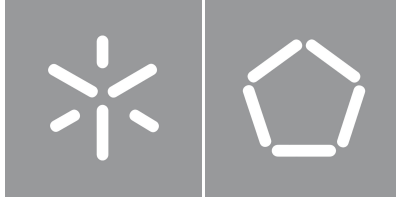


**University of Minho**  
School of Engineering

Ana Sofia Marinho Gomes

**Interweaving irregular numeric simulation  
workloads with ray forwarding techniques  
for in-situ visualization use case**





**University of Minho**  
School of Engineering

Ana Sofia Marinho Gomes

**Interweaving irregular numeric simulation  
workloads with ray forwarding techniques  
for in-situ visualization use case**

Master's Dissertation in Informatics Engineering

Dissertation supervised by

**António Luís Pinto Ferreira de Sousa (supervisor)**  
**João Carlos Garcia da Cunha Barbosa (supervisor  
in Research Center)**

# Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

## License granted to users of this work:



**CC BY**

<https://creativecommons.org/licenses/by/4.0/>



# Acknowledgements

Firstly, I would like to thank my supervisor, Professor António Sousa, for presenting me with this challenging project and guiding me throughout its development. Besides, I would also like to express my deepest thanks to Professor João Barbosa for his tireless help. I would be incapable of completing this thesis without his knowledgeable advice.

Furthermore, I am extremely grateful for having the best of friends by my side. Thank you, Adélia, for being the eldest sister I never had. Thank you, Pedro, for always being a beam of light that makes me laugh uncontrollably. Thank you, Rui, for bearing my wandering thoughts and endlessly supporting me through all these years.

Moreover, I would also like to extend my immeasurable gratitude to my boyfriend, Pedro. Thank you for being my safe harbor and for coloring my darkest days.

It is hard to put into words the never-ending appreciation I have for those who made me who I am today: my family. I would like to express a special thanks to Tio Luís, Tia Sameiro, and Vó Micas for being one of the foundations of my education and being by my side no matter what. Last but not least, I am forever indebted to my little sister, Maria, and my parents, Luís and Anabela. Thank you for keeping me grounded, fuelling my dreams, and always believing in me, even when I could not do it myself.

This work was financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

# Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, july 2024

Assinado por: **Ana Sofia Marinho Gomes**  
Num. de Identificação: 15996630  
Data: 2024.07.12 12:14:05 +0100

Ana Sofia Marinho Gomes

# Abstract

For many decades, post hoc processing was the base of scientific analysis and visualization. In this approach, the simulation program saves the generated data into permanent storage so the visualization program can posteriorly load it. This practice demands costly checkpoints that consume a lot of time and energy. Consequently, in situ processing has been gaining popularity to the detriment of post hoc processing.

In situ solutions do not depend on checkpoints, so they require fewer I/O operations when compared to post hoc ones. Nevertheless, they usually execute the simulation and visualization in distinct computing resources, conducting to a higher power consumption. LOOM is a task-scheduling framework that tries to solve this problem. It enables tightly coupled in situ visualization without substantially increasing the simulation execution time by applying an “*interweaving*” method. This method performs the visualization in the idle times of the simulation. However, the framework is still incapable of handling large-scale problems.

The goal of this thesis was to upgrade LOOM so it can deal with large-scale problems and to ally it with a ray-forwarding technique to try to overcome the distributed memory obstacles. For that, special primitives were created to handle efficient communication across multiple nodes while maintaining the philosophy behind the framework. The enhanced version of LOOM was tested on VADIS, a CFD model conceived to simulate the dispersion of air pollutants on urban morphologies. The results presented in this thesis show performance improvements of 2.64 for VADIS using multiple computing nodes.

**Keywords** High performance computing, scientific visualization, in situ processing

# Resumo

Durante várias décadas, o processamento post hoc foi a base da análise e visualização científica. Neste método, o programa de simulação guarda os dados gerados em memória permanente, de modo a que o programa de visualização os possa carregar posteriormente. Este comportamento exige que sejam efetuados *checkpoints* que conduzem ao consumo de grandes quantidades de tempo e energia. Consequentemente, o processamento in situ tem ganho popularidade em detrimento do processamento post hoc.

As soluções in situ não dependem de *checkpoints*, por isso requerem menos operações de I/O comparativamente com as soluções post hoc. Contudo, estas costumam executar a simulação e a visualização em recursos computacionais distintos, levando a um grande consumo de energia. O LOOM é uma *framework* de *scheduling* de tarefas que tenta solucionar este problema. Esta permite a execução de visualização in situ *tightly coupled* sem aumentar substancialmente o tempo de execução da simulação através da utilização de um método de “*interweaving*”. Este método executa a visualização nos tempos inutilizados pela simulação. Apesar disso, esta *framework* ainda é incapaz de lidar com problemas em grande escala.

O objetivo desta tese é melhorar o LOOM, de modo a que este consiga lidar com problemas em grande escala, bem como aliá-lo a uma técnica de *ray-forwarding* para ultrapassar os obstáculos da memória distribuída. Para isso, foram criadas primitivas especiais para estabelecer uma comunicação eficiente entre diversos nodos, mantendo a filosofia por detrás da *framework*. A versão melhorada do LOOM foi testada no VADIS, um modelo de dinâmica de fluidos computacional concebido para simular a dispersão de poluentes em áreas urbanas. Os resultados apresentados nesta tese demonstram melhorias de performance de 2.64 quando o VADIS é executado em múltiplos nodos computacionais.

**Palavras-chave**    Computação de alto desempenho, visualização científica, processamento in situ

# Contents

|          |                                                         |           |
|----------|---------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                     | <b>1</b>  |
| 1.1      | Problem                                                 | 1         |
| 1.2      | Objectives and Challenges                               | 2         |
| 1.3      | Document Structure                                      | 3         |
| <b>2</b> | <b>State of the Art</b>                                 | <b>4</b>  |
| 2.1      | Terminology for In Situ Systems                         | 4         |
| 2.2      | Processing Unit Architecture                            | 8         |
| 2.3      | Parallel Rendering Strategies                           | 10        |
| 2.4      | Intertwining Simulation with Visualization and Analysis | 13        |
| 2.4.1    | GoldRush                                                | 13        |
| 2.4.2    | LOOM                                                    | 15        |
| 2.5      | Distributed Memory Ray Tracing                          | 17        |
| 2.5.1    | Dynamic Scheduling                                      | 18        |
| 2.5.2    | SpRay                                                   | 19        |
| 2.5.3    | Galaxy                                                  | 21        |
| 2.6      | VADIS                                                   | 24        |
| 2.7      | Tools and Libraries                                     | 26        |
| 2.7.1    | High-Performance Rendering Libraries                    | 26        |
| 2.7.2    | Profiling Tools                                         | 27        |
| <b>3</b> | <b>Adapting LOOM and VADIS</b>                          | <b>29</b> |
| 3.1      | Baseline Version                                        | 29        |
| 3.1.1    | Synchronous Multi-Node Version of VADIS                 | 29        |
| 3.1.2    | Transition to the Asynchronous Multi-Node Version       | 31        |
| 3.1.3    | Communication System                                    | 33        |

|          |                                                    |           |
|----------|----------------------------------------------------|-----------|
| 3.1.4    | Voting Mechanism . . . . .                         | 34        |
| 3.1.5    | Asynchronous Multi-Node Version of VADIS . . . . . | 38        |
| 3.2      | Ray-Forwarding Version . . . . .                   | 39        |
| 3.2.1    | First Ray-Forwarding Version . . . . .             | 40        |
| 3.2.2    | Second Ray-Forwarding Version . . . . .            | 42        |
| 3.2.3    | Dicing of the Visualisation Tasks . . . . .        | 44        |
| <b>4</b> | <b>LOOM's Performance Assessment</b>               | <b>46</b> |
| 4.1      | Methodology . . . . .                              | 46        |
| 4.2      | Performance Evaluation . . . . .                   | 48        |
| 4.2.1    | Simulation . . . . .                               | 48        |
| 4.2.2    | Simulation and Visualization . . . . .             | 52        |
| <b>5</b> | <b>Conclusions and Future Work</b>                 | <b>59</b> |
| 5.1      | Conclusions . . . . .                              | 59        |
| 5.2      | Future Work . . . . .                              | 60        |
| <b>I</b> | <b>Appendices</b>                                  | <b>67</b> |
| <b>A</b> | <b>Message Dictionary</b>                          | <b>68</b> |
| A.1      | LOOM . . . . .                                     | 68        |
| A.2      | VADIS . . . . .                                    | 69        |

# List of Figures

|    |                                                                                                                                    |    |
|----|------------------------------------------------------------------------------------------------------------------------------------|----|
| 1  | The six axes for describing in situ systems, along with their categories and subcategories.                                        | 5  |
| 2  | Comparison of the single-core and multi-core architectures.                                                                        | 9  |
| 3  | Multi-core cluster.                                                                                                                | 10 |
| 4  | Rendering pipeline.                                                                                                                | 10 |
| 5  | Rendering pipeline with sort-first.                                                                                                | 11 |
| 6  | Rendering pipeline with sort-middle.                                                                                               | 12 |
| 7  | Rendering pipeline with sort-last.                                                                                                 | 12 |
| 8  | Scheduling with and without GoldRush.                                                                                              | 14 |
| 9  | GoldRush’s architecture.                                                                                                           | 14 |
| 10 | Task timeline plot where simulation and visualization tasks are in red and green, respectively. Images from Barbosa et al. (2021). | 15 |
| 11 | Ray Tracing Basics.                                                                                                                | 17 |
| 12 | SpRay version for in situ rendering.                                                                                               | 20 |
| 13 | Message processing layer.                                                                                                          | 22 |
| 14 | Task action execution layer.                                                                                                       | 23 |
| 15 | Ray list processing.                                                                                                               | 24 |
| 16 | Input and output data of FLOW and DISPER.                                                                                          | 25 |
| 17 | Open VKL algorithms. Images from Intel (2023b).                                                                                    | 27 |
| 18 | Flowchart of LOOM’s asynchronous communication system.                                                                             | 33 |
| 19 | Flowchart of voting mechanism I.                                                                                                   | 35 |
| 20 | Flowchart of voting mechanism II.                                                                                                  | 36 |
| 21 | Flowchart of voting mechanism III.                                                                                                 | 37 |
| 22 | Flowchart of the workaround found to voting mechanism III.                                                                         | 38 |
| 23 | Flowchart of the rendering procedure on the first ray-forwarding version.                                                          | 40 |

|    |                                                                                                                                                                                                                 |    |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 24 | Flowchart of the rendering procedure on the second ray-forwarding version. . . . .                                                                                                                              | 43 |
| 25 | Flowchart of the task <i>dicing</i> on the ray-forwarding versions. . . . .                                                                                                                                     | 44 |
| 26 | Viewports used on the tests. . . . .                                                                                                                                                                            | 47 |
| 27 | Number of particles processed by each rank in each phase. . . . .                                                                                                                                               | 49 |
| 28 | Particle distribution on each phase for the partition <code>-p 1 4 1</code> . . . . .                                                                                                                           | 50 |
| 29 | Particle distribution on each phase for the partition <code>-p 2 2 1</code> . . . . .                                                                                                                           | 50 |
| 30 | Ranks that constrain the simulation speed up on each phase. . . . .                                                                                                                                             | 51 |
| 31 | Output images of the street view using different partitions. . . . .                                                                                                                                            | 55 |
| 32 | Baseline version's task timeline plots for each rank for the partition <code>-p 1 4 1</code> when rendering the street view. The simulation and visualization tasks are in red and green, respectively. . . . . | 56 |
| 33 | Baseline version's task timeline plots for each rank for the partition <code>-p 2 2 1</code> when rendering the street view. The simulation and visualization tasks are in red and green, respectively. . . . . | 56 |
| 34 | Output images of the top view using different partitions. . . . .                                                                                                                                               | 57 |



## List of Tables

|   |                                                                                                                                                                            |    |
|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1 | Simulation's execution time (in seconds) and speedup for each partition. . . . .                                                                                           | 48 |
| 2 | Visualization's execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the top view with the baseline version. . . . .          | 53 |
| 3 | Visualization's execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the top view with the ray-forwarding version. . . . .    | 53 |
| 4 | Visualization's execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the street view with the baseline version. . . . .       | 54 |
| 5 | Visualization's execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the street view with the ray-forwarding version. . . . . | 54 |

# Acronyms

**AMR** Adaptative Mesh Refinement.

**API** Application Programming Interface.

**AVX** Advanced Vector Extensions.

**AVX-512** Advanced Vector Extensions 512.

**AVX2** Advanced Vector Extensions 2.

**CFD** Computational Fluid Dynamic.

**CMP** Chip Multiprocessors.

**DAG** Directed Acyclic Graph.

**DRAM** Dynamic Random Access Memory.

**GNRQ** Global Not Ready Queue.

**GRQ** Global Ready Queue.

**HPC** High Performance Computing.

**I/O** Input/Output.

**LRQ** Local Ready Queue.

**MPI** Message Passing Interface.

**OS** Operating System.

**RAM** Random Access Memory.

**RANS** Reynolds Averaged Navier-Stokes.

**SIMD** Single Instruction, Multiple Data.

**SMT** Simultaneous Multi-Threading.

**SPMD** Single Program, Multiple Data.

**SSE** Streaming **SIMD** Extensions.

# Chapter 1

## Introduction

In today's world, where efficiency is paramount, **High Performance Computing (HPC)** is being extensively researched to meet the demands of various applications, including those dedicated to numerical simulations. In this specific case, investigators use computational resources to generate colossal amounts of data and, most of the time, to create a graphical representation of the generated datasets to ease the analysis of the results ([Johnson and Hansen, 2004](#)).

For many decades, post hoc processing was the base of scientific analysis and visualization ([Childs et al., 2020](#)), but it has been losing popularity. Over the years, the computer capacity has been increasing, but so has the **I/O** gap. In post hoc approaches, the simulation program loads the generated data into permanent storage so the visualization program can posteriorly load it ([Childs et al., 2020](#)). This behavior demands costly checkpoints that consume a lot of time and energy. Therefore, in situ is replacing the post hoc processing to reduce the I/O impact ([Ribés and Raffin, 2020](#)).

### 1.1 Problem

In situ solutions do not rely on checkpoints, thus reducing the number of I/O operations. However, they typically use distinct computational resources to execute the simulation and the visualization, leading to greater power and energy consumption ([Ahrens et al., 2011](#)).

LOOM ([Barbosa et al., 2021](#)) is a task-scheduling framework that tries to solve this problem. It enables tightly coupled in situ visualization without considerably increasing the simulation execution time by applying an “*interweaving*” method. This approach executes the visualization in the idle times of the simulation, hiding up to 2/3 of the visualization cost.

Nevertheless, LOOM is still not ready to handle large-scale problems. For that to happen, it has to be capable of handling distributed workload among multiple computing nodes so they can cooperatively build the image using a parallel rendering strategy, such as sort-first, sort-middle, or sort-last. Besides

that, it needs to provide a communication system that allows an efficient connection between the distinct computing nodes. The latency in the communication between processors in distinct nodes can drastically decrease the performance of the cluster system [Shainer et al. \(2013\)](#). Some frameworks already try to manage scientific ray tracing in distributed environments, such as Galaxy ([Abram et al., 2018](#)) and SpRay ([Park et al., 2018](#)).

Handling visualization workloads across multiple nodes is not a trivial problem, even with dedicated resources. The problem is far more complex in tightly coupled visualization, the approach used in LOOM. Therefore, this thesis aims to study the impact of distributed rendering in the LOOM "*interweaving*" strategy.

## 1.2 Objectives and Challenges

The focal point of this thesis is to upgrade LOOM so it can deal with large-scale problems and to ally it with a ray-forwarding technique to try to overcome the distributed memory obstacles. The task handling and prediction model may need to be carefully re-calibrated to fit this case.

For that, it will be necessary to implement an asynchronous communication protocol to coordinate the work among multiple computational nodes, similar to the one presented on Galaxy ([Abram et al., 2018](#)). This protocol needs to allow the trade and processing of messages while maintaining the philosophy behind LOOM.

Besides, since each node will only hold a part of the data, close attention will be paid to the data distribution among the available resources. This management is extremely important since it can lead to workload imbalances that will negatively impact the program's execution. Additionally, it is pertinent to analyze if data migration between nodes would be advantageous.

The improvements made to the framework will be tested with VADIS (pollutant DISpersion in the atmosphere under VAriable wind conditions) ([Martins and Borrego, 1998](#); [Borrego et al., 2003](#); [Amorim et al., 2013](#)), a software capable of simulating flow and air pollutant dispersion in urban areas. This numerical simulation, detailed in section 2.6, already takes advantage of LOOM. VADIS will also need to be adapted to exploit LOOM's new features.

At the end of this work, it is expected to have an enhanced version of the LOOM framework, with special primitives to handle efficient communication across multiple nodes and to produce a rough estimator for workloads in several computational resources. The goal is to study the best technique to hide most visualization workloads and related communication costs.

## 1.3 Document Structure

The rest of this document is structured as follows. Chapter 2 explores the state of the art. It starts by mentioning a proposed terminology for describing in situ systems and the strategies used in parallel rendering. Then, it analyses the functioning of frameworks that intertwine the simulation with the visualization and distributed memory ray tracing. To conclude, it presents VADIS, as well as the used tools and libraries.

Chapter 3 depicts the adaptation of LOOM and VADIS. It describes the changes made to LOOM and the two implemented versions of VADIS: a synchronous version without ray-forwarding and an asynchronous version with ray-forwarding. Chapter 4 evaluates the LOOM's performance assessment. At last, chapter 5 dives into the conclusions and the prospects for future work.

## Chapter 2

# State of the Art

This chapter presents relevant concepts, namely a system for describing in situ applications and the existing parallel rendering strategies. It also explores two frameworks that enable the intertwining of simulation with analysis and visualization (GoldRush and LOOM).

Besides that, it presents three approaches that handle distributed memory ray tracing and dives into the VADIS functioning. At last, it concludes with an overview of the tools and libraries mentioned and used in this thesis.

## 2.1 Terminology for In Situ Systems

For many years, post hoc processing prevailed in scientific analysis and visualization. In this kind of processing, analysis and visualization applications only have access to the data generated by the simulation after it reaches the persistent storage. Usually, the simulation programs save the data repeatedly every once in a while.

However, in situ processing has been gaining popularity as an alternative to post hoc. In this type of processing, the analysis and visualization programs perform while the simulation generates the data. According to [Childs et al. \(2020\)](#), several terms have been utilized to characterize in situ processing and its derived approaches. Nevertheless, the community has still not agreed on a coherent classification. In the last decade, the expression "in situ" has been widely employed:

1. To depict a particular method for data processing;
2. As a blanket term for a whole paradigm of processing.

The absence of consistent terminology confuses the analysis and visualization community and its stakeholders. To solve this problem, a group of over fifty specialists ([Childs et al., 2020](#)) assembled to standardize the in situ terminology. At last, they established six axes that allow more precise descriptions

of in situ systems: integration type, proximity, access, division of execution, operation control and output type. Moreover, they suggested a new terminology based on the previously mentioned axes.

These axes are composed of multiple categories and subcategories, as shown in figure 1. The subcategories will only be briefly mentioned.

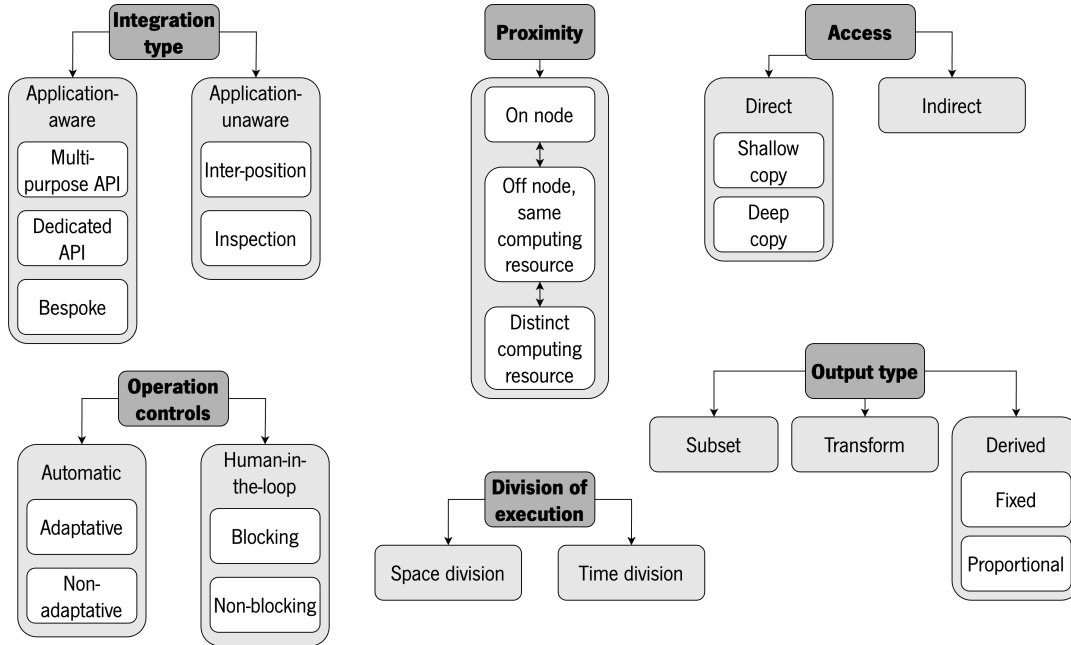


Figure 1: The six axes for describing in situ systems, along with their categories and subcategories.

## Integration Type

The integration type indicates how the simulation incorporates the analysis and visualization methods. It contains two distinct categories:

- **Application-Aware:** the simulation code is aware of the analysis and visualization incorporation. This category comprises three subcategories: Bespoke, Dedicated **API**, and Multipurpose API.
- **Application-Unaware:** the simulation code is unaware of the analysis and visualization incorporation. This category includes two subcategories: Inter-Position and Inspection.

InSt (Malakar et al., 2011), ParaView (Ayachit, 2015), and ADIOS (Liu et al., 2014) are Application-Aware in situ systems that fall into the Bespoke, Dedicated API, and Multipurpose API subcategories, respectively. On the other hand, ANSYS Enight (Frank and Krogh, 2012) and Freeprocessing (Fogal et al., 2014) are Application-Unaware systems that integrate the Inter-Position subcategory.



