Improving the throughput of an atmospheric model using an asynchronous parallel I/O server

Master's Thesis

for the award of the degree of

Master in Innovation and Research in Informatics (MIRI)

specialization in

High Performance Computing (HPC)

Xavier Yepes-Arbós
(xavier.yepes@bsc.es)

Supervisor
Mario C. Acosta
(mario.acosta@bsc.es)

Co-supervisor
Francisco J. Doblas-Reyes
(francisco.doblas-reyes@bsc.es)

Tutor
Daniel Jiménez-González
(djimenez@ac.upc.edu)

UPC
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

BSC
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

April 2018

Abstract

The increment of the spatial resolution for computational Earth system models is nowadays one of the main concerns of the scientific community in order to solve more complex problems, and thus, achieve more accurate solutions to the reality. However, the new complexity requires more computing power than only cutting-edge supercomputers can provide. This requires to use sophisticated HPC techniques to efficiently use the computational resources. In addition, such high resolutions lead to generate an enormous amount of data to meticulously represent accurate solutions.

Current Earth system models usually have inefficient sequential I/O schemes that used to run low grid resolutions, where the generated amount of data for the simulation results was not particularly big. However, sequential I/O schemes do not scale with current models where a lot of parallel resources are used. In order to address this issue, the most adopted approach is to use scalable parallel I/O solutions that offer both computational performance and efficiency.

This master's thesis analyzes the I/O process of IFS, one of the most important atmospheric models used around Europe for several institutions, which uses an inefficient sequential output scheme. Here it is presented an easy-to-use development that integrates an asynchronous parallel I/O server called XIOS into IFS. Moreover, different optimization techniques, such as computation and communication overlapping, are applied in the integration development to minimize the I/O overhead in the resulting IFS execution.

The results show that the use of XIOS in IFS to output data is certainly good. This new parallel scheme has significantly reduced the execution time of the original

sequential scheme. Using the proper configuration, XIOS proves to be a scalable I/O server that keeps a low overhead regardless the amount of IFS processes and the output size. Furthermore, XIOS offers a series of benefits that shorten the critical path of IFS experiments by concurrently running the post-processing task along the IFS execution: data format conversion, online post-processing and CMIP-compliant output. In this scenario, the total execution time is greatly reduced.

Acknowledgments

I would like to express my appreciation to Mario for his immeasurable patience and valuable suggestions during the supervision of this master's thesis. To Glenn Carver and Gijs van den Oord for their willingness to answer an endless number of questions about IFS.

I wish to thank Paco and Dani for their great advice to make this master's thesis document much more legible. My thanks to Kim and Pierre-Antoine for their technical support, as well as Miguel and Oriol for their useful tips.

I would also like to thank BSC and Master colleagues for doing lunchtime more enjoyable, gossiping and arguing, most of the time, about surreal things.

No voldria acabar aquestes línies sense donar les gràcies a família i amics. El seu suport anímic ha estat incondicional en els moments més difícils. En especial, voldria agrair-ho als meus pares, la Maria i en Josep Maria, a la meva germana la Isabel i als meus dos nebots, la Laia i en Biel, així com també al meu avi, en Josep. Finalment, gràcies a la Mixeta per les innombrables hores d'estudi que m'ha acompanyat, malgrat no haver pogut fer-ho en la recta final. Gràcies a tots.

Contents

List of Figures

List of Tables

List of Code snippets

Chapter 1

Introduction

Over the years, computing power of supercomputers has grown exponentially [1]. Scientific applications from all disciplines have benefited of it by increasing the complexity of the computational models used. This new complexity turns into extra computational cost added using different methods, from the increase in the horizontal or vertical resolution of spatial grids to the interaction in parallel of new components simulating additional features (biochemistry, ice, etc.). When this kind of applications have available more computational resources, they can afford to solve more complex problems leading to more accurate solutions. For example, the increase of the horizontal spatial resolution, in ocean models for areas near to the coast, allow to correctly simulate some small-scale processes such as eddies [2]. However, a higher resolution implies to generate much more data because the representation of these small-scale processes requires more points. One of the main problems is to efficiently write this such amount of data along the execution of applications. The second main problem is to post-process this data afterwards. Post-processing is the phase where data is transformed using defined operations, such as data format conversion, computation of new derived variables known as diagnostics, etc.

A good example where this extra computational power has been used is in the field of Earth System Modelling (ESM). Numerical weather and climate prediction has considerably improved the accuracy of forecasts, predictions and projections due to the increase of the grid resolution [3]. Obviously, taking advantage of this extra computational power requires to properly use efficient High Performance Computing (HPC) techniques. Traditionally, the focus of improvement from a computational point of view in ESM has been the calculation and communication of algorithms [4]: the increase of the Instruction Level Parallelism (ILP); the massively parallelization of code using heterogeneous platforms such as Graphic Processing Units (GPUs) or Intel Xeon Phis; the memory access; the compiler tuning; the optimization of Message Passing Interface (MPI) patterns; and many more.

One of the most important issues studied to improve the computational efficiency are workload imbalances between processes [5][6]. Workload imbalances happen when the amount of work to be solved for each one of the processes is not evenly distributed, so this causes that some processes have more work to solve than others. Then, processes that finish earlier have to wait for the slowest ones. This problem gets worse when more processors are used, since the workload distribution is more complex. Another issue to be taken into account are the possible workload imbalances due to the type of grid used in Earth system models. Grids are used as spatial representation to discretise the Earth's surface to solve equations on evenly distributed grid-points. However, depending on the type of grid, the domain decomposition could be more or less complex. As an example of the type of grid used for this work, using regular reduced Gaussian grids has some intrinsic problems on the Earth poles [7][8], where the computation and communication among neighbors (to fulfill spatial dependencies) is much more expensive compared to other regions of the Earth, such as the Equator. This is because it is necessary to use a more complex domain decomposition.

Although ESM community has done considerable efforts to improve models from an algorithmic point of view, there is a very important aspect that has almost been forgotten during many years because it was not significant enough in the past: the Input/Output (I/O). Due to the new complexity of models, it will become really difficult to exploit computational resources to achieve more accurate results without performing an efficient I/O, because Earth system models are significantly increasing the number of variables to be output, as well as the output frequency of variables. The output process is usually performed at the end of specific time steps, during the execution of the model. This process has increased the execution time during the last years, since more and more data has to be stored from higher grid resolutions. In the end, an inefficient I/O process could lead to a serialization where all resources are waiting to complete this critical task. Moreover, since we are rapidly approaching the exascale era, the I/O part will become a truly bottleneck [9], mainly because of the produced huge amount of data. Exascale computers are future machines that will have at least one exaFLOP, or a billion billion calculations per second. On these machines, models will be potentially able to simulate ultra-high resolutions, but if the I/O process is not optimized at the same time, everything will slow down and will not be possible to achieve the future ambitious goal of having more complex models.

One of the models that could be in this situation is the Integrated Forecast System (IFS). IFS [10] is a global data assimilation and forecasting system developed by the European Centre for Medium-Range Weather Forecasts (ECMWF) and used by several institutions around Europe. IFS has two different output schemes: a sequential output scheme which gathers all data in the master process, and the Météo-France (MF) I/O server which is an efficient I/O scheme that uses dedicated resources to perform the I/O. While ECMWF uses the MF I/O server for its operational forecasts, external institutions have to use the sequential output scheme due to a license restriction. This is the case of one of the global climate models most used around Europe, the EC-Earth model, which uses a limited version of IFS as its atmospheric component.

EC-Earth [11] is a global coupled climate model, which integrates a number of component models in order to simulate the Earth system. It is used for problems encompassing from seasonal-to-decadal climate prediction to climate change projections and paleoclimate simulations. In Figure 1.1 there is a scheme showing the components of EC-Earth.

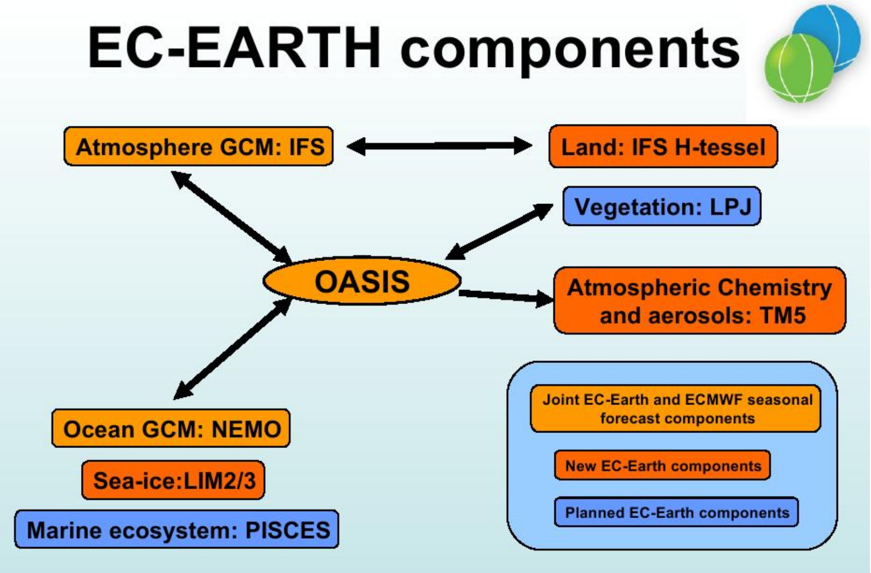


Figure 1.1: Components used in the EC-Earth model

The two main components are IFS as the atmospheric model and the Nucleus for European Modelling of the Ocean (NEMO) as the ocean model, both coupled using OASIS3-MCT.

Climate models such as EC-Earth are a very good example to prove that an efficient I/O will be needed for the future. They are run to simulate really extensive periods of time which turns into an enormous amount of data which has to be saved for prediction and projection analyses. For example, in a recent EC-Earth experiment consisting in a 100 years projection, it was generated a total amount of 244 TB of useful data. Only for IFS were used 4416 processors, consuming about 2475786 Central Processing Unit (CPU) hours.

Additionally, there are other tasks that are included typically in the workflow of an Earth system model, apart from the task used to solve the governing equations along the time. These tasks are known as pre-processing and post-processing and they are used to process the input and output data. For example, in the pre-processing task there could be the preparation of the initial conditions which are needed to run any model, whereas in the post-processing task there could be the computation of derived variables, known as diagnostics. Diagnostics are a type of variables computed from other variables, typically prognostic variables which are directly predicted by the model.

Figure 1.2 illustrates a really simple experiment which contains three tasks: pre-processing, simulation and post-processing. The time needed to complete the three tasks is known as critical path, because there are dependencies between them and must be sequentially executed.

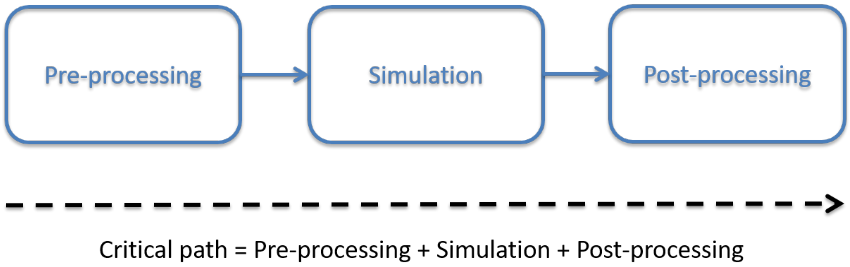

Pre-processing
Simulation
Post-processing
Critical path = Pre-processing + Simulation + Post-processing

Figure 1.2: Critical path of a basic experiment with three tasks sequentially executed

In particular, the post-processing task in EC-Earth is characterized by being quite expensive, because it needs to transform General Regularly-distributed Information in Binary form (GRIB) files output by IFS to Network Common Data Format (netCDF) files. This is necessary because IFS was originally developed for Numerical Weather Prediction (NWP), where the data format standard is GRIB. GRIB was designed to offer high performance for I/O operations, since in operational weather forecast, the time-to-solution is a critical process. However, the data format used in climate modelling as an accepted standard is netCDF. In addition, EC-Earth needs to compute expensive diagnostics using variables from both IFS and NEMO.

1.1 Motivation

As it has been explained, the sequential output of IFS is the scheme used in the IFS version of EC-Earth. This was not a problem using the resolution required for operational experiments, because the I/O did not represent a significant part in the total execution time.

However, we have recently started to perform very complex experiments in collaboration with different institutions around Europe, under the Horizon 2020 (H2020) PRocess-based climate sIMulation: AdVances in high-resolution modelling and European climate Risk Assessment (PRIMAVERA) project [12]. This project aims to simulate using higher resolution and it requires to produce a large number of variables. As a consequence, the community has experienced a considerably slowdown in the execution time, not only due to the higher resolution of the computational part, but also especially due to the I/O part, because it represents about 30% of the total execution time.

This is one of the critical issues to be solved for the community, because it will be present in new future experiments which will require similar PRIMAVERA configurations or even more complex.

For this reason, it is necessary to solve this problem by identifying which are the present and future community needs to use an efficient and functional I/O approach. The first need is to write in netCDF data format instead of GRIB, like IFS currently does, because climate modelling works with netCDF format. The second need is to produce netCDF files as requested by the CMIP6 data request. The Coupled Model Intercomparison Project (CMIP) [13] is a standard experimental protocol for studying the output of coupled Atmosphere-Ocean Global Circulation Models (AOGCMs). The third need is the ability to perform online post-processing, which means that the data to be output is processed along the simulation. Those three needs are very important to avoid the costly current post-process that we have to perform. The fourth need, from a computational point of view, is to find an efficient and scalable I/O approach able to exploit parallel resources.

Additionally, the new complexity of Earth system models where different components interact among them implies new needs where data produced by each component will not be independently processed anymore. Since we currently compute some diagnostics using variables from both IFS and NEMO, we will need a mechanism to concurrently compute them online for both components along the simulation, avoiding to do it in the post-processing task.

In order to fulfill the aforementioned community needs, different I/O tools were studied, selecting for this work the most suitable one. The XML Input/Output Server (XIOS) [14] is an asynchronous MPI parallel I/O server that we chose to be integrated with IFS, and as a consequence will be used in the future OpenIFS version that will substitute IFS in EC-Earth. OpenIFS is a free licensed and simplified version of IFS.

