter, this allows refinement regions on the mesh. They can be aligned to the axis (Cartesian) or be body fitting (curvilinear), and they can have overlapping blocks forming levels (composite or chimera meshes). Figure 3 shows graphical examples of these settings.

The main disadvantage of this type of mesh, especially for the non-matching or composite ones, is that conservation is not easily enforced at block boundaries, as it is necessary to interpolate the values of the studied physical properties between the less and more refined blocks. The main advantage is that more complex geometries can be more easily modeled, as the *domain* is decomposed into sub-regions better fitting the region geometry and, at the same time, allowing refinement areas for more precision of the physical phenomena studied in the simulation (FERZIGER; PERIĆ; STREET, 2019).

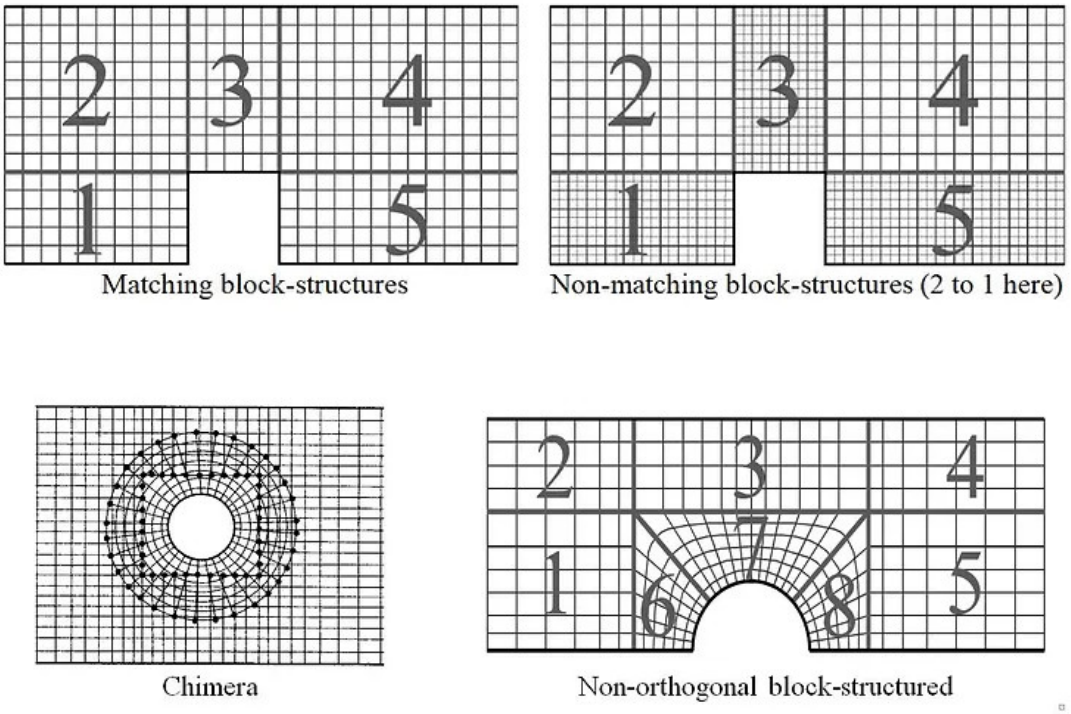


Figure 3 – Types of block-structured mesh (MANCHESTERCFD, 2024)

There is also a subtype of block-structured mesh, which adds adaptability, the Structured Adaptive Mesh Refinement (SAMR). This subtype allows the creation of refinement zones where the space width of the cells can be reduced to provide more accurate results for the discretized equations (BERGER; OLIGER, 1984). These zones can be superposed, creating finer grids that overlap the entire domain or are limited to a particular area, thus creating a locally refined area. The main advantage of this subtype is the possibility to

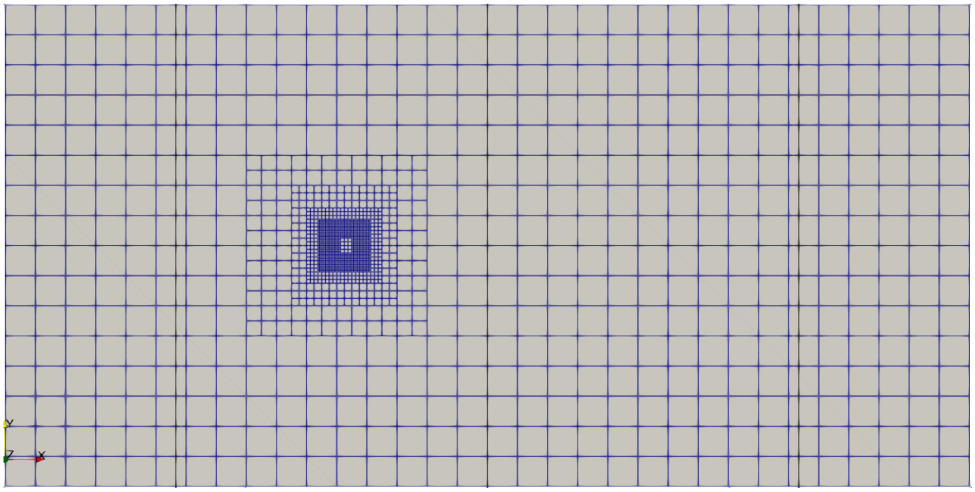


Figure 4 – Animated example of block-structured adaptive mesh - MFSim

convert the block-structured mesh refinement capabilities into a dynamic one, which, depending on the problem, is very useful (JUDE; SITARAMAN; WISSINK, 2022) (DUBEY et al., 2014). Figure 4 shows an animated example of a block-structured adaptive mesh.

Unstructured *mesh* is a type of grid where the cells can have any shape or number of neighbors, thus enabling any kind of geometry in the same mesh, without the need to create sub-partitions (like the block-structured mesh). Usually, these grids are made of triangles, quadrilaterals, polygons, or a mix of the tree (FERZIGER; PERIĆ; STREET, 2019) (MANCHESTERCFD, 2024). Figure 5 shows some examples of unstructured meshes.

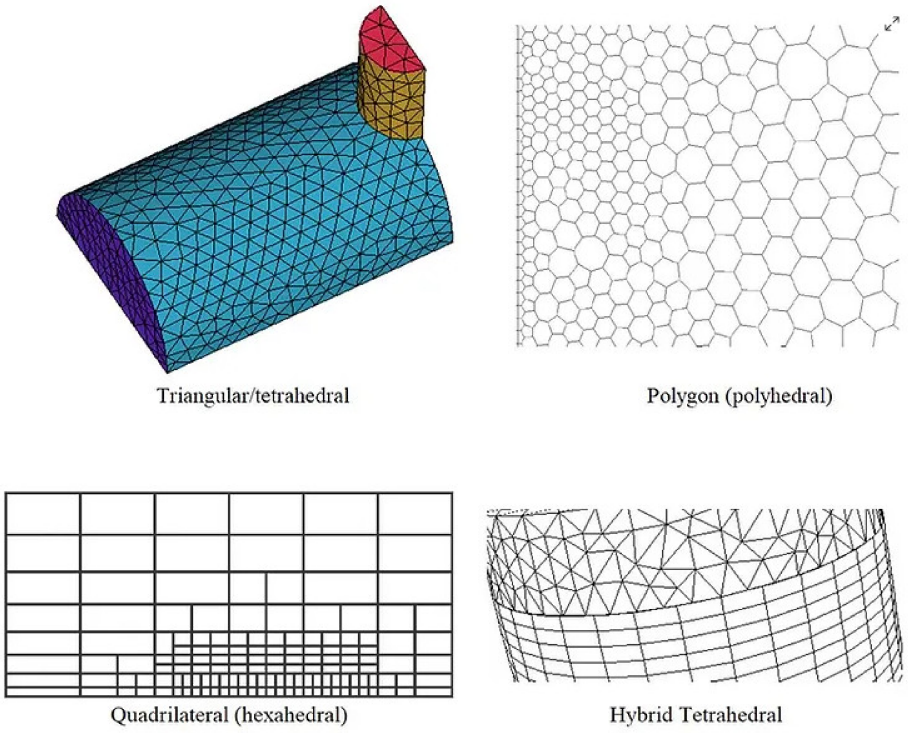


Figure 5 – Types of unstructured mesh (MANCHESTERCFD, 2024)

Though this type of mesh makes it easier to fit into more complex geometries, the irregularity of the data structure makes generating and plotting this type of mesh a more difficult task (MANCHESTERCFD, 2024). Also, the matrix of the algebraic equation system no longer has a regular, diagonal structure, which creates the need for reducing or reordering the number of points before trying to solve the systems, which takes time (FERZIGER; PERIĆ; STREET, 2019).

Lastly, meshes can also be classified according to their implementation. This can vary wildly, and combinations or hybrid forms are possible, but the main classes are the ones based on binary trees and the ones based on hashes.

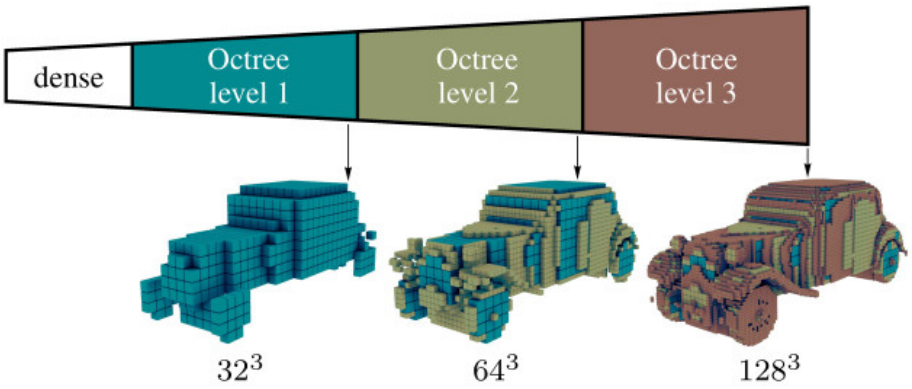


Figure 6 – Example of block-structured mesh implemented by an octree with single dimension blocks (TATARCHENKO; DOSOVITSKIY; BROX, 2017)

The most common meshes based on binary trees are the **quadtree**, **octree**, and **mtree**. A quadtree is a binary tree with four children for each node (hence its name), while an octree has eight children, with each child being used to map elements of the mesh-like nodes (of the mesh, not the tree), cells, or even blocks. A generalization of the quadtree/octree is the mtree that allows more children per node (SOUSA et al., 2019). Multiple trees can be used depending on the domain, implementation, or environment. Thanks to these characteristics, quad/octrees can be used with nearly all types of meshes, which makes this implementation one of the most common in multi-physics simulations (JUDE; SITARAMAN; WISSINK, 2022) (TATARCHENKO; DOSOVITSKIY; BROX, 2017) (NIEMöLLER et al., 2020) (TEUNISSEN; KEPPENS, 2019) (SAMPATH et al., 2008) (SOUSA et al., 2019) (CASTELO; AFONSO; BEZERRA, 2021) (SILVA et al., 2023).

When used with block-structured meshes, quad/oct/mtrees can have an interesting feature, which is generating the blocks (or patches) with the same dimension, which makes the load balancing task easier (JUDE; SITARAMAN; WISSINK, 2022) (see Figure 6 for an illustration of block-structured mesh generated by octree with single dimension blocks). Still, these trees are flexible enough to allow blocks with varying dimensions, which can

be more adequate for some problems (SILVA et al., 2023). Figure 7 shows an example of an mtree with blocks of different dimensions at the same level.

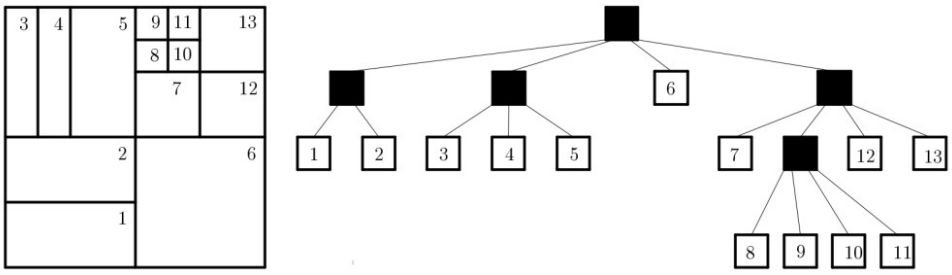


Figure 7 – Example of block-structured mesh implemented by an mtree with varying dimension blocks (SILVA et al., 2023)

The other mesh class regarding the implementation is the hash, which uses a hash to map the mesh elements: nodes, cells, or blocks/patches. In parallel environments, more than one hash may be used, with each parallel partition having its own hash containing the mesh (VILLAR et al., 2007). The most striking feature of this implementation, when combined with a block-structured adaptive mesh, is the generation of varying-sized blocks (or patches) for each refinement zone, called the patch-based Adaptive Mesh Refinement (AMR) approach (JUDE; SITARAMAN; WISSINK, 2022).

2.1.1.1 Discretization techniques

A discretization technique is built upon the mesh, influencing how the equations will be discretized. Three main techniques (or methods) are used by multi-physics simulations: finite difference, finite volume, and finite element (FERZIGER; PERIĆ; STREET, 2019).

The Finite Difference Method (FDM) was initially developed by Euler in the 18th century and consists of applying approximations to the partial derivatives of the equations for each point on the grid (or mesh). The result is one algebraic equation per grid node, in which the variable value at that node and a certain number of neighboring nodes appear as unknowns. Though this method is straightforward and effective, especially for structured meshes, enforcing conservation laws is challenging unless special care is taken (FERZIGER; PERIĆ; STREET, 2019).

The Finite Volume Method (FVM) uses the integral form of the equations and organizes the grid cells as **control volumes** in which the equations are applied. It can be used with any grid and, by design, easily enforces conservation laws. The main disadvantage of this method is the complexity of developing higher-order methods in 3D meshes (FERZIGER; PERIĆ; STREET, 2019).

The last of the three techniques is the Finite Element Method (FEM), which is similar to the Finite Volume Method, organizing the mesh into a set of discrete volumes or finite elements. These elements can be of any shape, making this method very useful for

unstructured meshes. The main difference with the FVM is that the equations are multiplied by a weight function before they are applied to the elements. Its main drawbacks are derived from the unstructured meshes commonly used with the method, which creates irregular matrices for the linear systems (FERZIGER; PERIĆ; STREET, 2019).

2.1.2 Time discretization

As is the case with space, which is a continuum that needs to be discretized for computers to solve the continuum equations that describe the physical phenomena studied in multi-physics simulations, time, another continuum, and part of the same equations, must also be discretized.

There are many methods and techniques used to achieve this (GOTTLIEB; KETCHESON, 2016), many of them directly related to spatial discretization techniques (FERZIGER; PERIĆ; STREET, 2019) (DONEA; QUARTAPELLE; SELMIN, 1987). But, generally, all of them follow the same principle: divide the time the simulation must run into smaller slices of time, the **timestep**, which evolves throughout the simulation until its completion.

In this principle, there are two primary forms for creating these time slices: by fixing the time width of each **timestep** or by varying the width throughout the simulation according to specific criteria (a dynamic **timestep**) (VILLAR et al., 2007). The use of each form depends on the problem studied by the simulation (FERZIGER; PERIĆ; STREET, 2019).

2.1.3 Physics coupling

To have a multi-physics simulation, we need to couple many different types of physics, each with its own governing laws and subsequent discretizations in terms of time and space (ZHANG et al., 2021) (POZZETTI et al., 2019). This coupling can be done **directly** or **indirectly**.

Directly coupling means a single code manages all discretizations from all equations. This can be done by using multiple space discretizations (grids) and/or multiple timesteps (POZZETTI et al., 2019) (VILLAR et al., 2007), each for each physics, with coupling techniques, like the fractional timestep (YUSTE; QUINTANA-MURILLO, 2012) (VILLAR et al., 2007), been used to communicate the result of a physics to another and vice-versa.

Indirectly coupling means specialized codes are used for each physics with an overseer code managing the specialized ones and the communication between them (BESSERON; ADHAV; PETERS, 2024) (TOTOUNFEROUSH, 2022). This is usually achieved by a coupling library like preCICE (CHOURDAKIS et al., 2022) integrated with the overseer code.

Another essential classification regarding physics coupling is how the physics interacts with each other. Some physics models only receive feedback from others, exhibiting a

passive behavior, while others interact with each other, employing a retroactive feedback mechanism. An example of a physics with passive behavior is acoustics, which receives the pressure field from the fluid dynamics to calculate sound propagation but does not influence the fluid dynamics itself (NIEMöLLER et al., 2020) (KALTENBACHER, 2018) (MOHAMED, 2016).

For an example of physics with retro-feed mechanism, fluid-structure interaction can calculate, for example, vibrations induced into a structure by the fluid dynamics (fluid -> structure interaction) and a following change in fluid direction caused by the structure vibration (structure -> fluid interaction) (DOWELL; HALL, 2001) (BELYTSCHKO, 1980). In the same way, combustion can be coupled with fluid dynamics to evaluate, with a good level of detail, how a chemical reaction occurs inside of a combustion chamber, with the combustion being influenced by the pressure and velocity vectors from the fluid (they can be the trigger to ignite the reaction) and, as the chemical reactions occur, the pressure and velocity end up been influenced by the changes in the chemical composition of the fluid, caused by the chemical reactions (MURRONE; SCHERRER, 2005) (YIN; ROSENDAHL; KÆR, 2011).

This can vary wildly from simulation to simulation. For example, the same simulation can have physics without the retro-feed mechanism and with the mechanism simultaneously.

2.1.4 Solver dependency

Once the discretized equations are in place, the space and time discretizations are defined, is time to solve the linear systems built upon the discretized equations. This is done by the solver, which, depending on the problem studied (and subsequent equations), will solve the linear systems directly or by an iterative (execute operations until convergence) method (FERZIGER; PERIĆ; STREET, 2019).

One of the many iterative methods is the Multigrid Method (MG). This method stems from the premise that as the convergence rate deteriorates as the meshes become more refined, the error frequencies must be smoothed by the grids with a more appropriate width (level of refinement) (VILLAR et al., 2007). Thus, the long wavelength part of the error is smoothed on coarser grids while the short wavelength part is reduced with a small number of iterations with a basic iterative method on the fine grid (WESSELING, 1995).

The discretized equations are then solved in each level, from the most refined to the coarsest, with the communication from the finest to the coarsest level being nominated *restriction*, which can be followed by a *relaxation*, and the communication from the coarsest to the finest level been the *prolongation/interpolation* (VILLAR et al., 2007). This ensures the same operations are executed at all levels, forming a cycle that can take the shape of a V or W. Figure 8 shows a graphical representation of these shapes for a 4-level mesh.

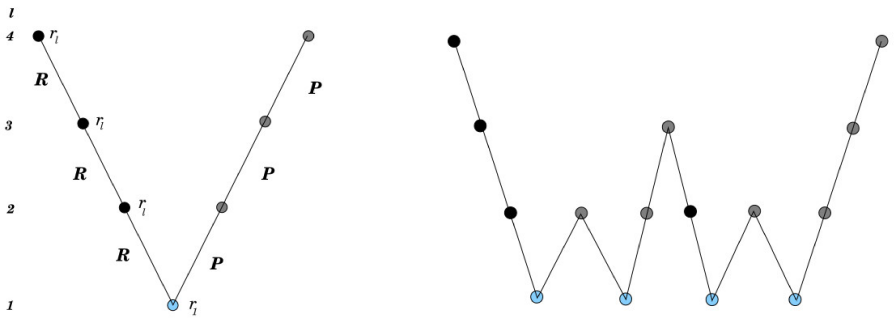


Figure 8 – V and W Multigrid cycles. Black circles: restriction with relaxation. Gray circles: prolongation. Blue circles: relaxation (VILLAR et al., 2007)

The cycles are repeated until the solution converges in the given **timestep** or a maximum number of iterations is met (VILLAR et al., 2007). Figure 9 illustrates the full v-cycle used to solve a discretization of the Poission equation for pressure in a 4-level mesh.

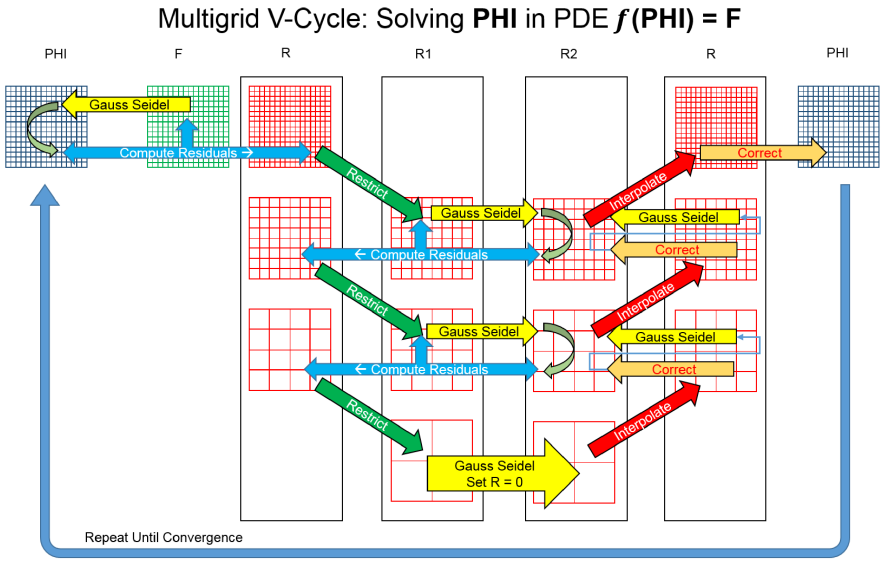


Figure 9 – Full multigrid v-cycle (WIKIPEDIA, 2024)

Generally, the multigrid method is implemented as GMG, building the coarse levels of the mesh from the fine level, geometrically increasing the level width of each level in a top-down manner to satisfy the v or w cycles in which the discretized equations are going to be solved (VILLAR et al., 2007) (WESSELING; OOSTERLEE, 2001).

There is, though, another type of multigrid, the Algebraic Multigrid Method (AMG). In the AMG, instead of developing the matrices of the linear system from the many levels in the mesh (a mesh-to-matrix approach), the idea is to develop the matrices from the problem equations directly and, therefore, develop the mesh from the matrices (a matrix-to-mesh approach) (WESSELING; OOSTERLEE, 2001) (RUGE; STüBEN, 1987) (STÜBEN et al., 2001). This makes the AMG attractive as a "black box" solver, as the number of levels and the operations between levels are derived from the problem's

equations and generated by the AMG algorithm directly without the need for intervention from the user. In addition, AMG can be used for many kinds of problems where the application of GMG is difficult or impossible (RUGE; STüBEN, 1987).

Another type of solver is those based on minimization algorithms like the Generalised Minimal Residual Method (GMRES). Instead of using an iterative method to solve the linear systems, like the multigrid solvers, this type of solver employs preconditioning to transform the system's equations into simpler equivalents, thereby facilitating their solution (hence its minimization). Usually, this solver requires more memory than the iterative ones, but can achieve faster convergence if the preconditioning is done correctly (JUDE et al., 2020).

2.1.5 Method accuracy and computational cost trade-off

Simulating multiple physics and their interactions over physical domains, which can be large depending on the problem, takes its toll on computational requirements and time (FERZIGER; PERIĆ; STREET, 2019). Hence, there are constant improvements in discretization methods, mesh generation, and solvers, some of which have already been discussed in the previous sections.

Another approach to mitigating the computational costs of multi-physics simulations considers the desired level of solution accuracy and the associated constraints on computational costs.

The more precise (or accurate) a simulation is, the greater the computational costs (MOHAMAD, 2011). The inverse is also true. Thus, considering the problem studied, it is necessary to decide if a more accurate but slower method is required or a less precise, but sufficiently accurate and faster method is a better choice (SUSS et al., 2023). The methods, algorithms, and techniques presented in sections 2.1.1.1, 2.1.2 and 2.1.4 are high accuracy but, depending on the problem, slower. This section presents an alternative, usually with lower accuracy, but a faster method.

This presentation is necessary because load balancing is directly affected by how accuracy will be approached and why some methods are naturally more scalable than others.

Continuing, one of the lower accuracy but faster methods available for multi-physics simulations is the Lattice-Boltzmann Method (LBM). In this method, instead of discretizing the domain into a grid composed of nodes, elements or control volumes, to represent parts (or slices) of the simulated space, with thousands of particles each, and in which the discretized equations will be applied by an iterative solver over several slices of time, we discretized the space into a grid of particles and then group these particles not in control volumes (or elements or nodes) but into distribution functions which are solved over slices of time (MOHAMAD, 2011) (ZHANG, 2011).

The main advantages of this method are its easy application to complex geometries (MOHAMAD, 2011), better implementation of physics in the particle level (ZHANG,

2011) and, since the distribution functions are explicit and locale, the method is relatively easy to parallelize (MOHAMAD, 2011), hence why it tends to be faster (SUSS et al., 2023). The main drawbacks come in the form of loss of accuracy (MARIÉ; RICOT; SAGAUT, 2009) (SUSS et al., 2023), which, depending on the physics (or techniques) involved in the simulation, can be a problem (SUSS et al., 2023) (ELHADIDI; KHALIFA, 2013).

2.2 Load balancing

As explained in the previous sections, multi-physics simulations can take their toll on computational requirements (FERZIGER; PERIĆ; STREET, 2019) (see also section 2.1.5). Hence the extensive use of parallel environments (PERMANN et al., 2020) (BERNASCHI et al., 2009) (DUBEY et al., 2009) (BARTUSCHAT; RÜDE, 2015), like the HPC (DADVAND et al., 2013) (SRIVASTAVA; DADHEECH; BENIWAL, 2011), enabled by the increasing computational capability (KEYES et al., 2013a), this creates a necessity to distribute the computational load throughout the many processing units, the **cores**, used during the simulation, to prevent that a core or group of cores do more useful tasks than others, creating a situation of **load imbalance** (NIEMöLLER et al., 2020).

To prevent this situation, a **load balancing** technique measures the efficiency loss due to non-useful tasks for each core and then redistributes the tasks between the cores (GARCIA-GASULLA et al., 2020). To implement this measure, the technique must first distinguish between useful and non-useful tasks. This is generally done by organizing the tasks into computations and communications.

Computations are tasks directly related to solving the discretized equations. At the same time, communications are operations related to synchronizations or data transfer between processes (or cores) and somewhat related to the computations (GARCIA-GASULLA et al., 2020) (NIEMöLLER et al., 2020). Though communications aren't directly related to solving the discretized equations, they are necessary to the overall simulation, so non-useful tasks (regarding solving equations) must not be considered as unnecessary tasks (NIEMöLLER et al., 2020). So, the premise for identifying a situation of load imbalance is when more time is spent with operations not related to the solution of the problem's equations than with operations directly related (HENDRICKSON; DEVINE, 2000).

Also, both computations and communications are problem-dependent. In other words, depending on the problem studied, the mesh used, discretization techniques, solver, physics coupled, methods, and even the hardware used, the time spent on computations and communications will vary (NIEMöLLER et al., 2020). This leads to load balancing techniques also being problem or context-dependent. Therefore, a technique well suited for a particular context may not be suited for another or at least require careful consideration before being used (GARCIA-GASULLA et al., 2020) (HENDRICKSON; DEVINE,

2000).

Then, after the tasks are grouped adequately by the technique, considering all relevant factors within the given context, a metric is developed to identify when a load imbalance is present. Usually, this involves attributing weights to the particles, nodes, control volumes, or other components with computations, the load-generating objects, and then counting these weights to know the workload of a core. If a core or group of cores has more weight than others, then a load imbalance situation is identified (SCHORNBAUM; RüDE, 2018) (NIEMöLLER et al., 2020) (RETTINGER; RüDE, 2019).

With the load imbalance identified, the technique uses a **load balancing algorithm** to redistribute the tasks (computations and communications) between the cores. Since the computations are tasks directly related to solving the discretized equations, and those discretizations are dependent on the discretization technique and the type of mesh used (see section), the load balancing algorithms are usually applied over the mesh (HENDRICKSON; DEVINE, 2000).

One of those algorithms is the RCB. Berger and Bokhari initially proposed this algorithm (BOKHARI, 1987), which redistributes the tasks by splitting the domain into two halves, each containing half of the tasks. Then, proceeds to apply the same approach to each half and continues to do so, recursively, until all generated subdomains, or partitions, have an equal or near equal number of tasks and all available cores are used (BOKHARI, 1987) (HENDRICKSON; DEVINE, 2000). Figure 10 illustrates a mesh balanced by the RCB.

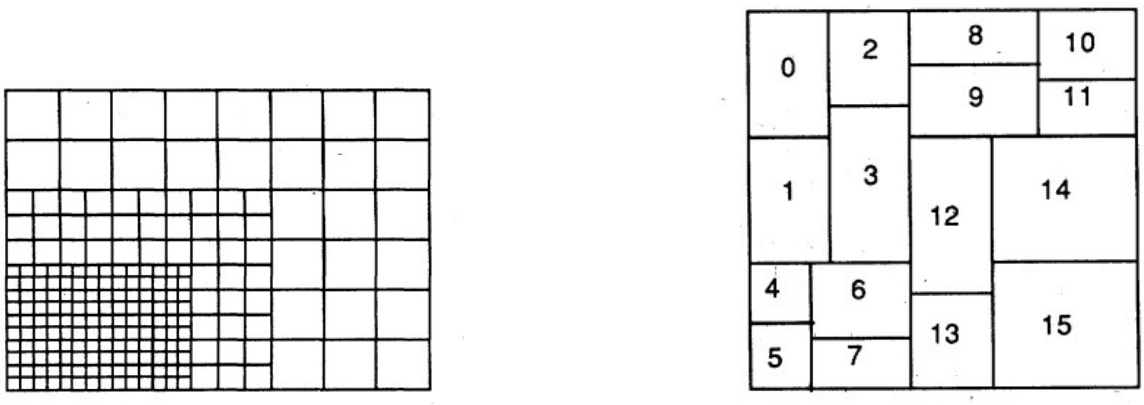


Figure 10 – A RCB redistribution (right) of a mesh (left) in 16 cores (BOKHARI, 1987)

The generated partitions have a rectilinear shape but are not necessarily the same size. This specific shape is beneficial for meshes that already utilize rectilinear subdomains, such as some implementations of block-structured adaptive meshes (LIMA et al., 2012). Being RCB a natural choice for a load-balancing algorithm for this kind of mesh, it is well-suited for this application.

Another load balancing algorithm type is based on the Hilbert curves, the SFC. The SFC has been used for scientific applications for some time (BADER, 2012) (BULUÇ et

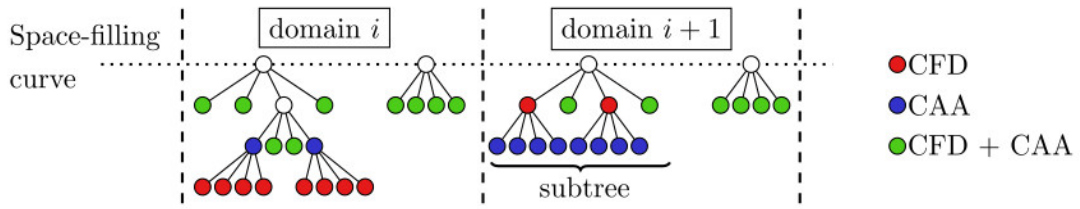
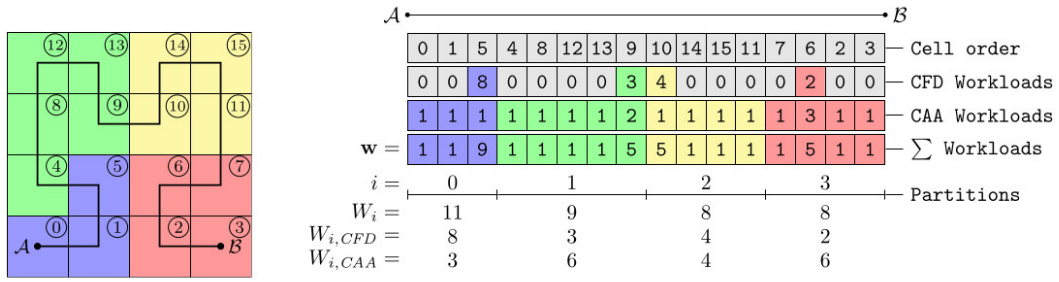


Figure 11 – Quadtree describing the workload of a multi-physics simulation (NIEMöLLER et al., 2020)

al., 2016), including mesh generations (LINTERMANN et al., 2014) (HENDRICKSON; DEVINE, 2000), and can also be used for load balancing (HENDRICKSON; DEVINE, 2000) (NIEMöLLER et al., 2020). In general, these algorithms, instead of using cutting planes to split the domain, create an octree (or quadtree in case of 2D meshes) to keep track of the load in the mesh (see Figure 11).