## Proximity

This axis describes the time or energy cost of accessing data. When thinking about this cost, it is relevant to regard the route that connects where data is stored to where it will be processed. Considering that this route can have a panoply of configurations, this axis is a continuous spectrum that contains the following categories:

- On Node;
- Off Node, Same Computing Resource;
- Distinct Computing Resource.

Some in situ systems, like ADIOS, fall into all the categories mentioned above, while others integrate a single category. For instance, Paraview only uses On Node proximity. Whereas SENSEI ([Ayachit et al., 2016](#)) fits into On and Off Node categories.

## Access

The access reflects how the analysis and visualization methods get access to the data generated by the simulation. It incorporates two categories:

- Direct: the simulation code and the visualization/analysis methods perform in the same logical memory space. This category comprises two subcategories: Shallow Copy and Deep Copy.
- Indirect: the simulation code and the visualization/analysis methods execute in a distinct logical memory space.

Freeprocessing and Ascent ([Larsen et al., 2017](#)) systems fit into the Direct category and the Shallow Copy subcategory. Additionally, Ascent also integrates the Deep Copy category. In contrast, the ANSYS Ensign system belongs to the Indirect category.

## Division of Execution

The division of execution specifies how are the computing resources partitioned between simulation and visualization/analysis. This axis contains two distinct categories:

- Space Division: the physical computing resources assigned to the simulation and the visualization/analysis are distinct, meaning that visualization/analysis methods have dedicated resources;

- Time Division: the simulation and the visualization/analysis share the same physical computing resources.

In terms of division of execution, ANSYS EnSight and Paraview are in situ systems that utilize a Space and Time Division, respectively.

## Operation Control

The operation control indicates the system that chooses which procedures are performed during run time. It comprises two categories:

- Automatic: users previously specify the analysis and visualization procedures that will be executed in situ. This category incorporates two subcategories: Adaptive and Non-Adaptive.
- Human-In-The-Loop: stakeholders select the analysis and visualization methods that will be performed in situ during the simulation's execution. This category also includes two subcategories: Blocking and Non-Blocking.

DataSpaces ([Docan et al., 2010](#)) and Ascent are Automatic systems that fit into the Adaptive and Non-Adaptive subcategories, respectively. However, Ascent also integrates the Human-In-Loop category along with ANSYS EnSight. Ascent and EnSight belong to the Blocking and Non-Blocking subcategories, respectively.

## Output Time

The output type specifies the kind of procedures that the visualization and analysis methods will execute on the data generated by the simulation. This axis holds three categories:

- Subset: the analysis and simulation methods select a portion of the simulation's data and discard the rest;
- Transform: the analysis and simulation methods alter every element of the simulation's data;
- Derived: the analysis and simulation methods produce new data that has a different type than the one generated by the simulation. This category comprises two subcategories: Fixed and Proportional.

For a more detailed explanation of this terminology and additional examples, consult the article written by [Childs et al. \(2020\)](#).

As aforementioned, the modifications made to LOOM will be tested on VADIS, an in situ system that already uses the framework. VADIS incorporates two sub-systems, FLOW and DISPER, but this thesis will focus on the latter. According to the presented terminology, DISPER can be classified as:

- Integration Type: Application-Aware with Dedicated API. The simulation uses libraries to perform the visualization. Simulation data is marshaled into those libraries through their APIs;
- Proximity: On Node;
- Access: Direct with Deep Copy. Visualization creates a copy of the data that comes from the simulation;
- Division of Execution: Time division;
- Operation Controls: Automatic and Adaptive. Visualization methods adapt their performance according to the simulation's execution;
- Output Time: Derived and Fixed. The size of the image rendered by the visualization process is independent of the input size.

## 2.2 Processing Unit Architecture

Over time, the decrease in the transistors' size has corroborated Moore's Law, which affirms that the number of transistors in a chip will approximately duplicate every two years ([Intel](#), [e](#)). Nevertheless, the speed increase on smaller transistors leads to greater power consumption and heat release. This rise led to the creation of the multi-core processor ([Geer, 2007](#)).

The multi-core processors, also known as **Chip Multiprocessors (CMP)**, integrate two or more "execution cores" in a single die. These "execution cores" are practically two separate processors with their batch of architectural and execution resources. These processors can even share a cache, depending on the used architecture ([Akhter and Roberts, 2006](#)).

The CMP can be combined with **Simultaneous Multi-Threading (SMT)**, which makes a single processor seem like multiple logical processors from a software perspective. With SMT, the applications and the operating systems can schedule several threads to a single processor as they would on multiprocessors ([Akhter and Roberts, 2006](#)). Figure 2 depicts a straightforward comparison of the single-core and multi-core architectures.

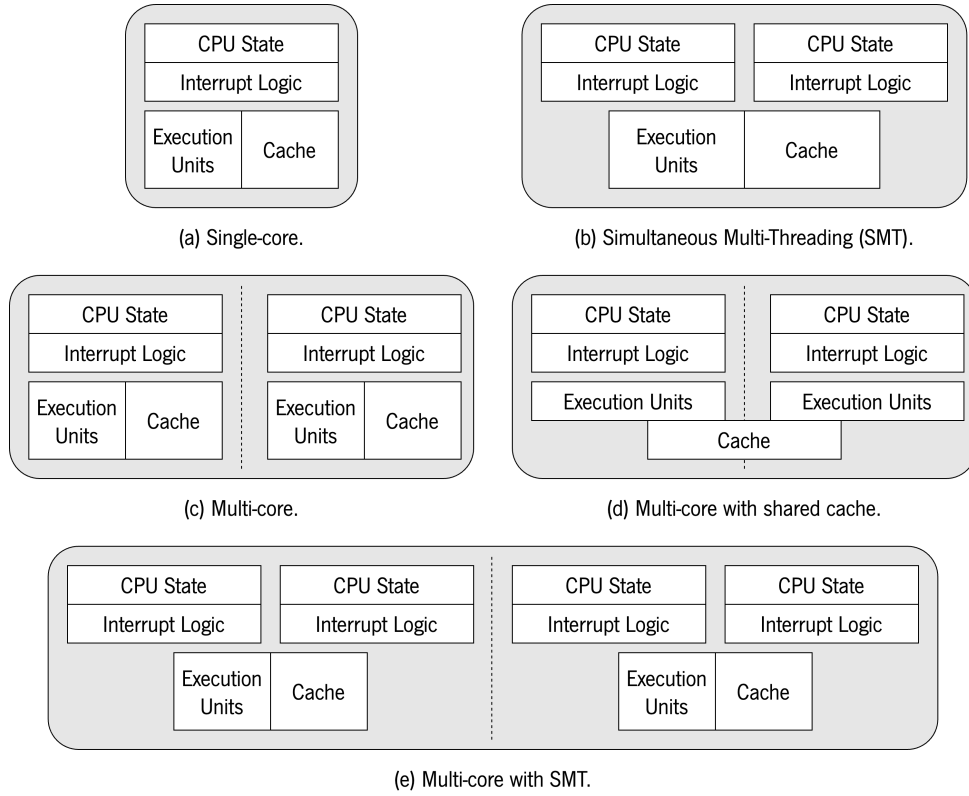


Figure 2: Comparison of the single-core and multi-core architectures.

Later, cluster computing integrated the multi-core processors since they presented greater computation power and a satisfactory cost-performance ratio (Chai et al., 2007). Figure X represents a multi-core cluster, i.e., a cluster composed of nodes with multi-core processors. The multi-core clusters usually own a hierarchical memory structure where the cores from a processor use the same L2 and L3 cache (Song et al., 2009), but they can also have separate caches (Hamid et al., 2014). In turn, the cores from different processors in the same node utilize the same **RAM** or **DRAM**, and the cores of distinct nodes do not share resources (Hamid et al., 2014). Figure 3 illustrates a multi-core cluster. Note that, on node 1, the cores of the same processor share the same memory resources. On node 0, they do not.

In a multi-core cluster, the interconnection network connects several computing nodes. This network is essential for providing scalability and efficiency to the applications since it deals with the networking requirements of each processor core (Dally and Towles, 2004). The multi-core cluster architecture usually inflicts more latency for communications between processors in distinct nodes (inter-CMP communication) than between processors in the same node (intra-CMP communication) (Hamid et al., 2014). This latency can drastically decrease the efficiency of the cluster system (Shainer et al., 2013).

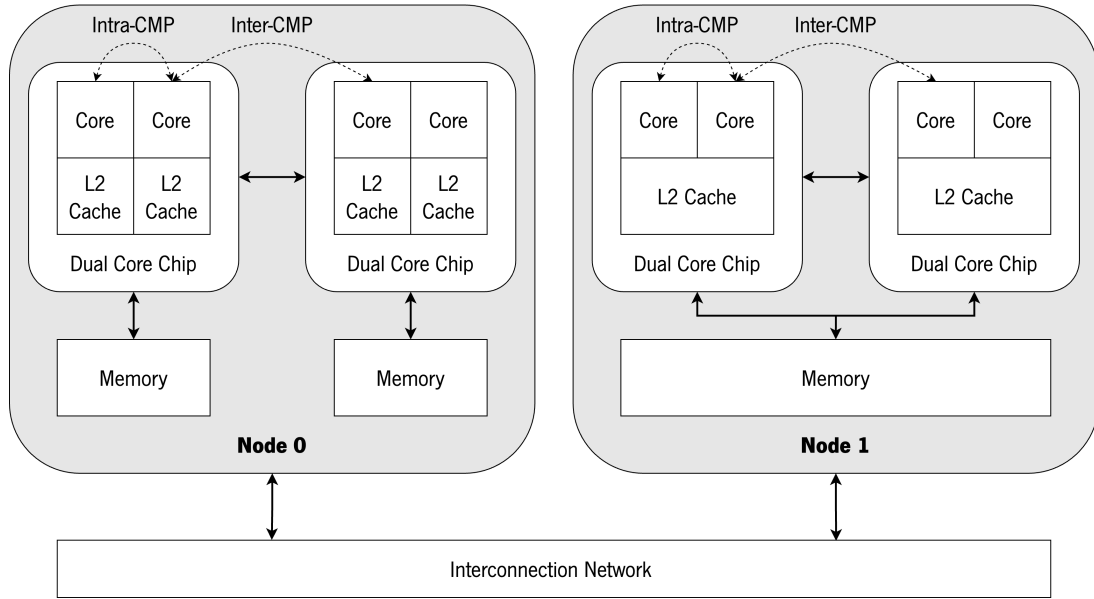


Figure 3: Multi-core cluster.

## 2.3 Parallel Rendering Strategies

The rendering pipeline is the process followed to create a two-dimensional image given three-dimensional objects, a virtual camera, light sources, material properties, and others. It consists of four stages, as presented in figure 4: application, geometry processing (vertex shading, projection, clipping, and screen mapping), rasterization (triangle setup and triangle traversal), and pixel processing (pixel shading and merging) (Akenine-Moller et al., 2018). Since the rendering procedure is infamously compute-intensive, especially in demanding applications, it has become essential to parallelize this pipeline to achieve high-performance graphics systems (Molnar et al., 1994).

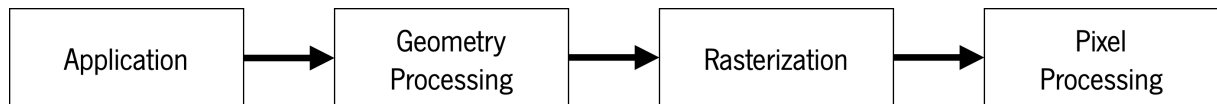


Figure 4: Rendering pipeline.

According to Molnar et al. (1994), parallel rendering, often used in scientific visualization applications, can be broadly categorized as sort-first, sort-middle, or sort-last. These three approaches distinguish themselves by how the program sorts and distributes the rendering work to process elements between the screen and data spaces. Sort-first and sort-middle divide the work among nodes by partitioning the screen space, while sort-last partitions the data space.

Let us take a closer look at each one of these strategies, starting with sort-first, represented in figure 5. Note that the figures that depict the sort-first, sort-middle, and sort-last approaches present a simplified version of the rendering pipeline. The graphics database is the product of the application phase, which corresponds to the "rendering primitives" (i.e., points, lines, and triangles) (Akenine-Moller et al., 2018).

The sort-first method distributes primitives during geometry processing to processes that can perform the rest of the rendering calculations. Hence, the application partitions the screen space into disjoint sections and assigns them to specific processors (the renderers). At first, primitives are distributed randomly among the renderers. Then, once the rendering process starts, the renderers calculate each primitive's section. If the section of a primitive is not the same as the section owned by the renderer, the primitive is sent to the renderer that detains the section where the primitive falls.

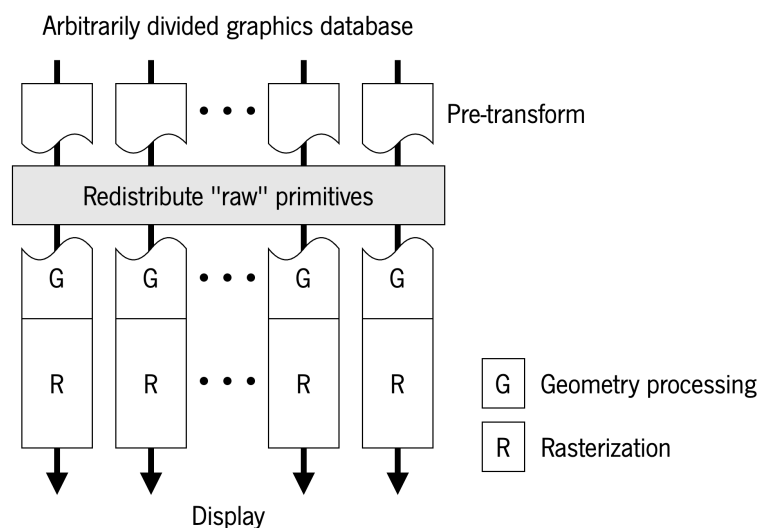


Figure 5: Rendering pipeline with sort-first.

The sort-middle approach (figure 6) redistributes the primitives between geometric processing and rasterization. At this stage, primitives are already converted into screen coordinates and are prepared to be rasterized.

In this strategy, subgroups of primitives are distributed randomly among the renderers, and a partition of the screen space is attributed to each rasterizer. Throughout each frame, renderers process their primitives and assign them to a partition of the screen space. After that, the renderers send the screen-space primitives to the corresponding rasterizers.

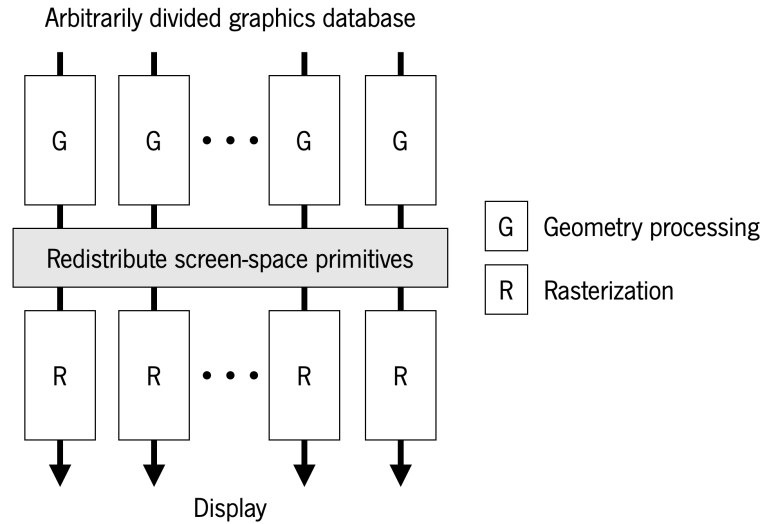


Figure 6: Rendering pipeline with sort-middle.

The sort-last method (figure 7) postpones the sorting process until the primitives are rasterized into pixels. Similarly to what happens in sort-middle, subgroups of primitives are distributed randomly among the renderers. These will determine the pixel values of their primitives, independently of the section they fall into. Afterward, the renderers send the calculated pixels to the "*compositing processors*" that will determine their visibility.

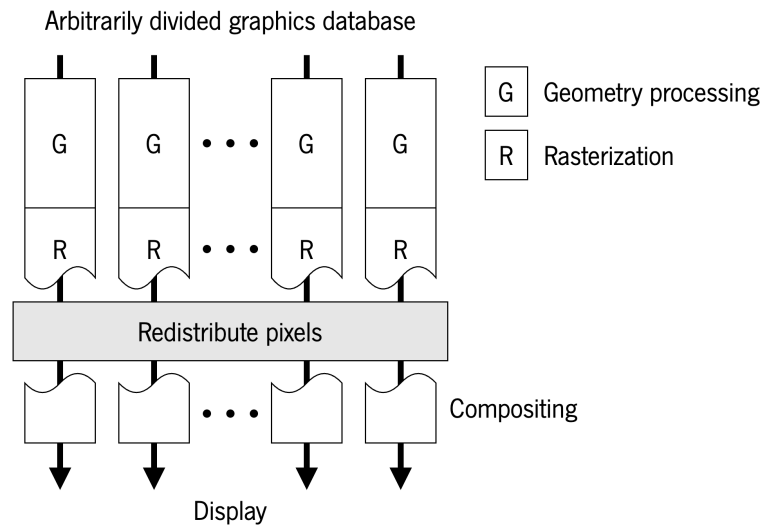


Figure 7: Rendering pipeline with sort-last.

In this thesis, it was necessary to craft two versions of VADIS to measure the impact of adding ray-forwarding to the visualization procedure: one without ray-forwarding (the baseline version) and another

that incorporates ray-forwarding.

On the one hand, the baseline version follows a sort-last approach where the data space is equally partitioned among the nodes. In this version, all the nodes generate and process the rays of the entire screen space. On the other hand, the ray-forwarding version is a hybrid of the sort-first and sort-last strategies where data and screen space are equally divided among the nodes. In this case, each node only generates and processes the rays corresponding to its partition of the screen space, demanding the forwarding of rays between nodes.

## 2.4 Intertwining Simulation with Visualization and Analysis

The traditional scheduling techniques used in some libraries for parallel processing, such as OpenMP ([Chandra et al., 2000](#)) and OneTBB ([Intel, c](#)), do not efficiently use the available resources. For instance, with OpenMP, developers specify the sections of the code that will perform in parallel. Consequently, when the main thread reaches one of these sections, it forks into multiple working threads that will divide the execution of that portion of the code. While the main thread performs sequential code, the workers idle and wait for the subsequent parallel section. In numerical simulation applications, these idle resources can be used to advance the analysis and visualization.

This section examines two frameworks that tackle this problem. LOOM and GoldRush use the simulation's idle resources to execute the analysis and the visualization methods. These have the ultimate goal of improving overall performance without affecting the simulation. LOOM and GoldRush are frameworks designed for single and multiple-node applications, respectively.

### 2.4.1 GoldRush

The principal focus of GoldRush ([Zheng et al., 2013](#)) is to utilize the simulation's idle times to perform in situ data analysis while interfering as little as possible with the simulation's execution. This framework's scheduling system boosts the performance of simulation programs by up to 42%.

GoldRush operates on both simulation and analysis processes, as depicted in blue in figure 9. For the simulation, it collects performance metrics to foresee the length of approaching idle times. When a predicted idle time is big enough to be utilized, the framework notifies the analysis processes so they continue their execution. At this point, the analysis processes operate on the cores granted by the simulation's OpenMP threads, and the head simulation threads maintain the execution on their dedicated cores. Once the idle time is over, GoldRush demands the analysis processes to pause their execution so that the



simulation's OpenMP threads recover exclusive access to their cores. A representation of the execution timeline with and without GoldRush is represented in figure 8.

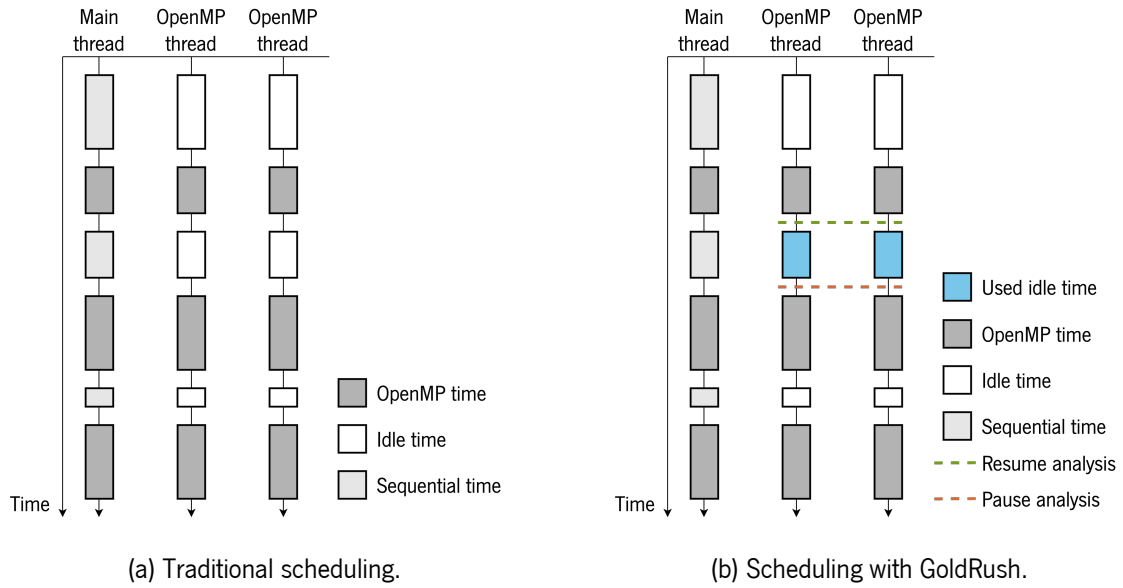


Figure 8: Scheduling with and without GoldRush.

Let us now inspect in more detail how the idle time prediction works. The framework saves every idle time's length and number of occurrences, plus uniquely labels each one with its start and end. Moreover, it keeps a history with the moving average of each idle time's length and number of occurrences. GoldRush uses this history to anticipate idle times as follows. At the start of an idle period, the framework searches the history for every idle time with the same start as the next idle time. Then, it chooses the one with the greatest number of occurrences and utilizes its average length as a prediction for the following idle time's length. If the predicted period is larger than a defined threshold or no history is discovered, the framework uses the subsequent idle time to continue the analysis. This heuristic has proved accurate for programs with regular execution flows by making correct predictions 88,7% to 100% of the time.

