UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

OPTIMIZE PARALLEL NUMERICAL APPLICATIONS FOR CLIMATE MODELLING

ORIOL DURAN ABELLO

Thesis supervisor: STELLA VALENTINA PARONUZZI TICCO (BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION)

Thesis co-supervisor: MARIO CESAR ACOSTA COBOS (Department of Computer Architecture)

Tutor: GLADYS MIRIAM UTRERA IGLESIAS (Department of Computer Architecture)

Degree: Bachelor Degree in Informatics Engineering (Computing)

Thesis report

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

26/06/2023

Acknowledgements

Thanks to Stella and Gladys for their guidance and patience during this project.

And to Mario, the co-supervisor of the project.

Contents

1 Table and figures Index

2 Abstract

Abstract

This project wants to evaluate the possible benefits of implementing shared memory parallelism in the most recent version of the NEMO model which currently uses distributed memory parallelism with MPI. Generally, hybrid parallelizations, which exploit distributed and shared memory, using both parallelism paradigms are more efficient. With the release of the latest version of NEMO 4.2 with improvements on the scalability, we want to evaluate the performance of OpenMP to implement the hybrid parallelism in order to improve the model's scalability and making it better suited for the new cluster architectures, which are tending towards increasing the amount of cores per node.

Resum

Aquest projecte vol avaluar els possibles beneficis d'implementar paral·lelisme amb memòria compartida en la versió més recent del model NEMO, el qual actualment només fa servir paral·lelisme amb memòria distribuida utilitzant MPI. Generalment les paral·lelitzacions híbrides, que explotan memòria distribuida i compartida, fent servir ambdós paradigmes de paral·lelisme són més eficients. Amb el llançament de l'última versió de NEMO 4.2 amb millores a l'escalabilitat, volem avaluar el rendiment de OpenMP per a implementar el paral·lelisme híbrid amb els objectius de millorar l'escalabilitat del model i preparar-lo per a les noves arquitectures de clusters, les quals estan tendint a incrementar el nombre de nuclis per node.

Resumen

Este proyecto quiere evaluar los posibles beneficios de implementar paralelismo con memoria compartida en la versión más reciente del modelo NEMO, el cual actualmente solo usa paralelismo con memoria distribuida utilizando MPI. Generalmente las paralelizaciones híbridas, que explotan memoria distribuida y compartida, usando ambos paradigmas son más eficientes. Con el lanzamiento de la última versión de NEMO 4.2 con mejoras a la escalabilidad, queremos evaluar el rendimiento de OpenMP para implementar el paralelismo híbrido con los objetivos de mejorar la escalabilidad del modelo y prepararlo para las nuevas arquitecturas de clusters, las cuales están tendiendo a incrementar el número de núcleos por nodo.

3 Context and Scope

3.1 Introduction and contextualization

As computational power available grows, so does the complexity of experiments and simulations available. In turn hardware architectures become more complex and models need to evolve to take full advantage of the new technologies [1,2,3].

When running experiments that require High Performance Computing (HPC), Scientists have various concerns. One of them being the time it takes to simulate a certain time period, also known as time to solution. Another concern would be to use the minimum resources required for the experiments, especially with the energetic crisis we're facing nowadays [4]. For this, a good efficiency and scalability of the model is required so that when using multicore clusters and supercomputers the time to solution is reduced without wasting resources.

NEMO [5] (Nucleus for European Modeling of the Ocean) is a state-of-the-art model for global ocean simulation. It's currently being used at the Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC - CNS) [6], a consortium dedicated to HPC and home to a variety of research teams, famous for being the host of the Marenostrum supercomputer [7].

This Bachelor Thesis of the Computer Engineering Degree, at the Facultat d'Informàtica de Barcelona, directed by Stella Paronuzzi, co-directed by Mario Acosta Cobos and tutored by Gladys Utrera Iglesias is done within the frame of the BSC's Computational Earth Sciences department.

3.2 Objectives and Justification

As it has been mentioned in the section above, time-to-solution as well as energy-to-solution are some significant concerns for earth science research groups [8] who rely on numerical models for simulations and experiments for important research. In the actual HPC paradigm where models may be executed using hundreds of thousands of processing cores [9], this means that achieving a good scaling performance, it is a must for numerical climate models. For this reason, the evolution of supercomputer technology requires the models to adapt to take full advantage of the new machines.

This project focuses on the model NEMO 4.2, which has a distributed memory based parallelism implementation. The planned architecture for new supercomputers seems to point at an increase of cores per shared memory node, especially with the appearance of new GPU clusters [10,11,12]. Since the implementation of new features in the latest NEMO 4.2 version [13], we want to evaluate the benefits of pursuing a hybrid distributed and shared memory parallel NEMO implementation using OpenMP [14] in the near future.

NEMO is an open source project written in Fortran 90 [15] and developed since 1993 from a consortium of over a hundred researchers. The large extent of the code makes implementing a full hybrid NEMO model, where all functions of the simulation timestep implement distributed and shared memory parallelism, too big of a task for the time available in this project.

For this reasons this project is focused on implementing proofs-of-concept hybrid versions with the following objectives:

1. Evaluate the viability of OpenMP multithreading for implementing a full hybrid NEMO model in the near future.
2. Evaluate if OpenMP multithreading may be used to improve the scalability at the bottlenecks in the current most time consuming NEMO functions.

3.3 Requisites and Validation

In order to ensure that the results are as desired we need to meet the following criteria:

1. A performance analysis of the proof of concept at the bottleneck. Compare the speedup and scalability obtained with the original function.

2. A performance analysis of OpenMP scalability for a hybrid model. Compare the speedup and scalability obtained with the original parallelization at the multithreaded region.

3. The hybrid proof of concept needs to execute producing the expected output. This means that an execution with a determinate input has to produce an output similar to the original version with the same input.
   The outputs will be compared and they can't differ more than a 5% as it has been accorded with the directors of the project.

3.4 Risks and Obstacles

- **My lack of experience**: it's my first time in a research job and I'm new to the environment as well as some of the technologies involved. There will be a learning and adaptation phase in the planning, but it might not be accurate with the actual learning curve.

- **Health accidents**: may it be physical injuries or sickness, it can happen to me or the project directors and would slow the work rate, in a worst case scenario it could render any of us unavailable to work for some days or weeks.

- **Technical accidents**: disk failures, electrical errors, or any other kind of accident that may render the BSC's computers unavailable to run the experiments. The BSC is equipped to minimize the impact of said accidents but in case any were to happen it would.

- **Not enough computing resources**: computing resources are not unlimited and there are other research teams at the BSC performing experiments at the same time. If there is too much workload on the computers, experiments will have to be delayed.

- **Parallelization may not be enough**: it may be possible that the current design and implementation is just not suited for big scalability. In that case, re-designing part of the algorithm, if not all, and re-implementing it to make a better use of the parallelization may be mandatory, and that would take a lot more time.

3.5 Stakeholders

Me as the main implicate in the project and who also depends on its completion to graduate as it's my TFG. And also the project's director Stella Paronuzzi, the co-director Mario Acosta Cobos and the tutor Gladys Utrera Iglesias.

The main stakeholder and the one who would be more interested in developing the project is the BSC - CNS, for it would be the most benefited by the efficiency gains of the models that they are currently using. Indirectly BSC's clients and collaborating companies and researchers get affected as well as it would allow changes and more flexibility on how to use the available resources and computing time.

3.6 Methodology

For the first month of working on the project there was a learning and adaptation phase, so that I could familiarize myself with the environment and frameworks that I needed to use for the project.

After that SCRUM [16] methodology was used until the end of the thesis. This helped to keep up with difficulties that arose, and the data collected from the experiments. A meeting with the team was scheduled every two weeks, and meetings with the thesis' directors were scheduled every week. The regular meetings allowed us to check that the data corresponds with what was expected and whether the project was being developed at a good pace.

The SCRUM sprints or iterations consisted of the following steps:

1. Study the code and deduce a possible parallelization strategy. This includes analyzing the potential impact of the proposed strategy and researching what steps are necessary to implement it.

2. Implementation of the strategy. Once we have agreed on a parallelization strategy to implement, we proceed to the proof of concept implementation. This includes debugging the code until it works as intended.

3. Experimenting. After we have the proof of concept running with the proposed parallelization we run the necessary experiments to gather the performance data.

4. Validating. The last step would be checking the outputs to make sure that we didn’t introduce any significant changes or errors in the simulation that would make the outputs incorrect.

By the end of the time we had available at the BSC, we had completed three iterations.

4 Temporal planning

In this chapter we discuss the original planning for the project, as specified on the early GEP deliverables, and how this planning has changed to the definitive planning.

4.1 Original Planning

Our initial intention was to carry out the project in the span of a semester, from September to January, to present it as early as possible. For that reason the original planification left out some of the hours available at the BSC as contingency hours, for the case we needed to daly the deadline, and made a plan that would account for only a total of 480 hours.

4.2 Causes of deviation from the plan

When it came to the realization of the planned tasks, there were some that required more time than that of the original planification along with unexpected troubleshooting, additionally to the risks foreseen in the GEP module.

4.2.1 Tools and Environment Setup Time

As most of the tasks required setting up new tools and or environment configurations, some required extra troubleshooting time in order to get running properly for the first time. Some of the most time consuming were:

- The compilation and execution of the NEMO model
- Setting up the DDT debugger [17]
- NEMO and proofs of concept executing with outputs enabled

4.2.2 Underestimated learning

Not only did the landing phase and setup take longer than expected, as the project progressed there were always new things to learn, be it about the parallelisation methods, the Marenostrum's job scheduler or the NEMO model.

4.2.3 Debugging

In the second iteration of implementing a hybrid strategy, we had a particular error where communications between processes were being truncated and it took us a long time to figure out why. We encountered this issue in January, when we had to make the decision to skip the first presentation turn because it would not be ready in time. Setting up the DDT debugger and getting it to work took us some extra time due to an issue with the versions and the support for the Marenostrum4 machine, and then we still had to navigate some node connection errors before getting it to work. While we couldn't get the debugger to work, I was trying to debug the program by coding custom debugging messages. After that, I had to get used to the DDT interface and find the useful information for the specific error. All of this took a substantial amount of time that increased the total hours invested in the implementation task of the second iteration.

4.3 Tasks and definitive plan

With the above causes for the time increases in various parts of the project we had to delay the original deadline and increase the total amount of hours of the project. This ended up using all of the 555 hours that we had available at the BSC. Most of the Documentation and final tasks were done later, accounting for a total of 600 hours. The tasks were divided in the following groups:

4.3.1 Project management

This group of activities was the one dedicated to all aspects related to project management. Due to the time increase of the whole project, the meeting hours naturally increased. The other tasks in this group were mostly done in the context of GEP and didn't suffer any alterations.

- **Define scope of the project**: Defining the frame of the project and its objectives within said frame and context. Indicating the relevance of the objectives and how it was going to be developed.
  Original estimated time: 10 hours - Definitive time: 10 hours

- **Temporal Planning**: planning of the work to be carried out during the project development, how much time will be dedicated to each task and when.
  Original estimated time: 10 hours - Definitive time: 10 hours

- **Economic Management**: Analysis of the economic cost of the project and its sustainability in a short and long term.
  Original estimated time: 10 hours - Definitive time: 10 hours

- **Meetings**: Throughout the development of the project there were various meetings with the directors and the team to ensure that the project was evolving correctly.
  Original estimated time: 30 hours - Definitive time: 50 hours

4.3.2 Study of the state-of-the-art

This group of activities was the one dedicated to making up for my lack of knowledge in Fortran 90 and deterministic climate prediction models as well as the current state of the NEMO model and familiarizing myself with its current implementation and performance.

- **Learned about deterministic climate prediction models** to have a better understanding of their objectives and how they work. Searched information about

different models and viewed the provided lesson videos on the topic on the landing phase.
Original estimated time: 10 hours - Definitive time: 10 hours

- **Read recent research on NEMO performance** about the performance data from the actual implementation of the model NEMO and the specifics of how it works.
Original estimated time: 30 hours - Definitive time: 35 hours

- **Learned Fortran 90** syntax and workflow to be able to understand the current NEMO implementation. This required some extra hours during the implementation phases.
Original estimated time: 30 hours - Definitive time: 35 hours

- **Analyzed the functions where the bottleneck is located**, studied their code and the structure of the calculations that need to be done. Looked for race conditions that could make shared memory parallelization inefficient.
Original estimated time: 40 hours - Definitive time: 40 hours

4.3.3 Practical Implementation

These were the main activities of the project as they constituted the actual development of the new implementations that strived to have a more effective performance. In our original planning we only considered enough time for two iterations, but with the new deadline we had enough time for three.

- **Study possible hybrid parallel strategies:** proposed and studied the viability of different hybrid parallelisation strategies for the identified bottleneck function.
Original estimated time: 40 hours - Definitive time: 50 hours

- **Implement a proof of concept** based on proposals made in the last task. This includes the necessary debugging until the compiled executable can run simulations without crashing.
Original estimated time: 40 hours - Definitive time: 80 hours

4.3.4 Experimentation, analysis and conclusion:

These tasks had the objective of gathering the data necessary for the requirements of the project.

- **Ran experiments on the Marenostrum** to get real world data of the performance of the proofs of concept implemented with a various amount of different cores per execution. I automated the experiment launching and results parsing with scripts, which made up for the time added by the troubleshooting and the extra iteration.
Original estimated time: 55 hours - Definitive time: 55 hours

- **Analyzed the experiment traces and results** from every execution, to get a basic view of whether the implementation was working as expected or not and determine if it was worth performing another iteration.
  Original estimated time: 50 hours - Definitive time: 70 hours

- **Validate**: Ran the model and the proofs of concept with outputs enabled and compared them to ensure the results of the new versions are valid.
  Original estimated time: 50 hours - Definitive time: 60 hours

Project Finalization
These are the necessary tasks to close project and prepare its defense:

- Documentation: The project's memory was written, gathering all the reports and data, along with the conclusions of the project.
  Original estimated time: 60 hours - Definitive time: 60 hours

- Oral presentation: hours destined to preparing the presentation of the project.
  Original estimated time: 15 hours - Definitive time: 15 hours

4.4 Gantt

Below (Figure 1) I attach a Gantt diagram of the distribution of the tasks made for the original planning that wanted to present the project in the first possible turn, which was in January and with the last the deadline for presenting the memory being the 16th of January.

After that (Figure 2) I attach the Gantt diagram of the distribution of the tasks in the definitive planification with the new deadline to hand in the memory being the 19th of June.

Meetings with the Computational Earth Sciences team were scheduled every two weeks. And a weekly meeting with the project's Directors to discuss more in detail the progress of the project.