The traversal of the tree relates the load of the sub-trees (which represent regions of the mesh) with each cell of the traversal, providing the overall workload of the whole domain. This can be used to calculate the ideal load of each core as the traversal contains both the total load of the domain and the load for each core. At the same time, since each cell relates to a sub-tree which refers to regions of the mesh, the load can be traced from and back to the mesh as well (HENDRICKSON; DEVINE, 2000) (NIEMöLLER et al., 2020) (see Figure 12). This approach enables treating the load balancing of 2D or 3D meshes as a One Dimension (1D) problem, rather than a multidimensional one (MIGUET; PIERSON, 1997), or as a chain-on-chains problem (PıNAR; AYKANAT, 2004), thereby greatly simplifying the solution.



(a) 2D Hilbert SFC at refinement level 2.

(b) SFC linearization, CFD/CAA and accumulated partition-cell workloads, total workloads W_i of the resulting partitions (solution of the CCP problem, average workload $\overline{W} = 9$) and CFD/CAA workload distribution $W_{i,CFD/CAA}$ among all subdomains.

Figure 12 – Tree traversal summarizing the workload in the mesh (NIEMöLLER et al., 2020)

When a load imbalance situation is found, the 1D representation of the workload is then used to identify which cells of the array should be moved between cores to redistribute the load between the cores in such a manner that no core or group of cores does more computational work than others. Since those cells can be mapped to mesh regions,

moving these cells between cores also implies moving areas of the mesh between cores. In other words, the load balancing is done by balancing the workload array (1D object) and propagating this balancing through to the mesh (2D or 3D object).

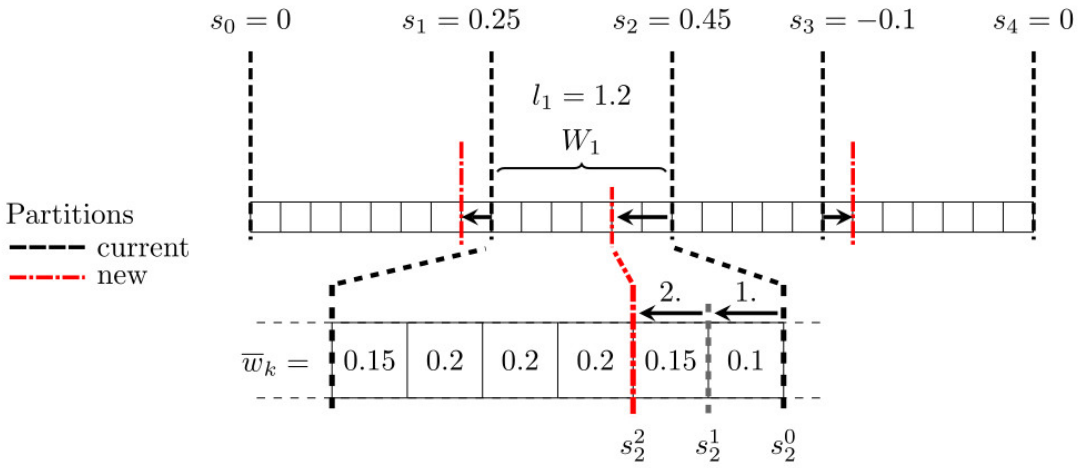


Figure 13 – Balancing the workload array in a SFC (NIEMöLLER et al., 2020)

Figure 13 illustrates this process, where the workload array Wk, divided into four partitions, has the first and second partitions doing more work than the others, with the second partition being more overloaded. Balancing the array Wk involves moving one cell from the first partition to the second, balancing the first partition, and then moving two cells from the second partition to the third (which was initially underloaded), thereby balancing the second partition. The third, which became more loaded with the newly received cells, also moved one cell to the last partition, thus balancing the entire array. This balancing is later propagated to the mesh, balancing the whole domain.

There is another type of load balancing algorithm that differs from both RCB and SFC in that it balances the domain locally rather than globally. In these algorithms, also called **diffusion** algorithms, when a load imbalance situation is found in one core, they move some of the load from that core to one of its neighboring cores. Then, they repeat that operation until the overall load imbalance is reduced to an acceptable level (HENDRICKSON; DEVINE, 2000) (SCHORNBAUM; RüDE, 2018).

Since the balancing operations are done locally or involve only the heavily loaded core and some of its neighbors, these algorithms are essentially asynchronous as they don't require knowing the load of every core in the simulation to do their work. This makes these algorithms very effective in dealing with minor changes in the load balance of the simulation, but also more challenging to implement as multiple cores can be more loaded and multiple load balancing operations can be engaged simultaneously (HENDRICKSON; DEVINE, 2000).

2.3 MFSim

Code capable of doing this is required to run a multi-physics simulation. MFSim is one of these codes. An MPI (UNIVERSITY, 2024) parallel Computational Fluid Dynamics (CFD) software written mostly in FORTRAN with some modules in C/C++, developed by the Fluid Mechanics Laboratory of the Federal University of Uberlândia (MFLAB) in Brazil. This computational platform's development began with the work of (VILLAR et al., 2007) and has been continuously developed over the years into a multidisciplinary and multiphysics code. Nowadays, this platform application allows simulating 3D problems involving: turbulent flow (VEDOVOTO; SERFATY; NETO, 2015) (DAMASCENO; VEDOVOTO; SILVEIRA-NETO, 2015), Fluid-Structure Interaction (FSI) (NETO et al., 2019) (SOUZA et al., 2022) (MORALES et al., 2023) (STIVAL et al., 2022), multi-phase flows (PIVELLO et al., 2014) (BARBI et al., 2018) (PINHEIRO et al., 2019) (PINHEIRO et al., 2021), gas-solid and gas-liquid flows (SANTOS, 2019), chemically-reactive flows (DAMASCENO; SANTOS; VEDOVOTO, 2018) (CASTRO et al., 2021) and counts even with Large Eddy Simulation (LES) approaches considering isotropic and anisotropic modelings. Recently, the MFSim code was used to assess hypersaline solutions disposal operations, as local environmental regulations are crucial for minimizing the impact on marine ecosystems (MOTA; VEDOVOTTO; ARISTEU, 2023).

Regarding the mesh, MFSim uses a multi-level, block-structured, with dynamically-sized blocks, adaptive mesh for the Eulerian space (fluid dynamics, turbulence, and other physics), and a mesh-less approach for the Lagrangian space (immersed boundary and particle-based physics) (VILLAR et al., 2007). It also employs two discretization techniques: the Finite Volume Method (main) and the Finite Element Method (primarily for fluid-structure interaction).

Regarding the solver, MFSim uses the GMG as the main solver and the solver toolkit provided by the library PETSc (BALAY et al., 1998) as a secondary solver. MFSim also features load balancing capabilities, utilizing the RCB algorithm implemented by the Zoltan library, part of the Trilinos Project (TRILINOS, 2020).

2.4 Related Work

The work of Hendrickson (HENDRICKSON; DEVINE, 2000) in 2000 was the first to relate load balancing to spatial discretization (mesh and discretization technique). Following, Lan (LAN; TAYLOR; BRYAN, 2001) proposed an update to the block-structured adaptive mesh algorithm initially developed by (BERGER; OLIGER, 1984), using a domain decomposition technique based upon graph partitioning.

Still, a few years later, Devine's work (DEVINE et al., 2005) shows that even with the advances in load balancing of multi-physics simulations already made, the problem was

far from being solved, as more and more physics were being coupled. Additionally, with the widespread use of heterogeneous hardware, the load balancing problem has resurfaced, necessitating new approaches.

In the same direction, Dubey (DUBEY et al., 2014) compared a series of adaptive mesh frameworks, evaluating them for their load-balancing capabilities, heterogeneous hardware support, coupled physics, and other criteria. They concluded that load balancing still needed improvements.

Following this conclusion, the work of Schornbaum (SCHORNBAUM; RüDE, 2018) proposed another upgrade for the block-structured adaptive mesh algorithm by implementing a forest of octrees to separate the load-balancing data structures from the mesh, with only the mesh topology and other metadata regarding each partition been directly handled by the load balancing algorithm who uses a diffuse approach (see section 2.2 for more details regarding diffuse load balancing algorithms). The authors implemented their proposal in the framework waLBerla, a multi-purpose code that models fluid, turbulence, particle, and rigid body physics. They balance the block-structured adaptive mesh over 450 thousand cores while using the Lattice-Boltzmann method (for details regarding this method, see section 2.1.5). This work was later expanded by Bauer (BAUER et al., 2021), who integrated GPU support into the load-balancing algorithm, still maintaining the Lattice-Boltzmann method.

Another work using the waLBerla framework was conducted by Rettinger (RETTINGER; RüDE, 2019), who employed the Lattice-Boltzmann method, a block-structured mesh without adaptability, and SFC as a load balancing algorithm. The authors managed to reduce the overall simulation time by 14% thanks to the better load balancing produced by their implementation.

Regarding more accurate but slower methods, the work of Niemöller (NIEMöLLER et al., 2020) developed a load balancing technique for directly coupled multi-physics simulations composed of fluid dynamics and acoustics, using the finite volume method (see section 2.1.1.1) as a discretization technique, applied over adaptive unstructured meshes. The authors developed an upgraded version of SFC (see section 2.2) with incremental balancing until a satisfactory level of load imbalance was achieved, a strategy usually found in diffuse load balancing algorithms. With this implementation, they improved load balance up to 20% in a simulation with 6144 cores on homogeneous hardware and achieved an evenly distributed load in a simulation on heterogeneous hardware.

Other work utilizing a high-accuracy method is that of Jude (JUDE; SITARAMAN; WISSINK, 2022), which employed an orchard of octrees to implement a block-structured mesh. The finite difference method was used as a discretization technique to solve rotorcraft simulations with immersed boundary and fluid dynamics. The load balancing algorithm is a version of SFC applied to the orchard of octrees, resulting in increased efficiency, especially in heterogeneous systems with mixed CPU and GPU.

2.4.1 State of the art summary

Table 1 highlights a comparative analysis of the diverse solutions identified within the state-of-the-art literature, visually summarizing the works as mentioned earlier by their most important features for load balancing of multi-physics simulations, which are: mesh type, mesh structure, presence of adaptability, discretization technique and solver, load balancing algorithm, presence of retro-feeding physics, physics simulated, level of specialization of the code and hardware support concerning processing units (Central Processing Unit (CPU) and GPU).

Table 1 – State of the art summary.

	Mesh Type	Mesh Structure	Adaptability	Discretization Technique & Solver	LB Algorithm	Retro-feeding physics	Physics	Specialization	Hardware Support
(SCHORNBAUM; RüDE, 2018)	Block-structured	octree	✓	LBM	Diffusion	○	Fluid Turbulence Particle Rigid Body	Multi-purposes	CPU
(BAUER et al., 2021)	Block-structured	octree	✓	LBM	Diffusion	○	Fluid Turbulence Particle Rigid Body	Multi-purposes	CPU-GPU
(RETTINGER; RüDE, 2019)	Block-structured	octree	○	LBM	SFC	○	Fluid Turbulence Particle Rigid Body	Multi-purposes	CPU-GPU
(NIEMöLLER et al., 2020)	Unstructured	octree	✓	FVM-AMG	SFC	○	Fluid Turbulence Acoustics	Computational Acoustics	CPU
(JUDE; SITARAMAN; WISSINK, 2022)	Block-structured	octree	✓	FDM-GMRES	SFC	✓	Fluid Turbulence Immersed Boundary [1]	Rotorcraft [2]	CPU-GPU

1. Immersed Boundary is a physical model to describe a body inserted into a fluid. This body can be rigid or fully deformable (VERZICCO, 2023).

2. Rotorcraft CFD simulates physics inside the scope of rotor movement used by helicopters, drones, and other rotor-based machinery (JUDE; SITARAMAN; WISSINK, 2022).

Chapter 3

Many-approach Loadbalancing

This section is divided into two main subsections. The first describes the challenges of load balancing in MFSim in its original state, and the second describes how this work addresses these challenges.

3.1 Problem delimitation

In this section, we discuss the challenges addressed by the proposition described in the section 3.2.

3.1.1 Mesh and Solver relation

In this work, we use the SAMR (see section 2.2 for more details) developed by Millena (VILLAR et al., 2007) as an upgrade to the original SAMR developed by Marsha Berger (BERGER; OLIGER, 1984). This implementation differs from the original by modifying how timesteps are utilized in the refinement levels. Specifically, Berger's implementation uses one timestep for each refinement level, whereas Millena's implementation employs a single timestep.

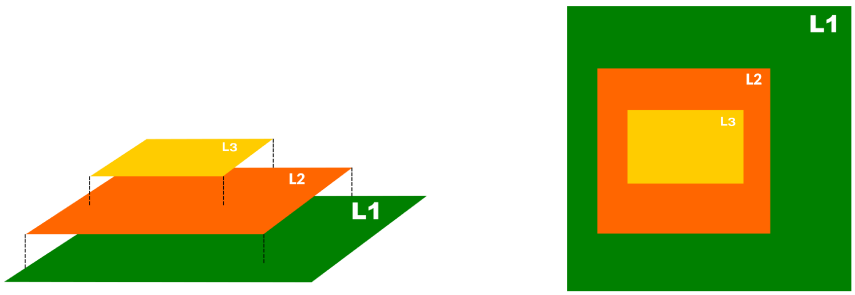


Figure 14 – Example of refinement levels. L1 is the coarsest level, superposed by L2 and L3, both finer grid levels (FREITAS et al., 2023)

The mesh is also structured in a hierarchy of levels, with the lower level being the coarsest, superposed by other levels corresponding to the ever finer grid, the refinement

levels. Figure 14 illustrates this level hierarchy.

The coarsest level encompasses the entire domain and is present in every core utilized by the simulation. The refinement levels, on the other hand, are created and destroyed dynamically as the simulation advances both in time and space and may or may not be present in all cores (VILLAR et al., 2007). These levels are created following two rules to reduce error propagation from the finer levels to the coarser ones: **first**, the border of a finer level must be contained inside the cells from the level immediately below; **Second**, the finer level must be contained inside the level bellow, except when touching domain or process boundaries (VILLAR et al., 2007).

Figure 15 shows two meshes in which these rules aren't observed. The mesh on the left has one coarse level (white filled) and one finer level (blue-filled). The finer level is not contained in the coarse level, which creates a problem for the interpolations used in the fine-coarse level communications used by the solver, hence the rule. The mesh shown on the right has three levels: white being the coarse, blue being the first refinement level, and cyan being the second refinement level. Though this time, each refinement level is entirely contained in the level directly below, the cyan level is bordering the white level directly, which again, creates a problem for the interpolations used by the solver during fine-coarse level operations, as these interpolations consider the level directly below (or above) e not levels times below (or above). That is why this mesh violates the second rule.

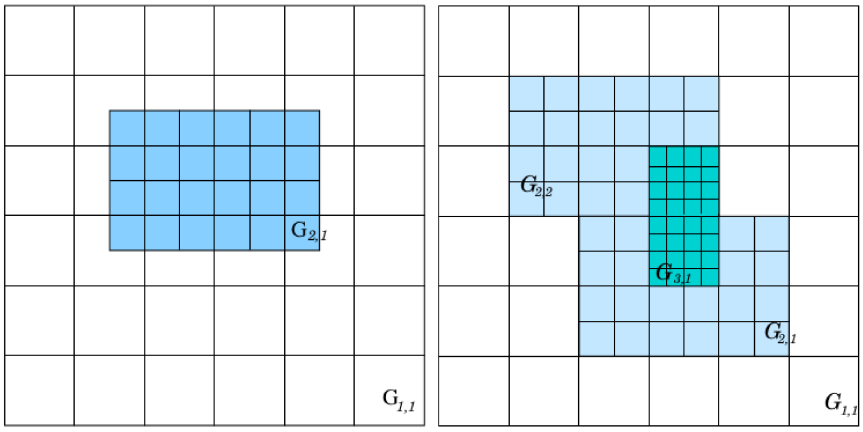


Figure 15 – Left: border of the finer level is not contained in the cells of the level below. Right: The finest level is not inside the level directly below (VILLAR et al., 2007)

In contrast, Figure 16 shows a mesh where the rules are observed. Again, there are three levels, but this time, all refinement levels are properly contained in the level directly below, and no refinement level borders another level not directly below or above.

This structure, though necessary, creates a situation where it is not always possible to divide a given sub-region of the mesh, which directly interferes with the load-balancing capabilities, as it requires redistributing portions of the mesh throughout the cores used

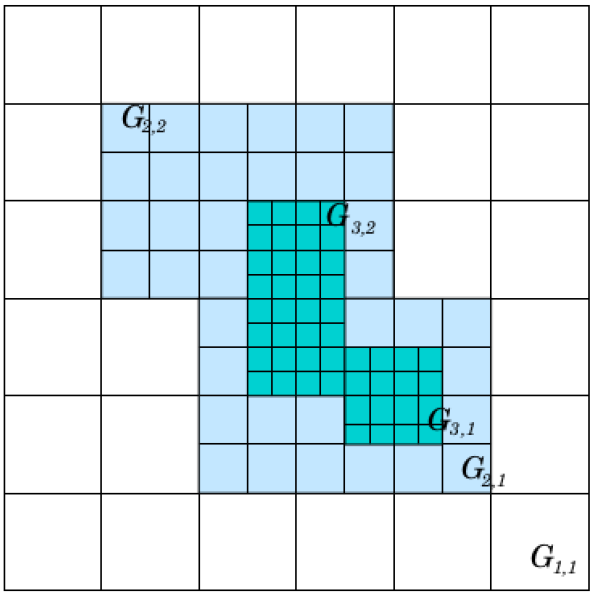


Figure 16 – Properly created nested levels considering the two rules (VILLAR et al., 2007)

in the simulation. This situation is called **badpoint** and describes a portion of the mesh that cannot be divided without violating one or both of the rules discussed previously.

Continuing, we also use a solver based on the GMG, which is intrinsically integrated into the discussed mesh structure. Generally, a GMG solver requires multiple coarse levels to reduce the error of the solution (more details regarding this solver are described in section 2.1.4). Thus, beyond having a base coarse level and as many as desired refinement levels, the mesh used in this work also has additional coarse levels, created below the main coarse level, to satisfy the solver (VILLAR et al., 2007) (LIMA et al., 2012). These levels created below the primary coarse levels are called **virtual levels** and are defined by the solver, while the main coarse level, called **lbot**, is defined by the user. The refinement levels can be determined by the user or generated by MFSim's refinement algorithm, considering user-defined constraints, such as the number of levels to be created and the conditions under which they are generated. Both the main coarse level and the refinement levels are called **physical levels**, with the first physical level (main coarse level) being called **lbot** and the finest level being called **ltop**.

Figure 17 shows an illustration of this relation between mesh and solver in this work, presenting virtual levels, main coarse level, refinement levels, and how each one relates to the GMG's v-cycle.

Both the virtual levels and the first physical level (lbot) always cover the whole domain, while refinement level coverage depends more on user configuration, which is case-dependent. Usually, refinement levels cover only a few parts of the domain.

All levels are filled with control volumes, which organize the level's grid cells and are later used by the GMG to solve the discretized equations in the point represented by

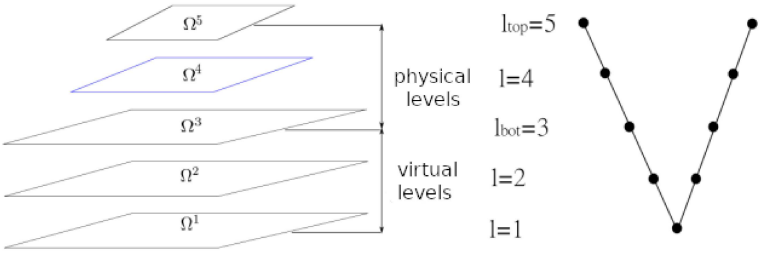


Figure 17 – Example of MFSim's mesh: levels l1 and l2 are virtual levels created for the solver. Level l3 is the main coarse level, the lbot. Levels l4 and l5 are refinement levels, with l5 been the ltop (LIMA et al., 2012)

the control volume (more details regarding control volumes are in section 2.1.1.1). The virtual levels contain fewer control volumes than any physical level, reducing the number as the level distances itself from the first physical level. On the other hand, refinement levels increase the number of control volumes as they move away from the lbot, which usually makes simulations with many refinement levels more loaded than the ones with fewer refinement levels.

Each level also groups control volumes in **patches**. These patches form the blocks, hence the block-structured mesh, which, along with the levels, allows for mesh refinement and the representation of more complex geometries. Figure 18 shows a graphic example of the relation between the levels, patches, and control volumes.

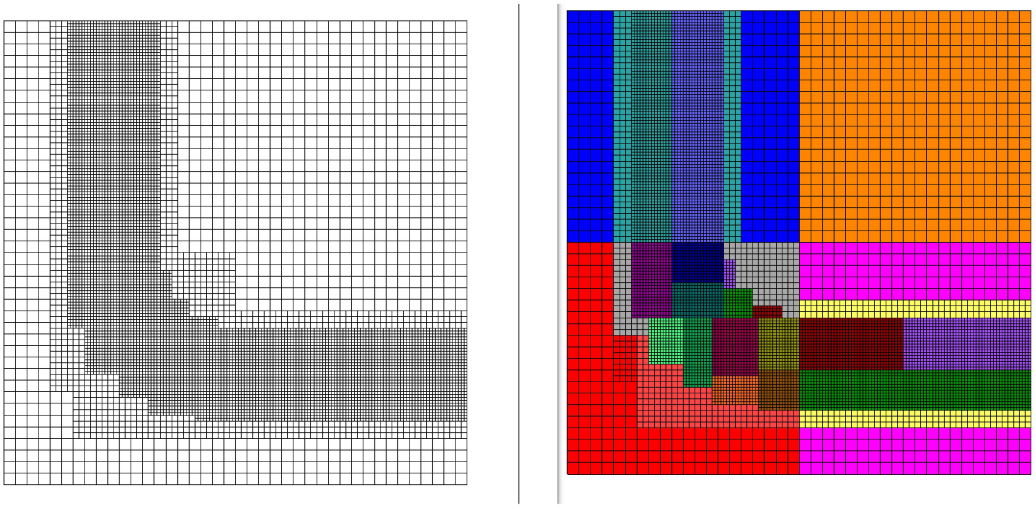


Figure 18 – Left: the mesh showing the levels and control volumes. Right: same mesh, but showing the patches (colored blocks) in which the control volumes are grouped in each level

Another critical aspect of the mesh is the **ghost cells**. These cells are virtual control volumes on the fringes of the patches and serve as a fine-coarse interface for the GMG v-cycle relaxation and prolongation operations (see section 2.1.4 for more details) and also for communication between cores (LIMA et al., 2012)

Figure 19 shows an example of a GMG v-cycle prolongation operation on the control

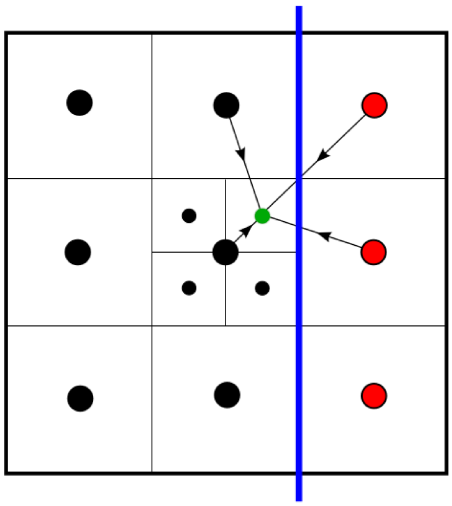


Figure 19 – Example of GMG v-cycle prolongation in a control volume on the fringe of a core (LIMA et al., 2012)

volume with the green point from the control volumes with black and red points of the level below (noted by the size of the point - larger the point, lower the level). The black points are in the same core as the green point and are directly accessed, while the red points are in another core and are accessed by ghost cells, which transfer the equivalent data from the other core to the current one (the core with the green point).

3.1.2 Influence over Loadbalancing

As explained in section 2.2, loadbalancing a multi-physics simulation requires redistributing parts of the mesh between the used cores. This is also the case in this work, with the addition that the structure presented in the previous section (3.1.1) must be considered.

Algoritmo 1 WeCA

```
initialize weight matrix with lbot
for every level from ltop to lbot+1 do
  for every patch in level do
    for every control_volume in patch do
      map control_volume to equivalent_cv in lbot
      increase weight of equivalent_cv by 1
    end for
  end for
end for
```

In this regard, the load balancing operation begins with measuring the load. Since we have a multilevel mesh, where computation is found at all levels, we must first account for all control volumes throughout the mesh.

As the physical levels are the ones with the majority of the control volumes of the mesh, only these levels are considered in this task (LIMA et al., 2012). Additionally, since the only physical level that can assuredly cover the entire domain is the lbot, the weight matrix representing the workload of the mesh is created on this level, mapping the control volumes (in lbot) to their associated weights.

To relate the control volumes in the levels above the lbot to the ones in the lbot, a loop is used from the finest level (ltop) to the level directly above the first physical level (lbot). Then, the inner loop is used to access all patches (blocks of control volumes) in the level. Then, another inner loop is used to access all control volumes in the patch and, by their coordinates, relate the control volume in the current refined level to the control volume, which is superposed in the lbot. This relation is possible because we use a block-structured mesh, a type of mesh in which each control volume has a sequential coordinate in relation to each axis of the grid, and finding the coordinates of a control volume below the current one is a matter of projecting the current control volume coordinates in the desired level by solving an arithmetic operation (more details regarding block-structured meshes can be found in section 2.2).

With the relation between the refined control volumes and their pairs in the lbot established, we can start assigning the weights. To do this, all control volumes receive a default weight of 1, with the weights from the refined levels being projected over the control volumes in the lbot they cover. Thus, the weight of a covered control volume in lbot is the sum of its weight (which is 1) and the weights of all control volumes that superpose it. This causes the covered control volumes (in lbot) to greatly increase their weights, allowing the later load balancing algorithm to identify areas with more control volumes and, therefore, more load in the mesh. This execution flow is summarized by the algorithm 1, and a visual example of a weight matrix generated by this algorithm in a 3-level refinement mesh is provided by figure 20.

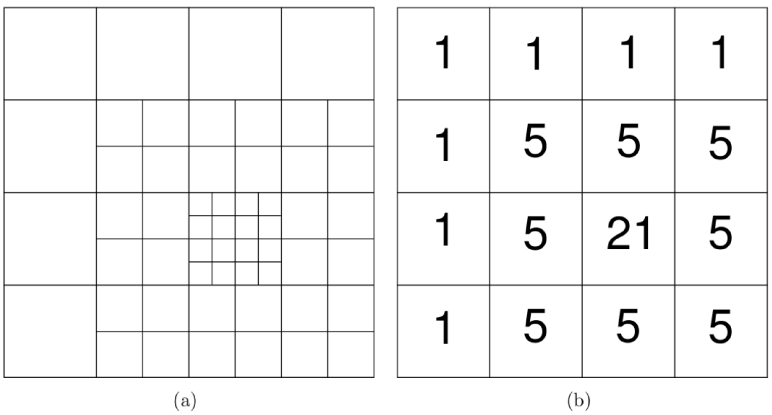


Figure 20 – WeCA (b) generated from a 3-level refined mesh (a) (LIMA et al., 2012)

With the weight matrix generated, the RCB (more details to this algorithm are in section 2.2) implementation on the library Zoltan (BOMAN et al., 2012) (ZOLTAN, 2025) is

than used to redistribute the weights throughout the mesh, generating a new weight matrix which is than used to rebuild the mesh. Since the original weight matrix relates lbot's control volumes to workload, the first step to rebuild the mesh is to relate those control volumes to the new load distribution. And here, things get more difficult because the mesh has its own rules and is intrinsically associated with the GMG's v-cycles.

That means the load-balancing operation **must** create a new, redistributed mesh, which at the same time, doesn't produce any improperly nested levels, violating one or both of the two rules, and doesn't prevent the GMG's v-cycles. This can be daunting to achieve depending on the mesh geometry and/or the physical phenomena studied in the simulation, as some geometries may present limitations on how many parts they can be subdivided with, thus imposing a limit on how many sub-parts the load balancing can safely create; And, some physics may require extra mesh elements, which may not be adequately computed by the load balancing, or even uses a somewhat inflexible stencil for solving the discretized equations, which again, creates another limit for how many sub-parts can be made upon a given area of the mesh (DUBEY, 2014).

Since the load balancing algorithm only knows the weight matrix, which hides the mesh internal structure, the RCB may try to redistribute a heavy workload area, densely packed with refined levels. This redistribution may violate one or both of the rules, thus creating a **badpoint** (more details of badpoints are defined in section 3.1.1). Since badpoints describe mesh partitions that degrade fine-coarse interface communications, introducing additional errors to the solver and potentially preventing solver convergence, this situation is a critical issue. When identified by MFSim (the code used in this work), it will result in an immediate stop to the simulation.

This poses a heavy constraint on the load balancing algorithm, as even if the RCB manages to successfully redistribute the workload throughout the simulation used cores, if a single badpoint is found in one of those cores, the simulation will crash.

Disabling or ignoring the mesh rules is not an option either, as they are directly related to a serious problem that can crash the solver if not observed. Therefore, the most common approach to this problem is to modify the badpoint recognition algorithm, which is time-consuming and heavily dependent on the simulation. Also, a tuning in this algorithm that works for one simulation may not work for another.

This is one of the main challenges the proposal described in section 3.2 faces.

Continuing, another problem that may happen in load balancing is related to what we call **zeroed cores**.

Since Zoltan's RCB receives only the weight matrix generated from the mesh, but without knowing any details regarding the mesh, the balanced weight matrix may result in some cores having not weight at all. This will generate a mesh with cores without any level, patch, or control volumes. This usually happens when, by the load-balancing algorithm standards, there are more cores than the actual weight in the matrix to be

balanced. This behavior is not inherently incorrect, since load balancing algorithms are designed to efficiently distribute load among the used cores (GARCIA-GASULLA et al., 2020); if there is more capacity than load, an efficient redistribution of load may imply that some cores have no load at all. The cores with no load are the **zeroed cores**.

Also, the behavior is unrelated to MFSim's integration with Zoltan's Application Programming Interface (API) and even with the RCB algorithm. Figure 21 shows the results of two load-balancing tests run directly in Zoltan, with both RCB and SFC algorithms (more details for both algorithms are in section 2.2), using the same weight matrix and having the same amount of available cores as a start condition. Both algorithms produced a zeroed core, indicating that the behavior does not depend on the load-balancing algorithm or the MFSim use of Zoltan API.

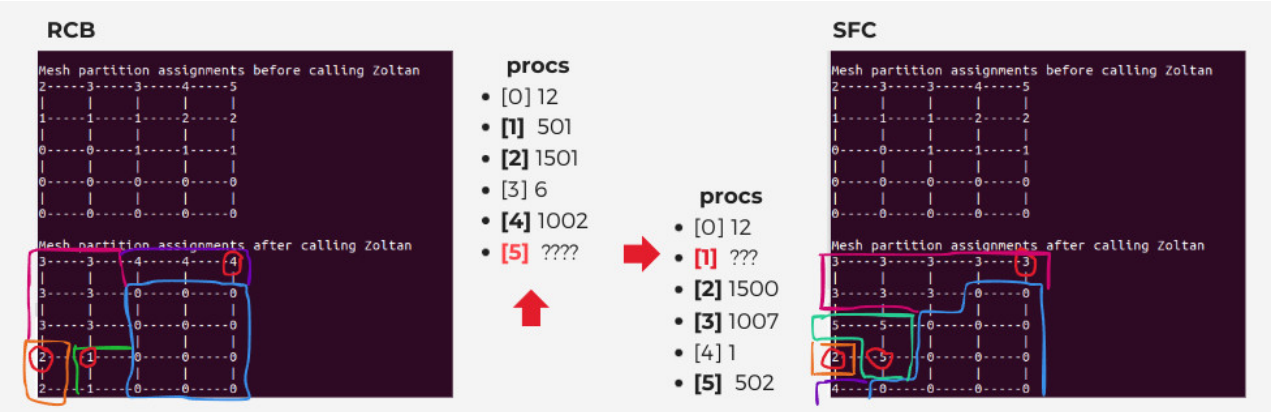


Figure 21 – Zeroed cores for both RCB (left) and SFC (right) algorithms on Zoltan. Numbers indicate the core ID of each matrix cell. Red circles indicate cells with a higher weight on the matrix. Other colors indicate core limits.