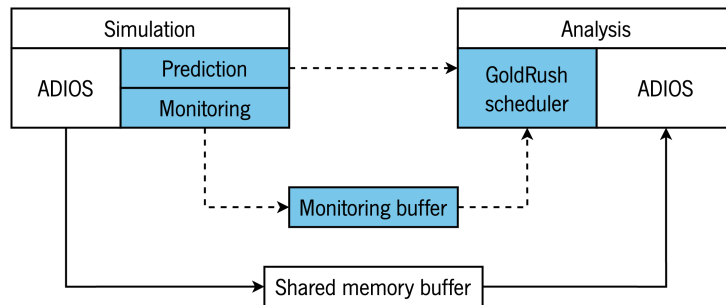
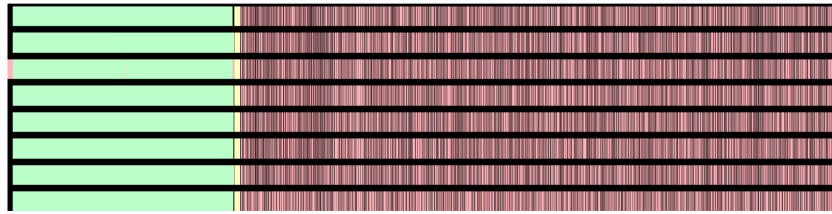


Figure 9: GoldRush's architecture.

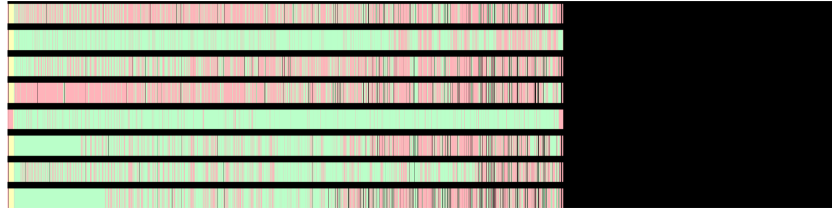
GoldRush may introduce possible overheads, for example, when monitoring performance, forecasting idle times, and pausing and restarting the analysis. Nevertheless, these are insignificant, allowing this framework's scheduler to outperform the OS scheduling system.

### 2.4.2 LOOM

LOOM ([Barbosa et al., 2021](#)) is a task-scheduling framework that enables tightly coupled in situ visualization without substantially increasing the simulation execution time by applying an “*interweaving*” method. This strategy executes the visualization tasks in the idle times of the simulation threads, which result from workload imbalances, to reduce the overall execution time (time to run both visualization and the simulation). Results show that LOOM can hide up to 2/3 of the visualization cost.



(a) Traditional in situ method



(b) LOOM method

Figure 10: Task timeline plot where simulation and visualization tasks are in red and green, respectively. Images from [Barbosa et al. \(2021\)](#).

The main goal of this framework is to intertwine the simulation and the visualization so that the obtained execution time is smaller than the traditional method time, where visualization runs once the simulation is over, and equal to the simulation time (equation 2.1).

$$T_s \leq T_{s||v} < T_{s+v} \quad (2.1)$$

$T_s$ : execution time of the simulation

$T_{s||v}$ : execution time of the “*interweaving*” approach

$T_{s+v}$ : execution time of the traditional method

Since one of the principal aims is to impair the simulation as little as possible, [Barbosa et al. \(2021\)](#) created a priority policy where simulation tasks have priority over the visualization ones. Consequently, they guarantee that the scheduler mainly provides simulation tasks between the I/O checkpoints and visualization tasks when there are idle simulation threads.

However, this policy may raise some issues. On one hand, if the workload of a visualization task is too big, it will detain the following scheduled simulation task. On the contrary, if the workload of a visualization task is too little, it will increment the scheduling overhead. To solve this, LOOM incorporates the "dicing" strategy ([Barbosa et al., n.d.](#); [Ribeiro et al., 2015](#)). This approach allows the scheduler to partition a visualization task to fit a particular time gap. It is up to the developer to create the method that defines the partitioning ("dice" method).

In the case of LOOM, the time gap is a predicted idle time between successive simulation tasks. Consequently, the scheduler executes the simulation timestep  $t + 1$  right after the simulation timestep  $t$  and tries to perform the visualization timestep  $t$  whenever threads idle in the simulation timestep  $t + 1$ . The threads' idle time and the execution time of the visualization tasks are estimated using a simple moving average.

Moreover, the framework includes three distinct queues to guarantee that tasks are executed at the best possible moment once all their dependencies are complete:

- **Global Not Ready Queue (GNRQ)**: includes tasks with unfulfilled dependencies. This queue is a **Directed Acyclic Graph (DAG)** to help keep track of the dependencies;
- **Global Ready Queue (GRQ)**: includes tasks with all their dependencies fulfilled;
- **Local Ready Queue (LRQ)**: present at each core, it is used to create a work-stealing system. Additionally, it reduces conflicts among cores when accessing GRQ. The cores utilize this queue as follows:
  - Transfer a batch of tasks from the GRQ to the empty LRQ;
  - If the GRQ does not have any tasks, steal tasks from the other cores;
  - Sleep while GRQ and LRQ are empty until a new task enters the GRQ.

LOOM assumes that simulation tasks are already efficiently decomposed, so only visualization tasks need to be "diced". For that, the framework defines the visualization tasks as "diceable" tasks. These

contain the "dice" method, enabling the scheduler to manipulate them to fit the simulation idle times.

Whenever a diceable task exits the GRQ, its "dice" method is called. If it is possible to "dice" the task to fit a predicted idle time, then the "dice" method returns a new task for that purpose and places it in the LRQ. The rest of the task will go back to the GRQ. If not, LOOM considers it cannot "dice" the task and sends it to the LRQ.

When comparing both frameworks, LOOM surpasses GoldRush in one focal aspect: GoldRush discards all the idle times it considers too small to be used. In contrast, LOOM "dices" the visualization tasks so they can fit the predicted idle times, resulting in better exploitation of the simulation's idle resources. Therefore, LOOM was the scheduling framework selected for the study case of this thesis.

## 2.5 Distributed Memory Ray Tracing

One of the most well-known rendering techniques is ray tracing. This method can realistically reproduce a scene's lightning by rendering physically precise phenomena, such as reflections, shadows, refractions, and indirect lightning. To do that, it traces the light's path by launching several rays from the camera. These rays go through the image plane and traverse the 3D scene until they reach a light source. During the traversal, the rays can be blocked by the scene's objects, reflect on a surface, or go through semi-transparent or transparent objects, leading to shadows, reflections, and refractions. The color of the image plane's pixels results from the combination of all these interactions ([Corporation, 2024](#)).

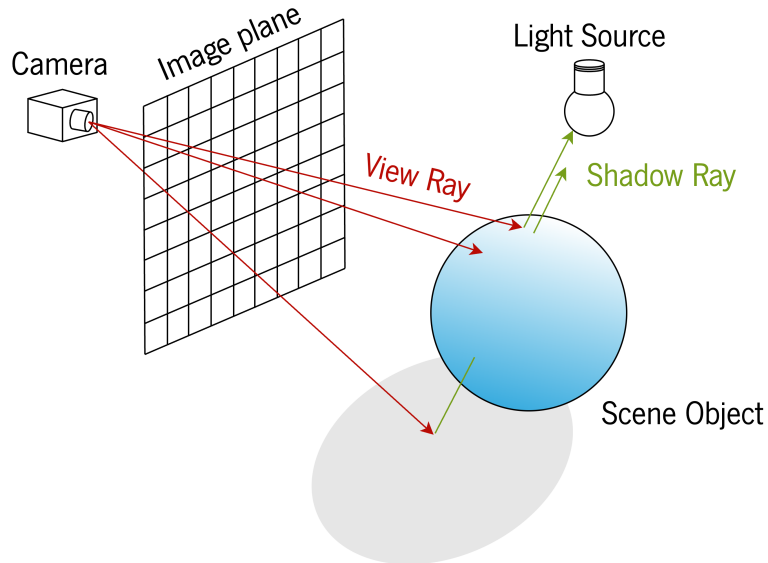


Figure 11: Ray Tracing Basics.

The fact that ray tracing can generate reliable, high-quality images that illustrate physically-based events makes it an alluring technique to visualize scientific data. Since ray tracing has a bad parallel reputation, it becomes a candidate to be used on distributed memory clusters to visualize large amounts of data ([Navrátil et al., 2012](#)).

This section dives into three approaches that handle distributed memory ray tracing. The first is a dynamic scheduling algorithm, whereas the other two (SpRay and Galaxy) are frameworks that can be incorporated into scientific visualization programs.

### 2.5.1 Dynamic Scheduling

[Navrátil et al. \(2012\)](#) created a distributed memory ray tracing strategy that utilizes a dynamic ray scheduler to increase memory locality while preserving load balance. This method can render a 650GB dataset using a cluster with 1024 nodes, surpassing approaches with static schedulers that could not even complete their execution. Besides that, it decreased by 10 to 48 times the data loads compared to static schedulers when using a smaller dataset.

To create a baseline for the performance, [Navrátil et al. \(2012\)](#) built two static schedulers that extend Pharr's ([Pharr et al., 1997](#)) approach to parallel ray tracing: one that statically decomposes the image plane (sort-first) and another that statically partitions the object domain (sort-last). In the serial ray tracing domain, [Pharr et al. \(1997\)](#) presented a strategy that permitted a more malleable scheduling of the ray intersection computations. Their method gathered data and rays into coherent groups (ray queues) to improve locality at the price of augmenting the memory state.

Let us now analyze the functioning of the implemented static schedulers, starting with the one that performs image plane decomposition. This scheduler starts by equally distributing rays among the processes according to a contiguous decomposition of the image plane. Then, each process loads data as ray computation demands and, at each step, chooses the domain that detains the most queued rays. This approach demonstrates good load balance and poor locality.

On the other hand, the domain decomposition scheduler partitions the dataset and distributes the portions among the processes. If the number of domains surpasses the number of processes, each process will be responsible for more than one domain. Otherwise, some processes will not have an assigned domain. During the rendering, processes load the data at first use and send the rays to the processes that own the data to calculate the traversal. At each step, processes choose their domain with the most queued rays. This method demonstrates good locality but poor load balance.

The dynamic scheduling strategy joins the advantages of the image plane and domain decompositions.

It starts by equally distributing rays according to the decomposition of the image plane. Some processes might share domains if there are too many rays in a specific domain. During the rendering, the scheduler switches to domain decomposition, and processes start to send rays to the processes that own the data needed to continue their traversal. After the ray exchange, processes choose their domain with the most queued rays. The processes that do not have any rays to compute in their loaded domains may be attributed to a new one.

Unlike the static algorithms, the dynamic scheduling method considers the rendering feedback to boost the scheduling coherence and to preserve a high usage of the available resources. The benefit of this approach is that it reorders rays and postpones calculations until the needed data is loaded.

This dynamic scheduling method will be one of the foundations of this work. The new version of LOOM will contain special primitives that ease the integration of this strategy's basis on scientific visualization programs. Furthermore, one of the goals of this thesis is to identify what would be necessary to try to automate load-balancing as presented by [Navrátil et al. \(2012\)](#). However, LOOM does not implement load-balancing. Consequently, on the simulation side, load-balancing is a user's responsibility and will depend on the used algorithm.

## 2.5.2 SpRay

SpRay ([Park et al., 2018](#)) introduces a speculative ray scheduler to process immense visualization data. It includes two versions, one for out-of-core rendering and another for in situ rendering. This method on ambient occlusion and path tracing proved to be up to 2.3 times faster than an adapted version of the dynamic scheduling algorithm LoadAnyOnce presented by [Navrátil et al. \(2014\)](#).

The out-of-core version uses a sort-first decomposition of the screen space, meaning that every node possesses all the scene's data and is responsible for rendering a portion of the image plane, called "*tile*". Since each node can generate its "*tile*" independently, communication between nodes only happens during the composition of the final image. Hence, there is no need to trade rays amongst nodes. Considering the study case of this thesis, the implementation of this approach will not be explored.

The in situ version uses a sort-last decomposition of the data space, meaning each node possesses a partition of the domain. Consequently, whenever a ray leaves the operation area of a node but remains in the dataset's range, it has to be dispatched to the node that holds the needed data to continue the traversal.

In this approach, each computing node follows the steps presented below:

1. Calculates the intersection of the input rays with the domains;

2. Speculatively assigns the rays to the hit domains, supposing they may or may not hit a domain;
3. Dispatches the speculative rays to the nodes that have the needed data;
4. Intersects the received rays with the primitives;
5. Caches the speculatively generated secondary rays locally, as well as the results of intersecting the received rays with the primitives.

Afterward, the nodes solve the speculations by executing a "*visibility compositing*". While there are still rays to process, the nodes jump to step 1, but they calculate the intersection of the solved secondary rays with the domains.

Let us now take a closer look at this strategy's implementation, presented in figure 12. Firstly, the program starts by allocating a partition of the image plane ("*tile*") to each computing node. Then, the nodes generate camera rays for their "*tile*", intersect the rays with the domains to separate them in ray set queues speculatively, and send the sets to the nodes that own the needed data to proceed with the traversal.

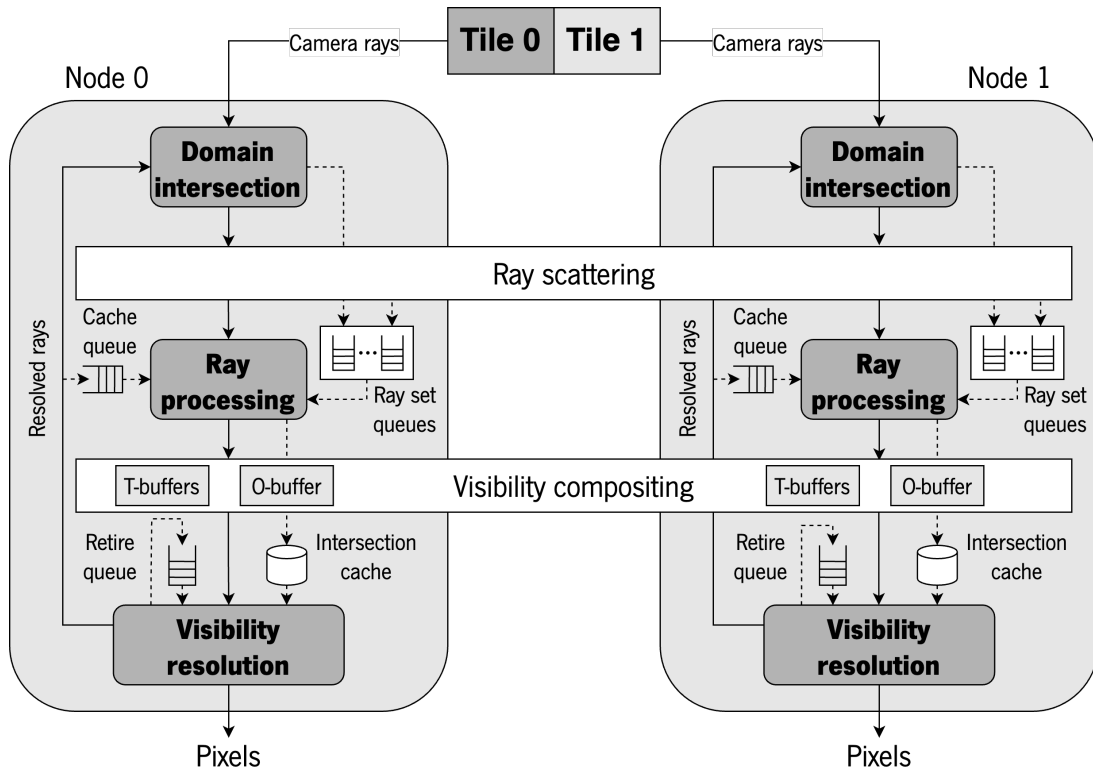


Figure 12: SpRay version for in situ rendering.

Once the ray communication is over, the nodes intersect the ray sets with the primitives. In cases

where this procedure generates secondary rays, the nodes immediately intersect the secondary rays with the primitives and save them with the test results in the "*intersection cache*". When all the rays are processed, the nodes synchronize to solve the gathered speculations.

Each node owns two "*distance buffers*" ("*t-buffers*") and one "*occlusion buffer*" ("*o-buffer*") to determine the visibility. They utilize one of the *t-buffers* to store the *t* values of the rays of the current iteration and the other to solve the visibility of the rays of the prior iteration.

After the compositing, the nodes solve the visibility of the rays stored on the "*intersection cache*". The rays analyzed during this stage are either radiance or shadow rays. If a radiance ray has an intersection, it is placed in a "*cache queue*" along with its hit information. Whenever a shadow ray is occluded, its correspondent position of the "*o-buffer*" is updated. If not, the node places the ray in a "*retire queue*" and forwards it to the other domains traversed by it. The node preserves the ray until its visibility is resolved in the succeeding iteration.

### 2.5.3 Galaxy

Galaxy (Abram et al., 2018) presents an asynchronous approach to tightly coupled visualization in distributed environments. This framework does not quite suit sort-first, sort-middle, or sort-last strategies, but it resembles sort-middle the most.

According to Abram et al. (2018), Galaxy distinguishes itself from the other recursive ray tracers of its time in three main characteristics:

- Efficiently communicates and renders rays by incorporating the best CPU-style ray traversal and intersection libraries where task scheduling is generalizable. Additionally, it includes a communication mechanism to achieve a scalable performance;
- Eliminates the costly barriers in image updates by executing the rendering process asynchronously;
- Takes advantage of a *subtractive lighting model* that rapidly generates a first image. If the image is static, it is later improved.

Let us now take a close look at its functioning. Galaxy has three layers: message processing, task scheduling, and task action execution. Below, a brief description of each one of these layers is presented.

#### Message Processing Layer

This layer, represented in figure 13, manages the communication between the Galaxy processes running in separate computing nodes using **MPI**. The message processing layer operates in a separate thread,



so it does not affect the workflow of the scheduling and execution layers.

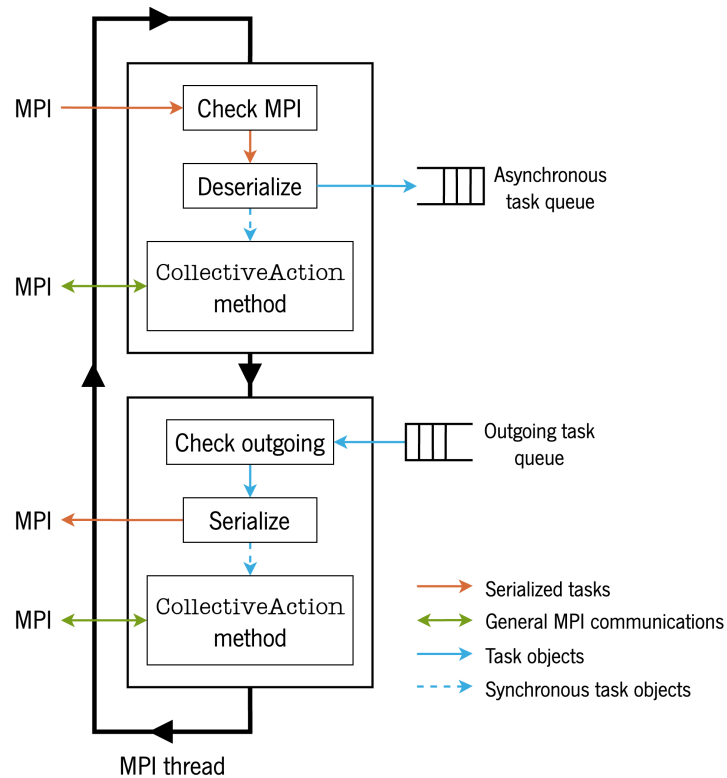


Figure 13: Message processing layer.

Whenever the message processing layer receives an incoming task from another node, it performs its deserialization and analysis. If the task is asynchronous, it is directed to the incoming task queue to be posteriorly scheduled by the scheduling thread. If the task is synchronous, its `CollectiveAction` method executes automatically in the communication thread.

Besides that, this layer tracks an outgoing task queue, which holds the tasks that need to travel to other nodes. Each task goes through a serialization process before being sent.

## Scheduling Layer

This layer, shown in figure 14, manages the incoming and outgoing task queues. Just like the message processing queue, it has an exclusive thread assigned to itself.

The scheduling layer assigns to the execution layer the execution of the `Action` method of each task in the incoming queue. If this method creates new tasks, they are sent to the scheduler and put in the outgoing queue. It is worth mentioning that Galaxy supports numerous task subclasses. These are processed together but trigger distinct actions because of the override of virtual methods, such as the `Action` method.

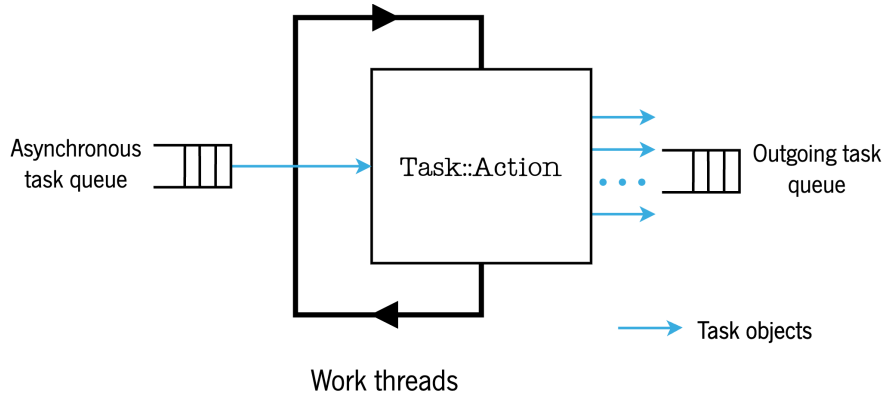


Figure 14: Task action execution layer.

### Task Action Execution Layer

As previously mentioned, this layer is responsible for executing the `Action` method of each task. The behavior of this method is defined by the user.

Since this thesis focuses on implementing a ray-forwarding technique, it is interesting to look at how [Abram et al. \(2018\)](#) implemented it. This process is represented in figure 15. Firstly, they built a `RayList` class that depicts a batch of rays that need to be locally processed. These lists are spatially coherent, which means that all the rays that belong to a list have to traverse through the local data of the same node. Note that `RayList` is a task subclass.

Each computing node has a local queue to store `RayList` objects. A `RayList` object is added to the queue whenever the task action layer executes its `Action` method. Then, a thread belonging to the thread pool is responsible for popping `RayList` objects from the queue and redirecting them to the libraries that will calculate their traversal in the node's local data.