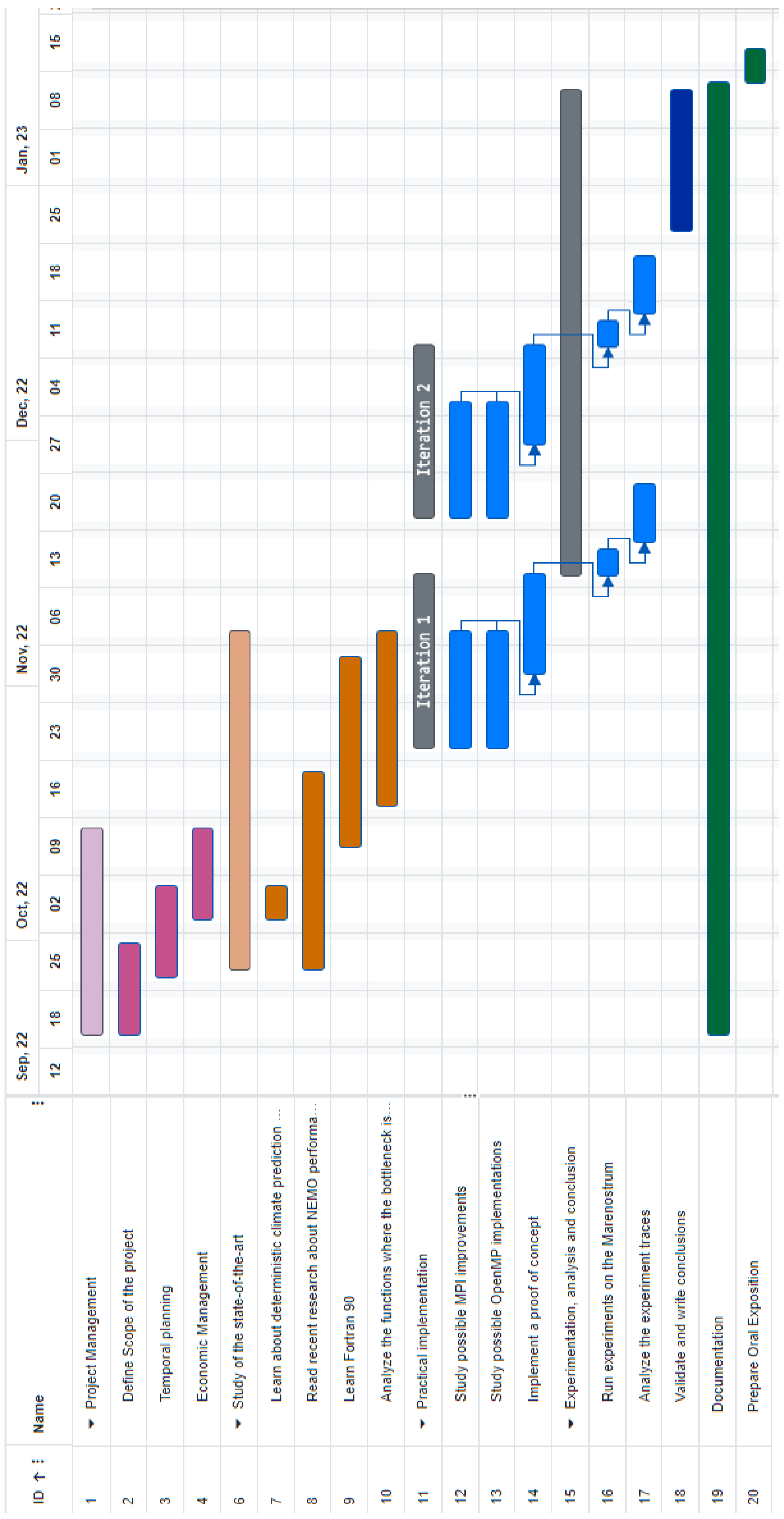


Figure 1. Gantt diagram of the Original planification

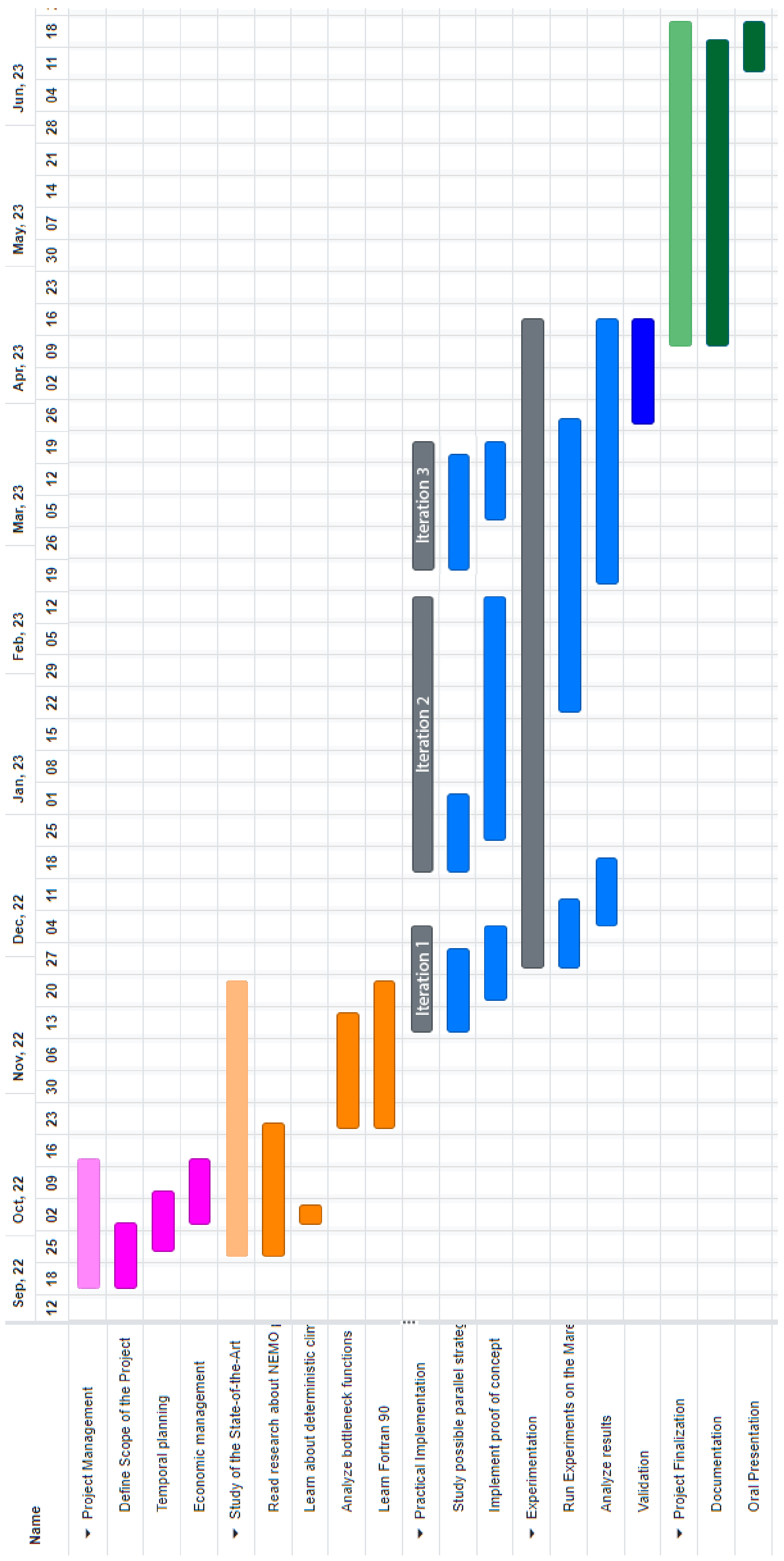


Figure 2. Gantt diagram of the definitive planification

5 Budget

In this section there's a detailed description of the economic cost of the project, first distributed by Human Resources, and Generic Costs. Then a summary of the total of the costs.

5.1 Human resources

Undergraduate student: Oriol Duran Abelló, the author and developer of this project.

Project Director: Stella Paronuzzi as the project manager from the BSC.

Project co-director: Mario Acosta Cobos as the second project manager from the BSC.

Support: developers and researchers from the Earth Sciences department of the BSC.

Table 1.1 Human Resources

Human resources	**Hours**	**€ / hour [18]**	**Total cost (€)**
Undergraduate Student	600	8	4800
Project Directors	70	50	3500
Support	20	30	600
Total	**690**	**-**	**8900**

5.2 Generic costs

5.2.1 Amortization

5.2.1.1 Hardware

Personal laptop: Laptop of the developer, DELL latitude 5020, also provided by the BSC.

Marenostrum4: A very expensive infrastructure ([7]), but we must use it to perform the experiments running NEMO, so we will compute the amortization for the estimated hours. To calculate the supercomputer's amortization the maximum Useful life is 5 years and it will be divided by its total nodes (3.456) and multiplied by 9 as that is the average amount I have been using for the experiments.

Table 1.2 Hardware Resources

Hardware Resources	Price (€)	Units	Useful Life (years)	Amortization
Laptop	1000	1	2	315.34€
Marenostrum 4	34M	-	5	3,119.08€
Total	**-**	**-**	**-**	**3,434.42€**

The following formula was used on the amortization calculation:

$$Cost\ of\ the\ equipment\ \div\ (Useful\ Life(max4)\ *\ WorkingHours(880))\ *\ HoursUsed$$

5.2.1.2 Software

Operating System: The project will be developed on the workstation provided by BSC with OpenSuse Leap 42.3 and on the developer personal laptop with Ubuntu.

Version Control: Git version control [19] on the private Earth Science Gitlab instance for the NEMO code.

BSC Toolset: Open source tools developed at the BSC that provide information about the experiment executions.

5.2.1.3 Space

The BSC Repsol Building is the building where the office is located. It is an enormous building, headquarters of the BSC-CNS. There is a huge number of workers there from a wide range of teams and departments, so we won't compute the amortization of the building as a cost of this project's budget.

For the furniture we will count the amortization of the items assigned to the developer's particular workspace computing a maximum Useful Life of 20 years:

Table 1.3 Furniture costs

Furniture	**Price (€)**	**Units**	**Useful Life (years)**	**Amortization (€)**
Office seat	200	1	2	63.07
Office table	240	1	2	75.68
Office drawer	110	1	2	34.69
Total	**550**	**-**	**-**	**173.44**

Total Furniture Costs: 173.44 €

Electricity: with the average values taken from [20], we can estimate a total cost of 24 € for the computer usage.

5.3 Total Cost

Table 2. Costs Summary

Human resources	**Hours**	**€ / hour**		**Total cost (€)**
Undergraduate Student	600	8		4800
Project Directors	70	50		3500
Support	20	30		600
Total	**545**	**-**		**8900**
Hardware Resources	**Price (€)**	**Units**	**Useful Life (years)**	**Amortization (€)**
Laptop	1000	1	2	315.34
Marenostrum 4	34M	-	5	3,119.08
Total	**2500**	**-**	**-**	**3,434.42**
Furniture	**Price (€)**	**Units**	**Useful Life (years)**	**Amortization (€)**
Office seat	200	1	2	63.07
Office table	240	1	2	75.68
Office drawer	110	1	2	34.69
Total	550	-	-	173.44
Electricity	-	**0.04 €/h**		24
Total Cost	-	-	-	**12,531.86**

Adding the total cost from the Human Resources and Generic costs we get a total of: **12,531.86€**

The price is nearly 10% higher than that calculated in the GEP budget because the contingency did account for the extra possible hours that we had, but not for the second project's director, as there was only one when the original planification was made.

6 Sustainability

6.1 Environmental Impact

The development of this project uses a minimal amount of resources, those being the electric energy needed in order to power the equipment. Though the biggest consumption would be the energy consumed by the Marenoustrum when running the experiments, we have no way right now to determine how much power is being used on the jobs related to the project from all the jobs running at the Marenostrum. On the other hand, the objective of this project is to make a more efficient use of the computation resources, so it will reduce the resources needed to run the simulations once deployed. This would mean a reduction in the actual carbon footprint and energy consumption.

On starting the project there's the risk that the efficiency can simply not be improved via the proposed methods and thus the energy spent on making the experiments can be seen as unnecessary, but even so one could consider that this would mean other ways need to be explored and the experiments were useful in determining that fact and pointing us to other directions.

6.2 Economic Impact

For the development of this project we have already discussed a budget that accounts for both human and material resources as well as some contingency plans for possible obstacles.
The cost of the project could be lower if the developer was experienced, thus requiring much less time on the learning and adaptation phase, though it could mean a higher cost per hour on the developer salary.

After the implementation is finished, the users of the NEMO model will be able to make a more efficient use of the multicore clusters at the Marenostrum supercomputer, thus reducing the economic cost of its usage.

6.3 Social Impact

The more efficient use of the Marenostrum will mean an improvement for all the research teams that use it, for it would mean that new studies performed here will have more resources, and this would in turn become beneficial for all of society since research from a wide variety of topics is performed there, some examples would be clima, health and computer science.

This project has also been an opportunity for me to take part in a project in a big and important research company such as the BSC and learn a lot of things from technical knowledge to project management and research.

7 State of the art

Climate modeling [21,22] is a scientific process that uses complex mathematical equations to simulate how the Earth's climate system works. It implies creating computer-based models that represent the different components of the climate system, including the atmosphere, oceans, land surface, and ice. These models are used to project future climate conditions [23] by changing various input parameters, such as greenhouse gas emissions, solar radiation, and volcanic activity.

The state of the art in climate modeling is constantly evolving, with new research and advancements in technology driving improvements in accuracy and precision. Modern climate models are based on a combination of observational data gathered at meteorological laboratories and theoretical physics, and they incorporate a wide range of feedback mechanisms and interactions between different components of the climate system.

These models are capable of making long-term predictions of future climate conditions, including changes in temperature, precipitation patterns, sea level rise, and extreme weather events. While there are uncertainties associated with these projections, they provide important insights into the potential impacts of human activities on the Earth's climate system and can inform policy decisions aimed at mitigating climate change. For example the CESM (Community Earth System Model) [24] which has been used in multiple studies about the climate changes in the future of our ecosystem [25,26]. As well as the CMIP (Coupled Model Intercomparison Project) [27,28,29].

Overall, climate modeling represents a critical tool for understanding the complex interactions between different components of the Earth's climate system and for predicting the potential impacts of human activities on our planet's climate.

Climate model simulations require a significant amount of computational resources and data. The computational resources needed to run a climate model simulation can vary depending on the complexity of the model, the size of the simulation domain, and the desired resolution of the output data. In general, climate model simulations require HPC infrastructure with a large amount of memory and storage capacity.

7.1 The NEMO model

As it is stated in the NEMO Comunity main web page [5]: NEMO standing for "Nucleus for European Modelling of the Ocean" is a state-of-the-art modeling framework for research activities and forecasting services in ocean and climate sciences, developed in a sustainable way by a European consortium.

NEMO is a community-developed model that is used by oceanographers around the world to simulate a wide range of ocean processes, including circulation, biogeochemistry, sea ice, and ocean-atmosphere interactions, published research can be found at [30]. The model is based on a set of mathematical equations that describe the physical and biogeochemical processes that govern the behavior of the oceans. These equations are solved using advanced numerical methods to simulate the behavior of the ocean system over time.

The three main components of the model are:

- NEMO-OCE: models the ocean {thermo}dynamics and solves the primitive equations.
- NEMO-ICE: models sea-ice {thermo}dynamics, brine inclusions and subgrid-scale thickness variations.
- NEMO-TOP-PISCES: models the {on,off}line oceanic tracers transport and biogeochemical processes.

7.1.1 ORCA grid

The ORCA grid is a specialized grid configuration utilized within the NEMO (Nucleus for European Modelling of the Ocean) framework for ocean modeling. This grid design incorporates a non-uniform distribution of grid points, specifically a tripolar arrangement, which accounts for the convergence of meridians at high latitudes. The ORCA grid offers several advantages for simulating oceanic processes, including improved representation of coastal regions, narrow straits, and boundary currents.

The resolution of the grid can be configured depending on the needs of the simulation. By achieving a balance between resolution and computational efficiency, the ORCA grid enables more accurate and realistic modeling of global or regional ocean dynamics along with flexibility within the NEMO framework [31].

8 Tools and Languages

In this section we will introduce the main software tools and languages used in this project.

8.0.1 NEMO 4.2 implementation

NEMO is a complex ocean modeling software framework that is implemented using a variety of software technologies. Some of the key technologies involved in the implementation of NEMO include:

Fortran 90: NEMO is primarily written in the **Fortran 90 programming language**, which is a high-level language commonly used for scientific computing.

MPI [32]: the **Message Passing Interface** (MPI) which is a parallel programming model that works on distributed and shared memory and supports C and Fortran among other languages. MPI is a standardized protocol that allows for efficient communication and synchronization between parallel processes, enabling NEMO to take advantage of high-performance computing clusters.

NetCDF: the **Network Common Data Form** (NetCDF) file format is used for storing and exchanging data [33]. NetCDF is a self-describing, machine-independent data format that is widely used in the scientific community. NEMO outputs data in NetCDF format, which can then be visualized and analyzed using specialized software tools such as Ncview.

XIOS: the **XML Input Output Server** (XIOS) [34] library is used to manage input and output operations. XIOS is a modular and flexible I/O server that enables NEMO to write output data to disk in parallel, reducing the time and resources required for data storage and analysis.

More detailed information can be found in the NEMO manual [31].

8.0.2 OpenMP

OpenMP (OMP): is a parallel programming model that supports C and Fortran, it will be explored whether it's worth implementing features with it in order to improve NEMO's efficiency.

8.0.3 BSC Tools

Paraver: [35] data browser to visualize information collected from experiment execution traces using parallelization implemented with MPI or OpenMP.

Extrae: [36] instrumentation library that we will use to catch OpenMP and MPI events at runtime generating log information for post-mortem analysis in parallelized programs

Paraver and Extrae are both part of an open-source toolset developed at the BSC. More detailed information about these tools can be found at [37].

8.0.4 DDT

ARM Distributed Debugging Tool (DDT) is a debugger specifically designed to be used in HPC environments, as its purpose is to keep track of the state of the program in every MPI node/process it uses, which is very helpful for debugging any errors encountered during the development of this project.