The problem with the zeroed cores is that the mesh is designed to work with the GMG v-cycles and having cores with no load/mesh at all, interferes with the fine-coarse interface operations like the interpolations, relaxations, and prolongations (the last two are part of the v-cycles - see sections 3.1.1 and 2.1.4 for more details) and ghost cells, which require data from other cores that in this scenario, have no data at all.

Figure 22 shows an example of this situation, highlighting the zeroed cores with a black marker. All patches neighboring the marked zone have ghost cells, which will not function properly as they attempt to communicate with cores that have no mesh and, therefore, no levels, patches, or control volumes to be virtualized by the ghost cells. Also, when the GMG iterates over those patches while running a v-cycle, relaxation and prolongation will be harmed by the malfunctioning ghost cells, which may introduce errors into the solution or fail altogether.

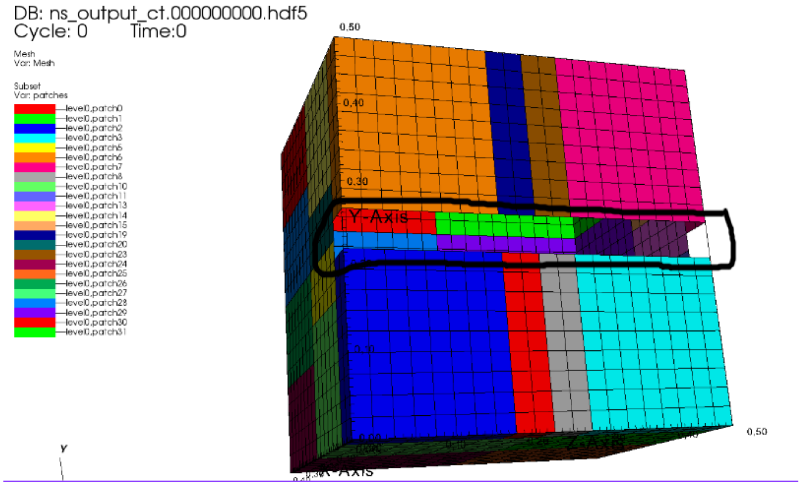


Figure 22 – Zeroed cores highlighted by the black marker

Depending on the case studied, this situation may be circumvented by some fail-safes (mostly in smoothers and interpolations) implemented on MFSim with minor implications to the solution of the discretized equations by the GMG, but that is not guaranteed, so, though it is not necessarily a kill zone like the badpoint, is a highly undesired situation which must therefore be avoided.

This is the second main challenge faced by the proposal described in section 3.2.

3.2 Many-approach Load-balancing

Now that the main challenges have been addressed, we can proceed with the proposal for the many-approach load balancing.

3.2.1 Level selection for weight calculation

The first change we proposed is in the scope of the WeCA (shown in algorithm 1), which feeds the weight matrix, which is then used by the RCB to balance the mesh.

The algorithm shows that the scope originally comes from the most refined level (ltop) to the second most coarse level (lbot+1), since refined levels tend to bear more weight than coarse levels. This setting usually works for most simulations, but not all of them. Depending on the case studied, it may be necessary to track some physical events at a coarser level while other events are tracked at a finer level.

For example, a simulation studying the corrosion of a tube caused by the inner corrosive flow may discretize the flow in terms of particles and track those particles until they collide with the tube, measuring where the corrosion occurs and at what rate. The particles will be tracked at the finest level, for better accuracy, while the collision, where the particles are deformed and converted into a liquid film (AGATI et al., 2022) (NARDECCHIA et al., 2015) (an object a lot larger than a particle), will be tracked in the level

directly below, where the tube is modeled as an immersed boundary (STIVAL et al., 2022) (SOUZA et al., 2022) (MORALES et al., 2023) (another object a lot larger than a particle).

In this simulation, tracking the particles is important as long as they collide with the tube, which is the main objective of the simulation. However, to track the particles, we need to refine the mesh since particles are small objects, and the larger the mesh, the more inaccurate the tracking will be. However, when the particles are set to collide, tracking is not necessary anymore, and we can work with a coarser level that sufficiently contains the immersed boundary representing the tube. Again, it is possible to cover the immersed boundary with the same refinement level of the particles, but this tends to create an overhead, multiplying the operations that were sufficiently solved in a coarser level, and since we are running those operations for thousands if not millions of timesteps, this additional refinement can severely impact the overall simulation's time.

For this simulation, the WeCA will correctly associate more weights to the areas where the particles are tracked, as they are in the finest level, and considering the resulting balancing doesn't produce bad points or zeroed cores, will assign more cores to the particle tracking operation. The problem with this balancing is that the main objective of the simulation is not to track the particle motion but to study how the liquid film formed by the collision of the particles with the tube is corroding it, which is tracked by a coarse level and therefore, results in less weight than the areas tracking the particles motion. Therefore, a better balancing should be able to attribute more weight to this operation rather than to the particle tracking one.

To address this situation, we proposed and implemented a change in the WeCA that enables users to specify the scope. This is done by packing the WeCA as a UDF (ALTERYX, 2024), which allows the user to change the algorithm without needing to recompile the full MFSim's code. Hence, depending on the case configuration, the user can set the scope to only some refinement levels, rather than all of them. Considering the example described earlier, a proper change in the WeCA would limit the loop from the level with the immersed boundary (ltop-1) to the second most coarse level (lbot+1), thus attributing more weights and, therefore, more cores after the RCB load balancing, to the areas where the liquid film is formed and interacts with the tube, corroding it, which is precisely what the simulation wants to know.

The UDF is implemented as a Fortran file, which is compiled at the start of the simulation as a shared library and then loaded on the fly into MFSim's memory space. Then is verified if the loaded library has an implementation of the WeCA. If it exists, the procedure pointer for the WeCA, later used by the load balancing operation to generate the weight matrix, will be assigned to the WeCA from the newly loaded library. If it doesn't exist, then an internal, default implementation of WeCA (shown in 1) will be assigned to the procedure pointer. Then, the flux continues, and when an imbalanced situation is

found, the load balancing operation is invoked. The procedure pointer configured earlier is then used to generate the weight matrix. This implementation will be used if a WeCA implementation was present at the UDF. If not, the default WeCA will. Figure 23 brings an illustration of this flux, considering the overall flux of the simulation.

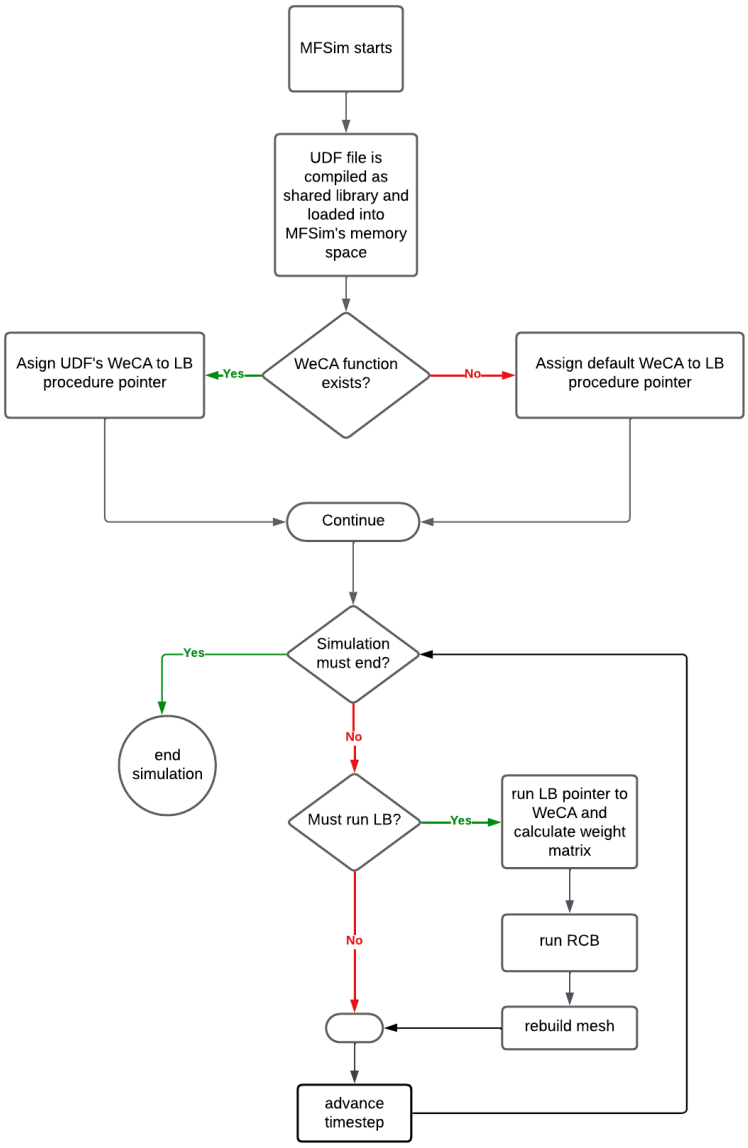


Figure 23 – WeCA UDF flux inside the general simulation flux

Taking the example simulation described earlier again, with a WeCA implementation at the UDF ranging from the level with the immersed boundary (ltop-1) to the lbot+1, when the load balancing operation is invoked during the simulation, the WeCA procedure pointer will be assigned to the UDF's WeCA and the produced weigh matrix will consider only the level ltop-1 to lbot+1, instead of the default ltop to lbot+1. Then the load balancing flux continues (as described in section 3.1.2), and assuming no badpoint or zeroed cores were produced, the new, balanced mesh will attribute more cores to the

areas where the liquid film is formed and interacts with the tube, which is exactly what the simulation want.

Since the WeCA UDF allows for more control over the load balancing operation, another use is to enable load balancing when it was previously impossible due to badpoints or zeroed cores. Depending on the case studied, a balancing ranging from ltop to lbot+1 may produce a mesh with zeroed cores, while the balancing ranging from ltop-1 to lbot+1, in the same case, may not. This, unfortunately, can vary wildly from case to case, making it nearly impossible to devise a heuristic or single-interval configuration that can safely balance the simulations. Again, this is precisely the scenario the UDF is designed for.

3.2.2 Time of Timestep

The second proposed change is in the triggers for load balancing and the identification of extra mesh load.

MFSim uses a block-structured adaptive mesh for the Eulerian space. The adaptability implies changes in the mesh during the simulation (see figure 4 for an example of this kind of mesh). These changes are generated during a **remesh** operation, which rearranges the mesh considering some criteria. After a remesh, the mesh stays static until the next remesh. The frequency and criteria used to trigger the remesh are configured by the user and are case-dependent. The critical part here is that when it comes to load-balancing, a remesh operation changes the mesh and requires a subsequent load-balancing operation, which is the default trigger.

Since the WeCA considers only the adaptive mesh (as this is the mesh that describes the domain and where the refinement levels are), any operation outside this mesh can't be adequately detected by the algorithm. This means that any heavy computing operation performed outside the mesh or, at least, not related to a refined area of the mesh, will be overlooked by the load balancing.

Then, it is necessary to, **first**, identify these heavy operations happening outside the mesh. **Second**, change the load-balancing operation to consider this outside mesh load, whether by increasing the weights of the areas where they occur, or by being a trigger to the load-balancing, or both.

To achieve that, we created a tool to analyze how the time is consumed in each timestep, the ToT.

While the timestep is, in itself, a measure related to how time is discretized in the simulation (see section 2.1.2 for more details), it consumes computing time to advance a simulation one slice of time (or timestep). Therefore, if we can measure this time and use it for load-balancing purposes, it may be possible to identify these extra mesh loads.

In fact, MFSim already computes the time consumed by the timestep in every core used in the simulation. However, upon analyzing this time, we discovered that it tends to be the same or have very close values for the used cores, which is highly unlikely to

happen unless we are running a fully balanced simulation. Figure 24 shows real data with this situation. An unbalanced simulation has the same (or very close) amount of time consumed by the timestep for all used cores, while we should be seeing more time consumed by the slower cores and less time in the faster cores.

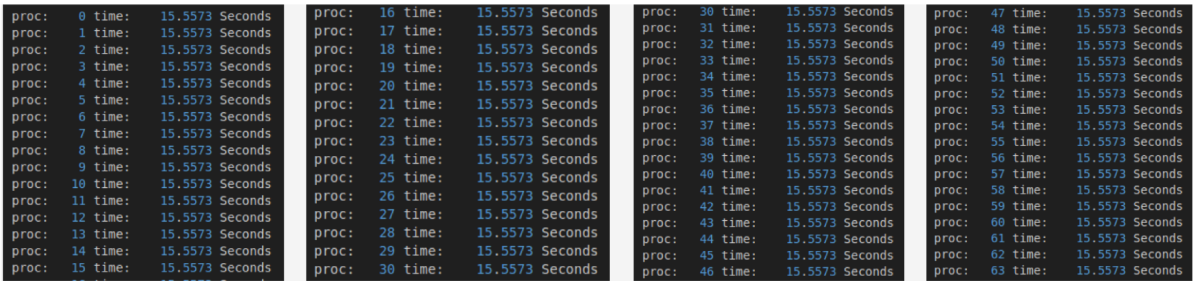


Figure 24 – Cores of a (real) simulation with the same time consumed in the timestep. Simulation: spray, 64 cores, 999th timestep

That indicates some operations are interfering with the computation of the time consumed by the timestep. So before using this data to influence the load-balancing, we must first refine the data, obtaining a time closer to what is expected in an unbalanced simulation.

Since MFSim uses MPI for its parallelization (see section 2.3 for more details), blocking communications between cores are the prime suspects of causing the interference. Hence, we searched the entire code and implemented a marker that was written into MFSim's log files each time a blocking communication was executed. This enables us to track not only the blocking communications during the timestep but also their location and the number of times they've run.

Knowing the location of the blocking calls is helpful to identify in which module they occur and therefore, how critical they are for the whole system, while knowing how many times they've happened, allows us to calculate the weight of those calls in the simulation, as more synchronizations means more overall time to run the simulation. If a blocking call is widespread but is in a very critical module, it may not be possible to refactor the code to have fewer of those calls. If not, we not only manage to find a group of operations interfering with the calculation of the time spent in a timestep, our main objective, but also find a possible improvement not related to the load balancing operation, but, to increase the overall performance, as less blocking calls means less time spent in timesteps which in turn means faster simulations.

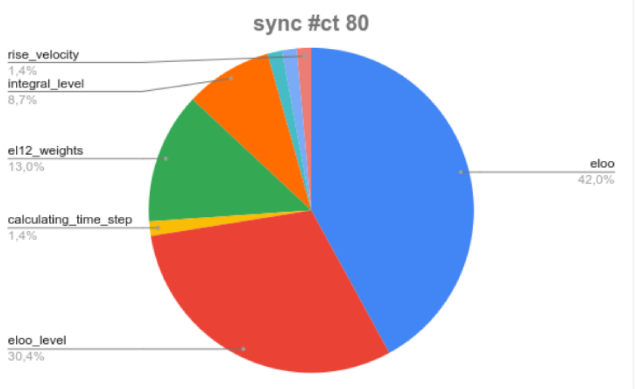


Figure 25 – Routines with blocking communications during a timestep

Figure 25 shows real data of this analysis over the 80th timestep of a simulation. The routines containing most of the blocking calls are related to the residual calculation, which is part of the GMG (see section 2.1.4 for more details). Since this is a critical module of the system (the part that solves the discretized equations), a reduction in these blocking calls isn't possible. But our main objective is achieved. We are aware that operations are interfering with the time measurement we seek to use for load balancing.

Knowing that we've implemented the ToT using a separate code from the already existing measurement of the time spent on the timestep. We did this to avoid interfering with other potential operations that utilize the existing calculation of total time spent in the timestep (which includes time spent on communication) and to have complete freedom over the implementation.

In that regard, we used a counter and two time registers. The counter is zeroed at the start of each timestep, and one of the registers collects the current time (OPENMPI, 2024). Then, we proceed to run the timestep. When an MPI blocking call is found, we set the second register with the current time and calculate the difference between the first register and the second. The difference is then stored on the counter. Then the blocking call is executed. While the blocking call is executed, no counting is done, as we intend to separate the time spent on computations from the time spent on communications. After the call is finished, we resume the counting by resetting the first register with the current time. This updated register will be used when the next MPI blocking call is found in the flux. This repeats until the end of the timestep is achieved. Then, we synchronize the ToT's obtained in all cores in such a manner that each core knows the ToT of every other core and we can safely use the data in the load-balancing. A graphical representation of the discourse ToT's flowchart is presented in Figure 26.

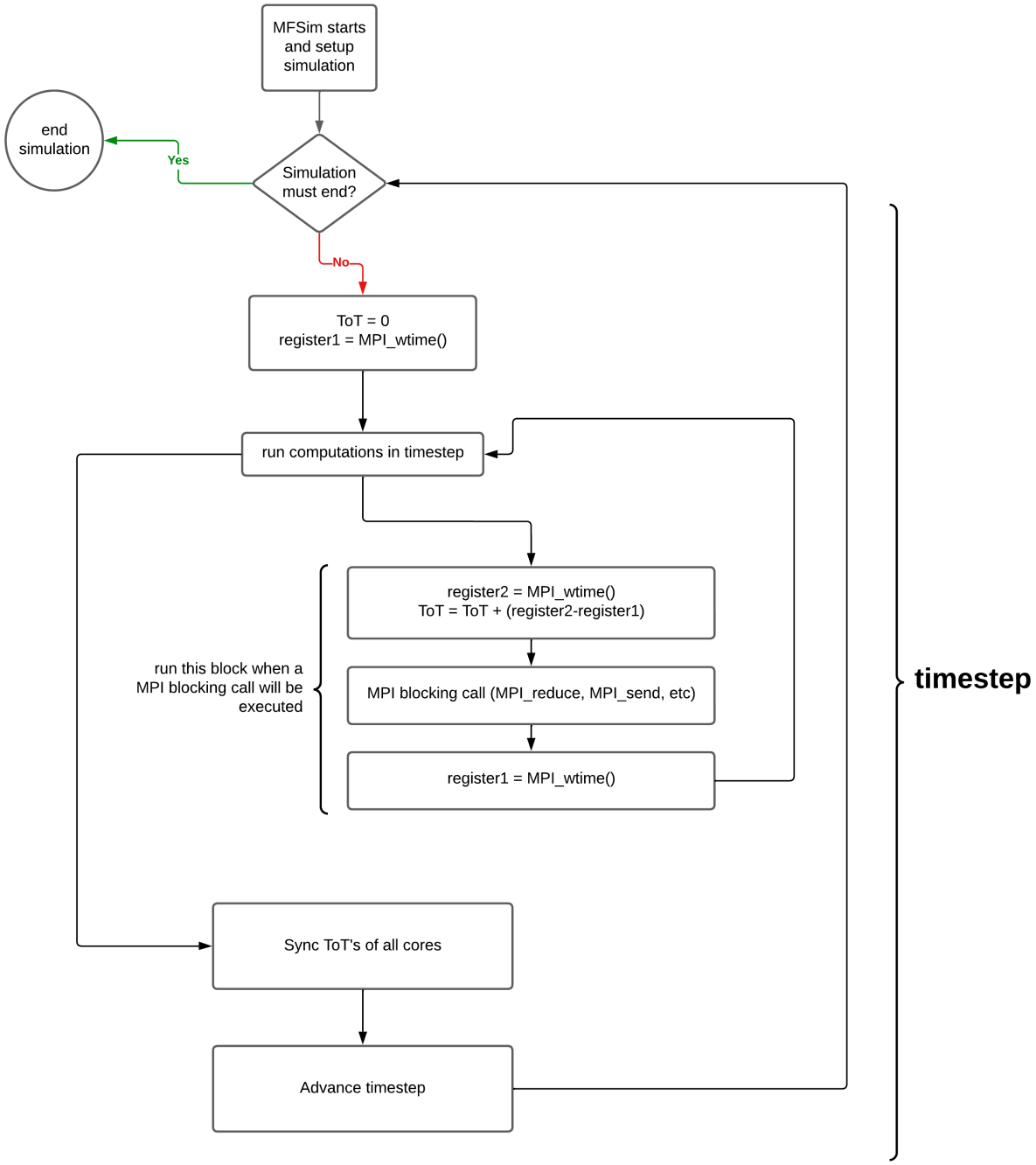


Figure 26 – ToT flowchart in each timestep

3.2.3 Loadbalancing decision

Now that we know how much time the simulation is spending on computations only, we can supplement the load balancing operation with this information to enable it to identify extra mesh loads and provide an additional tool for recognizing imbalance.

This can be done by simply analyzing the ToT of every core after they're synchronized before ending the timestep. We can generally compare the ToT's to find the fastest and the slowest core. If there is a significant difference between them, we can consider the

simulation unbalanced and a load-balancing operation necessary.

But, as explained before (in section 2.3) MFSim uses a block-structured adaptive mesh. That means the mesh and, therefore, the simulation is dynamic, and an overloaded core now may not be in the future as the simulation advances. So, analyzing a single ToT may enable us to detect an imbalance now, but show nothing of how the load is moving with the simulation.

For example, a simulation where a fluid is moving throughout a cavity, like fuel being transported by a tanker. The fuel is moving from one side of the tank to another as the tanker moves and the simulation is tracking the fluid movements. The tank is the domain, which is discretized into a mesh, with adaptability following the movement of the fuel. This means that for some time, there will be a greater load on the left side of the domain, and as the fuel moves, the load will migrate to the right side, then to the left side again, repeating the cycle. This phenomenon is called slosh, and Figure 27 provides an animation showing how the fluid can move inside the tank.

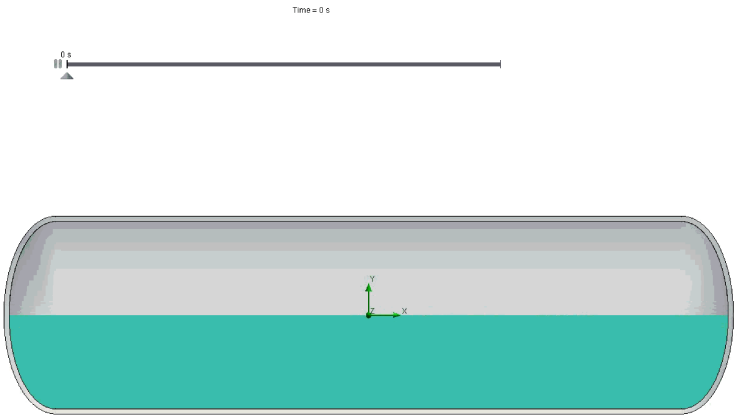


Figure 27 – Animated example of a slosh phenomena (PATEL, 2025)

By analyzing a single ToT we can probably detect where there is more load now and assign more cores to the area. Considering the example, if there is more load on the left side of the tank/domain, we will assign more cores to the left side. And when there is more load on the right side, we will assign more cores to the right side of the mesh. The problem here is that during the simulation, this cycle will repeat endlessly, and if the load balancing operation is costly, it may significantly impact the simulation.

In other words, though we may be able to find an area overloaded in the mesh by analyzing a single ToT, that doesn't mean this area will continue to be overloaded for the next timesteps, and by assigning more cores to this area via load balancing, this balancing may soon be unnecessary. In this scenario, we may end up running multiple balancing operations in a very short time. So, tracking the ToT's for more than one timestep may be interesting to see if an unbalanced situation is temporary or more permanent.

Additionally, we have adaptability in the mesh. Therefore, a remesh operation, as seen in MFSim (see Section 2.3), implies that we have at least two types of timesteps. One is where the mesh isn't changed and we only perform the operations to advance the simulation's time (solving the discretized equations, running the physical models, producing output data, etc.). Two, where we have a remesh, a rearranging of the mesh by some criteria, and the same operations regarding the advancement of the simulation time. This implies the ToT of a timestep of the first type is expected to be different from the ToT of a timestep of the second type, as there are more operations in the second. The real data shown in Figure 28 presents this exactly. The timestep with no remesh (on the left) is at least 5 times faster than the timestep with a remesh (on the right).

Figure 28 – Left: time spent in a timestep with no remesh operation. Right: time spent in a timestep with a remesh operation. Simulation: spray, 64 cores, 939th timestep (left) and 940th timestep (right)

If we track the ToT's without discriminating the type of timestep, we may disturb the analysis by detecting a load that only exists in remesh timesteps and, therefore, aren't exactly related to computational operations of the simulation.

To summarize, we need to track the ToT's by more than one timestep and separate ToT's from timesteps with remesh from those without it.

To do that, we created two memories for each type of timestep. One for the ToT from timesteps without remesh, the normal ToT's, and one for the ToT's from timesteps with remesh, the remesh ToT's. This gives us more than one ToT to analyze, which, depending on the simulation and the size of the memory, allow us to detect situations where a load balancing may end up soon replaced by another load-balancing and, at the same time, will enable us to treat each type of timestep separately.

Each memory is composed of the lowest and highest ToT, the difference between them, and a flag to indicate if the ToT was generated in a timestep with a load-balancing operation. The memories also store data from the newest to the oldest, using a FIFO (GEEKSFORGEEKS, 2024b) policy to prevent overflowing (the memories, of course, are limited).

To evaluate the memories, we initialize three counters: increase, decrease, and stability. Then, we loop throughout the memory comparing each ToT with its predecessor. If the current highest ToT is greater than its predecessor, then we have an increase. If slower, a decrease. If equal, a stability. Then, we try to obtain the tendency of the ToT's behavior in the memory. To achieve that, we calculate the percentages of each counter. If the rate of increases is greater than the percentages of decreases and stability, we consider that

the ToT's are showing a tendency for increase. If the percentage of decreases is greater, we believe the ToT's to have a decreasing tendency, and, if there is a greater percentage of stability, we consider the ToT's to be somewhat stable. If we can't identify one of those 3 tendencies, we believe the ToT's to be fluctuating. We also calculate the highest and lowest ToT in memory and the difference between them.

With the four possible tendencies for the ToT's, we can start to decide if a load-balancing operation is necessary or not. The most obvious scenario is when we tend to an increase in the ToT's. That may indicate the simulation is getting slower, and load balancing should be considered. If we have a tendency of stability or decrease in the ToT's, the simulation is most likely getting faster, and load-balancing isn't necessary. The trick scenario is when we have a fluctuation in the ToT's. For that case, we use the difference between the lowest and highest ToT, which shows us how far in the memory the slowest core deviates from the fastest one. We compare this to a user-configured threshold, and if the difference exceeds the threshold, load balancing should be considered. If not, it shouldn't.

This evaluation is conducted for each memory and, at its conclusion, provides recommendations regarding load balancing, considering the load in both normal timesteps (represented by the normal ToT's) and remesh timesteps (represented by the remesh ToT's). That way, we managed to analyze the load separately for each type of timestep, and now we have combined both analyses to determine whether load-balancing should be executed or not.

Again, the most obvious decision is when the analysis of both normal and remesh ToT's indicates that load balancing should be considered. The same applies when both indicate that load balancing shouldn't be considered. Those are the straight-to-the-point cases. The other two are more complex.

When the analysis of the normal ToT's indicates that load balancing should be considered, while the analysis of the mesh ToT's indicates that load balancing shouldn't be considered, we consider that the load balancing should be executed.

Inversely, if the analysis of the normal ToT's indicated that load-balancing shouldn't be considered while the analysis of the remesh ToT's indicated the load balancing should be considered, we assume that the load balancing shouldn't be executed.

The motive behind this decision to prioritize the analysis of the normal ToT's is that this kind of ToT's are (or should be) the majority of the timesteps in a simulation. That means if the load is increasing while the mesh is "static" (remember, changes in the mesh occur during a remesh), more load is likely being added to the simulation outside the mesh, which is the exact source of imbalance we are looking for. Therefore, it makes sense to prioritize a recommendation of balancing coming from the normal ToT's over the remesh ToT's.

Also, if the analysis of the remesh ToT's indicated a need for load balancing while the

analysis of the normal ToT indicates otherwise, it is possible (and very likely) that the mesh is getting more complex and by extent, more challenging to remesh as the simulation advances in time. That could explain why the remesh ToT's analysis detected a load in those operations. But, without the load being detected in the normal ToT's too, we can conclude that while the remesh operations are getting more expensive in terms of time (and probably other resources), the computing of the timestep normal operations (solution of the discretized equations, generation of output data, etc), which are the computations that compose the simulation itself, are not. In that scenario, it makes sense to postpone a load-balancing operation until the load is detected in the normal timesteps as well, especially if the load-balancing operations are costly.

Continuing, after the decision is finally made, the load-balancing operation is either applied or not. If applied, the memory of the ToT will register that for later use.

This leads us to the next section, where we elaborate on the load-balancing policies, which are also part of our proposal for balancing multi-physics simulations with a block-structured adaptive mesh.

However, before we proceed, we must clarify a few points regarding the load-balancing decision algorithm explained in this section.

The first thing is that the ToT evaluation we proposed here is heavily dependent on the size of the memories used in the analysis and in the threshold used to solve the fluctuation scenario. A too-short memory may be unable to capture a slow-growing load. At the same time, a large memory may flood the analysis with stable or decreasing ToT's for most of the memory while ignoring the newly added ToT's where a sharp increase can be observed. Additionally, for the threshold, load balancing may not occur if the limits are too high. If they are too low, load balancing may occur more than necessary, which brings us to the second thing.

As we stated earlier, nearly everything is case-dependent regarding multi-physics simulations. Load-balancing is not an exception to this (see section 2.2 for more details) and while the ToT analysis is a useful tool to improve the load balancing in those simulations, as we are going to present in chapter 4, is very difficult, if not impossible to elaborate a single or automatic configuration for the size of the memories and the threshold.

That is why we left those for the user to set as they see fit for their simulations. We set some default values for both memory size and threshold, but these are only default values and may not work for every simulation.

3.2.4 Loadbalancing policies

Considering the inherent constraints imposed by the multi-physics simulations, as we stated before, and the limitations of our propositions, as explained in the last section, we decided to expand the load-balancing decision by delegating more power to the user.

Initially, the load balancing decision in MFSim was to be executed at every remesh (LIMA et al., 2012). This approach may be effective for some simulations, but it may not work for others. In our experience, for industrial cases, it usually doesn't work. But, it is a valid policy if the user sees fit and was kept.

Another policy was a change from the last one. In this case, the load balance will be executed only in the first remesh operation. After that, no more load balancing will be executed.

Other policies developed based on the concept of threshold are used in the algorithm described in Section 3.2.3. The possible thresholds are:

- ❏ for every remesh until a specific one
- ❏ for every remesh after a specific one
- ❏ until a certain timestep
- ❏ after a certain timestep
- ❏ by analyzing the normal ToT's and testing if the average difference between the lowest and highest ToT's in the memory is above the threshold
- ❏ by analyzing the remesh ToT's in the same way of the above

The last policy is the automatic one, which combines the other policies. First, it tries to run the load balancing at the start of the simulation. If that is not possible, go for the analysis of the normal ToT's as described in the threshold policy. If that is not possible, analyze the remesh ToT's as described in the threshold policy. If that is not possible, try to decide by running the evaluation described in section 3.2.3, the ToT analysis. If none is possible, don't balance.

Summarizing, we have the following load-balancing policies:

- ❏ balance at the start of the simulation
- ❏ balance for every remesh
- ❏ balance by threshold:
  - for every remesh until a specific one
  - for every remesh after a specific one
  - balance until a certain timestep
  - balance after a certain timestep
  - balance by average ToT difference of the highest and smaller ToT
- ❏ balance by ToT analysis
- ❏ automatic

3.2.5 Rollback of loadbalancing

The final part of our proposal addresses the inherent risk associated with the load-balancing operation. As we explained in section 3.1, a load-balancing operation must not generate **badpoints** or **zeroed cores**, and everything we proposed in the last sections is to reduce the chance of those problems occurring.

But if they do occur anyway? There is always the risk.

That is why we also implemented a functionality inspired by the DBMS software: rollback.

In a DBMS, a rollback creates a "safe zone" where changes can be made to the database without actually changing the database permanently. If something goes awry, the changes are undone, thus preserving the database (GEEKSFORGEEKS, 2024a). We've implemented this concept for multi-physics simulation.