A ray's traversal is interrupted whenever one of the following situations happens:

1. The ray collides with a surface;
2. The ray's color contribution is opaque;
3. The ray leaves the node's operation area.

Galaxy divides the rays that come across one of these scenarios into two groups: rays that have (scenarios 1 and 2) and have not (scenario 3) completed their traversal. The color contribution of the first set of rays is asynchronously added to the frame buffer. The second set of rays is organized into `RayList` objects according to the nodes that hold the data needed to complete their traversal. Posteriorly, the scheduler layer receives and adds the new `RayList` objects to the outgoing ray queue.

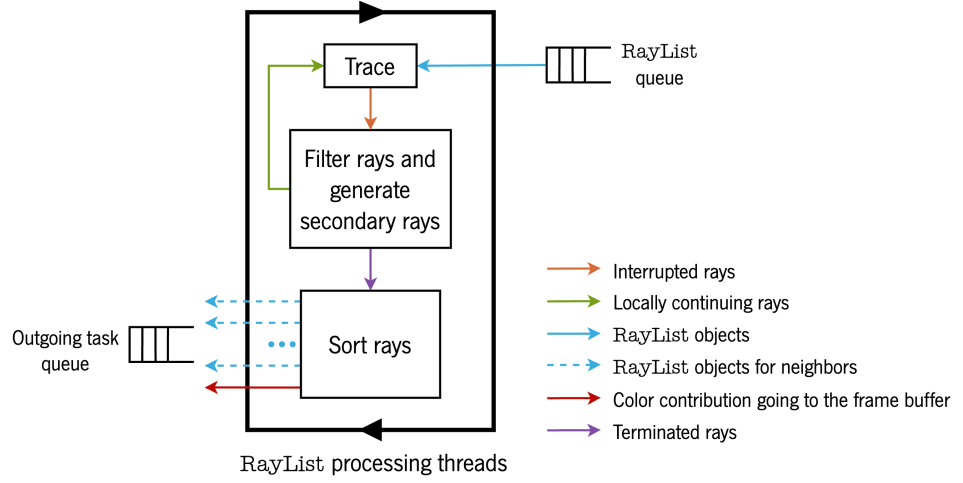


Figure 15: Ray list processing.

In this work, it is essential to have an efficient communication system to make LOOM capable of handling distributed workloads among several computing nodes. Therefore, the enhanced version of the framework will possess an asynchronous communication system inspired by the one present on Galaxy. This system is ideal for this purpose since it helps to maintain the philosophy behind LOOM.

## 2.6 VADIS

The modifications made to LOOM during this thesis will be tested on VADIS, a scientific visualization software that already takes advantage of the framework. VADIS will need to be adapted to exploit LOOM's new features, so it is relevant to explore its structure and functioning.

VADIS (pollutant DISpersion in the atmosphere under VArable wind conditions) is a **Computational Fluid Dynamic (CFD)** model conceived to simulate the dispersion of air pollutants on urban morphologies, developed by the University of Aveiro. It allows the definition of multiple obstacles and emission sources, along with time-varying emissions and flow fields (Martins and Borrego, 1998).

This model is composed of two distinct modules: FLOW and DISPER. The former estimates the wind, turbulent viscosity, pressure, turbulence, and temperature 3D fields using the numerical solution of the three-dimensional **Reynolds Averaged Navier-Stokes (RANS)** equation and the  $k - \epsilon$  turbulence closure. The latter utilizes the wind field calculated by FLOW to determine the 3D pollutant concentration based on the Lagrangian approach. This methodology considers that the pollutant's spatial and temporal dispersion is represented by randomly releasing a fair amount of numerical particles in the flow.

In figure 16, it is possible to observe the input and output data for both VADIS modules. FLOW receives

as input the position of obstacles (buildings and vegetation), the simulation domain, and the meteorological conditions (wind characteristics and air temperature). As mentioned above, its output is a 3D wind field that will serve as one of the DISPER's inputs.

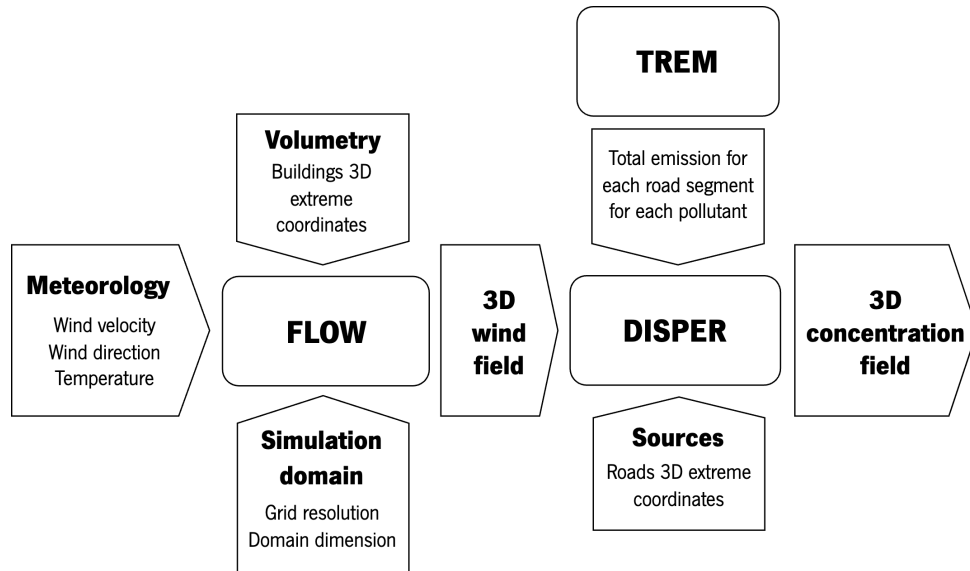


Figure 16: Input and output data of FLOW and DISPER.

Besides that, DISPER also receives the location of the pollution sources (roads) and the estimation of the road traffic emissions, calculated by TREM (Transport Emission Model for Line Sources). The user can define the obstacles and emission sources by indicating the coordinates of their inferior left and superior right corners and the angle of the obstacle/emission source with the cartographic grid. The DISPER's output is a 3D concentration field (Borrego et al., 2003). Additionally, it can optionally render an image of the urban area populated by the estimated pollution concentration.

Furthermore, VADIS incorporates the URban VEgetation (URVE) module. URVE determines the effects of vegetation on the dynamics of the flow and the dispersion patterns (Amorim et al., 2013).

The VADIS source code incorporates several libraries to ease the implementation, such as Intel Embree (Intel, 2023a), Intel Open VKL (Intel, 2023b), and LOOM. The first one, Embree, is used not only for geometry-image rendering but also to calculate particle collision with obstacles. The second one, Open-VKL, is utilized as a boost to the volume rendering performance. The last one, LOOM, is used as the parallelization framework.

## 2.7 Tools and Libraries

This section presents an overview of the tools and libraries used in this thesis. It is divided into two subsections: one dedicated to high-performance rendering libraries and another devoted to profiling tools.

### 2.7.1 High-Performance Rendering Libraries

#### Intel Embree

Intel Embree ([Intel, 2023a, 2021](#)) is a high-performance ray tracing library designed for developers who desire to optimize the performance of their photo-realistic rendering programs.

It is compatible with programs written with the Intel **SPMD** Program Compiler (ISPC) ([Intel, a](#)) by supplying an ISPC connection to the ray tracing algorithms. This feature enables the creation of renderers in ISPC that vectorize and use **SSE**, **AVX**, **AVX2**, and **AVX-512** instructions.

ISPC is a compiler for an alternative form of the C programming language with an add-on for "**Single Program, Multiple Data**" (SPMD). With SPMD, developers can write an apparent serial application that executes some "program instances" in parallel on the hardware ([Intel, a](#)).

Besides that, Embree supplies all the needed features for professional rendering, such as various primitive sorts, multi-level instancing, and motion blur. Additionally, it is an open-source library with an easy-to-use API.

#### Intel Open VKL

Intel Open Volume Kernel Library (Intel Open VKL) ([Intel, 2023b](#)) is a collection of high-performance kernels for volume computation. This library enhances volume rendering programs by employing a panoply of volumetric data structures, optimized traversals, and sampling algorithms.

Just like Embree, it supports SSE, AVX, AVX2, and AVX-512 instructions. Apart from that, it is also an open-source library.

Open VKL can handle multiple volume types, such as structured regular, structured spherical, VDB, **Adaptive Mesh Refinement (AMR)**, unstructured, and particle volumes. When it comes to algorithms, it supports point sampling (figure 17a), as well as hit and interval iterators (figures 17b and 17c, respectively).

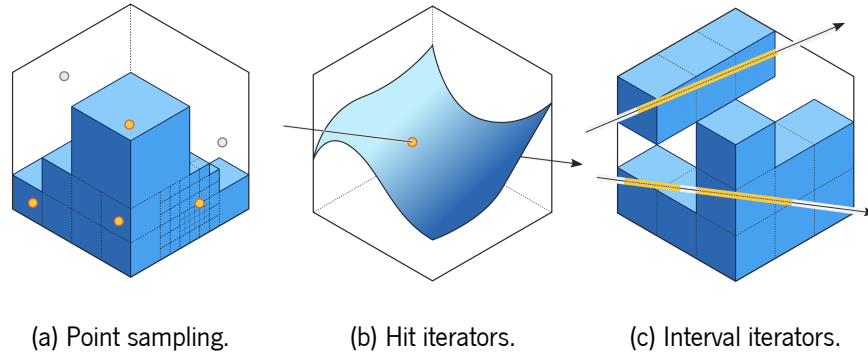


Figure 17: Open VKL algorithms. Images from [Intel \(2023b\)](#).

## Open MPI

Open MPI ([Kranzlmüller et al., 2004](#)) is an open-source implementation of the **Message Passing Interface (MPI)**, a standard for message trading in distributed environments. Open MPI is based on the LAM/MPI ([Squyres and Lumsdaine, 2003](#)), LA-MPI ([Graham et al., 2003](#)), and FT-MPI ([Fagg et al., 2003](#)) projects and is compatible with applications written in C, C++, and Fortran.

In this work, the enhanced version of LOOM uses Open MPI to enable efficient communication and synchronization between multiple processes running in distinct computing nodes.

## IceT

The Image Composition Engine for Tiles ([Technology and Engineering Solutions of Sandia, 2024](#); [Moreland, 2009](#)) is a high-performance library for sort-last parallel rendering. This library supplies fast rendering for standard displays and enables the generation of images for tiled displays.

In this thesis, IceT incorporates the final stage of the rendering process of the multi-node versions of VADIS. It composites the frame buffers of the multiple computational nodes using alpha blending, also known as alpha compositing.

## 2.7.2 Profiling Tools

### Intel Inspector

Intel Inspector ([Intel, b](#)) is a profiling tool that detects memory and non-deterministic threading errors. It was designed to debug C, C++, and Fortran programs on Windows and Linux operating systems.

In terms of threading errors, Inspector can discover deadlocks, cross-thread stack access errors, lock hierarchy violations, and data race conditions. In terms of memory errors, it can detect uninitialized mem-

ory read, memory leaks, illegal memory access, mismatched allocation and deallocation API, corruption, and inconsistent use of memory API. Besides that, this tool eases the identification of persistent memory errors.

## **Intel VTune**

Intel VTune ([Intel, d](#)) is a profiling tool that helps to locate and correct performance bottlenecks. It is compatible with applications written in C, C++, Fortran, Profile SYCL, OpenCL, Java, Google Go programming language, Assembly, .NET, or any mix of languages.

VTune presents numerous features, such as detecting the most time-consuming portions of the code (hotspots), diagnosing hardware problems that impact the application's performance, identifying bottlenecks of applications with large amounts of I/O, and describing large-scale MPI workloads performance.

## Chapter 3

# Adapting LOOM and VADIS

The main focus of this work is to analyze the LOOM's "*interweaving*" method ([Barbosa et al., 2021](#)) in a numeric simulation using a ray-forwarding technique to surmount the distributed memory obstacles. VADIS (pollutant DISpersion in the atmosphere under VArable wind conditions) was the chosen numeric simulation.

There was the need to create two versions to evaluate the impact of adding ray-forwarding to VADIS: one without ray-forwarding, which will be called the baseline version, and another that includes ray-forwarding. This chapter depicts the implementation of both versions, examining the steps taken to achieve each one and the encountered setbacks.

### 3.1 Baseline Version

This section explores the development of VADIS' baseline version, i.e., the one that does not include ray-forwarding. It starts by presenting the first multi-node version of VADIS. Then, it describes the transition from a synchronous to an asynchronous implementation, the communication system, and the voting mechanism. At last, it exhibits the asynchronous multi-node version of VADIS.

#### 3.1.1 Synchronous Multi-Node Version of VADIS

As explained in the previous chapter, VADIS is a scientific visualization software that calculates the dispersion of pollutants on urban morphologies composed of two modules: FLOW and DISPER. This thesis focuses on DISPER, the module responsible for calculating the 3D pollutant concentration field and for rendering an image of the urban area with the estimated pollution concentration.

DISPER can be broadly divided into three key actions: it generates particles based on the user-defined emission sources, moves the particles around the scene, and determines the pollutant concentration. The original version of DISPER executes these actions on a single node and uses LOOM as its parallelization



framework. Consequently, there was the need to create a multi-node version of DISPER to have a base for developing and testing the enhanced version of LOOM.

To do that, a sort-last strategy was used, where the data space is equally partitioned and distributed among the available processes. The user is the one who defines the partition using the command line option `-p x y z`, where  $x$ ,  $y$ , and  $z$  are the length, width, and height of the volume, respectively. The multiplication of  $x$ ,  $y$ , and  $z$  has to be equal to the number of processes so that the number of partitions is the same as the number of available resources. For example, with the partition `-p 2 1 2`, the length and height of the volume will be split in two, partitioning the data space into four subdivisions. Then, the program distributes these subdivisions among the four available processes.

After the data space distribution, processes load the data corresponding to their volume's portion and start the execution by generating and moving particles. For simplification purposes, let us consider that the operation area of a process is the subdivision of the data space attributed to the process. Whenever a particle leaves the process' operation area, the process cannot proceed with the traversal calculations because it does not own the necessary data to do so.

Consequently, the process must send the particle to the node with the needed data. Therefore, at the beginning of execution, processes determine their neighbors according to their operation area. Two processes are neighbors when their operation areas are close to each other. A process can have, at most, six distinct neighbors. These can be located on the right, left, top, bottom, back, and front. As a result, each process keeps six local queues in which they store the particles they need to send to each neighbor at the end of each iteration. These local queues help to reduce the number of communications between processes.

Note that, in this thesis, the Message Passing Interface is used to establish the communication between the multiple processes, each running on a different computing node. Besides, the term rank will be used as a synonym of process.

Let us now examine the algorithm constructed to move the particles, named `moveParticles`. In this method, each process:

1. Creates a local queue for each neighbor;
2. Calculates the movement of the particles present in its local queue. Anytime a particle escapes the process' operation area, it is placed in the queue correspondent to the neighbor to where it is headed. Since these calculations are parallel, each queue has an associated mutex to prevent data races. Besides, when a particle moves to another rank, a counter (`tosend`) is incremented. Later, the process uses this counter to find if particles are still moving around the scene;

3. Determines the number of particles it will send to each neighbor;
4. Sends the calculated number of particles to each neighbor;
5. Receives the number of particles that it will receive from each neighbor;
6. Waits until all the send and receive operations are fulfilled. This needs to be done since the non-blocking versions of send and receive were used;
7. Allocates the send and receive buffers;
8. Sends and receives particles. To send and receive particles, it uses `MPI_Ssend`<sup>1</sup> and `MPI_Irecv`, respectively;
9. Waits until it receives all the particles from the neighbors;
10. Stores all the received particles in the `received_particles` buffer;
11. Updates the start of the local queue (the beginning of the queue is equal to the end of the queue);
12. Inserts received particles in the local queue;
13. Updates the end of the queue (the end of the queue is equal to the size of the queue);
14. Clears the local queue of each neighbor;
15. Executes `MPI_Allreduce` on the `tosend` counter to find how many particles are still moving around the scene;
16. Jumps to step 2 if there are still particles moving or if it has received new particles to process. At the beginning of each iteration, the process sets the counter `tosend` to 0.

Once no more particles are moving, each rank determines the concentration of its operation area and renders a portion of the image corresponding to its partition of the data space.

### 3.1.2 Transition to the Asynchronous Multi-Node Version

In the created multi-node version of VADIS, processes establish synchronous communication. As explained in the previous section, in every iteration of `moveParticles`, each process has to wait for the number of particles it will receive from its neighbors. Besides that, it has to wait until its neighbors receive the

<sup>1</sup> In contrast to `MPI_Send`, `MPI_Ssend` postpones its return until the destination process calls the matching receive operation.

number of particles it will send to them. The same behavior happens when it comes to sending and receiving rays.

To better take advantage of the nodes, there was the need to transition from synchronous to asynchronous communication using LOOM. Consequently, nodes started to communicate with each other through LOOM. This subsection describes the steps carried out to make this transition.

Firstly, a wrapper for the MPI methods was created on LOOM. For that, it was necessary to abstract the MPI data types and requests. From this moment on, VADIS became unaware of the used communication protocol. Currently, it is the Message Passing Interface, but if it becomes more beneficial to use another protocol in the future, there is only the need to change LOOM. In this scenario, the programs that incorporate the framework can remain unchanged.

Secondly, two distinct communicators were defined: one to deal with the communication operations (send, receive, and reduction operations) and another to deal with the synchronization (`MPI_Barrier`). These originated from the duplication of the default MPI communicator (`MPI_COMM_WORLD`). The only thing that distinguishes the new communicators is the fact that they have distinct contexts. Contexts are separate environments for passing messages in MPI. Creating a new context is similar to adding a tag to differentiate messages. The use of multiple contexts does not allow "*pending*" communication or "*decoupled MPI activities*" of point-to-point procedures to affect collective communication inside a single communicator ([Forum, 2023](#)).

Then, an asynchronous communication system, similar to the one in Galaxy ([Abram et al., 2018](#)), was built. This communication system will be detailed in subsection 3.1.3. Besides that, IceT ([Technology and Engineering Solutions of Sandia, 2024](#); [Moreland, 2009](#)) was incorporated on VADIS. IceT is used to composite the frame buffer of each process using alpha blending. After the composition, a single process prints the result, generating the final image.

At this point, whenever a process received messages, LOOM would immediately trigger the actions associated with them. These actions started to be placed on LOOM's queues, just like the simulation and visualization tasks, to maintain the framework's workflow. Consequently, all the barriers on `moveParticles` had to be removed. This deletion led to the creation of a voting mechanism on LOOM that guarantees that all processes finished their work before moving on to the next step of the simulation. The voting mechanism is explored in subsection 3.1.4.

### 3.1.3 Communication System

As previously mentioned, an asynchronous communication system was implemented on LOOM (figure 18), similar to the one presented in Galaxy. This subsection explores it in detail. Each process owns a thread dedicated to the communication, which is responsible for receiving, decoding, and queueing messages. It is permanently on the lookout for messages using a matching probe operation (`MPI_Mprobe`). Just like `MPI_Probe`, `MPI_Mprobe` queries incoming messages without receiving them. Additionally, it presents a system that only receives messages with a specific tag regardless of other receive and probe operations (Forum, 2023). With the information collected by the probe, each process can determine the message size, allocate a buffer, and receive the message.

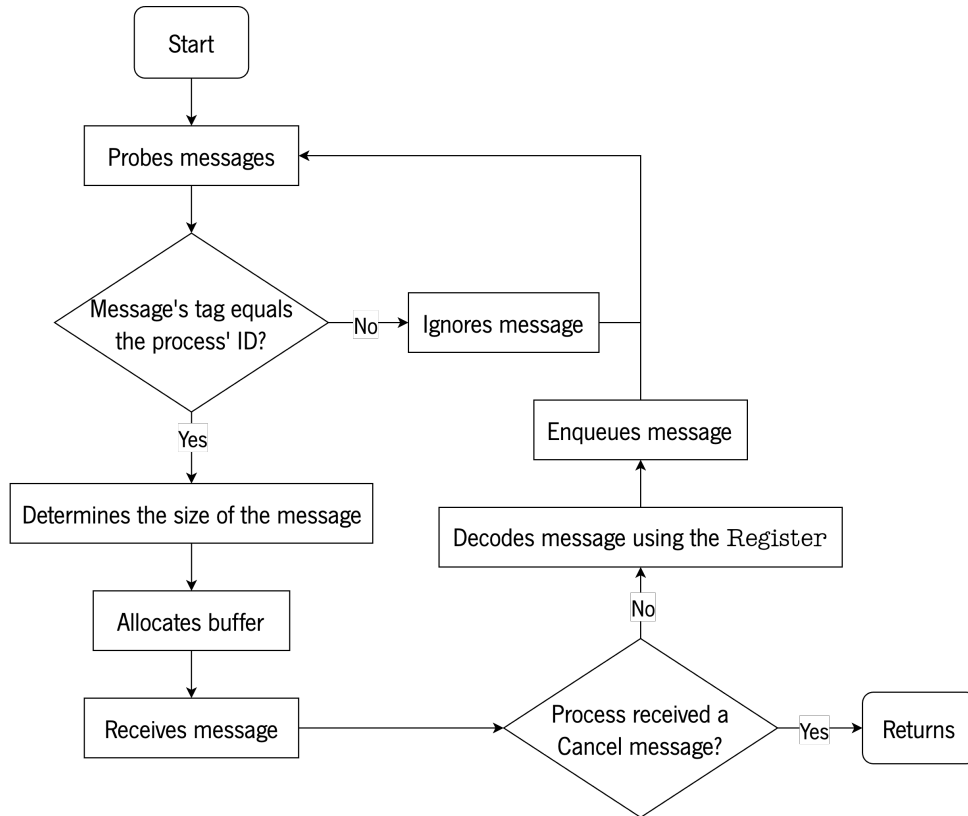


Figure 18: Flowchart of LOOM's asynchronous communication system.

The messages that travel in the communication system are serialized objects of the `Message` class. This class inherits the `Task` class already existing in LOOM. Thus, messages can be automatically added to LOOM's queues to be later executed. The `Message` class is an abstract class with a single attribute: the message type. Therefore, to create a new message type, it is necessary to create a subclass of the `Message` class that overrides the serialization, deserialization, and execution methods. Moreover, it can

include attributes if the message transports additional information besides the message type.