Some of the main features of DDT are:

- Interactively track and debug program crashes that may occur on certain nodes.
- Track memory related problems in your programs.
- Get more information about crashes.

This information has been extracted from the BSC's DDT user manual [17].

8.0.5 GIT

Git is a distributed version control system for software development that allows multiple people to work on the same codebase(or repository) concurrently [19]. It tracks changes made to files over time, allowing users to easily revert to previous versions if necessary. Git uses a branching model that allows developers to create independent versions of the codebase to work on without interfering with the main branch. It also supports merging changes made in separate branches back into the main branch. Git is widely used in software development due to its efficiency, flexibility, and powerful features.

The branching feature proved especially useful to keep the different implementations of the strategies we wanted to test, stored in the same repository and provided a straightforward way to swap between them.Issues were used to keep track of the progress in different aspects of the project as well as obstacles that came up. These however were not copied in the migration from gitlab to github.

8.0.6 Slurm

Slurm is a highly scalable and fault-tolerant cluster management and job scheduling system used in high-performance computing (HPC) environments [38]. It provides a simple and flexible interface for submitting and managing jobs, allowing users to control the resources allocated to their jobs, such as the number of CPUs, memory, and runtime limits. Slurm also enables efficient resource utilization through advanced scheduling and priority systems, as well as integration with parallel environments like MPI and OpenMP.

The Marenostrum4 cluster in which the experiments for this project were run (more details are provided in chapter 10 which focuses on experiments) is a Slurm managed cluster.

9 Parallelization Paradigms

In this chapter, we will discuss the basics of the MPI and OMP parallelization models more in detail than in the previous one and then also introduce how a hybrid model would work.

9.1 Message Passing Interface

MPI is a message-passing standard designed to work on parallel computer architectures. It is designed to work on distributed memory systems, though it can also work on shared memory. There are various libraries that implement this standard such as OMPI, MPICH or Intel MPI [32,39,40].

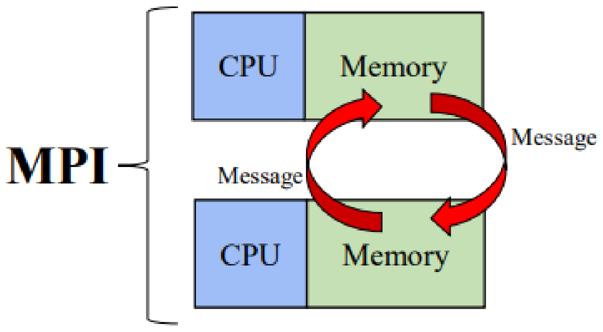


Figure 3. MPI functionment scheme from [41]

9.1.1 The Standard

To use the MPI standard in an implementation, the parallel code must first be initialized using the MPI_Init() function. This function initializes the MPI environment and sets up the necessary data structures for communication between processes.

Once the MPI environment is initialized, processes can communicate with each other using a variety of MPI calls. For example, the MPI_Send() function can be used to send a message from one process to another, while the MPI_Recv() function can be used to receive a message. These functions take as arguments the data to be sent or received, as well as information about the source and destination processes to match receiving calls with sending calls.

Other MPI functions include MPI_Barrier(), which can be used to synchronize the execution of processes, and MPI_Reduce(), which can be used to perform a reduction operation (such as summing the values of an array) across a group of processes.

To finalize the MPI environment and free up any resources that were allocated during initialization, the MPI_Finalize() function should be called. This function ensures that all processes have completed their work and have exited cleanly.

9.1.2 NEMO’s communications

Since the processes in NEMO need to send large amounts of data the calls to MPI communications have been implemented in a function wrapper called *lbc_lnk* to be able to call multiple communications and pass large arrays of data with a single line of code.

Depending on the options passed in the arguments, the wrapper will determine what data from the array has to be sent via *MPI_ISend* or updated with new data coming from the *MPI_Recv*. ISend is the non-blocking version of Send, meaning the sender process doesn't need to wait for the receive to complete, unless it’s later specified with an *MPI_Wait.*

The calls to *lbc_lnk* concerning the development of this project perform a communication with a common structure in parallel physics grid-oriented simulations known as 9-point halo communication. Every cell of the grid communicates with its eight adjacent neighbors to update the data regarding the points in the external halo of the cell.

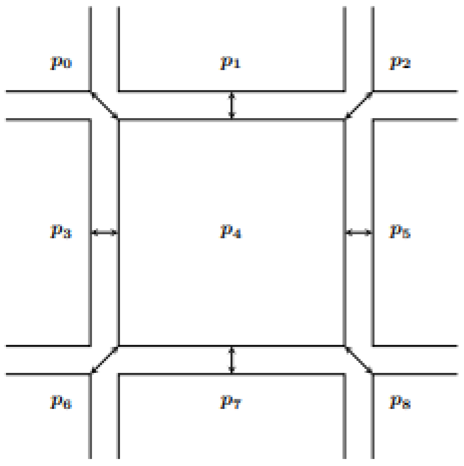


Figure 4. 9-point halo communication scheme from [42]

9.2 OpenMP

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) that enables parallel programming on shared memory architectures, such as multi-core CPUs and symmetric multiprocessing (SMP) systems. OpenMP uses a model in which a single program runs concurrently on multiple processing elements, each with its own local memory, while sharing a single address space.

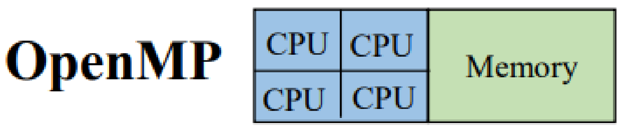


Figure 5. OpenMP functionment scheme from [41]

OpenMP accomplishes parallelism by providing a set of compiler directives, runtime library routines, and environment variables that enable a programmer to express parallelism in a high-level, platform-independent way. The programmer specifies which parts of the program can be executed concurrently by enclosing them in a multithreaded region, using the !$OMP PARALLEL/!$OMP END PARALLEL construct. Within a parallel region, threads are spawned, and each thread executes a copy of the enclosed code, with different portions of the data.

OpenMP also provides several synchronization and ordering constructs, such as locks and barriers, to coordinate access to shared resources and ensure correctness in the presence of multiple threads.

In OpenMP, variables can be declared as **private** or **shared** using specific clauses in the program. Private variables are copied to the stack of each thread and threads can only access their copy of the variable. Shared variables don't need to be copied to private thread memory and can be accessed by any of the process' spawned threads. To declare a variable as private, the programmer can use the 'private' clause followed by a list of variables separated by commas. Similarly, to declare a variable as 'shared', the programmer can use the shared clause followed by a list of variables.

9.3 Hybrid MPI and OMP

Hybrid parallel programs using both distributed and shared memory already exist and have been studied [43]. By utilizing shared memory within each node, communication and data sharing among threads or processes are faster, reducing overhead and improving overall efficiency. Yet, in a paradigm where we need to communicate data between different machines (or node when referring to HPC clusters), distributed memory protocols like MPI will always be necessary.

With the actual trend of new architectures tending towards increasing the amount of cores and threads per node [10,11,12,44], hybrid parallelizations will be needed to take full advantage of the new machines.

MPI could also be considered to implement shared memory parallelism in a hybrid model. And while MPI is generally known for providing better parallel scaling than OpenMP because of the forced data locality, a major drawback of the pure MPI implementation is its inefficient memory usage [45]. Another point in favor of OpenMP is that MPI is more complex to implement. OpenMP offers a simpler programming interface, enabling developers to express parallelism in a more intuitive manner.

10 Most time consuming functions

In this chapter we identify the most time consuming functions in the NEMO model and show an overview of their execution with the help of traces produced instrumenting the code with the Extrae library and visualized using the Paraver tool. This will identify the functions that we can work on to improve the model's scalability.

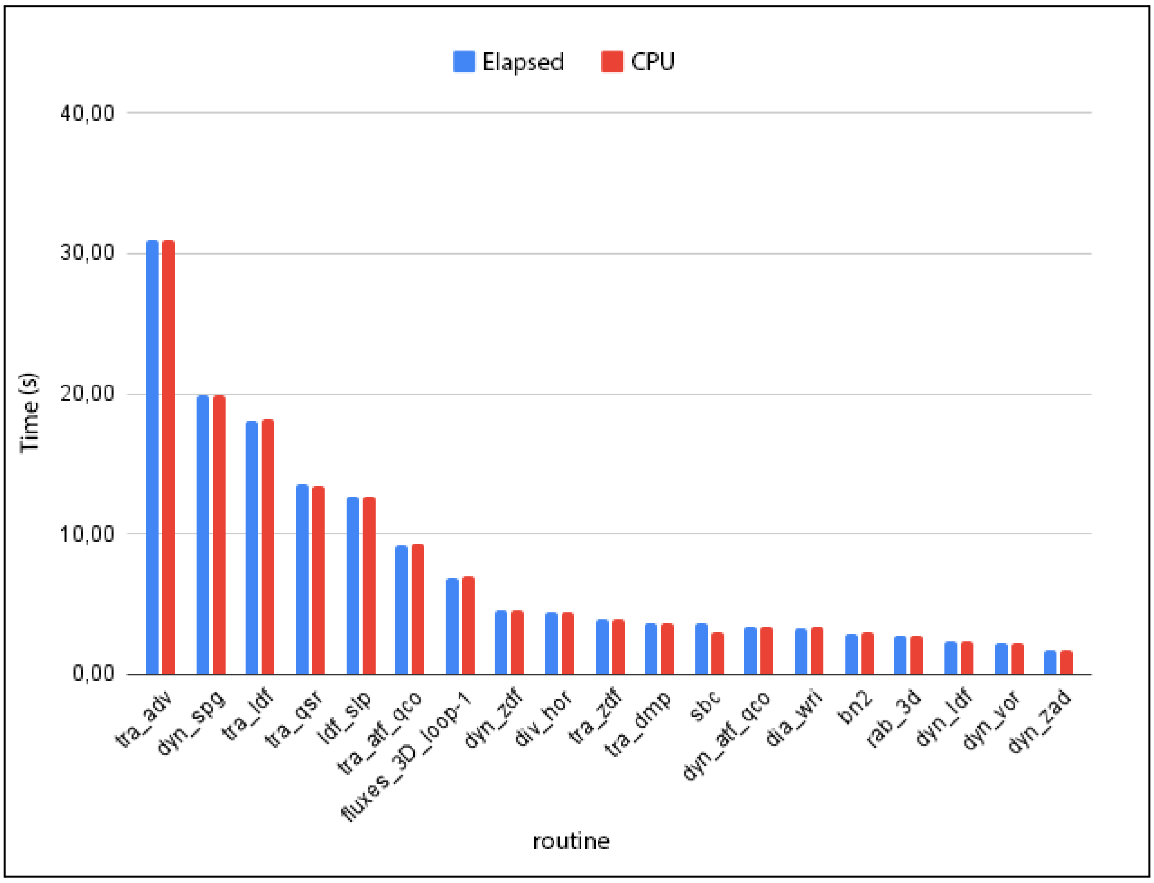


Figure 6. NEMO average routine time cost plot

Figure 6 shows a plot of the average time spent for each of NEMO's functions obtained with the default benchmarking timers (for details see the NEMO manual section 14.5 [31]) and ordered in decreasing order from left to right. Elapsed time corresponds to the average real time that passed since the start of the routine until it ended. CPU time corresponds to the average time it spent executing in the CPU. The most time consuming function according to the plot is tra_adv_fct.

10.1 tra_adv_fct

The 'tra_adv_fct' function implements numerical calculations for the advection of active tracers in seawater. Tracers are any physical or chemical quantity that can be used to track water masses, such as temperature, salinity, and other factors that can alter the flow.

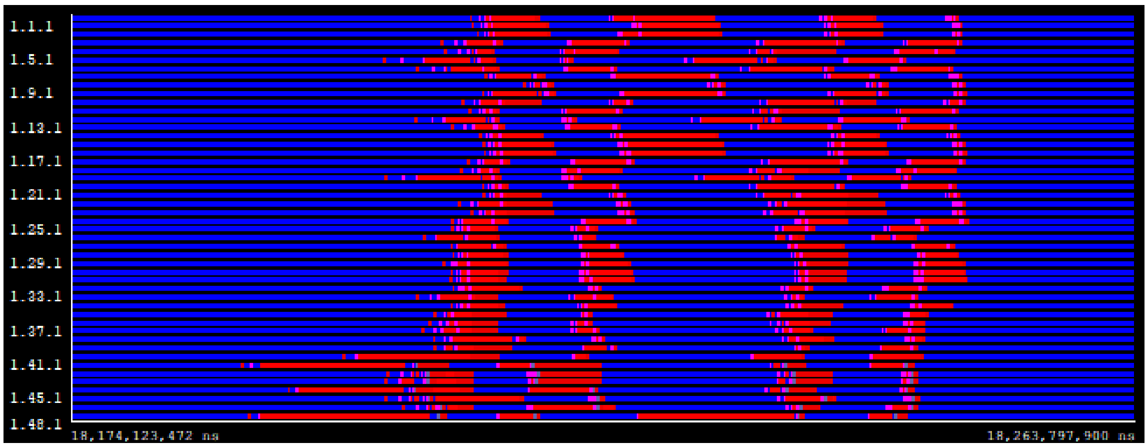


Figure 7. Paraver default view of tra_adv_fct during a timestep

🟦 Running	🟪 MPI ISend
🟥 MPI Wait	

Table 4. Average times the function spent in each state

Time (ns)	**Running**	**MPI Wait**	**MPI Send**
Average	74,107,128.79	15,016,471.89	2,727,032.21

Figure 7 shows the execution of the tra_adv_fct function in a timestep of the NEMO model, viewed with Paraver, in a trace from an execution instrumented with the extrae library. This view shows every process and the states they go through the execution timeline. The thread numeration X.Y.Z means “thread Z of the process Y of the application X”. The nanoseconds indicated in the horizontal axis are the timestamps corresponding to the start and end of the viewed timeline segment.

In Table 4 we have the averages of the time the function spent in each state across all processes. With this information we can see that, on average, 80% of the function time is computation and 20% are communications.

11 Improving function tra_adv_fct

We decided to focus this work on the function tra_adv_fct for being the most time consuming function. It also has a much higher computing time percentage with respect to the total execution time compared to dyn_spg_ts. Due to this fact, it is also likely that it would benefit more from a hybrid MPI/OpenMP parallelization.

In the following section 11.1 we will first discuss the workflow of the function in NEMO 4.2 and some basic points to consider for the OpenMP implementation. We will then present the three proof of concept versions proposed in this work, which use a hybrid implementation of MPI and OpenMP in the tra_adv_fct function, each with a different parallel strategy:

- the *loops* version
- the *parallel tracer loop* version
- the *nested* tracer loop version

11.1 Original workflow in NEMO 4.2

The main body of the function is a loop, with one iteration per tracer. From now on we will call this loop the "tracer loop".

```
DO jn = 1, kjpt  ! tracer loop
   DO_3D
      ! some calculations
   ENDDO

   call lbc_lnk(...) ! call to the communications wrapper

   DO_3D
      ! some calculations
   ENDDO

   .
   .
   .
ENDDO  ! end of the tracer loop
```

Figure 8. Pseudocode of the tracer loop

Figure 8 shows a pseudocode to illustrate the scheme of the tracer loop code. Inside the tracer loop there are several three dimensional loops which are the most time consuming parts and some calls to other functions which include calls to the communications wrapper lbc_lnk.

With the set of flags used in our simulations, the total number of calls to lbc_lnk inside a single iteration of the tracer loop is two.