From the problems we described in section 3.1, one is a killzone and the other is a highly undesired situation. MFSim already has the means to identify both conditions, so we implemented an algorithm that combines this identification part with the load balancing, evaluating if the generated balanced mesh has any **badpoint** or **zeroed core**. If any of those are found, the balanced but faulty mesh is discarded, and the original mesh, previously copied as a backup, is restored. The simulation then continues. If this occurs inside a remesh, the remesh is executed normally in the restored mesh, and the simulation continues. Figure 29 shows a flowchart of this algorithm.

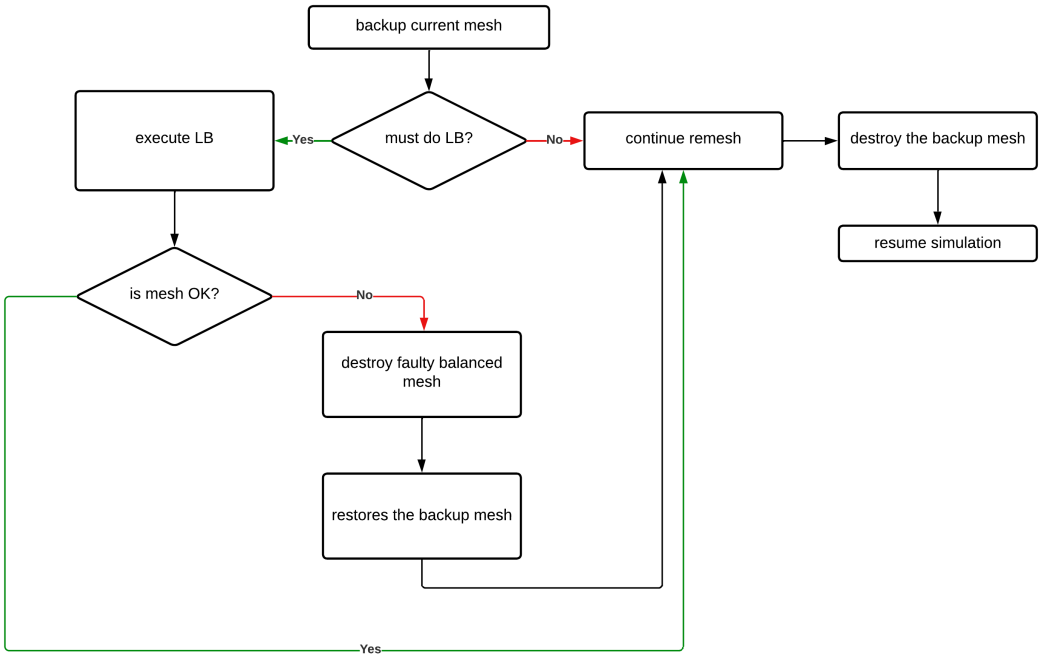


Figure 29 – Rollback of failed load-balancing operations

A register of failed balancing will be maintained, and if the load balancing continues

to fail when triggered again by the policy, the system will disable the load-balancing operation altogether.

Chapter 4

Experimental Evaluation and Analysis

This chapter presents the experiments performed to test and validate the proposition described in section 3.2.

The first section describes the environment used. The following section describes the simulation used, the test cases, both industrial and academic. Then, we describe the metrics used to evaluate the proposition. The last section presents the impact of the proposal on the test cases, which compose the results and discussion.

4.1 Environment

The environment used to run the experiments consists of 4 different machines and the MFSim code already described in section 2.3. One machine is a personal computer, with high-end hardware, which we'll assign the label **Ryzen**. The second is a workstation, with the label **Workstation**. The third is a server node, with the label **Node**, and the last is a cluster, with the assigned label of **Cluster**. Details for each of the used machines are provided in table 2.

Table 2 – Configuration of the machines used to test and validate the proposition

Machine	Processor & RAM	Operating System	Software Stack
Ryzen	1x AMD Ryzen 9 5900x 12c/24t - 64 GB DDR4	Ubuntu 22.04 LTS	GCC 12/GCC 13
Workstation	1x Intel Xeon Gold 6238R 28c/56t - 64 GB DDR4	Ubuntu 22.04 LTS	GCC 12/GCC 13
Node	1x AMD Epyc 7543P 32c/64t - 128 GB DDR4	Oracle Linux Server 8.9	GCC 13
Cluster	9x nodes: 2x AMD Epyc 7452 32c/64t - 256 GB DDR4 7x nodes: 2x Intel Xeon Silver 4114 10c/20t - 96 GB DDR4	CentOS 8.2	GCC 12/GCC 13

The processors for each machine are described using the notation number_of_cores**c** / number_of_threads**t**. This means a processor with `12c/24t` has 12 cores and 24 threads.

Also, the **Cluster** is a collection of many machines, all of which are eventually used for the tests. Still, the simulation is run on the compute nodes, which are machines reserved for computation purposes. That's why we only describe those machines of the **Cluster** in table 2.

Another critical detail in the previous table is the *Software Stack* column. This refers to the software collection required to run MFSim. Though we cited only the codename for each collection to indicate versions in which we run the tests (and any possible influence on the results), this part of the environment will be detailed further.

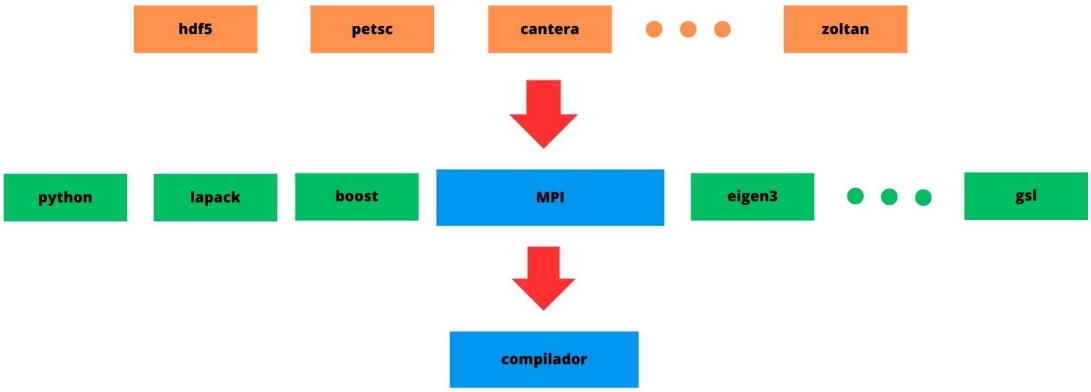


Figure 30 – Stack hierarchy. Blue boxes indicate the main components. Green boxes indicate libraries that depend on the compiler only. Orange boxes indicate libraries that depend on the MPI implementation and the compiler.

Generally, a stack is constructed around the compiler, which is built specifically to run MFSim, though it can also be used for other purposes. Above the compiler are built those libraries and tools that depend only on the compiler. Then, an MPI implementation is added to the stack, and above it, any library or tool that depends on MPI. Figure 30 brings an illustration of the stack organization and its hierarchy.

The stack's codename is generated considering the major version number of the compiler. Table 3 presents the libraries and tools composing each of the software stacks used. Most libraries and tools have very close versions, with some having identical versions and some belonging to only one stack.

Table 3 – Software Stack composition

Component	GCC12	GCC13
compiler	gnu 12.2	gnu 13.2
python	3.11.2	3.11.4
valgrind	3.20.0	3.21.0
cmake	3.25.2	3.27.1
lapack	3.11	3.11.0
swig	4.1.1	4.1.1
boost	1.81.0	1.82.0
eigen3	3.4.0	3.4.0
fmt	9.1.0	10.0.0
scons	4.5.2	4.5.2
googletest	1.13.0	1.13.0
coolprop	6.5.0	6.5.0
suitesparse	7.0.1	7.1.0
gsl	2.7.1	2.7.1
metis		5.2.1
openmpi	4.1.5	4.1.5
hdf5	1.14.0	1.14.1
trilinos/zoltan	14.0.0	14.2.0
scalapack	2.2.0	2.2.0
parmetis		4.0.3
mumps		5.6.1
petsc	3.19.0	3.19.5
slepc	3.19.0	3.19.2
sundials	6.5.0	6.6.0
cantera	2.6.0	2.6.0

4.2 Simulations and test cases

Now, we describe the simulations we tested the proposition upon. The three industrial-scale simulations were used because it was not possible to balance without the proposition, and they served to verify whether the proposition works and its limitations. The academic case was already balanced without the proposition and served as a control group. Additionally, since this simulation runs significantly faster than the others, it also enables us to conduct a statistical analysis, which requires numerous runs of the simulation to collect data for later processing.

4.2.1 SPRAY

The first industrial case we tested was the SPRAY. This case simulates a flow inserted by an injector into a tube of a boiler located at one of Petrobras refineries. This simulation aims to investigate how the injector-inserted flow reduces the corrosive capabilities of the flow already inside the tube by tracking the injected flow at a particle level and examining

how it propagates within the tube. This tracking enables us to analyze the behavior of the flow-induced corrosion in the tube for later mitigation. Figure 31 shows a rendering of SPRAY, illustrating both the injector and the injected flow inside the tube.

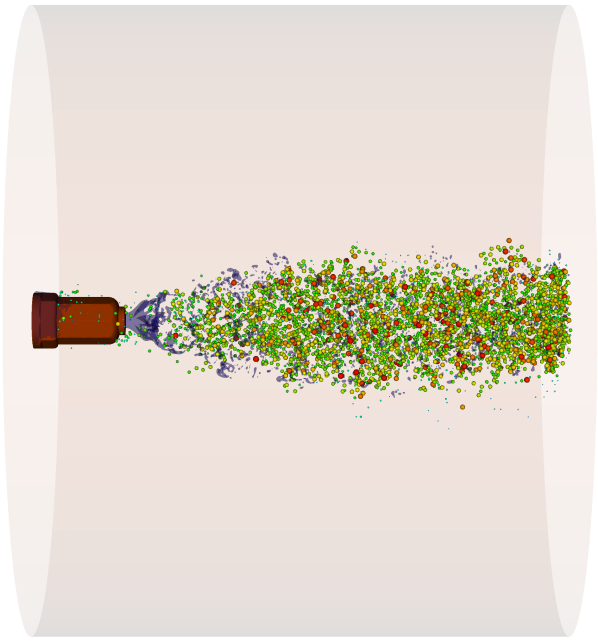


Figure 31 – Injector-inserted flow simulated at SPRAY

The domain has dimensions of 0.5 x 1 x 1 meters (X, Y, Z axes), discretized into a mesh with a minimum precision of 0.0125 meters and a maximum precision of 0.0004 meters, distributed across five refinement levels and 64 cores. The memory requirements start at 7 GB and continue to grow as more mesh is generated by the adaptive refinement and more particles are converted from the fluid. At the 20,000th timestep, memory consumption is at least 60 GB and continues to grow after that. Accompanying the memory growth are load-generating objects, which, by the simulation start, are measured in 530 thousand elements, and by the 20,000th timestep, there are at least 42 million elements. The timestep starts at 0.5 seconds (without load balance). It increases to 5 minutes at the 20,000th timestep, indicating that this is one type of simulation that begins relatively small and light and gradually becomes heavier and slower during its execution.

Thanks to that, this simulation was run only on the Cluster, and it took several weeks to progress from the start of the simulation to the 20,000th timestep.

4.2.2 FEIXES

This is the second industrial case, simulating a turbulent flow inside a boiler at one of Petrobras refineries. The simulation examines the vortex-induced vibration phenomenon resulting from turbulent flow over certain internal structures of the boiler. Hence, the simulated physics are fluid dynamics, immersed boundary, turbulence, and fluid-structure

interaction. The objective of the simulation is to measure the maximum workload of the boiler. Figure 32 provides a rendering of the simulated boiler.

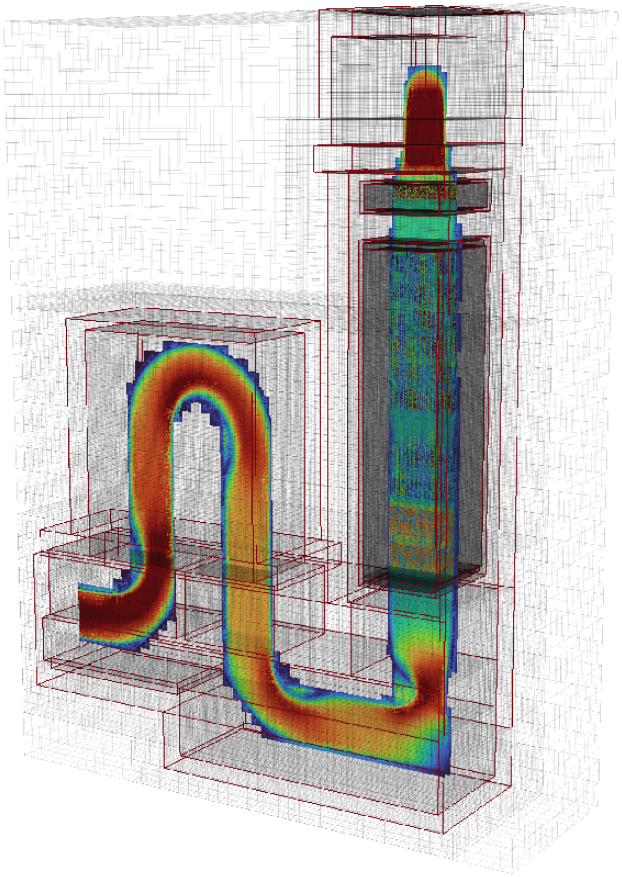


Figure 32 – Boiler simulated on FEIXES

The domain has dimensions of 8 x 32 x 24 meters (each number refers to the respective axis: X, Y, Z), discretized into a mesh with a minimum precision of 0.5 meters and a maximum precision of 0.015 meters, distributed across five refinement levels (the more refined, the more precise). There are at least 193 million load-generating elements, primarily control volumes and immersed boundary points, in this simulation. All of this load is distributed over 64 cores, and thanks to its large memory requirements, at least 230 GB of RAM, with some runs achieving nearly 900 GB of RAM, this case was run only in the Cluster. It is also the slowest of the cases, requiring an entire week's worth of execution to just initialize the simulation.

4.2.3 CALDEIRA

The last industrial case we tested, the CALDEIRA, also simulates a boiler of one of Petrobras refineries. The simulation investigates the combustion of the turbulent flow inside the boiler, measuring both the chemical reaction rate, how they occurred, and the generated by-products. Thanks to this, the physics underlying this simulation include

fluid dynamics and turbulence, which are used to study how chemical compounds flow inside the boiler. The immersed boundary method is employed to model the boiler structure and its influence on the flow, while multi-phase and chemical reaction models are used to simulate the combustion of the flow. An illustration of the boiler and flow is given in Figure 33.

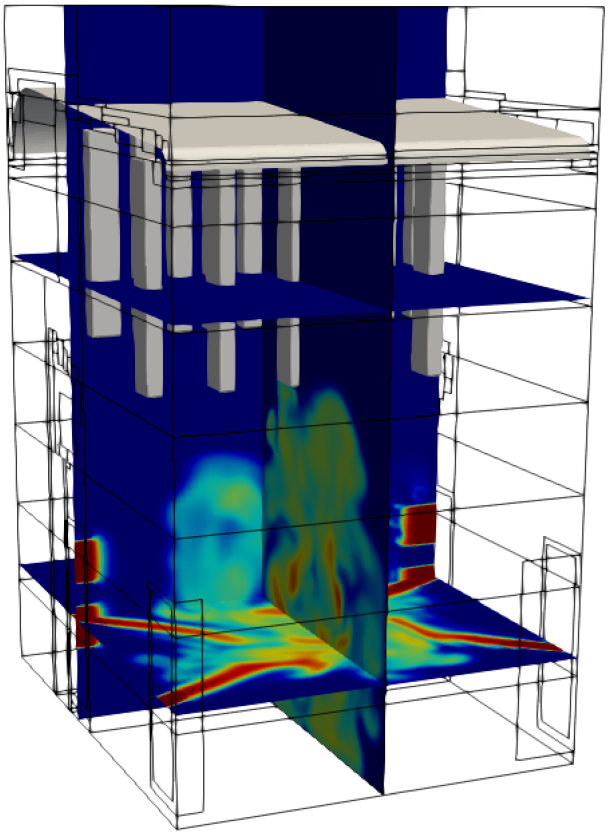


Figure 33 – Rendering of the simulated boiler, showing the dispersion of one of the chemical elements inside the boiler

The domain has dimensions of 8.584 x 15.022 x 10.756 meters in each axis, discretized into a mesh with a minimum precision of 0.134 meters and a maximum precision of up to 0.0335 meters, distributed over 2 refinement levels. This case was run in 3 different configurations: one with a reduced version of the boiler, which we call CALDEIRA SIMPLE; One with the full version of the boiler, but without detailed chemistry, the CALDEIRA FNR (Full No Reaction); One with the complete boiler and with detailed chemistry enabled, the CALDEIRA FR (Full and Reaction). The number of load-generating elements varies throughout the versions, with SIMPLE having at least 2 million elements, FNR having 11 million elements, and FR having 16 million elements.

The configuration of the load-generating elements also differs from the versions: SIMPLE and FNR have mainly control volumes and immersed boundary points as load-generating elements, while FR has those two and adds another, the chemical reactors, which are extra mesh load-generating elements. This is important because one of the

objectives of this proposition is to detect all load-generating elements in a multi-physics simulation, including the extra mesh ones.

This load was then distributed over 32 cores, with memory requirements up to 50 GB of RAM. All runs were executed in the Node.

4.2.4 SPHERE

This is an academic test case developed by Rubens (JUNIOR et al., 2005) as part of his thesis, with references from other similar works. The main objective is to study the flow around a stationary sphere, especially the vortex (turbulent flow) created at the back of the sphere. Figure 34 provides an animation of the full run of this test case.

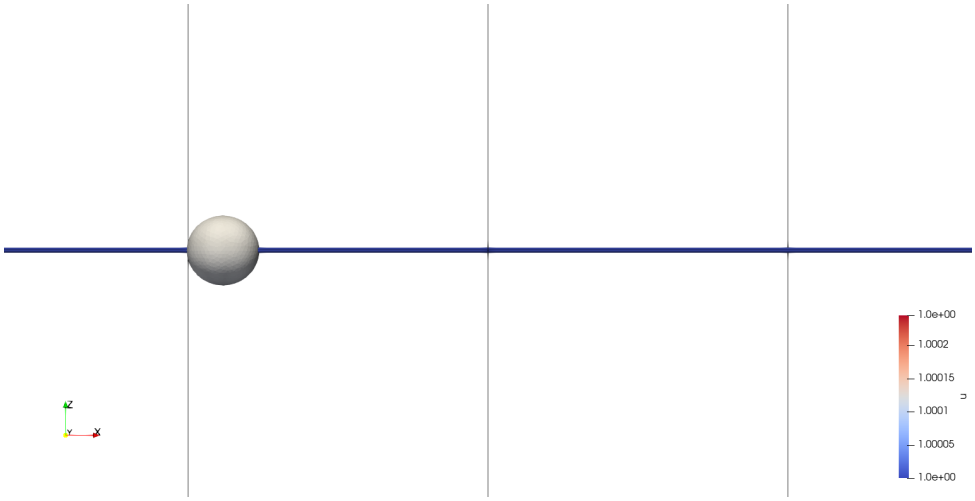


Figure 34 – Animated example of the sphere test case

The domain has dimensions of 1.024 x 0.512 x 0.512 meters (X, Y, and Z axes, respectively), discretized into a mesh with a minimum precision of 0.032 meters and a maximum precision of up to 0.002 meters, distributed across four refinement levels. The case begins with 114,000 load-generating elements, primarily control volumes, distributed across four cores, with an average RAM consumption of 450 MB. Throughout the simulation, thanks to the adaptive refinement, the load is increased up to 243 thousand load-generating elements.

4.2.5 Summary of Test Cases

To summarize the test cases, we created Table 4 presenting the most relevant data regarding the scope, simulated physics, and size (in terms of control volumes, particles, or other load-generating elements) of each test case.

Test Case	Scope	Physics	Size	LB criteria
SPRAY	Industrial	Fluid Turbulence Particle Multi-phase Volume of Fluid	+42 million	ToT analysis
FEIXES	Industrial	Fluid Turbulence Immersed Boundary Fluid-Structure Interaction	+193 million	auto
CALDEIRA	Industrial	Fluid Turbulence Immersed Boundary Multi-phase Chemical Reaction	2 to +16 million	auto
SPHERE	Academic	Fluid Turbulence Immersed Boundary	243 thousand	all

Table 4 – Test Cases Summary

4.3 Metrics

As explained in section 2.2, the main objective for load-balancing is redistributing the workload throughout the simulation used cores, as best as possible, in such a way that the overall time spent to run the simulation is reduced. That gives us two different measuring data to analyze: load-generating object distribution and time spent.

The first data, the **load generating object distribution**, can be obtained by counting, throughout the cores, the control volumes, particles, and other objects in the simulation over which computations are executed and therefore, workload generated (see section 3.1.2 for details). A well-balanced simulation will have cores with roughly the same number of those objects. A balanced enough simulation will present cores with an amount of load-generating objects that is possible to achieve, respecting the applicable constraints (see section 3.1 for details), and that reduces the overall time spent on the simulation.

The difference between a well-balanced and balanced enough simulation is necessary because of the complexity of the load-balancing operation, as described before. In general, the more physics a simulation has, the more complex and unbalanced it tends to be. So the ideal result will be very unlike, and we will have to settle for what is possible anyway.

The second data, the **time spent with the simulation**, can be measured using two different but related approaches. The first approach analyzes the overall time spent in the simulation, while the second analyzes the time spent in a range of timesteps. Both analyses are related because the overall time spent in a simulation heavily depends on the time spent on the timestep. The slower the timesteps, the greater the time spent on the

simulation, and vice versa. Which approach to choose when evaluating load-balancing, depends on the simulation.

In academic simulations, like SPHERE (section 4.2.4), which tends to run in a matter of minutes or a few hours, the first approach is the best choice, as enables us to evaluate the impact of the proposition in the whole simulation and not only in some parts. But, for industrial scale cases like SPRAY, FEIXES, or CALDEIRA (sections 4.2.1, 4.2.2 and 4.2.3 respectively), which run for weeks or even months on end, this approach may not be a possibility.

First, we need at least two runs of the same simulation (more details to follow) for comparison, which takes a considerable amount of time. For example, each FEIXES execution requires at least one week of runtime just to initialize all the simulated physics and another three months to run entirely. Since it is a memory-intensive simulation, it only runs on the Cluster and requires at least 2 nodes. In the best scenario, with all the required resources available all the time, it would be necessary to wait three entire months to run two versions of FEIXES, using 4 of the nine most powerful nodes of the Cluster, just to analyze if the proposition had produced a balanced enough simulation or not. This is not feasible in reality, both in terms of resources (the most realistic is having resources for just one run at a time) and time for analysis and adjustments for the proposition if needed.

The second problem with evaluating the overall time spent in the simulation is that some simulations, especially those at an industrial scale, may "grow" over time, creating more load-generating objects as they run. While it is fascinating to analyze how the proposition responds to this growth, it also limits the proposition's applicability, as the simulation workload may increase to a point where load balancing alone is insufficient to redistribute the load properly and, therefore, reduce the time spent. In other words, the simulation may end up adding more load than the available resources can handle (in terms of computing power, cores, and memory). In that scenario, load balancing won't have any practical effect.

This may exacerbate the situation, as it adds more tasks to the simulation. Since it is not possible to add more resources on the fly (a characteristic that MFSim inherits from its parallelization technique with MPI (UNIVERSITY, 2024)), the only way to add more resources to a simulation is to create a checkpoint, stop the simulation and restart the simulation with more cores assigned to it. This changes the evaluation entirely, as it is very different to analyze the redistribution of X elements over 64 cores than to investigate the redistribution of the same X elements over 128 cores.

That is when the second approach comes in handy. Instead of analyzing the whole time spent in the simulation, let us examine the time spent with some timesteps, which, as explained before, compose the entire time spent and compare that window of execution to verify if the proposition achieved the desired objective.

It is valid to point out that analyzing only some timesteps may also mislead the analysis conclusion, though, as after the timesteps are evaluated, the time difference (if it exists) may disappear between the control simulation and the load-balanced simulation. For example: by comparing the same range of timesteps between the control and balanced simulations, we find out that the balanced timesteps are 20% faster than the control simulation timesteps. But, after hundreds of timesteps later, the difference between the timesteps disappears, invalidating the conclusion.

We argue against that because, first, depending on the load-balancing policy defined by the user (more details in section 3.2.4), as soon a new situation of unbalance is detected, a new load-balancing operation will be executed, ensuring the timesteps of the balanced simulation stays faster than the timesteps of the control simulation. Second, since the overall time spent with the simulation is heavily dependent on the time spent in the timesteps, even if a single timestep of the balanced simulation is faster, then the overall balanced simulation will be faster. In general, analyzing a range of timesteps not only enables us to evaluate the impact of the proposition in large simulations within a reasonable time frame but also tends to produce acceptable results. It may not be the exact time gain with the load balance, but it will give us something to compare.

To finalize this section, we must define the concepts of control simulation and balanced simulation. Both of the simulations run the same case, which means, the same simulated physics, same starting conditions, same mesh, same extra-mesh objects. The only difference in configuration between than is that the **control simulation** doesn't have any load balancing enabled while the **balanced simulation** has load balancing enabled.

Additionally, both simulations must use the same environment, i.e., the same software stack, the same number of cores and memory requirements, and the same machine(s) used. The machine requirements for the test are crucial when using the Cluster, as AMD nodes are generally faster than Intel nodes. It's interesting to evaluate the performance of simulations in both types of nodes simultaneously to see how the proposition performs in heterogeneous hardware. However, to have a proper comparison, we must ensure that the control and balanced simulations run in the same configuration. For example, if the control simulation was run in 1 AMD node and 1 Intel node, the balanced simulation must also run 1 AMD node and 1 Intel node, even if the nodes aren't the same. That brings us to another requirement.

As nearly everyone in MFLab uses the Cluster, all simulations we run on it for this work were ensured to have full use of the nodes. That means we avoid using nodes that already have another simulation running to prevent any workload for the other simulation from disturbing our simulation and, therefore, inducing load by an external factor. This is also another motive why we described a few paragraphs earlier in this section that analyzing the whole time spent was not always feasible.

Summarizing, the metrics we use for this work are:

- load generating object distribution: the more even throughout the used cores, the **better**
- time spent on the timestep: the smaller, the **better**

4.4 Results

In this section, we present the results of applying the proposition described in section 3.2 in the test cases and environment described in sections 4.2 and 4.1 respectively. This section is organized as follows.

The first subsection will present the overall impact in a large simulation, providing a more macro analysis of the whole proposition, considering the metrics defined in 4.3.

The second subsection will present a more detailed analysis of the proposition, evaluating how the ToT and the policies impact the load balancing and, by extension, the simulations.

The third subsection examines how physics affects the load-balancing operation and how the proposition addresses this issue.

4.4.1 Macro Analysis

To start the evaluation of the proposition described in section 3.2, we are going to present the impact in the largest simulation we tested upon, the FEIXES 4.2.2.

For this, we have two runs of FEIXES. One with load balancing enabled and one without. The simulations are the same, except for whether the load balancing has been enabled. The balanced simulation used the automatic load balancing policy with default threshold and memory sizes values. This simulation was run for more than 8 thousand timesteps, while the non-balanced was run for only 821 timesteps.

We will use data from the first 800 timesteps as the primary source for our evaluation, as both simulations cover this range. However, we will also extrapolate data from the non-balanced simulation for additional analysis. The methods used to extrapolate the data are also presented.

The first point for the analysis is the first timestep, present in both simulations, and the second is the 4000th timestep, which also serves as the point for extrapolation. The reason for choosing this specific point is that it is present in the data for the balanced simulation and is neither too far nor too close to the last recorded timestep of the non-balanced simulation.

The reason the non-balanced simulation was not run beyond the 821st timestep is the time required to achieve the second point in the balanced simulation, which is at least three weeks.

Running any simulation requires resources, which, in this case, are Cluster resources, the same resources used by all Cluster-enabled simulations in MFLab. So, let us suppose it would require 3 weeks to run the non-balanced simulation up to the 4000th timestep, and both the balanced and non-balanced run in one AMD node each. That means we would be using 2 of the 9 AMD nodes in the Cluster (see table 2 for a list of the available resources on the Cluster), or 22% of the Cluster resources applicable to FEIXES, while both simulations produce the very same results (the only difference between them is having load balance enable or not!) and therefore, only one is needed. Considering the costs and the heavy use of the Cluster, this is simply not possible.

Hence, a very interesting subject to investigate is: If the balanced simulation took at least 3 weeks to achieve the 4000th timestep, how many times would the non-balanced simulation achieve the same timestep?

To answer this question, we first choose a range of 200 timesteps to evaluate on the first point. Then, we follow the same strategy employed by the ToT analysis (see sections 3.2.2 and 3.2.3), and calculate the average time spent on the timestep. We will also employ a memory, like the ToT, and divide the range into four groups, each with 50 timesteps, in which the mean will be calculated. The four groups are described as follows:

- 1st Group: timesteps 1 to 50
- 2nd Group: timesteps 50 to 100
- 3rd Group: timesteps 100 to 150
- 4th Group: timesteps 150 to 200

For each of these groups, we will calculate the average timestep to evaluate.

This is done first to resemble the analysis of the ToT. Second, to prevent any pollution caused by single work-intensive timesteps, like timesteps with output routines, which, for the FEIXES simulation, implies generating output files with 10 GB to 20 GB in size. In other words, some timesteps, in this case, have very slow I/O operations that can disturb our analysis. Third, since we will be extrapolating the non-balanced simulation behavior up to the 4000th timestep, we need to identify a pattern in this behavior compared to the balanced simulation. Otherwise, any extrapolation will have no hold in reality.

Now that the considerations are done, let's go to the data. Table 5 shows, for each simulation, the start time in minutes and the average timestep, also in minutes, for the four groups. Additionally, a separate row was included for each simulation, indicating how the timestep changes between the groups, which we can use to observe the pattern of the timestep: is it getting slower, faster, or remaining somewhat stable?

Version	Start Time	1st Group	2nd Group	3rd Group	4th Group
non balanced	10.00	6.61	7.75	11.69	11.69
Diff			1.14	3.94	0.00
balanced	4.96	4.47	5.20	7.93	8.22
Diff			0.73	2.73	0.29

Table 5 – Time spent on timestep in minutes for FEIXES at the start of the simulation and the average timestep for each analysis group, also in minutes. Diff rows show the change for the average timestep in each group.

By comparing the row Diff for each simulation in table 5, we can see that the behavior of the timesteps is pretty much the same for both simulations: it grows a little faster and then stabilizes. This is better illustrated by Figure 35 which shows the growth curve for the timesteps in each simulation.

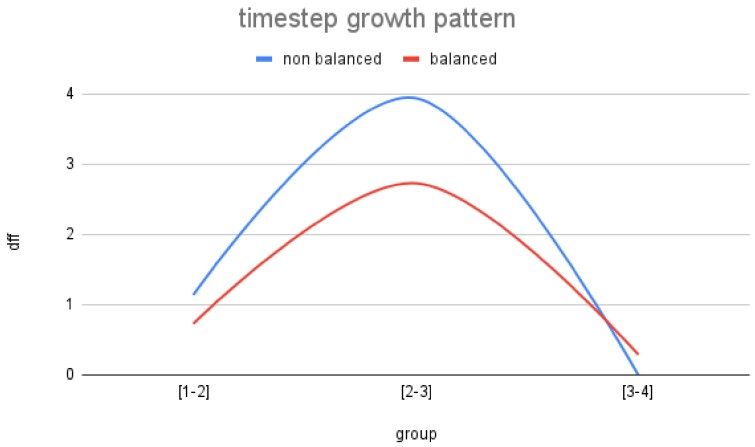


Figure 35 – Timestep growth pattern by comparing the difference between the analysis groups for each simulation

If the growth curves exhibit the pattern for the timestep, then advancing in time and taking another set of 200 timesteps to analyze, the mean timesteps are expected to remain close to the mean value of the last analysis group.