In addition, since NEMO is already using XIOS for outputting data, the future version of EC-Earth, which will use OpenIFS and NEMO, will be able to compute online diagnostics through XIOS with variables from both components at the same time.

Therefore, the XIOS integration will fulfill all the community needs and will increase the computational efficiency of IFS and will reduce the critical path by avoiding the post-processing task. Figure 1.3 shows how the critical path will be improved by using XIOS for IFS, OpenIFS and EC-Earth.

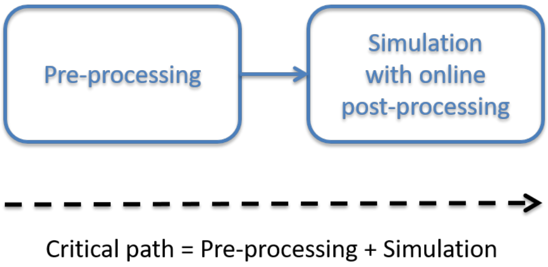

Pre-processing
Simulation with online post-processing
Critical path = Pre-processing + Simulation

Figure 1.3: Optimized critical path of a basic experiment with two tasks sequentially executed

1.2 Objectives

In this master's thesis our main objective is to improve the I/O performance of IFS to reduce the total execution time and achieve a better computational efficiency.

This is to get more throughput, which means to write more data in less time.

We also set the objective of reducing the critical path by removing the post-processing devoted to perform costly operations such as GRIB to netCDF data format conversion or diagnostics computation.

In addition, we also have the objective to increase the usability of IFS by using an easier output configuration file compared to the current approach.

In order to achieve the objectives, we identify the following tasks:

- Perform the state-of-the-art of I/O techniques used in HPC and ESM to identify the most appropriate approach to be used by IFS.
- Develop an integration between IFS and XIOS.
- Do a performance analysis of the development to detect potential bottlenecks and use proper optimization techniques to finally obtain an efficient integration.

1.3 European collaboration

We have actively worked and collaborated with two other European institutions to perform this project: the ECMWF and the Netherlands eScience Center (NLeSC)/ Koninklijk Nederlands Meteorologisch Instituut (KNMI).

ECMWF is interested in this work because of two reasons. Although IFS is used for the operational weather forecast using GRIB format, other departments are interested in the new features provided by this work. This means that the IFS-XIOS integration will be used for seasonal predictions, because they also have in the critical path the conversion from GRIB to netCDF files. The second reason is to make available XIOS with OpenIFS as an optional I/O scheme.

NLeSC joined us to develop future tasks that will follow up this master's thesis work, but they started collaborating earlier to help in some design decisions, setups, etc. They are interested in the success of the whole project because will benefit the EC-Earth consortium. Some of the involved institutions in EC-Earth are: Barcelona Supercomputing Center (BSC), KNMI, Swedish Meteorological and Hydrological Institute (SMHI), etc.

Since IFS is developed at the ECMWF, we were coordinated to use their infrastructure for development and testing: HPC machine, Git repository, issues tracking system, support, etc.

This work will contribute on a future H2020 project.

1.4 Document organization

This document is organized as follows. In Chapter 2 we research about the current available I/O schemes and discuss why we chose XIOS. We give an overview of IFS, XIOS and the computing environment in Chapter 3. We continue the document explaining the development done for the integration in Chapter 4. The next Chapter 5 is devoted to explain the performance analysis and optimization of the development. The computational performance of the integration is evaluated in Chapter 6. Finally, we conclude the work in Chapter 7.

Chapter 2

State-of-the-art

We have presented the motivation and objectives of this project, explaining why IFS needs a new I/O scheme that fulfills the community requeriments. In this chapter, we will review the state-of-the-art of the I/O solutions used on HPC, especially in the area of ESM, focusing on parallel I/O libraries and I/O servers.

In Section 2.1 we give a general overview about what is I/O and how it works the sequential I/O. Then, in Section 2.2 we explain what is parallel I/O and the different approaches that we can use. We also explain some well-established I/O libraries in HPC, including MPI-IO, Hierarchical Data Format (HDF), netCDF and Parallel I/O library (PIO). After that, in Section 2.3 we explain that there is a particular type of parallel I/O that uses dedicated resources, which are called I/O servers. We explain what is an I/O server and we present four different examples: ADaptable I/O System (ADIOS), CDI with parallel I/O (CDI-pio), Climate Fast Input/Output (CFIO) and XIOS. Finally, in Section 2.4 we compare the different presented schemes and justify why we chose to use XIOS.

2.1 I/O overview

The action of reading and writing data, commonly known as Input/Output (I/O), is a basic and essential process of almost all HPC applications to communicate with the outside world. Earth system models have traditionally performed the I/O using sequential writing [9]. It is typically done using the Portable Operating System Interface (POSIX) I/O Application Programming Interface (API). Figure 2.1 shows the basic layers of the I/O stack involved for reading and writing using POSIX I/O. Functions such as open, write or close, directly work with files.

In parallel applications, the sequential I/O implies to send all data to the master process, which performs the sequential write. In the meantime, the rest of the processes wait until this process is completed.

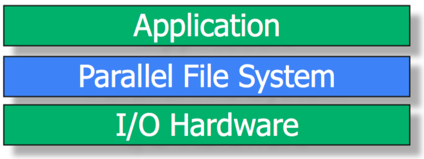


Figure 2.1: Sequential I/O stack traditionally used by HPC applications (Reproduced from [15])

Sequential I/O was fine several years ago because the amount of data was not too high and storage systems were able to deal with it. However, as we explained in the introduction Chapter 1, the increase of computing power enables the capacity to execute more accurate simulations, which leads to generate more output data. In addition, the path to exascale will accentuate this problem. This means that current sequential I/O schemes are not useful anymore, even if they use optimized techniques, because it will not scale. Therefore, applications need to use parallel I/O.

A good example of an efficient I/O optimization changing the sequential scheme to a parallel one is in the Community Atmosphere Model (CAM) [16]. They published this work in 2008, 10 years ago, which indicates that they were already aware of the inherent problems of the sequential I/O for parallel applications. This work proved that the transition from sequential to parallel I/O gives really good improvements in terms of computational performance.

However, the need of more computational power led to use much more processors in CAM, and for this reason, the parallel I/O scheme introduced in 2008, was replaced some years ago by the PIO library, which is explained more in detail in Section 2.2.

2.2 Parallel I/O

In order to increase the scalability of Earth system models, the current feasible approach is to make the I/O scalable as well, that is, use parallel I/O.

Parallel I/O is the ability to perform multiple input/output operations at the same time, such as simultaneously writing several files or concurrently writing into different regions of the same file from different processes.

Applying this concept to Earth system models [17], the main idea is to involve all the processes of a model so that they balance or re-distribute the data to be output from subdomains in a way that writing is efficiently performed and as fast as possible.

To this aim, there are two strategies commonly adopted [18]:

- Writing multiple files: data is output among several files (as many as MPI tasks). This means that each MPI task is responsible of its subdomain. The main advantage is that is very scalable, although in some file systems the creation of a lot of files can be a problem. In addition, post-processing is needed to joint files. On the left side of Figure 2.2 there is a scheme showing how each process writes its own file. As said, this can be done using the POSIX I/O API, or using MPI-IO point-to-point operations (explained later on in this section).

- Writing one file: data is output into one single file. In this case, no post-processing for joining files is needed, but the scalability is more difficult to achieve, which depends on the implementation and the number of MPI processes used, mainly due to overheads caused by conflicted I/O operations from all processes. On the right side of Figure 2.2 there is a scheme showing how all processes write into a shared file using MPI-IO collective operations.

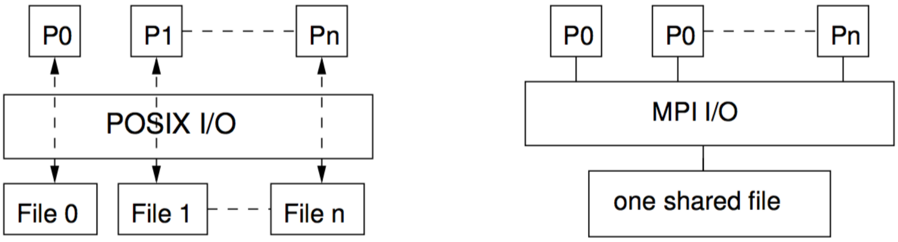


Figure 2.2: Common writing strategies: on the left side the strategy for writing multiple files and on the right side the strategy for writing one file (Reproduced from [18])

Nevertheless, the two previous strategies could not be suitable to scale models using a huge number of processes. Therefore, a feasible solution could be to use an intermediate solution. That is, writing a file for a subset of processes. Figure 2.3 shows a scheme of this intermediate strategy.

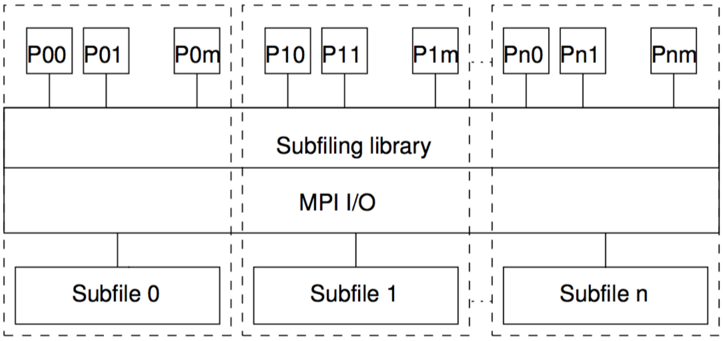


Figure 2.3: Intermediate I/O strategy: processes are grouped in subsets to share a file (Reproduced from [18])

Since POSIX I/O can not offer the possibility to implement parallel I/O, it is necessary to use a library able to offer this feature. In this case, the most popular one is MPI-IO. It is not easy to use and as a consequence has not been adopted by many applications. However, it is indirectly the most used parallel I/O library, because many other user-friendly high-level libraries are built on top of MPI-IO, such as netCDF or HDF. Since these high-level libraries are easy to use, they are commonly used for Earth system models.

This new approach to perform I/O changes the I/O stack by adding two new layers [19] as can be seen in Figure 2.4. In this new stack, applications use high-level libraries that usually offer a powerful API for efficiently organizing complex data objects with the corresponding metadata. At the same time, high-level libraries use I/O middlewares to efficiently store data into storage systems through parallel file systems.

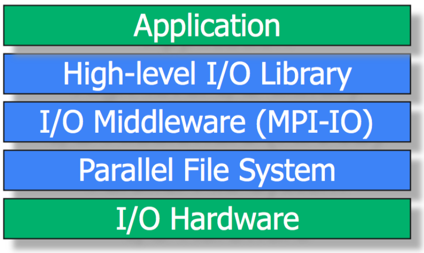


Figure 2.4: Parallel I/O stack adopted by HPC applications (Reproduced from [15])

In the following points we review some well-established I/O libraries in HPC applications [20][21], i.e., MPI-IO for the I/O middleware layer, and HDF5, netCDF and PIO for the high-level I/O library layer:

- MPI-IO: the MPI 2.0 standard [22] was extended by adding specific parallel I/O functionality. Since POSIX was designed for serial I/O, MPI-IO [15] aims to offer a high-level interface to split the reading and writing of data across several processes taking advantage of MPI messages. It is possible to use both individual or collective I/O operations. The interface syntax is based on MPI subroutines.

  Therefore, MPI-IO [23] allows to read and write a normal file from many different processes, but ensuring that the file will have the structure as if it is written using the standard I/O calls. Each MPI process has a description about how its data arrays are mapped to the proper place of the file. MPI-IO has several advantages: processes can use individual or shared file pointers, non-contiguous access of files and memory (using MPI Datatypes), explicit offsets, etc. Regarding the file data representation, there are three types: native, internal and external32.

Currently, many different I/O libraries are built upon MPI-IO, because it offers a good performance and deals with low-level implementation details. The performance can be tuned by setting a lot of different parameters because each HPC machine has its own hardware and software configuration.

- HDF5: the Hierarchical Data Format (HDF) [24] is a set of tools, libraries and file formats that are used to manage and store large amounts of data. It can also represent very complex data objects and a wide variety of metadata. In addition, it is able to store multi-dimensional arrays. HDF5 is the latest version.

- NetCDF: the Network Common Data Format (netCDF) [25][26] is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.

  The latest version is netCDF4 which can use HDF5 file format for storing data, but it is also compatible with the previous (and first) version of netCDF, which is netCDF3. Regarding the parallel use of netCDF, which by default uses serial writing, there are two mechanisms:

  - For writing in netCDF3 file format, it is necessary to use Parallel-netCDF (PnetCDF), built upon MPI-IO.

  - For writing in netCDF4 file format, it is necessary to use the parallel functionality of HDF5, built upon MPI-IO.

- PIO: the Parallel I/O library (PIO) [27][28][29] has been developed to improve the ability of component models of the Community Earth System Model (CESM) to perform I/O. For example, the atmospheric component is CAM, previously used as an example in Section 2.1. However, the PIO interface is generic enough to be used by other applications, not necessarily

Earth system models. Note that the PIO library is an independent software with regard the CDI-pio library, explained in Section 2.3.2.

PIO is mainly focused on writing data in netCDF file format, although is able to write in binary. It doesn't implement its own I/O schemes, but it uses other libraries: MPI-IO, netCDF3, netCDF4 and PnetCDF. Its optimal I/O scheme can vary significantly depending on the HPC platform.

Through a simple interface, PIO is able to use a different I/O decomposition with regard the computational decomposition. This implies that internally, data has to be re-distributed between processes to perform the output. Figure 2.5 shows an example of two different decomposition. This has the advantage of tuning the I/O decomposition for a specific platform, but without changing the computational decomposition of the model.