Since the communication thread is constantly looking out for messages, a message type was created to close the communication thread: `Cancel`. When the program's execution is over, a `Cancel` message is sent to the communication thread so it finishes its execution. This message type is immediately executed, unlike the other message types.

Whenever the communication thread receives a message, it decodes it and creates a message object using the `Register`. The `Register` is a map that corresponds each message type to its constructor, allowing the generic treatment of different message types. To correctly use this communication system, each message type must be registered at the beginning of the execution, and every message needs to be serialized before being sent.

### **3.1.4 Voting Mechanism**

Since multiple nodes are being used, there was a need to implement a voting mechanism on LOOM that assures that all processes have completed their work before proceeding to the following step of the simulation.

The developed voting mechanism is inspired by the two-phase commit protocol (2PC). This protocol guarantees data consistency among multiple nodes in distributed systems and is frequently used to synchronize and coordinate database transactions. As the name suggests, this protocol contains two phases. The first is the voting phase, where the coordinator asks the participants to vote. Their vote indicates if they can commit the transaction or not. The second is the commit phase, where the coordinator decides to commit or abort the transaction based on the participant's votes. If all the participants agree to commit the transaction, the coordinator decides to commit it. Otherwise, it decides to abort it. Afterward, the coordinator communicates his decision to the participants, and they trigger the necessary actions. This protocol is fault-intolerant, which means that it cannot recover from the failure of any of the involved processes. However, it can suit the study case of this thesis because if a process fails, the program cannot proceed with its execution.

To build the voting mechanism, a new message type was created to trigger the voting procedure, `Vote`, as well as an additional MPI communicator to deal with the voting mechanism's communication. Note that the `Vote` messages, like the `Cancel` messages, are immediately executed instead of being queued. Besides that, these are the only two message types defined on LOOM. It is up to the programs that take advantage of LOOM to create the message types needed for their execution.

Three versions of the voting mechanism were envisioned. The third was the only implemented version

since the others were unreliable. These approaches include a voting function that needs to be defined on the program that takes advantage of LOOM (in this study case, VADIS). This function returns the vote of a process at a certain point of the execution. If the process has finished all its work, the voting function outputs the value `true`. Otherwise, it returns `false`. Let us now analyze the three versions of the voting mechanism.

## Voting Mechanism I

Every time a process reaches the end of an iteration, it launches a thread that will initiate a new voting session. Let us name it the voting thread. The voting thread sends a `Vote` message to every process and calls the voting function. The boolean returned by the voting function is converted into an integer. Then, the voting thread executes `MPI_Allreduce` to perform the sum of the votes of every process. If the sum equals the number of processes, all the processes have finished their work. Otherwise, they did not.

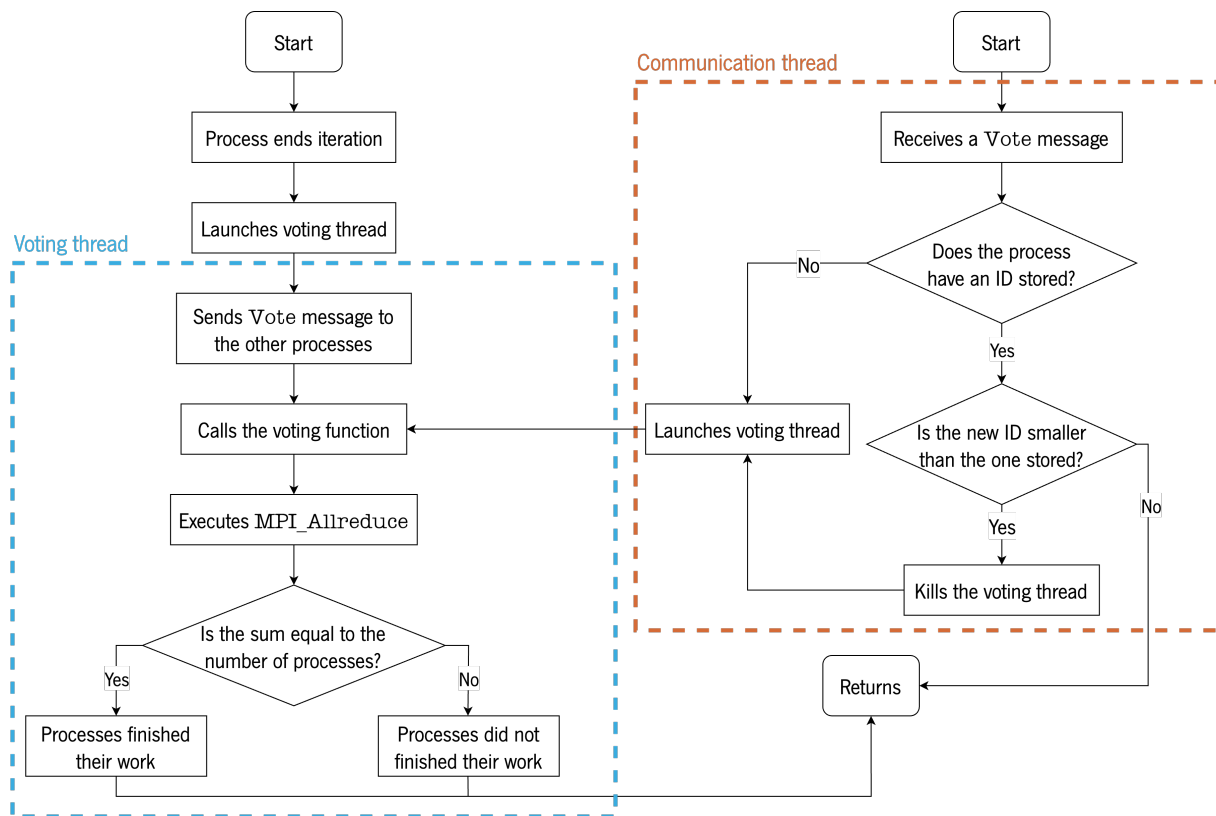


Figure 19: Flowchart of voting mechanism I.

When a process receives a `Vote` message, it stores the ID of the process that sent it and launches a voting thread. If a process ID is already stored, two scenarios can happen: (1) If the new ID is smaller than the one stored, the communication thread kills the voting thread and creates a new one; (2) Other-

wise, nothing happens. LOOM uses this behavior to establish a priority policy that prioritizes smaller IDs. This mechanism, presented in figure 19, was unreliable since a thread can only terminate its execution. Therefore, there was the need to notify the voting thread so it could end its execution.

## Voting Mechanism II

The second voting mechanism, depicted in figure 20, is very similar to the first one. However, instead of using MPI\_Allreduce, it utilizes its non-blocking version: MPI\_Iallreduce.

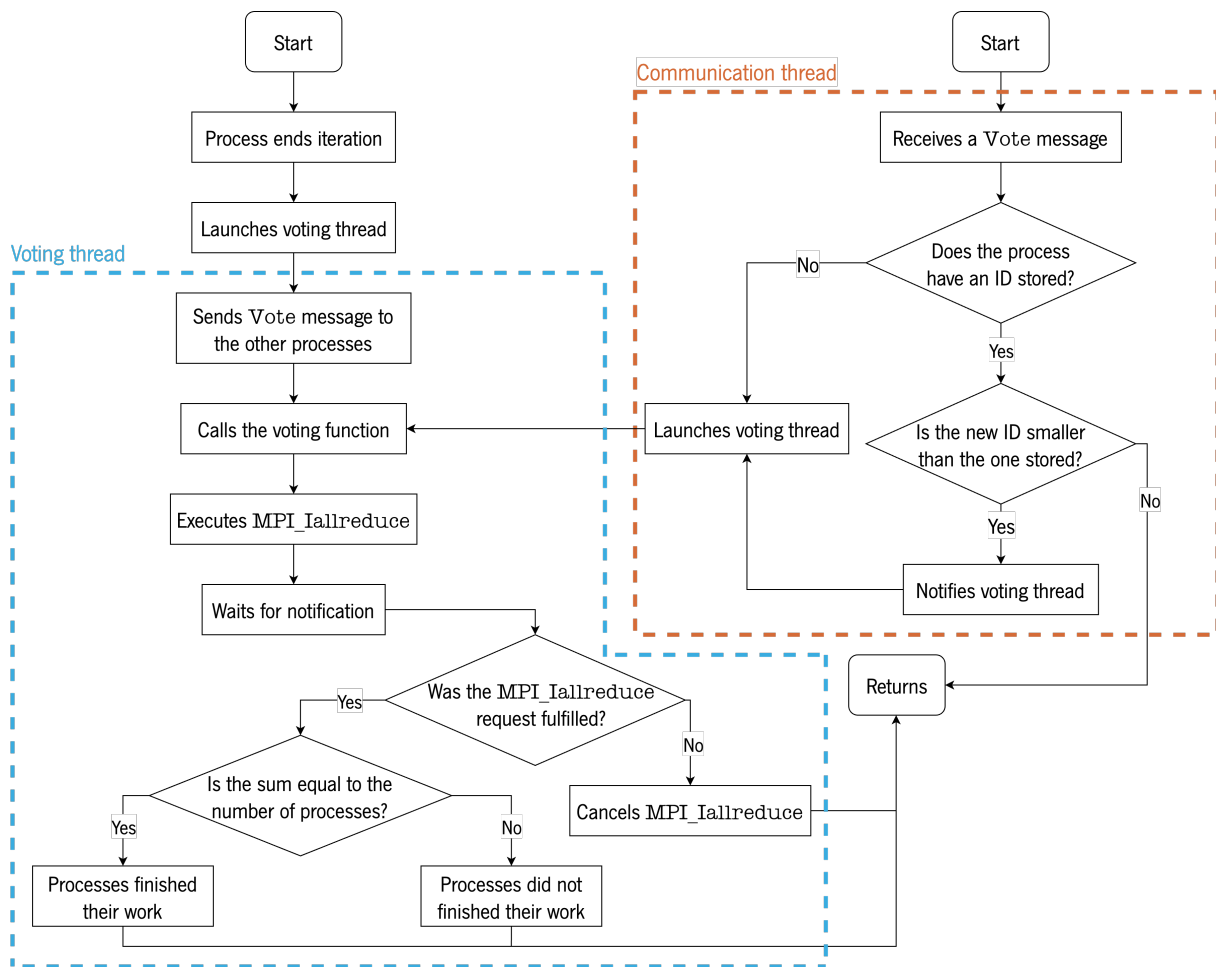


Figure 20: Flowchart of voting mechanism II.

When a process receives a Vote message, it launches a voting thread that will call the voting function and MPI\_Iallreduce. Then, the voting thread waits until one of two conditions is true: (1) the request of MPI\_Iallreduce was complete, or (2) the communication thread has received a Vote message from a process with a smaller ID than the one stored. In the second case, to follow the priority policy, the voting thread cancels the MPI\_Iallreduce, using MPI\_Cancel, and ceases its execution. Then, the

communication thread can create a new voting thread. Like the first voting mechanism, this approach was unreliable since only send and receive operations can be canceled ([Forum, 2023](#)).

### Voting Mechanism III

The third voting mechanism, represented in figure 21, follows a slightly different approach. When a process finishes an iteration, it does not create a new voting thread. Instead, it immediately sends a Vote message to the other processes, executes the voting function, and calls MPI\_Allreduce. When a rank receives a Vote message, it launches a new thread that calls the voting function and MPI\_Allreduce.

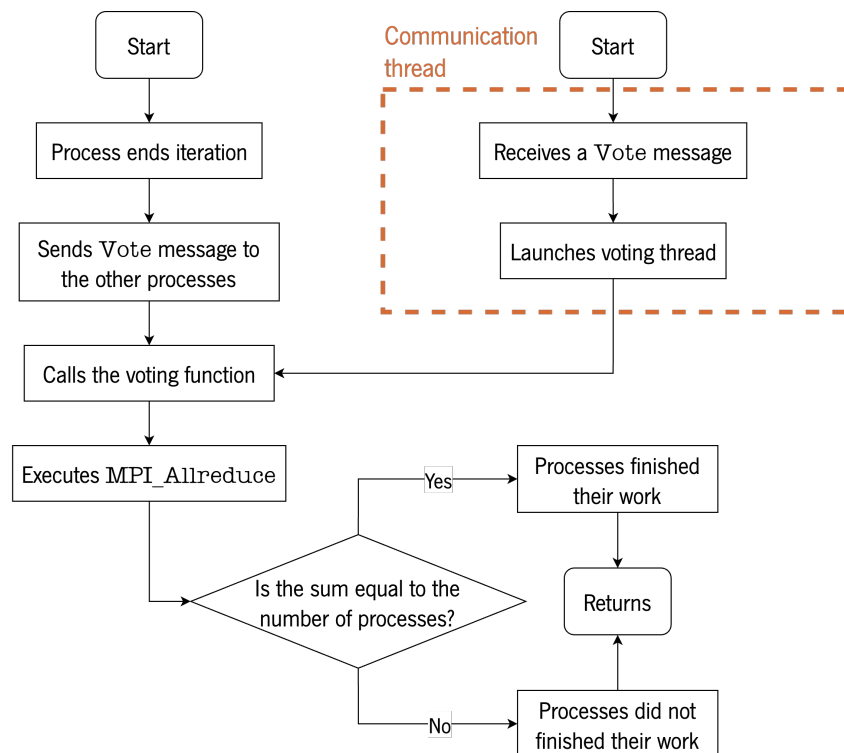


Figure 21: Flowchart of voting mechanism III.

However, this implementation does not work when several Vote messages are sent to processes simultaneously. Therefore, it was essential to have a system that could order the received messages. Vector clocks, for example, is an algorithm that can partially order events on distributed systems.

Since there was no opportunity to implement this algorithm, a workaround was found, presented in figure 22. Instead of evoking a voting session by sending a Vote message to the other processes, each process waits for the others to reach the end of the iteration using an MPI\_Barrier. Once all processes reach this barrier, they execute the vote function and call MPI\_Allreduce. The MPI\_Allreduce will sum the votes of every rank, and if the sum equals the number of processes, all the processes have



concluded their work.

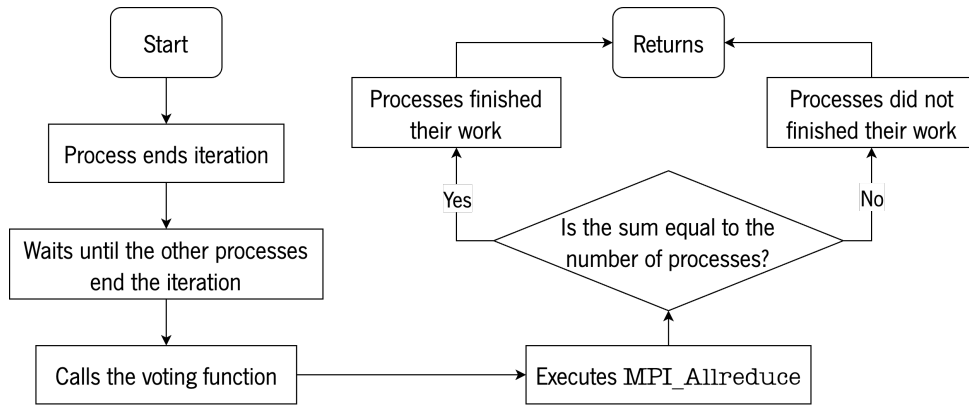


Figure 22: Flowchart of the workaround found to voting mechanism III.

### 3.1.5 Asynchronous Multi-Node Version of VADIS

Before analyzing the asynchronous multi-node version of VADIS, let us begin by exploring the message types created to support it. Firstly, there was the need to construct a message type on VADIS to transfer particles among processes: `Particles`. It includes the message type, the ID of the process that generated it, and a vector of particles. When a process receives a `Particles` message, it inserts the received particles on the `received_particles` buffer and dispatches an acknowledgment message to the process that sent the message.

During the development of this thesis, there was the need to use multiple acknowledgment messages for different purposes. Thus, instead of creating a new message type for each aim, a generic acknowledgment message was implemented: `Ack`. It includes the message and acknowledgment types. When a process receives an acknowledgment message, a specific action is triggered according to the acknowledgment type. For example, when a process receives an `Ack` message with the type 0, it decreases the counter `acks_count`. The purpose of this counter is to keep track of the dispatched messages. Each time a process sends particles to one of its neighbors, it increases `acks_count`. If the counter equals zero, all the sent messages were delivered. Otherwise, they were not.

Let us now explore the asynchronous multi-node version of VADIS. The only aspect that distinguishes the synchronous and the asynchronous versions is the `moveParticles` method. In the asynchronous version, each process:

1. Creates a local queue for each neighbor;
2. Calculates the movement of the particles placed in its local queue and buffers the particles that

- move to another neighbor's operation area;
3. Sends the particles stored in the local queue of each neighbor to the correspondent process using the LOOM's communication system;
  4. Updates the start of the local queue;
  5. Inserts the received particles on the local queue and clears the `received_particles` buffer;
  6. Updates the end of the local queue;
  7. Clears the local queue of each neighbor;
  8. Performs the voting mechanism;
  9. Jumps to step 2 if at least one of the processes has not finished its work.

In this case, a process has finished all its work and votes true when the following conditions are verified:

- The process does not have particles to send, i.e., it does not have any acknowledgments of the type 0 to receive;
- The process does not have any particles to move, i.e., its local queue is empty;
- The process does not have particles to receive, i.e., the `received_particles` buffer is empty.

## 3.2 Ray-Forwarding Version

As previously mentioned, the baseline version follows a sort-last approach, where the data space is distributed evenly among the available resources. In this method, each process generates and processes the rays of every pixel that compose the final image. Then, the contribution of all the processes is composed using alpha blending, also known as alpha compositing, using the Image Composition Engine for Tiles (IceT).

In contrast, the ray-forwarding version is a sort-last and sort-first hybrid, where data and screen space are divided among the available resources. In this strategy, processes will only generate the rays corresponding to their portion of the screen space. Consequently, when a ray leaves the operation area of a process, it is forwarded to the process that owns the data needed to proceed with the traversal.

This section explores the implemented ray-forwarding strategies and the adaptation of the dicing method to fit this case. The two ray-forwarding approaches complement the baseline method, meaning only the visualization procedure was modified. Let us begin by describing the first ray-forwarding implementation.

### 3.2.1 First Ray-Forwarding Version

What distinguishes both ray-forwarding versions is the place where the color accumulation occurs. Therefore, most of the steps presented below are common to both approaches. In the first version, represented in figure 23, the color accumulation of each ray happens in the process where it finishes the traversal.

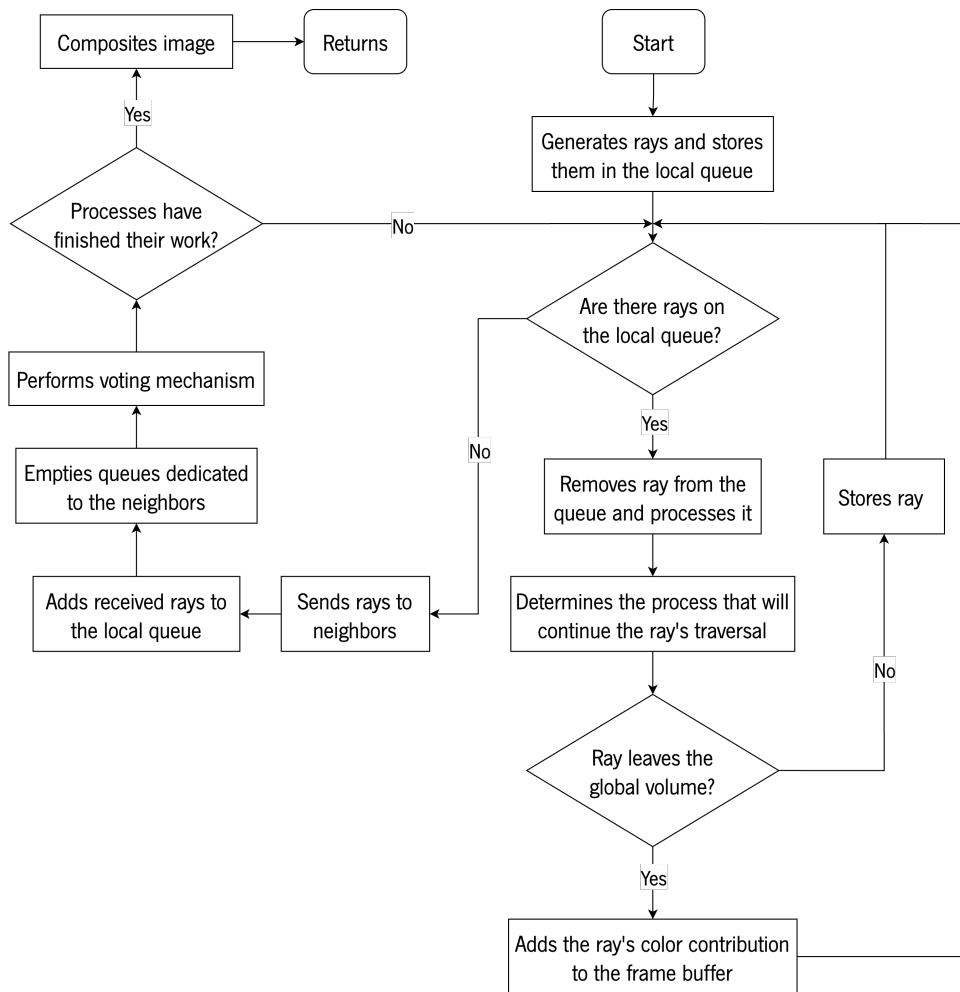


Figure 23: Flowchart of the rendering procedure on the first ray-forwarding version.

Firstly, VADIS divides the data and image spaces among the available processes. The division of the image space occurs according to the partitions defined by the user on the command line, just like the division of the object space. Then, each process shares its bounding box with the other processes. These

will posteriorly use this information to determine the process that will continue the traversal of the rays that leave their operation area. To enable the trade of bounding boxes, a new message type was created: `BoundingBox`. Besides the message type, this message contains a bounding box and the ID of the process that sent it. After dispatching its bounding box, a process waits until it has received the bounding boxes of the other processes. Therefore, there was no need to design an acknowledgment message for this case. When a process receives a `BoundingBox` message, it stores the bounding box and labels it with the ID of the process that sent it.

During the rendering stage, each process starts by generating the rays corresponding to its portion of the image space. These are stored on a local queue as a `MovingRay`, a structure that contains all the information needed during the ray-forwarding. This structure includes the ray's origin, direction, color contribution, and hit. Besides that, it contains the image plane coordinates where the ray was generated.