Let us do this by using the following set of 4 groups:

- 1st Group: timesteps 600 to 650
- 2nd Group: timesteps 650 to 700
- 3rd Group: timesteps 700 to 750
- 4th Group: timesteps 750 to 800

Version	1st Group	2nd Group	3rd Group	4th Group
non balanced	10.23	10.24	10.15	10.19
balanced	7.71	7.73	7.71	7.68

Table 6 – Average timestep for each of the new analysis groups

Then again, we present the average timestep for each group in Table 6 while Figure 36 shows the growth curve.

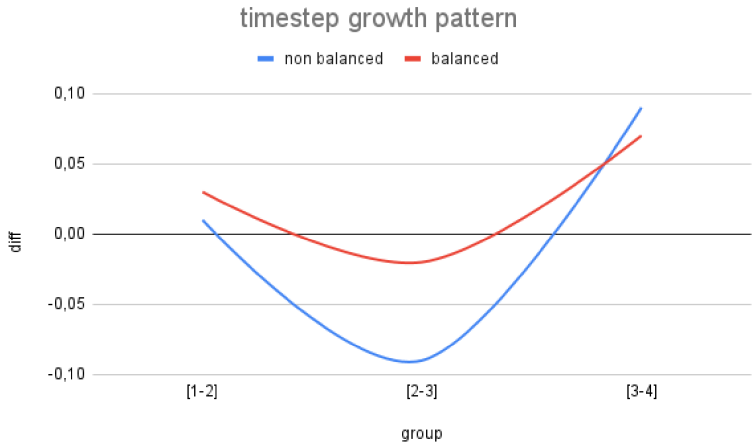


Figure 36 – Timestep growth pattern for the new analysis group

As expected, the values of the average timesteps for the new analysis groups, for each simulation, stay close to the values of the average timestep of the 4th group shown in table 5. Also, the growth curves shown in Figure 35 and 36 have the same shape, though they appear to be inverted to one another. This indicates the timesteps will increase and decrease throughout the simulation but will stay close to the reference values, which are 10 to 11 minutes in the non-balanced simulation, and 7 to 8 minutes in the balanced one.

We can at least confirm the reference value for the timestep of the balanced simulation, as this version ran for more than 4,000 timesteps (the non-balanced version was run for 821 timesteps before being stopped). We can again create groups after the 4000th timestep and check if the average timestep for each group is between the reference values.

As evidence, the following list presents a series of groups, each with 50 timesteps, along with the average timestep for each group after the 4000th timestep in the balanced simulation.

- 1st Group: timesteps 4000 to 4050 - Average: 7.73 minutes
- 2nd Group: timesteps 4050 to 4100 - Average: 7.69 minutes
- 3rd Group: timesteps 4100 to 4150 - Average: 7.75 minutes
- 4th Group: timesteps 4150 to 4200 - Average: 7.72 minutes

- 5th Group: timesteps 4200 to 4250 - Average: 7.85 minutes
- 6th Group: timesteps 4250 to 4300 - Average: 7.84 minutes
- 7th Group: timesteps 4250 to 4300 - Average: 7.64 minutes
- 8th Group: timesteps 4250 to 4300 - Average: 7.50 minutes
- 9th Group: timesteps 4250 to 4300 - Average: 7.51 minutes
- 10th Group: timesteps 4250 to 4300 - Average: 7.85 minutes
- 11th Group: timesteps 4250 to 4300 - Average: 7.77 minutes
- 12th Group: timesteps 4250 to 4300 - Average: 7.72 minutes

Now, we can calculate the overall average timestep for both non-balanced and balanced simulations and then extrapolate the non-balanced simulation to determine how much time will be required to run it to the 4000th timestep.

Considering the reference values for each, we have the following:

- non-balanced simulation: 10 to 11 minutes - Average: 10.5 minutes each timestep
- balanced simulation: 7 to 8 minutes - Average: 7.5 minutes each timestep

By multiplying the number of expected timesteps `N_t` with the average time for each timestesp `AVG_t`, we can measure how many minutes would be necessary to simulate to the desired point. This value can be divided into how many minutes we have in a day of 24 hours to calculate the required time in terms of days to achieve the point `Min_days`. That is what Equation 1 presents.

$$Min\_days = \frac{N\_t * AVG\_t}{24 * 60} \tag{1}$$

Applying Equation 1 to the data from the balanced simulation will result in a number close to 21 days, which is the time it would take for the simulation to achieve the 4000th timestep. In reality, it took 23 days to accomplish the timestep.

The difference is not entirely unexpected, as we explained before, there are some timesteps far slower than the others, like the ones generating output files, which for FEIXES means creating files of a few dozen gigabits throughout the whole simulation (all used cores). Also, there are the remesh timesteps (see section 3.2.2), which rearrange the mesh and also tend to be slower than "normal" timesteps, lastly, as the timestep growth curves presented in Figures 35 and 36, the timestep will sometimes get slower and sometimes get faster. That is primarily thanks to the simulated physics, which doesn't always have a clear pattern and can compute faster or slower depending on the current conditions of the timestep.

Overall, our prediction was off only for 2 days from which we can derive the error of our analysis in up to 10%.

Now, let us apply Equation 1 to the average timestep of the non-balanced simulation. This will produce a number close to 30 days for the simulation to achieve the 4000th timestep. Considering the same error from the prediction for the balanced simulation, we have 33 days. This is more than a whole month.

$$How\_faster = (1 - \frac{faster}{slower}) * 100 \quad (2)$$

By using the Equation 2 with the average timesteps for both simulations, we can obtain how faster (`How_faster`) the balanced simulation (`faster`) is, on average, than the non-balanced one (`slower`). The obtained value will be **28.6%**, which, considering the kind of simulation and all the limitations imposed by the mesh and solver (see section 3.1 for more details), is rather good.

We can verify this number by applying Equation 2 to the number of days required for each simulation to achieve the 4000th timestep, which will result in the balanced simulation being 30.3% faster than the non-balanced simulation. Again, it is not equal to the average value, but it is not far.

Considering one of the metrics presented in section 4.3: the smaller the time spent on the timestep, the better, and the data shown for FEIXES timesteps, we can conclude that the proposition indeed resulted in a smaller time spent in the timestep, with a average performance gain of 28.6%.

It remains to evaluate the load-generating object distribution now. This is a more straightforward evaluation, as we need only to count FEIXES load-generating objects, primarily the mesh control volumes, and the immersed boundary points, for each of the used cores, and plot that information into a chart.

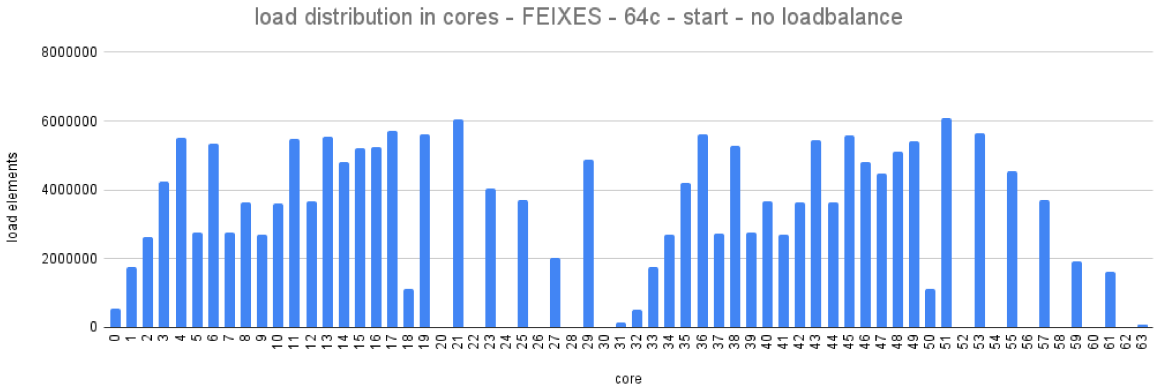


Figure 37 – FEIXES - load generating objects distribution - the start of the non-balanced simulation

Figures 37 and 38 present the load-generating object distribution for the non-balanced and balanced simulations at the start of both simulations. We can see the objects are

better distributed in Figure 38 than in Figure 37, especially when comparing cores 20, 22, 24, 28, 30, 31, 32, 52, 54, 56, 58, 60, 61, 62, 53 which had more load assigned to, in the balanced simulation, while cores 12 and 50 had less load assigned. That means, while the load balancing did reduce the load in 2 of the 64 cores (3%), in comparison with the non-balanced simulation, it increased load in 15 of the 64 cores (23%) who has been sub-used in the non-balanced simulation.

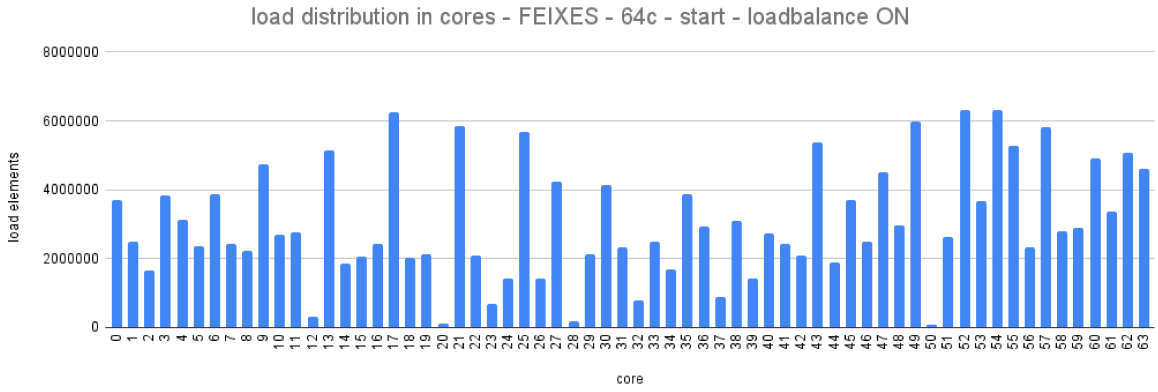


Figure 38 – FEIXES - load generating objects distribution - the start of the balanced simulation

So, by visually analyzing Figures 37 and 38, we can conclude that at least 26% of the cores had the load redistributed, which is rather good, especially when considering the size of FEIXES.

However, we can improve the analysis by comparing the load distribution of each simulation directly and again, plotting the comparison into a chart for better visualization. This is done by Figure 39, which plots side by side, core by core, the load distribution of each simulation.

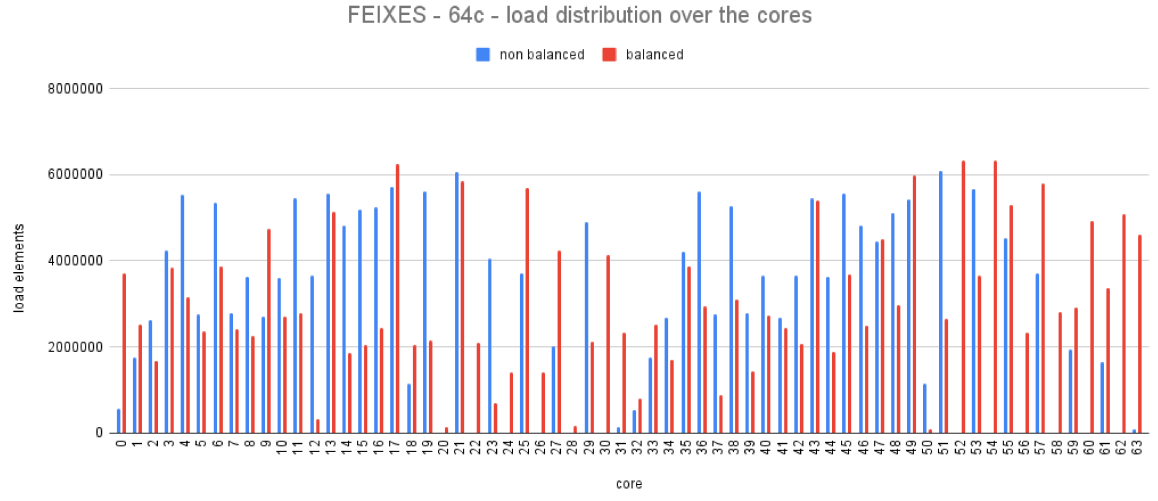


Figure 39 – FEIXES - load distribution over the cores - non-balanced and balanced simulation

Now, it is clearer that the load was more distributed in the balanced simulation, considering how many red columns are taller than their blue counterparts in Figure 39. This is expected again, as the basic idea behind load balancing is to redistribute the tasks over the cores (as described in section 2.2), which will make some cores, who, in a non-balanced simulation, had less load, will be assigned more tasks after a load balancing.

But, if more load has been assigned to some cores and we are not adding a lot more tasks by load balancing (the balancing is a task in itself, so the balanced simulation will have more tasks to do than the non-balanced simulation anyway), the load must come from some of the heavily loaded cores. Figure 39 shows this, as cores 2, 3, 10, 12, 35, 36, and many others have the blue column taller than the red one. Even so, the information in the Figure 39 chart is condensed and can lead to misinterpretations.

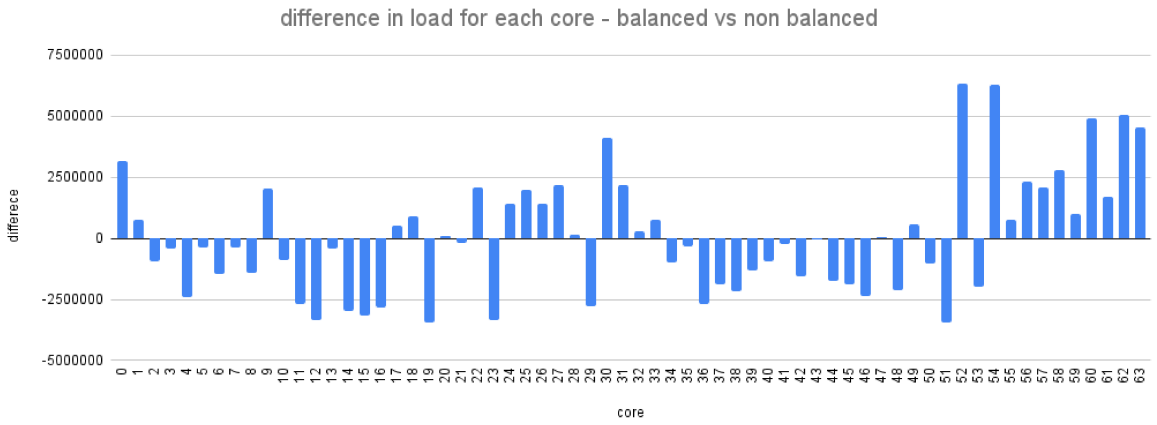


Figure 40 – FEIXES - load distribution difference between balanced and non-balanced simulation, core by core

To prevent that, let us consider Figure 40, which presents a new chart showing the difference in load-generating elements between the balanced and non-balanced simulation, grouped core by core.

We can now see that 35 of the 64 cores experienced a load reduction in the balanced simulation, while 29 had an increase. That means about 55% of the cores had their load redistributed for the other 45% of the used cores, therefore verifying that the load balancing operation indeed created a better distribution of the load generating objects, which satisfies the metrics we are using here.

4.4.2 Component Analysis

In the last section, we've made a macro analysis of the proposition's impact on the largest and slowest simulation we tested. In this section, we will conduct a more detailed analysis of the proposition, evaluating the behavior of its components and their influence on load balancing and, by extension, the simulations.

4.4.2.1 ToT

Starting with the main component of our proposition, the ToT (see section 3.2.2 for more details), we are going to analyze the ToT behavior in the industrial case SPRAY.

This case is interesting for this analysis as it starts relatively small and fast, with the timestep taking less than a second in the Cluster. However, as the simulation progresses, more and more loads are created, with the timestep taking increasingly longer to complete.

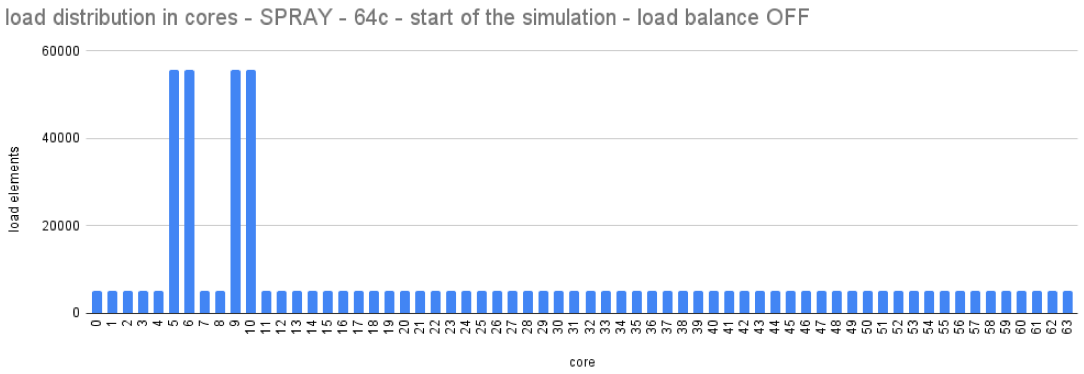


Figure 41 – SPRAY: load distribution over the 64 cores at the start of the simulation

Figure 41 shows how the load elements are distributed over the cores at the simulation start. It is visually unbalanced, with 4 of the 64 used cores concentrating the majority of the load. Without load balance enabled, this will continue for thousands of timesteps, as illustrated by Figure 42, which shows the load distribution at the 1000th timestep, which resembles the same distribution at the start of the simulation, even when 50% more load was added.

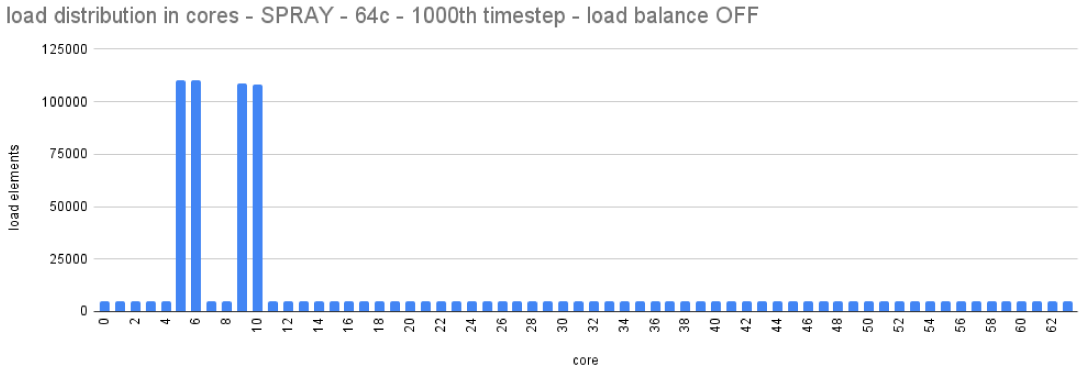


Figure 42 – SPRAY: load distribution over the 64 core at the 1000th timestep

This could indicate a prime candidate for load balancing, if not for the time consumed by the timestep (not the ToT, the whole timestep, including the synchronizations) been so smaller, less than a second, and, the fact that enabling load balance at the start of the simulation, and for at least the first 10 thousand timesteps, will generate one of the problems we discussed in section 3.1.2, the zeroed cores.

That means that while SPRAY is growing and adding more load constantly, there are still enough resources in the simulation to consider a visually unbalanced simulation, as balanced enough, since by trying to redistribute the load, at this time, we may end up creating a highly undesirable situation with the potential of crashing it.

A more straightforward conclusion is that we are over-parallelizing the simulation. If we reduce the number of cores used, it would be possible to use load balancing and properly redistribute the load without crashing the simulation. This conclusion, though not entirely wrong, fails to consider that the simulation will keep adding more load until there are not enough resources, and even the load balancing will lose its effect.

In fact, by examining the number of load-generating objects in the simulation at the 20,000th timestep, we have at least 42 million objects. In other words, though SPRAY starts with nearly 500 thousand load elements, it grows its load by a magnitude of at least 80 times. In that scenario, fewer than 64 cores will significantly impact the simulation performance.

It is valid to point out that this scenario may indicate an unsolvable problem, as if the simulation can continue to add more and more weight, it will be challenging (if not impossible) to determine how many cores it would need. That is why MFSim has tools to deal with such situations by allowing the change of the number of cores available to a simulation (FREITAS et al., 2023), but this is not the case with SPRAY.

This simulation requires thousands of timesteps just to initialize all the physics, and, while doing that, keep adding more and more elements. After that, which is somewhat close to the 20,000th timestep, the simulation slows down the load increase, and maintains nearly 42 million load-generating elements until its end. That is why we will use the 20,000th timestep to evaluate how the ToT influences load balancing.

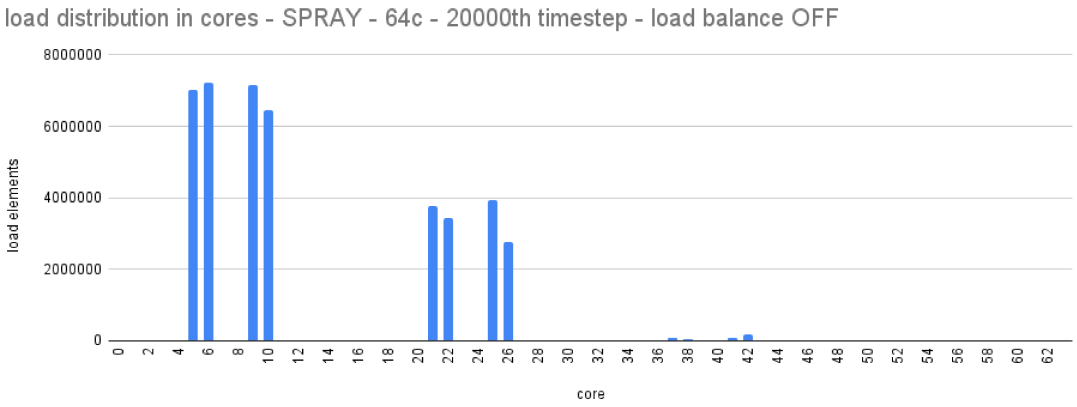


Figure 43 – SPRAY: load distribution at the 20000th timestep - without load balance

To do that, let's start by analyzing how the load is distributed at this timestep, which can be viewed in Figure 43. The imbalance is still present, as expected, but now we have more heavily loaded cores than we had at the start, or at the 1000th timestep. This unbalance doesn't pass unnoticed by the ToT (which stores the time spent with

computations, ignoring the synchronizations during a timestep) as shown in Figure 44, which plots both the fastest core, bearing the minimum ToT, and the slowest core, with the maximum ToT, for dozens of timesteps after the 20000th.

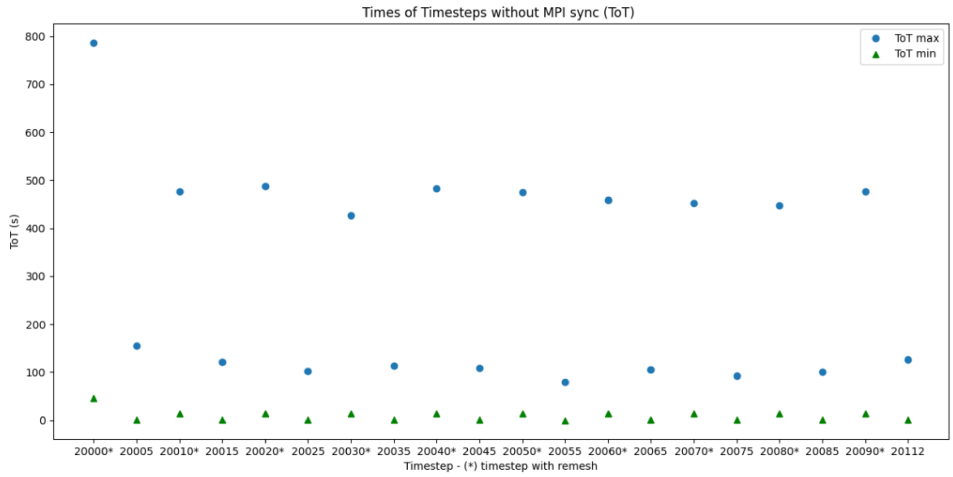


Figure 44 – SPRAY: fastest and slowest ToT after the 20000th timestep without load balance

The conclusion is that the simulation is unbalanced, particularly when a remesh is executed, significantly degrading overall performance. This is further corroborated by Figure 45, which plots how much time the fastest and slowest cores spend waiting for communication. We can see that the quickest core, i.e., the one represented by the green triangle, spent nearly all of its time waiting for communication rather than performing computations.

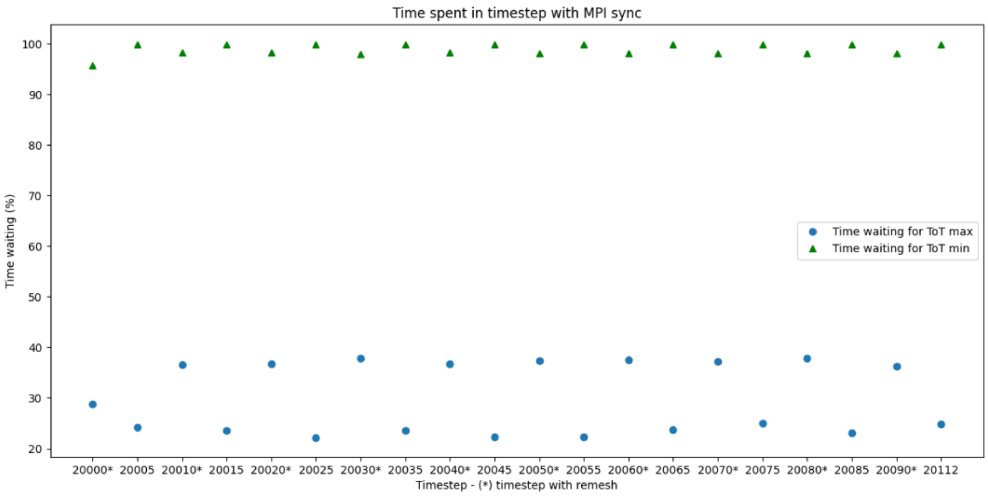


Figure 45 – SPRAY: time proportionally spent waiting for the fastest and slowest ToT after the 20000th timestep without load balance

The load balancing operation, then, should be able to improve the load distribution over the cores in such a way that the difference between the fastest and slowest ToT is

reduced and the time the quickest and slowest core spent waiting is reduced or at least approximated.

Also, we should be able to enable the load balancing at the 20000th timestep, especially when considering the distance between the fastest and slowest core shown in Figure 44. To evaluate the ToT, we are going to use the ToT analysis policy with a threshold of 5 for the non-remesh timesteps (normal timesteps) and 10 for the remesh timesteps.

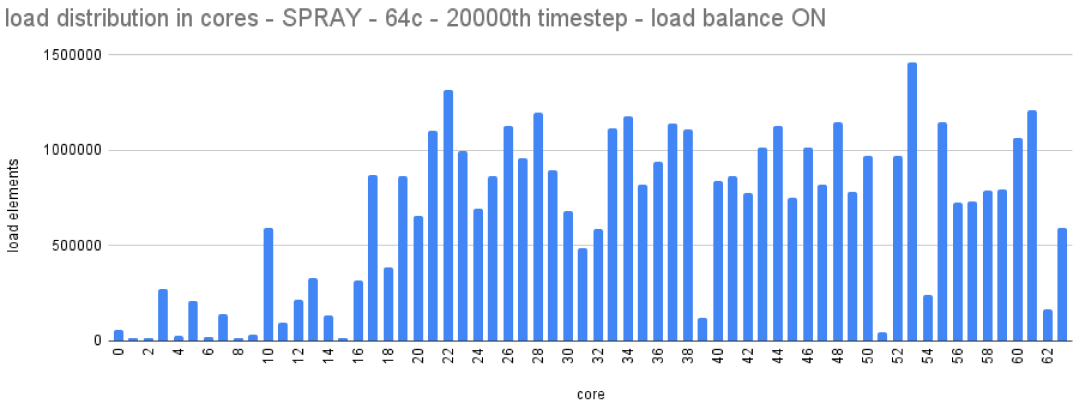


Figure 46 – SPRAY: load distribution at the 20000th timestep - load balance enabled by ToT analysis

As we can see in Figure 46, after the load balancing operation in the 20000th timestep, the majority of the cores now have more load, and this load is more evenly distributed, though some cores still have far less load than others. Even so, the balanced version improves the load distribution significantly, which can be perceived by plotting, side by side, the load distribution of the non-balanced and balanced 20000th timestep, as is in Figure 47. Note the red columns, representing the balanced load distribution, are more even in comparison to the blue columns, which represent the non-balanced distribution.

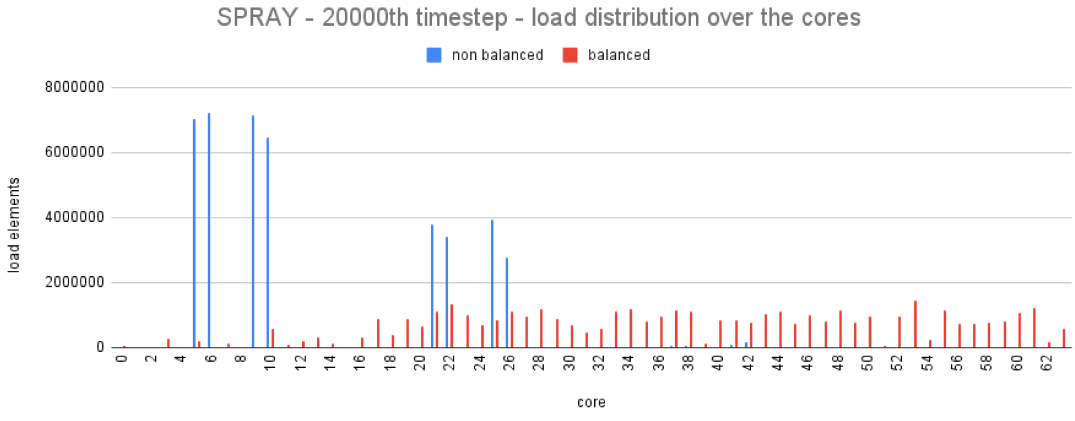


Figure 47 – SPRAY: load distribution at the 20000th timestep - non balanced and balanced versions

This can be further confirmed by plotting the difference in load for each core after the load balancing, where we can see the load was indeed moved from the heavily loaded cores to the less loaded ones, as shown in Figure 48.

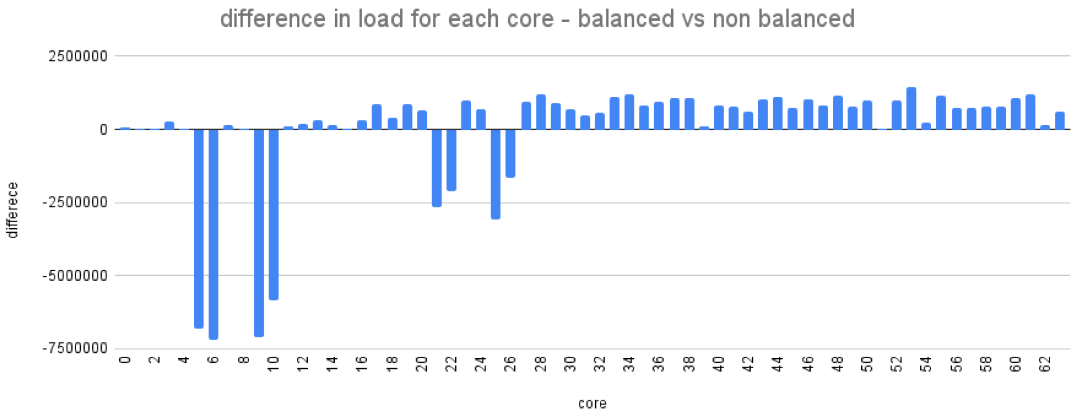


Figure 48 – SPRAY: load remap for each core between the non-balanced and balanced versions

As for the ToT, we expect that the load balancing reduced the difference between the fastest and slowest cores/ToT, and approximated the waiting times, which is precisely what happens as shown by Figures 49 and 50, respectively.