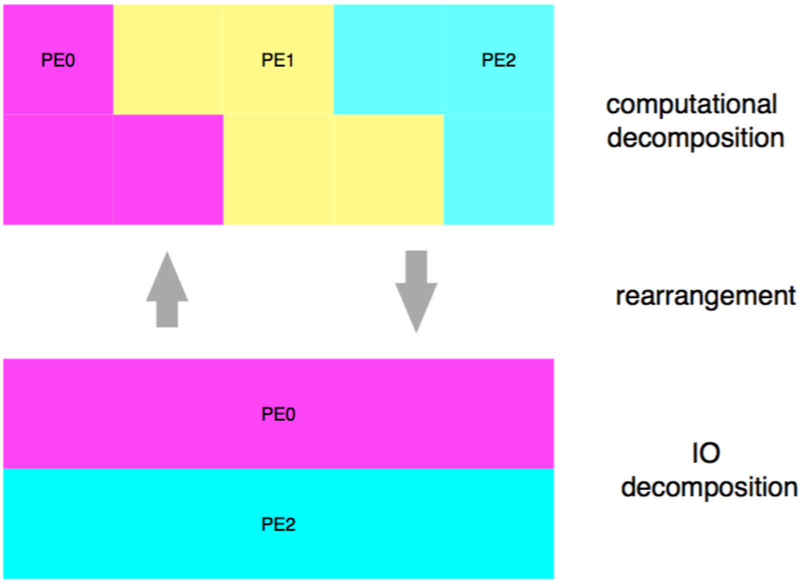


Figure 2.5: Example of a 2D array that uses 3 processes for the computational decomposition and 2 processes for the I/O decomposition (Reproduced from [27])

2.3 I/O servers

In order to keep improving the performance and the scalability of Earth System models, it is possible to go one step further and use some nodes exclusively dedicated to I/O. The model processes do not need to deal with the I/O, they only have to send the data to I/O nodes. This has the advantage that they can continue with the simulation without spending time outputting data.

This approach is known as I/O servers [28], which are the responsible of writing data into the storage system in order to theoretically hide the disks latency from the model processes and use as efficiently as possible the network bandwidth by using techniques such as aggregation.

There are different strategies to send data from model processes to I/O servers. Communication is typically done through MPI, by using synchronous or asynchronous operations. Furthermore, I/O servers can have different communication patterns. One I/O server could collect a subset of subdomains, i.e., each I/O server aggregates local data from a subset of subdomains for all variables. Or one I/O server could collect a subset of global variables, i.e., each I/O server aggregates local data from all subdomains (global domain) for a subset of variables.

I/O servers need to be configured by specifying many different parameters: how to write data, the data format, which I/O method internally use, the number of fields to be written and their dimensions, the data decomposition used in the model, etc. This can be hardcoded, done using an external configuration file which is parsed at runtime, and/or using an API, because some variables are dynamically set up on the model during runtime.

In addition, some servers can have extra functionality such as online post-processing, data conversion or data compression. If users want to use it, they simply have to set up the proper parameters in the configuration file (or through the API).

Figure 2.6 shows a scheme about I/O servers, where there are the model processes communicated with I/O servers. I/O servers, before writing data into the storage system, can perform online post-processing while model processes continue with the simulation.

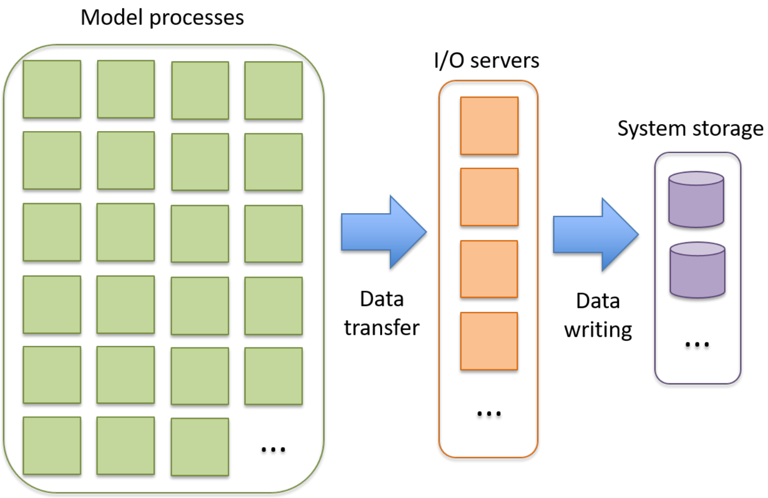


Figure 2.6: Overview of an I/O server. On the left there are the model processes which send data to the I/O servers (orange). Then, while models processes keep running the simulation, I/O servers write data into the storage system

In the following sections we will review some of the I/O servers available in the literature. All of them are designed for ESM, except ADIOS that is thought to be generic to any kind of HPC application.

2.3.1 ADIOS

The ADaptable I/O System (ADIOS) [30][31][32] is an I/O system that has proved to offer high performance in I/O operations. Its main feature is componentization, which basically abstracts the scientific code from the selection and implementation of I/O routines.

The configuration is done using an Extensible Markup Language (XML) file which describes the type of I/O that must be performed, offering several possibilities: synchronous or collective MPI-IO; parallel HDF5; parallel netCDF; asynchronous communication using Decoupled and Asynchronous Remote Transfers (DART) or DataTap methods; or no output.

ADIOS has an API to be called from the scientific code, which does not have to be modified in case of changing the I/O method. In order to change the method, it is done through the XML file, which is parsed at the beginning of the execution. Thus, re-compilation is not needed, only re-execution. Figure 2.7 shows an overview of the ADIOS API where different I/O methods can be chosen.

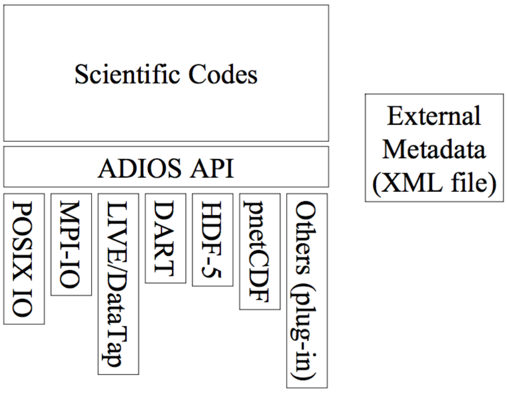


Figure 2.7: Overview of the ADIOS API architecture (Reproduced from [32])

Furthermore, it offers a feature called Data Transformations which are devoted to change the format and/or encoding of data to improve the performance of read/write operations and reduce the storage space. Figure 2.8 shows how the transform plugins are integrated in the ADIOS framework.

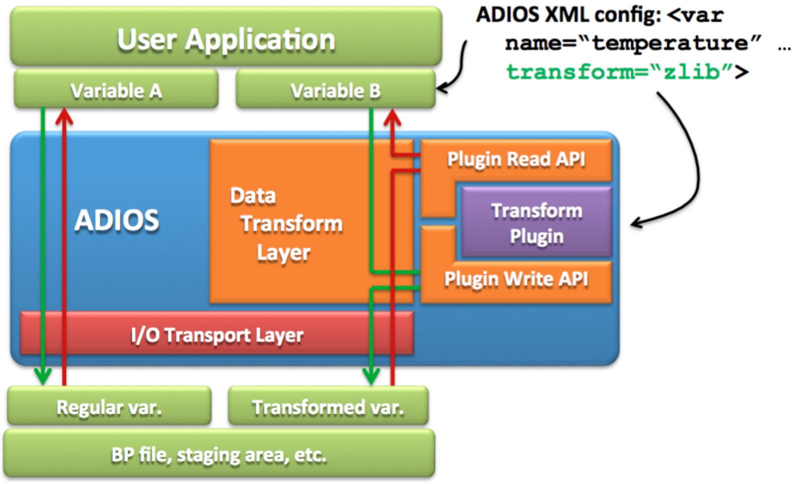


Figure 2.8: Overview of the ADIOS transform framework (Reproduced from [31])

2.3.2 CDI-pio

The Climate Data Interface (CDI) [28][33][34] is an I/O library which provides a machine and format independent interface for reading and writing data stored in scientific formats typically adopted in NWP and climate modelling. It is being jointly developed by the Max-Planck-Institute for Meteorology (MPG)) and the German Climate Computing Centre (DKRZ).

CDI used to support multi-thread, single-process scheme, but this is not scalable to current systems, where it is needed massive parallelism using distributed memory. CDI with parallel I/O (CDI-pio) addresses this issue implementing parallel writing over the serial CDI. It uses dedicated I/O processes asynchronously communicated to write data into the storage system. Furthermore, CDI-pio has support for both GRIB and netCDF formats. It offers an efficient data compression to considerably reduce the output data volume.

Internally, CDI-pio uses MPI-IO, PnetCDF/HDF5, POSIX I/O or C stdio depending on the configuration. In Figure 2.9 there is a comparison of the architectures between serial CDI and CDI-pio.

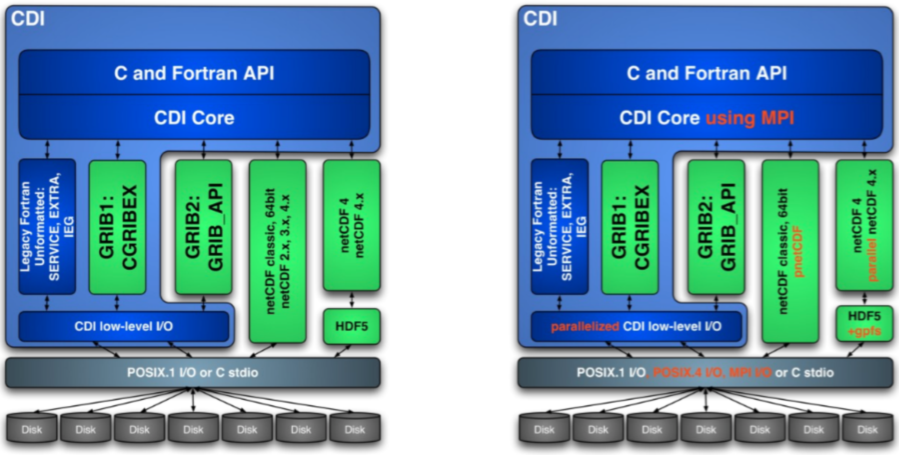

CDI
C and Fortran API
CDI Core
Legacy Fortran Unformatted: SERVICE, EXTRA, IEG
GRIB1: CGRIBEX
GRIB2: GRIB_API
netCDF classic, 64bit netCDF 2.x, 3.x, 4.x
netCDF 4 netCDF 4.x
CDI low-level I/O
HDF5
POSIX.1 I/O or C stdio
Disk
CDI
C and Fortran API
CDI Core using MPI
Legacy Fortran Unformatted: SERVICE, EXTRA, IEG
GRIB1: CGRIBEX
GRIB2: GRIB_API
netCDF classic, 64bit pnetCDF
netCDF 4 parallel netCDF 4.x
parallelized CDI low-level I/O
HDF5 +gpfs
POSIX.1 I/O POSIX.4 I/O, MPI I/O or C stdio
Disk

Figure 2.9: Comparison between serial CDI (on the left) and CDI-pio (on the right). Note the highlighted words in orange to represent the parallel extensions (Reproduced from [34])

One interesting point in CDI-pio is that I/O servers gather full fields, which means that an I/O server is in charge of gathering a subset of full-domain fields.

2.3.3 CFIO

The Climate Fast Input/Output (CFIO) library [35] provides a simple method to overlap the I/O phase with the computational phase automatically to reduce the execution time of high-resolution climate models. CFIO provides a similar interface to PnetCDF to minimize the source code modifications.

CFIO overlaps I/O with computing by implementing a client-server mechanism to deal with I/O forwarding and handling of I/O requests. It is possible to choose the number of servers in order to balance the time needed by computing processes and the time needed by the I/O processes to write data. Internally, CFIO uses PnetCDF to write data to disk through the parallel file system. In addition, it is possible to choose the type of communication between client and server processes: synchronous or asynchronous. They report that in runs using a lot of processors, the execution time is smaller using synchronous communications. In Figure 2.10

there is an overview of the architecture.

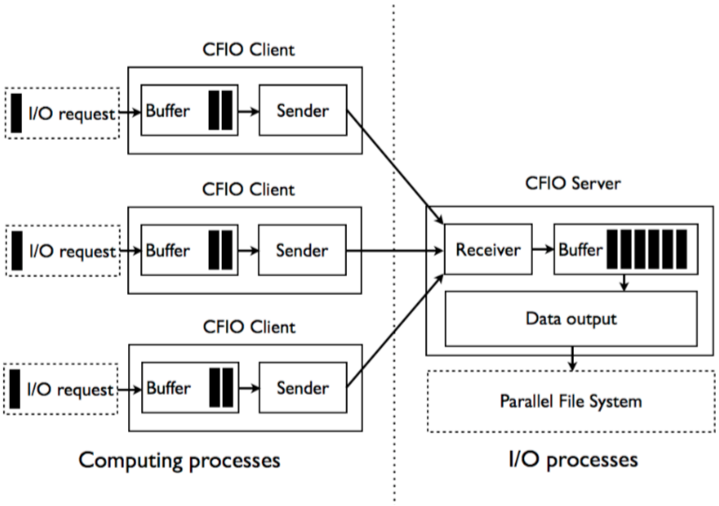


Figure 2.10: Overview of the CFIO architecture (Reproduced from [35])

2.3.4 XIOS

The XML Input/Output Server (XIOS) [14][28][34][36] is an asynchronous MPI parallel I/O server that is used by Earth system models to avoid contention in the I/O. It is developed at the Institute Pierre Simon Laplace (IPSL). Since XIOS is the I/O server used in this project, it is explained in detail in Section 3.2.

Nevertheless, it is interesting to give an overview of XIOS Interface for Arpege-climat and Surfex (XIAS) [37], which is a software specifically developed for Action de Recherche Petite Echelle Grande Echelle (Arpege) (atmospheric model) and Surface Externalisée (SURFEX) (surface model) so that both models can write data using XIOS. Therefore, XIAS is an interface to handle all the needed calls to use XIOS by implementing a thin software layer above SURFEX-provided routines. Then, since XIAS is implemented in SURFEX routines, Arpege uses XIAS through SURFEX using an inheritance mechanism.

The reason to research about XIAS is that Arpege and IFS are in essence the same atmospheric model, with the difference that Arpege is modified by MF to fulfill their specific needs. We contacted with the XIAS developers asking more information, but they told us that XIAS is not suitable for IFS, because its design was strongly influenced by SURFEX, so the interface should be re-designed to be used by IFS. For this reason we discarded to re-use XIAS.