After the ray generation, processes begin by calculating the traversal of each ray. Whenever a ray leaves the operation area of a process, two scenarios can happen: (1) the ray moves to another's process operation area, or (2) the ray exits the global volume. Consequently, processes determine which scenario happened by intersecting the ray with the stored bounding boxes. If they find an intersection, then they are upon the first scenario. Otherwise, they are upon the second scenario.

In the first scenario, processes store the ray so they can posteriorly send it to the process that owns the needed data to proceed with the traversal, which corresponds to the one with the bounding box that registers the smaller  $t_{near}$  when intersected with the ray. Just as it happens on `moveParticles`, processes have a local queue dedicated to each neighbor. In the second scenario, the process adds the ray's color contribution to its frame buffer.

When the local queue of a process is empty, it sends the stored rays to its neighbors. Each time a process sends rays to one of its neighbors, it increases the counter `ray_acks`. This procedure is similar to the one followed in the particle's exchange. To enable the exchange of rays among processes, a new message type was crafted: `MovingRays`. It includes the message type, the ID of the process that sent it, and a vector of `MovingRay` objects. Besides that, the acknowledgment type 1 was created.

When a process receives a `MovingRays` message, it stores the received rays on the `received_rays` buffer and sends a type 1 Ack message to the process that sent the rays. When a process receives an Ack message of type 1, it decreases the `ray_ack` counter. Once all the rays have been sent, each process updates the start of its local queue, adds the received rays to its local queue, clears the `received_rays` buffer, updates the end of its local queue, and empties the queue dedicated to each neighbor.

After this, each process performs the voting mechanism. This mechanism is the same as the one

executed on `moveParticles`. The difference is the conditions that evaluate if a process has finished its work. In this case, a process has completed its work and votes true when the following conditions are verified:

- The process does not have rays to send, i.e., it does not have any acknowledgments of the type 1 to receive;
- The process does not have rays to traverse, i.e., its local queue is empty;
- The process does not have rays to receive, i.e., the `received_rays` buffer is empty.

If a process has not finished, the processes repeat this procedure from the point where they calculate the traversal of the rays on their local queues. Once all processes complete their work, they composite the final image using `IceT`.

This approach worked perfectly with two processes. However, when more than two processes were involved, the program would not finish its execution. After debugging the application, it was discovered that the processes' bounding boxes were overlapping. Consequently, rays were traveling back and forth between different processes.

Therefore, an attribute was added to the `MovingRay` structure that indicates the processes already visited by the ray. It is a boolean array where each index corresponds to a process (index 0 corresponds to process 0, index 1 to process 1, et cetera). If a ray has visited a process, the corresponding index has the value `true`. Otherwise, it has the value `false`.

With this new attribute, the process that continues the traversal of a ray that moved outside of another's process operation area is the one that (1) owns the bounding box that registers the smaller `t_near` when intersected with the ray and (2) was not visited by the ray.

### 3.2.2 Second Ray-Forwarding Version

As already mentioned, the only difference between the two ray-forwarding versions is the place where the color accumulation occurs. In the previous version, the color accumulation of each ray happens in the process where it finishes the traversal. In turn, in this version, depicted in figure 24, the color accumulation of each ray occurs in the process that generated it.

Let us now examine the concrete differences between the two approaches. The aspects that are not mentioned in this section remain unchanged. Firstly, each process, besides calculating its partition of the data and screen spaces, also determines the starting and ending pixels of the other processes' screen space. They will posteriorly use this information to know the process that generated each ray.

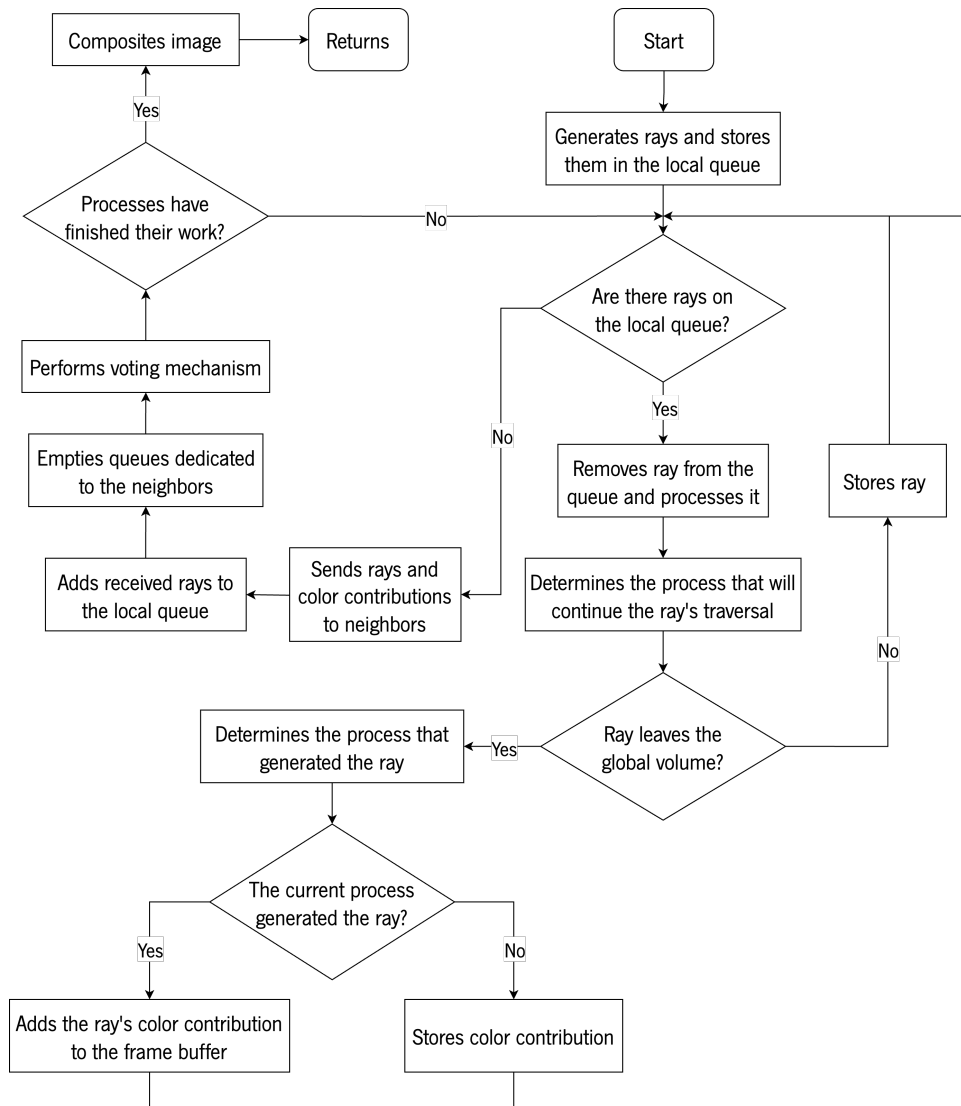


Figure 24: Flowchart of the rendering procedure on the second ray-forwarding version.

When a ray abandons the global volume, processes verify which process generated the ray instead of immediately adding its color contribution to the frame buffer. If the ray was generated by the current process, it adds the ray's contribution to the frame buffer. Otherwise, the process stores the ray's contribution to send it later to the corresponding process. For this purpose, new local queues dedicated to each neighbor were created. The processes store the color contributions in these queues as a `Color`, a structure that contains the ray's color contribution and pixel ID.

Once the process' local queue is empty, the process, besides sending the queue rays, sends the color contributions and increments the counter `color_acks`. For that to happen, a new message type was crafted: `Colors`. It contains the message type, the ID of the process that sent the message, and a vector of `Color` objects. When a process receives a `Colors` message, it adds the received color contributions

to the frame buffer and sends an Ack message of type 2. When a process receives an acknowledgment message of type 2, it decreases the `color_acks` counter.

Additionally, a new condition is added to the voting mechanism. A process has finished all its work when it also fulfills the following condition: the process does not have any color contributions to send, i.e., it does not have any acknowledgments of type 2 to receive.

### 3.2.3 Dicing of the Visualisation Tasks

Firstly, it was considered that the visualization tasks were not "*diceble*" to simplify the ray-forwarding implementation. Then, with the ray-forwarding implemented, it was time to modify the visualization tasks to fit the changes made to the code.

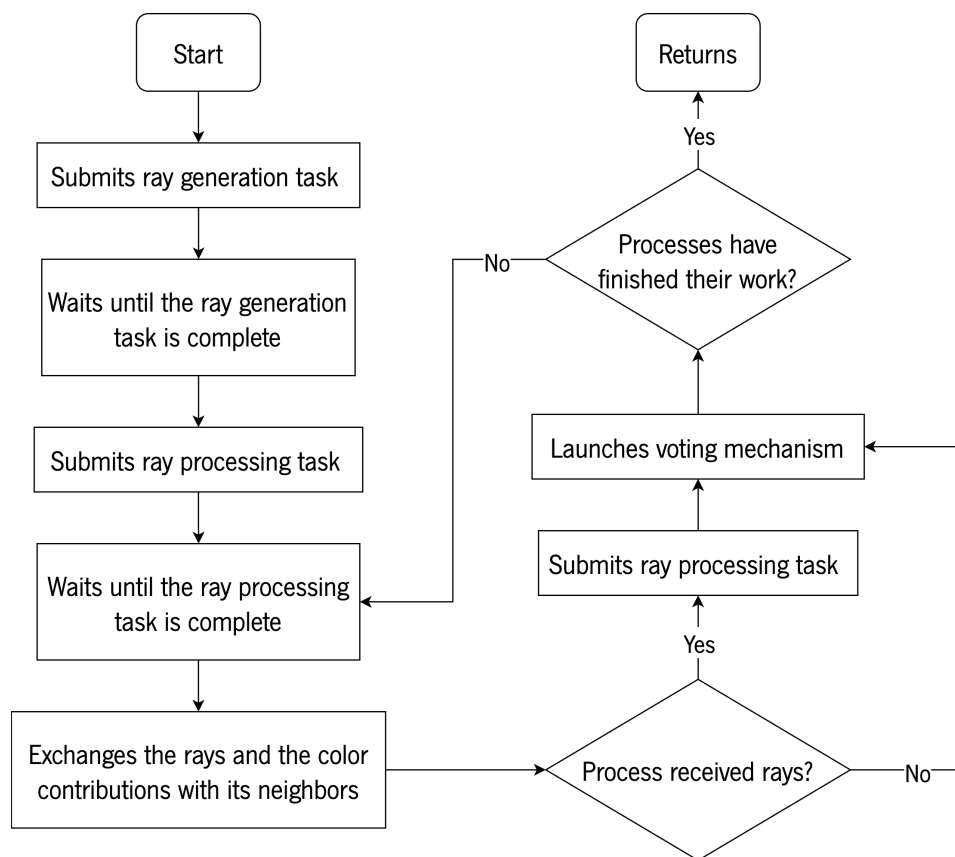


Figure 25: Flowchart of the task *dicing* on the ray-forwarding versions.

In the baseline version, VADIS had a single "*diceble*" task dedicated to the visualization. This task is responsible for generating and processing rays. Its "*dice*" method subdivides the task to fit the simulation's idle times based on the pixel number. In the ray-forwarding version, this approach was unreliable, considering the algorithm's structure. To enable the dicing of the visualization tasks, the task dedicated to

the visualization was divided into two "*diceble*" tasks: one for generating rays and another for processing rays. The "*dice*" method of each one of these tasks is based on the pixel number and the index of the rays on the process' local queue, respectively.

Let us look at how these tasks are handled during the rendering procedure (figure 25). Each process submits a ray generation task to generate all the rays corresponding to its operation area. Once the ray generation is complete, the processes submit a ray processing task to process the previously generated rays. When the ray processing is over, the process exchanges the rays and the color contributions with its neighbors. If the process receives new rays, it launches another ray processing task. Otherwise, nothing happens. Then, it starts the voting mechanism. If any process has not finished its work, the process repeats the procedure. It waits for the ray processing task to be over, exchanges the rays and the color contributions, et cetera.



## Chapter 4

# LOOM's Performance Assessment

This chapter contains two sections. The first one presents the methodology used during the testing procedure. The second section dives into the performance evaluation of both versions of VADIS under different scenarios.

### 4.1 Methodology

The two multi-node versions of VADIS were tested on Bob, a 600-node cluster hosted at the Minho Advanced Computing Center (MACC). Each node accommodates two eight-core Xeon E5-2680 processors at 2.7 GHz and 32 GB of local RAM. The nodes are connected with an Infiniband FDR network (54 Gbps) and have access to a 115 TB Lustre parallel filesystem.

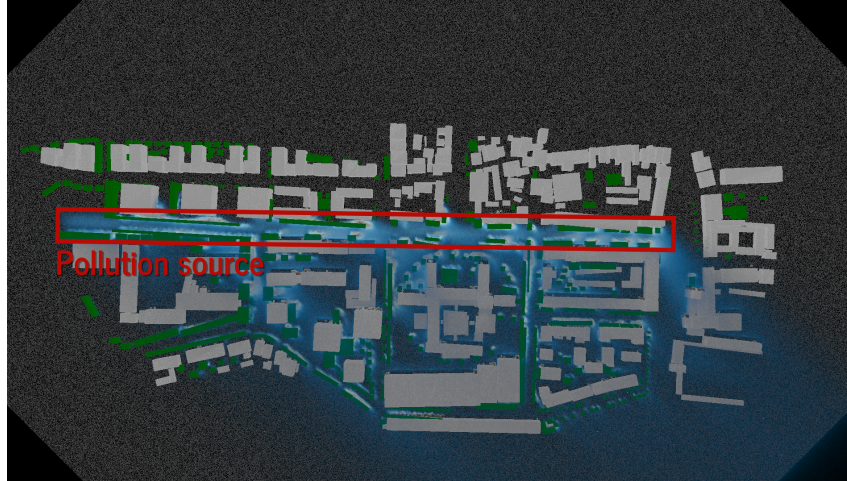
All the performed tests simulate the dispersion of air pollutants on an urban morphology that contains 297 buildings and 310 plants, making up a total of 607 obstacles. Besides that, it incorporates a single pollution source marked in red in figure 26a. During each run, the program calculates five distinct simulation phases. In each phase, it randomly generates a million particles and moves them according to a specific wind direction (180°, 220°, 260°, 310°, and 360°). The produced output images (one per phase) have a 1900 x 1080 resolution.

Both versions of VADIS were tested in different scenarios. The parameters that vary in each scenario are the following:

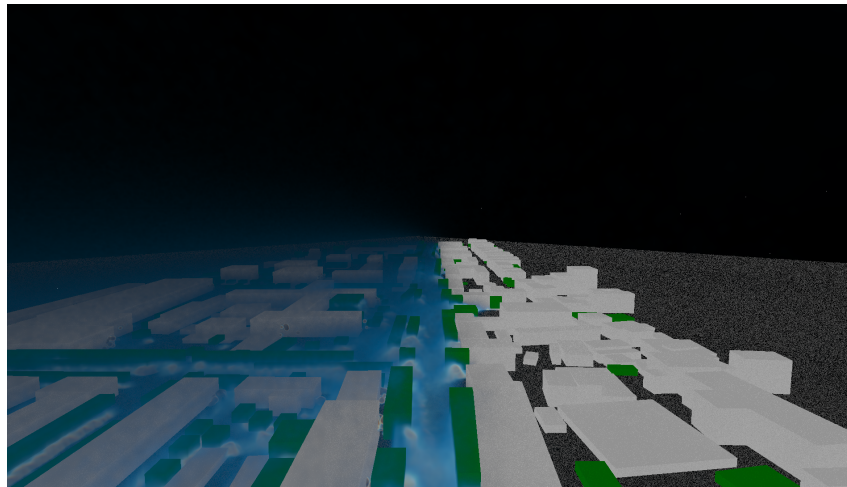
- **Viewport:** two different viewports were utilized: the top and the street views, represented in figures 26a and 26b, respectively;
- **Number of nodes:** the tests were performed in one and four nodes. When VADIS was executed in four nodes, two distinct partitions were used:  $-p \ 1 \ 4 \ 1$ , where the volume's width is partitioned into four equal parts, and  $-p \ 2 \ 2 \ 1$ , where the volume's length and width are divided into two

equal parts;

- **Visualization:** for each viewport and number of nodes, it was registered the time needed to perform only the simulation, as well as the time needed to perform both simulation and visualization with and without LOOM's "*interweaving*" method.



(a) Top view.



(b) Street view.

Figure 26: Viewports used on the tests.

Each scenario was executed ten times. The execution times presented in this document are an average of the three best results of each set of runs. Thus, the presented results correspond to the runs where the **OS** had less impact on the program's execution.

## 4.2 Performance Evaluation

### 4.2.1 Simulation

Executing the simulation of the five phases on a single node takes an average of 1034.653 seconds. In contrast, performing the simulation in four nodes takes an average of 722.981 and 372.209 seconds when using the partitions  $-p\ 2\ 2\ 1$  and  $-p\ 1\ 4\ 1$ , respectively. Therefore, the simulation's execution is 1.43 and 2.78 times faster using four nodes with partitions  $-p\ 2\ 2\ 1$  and  $-p\ 1\ 4\ 1$ , respectively.

| Nodes | Partition     | $T_s$     | <b>SS</b>    |
|-------|---------------|-----------|--------------|
| 1     | $-p\ 1\ 1\ 1$ | 1034.653s | -            |
| 4     | $-p\ 2\ 2\ 1$ | 722.981s  | <b>1.43x</b> |
|       | $-p\ 1\ 4\ 1$ | 372.209s  | <b>2.78x</b> |

Table 1: Simulation's execution time (in seconds) and speedup for each partition.

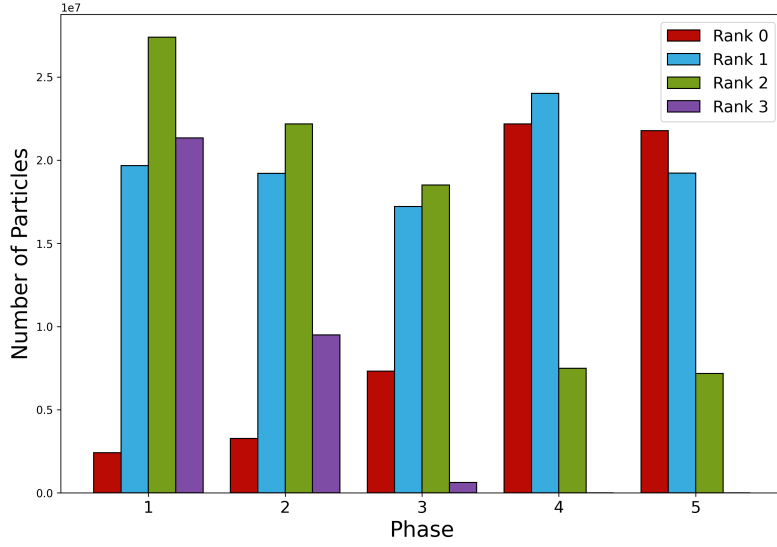
Table 1 presents the simulation execution time ( $T_s$ ) for each partition and the obtained speedup when using four nodes. The simulation speedup (SS) was calculated using the following formula:

$$\text{Simulation Speedup} = \frac{T_{s(-p\ 1\ 1\ 1)}}{T_{s(-p\ x\ y\ z)}} \quad (4.1)$$

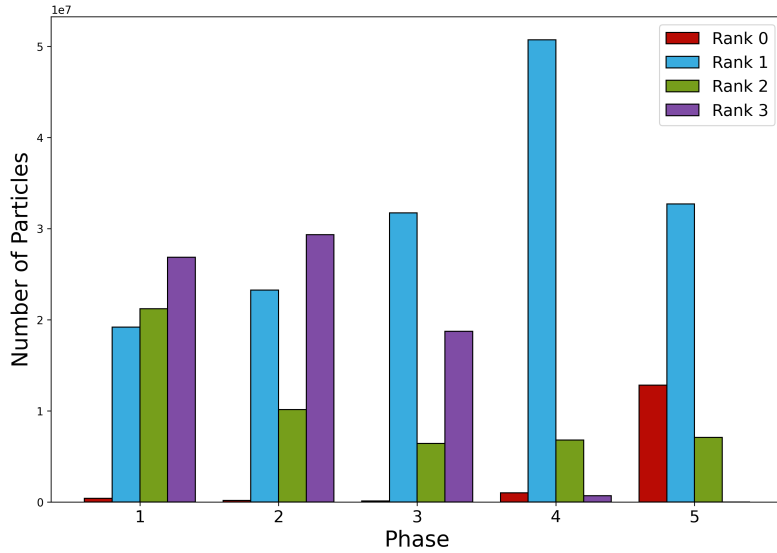
$T_{s(-p\ 1\ 1\ 1)}$ : simulation execution time using one node

$T_{s(-p\ x\ y\ z)}$ : simulation execution time using four nodes

When we compare the two partitions used to perform the simulation in four computing nodes, it is possible to see that executing VADIS with the partition  $-p\ 1\ 4\ 1$  is practically two times faster than executing it with the partition  $-p\ 2\ 2\ 1$ . This behavior is a consequence of the static workload distribution among the multiple nodes. Figures 27a and 27b depict the number of particles processed by each rank in every phase for the partitions  $-p\ 1\ 4\ 1$  and  $-p\ 2\ 2\ 1$ , accordingly. As we can see in these figures, the partition  $-p\ 1\ 4\ 1$  presents a more even distribution of the particles among the nodes compared to the partition  $-p\ 2\ 2\ 1$ . However, it is not ideal.



(a) Number of processed particles for the partition  $-p \ 1 \ 4 \ 1$ .



(b) Number of processed particles for the partition  $-p \ 2 \ 2 \ 1$ .

Figure 27: Number of particles processed by each rank in each phase.

It is relevant to highlight that, in partition  $-p \ 2 \ 2 \ 1$ , rank 0 is almost not used during phases 1, 2, 3, and 4. It only processes a considerable number of particles in phase 5. Furthermore, the poor workload distribution is evident in phases 4 and 5. In these cases, rank 1 has way more particles to process than the other ranks. Specifically, in phase 4, it is almost responsible for processing all the generated particles.

The particle distribution among the nodes on each phase for both partitions can also be observed in figures 28 and 29. These heatmaps portray the number of particles that crossed each grid's position during the simulation. The lighter the color, the greater the number of particles. The white lines represent the borders between the different ranks.