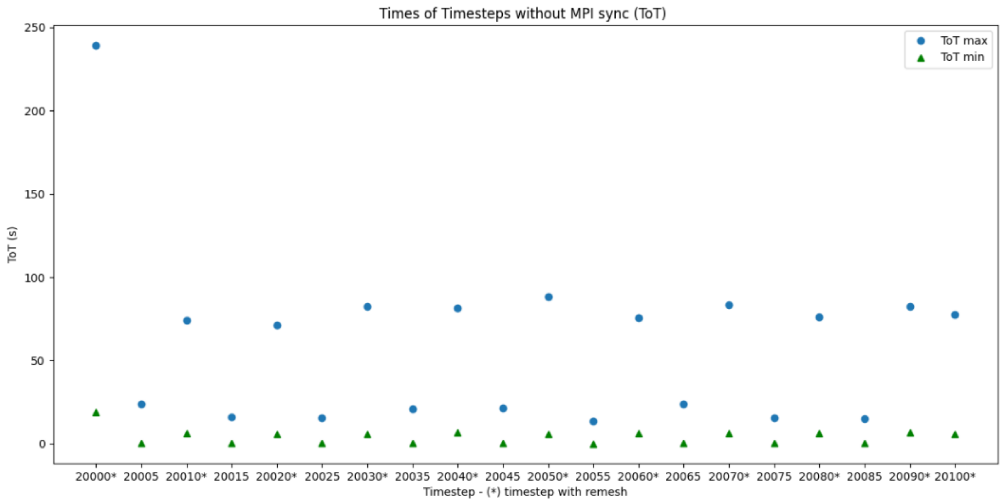


Figure 49 – SPRAY: fastest and slowest ToT after the 20000th timestep with load balance

With the load balancing, except for the 20000th timestep, which had a significant difference between the smaller ToT (the fastest core) and the largest ToT (the slowest core), all other timesteps had differences below the designed threshold. That is true for both normal timesteps and remesh timesteps, which is very interesting since remesh timesteps are often slower and tend to amplify any imbalance in the simulation. Additionally, this suggests that only one load-balancing operation was required to meet the specified threshold.

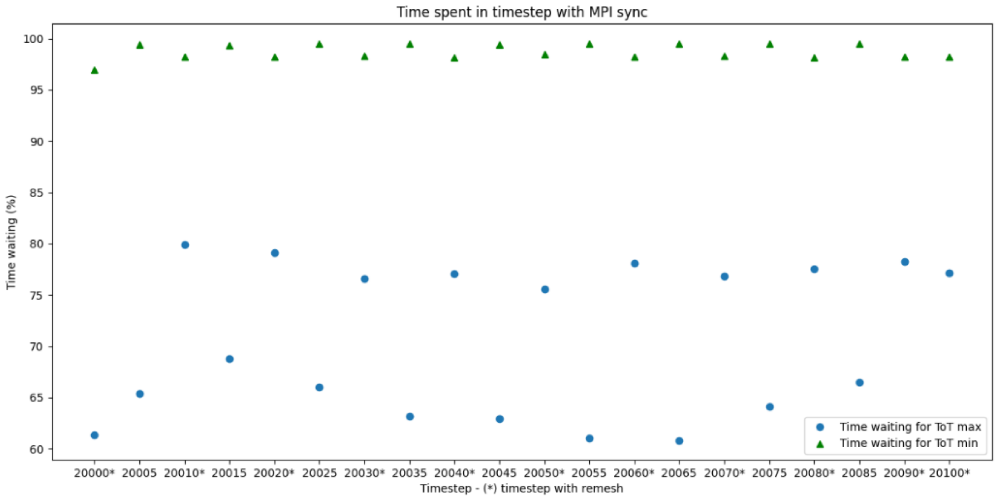


Figure 50 – SPRAY: time proportionally spent waiting for the fastest and slowest ToT after the 20000th timestep with load balance

Another important information is that the proportion of time spent waiting for both the slowest and fastest cores was indeed approximated. This shows the quickest and slowest cores, although they have different values of ToT, one doesn't need to wait too long for the other to achieve a synchronization point. This clearly indicates that the slowest core does not do more tasks than the fastest core, which is precisely what we expect from a more balanced simulation.

So, for the SPRAY, the ToT served as a trigger for the load balance when the simulation was already initialized and was again used to validate the load balancing, by showing that the load redistribution, though not ideal, indeed implicated in an overall reduction time spent with both computations and communications for the fastest and slowest cores, and therefore, for all cores.

To add final information to this analysis, and considering the metrics defined in section 4.3, the time spent in the timestep, again, not the ToT, but the whole time spent in a timestep, was reduced in 64% for a normal timestep and in nearly 50% for a remesh timesteps (see Table 7) which is outstanding result and satisfy our metrics fully.

Version	Normal Timestep	Remesh Timestep
non-balanced	2 min.	12.5 min.
balanced	43 sec.	6 min.
reduction	64%	50%

Table 7 – Time spent in timestep for SPRAY at the 20000th timestep

Though the result of load balancing for SPRAY is good, considering the details of this simulation, the time needed to initialize it, make the load balancing effective, and necessary. The limitations we faced (see section 3.1), we needed to run SPRAY without

load balancing for dozens of timesteps, to analyze the ToT and then establish the threshold for the ToT analysis. It worked, as we showed in this section. It did produce good results.

However, it appears that we parted from the desired results to the test configuration, rather than the other way around. This is partially true, of course, and considering the time and necessary resources to run an industrial-scale simulation, like SPRAY, FEIXES, or CALDEIRA, it does make sense to run the simulation for some time to test a few configurations and adjustments before letting the simulation run for weeks or months on end. We could call this calibration, and if the user calibrates the ToT analysis sufficiently, it will likely produce good results.

Yet, it feels we need a deeper analysis of the proposition to understand how it impacts the load balancing of multi-physics simulations.

That is why, in the next section, we are going to use an academic test case that runs fast (for a multi-physics simulation, at least) to analyze other components of the proposition. Still, this time, we'll conduct a statistical analysis to determine how the components and different calibrations may interfere with the load-balancing operation.

4.4.2.2 Threshold, ToT memory and LB policies

The simulation we chose for this analysis is SPHERE. It enables the basic core of physics of all the other simulations, the Fluid and Turbulence (see Table 4 for more details). It runs for a few minutes, which is essential for the multiple runs required by the statistical analysis. It doesn't have badpoints or zeroed cores problems, so we can use balancing in any way we want to test. It can be run on all available hardware (see Table 2), especially the Ryzen and Workstation models, which have high availability.

Since threshold and ToT memory are the components some load-balancing policies use, their evaluation is then linked to the policies that use them. Hence, we organize the evaluations by policy, which are described in section 3.2.4 and can be summarized in:

- ❏ balance at the start of the simulation
- ❏ balance for every remesh
- ❏ balance by threshold
  - for every remesh until a specific one
  - for every remesh after a specific one
  - balance until a certain timestep
  - balance after a certain timestep
  - balance by average ToT difference of the highest and smaller ToT
- ❏ balance by ToT analysis

❏ automatic

The first two and the last two policies had only one group of runs. In contrast, threshold policies involved multiple runs, as the main idea is to evaluate how different calibrations of these thresholds can influence load balancing. Each group run had 30 executions. All executions were done in the Workstation, with roughly the same system load for all of them. The only tasks we dispatched were the runs, and nothing else, to minimize external load influences. We also conducted a series of runs with 30 trials, using the non-balanced SPHERE as a control group.

Each run of SPHERE lasts more than 2,000 iterations, with 21 total remeshes. The number of iterations and remeshes are the same for balanced and non-balanced runs. Each group run executes the SPHERE 30 times. We run 2 different balance configurations by remesh threshold, 6 for balance by iteration threshold, and 4 for ToT difference threshold. As for ToT analysis, we vary ToT difference, ToT and tendency memories, totalizing 12 different configurations. Additionally, there are three additional group runs one for the non-balanced, one for the balance at the start of the simulation, and one for the balance in all remeshes. Table 8 gives a summary of all the different configurations (or calibrations) used.

Policy	Threshold	ToT mem	Tendency mem
no balance			
balance at start			
balance at every remesh			
balance until remesh	10th		
balance after remesh	10th		
balance until timestep	500th, 1000th, 1500th		
balance after timestep	500th, 1000th, 1500th		
balance if ToT diff is above	2x, 3x, 4x, 5x	50	
balance by ToT analysis	2x, 3x, 4x, 5x	10	10
balance by ToT analysis	2x, 3x, 4x, 5x	50	30
balance by ToT analysis	2x, 3x, 4x, 5x	100	80
automatic	5x	50	30

Table 8 – Group runs of SPHERE, with the respective policies, thresholds, and memory configurations

Since each group run has 30 executions and we used 27 group runs, we ran SPHERE 810 times. That's a lot of runs! As we have many, we are showing the evaluation for the load distribution, as part of the defined metrics (see Section 4.3), only for a select few of them. This is done to prevent showing dozens of plots that have roughly the same shape and add very little to the analysis.

We also focus on a variation of the second metric, which evaluates the time spent on the timestep. Again, we have 810 runs in total, with multiple ToT's for each run (at

least 2 thousand for each, multiplied by the 4 cores used, which give us something like 6 million ToT's) and, even if we evaluate the ToT averages for each run, it will simply generate a lot of information and a repeated one. However, since any gain in a timestep will influence the overall time spent in the whole simulation, we will consider this time. So, instead of analyzing the thousands of ToT's for each of the 30 runs in a group (and +6 million in total), we are analyzing the final time spent in the whole simulation, for each run of each group.

Then, we calculate the average and standard deviation for the whole group run and analyze the average using a T-Student distribution (TURCIOS, 2015) (HAYES, 2025), with 95% confidence level to then and calculate the lower and upper limits for the time spent in the simulation, for each group runs. It is those limits that we plot to analyze how the different calibrations tested (as described in Table 8) influence the load balancing.

Before we proceed to the results, let us justify why we conducted each test at each point (for those using a threshold, at least). The non-balanced group runs, as explained before, serves as a control group. The balance at the start and balance at every remesh doesn't require any changes to ToT difference threshold, ToT and tendency memories. The automatic uses the default values for those configurations, which is ToT diff of 5 times, ToT memory sized in 50 elements, and tendency memory with 30 elements. Since the automatic run combines all other possible configurations (except load balancing in every remesh), their results likely resemble a mix of the ToT diff and ToT analysis.

Now, the group runs with a threshold. Starting with the remesh, SPHERE has 21 remeshes throughout the whole simulation. We chose the 10th remesh because it is very close to half of the simulation. Therefore, we aim to assess the impact of using load balancing for approximately half of the simulation. The group runs with a timestep threshold takes the same approach. SPHERE runs for more than 2 thousand timesteps (actually, 2040 timesteps), and we want to see the influence in load balance for roughly every quarter of the simulation. Hence, we use three different timesteps, dividing the simulation into four parts.

The ToT difference threshold tests only the average difference between the fastest and slowest ToT in ToT memory, and compares it with the specified threshold (see section 3.2.4 for more details). We tested thresholds of 2, 3, 4, and 5 to determine if there is any difference in load balancing for SPHERE by having the fastest core 2, 3, 4, or 5 times faster than the slowest core. Since a load unbalance situation captured by a threshold of 5 times is also captured by a threshold of 4 times (or smaller), we are testing whether reducing the load balancing tolerance, which this threshold describes, produces an improvement in simulation time.

ToT analysis also does that, as it uses ToT difference threshold (tolerance) as well. However, it introduces the tendency analysis (see Section 3.2.3 for more details), which serves as a trigger for determining when load balancing will be used or not. That is why

we are changing the size of the memories, together with the ToT difference threshold, to see how these changes influence the tendency analysis and, therefore, when to use load balancing or not.

Now, the results.

Starting with the load distribution, as part of the metrics, Figure 51 plots the load distribution at the start of SPHERE of a non-balanced run and a balanced one. We can see that load balancing does make a better distribution of the load, but not significantly, with the first two cores still being more heavily loaded, even when load balancing is used.

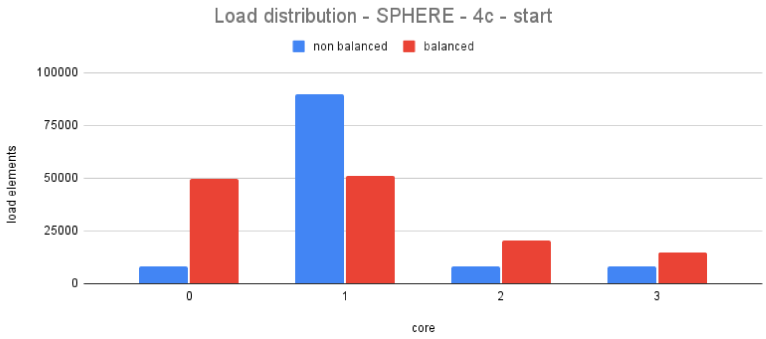


Figure 51 – Load distribution balanced and non-balanced, at the start of SPHERE

The same pattern can be observed after running nearly half of the simulation, as Figure 52 presents. Even when doubling the load elements, the first two cores are still more loaded, even when using load balancing. Part of this is related to the badpoints, which prevent the load balancing algorithm from redistributing more load, and part is related to the number of used cores.

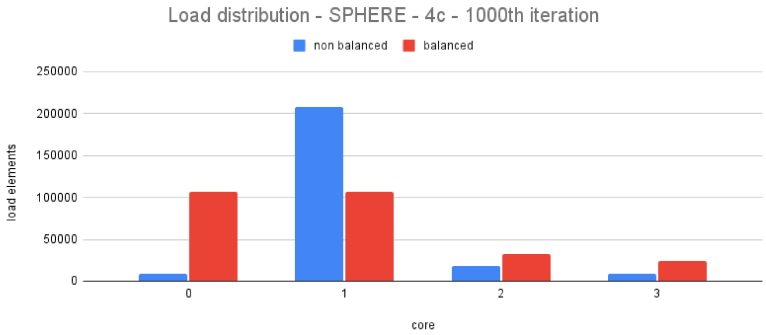


Figure 52 – Load distribution balanced and non balanced, at the 1000th iteration of SPHERE (half of the simulation)

To prove this, we conduct two additional runs to see how the load is distributed over eight cores instead of 4. The results are presented in Figure 53, and we can see that, by adding more cores to SPHERE, we can achieve a better load distribution, both at the start, with more loaded cores and at the 1000th timestep, with the load migrating from cores, thanks to the adaptive mesh refinement.

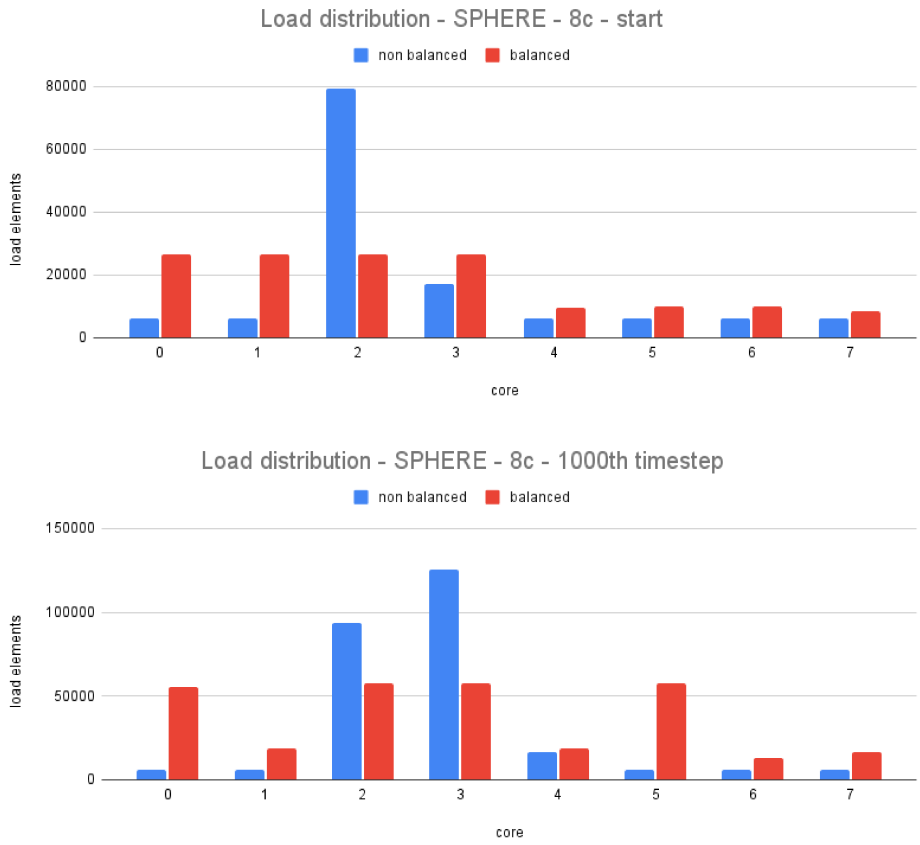


Figure 53 – Load distribution balanced and non-balanced, SPHERE with eight cores, start and at the 1000th timestep

Since our main objective in this section, as we stated before, is to analyze how the different settings of thresholds, memories, and policies influence the load balancing, we will consider this load distribution results as satisfactory and proceed with the rest of the analysis.

In that regard, Table 9 shows us the average time, plus the confidence interval, to run SPHERE under each one of the tested configurations. Both the times and intervals are presented in minutes, with 5-digit precision, whenever possible.

Considering the number of tests we ran, the table is quite large. That is why we decided to plot the minimum and maximum times, calculated from the averages and confidence intervals, for each test and sort them from the slowest to the fastest run to graphically evaluate how the configurations affected load balancing. This is what Figure 54 shows.

Version	Time (minutes)
non balanced	17.75643 ± 0.023670
lb: start	11.44975 ± 0.02009
lb: all remeshes	10.78856 ± 0.01250
lb: automatic	11.39658 ± 0.014705
lb: until 10th remesh	10.8078 ± 0.015673
lb: after 10th remesh	14.04893 ± 0.01722
lb: until 500th iteration	10.79631 ± 0.01320
lb: until 1000th iteration	10.78905 ± 0.01268
lb: until 1500th iteration	10.78285 ± 0.01494
lb: after 500th it	12.17004 ± 0.02116
lb: after 1000th it	14.05879 ± 0.01932
lb: after 1500th it	15.91252 ± 0.02394
lb: ToT diff 2	11.0925 ± 0.01401
lb: ToT diff 3	11.21428 ± 0.01170
lb: ToT diff 4	11.46775 ± 0.025765
lb: ToT diff 5	11.44887 ± 0.01916
lb: ToT th 2 mem 50 tend 30	11.12580 ± 0.01555
lb: ToT th 2 mem 10 tend 10	11.08987 ± 0.01483
lb: ToT th 2 mem 100 tend 80	11.09672 ± 0.01021
lb: ToT th 3 mem 50 tend 30	11.20112 ± 0.01390
lb: ToT th 3 mem 10 tend 10	11.22733 ± 0.01221
lb: ToT th 3 mem 100 tend 80	$11.20584 \pm 0,01327$
lb: ToT th 4 mem 50 tend 30	11.44264 ± 0.04280
lb: ToT th 4 mem 10 tend 10	11.47732 ± 0.026533
lb: ToT th 4 mem 100 tend 80	11.46194 ± 0.02204
lb: ToT th 5 mem 50 tend 30	11.483743 ± 0.01997
lb: ToT th 5 mem 10 tend 10	11.4733 ± 0.028884
lb: ToT th 5 mem 100 tend 80	11.48111 ± 0.02367

Table 9 – Time to run SPHERE under each configuration

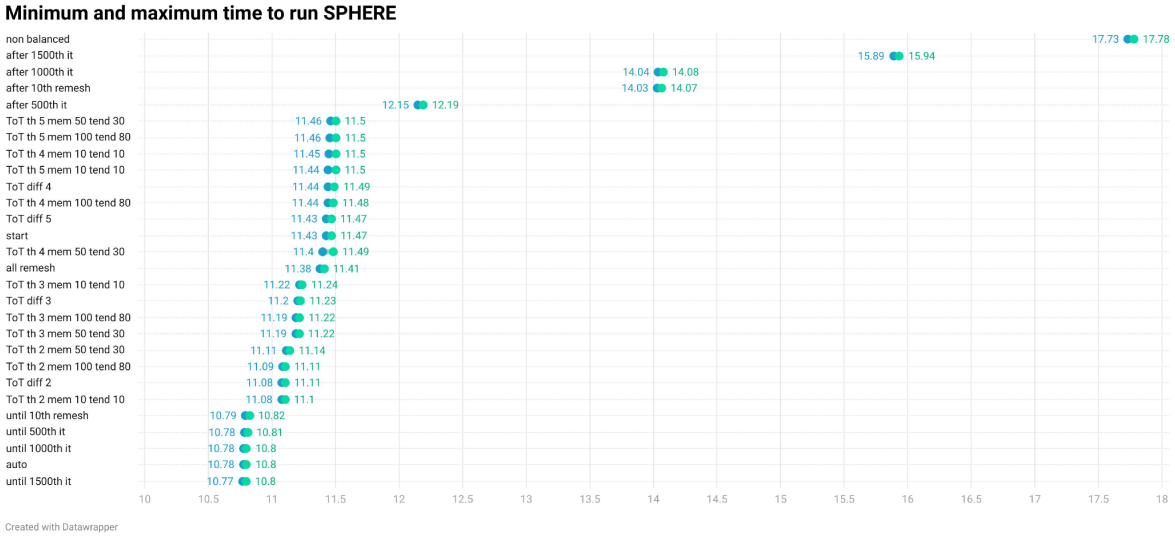


Figure 54 – Minimum and Maximum time to run SPHERE

As shown in Figure 54, the slowest run is the non-balanced, which is expected since load balancing works for SPHERE and improves performance. The following three slowest configurations enable load balancing in the last quarter or the latter half of the simulation. Interestingly enough, enabling the load balancing at the 10th remesh or the 1000th iteration produces practically the same result, even when the 10th remesh takes place nearly a hundred iterations before the 1000th iteration.

That indicates that the majority of the load is created in the first half of the simulation. Enabling load balancing after that, although it produces better results than no load balancing at all, doesn't improve performance as much as enabling load balancing earlier. This can be corroborated by the fact that allowing load balancing in the second quarter of the simulation ("after 500th it") improves the performance better than enabling it later.

Now, let's jump to the bottom of the graph. We can see that enabling load balance for 3 quarters of SPHERE produces the best performance, indicating that any load balancing taking place in the last quarter will not significantly improve performance. This can be verified by the fact that enabling the operation only in the first half of the first quarter of the simulation also produces good performance, akin to allowing it for three quarters automatically, and for the first 10 remeshes (also within the first half of the simulation).

Another interesting fact is that enabling load balancing for all remeshes produces a "middle ground" performance, indicating that for SPHERE, some well-placed load balancing operations are more effective. This is even more interesting because this was the only policy originally available in MFSim (see section 3.2.4 for more details). So, this policy produces a *middle ground* performance. In contrast, other policies enabled by this proposition produce better results, not only showing that we managed to keep at least the load-balancing capabilities previously available in the used code, but also expanded it, as presented by Figure 54.

As for the ToT difference configurations, who measures the tolerance for enabling or not the load balancing, the less tolerant configurations produce better results than the more complacent ones. This is true, even when combined with the tendency analysis. Furthermore, it is very interesting to see that the memory configurations work closely with the tolerance. If the tolerance is small, then smaller memories produce better results. But as the tolerance increases, more significant memories tend to produce better results, though there isn't a clear pattern for the size of the memories.

This can be explained by the fact that with larger tolerances, it may require more time to achieve that tolerance, and since the memories record the ToT's behavior through the timesteps, the larger the memory, the more probable to store a higher difference between the slowest and fastest ToT.

To conclude, we can observe that enabling load balancing at the start also yields a "middle ground" performance gain, and the automatic configurations, which combine the other policies, produce a result nearly as good as the best result. This is very, but very

interesting.

First, because two of the tested industrial cases in this work use this configuration (calibrating an industrial case is very hard and time-consuming), so, if our statistical analysis, with a 95% confidence level in a much simpler simulation, produces a result that is good for auto, then we can be more comfortable in using automatic load balancing first in an industrial case. If this doesn't produce a good result, then we go for calibrations. Second, the average CFD user is prone to use this configuration as default, without regard (or interest) to change or calibrate anything for the load balancing. This is not a problem, evidently, but is another good motive to consider a good automatic result as a very interesting one.

In conclusion, we successfully demonstrated that different calibrations can and will interfere with load balancing. As stated many times in this work, multi-physics simulations can vary significantly in terms of load-generating elements, discretization techniques, solvers, physics coupling, and numerous other factors. Having a load-balance configuration available out of the box, such as auto, is very helpful in increasing performance. However, depending on the simulation, it may be necessary to go to the lengths of calibrating the load-balancing components to achieve better results.

To finalize the results section, we now need to evaluate the physical influence on load balancing. Until now, we have evaluated the entire proposition without regard for its details. We then assessed the components of the load balancing and their impact on the simulation. In the next section, we will evaluate how physics impacts load balancing and how our proposition addresses this.

4.4.3 Physics influence

The last test case we present in this work is another industrial-scale simulation, the CALDEIRA. This case is not as heavy as SPRAY or FEIXES, but also not as light as SPHERE, which is suitable for multiple runs. Additionally, it can be simplified by disabling or reducing certain physics. This allows us to analyze how physics influences not only the simulation but also the load generation and, therefore, the load balancing operation.

This is the main objective of this section: to analyze the physics influence over the load balancing operation. For that purpose, we run three versions of this case, each with and without load balancing. The SIMPLE version is smaller and faster, with only the must-have physics enabled and a partially immersed boundary. FNR has complete immersed boundary physics but without chemical reaction physics. This is intentional, as detailed chemistry introduces load-generation elements outside the mesh, and this load is usually substantial. FR has the full CALDEIRA, with everything enabled.

The size and number of iterations (timesteps) for each run differ across the versions. Size is directly related to the simulation, and as the simulation incorporates more physics,

its size also increases. This influences the time spent on the timesteps (whole timestep, not the ToT) which affect how many iterations we run. The heavier the simulation, the more time required, so we established a week's worth of runs for each version. SIMPLE and FNR tend to stabilize the time spent on the timestep after a few hundred iterations and therefore, it was possible to run a few dozen thousand of them. But FR is more complicated. As soon as the chemical reaction starts occurring, the time spent on the timestep begins growing, and therefore, only a few thousand of them were possible to run in a week.

The primary reason for this week's threshold was hardware availability. Considering the memory and processing requirements of this simulation, only the Cluster and Node could be used for the runs, and both are heavily occupied. Node was less occupied than Cluster and, therefore, was reserved for a few weeks to run CALDEIRA specifically on it.

For easy access to the scope of each version, we summarized this information in Table 10, which can later be referenced while discussing the results

Version	Physics	Size	Iterations
SIMPLE	Fluid Turbulence Partial Immersed Boundary Multi-phase	+2 million	+160,000
FNR	Fluid Turbulence Full Immersed Boundary Multi-phase	+11 million	+16,000
FR	Fluid Turbulence Full Immersed Boundary Multi-phase Chemical Reaction	+16 million	+2,000

Table 10 – CALDEIRA versions

4.4.3.1 Influence over non-balanced runs

Starting the analysis, we can see by Figures 55, 56, and 60 how the load is distributed in each of the 32 cores used in CALDEIRA, for all versions, without load balancing, at the start of the simulation.

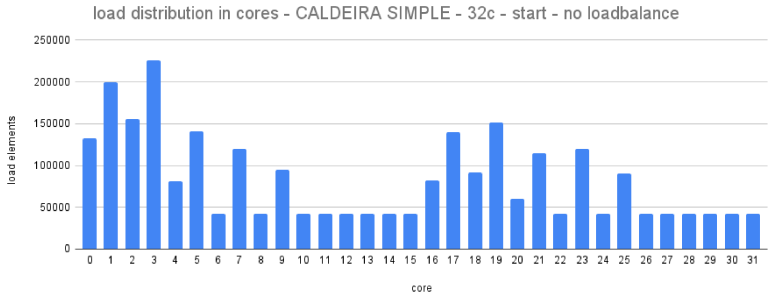


Figure 55 – Load distribution over the cores, CALDEIRA SIMPLE at the start, without load balance

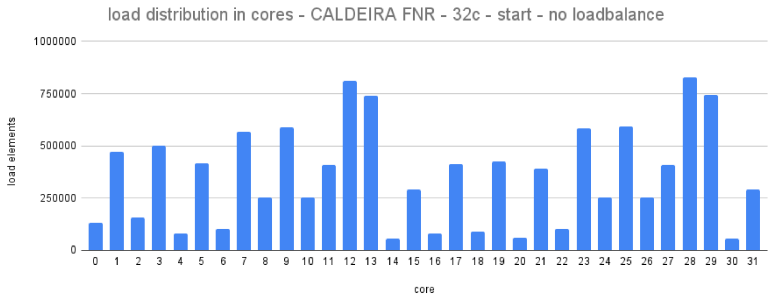


Figure 56 – Load distribution over the cores, CALDEIRA FNR at the start, without load balance

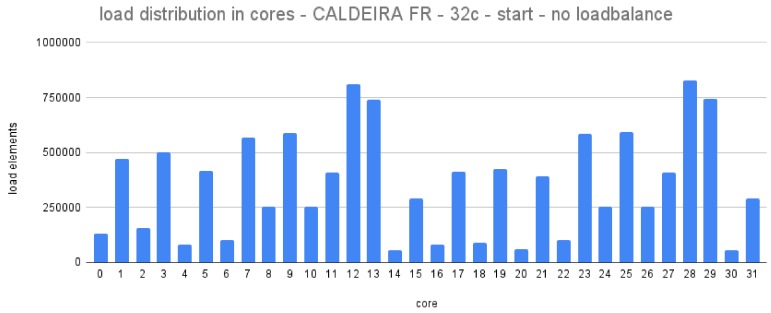


Figure 57 – Load distribution over the cores, CALDEIRA FR at the start, without load balance

Remembering that each version adds more load by enabling physics, the figures clearly show a difference in the distribution pattern between SIMPLE, FNR, and FR versions, with the last two having roughly the same pattern. In contrast, SIMPLE has a pattern different from the previous two.

This can be explained by the fact that FNR and FR only differ by having an extra-mesh load generator, provided by the chemical reaction physics, which is only present in the latter. In other words, when considering only the load generated by elements directly related to the domain, which is the main scope of the load balancing operation (see section 2.2 for more details), any load generator outside this scope, will be at least

partially ignored in a load distribution, even without load balancing. This is exactly what we see in Figures 56 and 60.

On the other hand, if the physics interferes directly with any element related to the domain, like mesh or immersed boundary points, which is the case in CALDEIRA, then the load distribution will tend to reflect this interference. This explains why SIMPLE has a significantly different pattern in its load distribution compared to FNR and FR, as it has fewer mesh and immersed boundary points than the other two.

This gives us an example of how the physics impact a simulation, independent of the load balancing, has been enabled or not.

Also, if we consider that some simulations can make use of adaptive mesh, and the mesh is one of the elements in the load balancing scope, and, depending on the case, it will be the physics directing the mesh adaptability, it is not incorrect to assume that the physics impact can be even more significant, as it will be them adding more load generating elements in the simulation. A simulation that has this kind of setting is SPRAY (see section 4.4.2.1), where the simulation starts with hundreds of thousands of load-generating elements and grows to millions of load-generating elements over time. Those millions of elements are added by physics-directed remesh.

But even for simulations with extra mesh load, like CALDEIRA, the physics impact can be severe, as we are going to present later here.

For now, we have started to visualize the physics influence over a non-balanced simulation. Let us proceed to the next section and examine how physics affects load balancing, our primary interest in this work.

4.4.3.2 Influence over balanced runs

Again, we will present the results of the load distribution for each of the CALDEIRA versions. But this time, with load balancing enabled, to see how this operation will behave as more physics are added to the simulation.

Figures 58, 59 and 60 show that for CALDEIRA SIMPLE, FNR and FR, respectively.