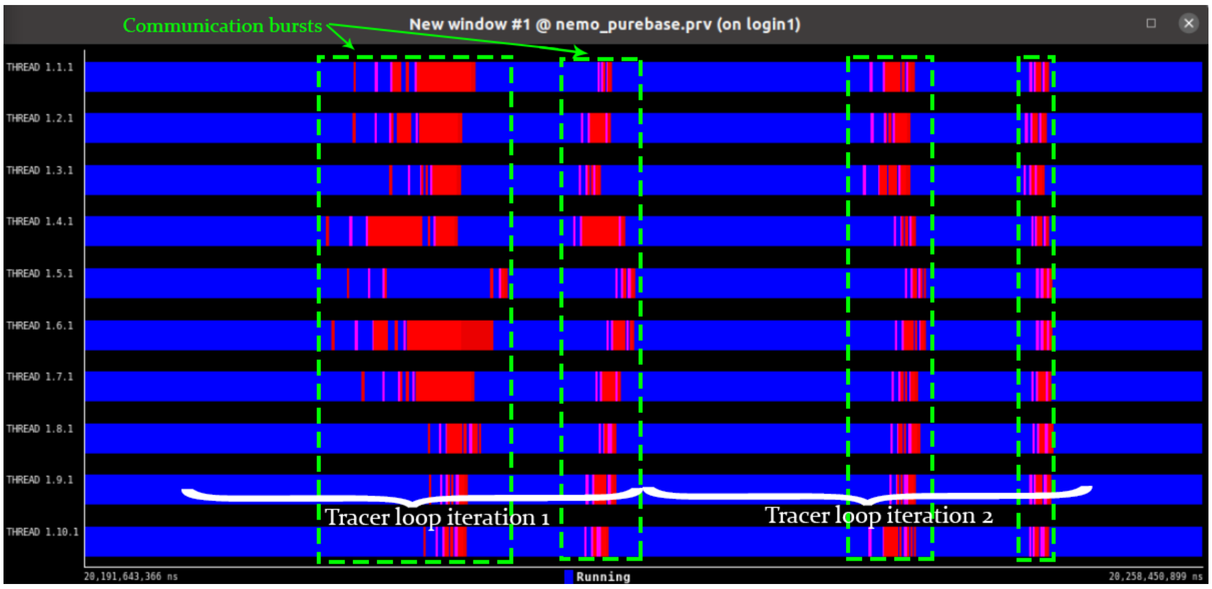


Figure 9. Trace of the original function

🟦 Running	🟪 MPI ISend
🟥 MPI Wait	

Figure 9 shows a trace of an execution of the tra_adv_fct function during a timestep, zoomed in the first 10 processes to illustrate the behavior of the function. We can clearly distinguish the four bursts of MPI communications in between computation time. Every one of these bursts corresponds to a call to the communications wrapper *lbc_lnk*, and every two of these calls to an iteration of the tracer loop.

11.2 About the OpenMP programming

In this section we discuss some criteria and considerations we had during the planification and implementation of the hybrid strategies for the proofs of concept such as the scope of the multithreaded region and the constructs used to divide the workload among the threads.

11.2.1 The parallel region constructor

As explained in the OpenMP section of the *Parallelization Paradigms* chapter, since a multithreaded region needs to be created/opened and destroyed/closed, two directives are necessary, forming a so-called directive-pair: **!$OMP PARALLEL/!$OMP END PARALLEL**.

In order to minimize the overhead associated with opening the multithreaded region [46], our approach focused on limiting the number of OMP PARALLEL directives employed within the code. By making the multithreaded region start right before the tracer loop and closing the region after the loop ends, we can enclose all of the function's computation within a single multithreaded region.

11.2.2 Work-sharing constructs

The work sharing constructs from the OpenMP API [14] we used within the multithreaded region are the following:

!$OMP DO/!$OMP END DO
This directive-pair makes the immediately following do-loop to be executed in parallel.
For example:

```
!$OMP DO
do i = 1, 1000
... enddo
!$OMP END DO
```

The iterations are distributed among the threads, so that each one takes part of the workload of the loop. In the example with 1000 iterations, if 100 threads are spawned, then in general each thread computes 10 of the iterations: thread 0 computes from 1 to 10, thread 1 from 11 to 20 and so on.

!$OMP SINGLE/!$OMP END SINGLE
The code enclosed in this directive-pair is only executed by one of the threads, namely the one who first arrives at the opening-directive !$OMP SINGLE. All the remaining threads wait at the implied synchronization in the closing-directive !$OMP END SINGLE, unless the NOWAIT clause is specified.

11.2.3 Multithreaded Workload and SubroutinesOther considerations

Since we already measured the time distribution in section 10.1 we know that the tra_adv_fct function time has the following distribution: 40% computation time, 40% subroutines computation time, 20% communications time. Newly declared variables inside an OpenMP multithreaded region are private by default. This presents a problem when we find subroutine calls between the !$OMP PARALLEL and !$OMP END PARALLEL. Because when we parallelize the calculations and the results are written in these private variables threads won't have access to the data they need for the next steps.

The straightforward solutions for this problem are very inefficient:

1. Close and re-open the parallel region specifying the new variable safely: this would add a lot of overhead from spawning/despawning the threads every time.

2. Declare these variables with the SAVE attribute: In Fortran the variables declared with this attribute are shared by default in OpenMP. This would also force them to be stored in memory from the first subroutine call where they are used until the last one. It would mean that they would be occupying memory for the entire execution of the model, even when they are not being used. Thus penalizing the timings of other functions.

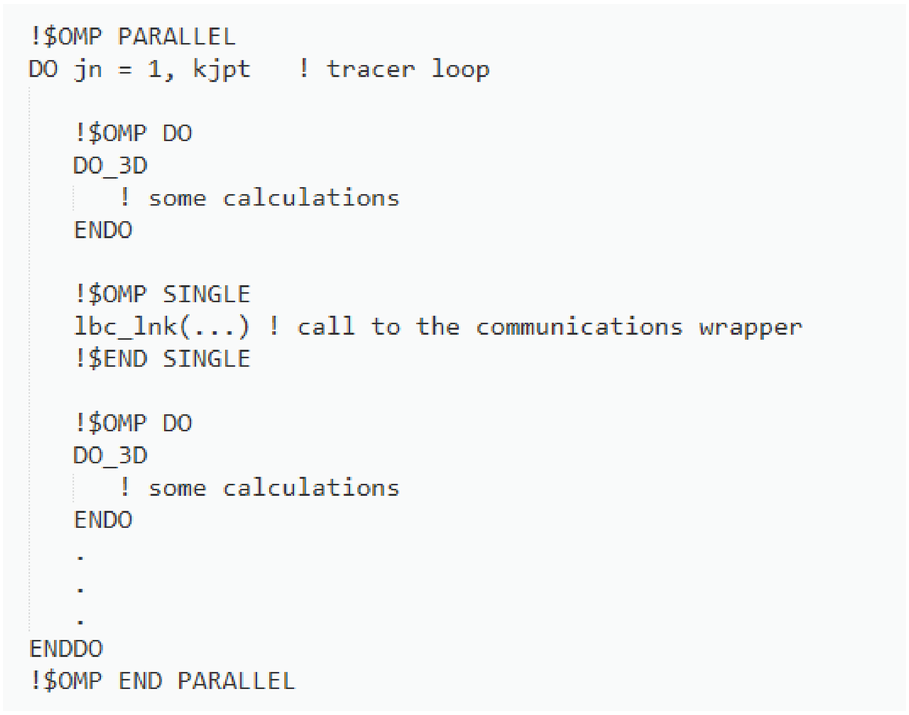
The downsides of these straightforward methods would hinder the performance results and not correspond with a practical full hybrid NEMO implementation. But our time is limited and having to find workarounds for parallelizing subroutines would increase the time requirement. For these reasons our *loops* and *nested* strategies focused on applying OpenMP parallelism to the 40% computation time that corresponds solely to tra_adv_fct code, while using SINGLE constructs to keep subroutine calls within the declared multithreaded region executed in a single thread. The coarse grain parallelism of the *parallel tracer loop* strategy allowed us to use OpenMP parallelism in the 100% of the function.

11.2.4 MEMORY REQUIREMENTS

Depending on the parallelization strategy, we need to declare some variables as private to avoid race conditions. This means that they need to be copied once for every thread increasing the amount of memory needed per process.

11.3 Loops Version

In this first hybrid version we keep the workflow of the original function and divide the workload of the 3D loops equally among the available threads. This is achieved by adding to these loops the construct OMP DO as described in the previous section *Work-sharing Constructs*. In this way the loops' iterations will be distributed among the threads spawned by OpenMP.



```
!$OMP PARALLEL
DO jn = 1, kjpt   ! tracer loop

   !$OMP DO
   DO_3D
      ! some calculations
   ENDO

   !$OMP SINGLE
   lbc_lnk(...) ! call to the communications wrapper
   !$END SINGLE

   !$OMP DO
   DO_3D
      ! some calculations
   ENDO
   .
   .
   .
ENDDO
!$OMP END PARALLEL
```

Figure 10. Pseudocode of the tracer loop in the *loops* version

Figure 10 shows a pseudocode of the tracer loop in the loops version to illustrate the mentioned changes in the code. Notice that, as mentioned in the previous section, the multithreaded region will be opened once per timestep before the tracer loop starts and closed after it finishes. And also, there are SINGLE clauses to encapsulate calls to external functions that are not parallelized and consequently correspond to code executed in a single thread

11.3.1 Memory requirements

The only private variables declared in this version were auxiliary variables with a negligible impact on the memory demand and no other changes were introduced in the declared variables of the function.

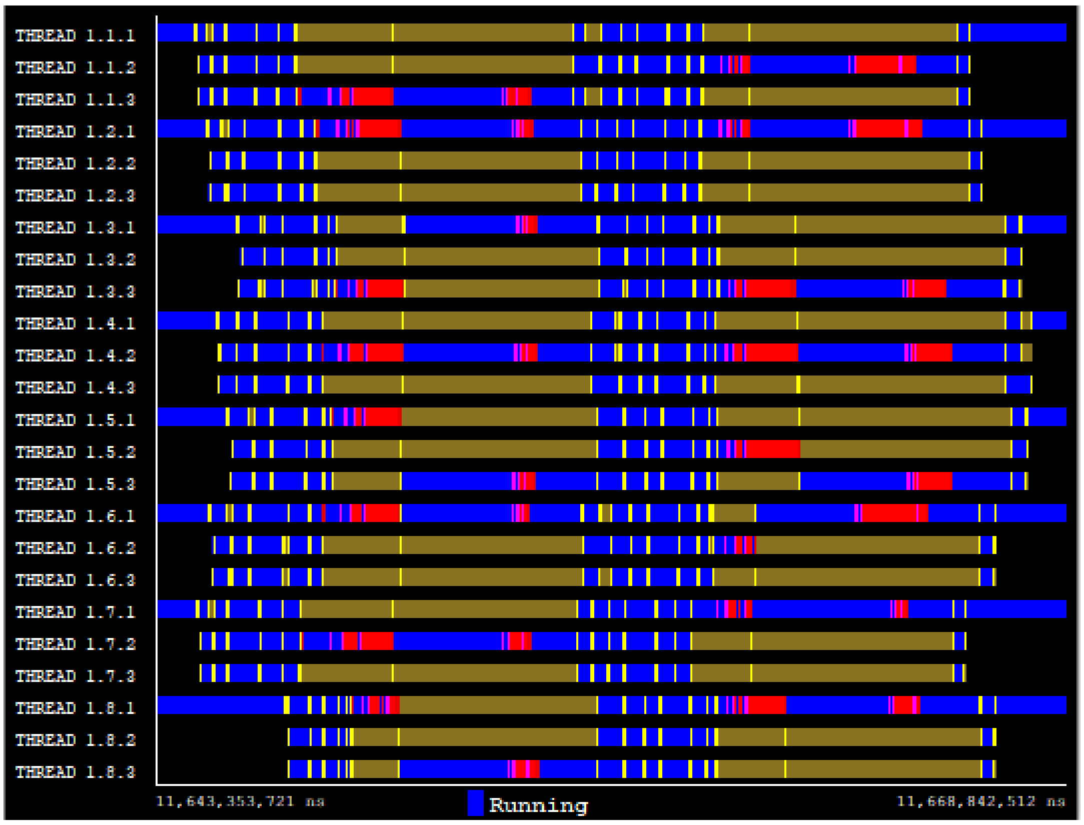


Figure 11. Trace of the *loops* version

🟦 Running	🟪 MPI ISend
🟥 MPI Wait	🟫 Synchronization
🟨 Fork/join and Scheduling	

The trace in Figure 11 illustrates the effect of the implemented multithreading in this version: The number of OpenMP threads of the execution was 3, and so, we can see that for every MPI process, two additional threads are spawned in a concrete time span. This corresponds to the time that goes from the OMP PARALLEL until reaching the OMP END PARALLEL directive.

Fork/Join time in this image refers to the time the threads are being spawned, despawned, and/or distributing workload from the OpenMP work-sharing constructs (DO and SINGLE in this case). Thanks to this we can see when it enters a new construct.

Synchronization time is when a thread is waiting for other threads before proceeding. It can mostly be seen when the function enters a SINGLE construct: the threads that don't get workload will have to wait for the one working.

11.4 Parallel Tracer Loop Version

In this version only one OMP DO construct will be applied and it will be on the tracer loop. The intent behind this is to utilize the multithreading to parallelize all the computation time in the function as well as the communication calls to see if this could achieve a better speedup.

The tracer loop only has two iterations, consequently this version will not benefit from spawning more than two OpenMP threads. What we will see in the trace is that for every MPI process there are two threads on the tracer loop, each one taking on the full length of one of the iterations, including the parts that had to be put inside SINGLE constructs in the *loops* version. SINGLE constructs are not necessary here since each iteration of the tracer loop is completely independent from the other and each will be executed by one thread.

```
!$OMP PARALLEL
!$OMP DO
DO jn = 1, kjpt  ! tracer loop

   DO_3D
      ! some calculations
   ENDO

   lbc_lnk(...) ! call to the communications wrapper

   DO_3D
      ! some calculations
   ENDO
   .
   .
   .
ENDDO
!$OMP END PARALLEL
```

Figure 12. Pseudocode of the tracer loop in the *parallel tracer loop* version

11.4.1 Memory Requirements

This strategy is more memory demanding since, to avoid race conditions, we need to declare them as private matrices whose size scales with the process' domain size. It's worth remembering that private variables are stored in the memory from the original thread's stack and then copied once more to the stack of each working thread, so that the memory requirement will increment following this formula:

$$M_n = M_o * (1 + omp)$$

Where M_n is the memory required by the new version, M_o is the memory required by the original version, *omp* is the number of threads per process. For executions with a low amount of MPI processes we even needed to increase the size of the thread stack for it to execute without errors.

11.4.2 Multiple Thread Communications

The main problem for parallelizing the iterations of the tracer loop are the MPI communications that are performed inside each of them. As we saw in the trace in Figure 9, the two loop iterations result in four total communication bursts, two for each tracer. By parallelizing the tracer loop we will have both iterations running in parallel so we will still have four different calls to lbc_lnk but looking at the timeline they will appear as two sequential communication bursts.

In multithreaded MPI the *tag* parameter is used so that threads in the receiving process can properly match different messages with the same sender process ID. In the original implementation of the lbc_lnk call, the tags assigned are a numeric value within the range from 1 to 8 as to differentiate the 8 possible neighbors: North, North-East, East, South-East, South, South-West, West; but the tag is the same in the first and second iteration of the tracer loop. This would result in a race condition between the communications when the loop is parallelized, as it could be that the messages are received and processed by a thread that does not correspond to the same tracer loop iteration.

We solved this by adding *10*iteration_index* (where *iteration_index* identifies the current iteration of the tracer loop) to this value to make sure that the values are different in between iterations, but stay consistent within the same iteration. So the Tags in the first iteration will range from 11 to 18 and the ones in the second from 21 to 28, and could easily be further scaled for cases with more parallel communications. With this method we will be sure that the messages being processed are the correct ones.

11.4.3 Allowing Asynchronous MPI Progress

In order to allow multiple threads to make progress on multiple MPI communications in the same process at the same time, we need to enable Asynchronous Progress and also indicate the number of threads per process that will be performing asynchronous progress. In our implementation, which is using Intel MPI, this is done by setting the following environment variables:

- **I_MPI_ASYNC_PROGRESS**: set to true allows asynchronous progress
- **I_MPI_ASYNC_PROGRESS_THREADS**: the number of threads per process that will perform asynchronous progress (two in our case)

11.4.4 Merging the OMP PARALLEL and OMP DO

In this section we discuss an unexpected behavior that showed up during the implementation of this strategy that we could identify thanks to the Extrae and Paraver tools.