2.4 Discussion: comparison and choice

We have reviewed several options in the literature that could potentially be integrated with IFS to solve the I/O bottleneck. Although we have not explained too much about XIOS in Section 2.3.4 (we said that is explained in detail in Section 3.2), we can make a comparison between the different schemes because we already know their main features.

In the motivation Section 1.1 of this project, we have clearly explained the specific needs that we have, so all of them should be fulfilled by one of the I/O schemes that we have presented.

If we focus on the execution time of the simulation, as well as only outputting data, we think that it is not clear which could be the fastest I/O server, because the respective authors report good performance, and I/O is something very machine-dependent and even context-dependent. For example, you could be running your scalability tests in a moment where there are other users that are executing I/O-intense applications. This would speed down your tests.

Nevertheless, there are other aspects that influence our choice. In both IFS standalone and IFS within EC-Earth the post-processing phase is needed, so if we can move this work to the I/O servers we can save a lot of resources and time. In the current critical path of an experiment, after the simulation we have to write temporary files that will be read in the post-processing phase to be then written again once they are processed. With online post-processing, after the simulation we would be able to directly write the definitive files. I/O servers such as CDI-pio and CFIO do not offer online post-processing. In ADIOS, there is the possibility to add a plugin to transform data, but in case of being possible, it would be needed a lot of extra work to add those post-processing functionality that in XIOS is already available.

Furthermore, EC-Earth contributes with climate simulations to the CMIP project, where data organization must follow an strict standard that XIOS is aware. Despite the fact that the other I/O servers can write in netCDF, they do not take into account the CMIP standard.

Last but not least, as we exposed in the introduction, in EC-Earth we have to calculate diagnostics that are derived using fields from both IFS and NEMO. Since NEMO is already using XIOS, if IFS was using XIOS as well, we would be able to calculate these diagnostics online instead of doing it at the post-processing phase. ADIOS may not support online diagnostics using variables from more than one component. However, if it was possible, we would have much more work, because it would be needed to integrate ADIOS with IFS and NEMO.

Therefore, according to what we need and what I/O servers offer, it is quite irrefutable that we have to choose XIOS.

Chapter 3

Context

In this chapter we will explain the two main components used in this project. In Section 3.1 we give an overview of the scientific part of IFS and a more detailed explanation about the computational part, including the new octahedral reduced Gaussian grid, the parallelization and domain decomposition, the data structures, the two available output schemes and two environment tools to use IFS. In Section 3.2 we explain the main features of XIOS and how it is used. We give an overview of all the XIOS elements: axis, domain, grid, field, file, filters and performance variables. Finally, in Section 3.3 we explain the main characteristics of the Cray XC40 supercomputer that we use to develop this project.

3.1 IFS description

We will mainly focus on the technical and computational part of IFS, and only briefly describing the scientific part.

3.1.1 Brief overview

One of the most advanced NWP models is the Integrated Forecast System (IFS). IFS is an operational global meteorological forecasting model and data assimilation system developed and maintained by ECMWF.

IFS [38][39][40] is a spectral model that discretises the Euler equations of motion, resolving flow features to approximately 4-6 grid-cells at the nominal resolution. The subgrid-scale features and unresolved processes are described by atmospheric physics parametrizations. There are many different unresolved physical processes in the atmosphere, such as radiation, clouds and subgrid turbulent motions. In Figure 3.1 there is an overview of the different physical processes.

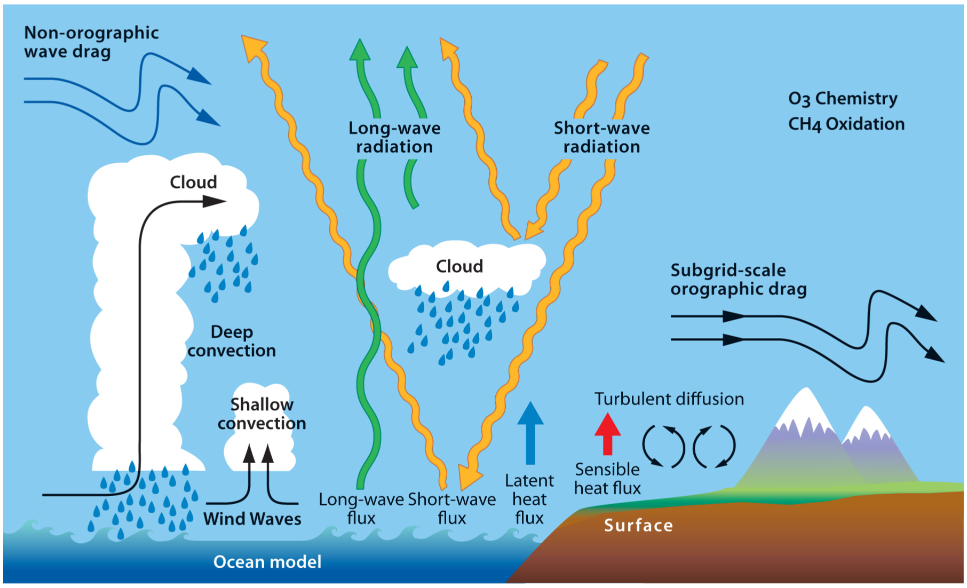


Figure 3.1: Overview of the unresolved physical processes in the atmosphere (Reproduced from [39])

The dynamical core of IFS is hydrostatic, two-time-level, semi-implicit, semi-Lagrangian and applies spectral transforms between grid-point space (where the physical parametrizations and advection are calculated) and spectral space. In the vertical the model is discretised using a finite-element scheme. A reduced Gaussian grid is used in the horizontal.

In order to simulate more accurate forecasts, it is possible to use other components of the Earth system. For example, IFS can be run coupled with the community ocean model NEMO or with the WAve Model (WAM).

3.1.2 Octahedral reduced Gaussian grid

Although IFS is a spectral model, we have mentioned that part of the processes are resolved in grid-point space. Grids are a fundamental part of Earth system models, not only due to scientific reasons, but also due to computational performance.

Each one of the grid-points have an associated latitude and longitude to place them on Earth. Latitude is the geographic coordinate that specifies the north–south position of a point on the Earth's surface, while longitude is the geographic coordinate that specifies the east-west position of a point on the Earth's surface. Figure 3.2 shows the concept of latitude and longitude, which will be needed to set up XIOS.

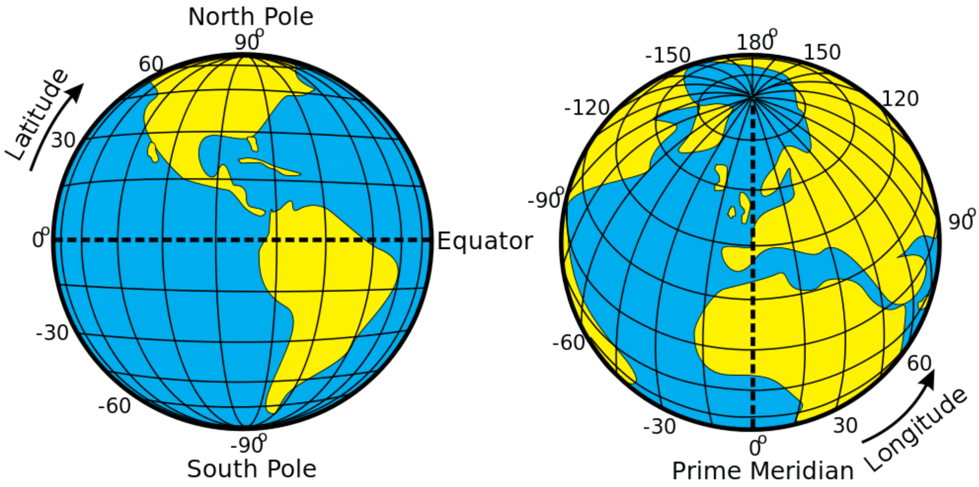


Figure 3.2: Concept of latitude and longitude

One field of study is the shape of the grid. There are different proposals in the literature to find a good distribution of grid-point over the Earth. The three basic grids used in NWP are [41]:

- Rectangular or regular Gaussian
- Triangular
- Hexagonal

IFS used to use a regular Gaussian grid, but this type of grids suffer from ”the polar problem” where density of grid-points in the poles is huge, since all latitudes have the same number of longitude points. To solve this, they later introduced the reduced Gaussian grid, where the number of longitude points for each latitude decreases when closer to the poles. Figure 3.3 shows the difference between a regular Gaussian grid (left) and a reduced Gaussian grid (right).

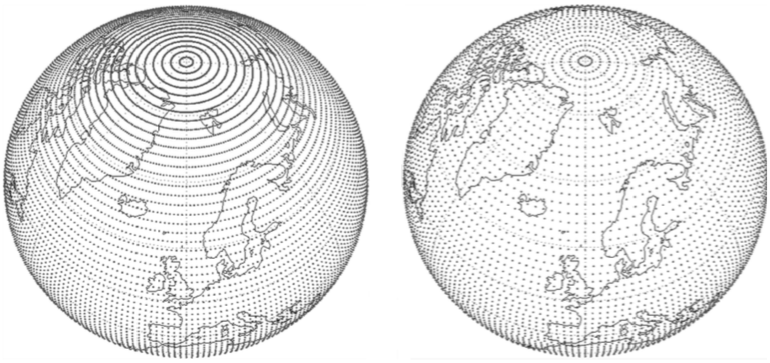


Figure 3.3: Comparison between a regular Gaussian grid (left) and a reduced Gaussian grid (right) (Adapted from [42])

In order to keep improving the grid, IFS is currently using a new grid called octahedral reduced Gaussian grid [42][43]. It brings significant benefits in terms of computational efficiency and effective resolution.

The new method to generate the octahedral reduced Gaussian grid, optimizes the total number of points around the globe and introduces a regular reduction of the number of points per latitude circle towards the poles. The process to generate this grid is illustrated in Figure 3.4. The idea is to divide each hemisphere of the globe into 4 quarters, where each quarter corresponds to one face of an octahedron. Then, start with 20 points, five per quarter at the Gaussian latitude closest to the pole. After that, add one point per quarter for each new Gaussian latitude towards the equator (this implies four additional points per each Gaussian latitude). Due to Earth’s curvature, the distance between grid-points of a latitude (dx), varies with regard to other latitudes’ dx distance.

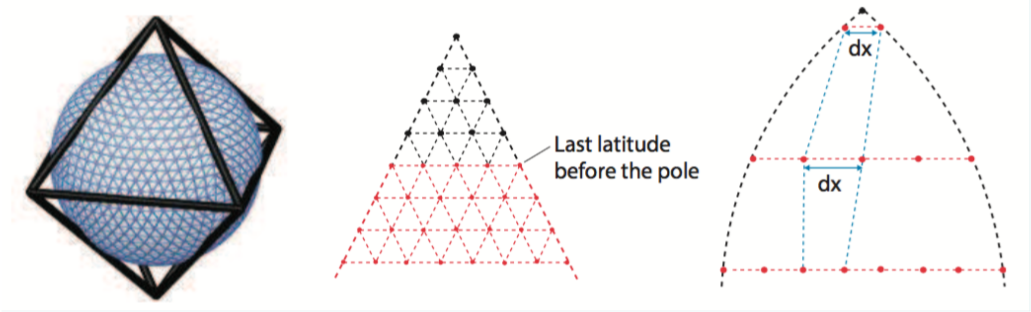


Figure 3.4: Process to generate an octahedral reduced Gaussian grid (Reproduced from [43])

Figure 3.5 shows a resolution comparison between the reduced Gaussian grid (left) and the octahedral reduced Gaussian grid (right). The octahedral grid has a locally more uniform dual-mesh resolution than the reduced grid.

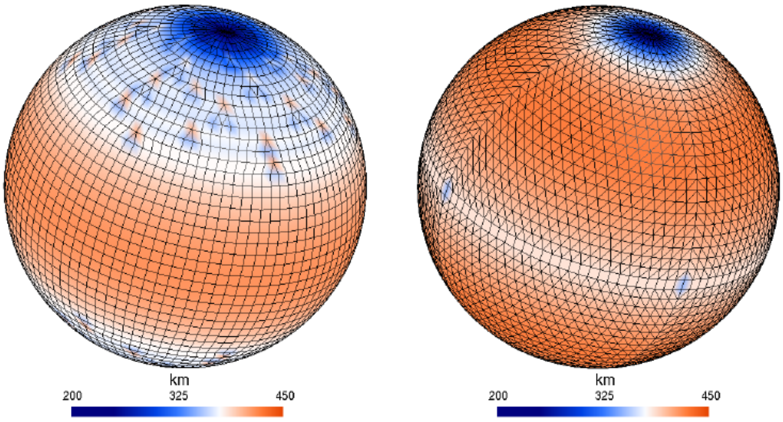


Figure 3.5: Dual resolution comparison. The octahedral grid (right) has a locally more uniform dual-mesh resolution than the reduced grid (left) (Adapted from [43])

3.1.3 Parallelization and domain decomposition

A meteorological model such as IFS [44] that has 3D fields, may have a grid-point structure such as the one in the Code snippet 3.1. There are 3 dimensions for all fields.

```
REAL Model_Data(1:Horiz_i, 1:Horiz_j, 1:Levels_k, 1:Fields)
```

Code snippet 3.1: Typical data structure for 3D fields

Since IFS is an MPI+Open Multi-Processing (OpenMP) hybrid model and used to run in vector processors, ECMWF designed a more sophisticated data structure to be efficiently parallelized. Code snippet 3.2 shows the current data structure used in IFS. The idea of the current parallelization is that fields are processed using a blocking technique, where the first *NPROMA* dimension can be adjusted by the user at run-time to fit the memory cache. Depending on the size of *NPROMA*, there will be more or less *NGPBLKS* blocks. The 2D i-j horizontal dimension is transformed using these two new dimensions.

```
REAL Model_Data(1:NPROMA, 1:NFLEVG, 1:NFIELDS, 1:NGPBLKS)
```