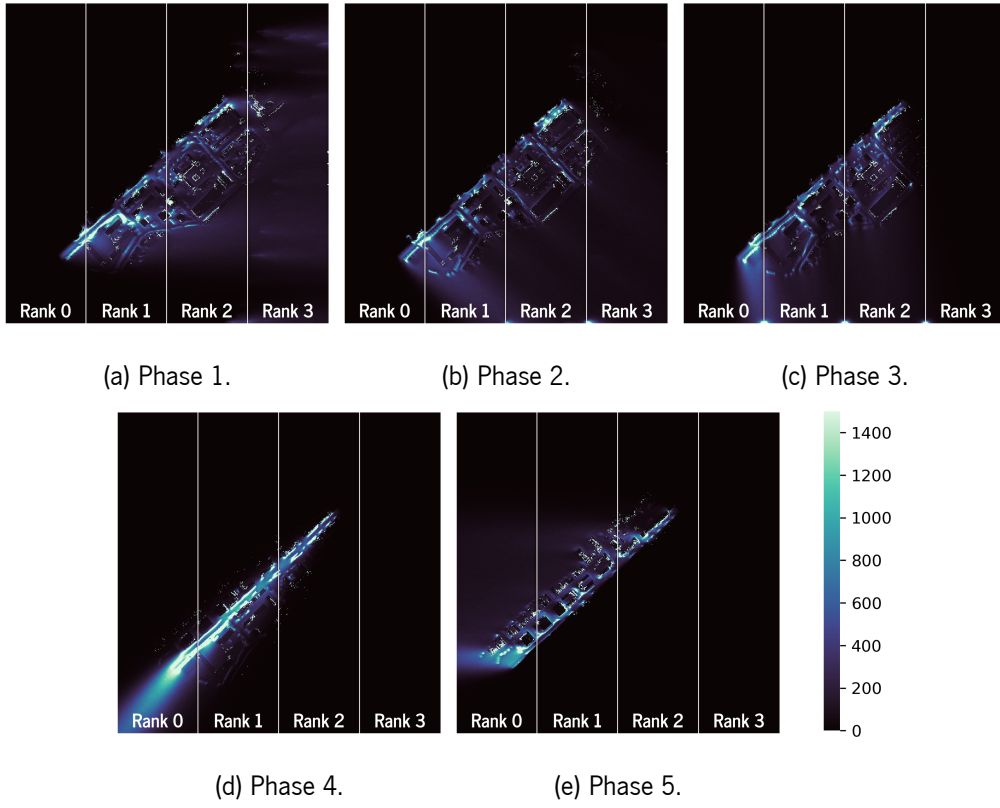


Figure 28: Particle distribution on each phase for the partition  $-p \ 1 \ 4 \ 1$ .

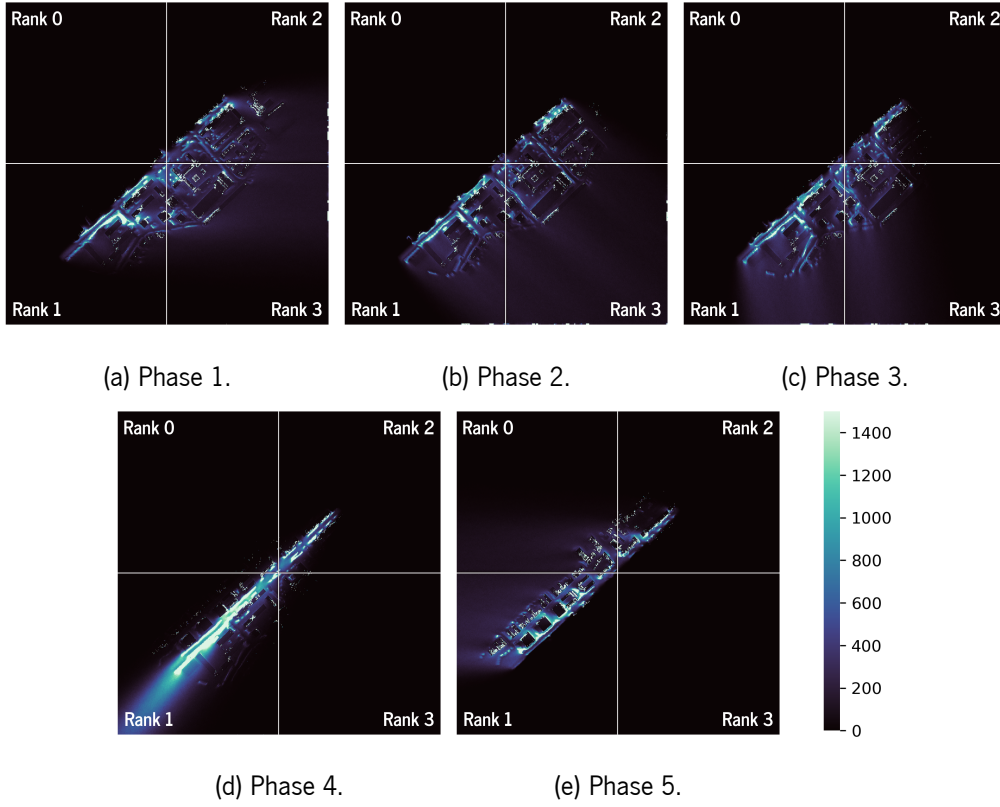
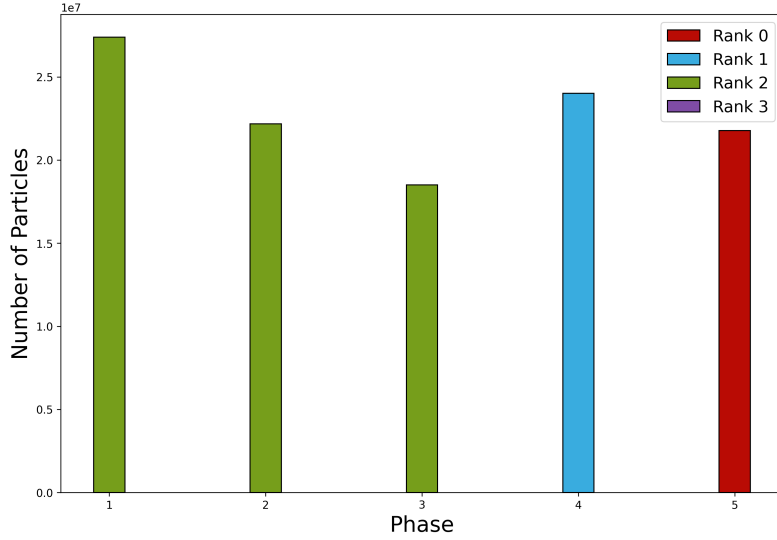
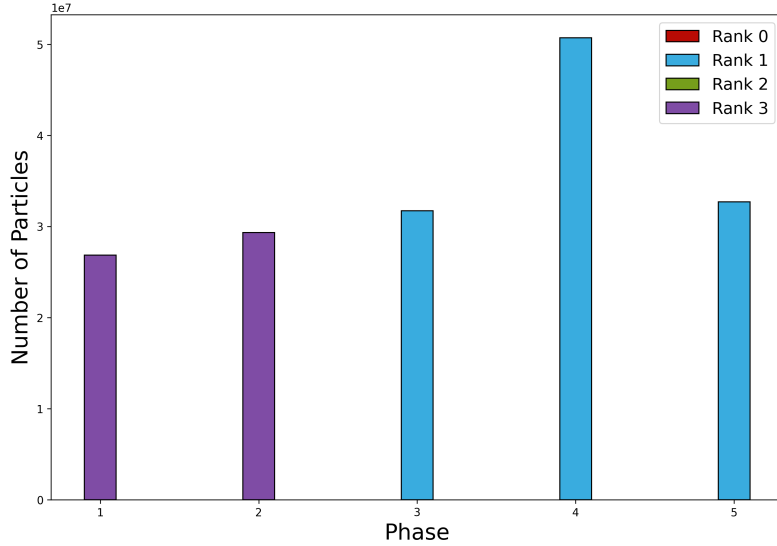


Figure 29: Particle distribution on each phase for the partition  $-p \ 2 \ 2 \ 1$ .

Currently, the workload distribution among the nodes is static, which means that the rank with the most particles to process limits the simulation's speedup. In figure 30, we can observe the ranks that constrain the speedup on each phase. On partition  $-p\ 1\ 4\ 1$ , rank 2 limits phases 1 to 3, and rank 1 and 0 constraint phases 4 and 5, respectively. On partition  $-p\ 2\ 2\ 1$ , rank 3 restrains phases 1 and 2, while rank 1 limits phases 3 to 5.



(a) Partition  $-p\ 1\ 4\ 1$ .



(b) Partition  $-p\ 2\ 2\ 1$ .

Figure 30: Ranks that constrain the simulation speed up on each phase.

Implementing a dynamic scheduling method like the one presented in Navrátil et al. (2012) can eliminate this limitation. In this approach, the scheduler assigns the work according to the availability of each rank as the simulation advances, enabling a more even distribution of the workload among nodes.

## 4.2.2 Simulation and Visualization

Before showing the results obtained when executing the simulation and visualization, it is pertinent to mention that the notation used in this chapter is the same as the one created by [Barbosa et al. \(2021\)](#).

Let us recall it:

- $T_s$ : execution time of the simulation
- $T_{s+v}$ : execution time of the traditional method
- $T_{s||v}$ : execution time of the "interweaving" approach

Besides that, this section also mentions the "improvement" concept presented in the same paper. The "improvement" ( $\downarrow TTV$ ) is the decrease in the time needed to perform the visualization using the "interweaving" method in comparison with the traditional approach. It can be calculated with the following formula:

$$\downarrow TTV = \frac{T_{s+v} - T_s}{T_{s||v} - T_s} \quad (4.2)$$

Additionally, this section refers to the global speedup (GS) and the percentage of hidden visualization (HV%). These are determined using the formulas 4.3 and 4.4, respectively.

$$\text{Global Speedup} = \frac{T_{s||v}(-p111)}{T_{s||v}(-pxyz)} \quad (4.3)$$

$T_{s||v}(-p111)$ : execution time of the "interweaving" approach using one node

$T_{s||v}(-pxyz)$ : execution time of the "interweaving" approach using four nodes

$$\text{Percentage of Hidden Visualization} = \frac{(T_{s+v} - T_s) - (T_{s||v} - T_s)}{T_{s||v} - T_s} \quad (4.4)$$

After comprehending this nomenclature, we can move forward to the result analysis. Firstly, let us examine the results for the baseline version when the viewport is the top view, presented in table 2. When executing DISPER with a single computational node, it is possible to hide 71% of the visualization time, corresponding to an improvement of 3.48 times. In contrast, when using four nodes with the partitions  $-p221$  and  $-p141$ , the percentage of hidden visualization time drops to 31% and 32%, respectively. Consequently, the improvement also reduces to 1.45 and 1.47. Nevertheless, executing the program in four computing nodes with the partitions  $-p221$  and  $-p141$  is 1.41 and 2.56 times faster than executing it with a single node.

| Nodes | Partition | $T_{s+v}$ | $T_{s  v}$ | HV% | $\downarrow TTV$ | GS           |
|-------|-----------|-----------|------------|-----|------------------|--------------|
| 1     | -p 1 1 1  | 1270.870s | 1102.58s   | 71% | <b>3.48x</b>     | -            |
| 4     | -p 2 2 1  | 811.699s  | 784.190s   | 31% | <b>1.45x</b>     | <b>1.41x</b> |
|       | -p 1 4 1  | 458.658s  | 431.123s   | 32% | <b>1.47x</b>     | <b>2.56x</b> |

Table 2: Visualization’s execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the top view with the baseline version.

When we use the ray-forwarding version to render the top view, there is no drastic change in the percentage of hidden visualization, improvement, and global speedup for the partition -p 2 2 1. For partition -p 1 1 1, the percentage of hidden visualization drops from 71% to 68%, leading to the decrease of the improvement to 3.13. However, the most substantial change is verified on the partition -p 1 4 1. In this specific case, the percentage of hidden visualization increases from 32% to 53%, raising the improvement from 1.47 to 2.15. The results related to the top view for the ray-forwarding version can be observed in table 3.

| Nodes | Partition | $T_{s+v}$ | $T_{s  v}$ | HV% | $\downarrow TTV$ | GS           |
|-------|-----------|-----------|------------|-----|------------------|--------------|
| 1     | -p 1 1 1  | 1301.627s | 1119.837s  | 68% | <b>3.13x</b>     | -            |
| 4     | -p 2 2 1  | 842.848s  | 802.426s   | 34% | <b>1.51x</b>     | <b>1.40x</b> |
|       | -p 1 4 1  | 483.678s  | 424.141s   | 53% | <b>2.15x</b>     | <b>2.64x</b> |

Table 3: Visualization’s execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the top view with the ray-forwarding version.

Even though the global speedup is practically the same on both versions (approximately 1.4 and 2.6 for the partitions -p 2 2 1 and -p 1 4 1, respectively), it is relevant to highlight that the ray-forwarding version is slightly slower than the baseline version for both traditional and "interweaving" approaches on most cases. For instance, executing VADIS for the partition -p 2 2 1 takes 784.190 seconds using the baseline version and 802.426 seconds using the ray-forwarding version. This behavior means that, for this application, it is more expensive to trade rays among the multiple computational nodes than to trade the buffers to perform the image compositing.

When we modify the viewport to the street view, the percentage of hidden visualization, the improvement, and the global speedup decrease for both versions of VADIS. When executing the baseline version with a single node, it is possible to hide 55% of the visualization time, corresponding to an improvement of



2.22 times. With four computational nodes, this percentage declines to 22% and 9% when using the partitions  $-p\ 2\ 2\ 1$  and  $-p\ 1\ 4\ 1$ , respectively, representing an improvement of 1.10 and 1.28. However, once again, using four nodes leads to better performance. Executing the program with the partitions  $-p\ 2\ 2\ 1$  and  $-p\ 1\ 4\ 1$  is 1.32 and 1.69 times faster than using the partition  $-p\ 1\ 1\ 1$ . These results can be consulted in table 4.

| Nodes | Partition     | $T_{s+v}$ | $T_{s  v}$ | HV% | $\downarrow TTV$ | GS           |
|-------|---------------|-----------|------------|-----|------------------|--------------|
| 1     | $-p\ 1\ 1\ 1$ | 1650.637s | 1311.66s   | 55% | <b>2.22x</b>     | -            |
| 4     | $-p\ 2\ 2\ 1$ | 1073.343s | 995.988s   | 22% | <b>1.10x</b>     | <b>1.32x</b> |
|       | $-p\ 1\ 4\ 1$ | 816.350s  | 774.784s   | 9%  | <b>1.28x</b>     | <b>1.69x</b> |

Table 4: Visualization's execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the street view with the baseline version.

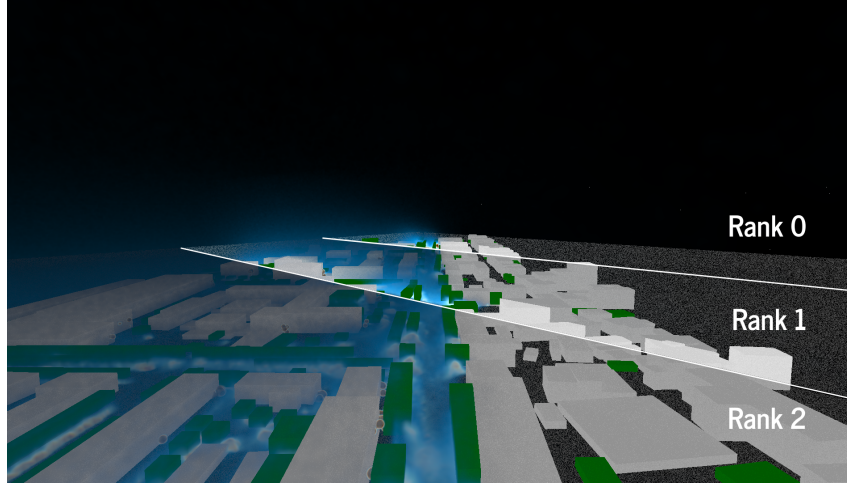
Switching to the ray-forwarding version, the percentage of hidden visualization and improvement is practically the same as in the baseline version. For the partitions  $-p\ 1\ 1\ 1$  and  $-p\ 2\ 2\ 1$ , these values slightly dropped, while for partition  $-p\ 1\ 4\ 1$  they registered a minor increase. The global speedup decreased for partitions  $-p\ 2\ 2\ 1$  and  $-p\ 1\ 4\ 1$  to 1.26 and 1.62, respectively. The results related to the street view for the ray-forwarding version are presented in table 5

| Nodes | Partition     | $T_{s+v}$ | $T_{s  v}$ | HV% | $\downarrow TTV$ | GS           |
|-------|---------------|-----------|------------|-----|------------------|--------------|
| 1     | $-p\ 1\ 1\ 1$ | 1683.853s | 1352.870s  | 51% | <b>2.04x</b>     | -            |
| 4     | $-p\ 2\ 2\ 1$ | 1163.110s | 1075.397s  | 20% | <b>1.16x</b>     | <b>1.26x</b> |
|       | $-p\ 1\ 4\ 1$ | 907.856s  | 836.022s   | 13% | <b>1.25x</b>     | <b>1.62x</b> |

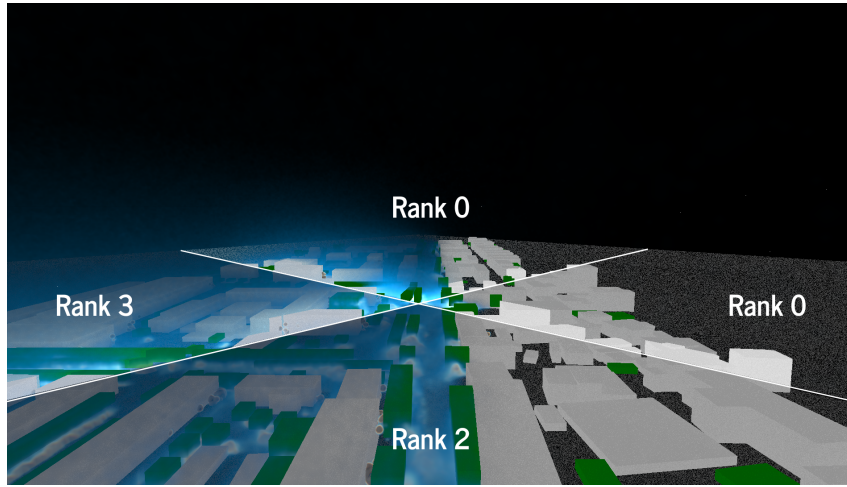
Table 5: Visualization's execution time (in seconds), hidden visualization percentage, improvement, and global speedup for the street view with the ray-forwarding version.

The discrepancy between the execution times of the baseline version and the ray-forwarding version is even more notorious when the viewport is defined as the street view. This happens because the visualization workload distribution is less uniform in the street view, increasing the number of rays traded among the nodes. For example, the VADIS execution with the partition  $-p\ 2\ 2\ 1$  takes 995.988 seconds using the baseline version and 1075.397 seconds using the ray forwarding version, resulting in a difference of 79.409 seconds. When rendering the top view with the same partition, this difference was 18.236 seconds.

On both versions of DISPER, the percentage of hidden visualization time is considerably smaller when the program uses the partition  $-p \ 1 \ 4 \ 1$  to render the street view compared to the partition  $-p \ 2 \ 2 \ 1$ . This happens because the data space portion of rank 3 is out of sight for partition  $-p \ 1 \ 4 \ 1$ , as it is possible to see in Figure 31a. When the partition  $-p \ 2 \ 2 \ 1$  is used, the data space portion of each rank is visible, just like Figure 31b shows.



(a) Partition  $-p \ 1 \ 4 \ 1$ .



(b) Partition  $-p \ 2 \ 2 \ 1$ .

Figure 31: Output images of the street view using different partitions.

When the partition  $-p \ 1 \ 4 \ 1$  is used in the baseline version, rank 3 does not participate in the rendering process, so the scene is only rendered by ranks 0, 1, and 2. In the ray-forwarding version, rank 3 takes part in the rendering procedure by generating the rays of its partition of the screen space and accumulating their color contribution. Consequently, the ray-forwarding version hides 13% of the visualization time, while the baseline version only hides 9%.

Below, in figures 32 and 33 are presented the baseline version's task timeline plots for each rank for

the partitions  $-p \ 1 \ 4 \ 1$  and  $-p \ 2 \ 2 \ 1$  when rendering the street view. As previously mentioned, in these plots, the simulation tasks are in red, and the visualization tasks are in green. The red vertical lines indicate the end of each simulation phase.

It is relevant to highlight that all the task time plots have the same scale. Consequently, the last portion of the plots represented in figure 32 is colored in black because executing VADIS with the partition  $-p \ 1 \ 4 \ 1$  is faster than executing it with the partition  $-p \ 2 \ 2 \ 1$ .

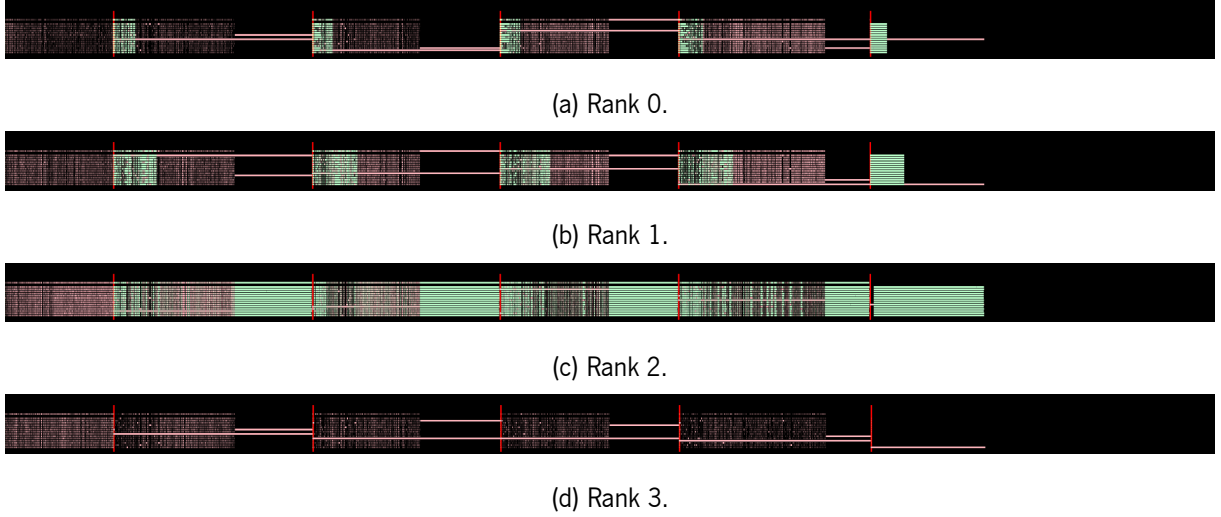


Figure 32: Baseline version's task timeline plots for each rank for the partition  $-p \ 1 \ 4 \ 1$  when rendering the street view. The simulation and visualization tasks are in red and green, respectively.