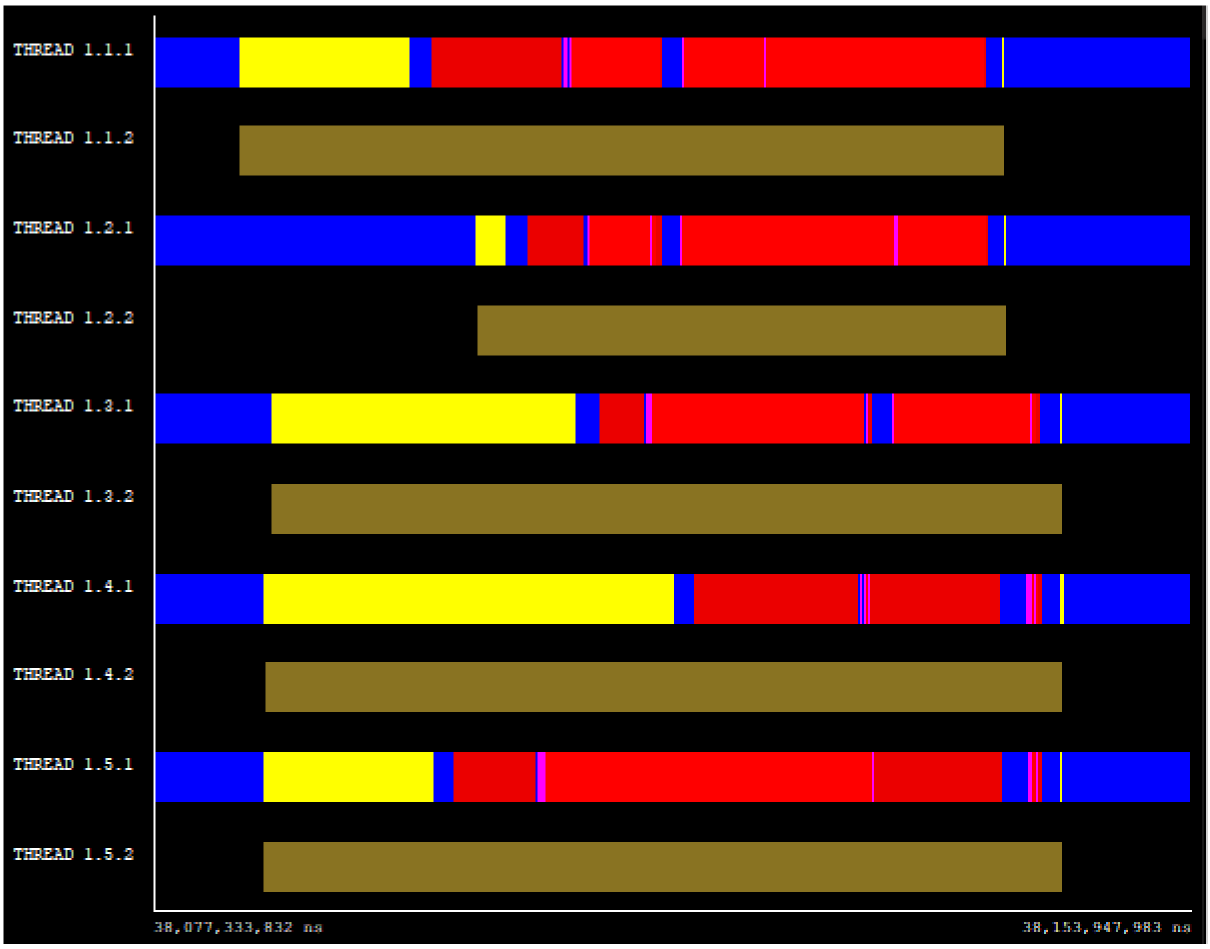


Figure 13. A timestep exemplifying the workload bug in the *parallel tracer loop* version

🟦 Running	🟪 MPI ISend
🟥 MPI Wait	🟫 Synchronization
🟨 Fork/join and Scheduling	

Figure 13 shows a trace of the second hybrid version where the parallelization is not working as intended. In it we can see that a second thread was being spawned but the first thread was taking all the workload from the tracer loop and the second thread was doing nothing. This problem can be clearly seen because the second thread is only in Synchronization state, which means that it's taking none of the loop's workload and thus there is no effective parallelization.

This was solved by merging the OMP PARALLEL and OMP DO clauses into a single OMP PARALLEL DO. In this way, the workload distribution is done alongside the thread spawning (fork), whereas before the first thread could, sometimes, exit the fork and finish the first iteration of the loop before the second thread could take workload from the loop.

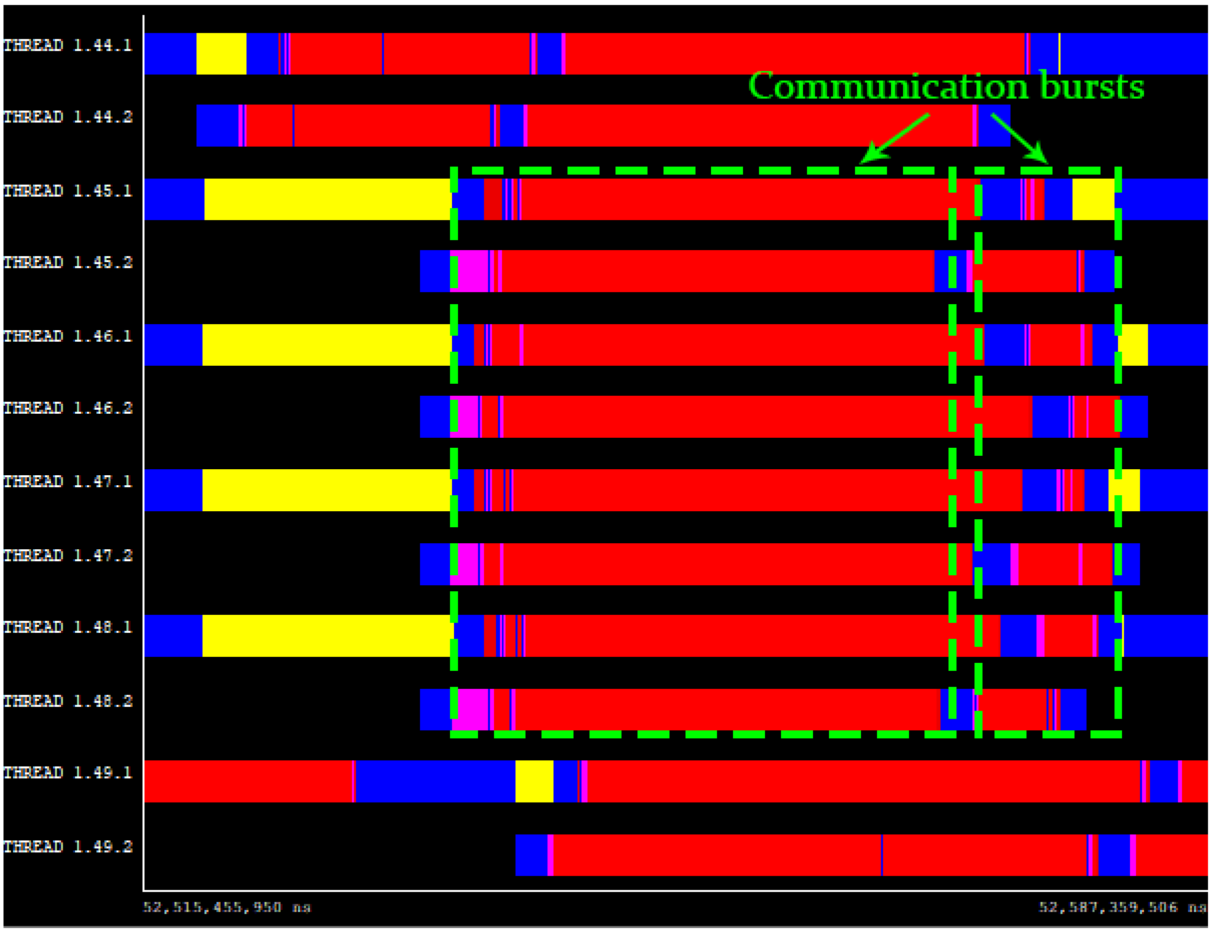


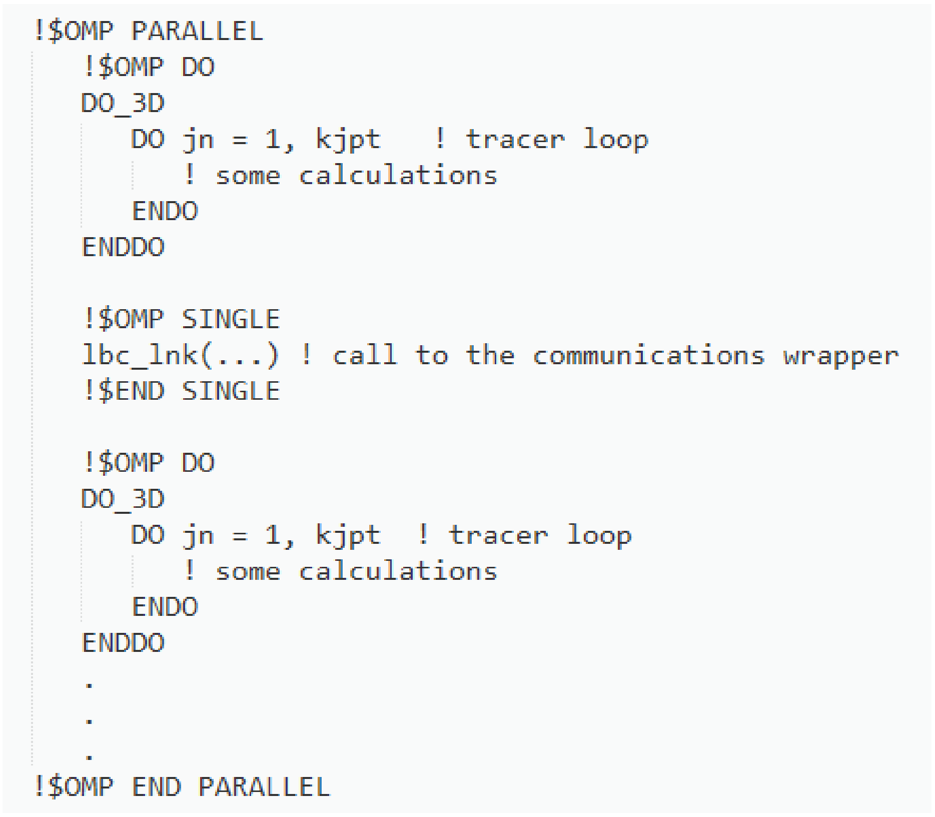
Figure 14. Trace of the *parallel tracer loop* version after fixing the bug

🟦 Running
🟪 MPI ISend
🟥 MPI Wait
🟫 Synchronization
🟨 Fork/join and Scheduling

Figure 14 shows a trace of a timestep of tra_adv_fct in the second hybrid version after merging the directives where we can see that the workload is being distributed as intended among the threads. Due to this version not having SINGLE constructs there are no omp synchronization times and the MPI calls have been parallelized (the Paraver view in Figure 14 has been focused on the processes 45 to 48 for visual clarity of this phenomenon).

11.5 Nested Tracer Loop Version

For this third proof of concept, we reorganize the original nested loop sequence: from the various loops being nested inside the tracer loop we now make the tracer loop to be nested inside each of the other loops. This will allow us to merge the calls to *lbc_lnk* from both tracers from having 4 to having 2 but now the size of the message is increased by communicating the data of both the tracers in a single call. Additionally we add OMP DO and SINGLE constructs with the same strategy of the *loops* version.



```
!$OMP PARALLEL
   !$OMP DO
   DO_3D
      DO jn = 1, kjpt   ! tracer loop
         ! some calculations
      ENDO
   ENDDO

   !$OMP SINGLE
   lbc_lnk(...) ! call to the communications wrapper
   !$END SINGLE

   !$OMP DO
   DO_3D
      DO jn = 1, kjpt  ! tracer loop
         ! some calculations
      ENDO
   ENDDO
   .
   .
   .
!$OMP END PARALLEL
```

Figure 15. Pseudocode of the tracer loop in the *nested* version

Figure 15 shows a pseudocode to illustrate the changes in the tracer loop code for the *nested* version.

11.5.1 Memory requirements

In the original, the calculations for every tracer were made one after another, which allowed the re-use of the same variables for all of the tracers.In this new version we are parallelizing the calculations of the tracers, so we need to be able to store all the values at the same time, which causes the memory required to be multiplied by the number of tracers.

This strategy changes the memory requirements of the function by the following formula:

$$M_n = M_o * tracers$$

Where M_n is the memory required by the new version, M_o is the memory required by the original version, and $tracers$ is the number of tracers to be calculated in the simulation.

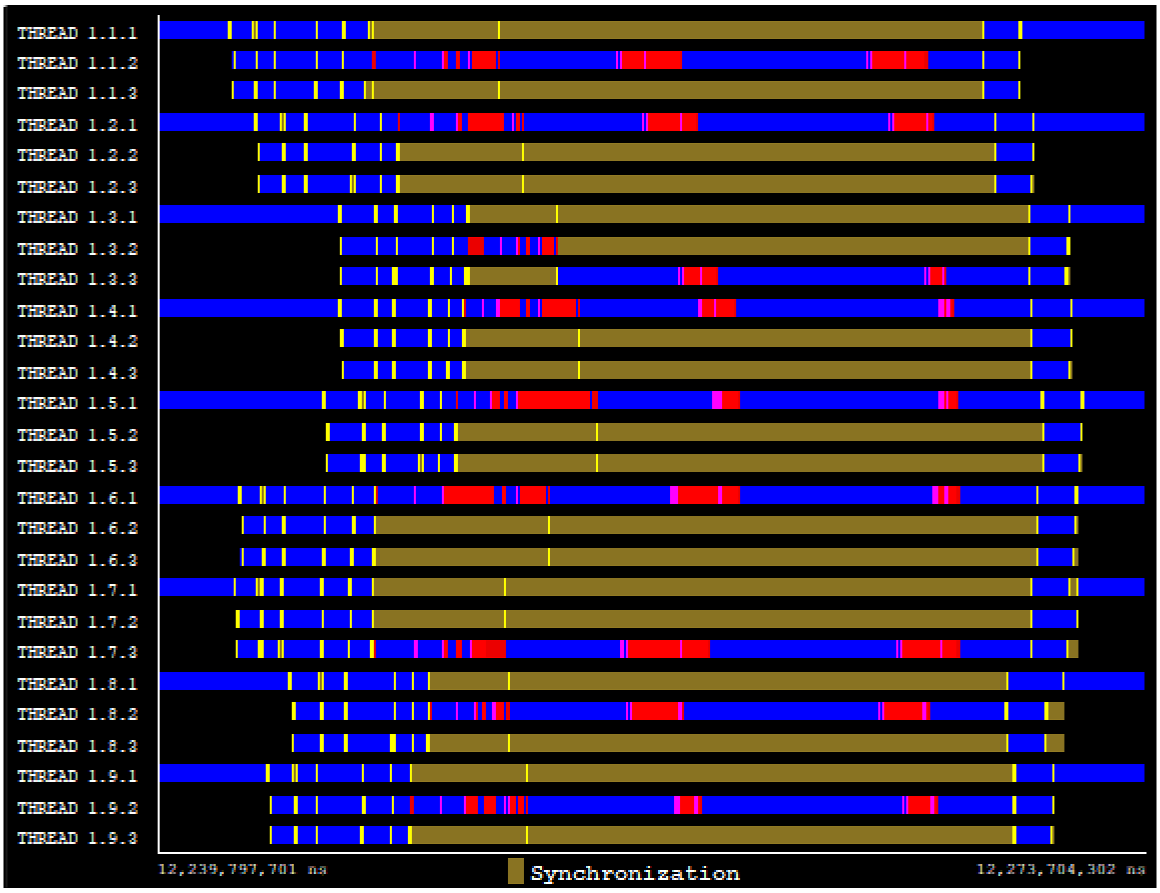


Figure 16. Trace of the *nested* strategy version of the function

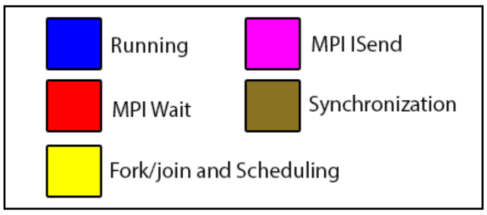


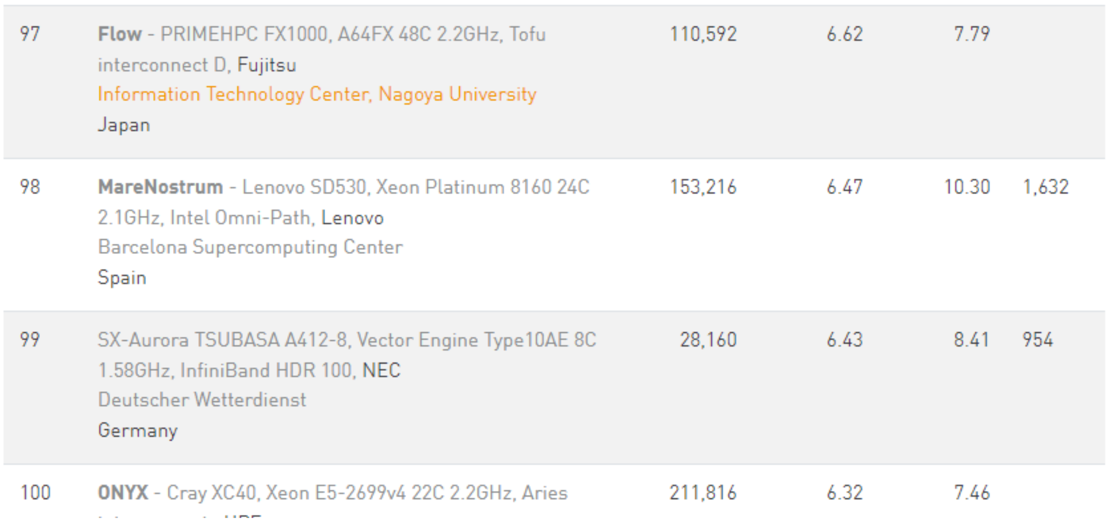
Figure 16 shows the function tra_adv_fct in a timestep of the nested. As we can see, the behavior of this version is similar to the first one's Figure 11 in the way it distributes computing time and just one thread takes care of the SINGLEs constructs, something that we were already expecting since they use the same hybrid MPI/OpenMP parallel structure. The main difference in the workflow is seen in the communications: this version has half the number of communication calls, but they are double in size since the data sent is the sum of the original data. We can see them every time the threads enter a SINGLE construct and two of the three threads of the process enter the Synchronization state

12 Experiments

In this chapter we talk about the architecture of the platform in which the experiments have been run as well as the specific libraries used for the compilation of the model. We explain the methodology and steps carried through the experiments and finally we discuss the results as well as some additional tests carried out to understand the results obtained.