Code snippet 3.2: Data structure used in IFS

The other two dimensions remain the same, which are *NFLEVG* (or vertical dimension) and *NFIELDS* (the number of fields). However, note that these two dimensions now are not in the outter dimensions of the array. This implies that for a given *NGPBLKS* block, all fields are processed. Code snippet 3.3 shows the OpenMP parallelization applied in IFS for its data arrays. In addition, Figure 3.6 shows the NPROMA blocking used in IFS.

```
!$OMP DO SCHEDULE(STATIC)
DO iblock = 1, NGPBLKS
  DO ifld = 1, NFIELDS
    DO ilvl = 1, NFLEVG
      DO i = 1, NPROMA
        Model_Data(i, ilvl, ifld, iblock)
      END DO
    END DO
  END DO
END DO
!$OMP END DO
```

Code snippet 3.3: OpenMP parallelization used in IFS

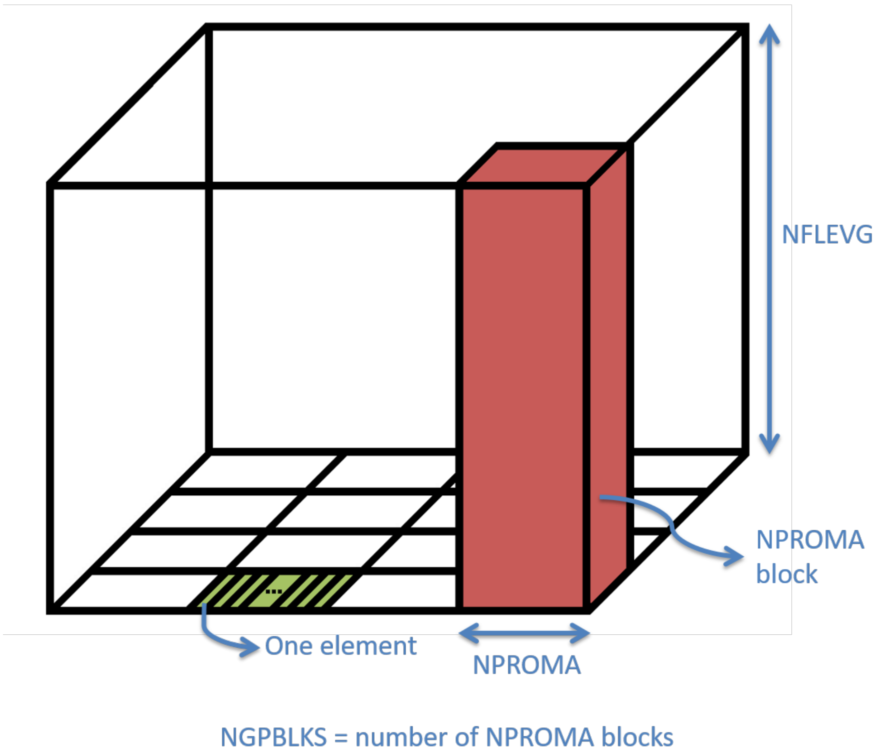


Figure 3.6: NPROMA blocking strategy used for IFS data arrays. For each NPROMA block, IFS iterates over all elements (*NPROMA*) of each vertical level (*NFLEVG*) and for each field (*NFIELDS*)

The details of the IFS intra-node shared-memory parallelization are very important to understand how the integration with XIOS will be done.

On the other hand, IFS also uses inter-node distributed-memory parallelization through MPI. First of all, it is important to briefly overview the four major algorithmic steps of IFS to better understand the strategy followed in the parallelization: grid-point computations, spectral computations, Fourier transform and Legendre transform. Figure 3.7 shows an overview of these four algorithmic steps in a single time step of IFS. The blocks in the centre of the figure represent the data decomposition used at any step within the time step. The idea of using this approach is that only in the transpositions there is data movement between MPI processes, thus in the computation steps all dependencies are satisfied, so no further communications are needed (there is an exception in the grid-point calculations where a few communications are performed for the semi-Lagrangian phase).

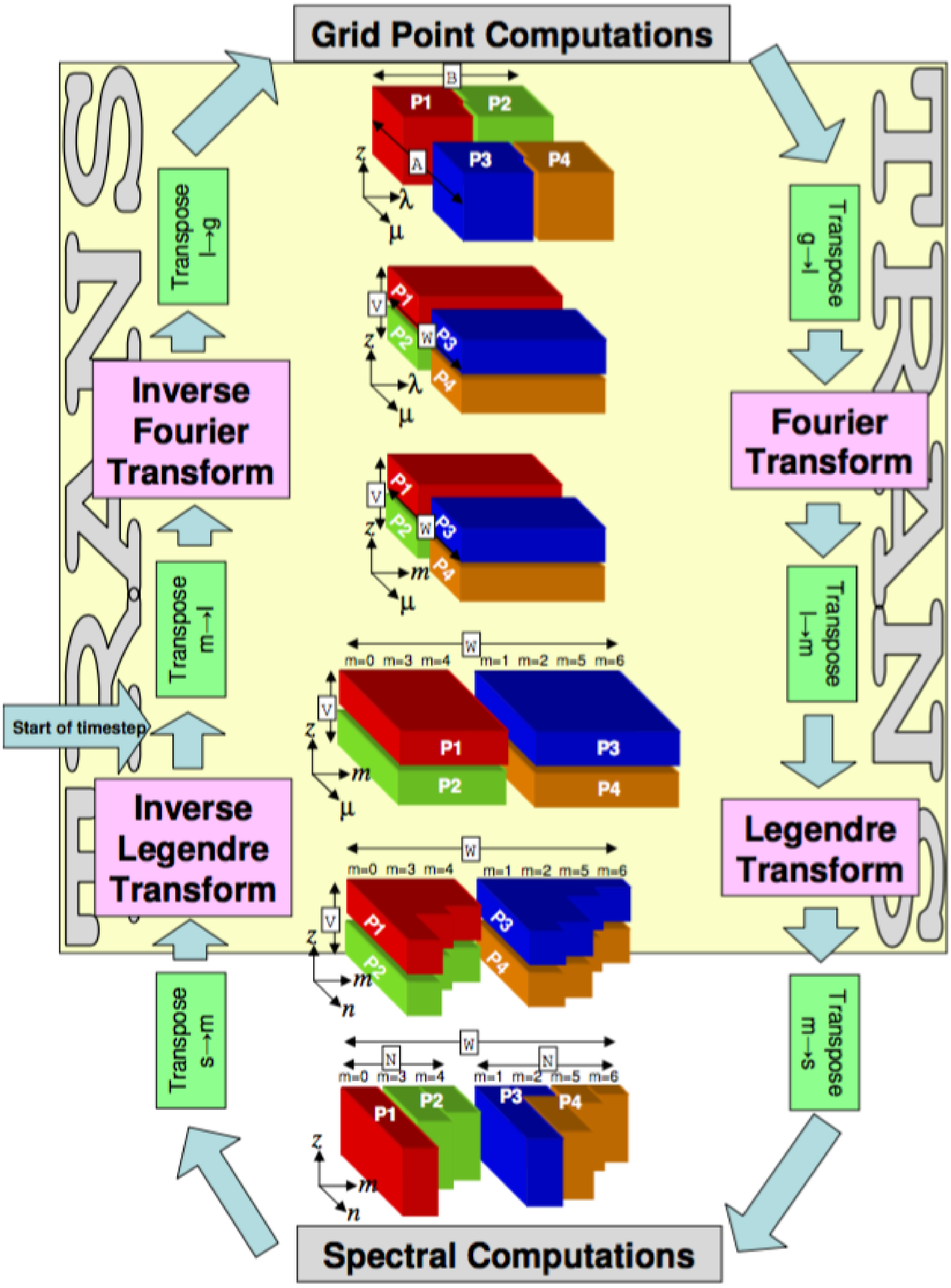


Figure 3.7: Major algorithmic steps in an IFS time step. The blocks in the centre are the different data decomposition used at each step (Reproduced from [44])

We will only focus on the grid-point computations step because XIOS requires to output data in grid-point state.

The grid-point dynamics and physics computation only has vertical dependencies, so all grid columns can be considered independent of each other, allowing an arbitrary distribution of columns between MPI processes.

The domain decomposition has been considerably improved since the beginning in order to have the maximum workload balance between MPI processes. There are several aspects that determine a good data decomposition. For example, as we mentioned, in reduced Gaussian grids we have less grid-point longitudes in latitudes close to poles, so it can become a source of imbalance. Or for example, the subdomains' shape should be as squarer as possible to minimize the communications' size of the semi-Lagrangian phase.

In order to achieve a good workload balance, IFS can use two decomposition strategies, the 2D scheme, the original strategy, and the EQ_REGIONS scheme, the current default strategy. In Table 3.1 there are the most basic variables to describe the type of decomposition.

Variable	**Description**
NPROC	Total number of processors to be used
LEQ_REGIONS	Logical controlling use of EQ_REGIONS partitioning
NPRGPNS	# Proc. in the North-South direction (LEQ_REGIONS=F)
NPRGPEW	# Proc. in the East-West direction (LEQ_REGIONS=F)
LSPLIT	Allows the splitting of latitude rows

Table 3.1: Variables to control the grid-point decomposition (Adapted from [44])

Through an example we will explain how is performed the 2D scheme. Later on, we will show the benefit of using the EQ_REGIONS scheme with a graphical example.

Our example has 6 processors and we are using a reduced Gaussian grid with 19 latitudes. We could set up the decomposition variables as it is shown in Code snippet 3.4 to have a good distribution. Figure 3.8 illustrates how the calculation of the decomposition is carried out in two steps.

```
NPROC = 6
LEQ_REGIONS = .FALSE.
NPRGPNS = 3
NPRGPEW = 2
LSPLIT = .TRUE.
```

Code snippet 3.4: Basic decomposition variables

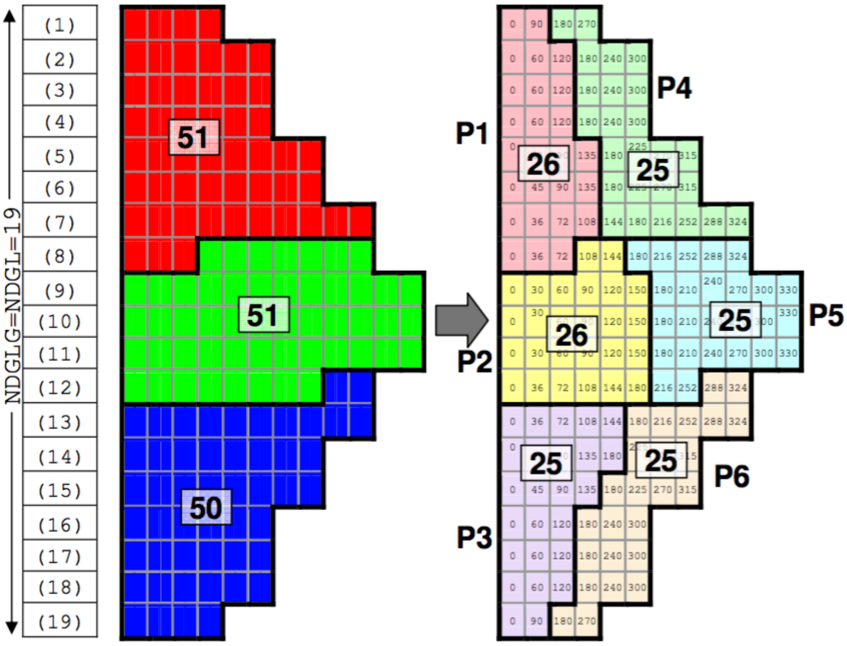


Figure 3.8: Example of a grid-point decomposition using 6 processors. On the left, the first step with the North-South partitioning and on the right, the second step with the East-West partitioning (Reproduced from [44])

In the first step, the total number of grid-points is split as equally as possible in the North-South direction (the ”A” set), i.e., between the number of NPRGPNS sets. Note that there are latitudes shared between ”A” sets due to LSPLIT=.TRUE.. This introduces a difficulty in the addressing of some arrays. This is taken into account with some additional variables used to address the subdomains. There is a more complete list in Table A.1 of Appendix A.

In the second step, the number of grid-points of each one of the ”A” sets are split as equal as possible in the East-West direction (the ”B” set), i.e., between the number of NPRGPEW sets.

According to Figure 3.8, in our example there will be two processors with 26 grid-points and 4 processors with 25 grid-points. As we said, in Appendix A there is a more complete list of the variables used to address the global domain and local subdomains. In addition, Figures A.1 and A.2 of Appendix A graphically show the usage of the decomposition variables based on our example.

Although the example is really small, it is even visible in Figure 3.8 that the shape of the subdomains it not square. Figure 3.9 shows an example of the 2D scheme partitioning using 512 MPI processes. It is quite obvious that using the same amount of ”B” sets for all ”A” sets is not a good startegy.

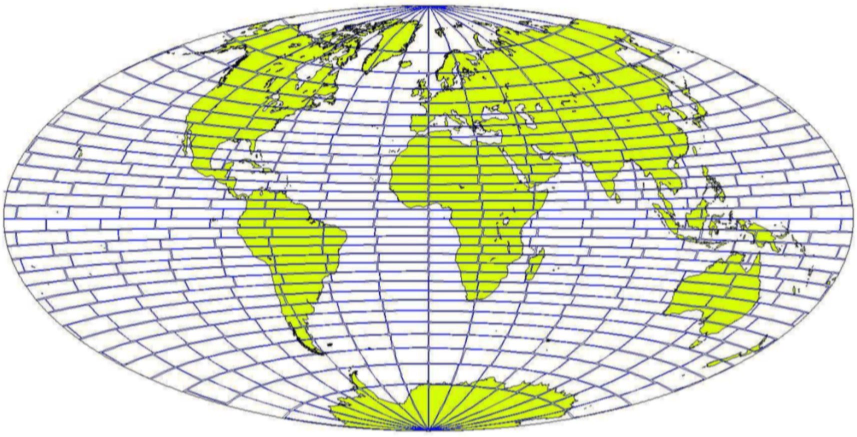


Figure 3.9: 2D scheme partitioning using 512 MPI processes (Reproduced from [44])

For this reason, ECMWF introduced the EQ_REGIONS scheme, where ”A” sets close to poles have less ”B” sets (Figure 3.10). This results in squarer partitions of equal area and small diameter.