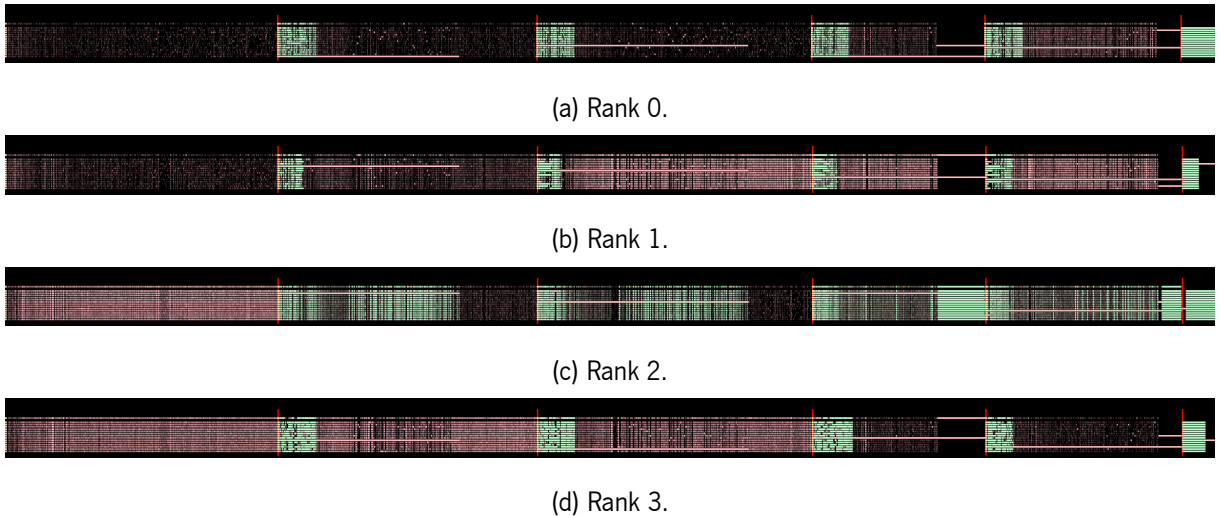
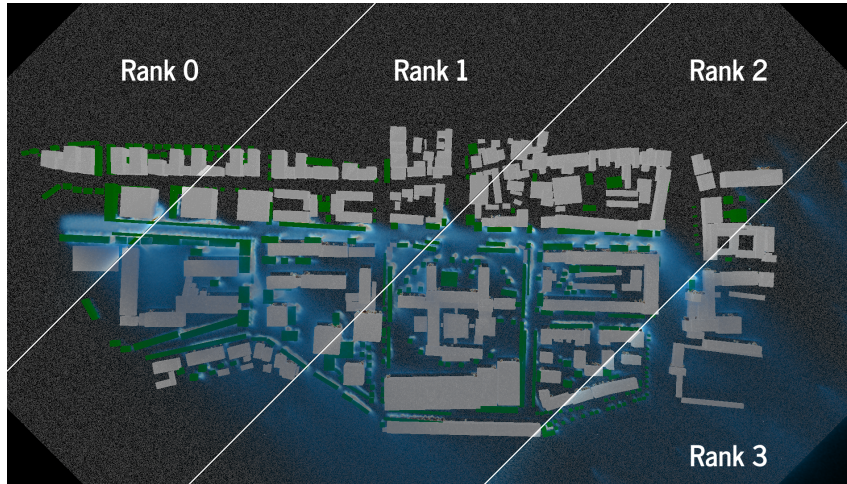


Figure 33: Baseline version's task timeline plots for each rank for the partition  $-p \ 2 \ 2 \ 1$  when rendering the street view. The simulation and visualization tasks are in red and green, respectively.

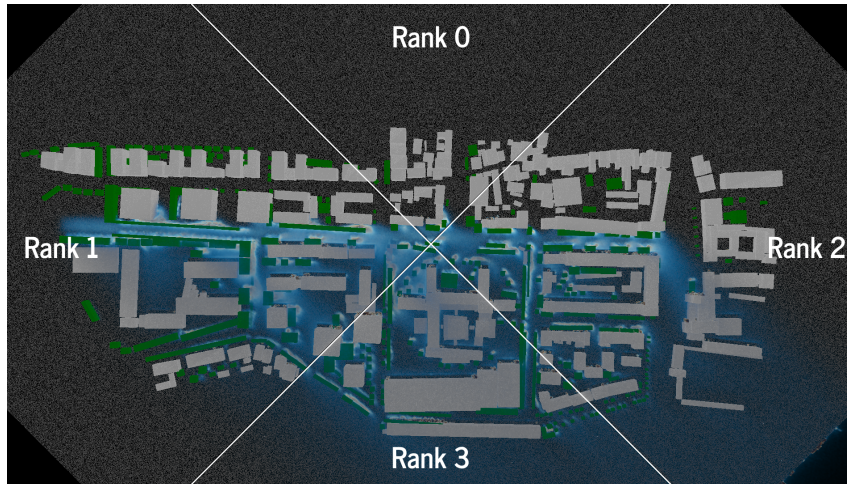
As we can verify in figure 32d, rank 3 does not integrate the visualization procedure on the partition

$-p \ 1 \ 4 \ 1$  since there are no green markings on the plot. In turn, in figure 33, we can see that all the ranks participate in the visualization for the partition  $-p \ 2 \ 2 \ 1$ .

In the baseline version, the nonparticipation of rank 3 on the visualization affects the global speedup of the partition  $-p \ 1 \ 4 \ 1$ . When rendering the top view, this partition reaches a global speedup of 2.56, which decreases to 1.69 when rendering the street view. The partition  $-p \ 2 \ 2 \ 1$  registers a smaller drop in the global speedup, going from 1.41 to 1.32. Even with this decrease, partition  $-p \ 1 \ 4 \ 1$  continues to be faster than partition  $-p \ 2 \ 2 \ 1$ .



(a) Partition  $-p \ 1 \ 4 \ 1$ .



(b) Partition  $-p \ 2 \ 2 \ 1$ .

Figure 34: Output images of the top view using different partitions.

It is also possible to observe that, on both partitions, rank 2 has a lot more rays to process than the rest of the ranks when rendering the street view. This happens because there is a static distribution of the visualization workload, where each rank is responsible for processing the rays that cross its portion of the data space. Since rank 2 owns the most visible portion of the data space, as it is possible to see in figure

31, it has more rays to process than the other ranks.

On the top view, the visualization is more equally distributed between the nodes on both partitions because they have approximately the same visible portion of the data space, as presented in figure 34. Nevertheless, this distribution is not even because some portions of the data space have more elements to render than others.

A dynamic scheduling method can solve the poor workload distribution in simulation and visualization. In the visualization, the scheduler would redistribute the rays according to the rank's availability as the program's execution advances. In the simulation, it would reassign the particles.

At last, it is relevant to mention that the results above show that LOOM is extensible to multiple computational nodes. Even though the improvement decreases when the program executes in four nodes, LOOM can still reduce the visualization up to 2.15 times, enhancing the overall performance of VADIS.



## Chapter 5

# Conclusions and Future Work

### 5.1 Conclusions

LOOM is a task-scheduling framework that enables tightly coupled in situ visualization without substantially increasing the simulation execution time by applying an “*interweaving*” method. This method can improve the performance of scientific visualization programs by using the simulation idle times to carry out the visualization tasks. The main goal of this thesis was to create an enhanced version of LOOM that could handle large-scale problems and ally it with ray-forwarding to overcome obstacles posed by distributed memory environments.

The improved version of LOOM was tested with VADIS, a scientific visualization software that simulates the dispersion of air pollutants in urban morphologies. Even though VADIS already took advantage of the framework, it had to be adapted to exploit its new features. In a search for better performance on the visualization, two versions of VADIS were created: one with and another without ray-forwarding.

The first version to be implemented was the one without ray-forwarding, called the baseline version. Its creation started with the construction of a synchronous multi-node version of VADIS. The synchronous version later transitioned into an asynchronous one, demanding the design of a communication system and a voting mechanism.

The ray-forwarding version was built on top of the baseline version. The only difference between both versions is the visualization procedure. In the baseline, all the ranks generate and process the rays of the entire screen space. In the ray-forwarding version, each rank generates the rays of its portion of the screen space and processes the ones that cross its partition of the data space. Firstly, the color accumulation of each ray occurred in the rank where it finished the traversal. Later, this was changed so that the color accumulation happens in the rank that generated the ray.

After wrapping the implementation, both versions were tested under variable conditions. The tests were performed with different viewports (top and street views) and number of nodes (one and four nodes).

When VADIS was executed in four nodes, two distinct partitions were used: -p 1 4 1, where the volume's width is partitioned into four equal parts, and -p 2 2 1, where the volume's length and width are divided into two equal parts. For each test, the time needed to perform only the simulation was collected, as well as the time required to perform both simulation and visualization with and without LOOM's "*interweaving*" method.

As expected, executing VADIS on multiple computational nodes leads to better performance. When executing the simulation using four nodes, it is possible to obtain a speedup of 1.43 and 2.78 times with the partitions -p 2 2 1 and -p 1 4 1, respectively. The partition -p 1 4 1 ends up being almost two times faster than the partition -p 2 2 1. This happens because the static workload distribution among the multiple computational nodes is more uneven for the partition -p 2 2 1 than for the partition -p 1 4 1.

When the visualization is enabled, the improvement obtained using the "*interweaving*" method decreases when executing VADIS with multiple computational nodes. Nonetheless, executing VADIS in four nodes can be up to 2.64 times faster than executing it with a single node. In general, the ray-forwarding version displayed a worse performance when compared to the baseline version. This means that, for this application, trading rays between the nodes is more expensive than trading the buffers to perform the compositing of the final image.

In this thesis, it was proven that LOOM is extensible for multiple computing nodes. This framework showed that it can still reduce the visualization up to 2.15 times when the program executes in four nodes, boosting VADIS performance.

## 5.2 Future Work

In the future, it would be interesting to implement a dynamic scheduling strategy to solve the uneven workload distribution among the nodes for both simulation and visualization. Besides that, it may also be pertinent to recalibrate the model used by LOOM to predict the idle times. This recalibration would improve the accuracy of the predictions. Consequently, the size of the "*diced*" visualization tasks would closely match the size of the idle times, boosting the framework performance.

# Bibliography

- Greg Abram, Paul Navrátil, Pascal Grossett, David Rogers, and James Ahrens. Galaxy: Asynchronous ray tracing for large high-fidelity visualization. *2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 72–76, 2018. doi: 10.1109/LDAV.2018.8739241.
- James Ahrens, A. Shoshani, K. L. Ma, A. Choudhary, T. Critchlow, S. Klasky, and V. Pascucci. *Scientific Discovery at the Exascale: Report from the (DOE) (ASCR) 2011 Workshop on Exascale Data Management, Analysis, and Visualization*. Office of Scientific and Technical Information (OSTI), 2011. doi: 10.2172/1011053.
- Tomas Akenine-Moller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering, Fourth Edition*. CRC Press, Boca Rota, FL, USA, 2018. ISBN 9780429225406.
- Shameem Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, Oregon, 2006.
- J.H. Amorim, V. Rodrigues, R. Tavares, J. Valente, and C. Borrego. Cfd modelling of the aerodynamic effect of trees on urban air pollution dispersion. volume 461-462, pages 541–551, 2013. doi: 10.1016/j.scitotenv.2013.05.031.
- Utkarsh Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., Clifton Park, NY, USA, 2015. ISBN 1930934300.
- Utkarsh Ayachit, Brad Whitlock, Matthew Wolf, Burlen Loring, Berk Geveci, David Lonie, and E. Wes Bethel. The sensei generic in situ interface. In *Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV '16, page 40–44. IEEE Press, 2016. ISBN 9781509038725.
- João Barbosa, Paul Navrátil, Luís Santos, and Donald Fussel. Loom: Interweaving tightly coupled visualization and numeric simulation framework. *ISAV@SC*, 2021. doi: <http://doi.org/10.1145/3490138.3490139>.



João Barbosa, Donald Fussel, and Calvin Lin. Simple heterogeneous programming using multigranular tasks. n.d.

C. Borrego, O. Tchepele, A.M. Costa, J.H. Amorim, and A.I. Miranda. Emission and dispersion modelling of lisbon air quality at local scale. volume 37, pages 5197–5205, 2003. doi: 10.1016/j.atmosenv.2003.09.004.

Lei Chai, Qi Gao, and Dhabaleswar K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 471–478, 2007. doi: 10.1109/CCGRID.2007.119.

Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, California, 2000. ISBN 9781558606715.

Hank Childs, Sean D. Ahern, James Ahrens, Andrew C. Bauer, Janine Bennett, E. Wes Bethel, Peer-Timo Bremer, Eric Brugger, Joseph Cottam, Matthieu Dorian, Soumya Dutta, Jean M. Favre, Thomas Fogal, Steffen Frey, Christoph Garth, Berk Geveci, William F. Godoy, Charles D. Hansen, Cyrus Harrison, Bernd Hentschel, Joseph Insley, Chris R. Johnson, Scott Klasky, Aaron Knoll, James Kress, Matthew Larsen, Jay Lofstead, Kwan-Liu Ma, Preeti Malakar, Jeremy Meredith, Kenneth Moreland, Paul Navrátil, Patrick O’Leary, Manish Parashar, Valerio Pascucci, John Patchett, Tom Peterka, Steve Petruzza, Norbert Podhorszki, David Pugmire, Michel Rasquin, Silvio Rizzi, David H. Rogers, Sudhanshu Sane, Franz Sauer, Robert Sisneros, Han-Wei Shen, Will Usher, Rhonda Vickery, Venkatram Vishwanath, Ingo Wald, Ruonan Wang, Gunther H. Weber, Brad Whitlock, Matthew Wolf, Hongfeng Yu, and Sean B. Ziegeler. A terminology for in situ visualization and analysis systems. volume 34, pages 676–691, 2020. doi: 10.1177/1094342020935991.

NVIDIA Corporation. Ray tracing. <https://developer.nvidia.com/discover/ray-tracing>, 2024.

William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 9780080497808.

Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on*

- High Performance Distributed Computing*, HPDC '10, page 25–36, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589428. doi: 10.1145/1851476.1851481.
- Graham Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Antonin Bukovsky, and Jack Dongarra. Fault tolerant communication library and applications for high performance computing. *Los Alamos Computer Science Institute Symposium*, October 2003.
- Thomas Fogal, Fabian Proch, Alexander Schiewe, Olaf Hasemann, Andreas Kempf, and Jens Krüger. Freeprocessing: Transparent in situ Visualization via Data Interception. In Margarita Amor and Markus Hadwiger, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2014. ISBN 978-3-905674-59-0. doi: 10.2312/pgv.20141084.
- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, 2023.
- Randall Frank and Michael Krogh. *The EnSight Visualization Application*, pages 429–442. 09 2012. ISBN 978-1-4398-7572-8. doi: 10.1201/b12985-26.
- D. Geer. For programmers, multicore chips mean multiple challenges. *Computer*, 40(09):17–19, sep 2007. ISSN 1558-0814. doi: 10.1109/MC.2007.311.
- Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4):285–303, August 2003. ISSN 0885-7458. doi: 10.1023/A:1024504726988.
- Norhazlina Hamid, Robert Walters, and Gary Wills. An architecture for measuring network performance in multi-core multi-cluster architecture (mcmca). *International Journal of Computer Theory and Engineering*, 7:57–61, 02 2014. doi: 10.7763/IJCTE.2015.V7.930.
- Intel. Intel implicit spmd program compiler. <https://ispc.github.io/>, a.
- Intel. Intel inspector. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/inspector.html>, b.
- Intel. Intel oneapi threading building blocks. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html>, c.

- Intel. Intel vtune profiler. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>, d.
- Intel. Embree. <https://spec.oneapi.io/oneart/0.5-rev-1/embree.html>, 2021.
- Intel. Embree. <https://www.embree.org/>, 2023a.
- Intel. Open vkl. <https://www.openvkl.org/>, 2023b.
- Corporation Intel. Moore's law and intel innovation. <https://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html?wapkw=moore+laws>, e.
- Christopher R. Johnson and Charles D. Hansen. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004. ISBN 012387582X.
- Dieter Kranzlmüller, P. Kacsuk, Jack Dongarra, E. Gabriel, G. Fagg, George Bosilca, Jeffrey Squyres, V. Sahay, P. Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph Castain, D. Daniel, Richard Graham, and T. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. *Lecture Notes in Computer Science*, 3241:97–104, 01 2004.
- Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. The alpine in situ infrastructure: Ascending from the ashes of strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, page 42–46. Association for Computing Machinery, 2017. ISBN 9781450351393. doi: 10.1145/3144769.3144778.
- Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello adios: the challenges and lessons of developing leadership class i/o frameworks: Hello adios. *Concurrency and Computation: Practice and Experience*, 26, 05 2014. doi: 10.1002/cpe.3125.
- Preeti Malakar, Vijay Natarajan, and Sathish Vadhiyar. Inst: An integrated steering framework for critical weather applications. *Procedia CS*, 4:116–125, 12 2011. doi: 10.1016/j.procs.2011.04.013.
- J.M. Martins and C. Borrego. Describing the dispersion of pollutants near buildings under low wind-speeds: Real scale and numerical results. pages 149–158, 1998. doi: 10.2495/ENV980151.
- S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994. doi: 10.1109/38.291528.

- Kenneth Moreland. *IceT Users' Guide and Reference*. Sandia National Laboratories, 2009.
- Paul A. Navrátil, Donald S. Fussell, Calvin Lin, and Hank Childs. Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing. In Hank Childs, Torsten Kuhlen, and Fabio Marton, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012. ISBN 978-3-905674-35-4. doi: 10.2312/EGPGV/EGPGV12/061-070.
- Paul A. Navrátil, Hank Childs, Donald S. Fussell, and Calvin Lin. Exploring the spectrum of dynamic scheduling algorithms for scalable distributed-memory ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 20(6):893–906, 2014. doi: 10.1109/TVCG.2013.261.
- Hyungman Park, Donald Fussell, and Paul Navrátil. Spray: Speculative ray scheduling for large data visualization. *2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 77–86, 2018. doi: 10.1109/LDAV.2018.8739224.
- Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 101–108, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0897918967. doi: 10.1145/258734.258791.
- Roberto Ribeiro, João Barbosa, and Luís Santos. A framework for efficient execution of data parallel irregular applications on heterogeneous systems. *Parallel Processing Letters*, 25:1550004, 06 2015. doi: 10.1142/S0129626415500048.
- Alejandro Ribés and Bruno Raffin. The challenges of in situ analysis for multiple simulations. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV'20, page 32–37, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388122. doi: 10.1145/3426462.3426468.
- Gilad Shainer, Pak Lui, Martin Hilgeman, Jeffrey Layton, Cydney Stevens, Walker Stemple, Scot Schultz, Guy Ludden, Joshua Mora, and Georg Kresse. Maximizing application performance in a multi-core, numa-aware compute cluster by multi-level tuning. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, pages 226–238, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38750-0.
- Fengguang Song, Shirley Moore, and Jack Dongarra. Analytical modeling and optimization for affinity

based thread scheduling on multicore systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009. doi: 10.1109/CLUSTER.2009.5289173.

Jeffrey M. Squyres and Andrew Lumsdaine. A component architecture for lam/mpi. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 379–387, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39924-7.

National Technology and LLC Engineering Solutions of Sandia. Icet. <https://www.sandia.gov/ccr/software/icet/>, 2024.

Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013. doi: 10.1145/2503210.2503279.

# **Part I**

## **Appendices**

## Appendix A

# Message Dictionary

This appendix presents a dictionary that compiles all the message types mentioned in this thesis. It incorporates two sections, [A.1](#) and [A.2](#), dedicated to the messages created on LOOM and VADIS, respectively. Each entry of these sections is composed of the name, description, attributes, and execution method of the message type.

### A.1 LOOM

#### Cancel

Message that closes the communication thread.

##### Attributes

- Message type (uint64\_t)

##### Execution method

Prints the message "Terminate communication thread".

---

#### Vote

Message that triggers the voting procedure.

##### Attributes

- Message type (uint64\_t)

##### Execution method

Creates a thread that waits until all the processes have finished the iteration. Then, it calls the voting function, executes `MPI_Allreduce`, and indicates if processes terminated their work.

## A.2 VADIS

### Ack

Acknowledgement message.

#### Attributes:

- Message type (uint64\_t)
- Acknowledgement type (int)

#### Execution method

Decrements an acknowledgment counter according to the acknowledgment type. Bellow, there are the actions triggered by each acknowledgment type:

- **Type 0:** Decreases the particles' acknowledgment counter (acks\_count)
- **Type 1:** Decreases the rays' acknowledgment counter (ray\_acks)
- **Type 2:** Decreases the colors' acknowledgment counter (color\_acks)

---

### BoundingBox

Message that transports a bounding box.

#### Attributes:

- Message type (uint64\_t)
- ID of the process that sent the message (int)
- Bounding box (vkl\_box3f)

#### Execution method

Stores the received bounding box labeled with the ID of the process that sent it.

---

### Colors

Message that transports a set of color contributions.



**Attributes:**

- Message type (`uint64_t`)
- ID of the process that sent the message (`int`)
- Set of color contributions (`std::vector<openvkl::render::Color>`)

**Execution method**

Adds the received color contributions to the framebuffer and sends an acknowledgment message of type 2.

---

## MovingRays

Message that transports a set of rays.

**Attributes:**

- Message type (`uint64_t`)
- ID of the process that sent the message (`int`)
- Set of rays (`std::vector<openvkl::render::MovingRay>`)

**Execution method**

Stores the received rays and sends an acknowledgment message of type 1.

---

## Particles

Message that transports a set of particles.

**Attributes:**

- Message type (`uint64_t`)
- ID of the process that sent the message (`int`)
- Set of particles (`std::vector<vadis::vec3f>`)

**Execution method**

Stores the received particles and sends an acknowledgment message of type 0.





This work was financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.