12.1 Marenostrum4 Architecture

For the development of this project the BSC has granted us user credentials with permission to run jobs on the Marenostrum4 supercomputer to execute the necessary tests through their HPC network. The Marenostrum4 supercomputer is a highly significant machine in high-performance computing, offering immense computational power for complex research across disciplines. It ranked 13 in the top 500 [47] most powerful commercially available computer systems the year it was installed (2017) and is still within the top 100 of the list to this day.



97	**Flow** - PRIMEHPC FX1000, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu Information Technology Center, Nagoya University Japan	110,592	6.62	7.79	
98	**MareNostrum** - Lenovo SD530, Xeon Platinum 8160 24C 2.1GHz, Intel Omni-Path, Lenovo Barcelona Supercomputing Center Spain	153,216	6.47	10.30	1,632
99	SX-Aurora TSUBASA A412-8, Vector Engine Type10AE 8C 1.58GHz, InfiniBand HDR 100, NEC Deutscher Wetterdienst Germany	28,160	6.43	8.41	954
100	**ONYX** - Cray XC40, Xeon E5-2699v4 22C 2.2GHz, Aries	211,816	6.32	7.46	

Figure 17. Marenostrum4 position in the top 500 list [47]

The tests in this project have been executed in the Compute nodes which have the following specifications:

- 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz for a total of 48 cores per node
- L1d 32K; L1i cache 32K; L2 cache 1024K; L3 cache 33792K
- 96 GB of main memory 1.880 GB/core (216 nodes high memory, 10368 cores with 7.928 GB/ core)
- 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E adapter
- 10 Gbit Ethernet
- 200 GB local SSD available as temporary storage during jobs

The processors support well-known vectorization instructions such as SSE, AVX up to AVX-512.

The Marenostrum Slurm cluster has different queues to request job executions. Our credentials gave us access to the *bsc_es* and *debug* queues. Due to the bsc_es queue being constantly loaded with higher priority jobs it was very difficult for us to run jobs there, we could be waiting for jobs to be executed for more than one week. For this reason we used the debug queue for all of our experiments. This limits the maximum number of nodes we can request up to sixteen.

12.2 Experiment Parameters

When submitting jobs to the Slurm scheduler from the Marenostrum cluster, the parameters we need to take into account differ significantly from pure MPI implementations to Hybrid MPI/OMP programs.

For pure MPI jobs the settings are straightforward, one can ask for a specific number of processes and use all of them for MPI processes..

For every execution using a hybrid implementation of MPI and OpenMP we need to request full nodes without filling them completely with MPI processes as to have available cores for the OMP threads, so one needs to define at least 3 parameters:

- **Number of Nodes**: The amount of nodes that will be requested for the execution
- **Tasks per Node**: The amount of MPI processes that will be executed in each node
- **OMP_NUM_THREADS**: The number of threads every MPI process will be spawning

With this parameters the total number of cores will be defined by the following formula:

$$Number\ of\ Nodes\ *\ Tasks\ per\ Node\ *\ Omp\ Num\ Threads$$

Every node has 48 cores: this means that the total number of cores should be a multiple of 48 to make complete use of the nodes we reserve for the execution (in both MPI only and hybrid implementations).

At the same time *Tasks per node * Omp Num Threads* should be equal to and can't be greater than 48 as that's the maximum number of cores per node.

There has to be the same amount of MPI processes per node and every process will spawn the same amount of OpenMP threads, so the total number of MPI processes always will be a multiple result of *Tasks per Node * Number of Nodes*.

At the start of the execution, the domain of the simulation is divided into subdomains to be distributed each to a different MPI process. In [2] we have the study of the current domain decomposition algorithm which focuses on obtaining the domain decomposition with the smaller maximum subdomain size while eliminating unnecessary data to be processed from the domain. This means that the more MPI processes are created at the start of the execution, the more subdomains the domain will be subdivided into, and the smaller these subdomains will be. The smaller the subdomains the less computing each process has to do.

At the same time, increasing the number of threads per process means we need to have cores available for the new threads, thus taking away potential MPI processes from the domain decomposition and making the resulting subdomains bigger. And subdomains being bigger means increased chances of having more land points inside the subdomains that have both land and water points which means useless calculations added.

12.3 Scalability tests

In this section we present the results of a scalability test we performed in order to decide the resolution of the grid we should use for experiments aimed at testing the performance of our new implementations.

ORCA2 is the grid resolution usually used for development, as it is a coarse resolution that allows for fast testing and being relatively inexpensive in computational resources. For reference, the minimum number of cores where NEMO executes using an ORCA2 grid is four.

ORCA1 is a finer resolution grid with a higher demand of computational resources. For reference, the minimum number of cores where NEMO executes using an ORCA1 grid is thirty, more than seven times the minimum of ORCA2.

Since ORCA2 is a light resolution it may appear that its scalability is poor because it simply has not enough workload to parallelize among different MPI processes and, consequently, not enough workload to parallelize with OpenMP threads.

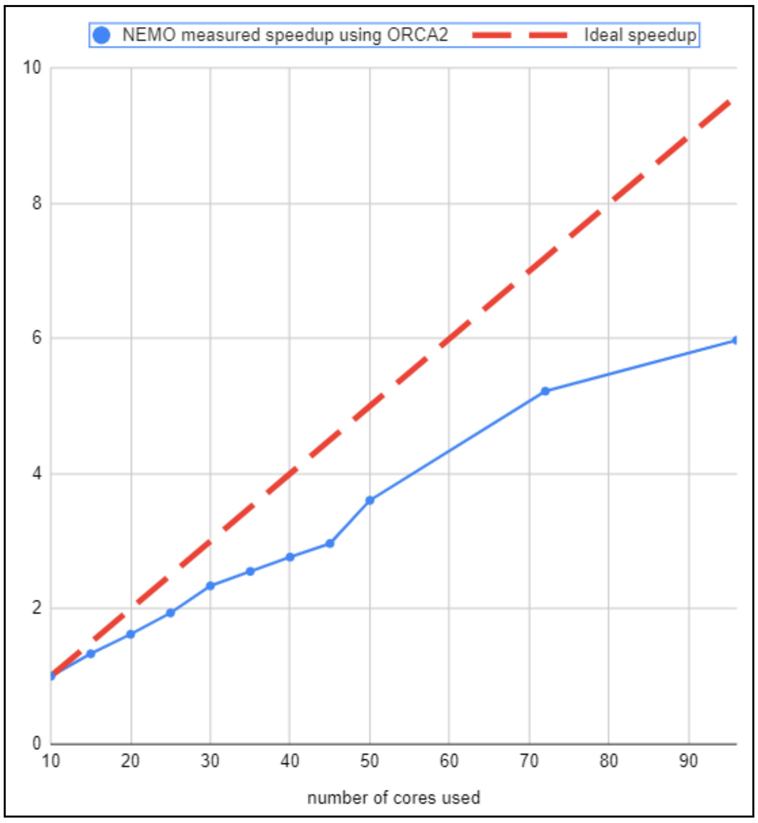


Figure 18. Speedup curve measured for NEMO running an ORCA2 resolution

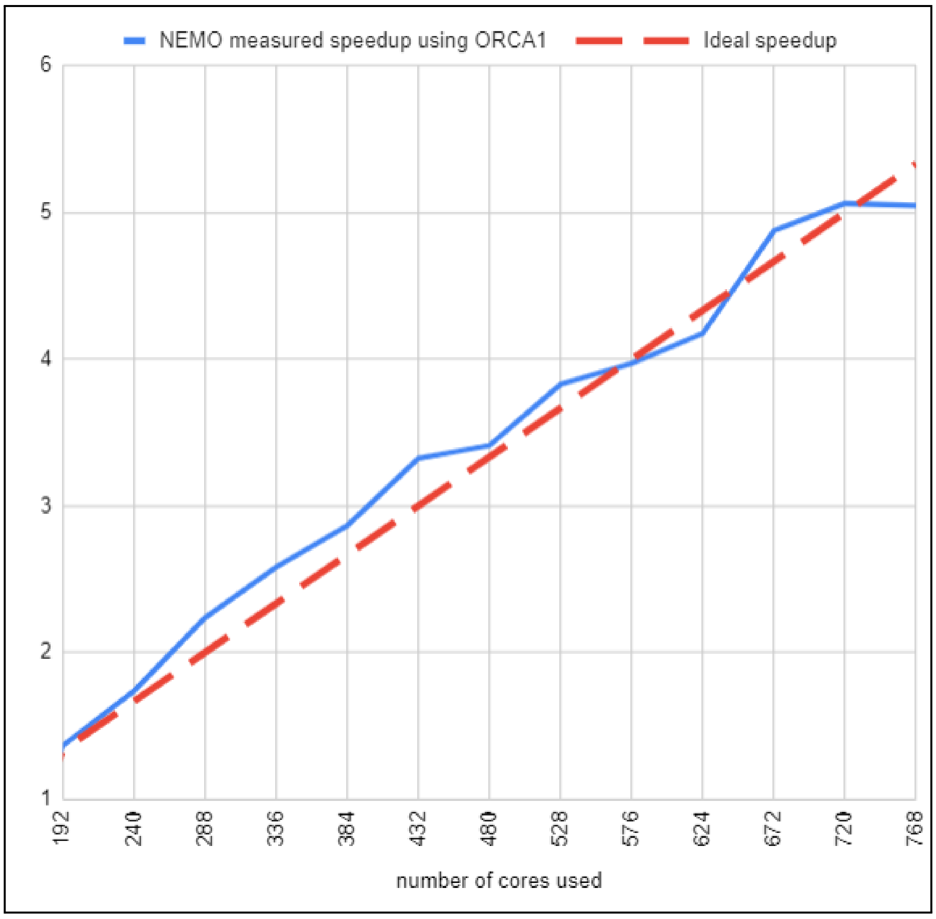


Figure 19. Speedup curve measured for NEMO running an ORCA1 resolution

In Figure 18 we plotted the measured speedup curve for the NEMO model running an ORCA2 resolution, and Figure 19 plots the same measured speedup but running an ORCA1 resolution.From what we see in Figure 18 ORCA2's speedup is lower than the ideal even in the smaller number of cores, and the curve starts to notably lose scalability after the 70 cores. On the other hand from Figure 19 we can conclude that, with a finer resolution, the speedup is ideal up to 720 cores (equivalent to 15 Marenostrum nodes) and then seems to have no speedup after that. This means that we need to test the performance of our hybrid versions in ORCA1 since ORCA2 doesn't have enough workload to make valid performance evaluations on the parallelization of the model.

12.4 Experiment Methodology

In the last sections we have discussed the parameters for experiment configurations and the minimum grid resolution for our performance analyses. Now we will talk about the necessary analyses to assess the objectives of the project.

12.4.1 Function speedup analysis

The first objective of our proofs of concept is to increase the scalability of the bottleneck which we identified to be the function tra_adv_fct. To see whether they achieve this, we measured the speedup obtained at the total time of the function for all our proposed versions and the original to compare them.

12.4.2 Parallel Tracer loop efficiency analysis

Since we already know the *parallel tracer loop* strategy has a limited scalability from the nature of its workload distribution, to evaluate its effectiveness we compared the speedup achieved at the different parts of function to their equivalents in the other versions.

12.4.3 OpenMP performance analysis

Our second objective is to study the performance of OpenMP for a fully hybrid NEMO implementation. For that, we measured and compared the speedup achieved only at the part of the code that is executed using multithreading. It's important to note that we used SINGLE constructs exclusively for calls to other subroutines, which allows us to refer to this measured time as the exclusive time.

12.4.4 Time measurements

In order to measure the the speedup we added timers in the code to measure the following:

- Total time of the function execution
- Time inside the multithreaded region
- Time spent inside the OMP SINGLE directives

With the first timer we gathered the times necessary for the function speedup analyses, and with the other two timers we can compute the exclusive time with the following subtraction:

$$Exclusive\ Time\ =\ Multithreaded\ Time\ -\ Singles\ Time$$

All time measurements used to calculate the speedups presented here are the result of the average from a sample of timesteps of the function in a simulation running the NEMO model with ORCA1 grid resolution.

For the *loops* and *nested* strategies we tested the following node configurations:

- 24 MPI processes and 2 OMP threads
- 12 MPI processes and 4 OMP threads

12.5 Experiments Results and Analysis

In this section we present the results obtained from the various speedup measurements we performed as explained in the previous section.