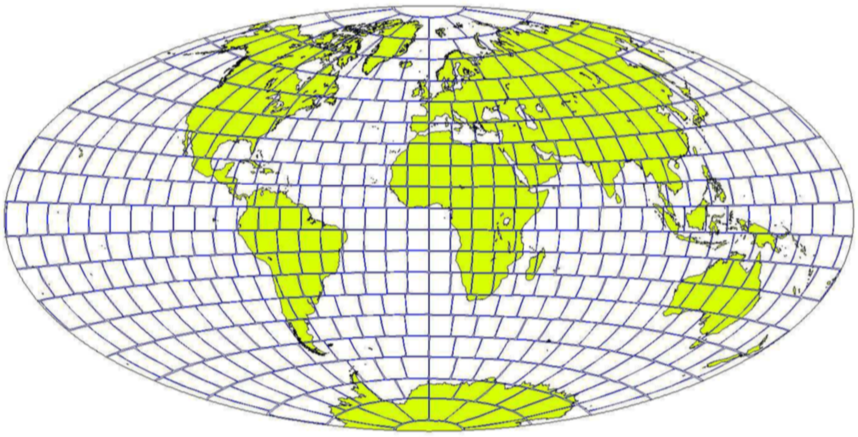


Figure 3.10: EQ_REGIONS scheme partitioning using 512 MPI processes (Reproduced from [44])

Finally, note that concepts related to grids and domain decomposition is often called the geometry of the model. From now on, we will indistinctly use all terms.

3.1.4 Data structures

In IFS there are some major data structures [44] to store all data for spectral and grid-point fields. For spectral data, IFS uses the YOMSP module, in which the arrays SPA3 and SPA2 hold the 3D and 2D state variable spectral fields respectively.

Nevertheless, as we said we will focus on grid-point fields. There are two core data-structures: GMV and GFL.

The GMV structure contains prognostic variables involved in the semi-implicit: wind components, temperature, surface pressure, vorticity, etc. All GMV fields have a spectral representation.

The is a quite fixed structure which is supposed to have almost no modification. Data arrays are accessed using pointers, as many as GMV fields. In addition, there can be 3D and 2D fields: the 3D ones are stored in the GMV array and the 2D ones in the GMVS array. GMV fields do not have attributes.

On the other hand, the GFL structure contains the rest of variables: specific humidity, snow, rain, ozone, etc. Unlike GMV, GFL structure is much more flexible and can be easily extended with additional fields. It only contains 3D grid-points fields that may have a spectral representation. Data is stored in array GFL.

One of the main characteristics are attributes. They are used to govern the behaviour of the individual fields of the GFL structure. The idea is to loop over all fields in GFL and perform the action defined by the setting of the appropriate attribute. However, it is still possible to treat fields separately through individual pointers.

For both GMV and GFL structures there are more arrays than the mentioned ones, however, we will not explain them since they are not necessary for this project.

3.1.5 IFS output schemes

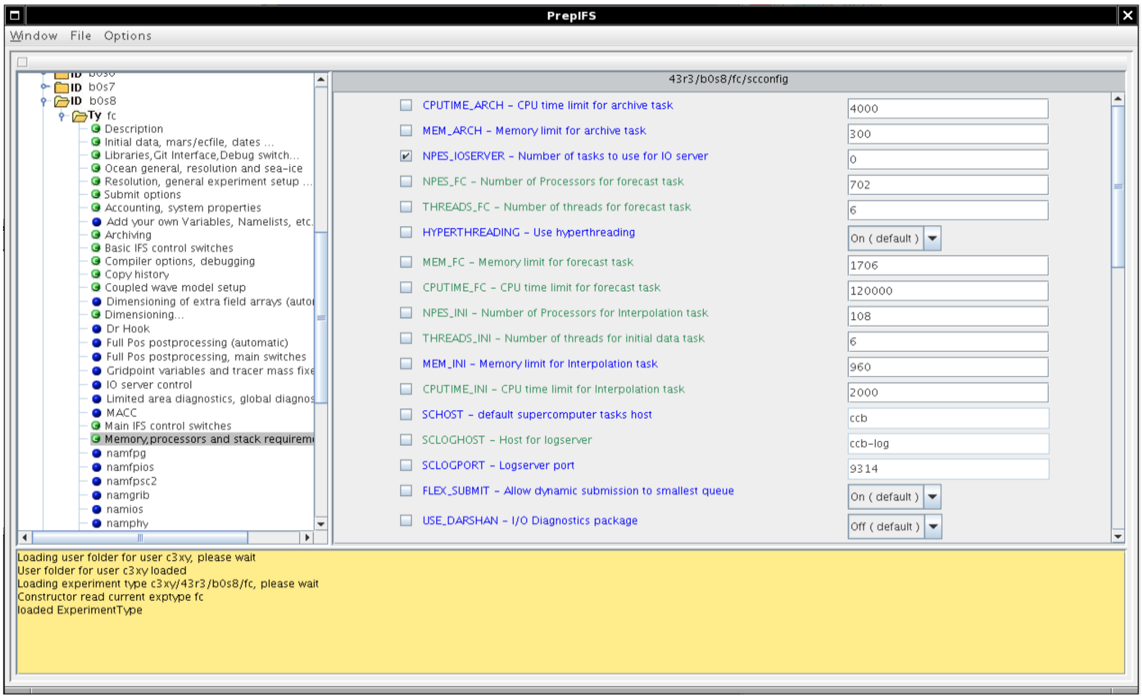
IFS has two different output schemes, one better than the other, since one was replaced by the other to improve the throughput:

- Sequential output: this is the slowest one, since the output is sequentially performed by the master process. It performs a gather of all subdomains from the rest of processes, to build the global domain and write it into the storage system. This type of output is not scalable, so it has an important negative impact in the performance of the model. This output scheme is the only available in the IFS version used in EC-Earth, so in huge simulations we are suffering a considerable slow down.
- MF I/O server: the Météo-France I/O server was introduced in IFS to avoid the low performance of the sequential output scheme. It uses the concept of dedicated processes as servers to perform the I/O, such as XIOS. Although it offers really good throughput, there are some issues: it does not perform post-processing (done by FullPos), it writes data using GRIB format and it is not available in OpenIFS (future version for EC-Earth).

3.1.6 Environment tools

In order to run IFS simulations it is necessary to create and manage an experiment. There are two basic tools that we have to use: prepIFS and XCdp.

PrepIFS [45] is a meteorological workflow manager to prepare research experiments using IFS at ECMWF. It is possible to set up a lot of different parameters, both scientific and computational. For our purposes, we used as base an experiment with default values, only changing basic parameters such as length of the forecast, number of processes, type of grid, etc. In Figure 3.11 there is a snapshot of prepIFS.


PrepIFS
Window File Options
ID b0s7
ID b0s8
Ty fc
Description
Initial data, mars/ecfile, dates ...
Libraries,Git Interface,Debug switch...
Ocean general, resolution and sea-ice
Resolution, general experiment setup ...
Submit options
Accounting, system properties
Add your own Variables, Namelists, etc.
Archiving
Basic IFS control switches
Compiler options, debugging
Copy history
Coupled wave model setup
Dimensioning of extra field arrays (autor
Dimensioning...
Dr Hook
Full Pos postprocessing (automatic)
Full Pos postprocessing, main switches
Gridpoint variables and tracer mass fixe
IO server control
Limited area diagnostics, global diagnos
MACC
Main IFS control switches
Memory,processors and stack requireme
namfpg
namfpios
namfpsc2
namgrib
namios
namphy
43r3/b0s8/fc/scconfig

☐ CPUTIME_ARCH – CPU time limit for archive task: 4000
☐ MEM_ARCH – Memory limit for archive task: 300
☑ NPES_IOSERVER – Number of tasks to use for IO server: 0
☐ NPES_FC – Number of Processors for forecast task: 702
☐ THREADS_FC – Number of threads for forecast task: 6
☐ HYPERTHREADING – Use hyperthreading: On (default)
☐ MEM_FC – Memory limit for forecast task: 1706
☐ CPUTIME_FC – CPU time limit for forecast task: 120000
☐ NPES_INI – Number of Processors for Interpolation task: 108
☐ THREADS_INI – Number of threads for initial data task: 6
☐ MEM_INI – Memory limit for Interpolation task: 960
☐ CPUTIME_INI – CPU time limit for Interpolation task: 2000
☐ SCHOST – default supercomputer tasks host: ccb
☐ SCLOGHOST – Host for logserver: ccb-log
☐ SCLOGPORT – Logserver port: 9314
☐ FLEX_SUBMIT – Allow dynamic submission to smallest queue: On (default)
☐ USE_DARSHAN – I/O Diagnostics package: Off (default)

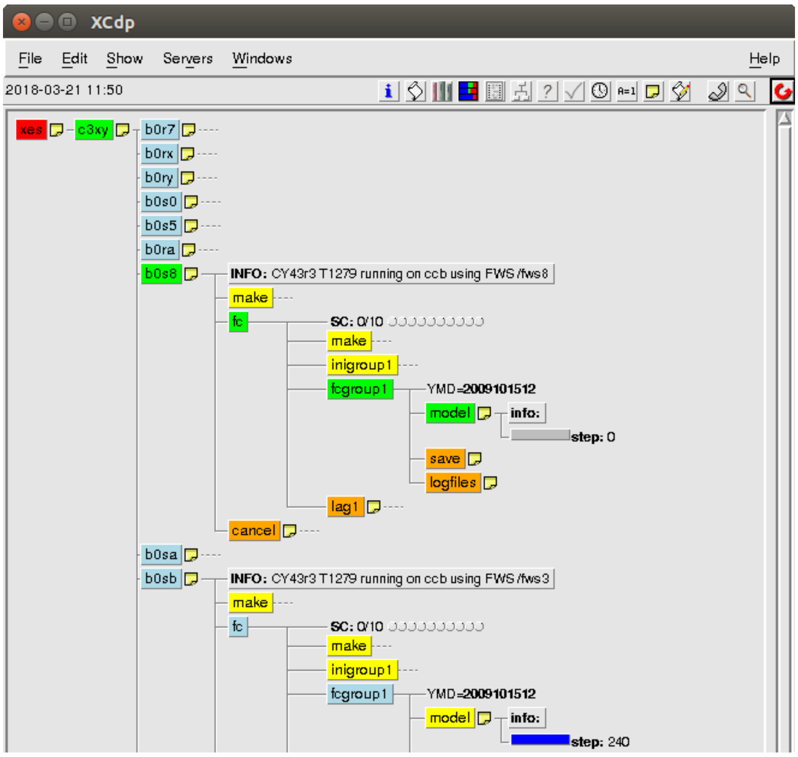
```
Loading user folder for user c3xy, please wait
User folder for user c3xy loaded
Loading experiment type c3xy/43r3/b0s8/fc, please wait
Constructor read current exptype fc
loaded ExperimentType
```

Figure 3.11: Snapshot of prepIFS showing some computational parameters of the experiment b0s8

Furthermore, in prepIFS we have to specify the Git repository of our development. It expects two different repositories, one for the source code and the other one for the scripts.

Once the experiment's setup is done, prepIFS is able to submit the experiment, which is managed with another tool called XCdp.

XCdp is a graphic interface to make use of the Supervisor Monitor Scheduler (SMS), which is in charge of submitting batch jobs to different hosts. In Figure 3.12 there is a snapshot of the XCdp tool. The forecast task (model) of the experiment b0s8 is in green, which means that is currently running.



INFO: CY43r3 T1279 running on ccb using FWS /fws8

INFO: CY43r3 T1279 running on ccb using FWS /fws3

Figure 3.12: Snapshot of XCdp showing some tasks of the experiment b0s8

It is also possible to individually set up variables for each task of XCdp. These variables need to be between percentage symbols: %variable_name%. We have used this strategy for some XIOS variables, such as the number of servers, server or attached mode, etc., because it is not possible to set it up in prepIFS.

3.2 XIOS description

3.2.1 Overview

The XML Input/Output Server (XIOS) [14][28][34][36][46][47] is an asynchronous MPI parallel I/O server that is used by Earth system models to avoid contention in the I/O. It focuses on offering high performance to achieve very high scalability with support for high-resolution output. XIOS is developed by the IPSL with an Open Source CEA CNRS INRIA Logiciel Libre (CeCILL) License. It has the following features:

- Usability in the definition and management of the I/O with a user-friendly XML configuration file.
- Avoid the I/O performance issue with dedicated parallel and asynchronous servers.
- Post-processing of fields can be performed online using an internal parallel workflow and dataflow.

XIOS is especially targeted to Earth system models with these characteristics:

- Coupled models
- Long simulations
- A lot of data is generated
- Contribute to the CMIP project

Interestingly, all the previous points (except the last one that depends on the purpose of the model) are inherent in climate models, such as EC-Earth. They are made of several coupled components that perform really long simulations. This implies to generate a lot of data that in some cases is used to contribute to the CMIP project.

In Figure 3.13 there is an overview of the schematic architecture used in XIOS. Each one of the model processes run its own XIOS client using the XIOS API. This is part of the client side, i.e., it is run on the model processes. Then, XIOS clients communicate data to XIOS servers using asynchronous MPI messages. They are run on independent nodes with regard to the model nodes. This is the server side, which uses its own MPI communicator to perform online post-processing over the received data. After that, XIOS servers can write post-processed data into the storage system using two different strategies: one single file or multiple files (one per XIOS server). The whole configuration is described in the *iodef.xml* file. In Figure 3.13 is used the one single file strategy.