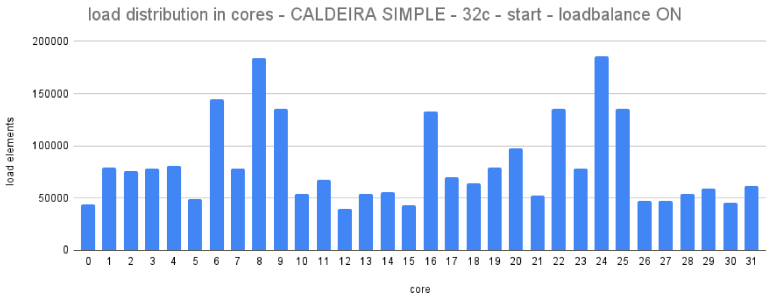


Figure 58 – Load distribution over the cores, CALDEIRA SIMPLE at the start, with load balance

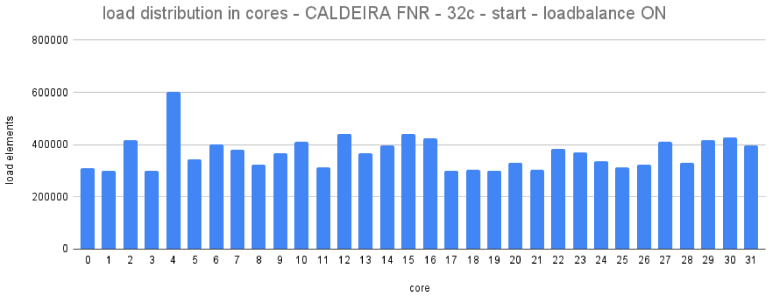


Figure 59 – Load distribution over the cores, CALDEIRA FNR at the start, with load balance

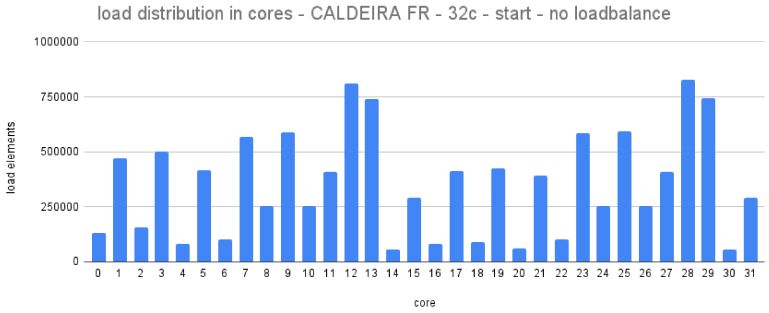


Figure 60 – Load distribution over the cores, CALDEIRA FR at the start, with load balance

And now things start to get interesting because this time, each version has a different pattern.

For SIMPLE, it looks like the load balancing doesn't make much of a difference, which can be corroborated by Figure 61, which puts side-by-side the load distribution for SIMPLE of the balanced and non-balanced simulations. By comparing the blue (non-balanced) and red (balanced) columns in those figures, we can be led to consider that the load balancing degraded the simulation load distributions instead of improving it.

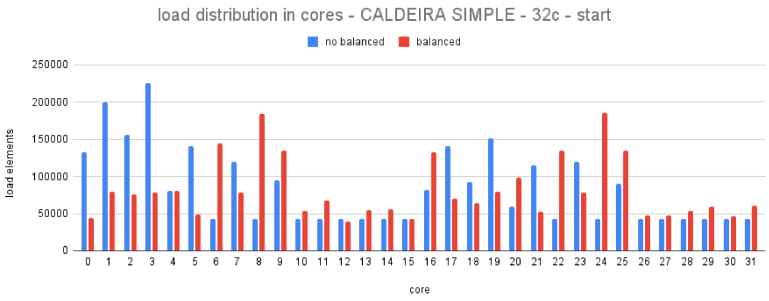


Figure 61 – Load distribution over the cores, CALDEIRA SIMPLE at the start, non-balanced vs balanced versions

We had this situation before, with SPRAY (see section 4.4.2.1), where the load balancing, at the start of the simulation, was ineffective (or not possible to use) because there

were far too many resources for far too few computations in the simulation. Though CALDEIRA SIMPLE doesn't suffer from badpoints or zeroed cores, as SPRAY does, the load balancing for this simulation, in this configuration, doesn't produce good results.

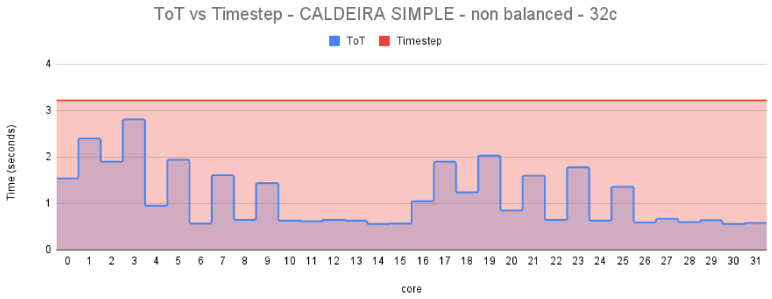


Figure 62 – ToT vs Timestep for each core, CALDEIRA SIMPLE, non-balanced

This can be further verified by comparing the ToT (time spent with computations) vs Timestep (time spent to do everything in the timestep) for the non-balanced and balanced versions of SIMPLE, as shown, respectively, by Figures 62 and 63.

Both of those Figures present a plot that overlays the ToT over the Timestep, and we can observe that, even with load balancing, the ToT varies wildly from core to core. In contrast, in a proper load-balancing operation, they would be somewhat closer.

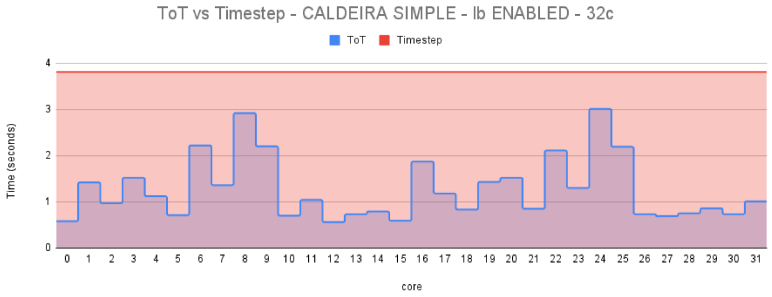


Figure 63 – ToT vs. Timestep for each core, CALDEIRA SIMPLE, balanced

Also, the timestep is smaller for the non-balanced version (Figure 62) than for the version with load balancing enabled. While in the first, the timestep is closer to 3 seconds, in the latter, it is closer to 4 seconds, indicating the load balancing for SIMPLE doesn't produce good results.

The main question is if this will change for FNR and FR. They have different load distribution patterns than SIMPLE (and each other) after using load balancing, and as we stated before, both versions have more physics enabled.

Continuing with the FNR and comparing the load distributions of the non-balanced and balanced simulations, side by side, as presented in Figure 64, we can see that for this version, the load was indeed better distributed, with some cores losing load while other cores gained load. This is reflected by the better ToT distribution over the cores for the

balanced version compared with the non-balanced version, as shown by Figures 65 and 66.

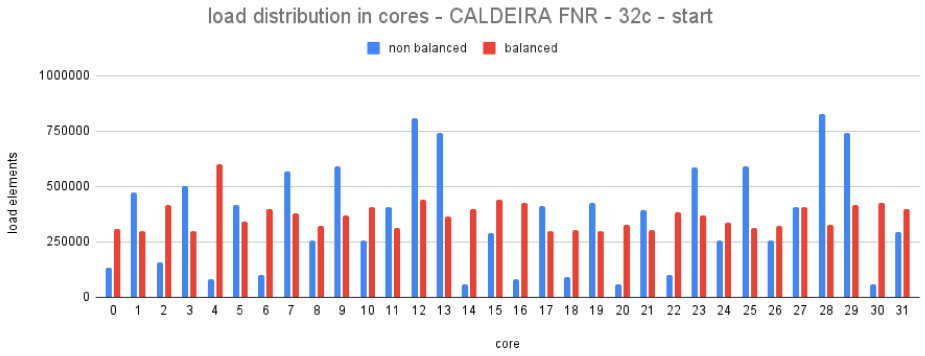


Figure 64 – Load distribution over the cores, CALDEIRA FNR at the start, non-balanced vs balanced versions

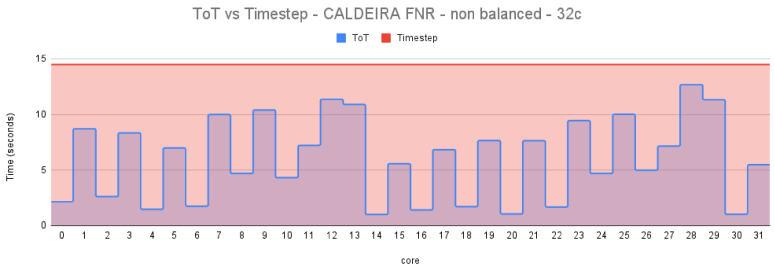


Figure 65 – ToT vs Timestep for each core, CALDEIRA FNR, non-balanced

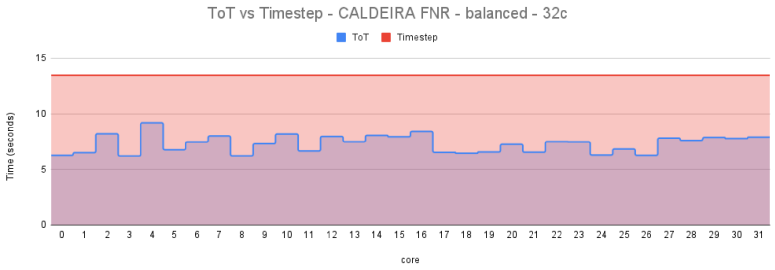


Figure 66 – ToT vs Timestep for each core, CALDEIRA FNR, balanced

We can see as well that in Figure 66, the time spent with computations (ToT) is very close for each core, indicating that most of the cores have roughly the same amount of tasks to perform, which is the expected behavior in a balanced simulation. This contrasts with the more uneven ToT of the non-balanced simulation, as presented by Figure 65, while some cores are doing a lot more work than others.

Also, it is possible to note the timestep is reduced in the balanced simulation in comparison with the non-balanced. However, the reduction is not significant, as the timestep in both versions is somewhat close to 15 seconds.

A good explanation for this is that adding more physics, especially those with retro-feeding mechanisms (like immersed boundary), will require more work anyway, which may make it more challenging to achieve a better timestep reduction.

Also, as explained in sections 2.2, 3.1.2 and 4.3, our proposal has limits inherent to the type of mesh, solver, discretization techniques, and physics coupling, to the point that we work with the concept of balanced enough simulation instead of the ideal balancing. In addition, as all simulations are iterative, any gain from a single timestep has the potential to influence the overall simulation positively, and therefore, it is a win.

This can be further elaborated by analyzing how the timestep changes throughout the simulation. Since CALDEIRA FNR was run for at least 16 thousand iterations, with load balancing enabled and disabled, we can compare both runs to see if there is any gain in the long run, resulting from small gains in each timestep.

For this evaluation, we will compare the time spent in each timestep at the start of the simulation and at the 16800th timestep, for both runs. Table 11 shows this, including the gain in each timestep for using load balancing.

Version	Start Timestep	16800th Timestep
non balanced	14.5 sec.	18 sec.
balanced	13.5 sec.	16 sec.
reduction	7%	12%

Table 11 – Time spent in timestep for CALDEIRA FNR at the start and 16800th timesteps

Now, we are going to change equation 1 presented in section 4.4.1 for the reality of CALDEIRA (timesteps in seconds instead of minutes) to calculate how many days we would, at least, need to achieve a certain timesteps with and without load balancing.

Again, we consider the number of expected timesteps as `N_t`, the average time for each timestep as `AVG_t`, and the minimum days as `Min_days`. The average timesteps for each version we are calculating from the data are shown in Table 11 and, again, we are ignoring timesteps with remesh or I/O operations, which we know to be far slower than timesteps without those operations. All of this is presented in Equation 3

$$Min\_days = \frac{N\_t * AVG\_t}{24 * 60 * 60} \tag{3}$$

Now, applying Equation 3 to the averages 16.25 seconds for non-balanced and 14.75 seconds for the balanced run, and the desired timestep, we have, at least 3 days to achieve the 16800th timestep in the non-balanced simulation and at least 2 days to accomplish the same point in the balanced simulation. The difference appears to be small, but if we keep increasing the desired timestep, even when fixing the value for the average timestep (which is more likely to increase over time), the difference gets more significant, proving

what we claim that small gains in balance, over time, can reduce the overall time spent in the simulation. This is what Table 12 shows.

Version	Average Timestep	16800th T.	168000th T.	1680000th T.
non balanced	16.25 sec.	3 days	31 days	315 days
balanced	14.75 sec.	2 days	28 days	286 days

Table 12 – Simulated time for CALDEIRA FNR to achieve some timestep (T.) checkpoints

Continuing the analysis, we have shown so far that adding more physics increases the load, and the load balancing also tends to degrade. To further investigate this, we are going to analyze now the data regarding the CALDEIRA FR, which has all the elements that CALDEIRA FNR has, plus the detailed chemistry physics, i.e., FR has all the physics FNR, plus one additional physics and one that adds loads outside the mesh or other elements directly related to the domain.

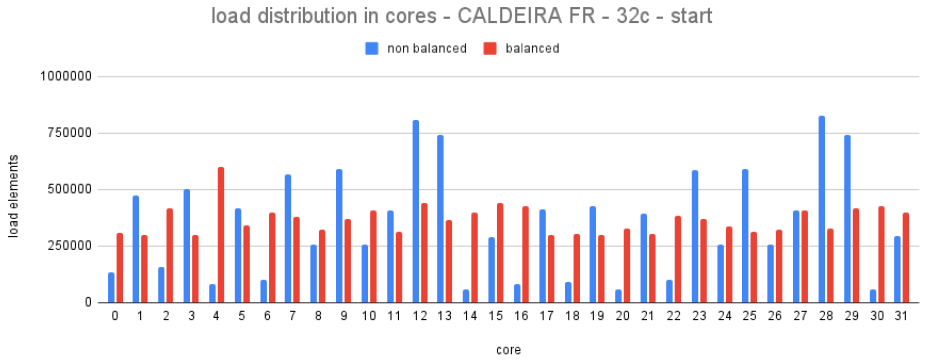


Figure 67 – Load distribution over the cores, CALDEIRA FR at the start, non-balanced vs balanced versions

As we can see in Figure 67, even for FR, the load is better distributed by the load balancing operation than without, which again is a good result.

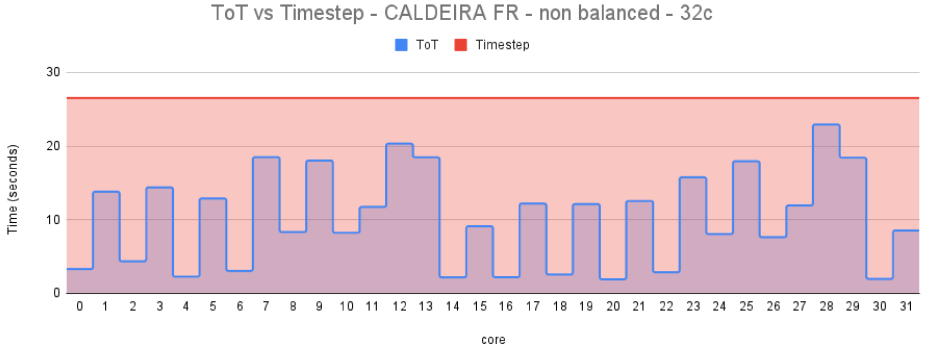


Figure 68 – ToT vs. Timestep for each core, CALDEIRA FR, non-balanced

This is further supported by analyzing the ToT and timestep relation for non-balanced and balanced versions of CALDEIRA FR, as presented by Figures 68 and 69, respectively.

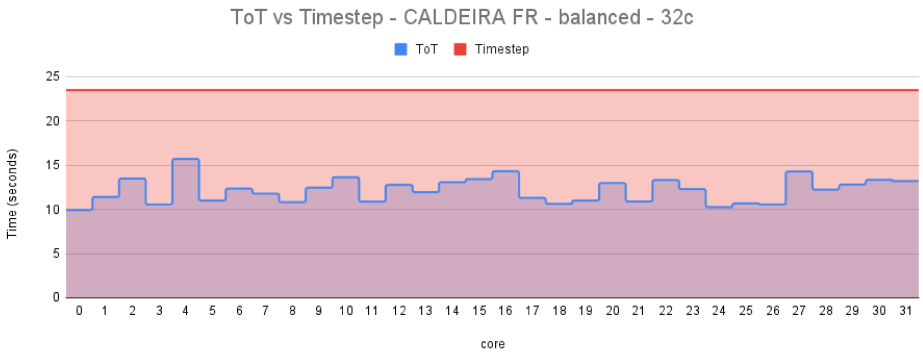


Figure 69 – ToT vs. Timestep for each core, CALDEIRA FR, balanced

Observe that, again, the ToT for each core is more closely related in the balanced version than in the non-balanced, indicating that in the first, the majority of the cores are doing roughly the same amount of work, which is a primary characteristic of a better-balanced simulation. Also, the timestep is smaller in the balanced simulation than in the non-balanced. Hence, we can conclude that the load balancing was adequate for the CALDEIRA FR as it was for the more *simpler* CALDEIRA FNR.

Since we already performed the gain analysis for CALDEIRA FNR, we will not repeat the same analysis for CALDEIRA FR. Instead, we are focusing on this section's main objective, which is to show the physics influence on the load balancing operation.

Until now, we presented that the "smaller" physical package of CALDEIRA SIMPLE had a negative effect on the load balancing, while CALDEIRA FNR and FR had a positive one. Now, we will use overlays to combine the data into single plots and then visualize how the cores behave in each version, with and without load balancing.

Commencing with the non-balanced versions, Figure 70 shows the ToT (time spent with computations) for each core, overlaying the SIMPLE, FNR, and FR versions. We can now see that the shapes of all curves resemble each other, with most of the higher and lower columns matching one another throughout the versions. This is expected, as all simulations have a common basic core of physics (see Table 10 for a comparison of the enabled physics in each case). Yet, the time for each ToT, grows as the simulation adds more physics.

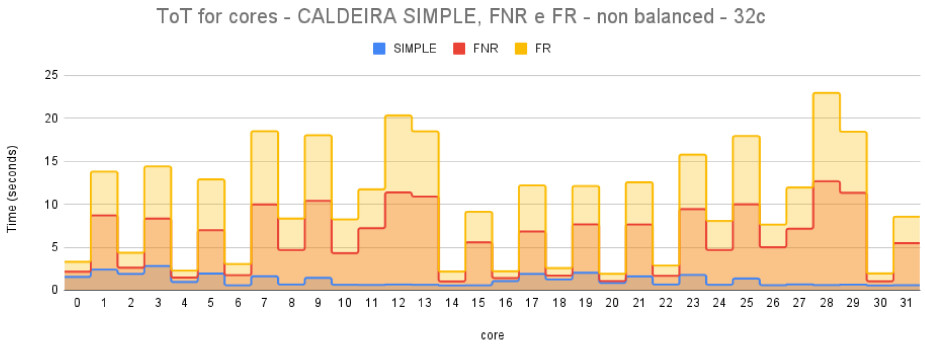


Figure 70 – ToTs for each core, all CALDEIRA versions, non-balanced

The same patterns can also be observed for the balanced versions, as Figure 71 presents. Especially the curves for FNR and FR, the shapes nearly match each other, showing that though detailed chemistry indeed adds more load, making the FR slower than FNR, the ToT's for both behave in the same way. This indicates that the additional physics of FR impacts the simulation, making it slower, but not enough to change the number of tasks assigned to each core.

A good explanation for this is that the physics FR adds generates load outside the mesh or other domain-related elements. Though it can be detected by the ToT, as presented by Figure 71, the load balancing technique can't properly handle this load.

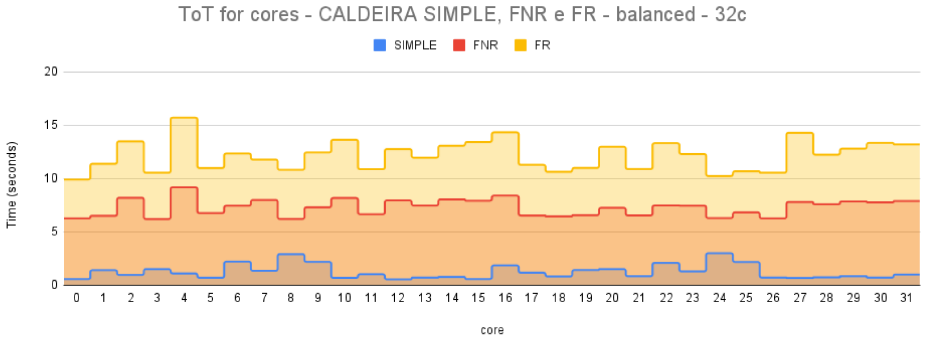


Figure 71 – ToTs for each core, all CALDEIRA versions, balanced

Even so, as presented in Figure 72, the load balancing is capable of enhancing the FR simulation to a point where its cores behave more closely to the balanced and even the non-balanced FNR, which is a simulation without the chemical reaction physics and therefore, *simpler*.

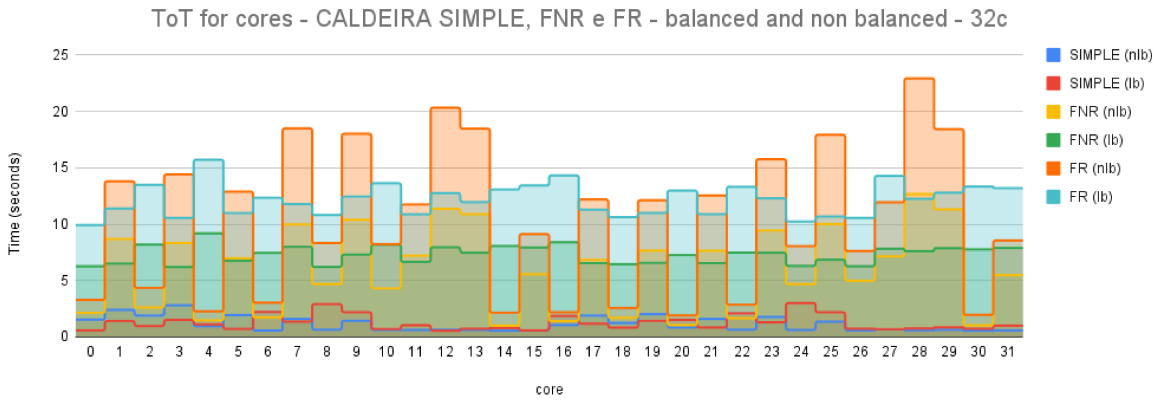


Figure 72 – ToTs for each core, all CALDEIRA versions, balanced and non-balanced

In conclusion, we demonstrate that incorporating physics into a simulation not only tends to slow it down but also affects load balancing, sometimes rendering it ineffective and at other times achieving the limits of our proposition.

This opens the door for further investigation and improvements of the proposition to enlarge what is a balanced enough simulation. This will be better described in section 5.2. For now, let us conclude this section.

4.5 Overall Analysis

The results we presented in the previous sections show that our proposition described in section 3.2, is capable of identifying the load generated in a parallel multi-physics simulation, using block-structured adaptive mesh, GMG as the main solver, directed-coupled physics like fluid-dynamics, immersed boundary, multi-phase, particle, turbulence, chemical reaction, etc., and redistribute this load throughout the used cores, while respecting the limitations imposed by the solver (GMG) and mesh implementation.

In section 2.4.1, we presented a summary of the state of the art for load balancing of parallel multi-physics simulations. We will now compare this state-of-the-art with our proposal to highlight our contribution. We will use the help of Table 13 for that purpose.

Table 13 – State of the art summary with this work

	Mesh Type	Mesh Structure	Adaptability	Discretization Technique & Solver	LB Algorithm	Retro-feeding physics	Physics	Specialization	Hardware Support
(SCHORNBAUM; RüDE, 2018)	Block-structured	octree	✓	LBM	Diffusion	○	Fluid Turbulence Particle Rigid Body	Multi-purposes	CPU
(BAUER et al., 2021)	Block-structured	octree	✓	LBM	Diffusion	○	Fluid Turbulence Particle Rigid Body	Multi-purposes	CPU-GPU
(RETTINGER; RüDE, 2019)	Block-structured	octree	○	LBM	SFC	○	Fluid Turbulence Particle Rigid Body	Multi-purposes	CPU-GPU
(NIEMöLLER et al., 2020)	Unstructured	octree	✓	FVM-AMG	SFC	○	Fluid Turbulence Acoustics	Computational Acoustics	CPU
(JUDE; SITARAMAN; WISSINK, 2022)	Block-structured	octree	✓	FDM-GMRES	SFC	✓	Fluid Turbulence Immersed Boundary [1]	Rotorcraft [2]	CPU-GPU
This work	Block-structured	Hash	✓	FVM,FEM-GMG	RCB	✓	Fluid Turbulence Immersed Boundary [1] Fluid-Structure Interaction [3] Particle Multi-phase Gas-Solid Gas-Liquid Chemical Reaction [4]	Multi-purposes	CPU

Table references:

1. Immersed Boundary is a physical model to describe a body inserted into a fluid. This body can be rigid or fully deformable (VERZICCO, 2023).

2. Rotorcraft CFD simulates physics inside the scope of rotor movement used by helicopters, drones, and other rotor-based machinery (JUDE; SITARAMAN; WISSINK, 2022).

3. Fluid-Structure Interaction may or may not be implemented with Immersed Boundary. Fluid-induced physical phenomena like vibrations are studied by this model (DOWELL; HALL, 2001).

4. Chemically Reactive Flows studied physical phenomena related to chemical reaction and fluid dynamics like combustion (RAMANUJA et al., 2023).

We can see in Table 13 that only the first three listed works and this work utilize multi-purpose codes. This is important because specialized codes will tend to run faster for the scopes they are designed for but will rarely face any problems outside those scopes. Also, they can implement the best structures for their job, while a more general code, like this work, will have to consider other parts, modules, and physics while implementing anything. The downside of a specialized code is its limited range. We can run CALDEIRA in MFSIM because it has chemical reaction physics. But we can't run this simulation in a rotorcraft code, as a boiler has nothing to do with rotorcraft.

Even so, the other works with multi-purpose codes all use a discretization technique that doesn't provide the same level of accuracy as this work (see section 2.1.5 for details). To exemplify this claim, the SPHERE test case we used here could probably also be tested in any of the three first works shown in Table 13, but will produce less accurate results than this work. This is the downside of their discretization technique. Ironically, the LBM used in those works is more load-balancing friendly than the FVM,FEM-GMG used in this work.

The mesh structure is another essential information we can extract from Table 13. Except for this work, all of the other uses octree, a tree-based data structure that functions very well with Hilbert curves-based load balancing algorithms like the SFC (see section 2.2 for more details). Though there is a discussion if the SFC is better than the other algorithms or not (HENDRICKSON; DEVINE, 2000) (JUDE; SITARAMAN; WISSINK, 2022), the fact that a tree-based structure allows to easily access spatial and dimensional information (SKOPAL et al., 2003), tends to make the load balancing a less complex task (JUDE; SITARAMAN; WISSINK, 2022), after all, load balancing a multi-physics simulation requires remapping parts of the mesh, i.e, spatial information, thorough the used cores. Also, tree-based structures in the mesh are known to be particularly very

adequate to adaptive parallel problems (WU; FIELD; KELLY, 1997), again, because of its more "easy to go" parallelization and, by extension, load balancing.

In other words, while all of the other works used a mesh structure that facilitates the load-balancing operation, this work uses a structure that complicates the operation. Hence, we described the limitations of this work in section 3.1 and why we used the concept of balanced enough simulation (section 4.3) to evaluate the results.

Also, we have retro-feeding physics, which adds layer of complexity (see section 2.1.3). The work of (JUDE; SITARAMAN; WISSINK, 2022) also has this feature but uses a mesh more adequate for parallelization and load balancing and a solver not so intrinsically tied to the mesh beyond being a specialized code, with fewer physics coupled.

In summary, that fact that we managed to apply load balancing in this mesh, so hardly tied up to the GMG solver, with many direct-coupled physics, some of them with retro-feeding mechanism, in a general purpose CFD code, and test this load balancing with industrial-scale simulations, with some degree of success, is our most significant contribution to the state of the art of load balancing for parallel multi-physics simulations with block-structured adaptive mesh.

There is room for improvement, as we pointed out before in this work, and we will describe this in section 5.2.

Chapter 5

Conclusion

We started this work with the hypothesis that balancing a multi-physics simulation is a complex task and a multiple approach would provide a satisfactory result, the balanced enough simulation (see section 1.3).

The results and discussion presented in Chapter 4 support our hypothesis, as we successfully achieved a balanced simulation for all three industrial cases and maintained balance over the academic case, which was already possible to balance.

Furthermore, the results also show that the best load-balancing results tend to be generated by tuning the components of the proposition, which opens some paths for future improvements. Similarly, the limits we found also demonstrate that it is possible to enhance the proposition and the code used by making specific changes to the mesh, solver, and load distribution when employing extra-mesh load generators. This is the subject of section 5.2.

Last but not least, this work produced some publications and additional investigations into the subject of load-balancing multi-physics simulations.

Cases like SPRAY, which grow over time, always tend to reach the limits of what load-balancing can produce, as eventually, there won't be enough resources to run the simulation. In this scenario, an "unorthodox" load distribution technique could produce interesting results outside the scope of the ordinary load-balance operation. As a spin-off of this work, we investigated this topic and published a paper with the results of our investigation.

5.1 Publications

The primary publication of this work is the paper *A multiple approach for the load balancing problem in parallel multi-physics simulations with block-structured mesh* that is under preparation for the SIAM Journal on Scientific Computing. This paper summarizes the proposition, its challenges, and the key findings of applying it to industrial-scale simulations.

This work's second publication is part of a by-product investigation of additional methods to improve load-balancing outside the main scope of redistribution of load-generation elements during a simulation. We investigate whether adding more computational resources and using data from one simulation to start another would produce a better load distribution and, therefore, improvements in the overall performance of a simulation. The investigation and the results were published in the paper "Reducing initial computational cost of complex turbulent flows simulations by using recorded data from an existing simulation" (FREITAS et al., 2023), presented orally at the 27th International Congress of Mechanical Engineering.

Since the tools developed in this derivative investigation were also used to speed up some of the FEIXES runs analyzed in this work, we can consider this offspring to also produce results for this work.

5.2 Future Work

To describe the future work, we must recall some of the results we presented in section 4.4.

One thing that is very clear throughout section 4.4 is the need for the user of this proposition to understand, at least at some level, how the simulation he is pretending to load balance behaves.

Even for simulations using the "auto" mode of load balancing, this is necessary. Though, as shown in section 4.4.2.2, this mode can produce good results, better results are achieved by calibrating the proposition components, which are heavily dependent on good knowledge of how the simulation behaves.

Also, the automatic mode is not always possible, as shown in section 4.4.2.1 (SPRAY results). If the user is oblivious to the intrinsic behavior of the simulation and always goes for auto, which is the default, he would consider SPRAY not possible to load balance when possible with the proper calibration of the ToT analysis.

This brings us to the first improvement in the proposition: a tool to help the user properly set the load balancing. More advanced CFD users probably won't need this, but less experienced ones can benefit.

Considering that we are in the vast scope of multi-physics simulations, and what we have shown in section 4.4, it is reasonable to assume that the calibration for one multi-physics simulation will not always apply to others. Since the calibration primarily involves adjusting thresholds and memory size to detect load behavior in a simulation, we could potentially train an Artificial Intelligence (AI) to assist the user in this task.

Although we haven't tested this hypothesis yet, we consider it at least viable to investigate, as pattern recognition is a fundamental part of AI and calibrating the load

balancing involves precisely recognizing the pattern of the load in the simulation to set thresholds and memory sizes adequately.

Another possible improvement is related to the basis of the proposition. As described in section 3.1, we built our proposition over a mesh unsuitable for parallelization and load balancing, hardly tied to a specific solver, and with some severe constraints regarding mesh construction and load distribution. Compared with the other works in the same area, as shown in Table 13, we are, to the best of our knowledge, one of the few, if not the only ones, trying to balance a multi-purpose CFD, with many directed-coupled physics, using this kind of constraints in regards to mesh and solver.

So an improvement is to change this by allowing MFSim, our base code, to use other types of mesh that make both parallelization and the load balancing operation more natural (JUDE; SITARAMAN; WISSINK, 2022) (WU; FIELD; KELLY, 1997), and at the same time, is not so hardly tied to the GMG, enabling the proposition to be applied to other solvers as well.

Also, since CFD is changing towards GPU (PISCAGLIA; GHIOLDI, 2023) (HE et al., 2020) (JESPERSEN, 2010), enabling other types of mesh and solver is vital to bring MFSim to this new era, which, of course, will demand more investigation regarding load balancing.

This is already a work in progress, with the developer team at MFLAB implementing tools to disassociate the mesh from the solver, new types of meshes, and integrating an implementation of the AMG solver. More publications will be regarding those changes, but it is a work in progress for now.