12.5.1 Overall function scalability

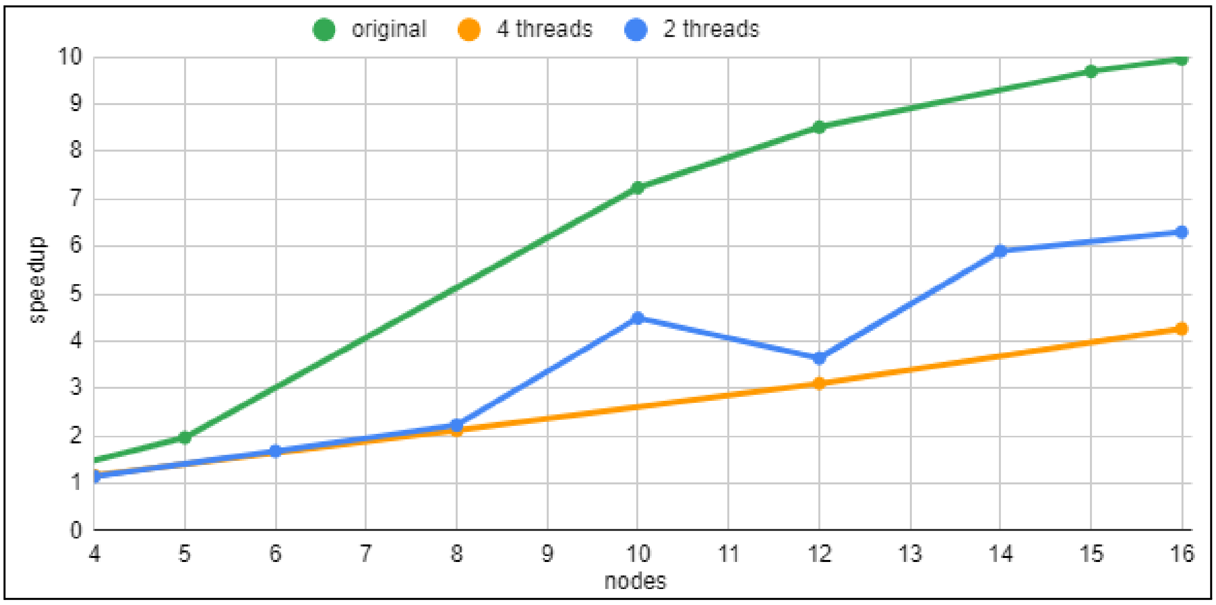


Figure 20. Measured function scalability, *loops* strategy

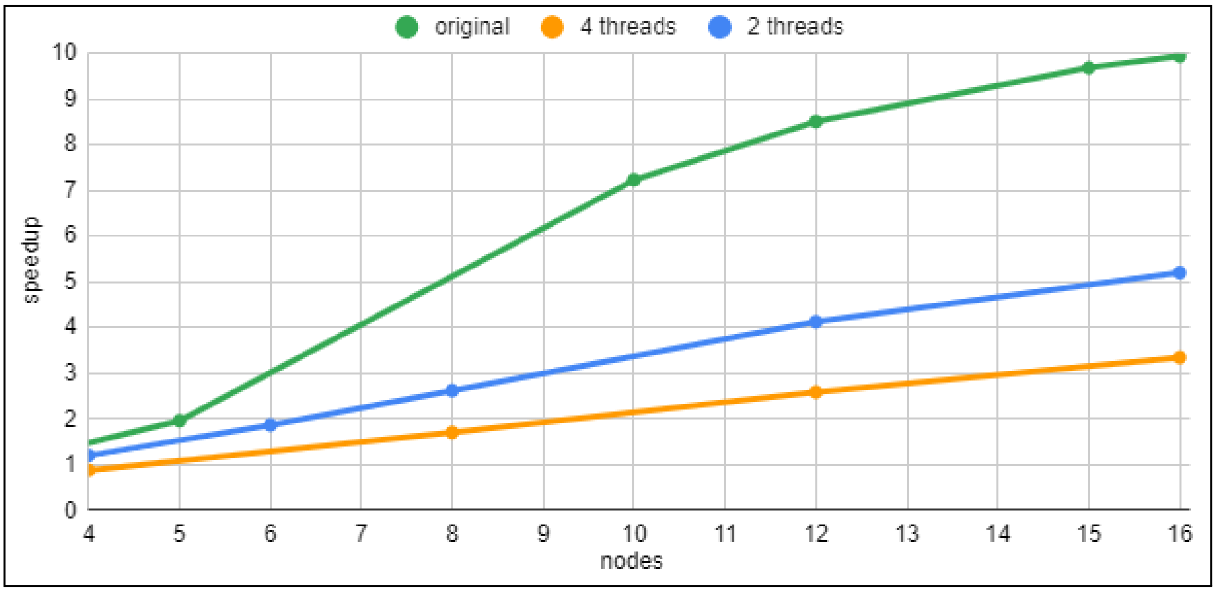


Figure 21. Measured function scalability, *nested* strategy

Figure 20 shows the plotted scalability curve of the evaluated node configurations for the *loops* strategy version against the original. Figure 21 is the same plot but for the *nested* strategy.

In both plots, the scalability curve of the original model starts a bit higher but close to those of the new versions, but we can clearly see that the original version's speedup increases greatly at ten nodes and outperforms both of the new hybrid versions by more than doubling their speedup.

We can also see that the *loops* version generally performs better than the *nested* one, which was already to be expected after having seen the trade-offs in the previous experiment, as well as the fact that the configurations with less threads and more processes have a better overall speedup than those with more threads and less processes.

12.5.2 Parallel Tracer loop Version

For the *parallel TL* strategy we measured the best time we can get by filling all the 16 nodes with the only configuration possible, 24 MPI processes with 2 omp threads per process. Since this version parallelizes the full extent of the tracer loop, this version will be compared with the configurations that achieved a better overall function scalability.

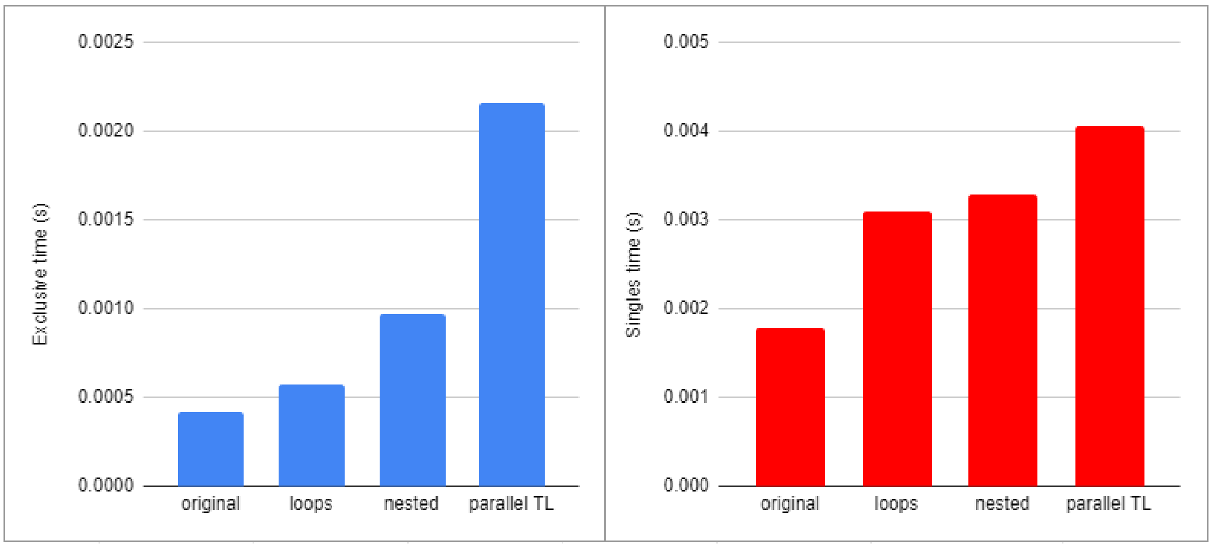


Figure 22. Average time comparisons with the parallel Tracer Loop strategy

Figure 22 shows the best achievable time with the *parallel TL* hybrid strategy using all the resources at our disposal for this project compared to the equivalents from other versions. We can clearly see that the exclusive time of this version more than doubles that of the others. This we can attribute to the overhead seen in Figure 14 when opening the multithreaded region due to the size of the private variables that need to be declared. But the fact that the time inside the SINGLEs in the other hybrid versions is still inferior by nearly 25% of the equivalent part of the code in this version, already indicates to us that this workload distribution based on the tracers would still be inefficient compared to any of the other versions.

12.5.3 OpenMP Performance analysis

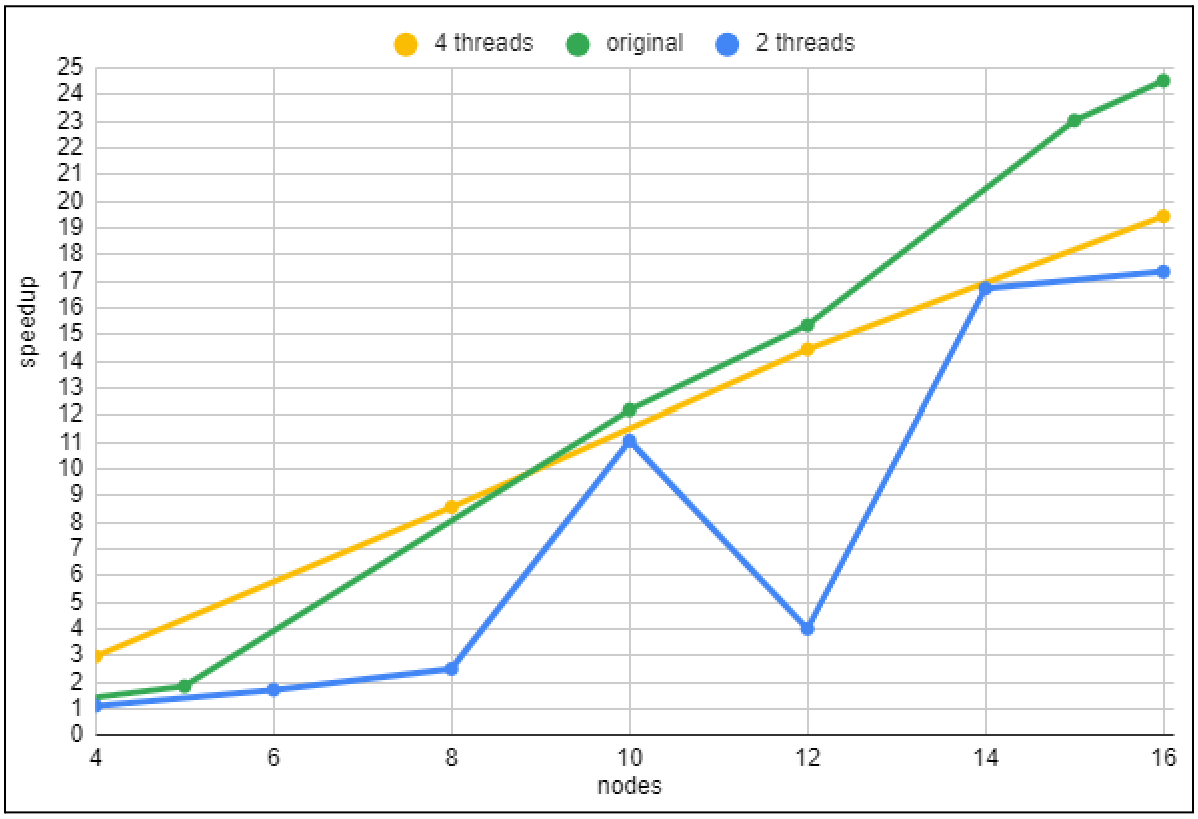


Figure 23. Measured exclusive time scalability, *loops* version

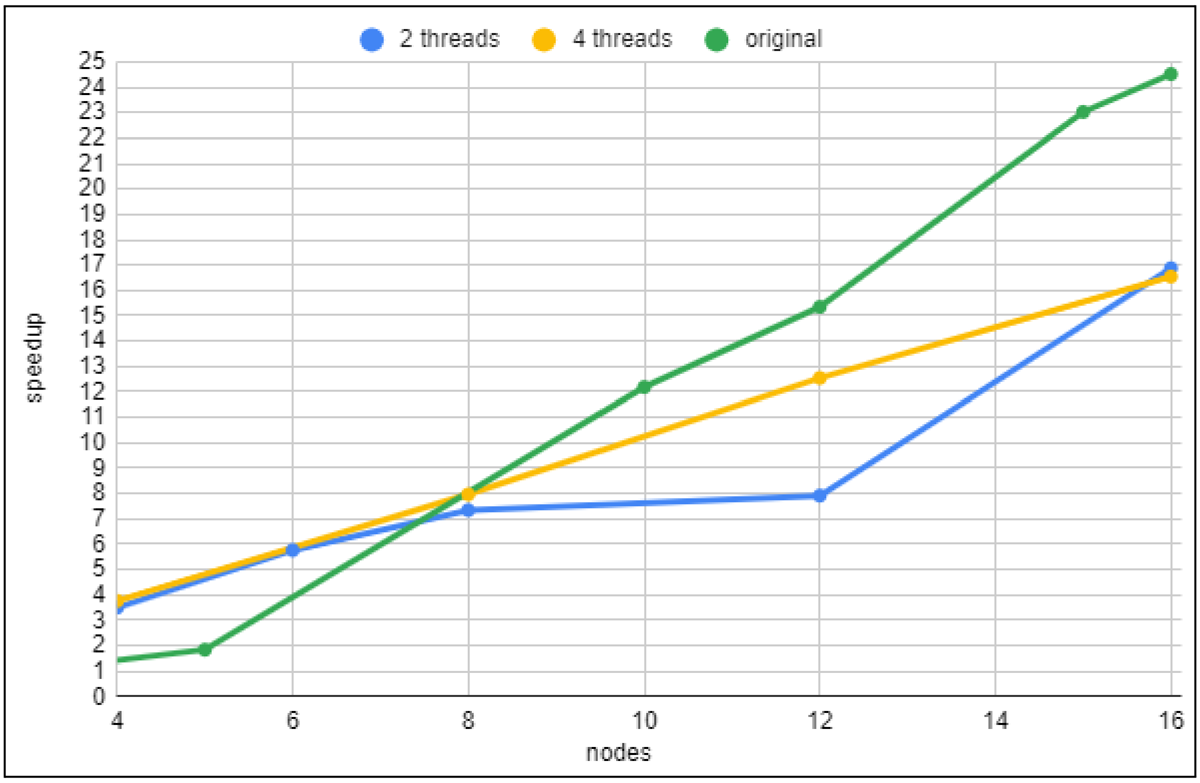


Figure 24. Measured exclusive time scalability, *nested* version

Figure 23 plots the speedup per node of the evaluated node configurations of the *loops* version in the multithreaded time excluding the SINGLE constructs. Figure 24 is the same plot but for the *nested* version.

Similarly to all the previous plots, we can see in these that the *nested* version generally underperforms the *loops* version also in the exclusive time of the function.

The configuration with 12 MPI processes and 4 OMP threads (yellow) in both versions starts with a better speedup than the original (green) and has a very consistent trend, but in the *nested* version falls behind the original version at the eight nodes point and gets outperformed by it quite visibly. Meanwhile, in the *loops* version the original doesn't catch up until what would be the equivalent to nine nodes and both have very close speedups up to twelve nodes, after which the original version has another rapid increase in the speedup, while that of the hybrid configuration remains constant all the way up to the sixteen nodes.

The configuration with 24 MPI processes and 2 OMP threads (blue) on the other hand, has clearly different behaviors from the *loops* to the *nested* version:

In the *loops* version starts with a very low speedup up to the eight nodes then quickly seems to catch up to the other configurations but has a dip in the twelve nodes point and then goes back up to catch the yellow line at fourteen nodes but doesn't seem to gain much speedup at sixteen nodes.

In the *nested* version it starts identically to the yellow configuration, but starts going a little lower at eight nodes and doesn't seem to gain any speedup up to twelve nodes, but then catches up to the yellow line at sixteen nodes and even goes a bit higher than it.

From the scalability curves we can tell that the best configuration per node in this region is using 12 MPI processes per node with 4 OMP threads for each process in the *loops* hybrid version. Two important things to highlight would be (1) that this configuration has a close speedup to the original version without yet having a domain decomposition at thread level, and with that (2) this configuration performing even slightly better than the one with more MPI processes and less OMP threads speaks favorably of the OpenMP multithreading performance.

12.6 Best performance

With the results seen in the various plots presented showing the overall speedup of the function, as well as the efficiency performance of the *parallel TL* strategy, we can already conclude that the bottleneck focused strategies from these proof of concepts won't suffice to increase the scalability of the NEMO model.

To determine whether the performance of OpenMP here is good enough for a fully hybrid implementation of the model, we need to plot the measured speedup at the multithreaded parts of the code against its theoretical ideal calculated using an efficient base time of the original NEMO.