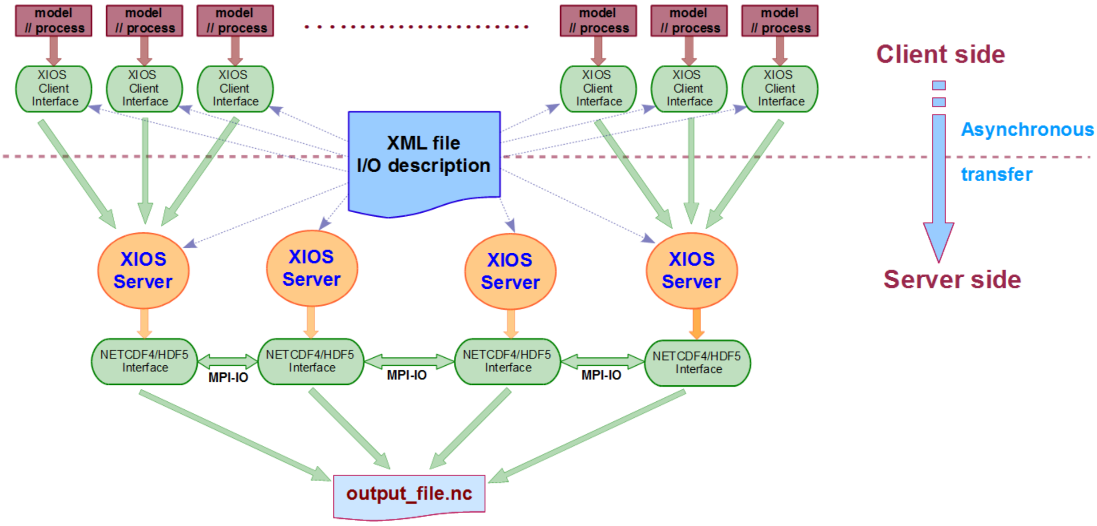


Figure 3.13: Overview of the XIOS architecture. Model processes are communicated with the XIOS servers using asynchronous MPI messages. All the framework is configured using an XML file (Reproduced from [14])

Furthermore, although Figure 3.13 shows an XIOS configuration using the server mode (dedicated I/O processes), it is also possible to use the client mode. In client mode, XIOS servers are not used and as a consequence, XIOS clients are the responsible of doing online post-processing and writing data into the storage system, either using one single file or multiple files. This implies that model processes cannot continue with the simulation until they have finished post-processing and data writing.

Regarding the aggregation strategy used to send data from clients to servers, XIOS re-distributes clients' data between servers as evenly as possible to use an optimal balance. Figure 3.14 illustrates an small example of 6 clients and 3 servers, where data is re-distributed so that servers have a proportional amount of data.

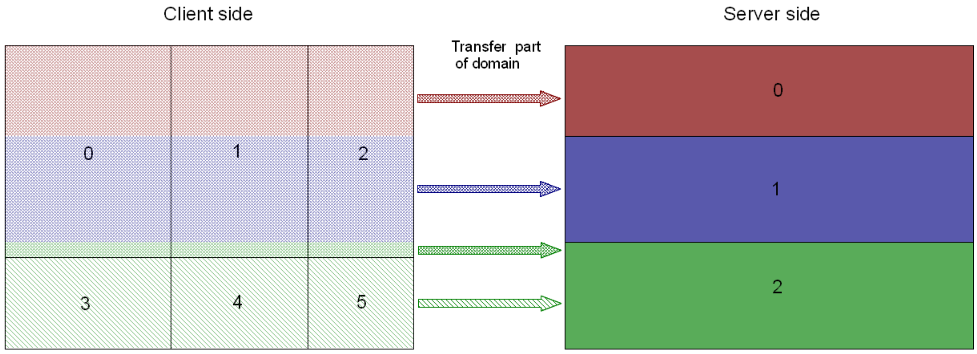


Figure 3.14: Example of XIOS data distribution between clients and servers (Reproduced from [14])

3.2.2 Concept

The philosophy of XIOS is really simple: at each time step the model can expose its data to XIOS using just one subroutine. But first, it is needed to update the calendar of XIOS to inform which is the current time step of the model. This is done using the subroutine of the Code snippet 3.5.

```fortran
CALL xios_update_timestep(ts)
```

Code snippet 3.5: XIOS subroutine to update the calendar

Then it is possible to use the respective subroutines to read (Code snippet 3.6) or write (Code snippet 3.7) data. They must be called for each one of the model variables or fields that we want to read or write.

```fortran
CALL xios_recv_field("field_id",field)
```

Code snippet 3.6: XIOS subroutine to read a field

```
CALL xios_send_field(”field_id”, field_out)
```

Code snippet 3.7: XIOS subroutine to write a field

Both subroutines have two arguments, the first one is the identifier of the variable or field and the second one is the data array, which can have different dimensions depending on how was defined.

Since the approach of XIOS is to call these two subroutines at each time step, there could be some situations where this is not necessary. For example, if a field is disabled in the XML configuration file so that it will not be output, we could avoid to send data that will not be used. The Code snippet 3.8 shows how to use the function that queries if a field is used in the simulation.

```
IF (xios_field_is_active(”field_id”)) THEN
    CALL xios_send_field(”field_id”, field_out)
ENDIF
```

Code snippet 3.8: XIOS subroutine to query if a field is active

There is another similar situation: if a field is not written, but it is used to calculate a diagnostic. In this case if the sampling frequency to post-process the diagnostic is lower than the time step frequency, in some time steps the data of the field is not needed. Therefore, we could avoid to send data that will not be used. We can use the optional second argument of the previous function setting it up to true. The function of the Code snippet 3.9 queries if a field is active or not in the current time step.

```
xios_field_is_active(”field_id”, .true.)
```

Code snippet 3.9: XIOS subroutine to query if a field is active in the current time step

Alternatively, we can set the field attribute *check_if_active* to true so that XIOS internally will check if a field is active before sending any data. Code snippet 3.10 shows how to do it. It is equivalent to use the function of the Code snippet 3.9.

```xml
<field_definition>
  <field id=”field_id” grid_ref=”grid_id” operation=”instant” check_if_active=”true”
      ↪ />
</field_definition>
```

Code snippet 3.10: XML definition to output a field if active in the current time step

Finally, one important aspect of the XIOS philosophy is to make the initial configuration as easy as possible and posterior re-configurations to be as quick as possible by avoiding re-compilations. This is simply achieved by changing parameters in the XML configuration file. Therefore, there are two types of setup that are complemented:

- Static setup through the XML file. It is parsed at runtime at the beginning of the execution to modify the XIOS internal workflow and the user output definition. It follows a hierarchical approach using the inheritance concept.
- Dynamic setup through the Fortran API. It is used to modify the XML definition, or extend it. For example, the grid setup should be done dynamically since subdomain decomposition is different according to the number of MPI tasks and is done at model runtime.

The key point is that if the integration of XIOS is properly done, users only need to make use of the static setup. The dynamic setup is only needed is particular cases, probably for model developers. For example, the model internally computes a new diagnostic that should be output through XIOS. It would be needed to make some easy changes to send the new diagnostic data array to XIOS.

3.2.3 Initialization

There are two steps to initialize XIOS:

- XIOS initialization: in this step XIOS is initialized, the XML is parsed and MPI may be initialized (if it has not already been done). The *MPI_COMM_WORLD* communicator is shared between the model and XIOS, so it is returned a local communicator to be potentially used by the model. This is done with the subroutine of the Code snippet 3.11.
- Context initialization: it performs the configuration of the context associated with ”context_id” defined in the XML file. In addition, it opens the scope where to put setup subroutines: geometry, calendar, fields, etc. This is done with the subroutine of the Code snippet 3.12.

```
CALL xios_initialize(”code_id”, return_comm=communicator)
```

Code snippet 3.11: Subroutine to initialize XIOS

```
CALL xios_context_initialize(”context_id”,communicator)
```

Code snippet 3.12: Subroutine to initialize the XIOS context

3.2.4 Finalization

The two steps of the initialization are inversely undone:

- Context finalization: close contexts once they are processed. This is done with the subroutine of the Code snippet 3.13.
- XIOS finalization: close servers, opened files and generate the performance report to know if we are using enough or too servers. It also finalizes MPI if it was initialized by XIOS. This is done with the subroutine of the Code snippet 3.14.

```
CALL xios_context_finalize
```

Code snippet 3.13: Subroutine to finalize the XIOS context

```
CALL xios_finalize
```

Code snippet 3.14: Subroutine to finalize XIOS

3.2.5 Grid definition

It is necessary to set up on XIOS the grid that is exactly used by the model and the data decomposition for each MPI process as well.

XIOS is able to use grids of any dimension: 0D, 1D, 2D, 3D, etc. The most commonly used is a 3D grid which is made of a 1D-vertical axis and a 2D-horizontal domain. Axis and domain are the two elements used in XIOS to define any kind of grid.

Axes are generally used to describe the vertical direction of a grid. There are several variables to describe how data is stored in memory and mapped into the grid. Figure 3.15 shows the basic variables of the axis element.

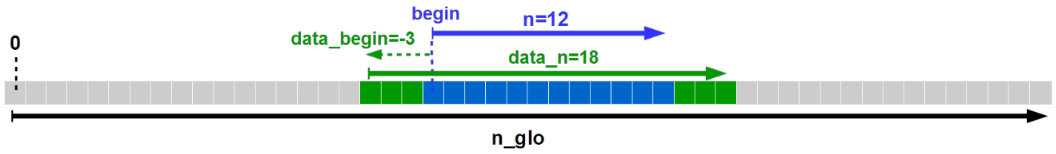


Figure 3.15: Basic variables to describe an axis (Reproduced from [14])

Domains describe the type of horizontal layer that maps the Earth's surface. In XIOS there are four different domains available:

- Regular Cartesian
- Curvilinear
- Reduced Gaussian
- Unstructured

There is an example of a reduced Gaussian domain in Figure 3.16. As mentioned, this type of domain is the new one used in IFS.

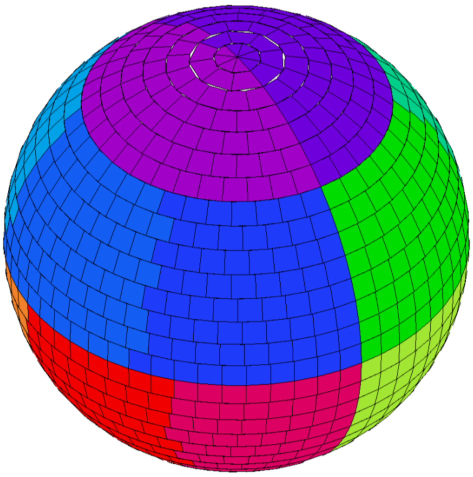


Figure 3.16: Example of a reduced Gaussian domain (Reproduced from [14])

As in the axes, domains also have some basic variables used to describe the geometry of the global domain and local subdomains. Figure 3.17 shows the basic variables, which for describing a regular Cartesian domain would be enough, but for a reduced Gaussian one, we would need to set up some different and a bit more difficult variables. This will be explained in the development Section 4.

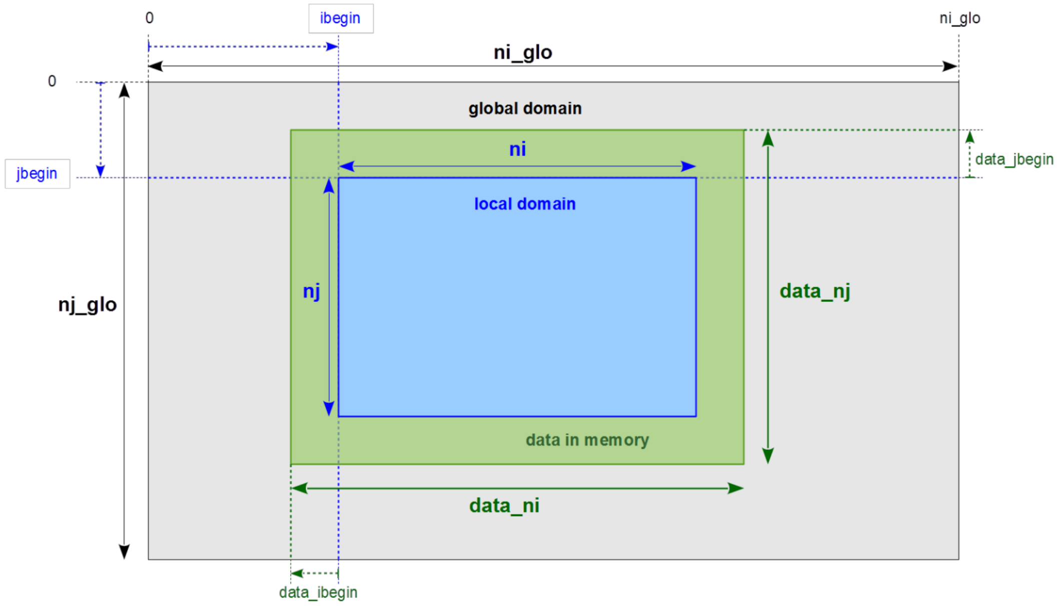


Figure 3.17: Basic variables to describe the global domain and a subdomain (Reproduced from [14])

3.2.6 Fields

Fields are typically declared through the XML file as it shows Code snippet 3.15.

```
<field_definition>
  <field id="temp" grid_ref="grid_3d" />
  <field id="precip" grid_ref="grid_3d" />
  <field id="pressure" domain_ref="domain_2d" />
</field_definition>
```

Code snippet 3.15: Example of an XML field definition

Data of fields is distributed according to the underlying grid description. They have several attributes that will be associated in the output file: name, standard_name, unit, etc. In addition, there are other configurable parameters such as the sampling frequency, the compression level, etc.

3.2.7 Files

The declaration of files is similar to the fields one, also using the XML file. In Code snippet 3.16 there is one file defined with the name ”daily_output”, containing several fields with different online post-processing options.

```
<file_definition>
  <file name=”daily_output” freq_output=”1d”>
    <field field_group_ref=”fields_3d” operation=”average” />
    <field_group operation=”instant”>
      <field field_ref=”temp” name=”temp_inst” />
      <field field_ref=”pressure” name=”pressure_inst” />
    </field_group>
    <field field_ref=”pressure” operation=”average” />
  </file>
</file_definition>
```

Code snippet 3.16: Example of an XML file definition

There are some configurable parameters: write one or multiple files, compression, netCDF version, etc.

3.2.8 Filters

An essential part of XIOS for performing online post-processing are filters. Depending on the type of post-processing operations, there are several different filters that are applied to fields during runtime. Internally, filters are applied to data fluxes with a timestamp, which represent fields. Figure 3.18 shows a workflow with its respective fluxes and filters to generate diagnostics to be written into files. These workflows are internally used by XIOS, users do not have to deal with them.