Last, and considering what we see with the results of CALDEIRA in section 4.4.3, balancing extra-mesh load brings our proposition to the limits and presents an entirely new challenge. Fortunately, this is starting to garner the attention of other teams as well (GÄRTNER et al., 2024), which we will also investigate in the future.

Bibliography

AGATI, G. et al. Liquid film formation: prediction accuracy of different numerical approaches. In: IOP PUBLISHING. **Journal of Physics: Conference Series**. 2022. v. 2385, n. 1, p. 012138. Disponível em: <https://doi.org/10.1088/1742-6596/2385/1/012138>.

ALTERYX. **What Is User Defined Function (UDF)?** 2024. Https://www.alteryx.com/pt-br/glossary/user-defined-function-udf. Accessed: 2025-03-27.

ARC4CFD. **CFD codes for HPC**. 2025. Https://arc4cfd.github.io/cfd$_c$*odes*/.*Accessed* : $2025 - 03 - 27$.

BADER, M. **Space-filling curves: an introduction with applications in scientific computing**. [S.l.]: Springer Science & Business Media, 2012. v. 9.

BAIGES, J. et al. Large-scale stochastic topology optimization using adaptive mesh refinement and coarsening through a two-level parallelization scheme. **Elsevier**, 2018.

BALAY, S. et al. Petsc, the portable, extensible toolkit for scientific computation. **Argonne National Laboratory**, v. 2, n. 17, 1998.

BARBI, F. et al. Numerical experiments of ascending bubbles for fluid dynamic force calculations. **Journal of the Brazilian Society of Mechanical Sciences and Engineering**, v. 40, n. 11, p. 519, Oct 2018. ISSN 1806-3691. Disponível em: <https://doi.org/10.1007/s40430-018-1435-7>.

BARTUSCHAT, D.; RÜDE, U. Parallel multiphysics simulations of charged particles in microfluidic flows. **Journal of Computational Science**, Elsevier, v. 8, p. 1–19, 2015. Disponível em: <https://doi.org/10.1016/j.jocs.2015.02.006>.

BAUER, M. et al. walberla: A block-structured high-performance framework for multiphysics simulations. **Computers & Mathematics with Applications**, Elsevier, v. 81, p. 478–501, 2021. Disponível em: <https://doi.org/10.1016/j.camwa.2020.01.007>.

BAYAT, M. et al. A review of multi-scale and multi-physics simulations of metal additive manufacturing processes with focus on modeling strategies. **Additive Manufacturing**, Elsevier, v. 47, p. 102278, 2021. Disponível em: <https://doi.org/10.1016/j.addma.2021.102278>.

BELYTSCHKO, T. Fluid-structure interaction. **Computers & Structures**, Elsevier, v. 12, n. 4, p. 459–469, 1980. Disponível em: <https://doi.org/10.1016/0045-7949(80)90121-2>.

BERGER, M. J.; OLIGER, J. Adaptive mesh refinement for hyperbolic partial differential equations. **Journal of Computational Physics**, v. 53, n. 3, p. 484–512, 1984. ISSN 0021-9991. Disponível em: <https://doi.org/10.1016/0021-9991(84)90073-1>.

BERNASCHI, M. et al. Muphy: A parallel multi physics/scale code for high performance bio-fluidic simulations. **Computer Physics Communications**, Elsevier, v. 180, n. 9, p. 1495–1502, 2009. Disponível em: <https://doi.org/10.1016/j.cpc.2009.04.001>.

BESSERON, X.; ADHAV, P.; PETERS, B. Parallel multi-physics coupled simulation of a midrex blast furnace. In: **Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops**. [s.n.], 2024. p. 87–98. Disponível em: <https://doi.org/10.1145/3636480.3636484>.

BOKHARI. A partitioning strategy for nonuniform problems on multiprocessors. **IEEE Transactions on Computers**, IEEE, v. 100, n. 5, p. 570–580, 1987. Disponível em: <https://doi.org/10.1109/TC.1987.1676942>.

BOMAN, E. G. et al. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. **Scientific Programming**, v. 20, n. 2, p. 129–150, 2012. Disponível em: <https://doi.org/10.3233/SPR-2012-0342>.

BULUÇ, A. et al. **Recent advances in graph partitioning**. [S.l.]: Springer, 2016.

CASTELO, A.; AFONSO, A. M.; BEZERRA, W. D. S. A hierarchical grid solver for simulation of flows of complex fluids. **Polymers**, v. 13, n. 18, 2021. ISSN 2073-4360. Disponível em: <https://www.mdpi.com/2073-4360/13/18/3168>.

CASTRO, L. P. d. et al. Implementation of a hybrid lagrangian filtered density function–large eddy simulation methodology in a dynamic adaptive mesh refinement environment. **Physics of Fluids**, v. 33, n. 4, p. 045126, 2021. Disponível em: <https://doi.org/10.1063/5.0045873>.

CHOURDAKIS, G. et al. preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]. **Open Research Europe**, v. 2, n. 51, 2022. Disponível em: <https://doi.org/10.12688/openreseurope.14445.2>.

DADVAND, P. et al. Migration of a generic multi-physics framework to hpc environments. **Computers & Fluids**, Elsevier, v. 80, p. 301–309, 2013. Disponível em: <https://doi.org/10.1016/j.compfluid.2012.02.004>.

DAMASCENO, M.; VEDOVOTO, J.; SILVEIRA-NETO, A. Turbulent inlet conditions modeling using large-eddy simulations. **CMES - Computer Modeling in Engineering and Sciences**, v. 104, p. 105–132, 01 2015.

DAMASCENO, M. M. R.; SANTOS, J. G. de F.; VEDOVOTO, J. M. Simulation of turbulent reactive flows using a fdf methodology – advances in particle density control for normalized variables. **Computers & Fluids**, v. 170, p. 128 – 140, 2018. ISSN 0045-7930. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0045793018302494>.

DAS, S.; HARVEY, D.; BISWAS, R. Parallel processing of adaptive meshes with load balancing. **IEEE Transactions on Parallel and Distributed Systems**, v. 12, n. 12, p. 1269–1280, 2001. Disponível em: <https://doi.org/10.1109/71.970562>.

DESAI, P. S.; SAWANT, N.; KEENE, A. On covid-19-safety ranking of seats in intercontinental commercial aircrafts: A preliminary multiphysics computational perspective. **Building Simulation**, v. 14, p. 1585–1596, 2021. Disponível em: <https://doi.org/10.1007/s12273-021-0774-y>.

DEVINE, K. D. et al. New challenges in dynamic load balancing. **Applied Numerical Mathematics**, v. 52, n. 2, p. 133–152, 2005. ISSN 0168-9274. ADAPT '03: Conference on Adaptive Methods for Partial Differential Equations and Large-Scale Computation. Disponível em: <https://doi.org/10.1016/j.apnum.2004.08.028>.

DONEA, J.; QUARTAPELLE, L.; SELMIN, V. An analysis of time discretization in the finite element solution of hyperbolic problems. **Journal of Computational Physics**, Elsevier, v. 70, n. 2, p. 463–499, 1987. Disponível em: <https://doi.org/10.1016/0021-9991(87)90191-4>.

DOWELL, E. H.; HALL, K. C. Modeling of fluid-structure interaction. **Annual review of fluid mechanics**, Annual Reviews 4139 El Camino Way, PO Box 10139, Palo Alto, CA 94303-0139, USA, v. 33, n. 1, p. 445–490, 2001. Disponível em: <https://doi.org/10.1146/annurev.fluid.33.1.445>.

DUBEY, A. Stencils in scientific computations. In: **Proceedings of the Second Workshop on Optimizing Stencil Computations**. [s.n.], 2014. p. 57–57. Disponível em: <https://doi.org/10.1145/2686745.2686756>.

DUBEY, A. et al. A survey of high level frameworks in block-structured adaptive mesh refinement packages. **Journal of Parallel and Distributed Computing**, v. 74, n. 12, p. 3217–3227, 2014. ISSN 0743-7315. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. Disponível em: <https://doi.org/10.1016/j.jpdc.2014.07.001>.

_____. Extensible component-based architecture for flash, a massively parallel, multiphysics simulation code. **Parallel Computing**, Elsevier, v. 35, n. 10-11, p. 512–522, 2009. Disponível em: <https://doi.org/10.1016/j.parco.2009.08.001>.

ELHADIDI, B.; KHALIFA, H. E. Comparison of coarse grid lattice boltzmann and navier stokes for real time flow simulations in rooms. In: SPRINGER. **Building Simulation**. 2013. v. 6, p. 183–194. Disponível em: <https://doi.org/10.1007/s12273-013-0107-x>.

EWERT, R.; KREUZINGER, J. Hydrodynamic/acoustic splitting approach with flow-acoustic feedback for universal subsonic noise computation. **Journal of Computational Physics**, v. 444, p. 110548, 2021. ISSN 0021-9991. Disponível em: <https://doi.org/10.1016/j.jcp.2021.110548>.

FERZIGER, J. H.; PERIĆ, M.; STREET, R. L. **Computational methods for fluid dynamics**. [S.l.]: springer, 2019.

FLúIDOS, L. de Mecânica de. **MFSim**. 2022. Https://www.mflab.mecanica.ufu.br/mfsim/. Accessed: 2025-03-27.

FORTUNA, A. de O. **Técnicas Computacionais para Dinâmica dos Fluídos Vol. 30**. [S.l.]: Edusp, 2000.

FREITAS, J. et al. Reducing initial computational cost of complex turbulent flows simulations by using recorded data from a existent simulation. **27th International Congress of Mechanical Engineering**, 2023. URL requires registration on ABCM. Disponível em: <http://dx.doi.org/10.26678/ABCM.COBEM2023.COB2023-0819>.

GARCIA-GASULLA, M. et al. A generic performance analysis technique applied to different cfd methods for hpc. **International Journal of Computational Fluid Dynamics**, Taylor & Francis, v. 34, n. 7-8, p. 508–528, 2020. Disponível em: <https://doi.org/10.1080/10618562.2020.1778168>.

GÄRTNER, J. W. et al. A chemistry load balancing model for openfoam. **Computer Physics Communications**, Elsevier, v. 305, p. 109322, 2024. Disponível em: <https://doi.org/10.1016/j.cpc.2024.109322>.

GEEKSFORGEEKS. **Cascading Rollback**. 2024. Https://www.geeksforgeeks.org/cascading-rollback/. Accessed: 2025-03-27.

_____. **FIFO (First-In-First-Out) approach in Programming**. 2024. Https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/. Accessed: 2025-03-27.

GENTILI, F.; PETRINI, F. On the role of the numerical analyses in forensic investigations of fire-induced progressive collapses of tall buildings. **International journal of forensic engineering**, Inderscience Publishers (IEL), v. 3, n. 1-2, p. 45–68, 2016. Disponível em: <https://doi.org/10.1504/IJFE.2016.075996>.

GOTTLIEB, S.; KETCHESON, D. I. Time discretization techniques. In: **Handbook of Numerical Analysis**. Elsevier, 2016. v. 17, p. 549–583. Disponível em: <https://doi.org/10.1016/bs.hna.2016.08.001>.

HAYES, A. **What Is T-Distribution in Probability? How Do You Use It?** 2025. Https://www.investopedia.com/terms/t/tdistribution.asp. Accessed: 2025-03-27.

HE, Y. et al. A cpu-gpu cross-platform coupled cfd-dem approach for complex particle-fluid flows. **Chemical Engineering Science**, Elsevier, v. 223, p. 115712, 2020. Disponível em: <https://doi.org/10.1016/j.ces.2020.115712>.

HENDRICKSON, B.; DEVINE, K. Dynamic load balancing in computational mechanics. **Computer Methods in Applied Mechanics and Engineering**, v. 184, n. 2, p. 485–500, 2000. ISSN 0045-7825. Disponível em: <https://doi.org/10.1016/S0045-7825(99)00241-8>.

JANSSON, N. et al. Cube: A scalable framework for large-scale industrial simulations. **The international journal of high performance computing applications**, Sage Publications Sage UK: London, England, v. 33, n. 4, p. 678–698, 2019. Disponível em: <https://doi.org/10.1177/1094342018816377>.

JESPERSEN, D. C. Acceleration of a cfd code with a gpu. **Scientific Programming**, Wiley Online Library, v. 18, n. 3-4, p. 193–201, 2010. Disponível em: <https://doi.org/10.3233/SPR-2010-0309>.

JUDE, D. et al. An overset generalised minimal residual method for the multi-solver paradigm. **International Journal of Computational Fluid Dynamics**, Taylor & Francis, v. 34, n. 1, p. 61–74, 2020. Disponível em: <https://doi.org/10.1080/10618562.2019.1710137>.

JUDE, D.; SITARAMAN, J.; WISSINK, A. An octree-based, cartesian navier–stokes solver for modern cluster architectures. **The Journal of Supercomputing**, Springer, v. 78, n. 9, p. 11409–11440, 2022. Disponível em: <https://doi.org/10.1007/s11227-022-04324-7>.

JUNIOR, R. C. et al. Modelagem matemática tridimensional para problemas de interação fluido-estrutura. Universidade Federal de Uberlândia, 2005. Disponível em: <https://repositorio.ufu.br/handle/123456789/14794>.

KALTENBACHER, M. **Computational acoustics**. [S.l.]: Springer, 2018.

KEYES, D. E. et al. Multiphysics simulations: Challenges and opportunities. **The International Journal of High Performance Computing Applications**, v. 27, n. 1, p. 4–83, 2013. Disponível em: <https://doi.org/10.1177/1094342012468181>.

_____. Multiphysics simulations: Challenges and opportunities. **The International Journal of High Performance Computing Applications**, v. 27, n. 1, p. 4–83, 2013. Disponível em: <https://doi.org/10.1177/1094342012468181>.

KO, S.-H. et al. Efficient runtime environment for coupled multi-physics simulations: Dynamic resource allocation and load-balancing. In: **2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing**. [s.n.], 2010. p. 349–358. Disponível em: <https://doi.org/10.1109/CCGRID.2010.107>.

LAN, Z.; TAYLOR, V.; BRYAN, G. Dynamic load balancing for structured adaptive mesh refinement applications. In: **International Conference on Parallel Processing, 2001.** [s.n.], 2001. p. 571–579. Disponível em: <https://doi.org/10.1109/ICPP.2001.952105>.

LIMA, R. S. d. et al. Desenvolvimento e implementação de malhas adaptativas bloco-estruturadas para computação paralela em mecânica dos fluidos. Universidade Federal de Uberlândia, 2012. Disponível em: <https://doi.org/10.14393/ufu.te.2012.84>.

LINTERMANN, A. et al. Massively parallel grid generation on hpc systems. **Computer Methods in Applied Mechanics and Engineering**, v. 277, p. 131–153, 2014. ISSN 0045-7825. Disponível em: <https://doi.org/10.1016/j.cma.2014.04.009>.

LONG, J. et al. Review of researches on coupled system and cfd codes. **Nuclear Engineering and Technology**, Elsevier, v. 53, n. 9, p. 2775–2787, 2021. Disponível em: <https://doi.org/10.1016/j.net.2021.03.027>.

LONGARES, J. M.; GARCÍA-JIMÉNEZ, A.; GARCÍA-POLANCO, N. Multiphysics simulation of bifacial photovoltaic modules and software comparison. **Solar Energy**, Elsevier, v. 257, p. 155–163, 2023. Disponível em: <https://doi.org/10.1016/j.solener.2023.04.005>.

MANCHESTERCFD. **All there is to know about different mesh types in CFD!** 2024. Https://www.manchestercfd.co.uk/post/all-there-is-to-know-about-different-mesh-types-in-cfd. Accessed: 2025-03-27.

MARIÉ, S.; RICOT, D.; SAGAUT, P. Comparison between lattice boltzmann method and navier–stokes high order schemes for computational aeroacoustics. **Journal of Computational Physics**, Elsevier, v. 228, n. 4, p. 1056–1070, 2009. Disponível em: <https://doi.org/10.1016/j.jcp.2008.10.021>.

MATSUSHITA, S.; AOKI, T. Gas-liquid two-phase flows simulation based on weakly compressible scheme with interface-adapted amr method. **Journal of Computational Physics**, Elsevier, v. 445, p. 110605, 2021. Disponível em: <https://doi.org/10.1016/j.jcp.2021.110605>.

MERZARI, E. et al. Exascale multiphysics nuclear reactor simulations for advanced designs. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [s.n.], 2023. p. 1–11. Disponível em: <https://doi.org/10.1145/3581784.3627038>.

MIGUET, S.; PIERSON, J.-M. Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing. In: SPRINGER. **High-Performance Computing and Networking: International Conference and Exhibition Vienna, Austria, April 28–30, 1997 Proceedings 5**. 1997. p. 550–564. Disponível em: <https://doi.org/10.1007/BFb0031628>.

MOCZ, P. **Awesome Astrophysical Simulation Codes**. 2025. Https://github.com/pmocz/awesome-astrophysical-simulation-codes. Accessed: 2025-03-27.

MOHAMAD, A. **Lattice boltzmann method**. [S.l.]: Springer, 2011. v. 70.

MOHAMED, M. Reduction of the generated aero-acoustics noise of a vertical axis wind turbine using cfd (computational fluid dynamics) techniques. **Energy**, Elsevier, v. 96, p. 531–544, 2016. Disponível em: <https://doi.org/10.1016/j.energy.2015.12.100>.

MORALES, F. A. P. et al. Fluid–structure interaction with a finite element–immersed boundary approach for compressible flows. **Ocean Engineering**, v. 290, p. 115755, 2023. ISSN 0029-8018. Disponível em: <https://www.sciencedirect.com/science/article/pii/S002980182302139X>.

MOTA, P. H. A.; VEDOVOTTO, J. M.; ARISTEU, S.-N. Assessment of critical brine disposal operations conditions by cfd modeling and a kriging metamodel. **Environmental Fluid Mechanics**, v. 167, p. 1573–1510, 2023. Disponível em: <https://doi.org/10.1007/s10652-023-09911-7>.

MURRONE, A.; SCHERRER, D. Large eddy simulation of a turbulent premixed flame stabilized by a backward facing step. In: **1st INCA Workshop**. [S.l.: s.n.], 2005. p. 1–9.

NARDECCHIA, F. et al. A numerical approach for the heat fluxes analysis in the near-wall region. In: **33rd UIT HEAT TRANSFER CONFERENCE**. [s.n.], 2015. v. 33. Disponível em: <https://www.researchgate.net/profile/Francesca-Pagliaro/publication/280563421_A_numerical_approach_for_the_heat_fluxes_analysis_in_the_near-_wall_region/links/55b9e4f808ae9289a090201a/A-numerical-approach-for-the-heat-fluxes-analysis-in-the-near-wall-region.pdf>.

NASA. **Turbomachinery and Turboelectric Systems Expertise**. 2025. Https://www1.grc.nasa.gov/research-and-engineering/cfd-codes-turbomachinery/. Accessed: 2025-03-27.

NETO, H. R. et al. Influence of seabed proximity on the vibration responses of a pipeline accounting for fluid-structure interaction. **Mechanical Systems and Signal Processing**, Elsevier, v. 114, p. 224–238, 2019. Disponível em: <https://doi.org/10.1016/j.ymssp.2018.05.017>.

NIEMöLLER, A. et al. Dynamic load balancing for direct-coupled multiphysics simulations. **Computers & Fluids**, v. 199, p. 104437, 2020. ISSN 0045-7930. Disponível em: <https://doi.org/10.1016/j.compfluid.2020.104437>.

NORDSLETTEN, D. et al. Coupling multi-physics models to cardiac mechanics. **Progress in biophysics and molecular biology**, Elsevier, v. 104, n. 1-3, p. 77–88, 2011. Disponível em: <https://doi.org/10.1016/j.pbiomolbio.2009.11.001>.

ONLINE, C. **CFD Codes**. 2025. Https://www.cfd-online.com/Wiki/Codes. Accessed: 2025-03-27.

OPENMPI. $\mathbf{MPI}_W time.2024.Https : //www.open - mpi.org/doc/v3.0/man3/MPI_W time.3.php.Accessed : 2025 - 03 - 27.$

PATEL, S. **Tank Sloshing Using SOLIDWORKS Flow Simulation**. 2025. Https://www.goengineer.com/blog/tank-sloshing-using-solidworks-flow-simulation. Accessed: 2025-03-27.

PERMANN, C. J. et al. Moose: Enabling massively parallel multiphysics simulation. **SoftwareX**, Elsevier, v. 11, p. 100430, 2020. Disponível em: <https://doi.org/10.1016/j.softx.2020.100430>.

PINHEIRO, A. P. et al. Modelling of aviation kerosene droplet heating and evaporation using complete fuel composition and surrogates. **Fuel**, v. 305, p. 121564, 2021. ISSN 0016-2361. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0016236121014459>.

_____. Ethanol droplet evaporation: Effects of ambient temperature, pressure and fuel vapor concentration. **International Journal of Heat and Mass Transfer**, v. 143, p. 118472, 2019. ISSN 0017-9310. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0017931019309214>.

PISCAGLIA, F.; GHIOLDI, F. Gpu acceleration of cfd simulations in openfoam. **Aerospace**, MDPI, v. 10, n. 9, p. 792, 2023. Disponível em: <https://doi.org/10.3390/aerospace10090792>.

PIVELLO, M. et al. A fully adaptive front tracking method for the simulation of two phase flows. **International Journal of Multiphase Flow**, v. 58, p. 72–82, 2014. ISSN 0301-9322. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0301932213001286>.

POZZETTI, G. et al. A parallel dual-grid multiscale approach to cfd–dem couplings. **Journal of computational physics**, Elsevier, v. 378, p. 708–722, 2019. Disponível em: <https://doi.org/10.1016/j.jcp.2018.11.030>.

PROJECT, E. C. **Codes to Solve Multiphysics Problems**. 2025. Https://www.exascaleproject.org/computer-codes-solving-multiphysics-problems/. Accessed: 2025-03-27.

PıNAR, A.; AYKANAT, C. Fast optimal load balancing algorithms for 1d partitioning. **Journal of Parallel and Distributed Computing**, v. 64, n. 8, p. 974–996, 2004. ISSN 0743-7315. Disponível em: <https://doi.org/10.1016/j.jpdc.2004.05.003>.

RALSTON, A. Views: Discrete mathematics: The new mathematics of science: The computer revolution has made discrete mathematics as indispensable as the calculus to science and technology. **American Scientist**, JSTOR, v. 74, n. 6, p. 611–618, 1986.

RAMANUJA, M. et al. Effect of chemically reactive nanofluid flowing across horizontal cylinder: Numerical solution. **Journal of Advanced Research in Numerical Heat Transfer**, v. 12, n. 1, p. 1–17, 2023.

RESEARCH, V. M. **$50.3 Billion Global High Performance Computing (HPC) Market with Rising CAGR at 6.3% by 2028**. 2022. Https://www.globenewswire.com/en/news-release/2022/06/14/2462328/0/en/50-3-Billion-Global-High-Performance-Computing-HPC-Market-with-Rising-CAGR-at-6-3-by-2028-Increasing-Investment-Industrial-IOT-are-Driving-the-Market-Top-companies-Landscape-Segmen.html. Accessed: 2025-03-27.

RETTINGER, C.; RüDE, U. Dynamic load balancing techniques for particulate flow simulations. **MDPI Computation**, 2019. Disponível em: <https://doi.org/10.3390/computation7010009>.

RUGE, J. W.; STüBEN, K. 4. algebraic multigrid. In: _____. **Multigrid Methods**. [s.n.], 1987. cap. 4, p. 73–130. Disponível em: <https://epubs.siam.org/doi/abs/10.1137/1.9781611971057.ch4>.

SAKANE, S.; AOKI, T.; TAKAKI, T. Parallel gpu-accelerated adaptive mesh refinement on two-dimensional phase-field lattice boltzmann simulation of dendrite growth. **Computational Materials Science**, Elsevier, v. 211, p. 111507, 2022. Disponível em: <https://doi.org/10.1016/j.commatsci.2022.111507>.

SAMPATH, R. S. et al. Dendro: parallel algorithms for multigrid and amr methods on 2: 1 balanced octrees. In: IEEE. **SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing**. 2008. p. 1–12. Disponível em: <https://doi.org/10.1109/SC.2008.5218558>.

SANTOS, J. G. d. F. Mathematical and computational modeling of gas-solid flows in dynamic adaptive mesh. **Master thesis, Universidade Federal de Uberlândia**, Uberlândia, Brazil, 2019.

SCHORNBAUM, F.; RüDE, U. Extreme-scale block-structured adaptive mesh refinement. **SIAM Journal on Scientific Computing**, v. 40, n. 3, p. C358–C387, 2018. Disponível em: <https://doi.org/10.1137/17M1128411>.

SILVA, A. T. d. et al. Validation of hig-flow software for simulating two-phase flows with a 3d geometric volume of fluid algorithm. **Mathematics**, MDPI, v. 11, n. 18, p. 3900, 2023. Disponível em: <https://doi.org/10.3390/math11183900>.

SIRCAR, A. et al. **Enabling multiphysics simulation of fusion blankets on exascale architecture using OpenFOAM**. [S.l.], 2023.

SKOPAL, T. et al. Revisiting m-tree building principles. In: SPRINGER. **Advances in Databases and Information Systems: 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003. Proceedings 7**. 2003. p. 148–162. Disponível em: <https://doi.org/10.1007/978-3-540-39403-7_13>.

SMAN, R. van der. Multicubed: Multiscale-multiphysics simulation of food processing. **Food Structure**, v. 33, p. 100278, 2022. ISSN 2213-3291. Disponível em: <https://doi.org/10.1016/j.foostr.2022.100278>.

SOUSA, F. et al. A finite difference method with meshless interpolation for incompressible flows in non-graded tree-based grids. **Journal of Computational Physics**, v. 396, p. 848–866, 2019. ISSN 0021-9991. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0021999119304966>.

SOUZA, P. R. C. et al. Multi-phase fluid–structure interaction using adaptive mesh refinement and immersed boundary method. **Journal of the Brazilian Society of Mechanical Sciences and Engineering**, v. 44, p. 152, 2022. Disponível em: <https://doi.org/10.1007/s40430-022-03417-x>.

SRIVASTAVA, S.; DADHEECH, P.; BENIWAL, M. K. Load balancing using high performance computing cluster programming. **International Journal of Computer Science Issues (IJCSI)**, Citeseer, v. 8, n. 1, p. 62, 2011.

STIVAL, L. J. et al. Wake modeling and simulation of an experimental wind turbine using large eddy simulation coupled with immersed boundary method alongside a dynamic adaptive mesh refinement. **Energy Conversion and Management**, v. 268, p. 115938, 2022. ISSN 0196-8904. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0196890422007348>.

STÜBEN, K. et al. An introduction to algebraic multigrid. **Multigrid**, p. 413–532, 2001. Disponível em: <https://doi.org/10.1016/B978-0-444-50616-0.50012-9>.

SUSS, A. et al. Comprehensive comparison between the lattice boltzmann and navier–stokes methods for aerodynamic and aeroacoustic applications. **Computers & Fluids**, Elsevier, v. 257, p. 105881, 2023. Disponível em: <https://doi.org/10.1016/j.compfluid.2023.105881>.

TATARCHENKO, M.; DOSOVITSKIY, A.; BROX, T. Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs. In: **Proceedings of the IEEE international conference on computer vision**. [s.n.], 2017. p. 2088–2096. Disponível em: <https://openaccess.thecvf.com/content_iccv_2017/html/Tatarchenko_Octree_Generating_Networks_ICCV_2017_paper.html>.

TEUNISSEN, J.; KEPPENS, R. A geometric multigrid library for quadtree/octree amr grids coupled to mpi-amrvac. **Computer Physics Communications**, Elsevier, v. 245, p. 106866, 2019. Disponível em: <https://doi.org/10.1016/j.cpc.2019.106866>.

TOTOUNFEROUSH, A. Data-integrated methods for performance improvement of massively parallel coupled simulations. 2022. Disponível em: <http://dx.doi.org/10.18419/opus-12375>.

TRILINOS. **The Trilinos Project Website**. 2020. Https://trilinos.github.io. Accessed: 2025-03-27.

TURCIOS, R. A. S. t-student: Usos y abusos. **Revista mexicana de cardiología**, Asociación Nacional de Cardiólogos de México, Sociedad de Cardiología ..., v. 26, n. 1, p. 59–61, 2015. Disponível em: <https://www.scielo.org.mx/scielo.php?pid=s0188-21982015000100009&script=sci_arttext>.

UNIVERSITY, C. **Introduction to MPI**. 2024. Https://carleton.ca/rcs/rcdc/introduction-to-mpi/. Accessed: 2025-03-27.

VEDOVOTO, J. M.; SERFATY, R.; NETO, A. D. S. Mathematical and numerical modeling of turbulent flows. **Anais da Academia Brasileira de Ciências**, Academia Brasileira de Ciências, v. 87, n. An. Acad. Bras. Ciênc., 2015 87(2), p. 1195–1232, Apr 2015. ISSN 0001-3765. Disponível em: <https://doi.org/10.1590/0001-37652015201400510>.

VERZICCO, R. Immersed boundary methods: Historical perspective and future outlook. **Annual Review of Fluid Mechanics**, Annual Reviews, v. 55, n. 1, p. 129–155, 2023. Disponível em: <https://doi.org/10.1146/annurev-fluid-120720-022129>.

VILLAR, M. M. et al. Análise numérica detalhada de escoamentos multifásicos bidimensionais. Universidade Federal de Uberlândia, 2007.

WESSELING, P. **Introduction to multigrid methods**. [S.l.], 1995.

WESSELING, P.; OOSTERLEE, C. W. Geometric multigrid with applications to computational fluid dynamics. **Journal of computational and applied mathematics**, Elsevier, v. 128, n. 1-2, p. 311–334, 2001. Disponível em: <https://doi.org/10.1016/S0377-0427(00)00517-3>.

WIKIPEDIA. **Multigrid v-cycle**. 2024. Https://en.wikipedia.org/wiki/Multigrid$_m ethod. Accessed: 2025 - 03 - 27$.

WILLIAMSON, R. et al. Multidimensional multiphysics simulation of nuclear fuel behavior. **Journal of Nuclear Materials**, v. 423, n. 1, p. 149–163, 2012. ISSN 0022-3115. Disponível em: <https://doi.org/10.1016/j.jnucmat.2012.01.012>.

WU, Q.; FIELD, A.; KELLY, P. H. M-tree: A parallel abstract data type for block-irregular adaptive applications. In: SPRINGER. **Euro-Par'97 Parallel Processing: Third International Euro-Par Conference Passau, Germany, August 26–29, 1997 Proceedings 3**. 1997. p. 638–649. Disponível em: <https://doi.org/10.1007/BFb0002795>.

YIN, C.; ROSENDAHL, L. A.; KÆR, S. K. Chemistry and radiation in oxy-fuel combustion: a computational fluid dynamics modeling study. **Fuel**, Elsevier, v. 90, n. 7, p. 2519–2529, 2011. Disponível em: <https://doi.org/10.1016/j.fuel.2011.03.023>.

YU, Z.; FAN, L.-S. An interaction potential based lattice boltzmann method with adaptive mesh refinement (amr) for two-phase flow simulation. **Journal of Computational Physics**, Elsevier, v. 228, n. 17, p. 6456–6478, 2009. Disponível em: <https://doi.org/10.1016/j.jcp.2009.05.034>.

YUSTE, S. B.; QUINTANA-MURILLO, J. A finite difference method with non-uniform timesteps for fractional diffusion equations. **Computer Physics Communications**, Elsevier, v. 183, n. 12, p. 2594–2600, 2012. Disponível em: <https://doi.org/10.1016/j.cpc.2012.07.011>.

ZHANG, J. Lattice Boltzmann method for microfluidics: models and applications. **Microfluidics and Nanofluidics**, v. 10, n. 1, p. 1–28, jan. 2011. ISSN 1613-4990. Disponível em: <https://doi.org/10.1007/s10404-010-0624-1>.

ZHANG, W. et al. Amrex: Block-structured adaptive mesh refinement for multiphysics applications. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 35, n. 6, p. 508–526, 2021. Disponível em: <https://doi.org/10.1177/10943420211022811>.

ZOLTAN. **Recursive Coordinate Bisection (RCB)**. 2025. Disponível em: <https://sandialabs.github.io/Zoltan/ug_html/ug_alg_rcb.html>.