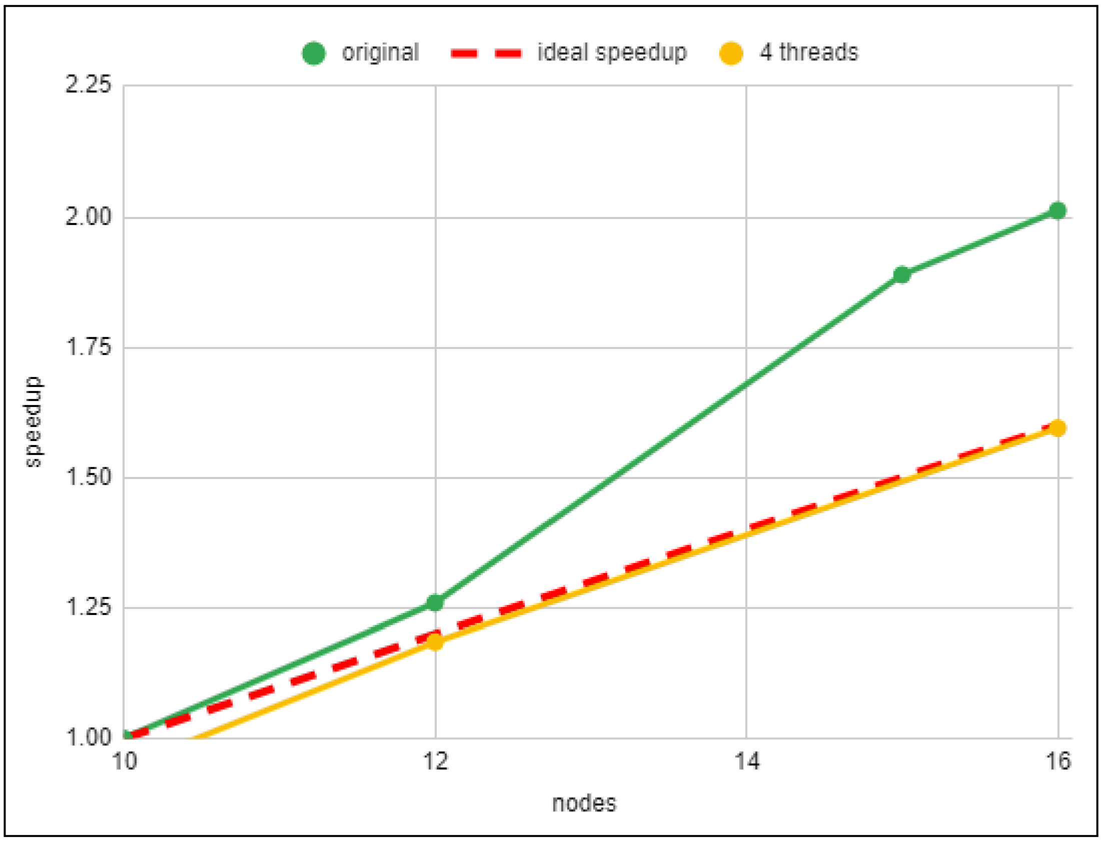


Figure 25. Best multithreaded configuration scalability against ideal

Figure 25 shows the scalability curve of the first hybrid version with a configuration of 4 omp threads, which we have seen in the previous plot has the best scalability in the exclusive time, plotted against the original function's scalability curve and what would be the theoretical ideal speedup based on the times achieved by the original version using ten nodes. Despite our version not matching the original speedup, we can see in this plot that it scales very close to the ideal. It could be considered that it scales as the ideal since the speedup achieved at the maximum number of nodes is '1.5946' and the ideal would be '1.6'. The observed superlinear scalability of the original NEMO is due to the nature of its domain decomposition, which is able to remove more useless land points from the domain the smaller the subdomains are [2].

13 Validation

To see whether the results of our new hybrid versions are valid or they have introduced errors from possible race conditions we might have missed, we need to compare their output with the one from the original using the same input data in all four executions.

For this we enabled the output from the OCE module which will write the simulation results at every timestep formatted with the Netcdf standard in files with the extension .nc , which we can then use the command *cdo sub* from the Climate Data Operators (CDO) module to subtract the values from one file to the other which will give a resulting netcdf file only containing the differences.

Due to inevitable uncertainties and inconsistencies in the observational data used for the initial state in environmental simulations, it is a common procedure to run the model until it stabilizes and the outputs can then be trusted. The period of time simulated to achieve this state is called spin-up time. This method was already used in the early days of regional climate modeling back in 1999 [48,49], and has continued to be studied in different models [50,51].

In our case we ran a 10 year spin-up with the original NEMO implementation to produce a trustable initial state for the simulations. And then, from the obtained simulation state, we then simulated 5 years with every proof of concept and the original model.

The resulting output files had no differences, meaning the outputs produced by all of the versions were identical, and thus is proof that the OpenMP parallelization strategies introduced in our proofs of concept don't alter the model simulation's results.

14 Conclusions

In this project, we evaluate the following points regarding the implementation of a multithreading parallelization level in NEMO: (1) multithreading to relieve the bottleneck found in some of the most time consuming functions; (2) OpenMP's performance for the implementation of a full hybrid NEMO model implementation.

First of all we discussed the workflow of the function identified as the bottleneck and proposed three strategies for hybrid parallel implementations as well as the experiments to be carried out to evaluate their performance in regards to our hypotheses. The experiments were carried out in a Slurm cluster running in the Marenostrum4 supercomputer.

The speedup obtained in the total function time was measured to evaluate the overall improvement in the execution time of the specific function identified as the bottleneck. This measurement allowed us to assess the effectiveness of our approach in reducing the execution time of the bottleneck function.

The exclusive time was measured to determine the speedup achieved in the section of the code that utilized multithreading. By isolating and timing this particular portion, we could evaluate the efficiency and performance gains resulting from the use of multithreading through OpenMP.

Summing up the results: it can be concluded that (1) these multithreading strategies won't improve scalability at the bottleneck identified in the most time consuming functions; (2) OpenMP performs good enough to consider pursuing a fully hybrid parallel NEMO based on the fact that we were able to achieve an ideal speedup curve on the multithreaded computing time with the first proposed strategy. It is important to highlight that this speedup was achieved without implementing an optimal domain decomposition at thread level yet.

This project was developed during 6 months and gave me the opportunity to work in an international research institution as well regarded as the BSC-CNS with experienced researchers in the field. It has been my first experience developing code to be executed in an HPC cluster and working with such a machine like the Marenostrum4 has been an invaluable learning experience with unique challenges.

15 Future Work

In this chapter we talk about how what we learned in this project shows us options to be explored in future projects.

15.1 Towards a full hybrid NEMO implementation

In this subsection we present steps needed to advance from the proof of concept presented in this thesis to a full hybrid NEMO implementation

15.1.1 Adding subroutines to the multithreaded region

In the results of the first hybrid version we have seen that achieving a nearly ideal speedup with a hybrid MPI-OpenMP implementation is possible in the parts of the code where the workload is being distributed among all the working threads of the process. The next obvious step should be to make the necessary changes in the subroutine calls making sure needed variables are available to all the threads that need them in order to increase the multithreaded region adding the least overhead possible.

15.1.2 Optimize Domain Decomposition at thread level

The impact of a pre-processing domain decomposition phase at MPI process level has been already studied and optimized in past research [52]. The currently implemented domain decomposition in NEMO explained in [2], benefits from increasing the amount of MPI processes allowing it to reduce the size of the biggest subdomain, but doesn't take into account the possible parallelization of using threads.
The proof-of-concept implemented in this project implements a second level of parallelism that benefits partially from the decomposition made in the pre-processing step. With the evidence from past research, and seeing that the speedup of our hybrid versions at the exclusive time (Figure 23 and Figure 24) is close yet still inferior to the original with pure MPI, we could expect this gap to be closed by taking advantage of the OpenMP threads in the domain decomposition phase.

15.2 Porting to GPUs

With the new architectures including GPUs that greatly increase the amounts of cores per node [10,11,12], porting the hybrid implementation to CUDA [53] for execution with GPUs to evaluate the possible speedup of the model.

16 Bibliography

[1] O. T. Prims *et al.*, "Finding, analyzing and solving MPI communication bottlenecks in Earth System models," ed. Journal of Computational Science, 2019. https://doi.org/10.1016/j.jocs.2018.04.015

[2] Irrmann, G., Masson, S., Maisonnave, É., Guibert, D., and Raffin, E.: Improving ocean modeling software NEMO 4.0 benchmarking and communication efficiency, Geosci. Model Dev., 15, 1567–1582, https://doi.org/10.5194/gmd-15-1567-2022 , (accessed 2022).

[3] Müller, M., Aoki, T. (2018). Hybrid Fortran: High Productivity GPU Porting Framework Applied to Japanese Weather Prediction Model. In: Chandrasekaran, S., Juckeland, G. (eds) Accelerator Programming Using Directives. WACCPD 2017. Lecture Notes in Computer Science(), vol 10732. Springer, Cham. https://doi.org/10.1007/978-3-319-74896-2_2

[4] Goldthau, A., & Tagliapietra, S. (2022). Energy crisis: Five questions that must be answered in 2023. Nature, 612(7941), 627–30.

[5] “Nemo community ocean model for multifarious space and time scales” https://www.nemo-ocean.eu ,(accessed 2022)

[6] Barcelona Supercomputing Center “What we do” web page , https://www.bsc.es/discover-bsc/the-centre/what-we-do , accessed 022

[7] "Marenostrum 4 begins operation." https://www.bsc.es/news/bsc-news/marenostrum-4-begins-operation , 2017

[8] BSC - CNS Earth Sciences Department https://www.bsc.es/discover-bsc/organisation/scientific-structure/earth-sciences

[9] O. Tintó Prims. Accelerating NEMO: Towards Exascale Climate Simulation. Master’s thesis, Universitat Autònoma de Barcelona, 2014.

[10] Nvidia Hopper architecture in depth, 2022 , https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/,

[11] LUMI’s full system architecture, 2021 , https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/, accessed 2023

[12] Frontier System Specs, 2022, https://www.olcf.ornl.gov/frontier/#4 , accessed 2023

[13] NEMO 4.2.0 release update features overview , https://sites.nemo-ocean.io/user-guide/changes.html#changes-since-4-0-7 accessed 2022

[14] OpenMP Standard Documentation https://www.openmp.org

[15] Fortran90 Standard documentation https://www.fortran90.org

[16] What is Scrum ? https://aws.amazon.com/what-is/scrum/?nc1=h_ls , accessed 2023

[17] DDT User Guide - BSC Documentation https://www.bsc.es/supportkc/docs-utilities/ddt/

[18] BSC Salaries https://www.glassdoor.com/Salary/Barcelona-Supercomputing-Center-Salaries-E382342.htm ,accessed 2022.

[19] Git documentation, https://git-scm.com/docs/git , accessed 2022

[20] PC average consume cost. https://chcenergia.es/blog/cuanto-consume-un-ordenador-o-pc/ , 2019

[21] Climate modeling explanation by the Geophysical Fluid Dynamics Laboratory, https://www.gfdl.noaa.gov/climate-modeling/ accessed 2023

[22] Climate models explained by the National Oceanic and Atmospheric Administration, https://www.climate.gov/maps-data/climate-data-primer/predicting-climate/climate-models, accessed 2023

[23] MIT on Predicting Climate change , https://climateprimer.mit.edu/climate-change/#predicting-climate

[24] CESM main webpage https://www.cesm.ucar.edu (accessed 2022)

[25] Michael A. Alexander, James D. Scott, Kevin D. Friedland, Katherine E. Mills, Janet A. Nye, Andrew J. Pershing, Andrew C. Thomas; Projected sea surface temperatures over the 21st century: Changes in the mean, variability and extremes for large marine ecosystem regions of Northern Oceans. Elementa: Science of the Anthropocene 1 January 2018; 6 9. doi: https://doi.org/10.1525/elementa.191

[26] Alo, C. A., and Wang, G. (2008), Potential future changes of the terrestrial ecosystem based on climate projections by eight general circulation models, J. Geophys. Res., 113, G01004, doi:10.1029/2007JG000528.

[27] WCRP Coupled Model Intercomparison Project https://www.wcrp-climate.org/wgcm-cmip , accessed 2022

[28] Climate and Ocean Variability, Predictability and Change (CLIVAR) https://www.wcrp-climate.org/clivar, accessed 2022

[29] World Climate Research Programme, 2021. Emerging climate risks and what will it take to limit global warming to 2.0°C? Statement from the 26th Conference of the Parties, Glasgow, 2021. https://www.wcrp-climate.org/WCRP-publications/2021/COP26-Statement.pdf , accessed 2022

[30] NEMO consortium bibliography database https://www.nemo-ocean.eu/bibliography/publications/

[31] NEMO ocean engine, NEMO System Team, Scientific Notes of Climate Modelling Center, 27, ISSN 1288-1619 Institut

Pierre-Simon Laplace (IPSL), https://doi.org/10.5281/zenodo.6334656

[32] Intel MPI documentation https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html#gs.ya2is3 , accessed 2022

[33] NetCDF explained by the Unidata Community https://www.unidata.ucar.edu/software/netcdf/

[34] XIOS wiki page, https://forge.ipsl.jussieu.fr/ioserver/wiki , accessed 2022

[35] BSC Tools - Paraver https://tools.bsc.es/paraver

[36] BSC Tools - Extrae https://tools.bsc.es/extrae

[37] BSC Tools documentation https://tools.bsc.es

[38] Slurm Workload Manager Overview, https://slurm.schedmd.com/overview.html , accessed 2022

[39] Open MPI https://www.open-mpi.org

[40] MPICH main webpage https://www.mpich.org

[41] Introduction to Parallel Programming with MPI and OpenMP https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf ,(accessed 2022)

[42] Low-cost MPI Multithreaded Message Matching Benchmarking https://www.osti.gov/servlets/purl/1830913 ,(accessed 2023)

[43] Rabenseifner, R., Hager, G., Jost, G., Keller, R. (2006). Hybrid MPI and OpenMP Parallel Programming. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2006. Lecture Notes in Computer Science, vol 4192. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11846802_10

[44] AMD EPYC Series 7003 Processors, https://www.amd.com/en/processors/epyc-7003-series , accessed 2023

[45] Hongyang Zhou, Gábor Tóth, Efficient OpenMP parallelization to a complex MPI parallel magnetohydrodynamics code, Journal of Parallel and Distributed Computing, Volume 139, 2020, Pages 65-74, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2020.02.004.

[46] Carribault, P., Pérache, M., & Jourdren, H. (2010). Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. IWOMP, 6132, 1-14. https://link.springer.com/content/pdf/10.1007/978-3-642-13217-9.pdf#page=10

[47] Top 500 the list, June 2023 https://www.top500.org/lists/top500/list/2023/06/

[48] CHRISTENSEN, O.B. (1999), Relaxation of soil variables in a regional climate model. Tellus A, 51: 674-685. https://doi-org.recursos.biblioteca.upc.edu/10.1034/j.1600-0870.1999.00010.x

[49] Pan, Z., E. Takle, W. Gutowski, and R. Turner, 1999: Long Simulation of Regional Climate as a Sequence of Short Segments. Mon. Wea. Rev., 127, 308–321, https://doi.org/10.1175/1520-0493(1999)127<0308:LSORCA>2.0.CO;2

[50] Zhang, S. & Harrison, Matthew & Wittenberg, Andrew & Rosati, Anthony & Anderson, J. & Balaji, V.. (2005). Initialization of an ENSO Forecast System Using a Parallelized Ensemble Filter. Monthly Weather Review - MON WEATHER REV. 133. 10.1175/MWR3024.1.

[51] Lavin-Gullon, A., Milovac, J., García-Díez, M. et al. Spin-up time and internal variability analysis for overlapping time slices in a regional climate model. Clim Dyn (2022). https://doi.org/10.1007/s00382-022-06560-2

[52] Tinto, O., M.C. Acosta., M. Castrillo, A. Cortes, A. Sanchez, K. Serradell, F.J. Doblas-Reyes. Optimizing domain decomposition in an ocean model: the case of NEMO. Procedia of Computer Sciences. http://www.sciencedirect.com/science/article/pii/S1877050917308888 , 2017

[53] Fortran CUDA , https://developer.nvidia.com/cuda-fortran , accessed 2023

Glossary

Distributed memory parallelism: A parallel computing approach where multiple computing units each have their own local memory and communicate by exchanging messages, enabling the simultaneous execution of tasks across the multiple computing units.

Process: An instance of a program running on a computer system, typically having its own memory space and resources, and capable of executing tasks independently. Note that each instance of an MPI parallel execution is a process.

Shared memory parallelism: A parallel computing approach where multiple threads or processes share a common memory space, allowing them to access and update shared data, leading to concurrent execution of tasks.

Thread: A sequence of instructions within a process that can be scheduled and executed independently, allowing for concurrent execution of tasks within a process. Note that OpenMP spawns different threads within processes.

Multithreaded region: the section of code designed to be executed concurrently by multiple OpenMP threads, with each thread executing a different portion of the code.

Hybrid parallelism: A combination of distributed memory parallelism and shared memory parallelism, utilizing both approaches to achieve efficient parallel execution on a parallel computing system.

Bottleneck: A point or component in a system that limits or restricts the overall performance or efficiency of the system, often causing delays or hindering scalability.

Speedup: A measure of the performance improvement achieved by parallelizing computation workload compared to the original execution, expressed as the ratio of the execution time in the original version to the execution time in the new parallel version.

Scalability: The ability of the application to maintain or improve speedup as the problem size or number of processing units increases.

Code

The code generated during the development of this project, and instructions on how to compile it and run it, are available at:

https://github.com/OODDer/nemo_omp_tfg