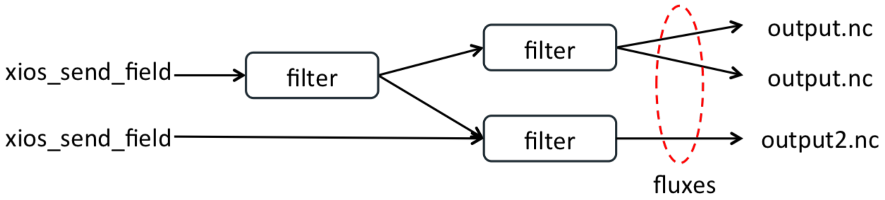


Figure 3.18: Example of a workflow applying filters to fields

There are three types of filters:

- Arithmetic filters: they combine fluxes of the same timestamp together applying arithmetic operations. Fluxes must be represented with the same grid. Code snippet 3.17 shows how to represent the two following arithmetic operations using the XML:

$$C = \frac{A + B}{A * B}$$

$$D = \frac{e^{-C*D}}{3}$$

- Time integration filters: they integrate a flux over a period of time. There are different types of operations: once, instant, maximum, minimum, average and accumulate. It is possible to chain time filters with "@". Code snippet 3.18 shows an example.
- Spatial filters: they are used to change the geometry of the fields. There are different parallel operations:
  - Data extraction: zooming, slicing, etc
  - Global or spatial reduction: mean, max, min, etc
  - Horizontal interpolation
  - Polynomial vertical interpolation
  - Pressure levels interpolation
  - Connectivity discovery

– ...

Code snippet 3.19 shows a code to interpolate a regular domain to an unstructured domain.

```
<field id=”A” />
<field id=”B” />
<field id=”C” > (A + B)/(A∗B) </field>
<field id=”D” > exp(−C∗this)/3 </field>
```

Code snippet 3.17: Example of an arithmetic filter

```
<field id=”temp” operation=”average” />
<field id=”temp_min” field_ref=”temp” operation=”minimum” />
<field id=”temp_max” field_ref=”temp” operation=”maximum” />

<file name=”monthly_output” freq_output=”1mo” >
  <field name=”ave_daily_min” operation=”average” freq_op=”1d”> @temp_min </
      ↪ field>
  <field name=”ave_daily_max” operation=”average” freq_op=”1d”> @temp_max </
      ↪ field>
  <field name=”min_daily_ave” operation=”minimum” freq_op=”1d”> @temp </
      ↪ field>
  <field name=”max_daily_ave” operation=”maximum” freq_op=”1d”> @temp </
      ↪ field>
</file>
```

Code snippet 3.18: Example of a time integration filter

```
<field id=”temp” grid_ref=”grid_regular” />
<field id=”new_temp” field_ref=”temp” grid_ref=”grid_unstructured” />

<axis id=”vert_axis” n_glo=”100” />
<domain id=”regular” ni_glo=”360” nj_glo=”180” type=”rectilinear” />
<domain id=”unstructured” ni_glo=”10000” type=”unstructured” />

<grid id=”grid_regular”>
  <domain domain_ref=”regular” />
  <axis axis_ref=”vert_axis” />
</grid>

<grid id=”grid_unstructured”>
  <domain domain_ref=”unstructured”>
    <interpolate_domain />
  </domain>
  <axis axis_ref=”vert_axis” />
</grid>
```

Code snippet 3.19: Example of a spatial filter

3.2.9 Performance tuning

At the end of the execution, XIOS generates individual performance reports for all clients and servers to know details about memory consumption and different execution times (total, waiting, ratio, etc).

There are different options to tune the performance of XIOS:

- Test the two possible run modes: attached mode or server mode. Theoretically, server mode should give more performance, but additional computational resources are needed.

- Test the two possible writing modes: single file or multiple files. Theoretically, multiple files should give more performance due to higher scalability, but post-processing to merge all files is needed.

- Test different values for some memory buffer parameters.

3.2.10 CMIP

Some climate models are tuned to perform simulations that will contribute to the CMIP project. The Coupled Model Intercomparison Project (CMIP) [13][48][49] is a standard experimental protocol for studying the output of coupled AOGCMs. The ongoing version is the sixth, known as CMIP6. One of the requisites is that data must follow a very strict standard, so almost all models need to post-process data to fulfill the standard. This process is known as Climate Model Output Rewriter (CMOR) post-processing or CMORization.

As we mentioned, XIOS is able to ideally produce netCDF files according to the CMIP6 data request, being ready to be distributed and published. However, this is not automatically done, so a minimum user effort is needed to write the proper XML configuration file.

This XIOS feature avoids (or at least minimize) to perform the costly and slow CMORization.

3.3 Computing environment

The ECMWF's High Performance Computing Facility (HPCF) [50] is a Cray system that has two identical Cray XC40 clusters. They have their own storage, but with equal access to the high performance working storage of the other cluster. This cross-connection of storage allows most of the benefits of having one very large system, but dual clusters add significant resilience to the whole system, allowing flexibility in performing maintenance and upgrades. In addition, this is combined with separate resilient power and cooling systems that provide protection against a wide range of possible failures. There is an overview of the Cray HPCF in Figure 3.19. It shows the two Cray XC40 clusters and how they are interconnected. There are components such as storage, compute nodes, I/O nodes, etc.

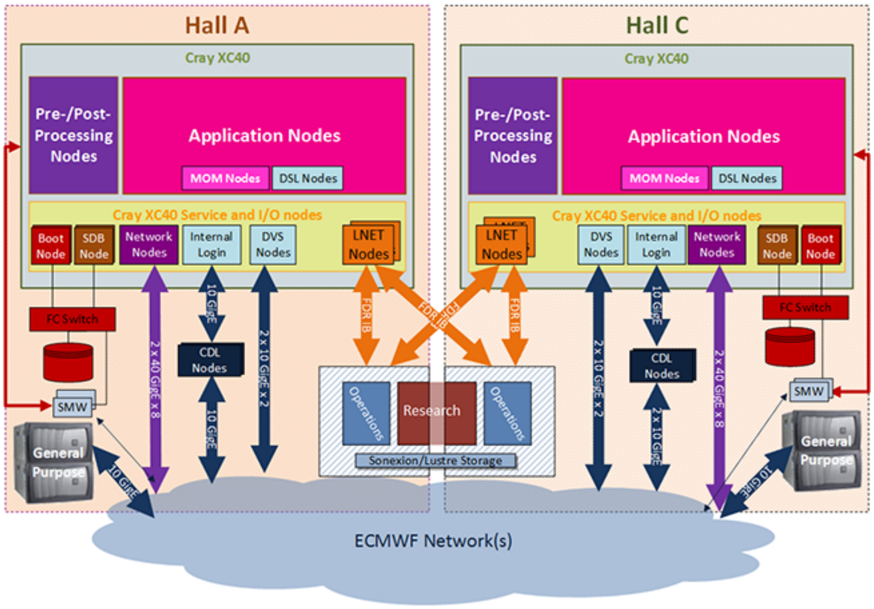


Figure 3.19: Overview of the main components of the Cray HPCF (Reproduced from [50])

Each Cray XC40 cluster has 20 cabinets of compute nodes and 13 of storage. The bulk of the system consists of compute nodes with two Intel Xeon EP E5-2695 V4 ”Broadwell” processors each with 18 cores. Four compute nodes sit on one blade, sixteen blades sit in a chassis and there are three chassis in a frame. This gives a maximum of 192 nodes or 6912 processor cores per cabinet. The number of actual compute nodes in a cabinet will sometimes be less than the maximum since each cluster has a number of ”Service Nodes”. In Tables 3.2, 3.3, 3.4 and 3.5 there is a summary of the main characteristics of the whole system.

ECMWF Cray HPCF	
Compute clusters	2 XC40
Peak performance (teraflops)	8499
Sustained performance on ECMWF codes (teraflops)	333

Table 3.2: Summary of the Cray HPCF (Adapted from [50])

Cray XC40 compute cluster	
Compute nodes	3610
Compute cores	129960
Operating system	Cray CLE 5.2 UP04
High performance interconnect	Cray Aries
High performance parallel storage (PB)	10.0
General-purpose storage (TB)	38

Table 3.3: Summary of the Cray XC40 (Adapted from [50])

Cray XC40 compute node	
Memory (GiB)	128 (4 with 256)
Processor type	Intel E5-2695v4 ”Broadwell”
Processors per node	2
Cores per processor	18

Table 3.4: Summary of a Cray XC40 compute node (Adapted from [50])

Cray XC40 core	
Threads per core	1, 2
Clock frequency (GHz)	2.1
Operations per clock cycle	16
L1/L2/L3 cache	64KiB/256KiB(private)/45MiB(shared)

Table 3.5: Summary of a Cray XC40 core (Adapted from [50])

This cluster uses the Aries™ Interconnect network technology developed by Cray. This interconnect uses a ”dragonfly” topology, which has a large number of local electrical connections and a relatively small number of longer distance optical connections.

The nodes of the Cray system are optimized to run in ”Extreme Scalability Mode”. In this mode, each node runs a stripped down version of the Linux operating system. Reducing the number of operating system tasks running on a node to the minimum is a key element to provide a highly scalable environment for HPC applications. However, there are other types of nodes that run full version of the Linux operating system. They are used to run jobs that require less than one node to run, basically pre- and post-processing jobs for the main parallel jobs.

Finally, there are two types of storage: the high performance storage used to offer performance for compute clusters and provided by Lustre; and the general-purpose storage used to provide space for home file systems and for storing applications.

Chapter 4

Development of the IFS-XIOS integration

4.1 Overview

In previous Chapter 3 we have given an in-depth explanation of both IFS and XIOS to understand the details of their integration.

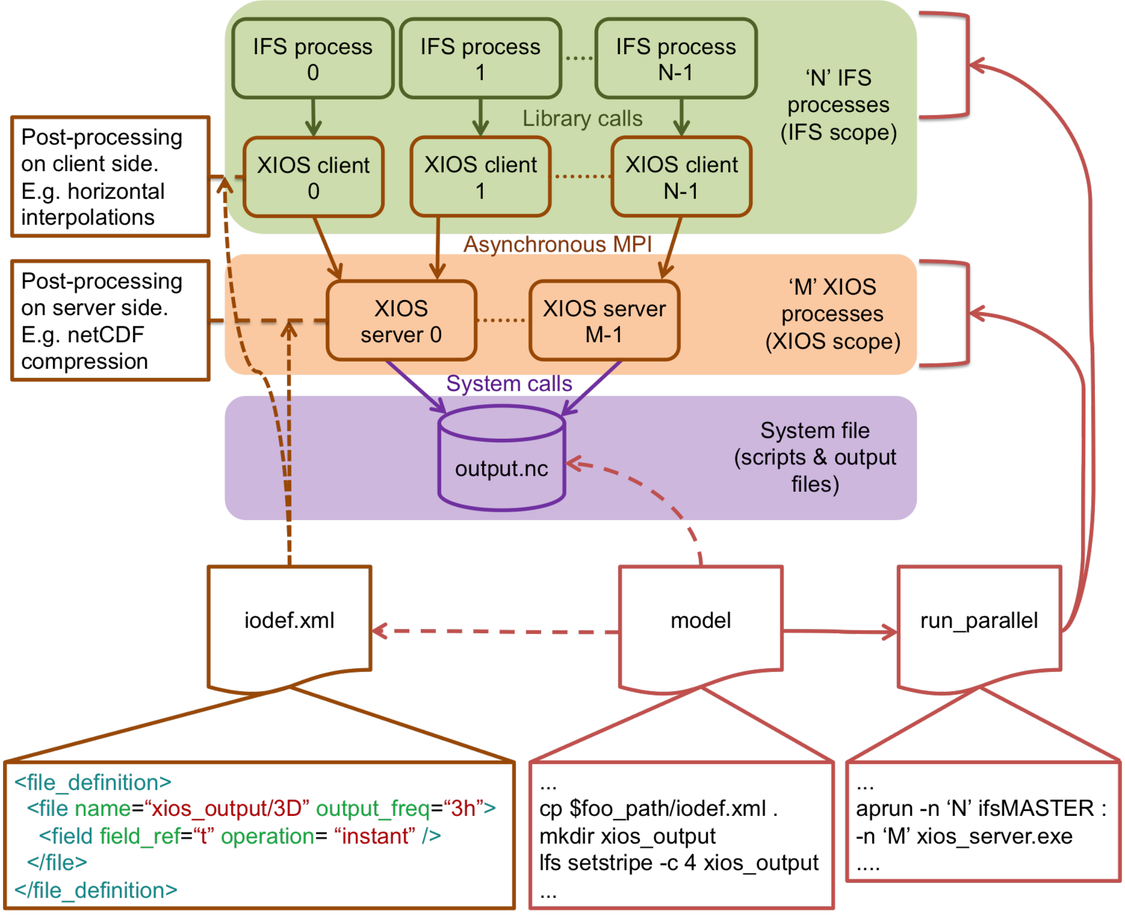
Apart from the work done during the development phase, there are two aspects that take a lot of time and should be briefly mentioned. First of all, IFS is a really complex code, so a lot of effort is needed to understand how it works. In addition, the documentation is quite limited and the source code does not have too many comments.

The second aspect is related to the IFS environment. It has a complex hierarchy of scripts to prepare and run the model, which are managed using the previously explained prepIFS and XCdp tools. The main issue is that this environment is quite closed and restrictive, so the development was done taking into account several requirements.

Figure 4.1 shows the IFS-XIOS integration scheme implemented and illustrates how the different parts of both components are interconnected. In green, it is shown the IFS processes which execute the client side of XIOS through its API. They send data using asynchronous MPI communications to XIOS servers. XIOS servers are run on server side, which are represented in orange. Finally, servers send data to the storage system (in purple) through system calls. Both XIOS clients and servers are configured with the *iodef.xml* file.

Furthermore, this figure also shows that post-processing is performed in both clients and servers. This is because depending on the type of post-processing, some operations are performed on client side, such as horizontal interpolations, and some other on server side, such as netCDF compression.

There is a pair of scripts, *model* and *run_parallel*, that are in charge of executing the whole integrated system. In particular, *run_parallel* executes both binaries in Multiple-Program Multiple-Data (MPMD) mode using aprun.


IFS process 0
IFS process 1
IFS process N-1
'N' IFS processes (IFS scope)
Library calls
XIOS client 0
XIOS client 1
XIOS client N-1
Post-processing on client side. E.g. horizontal interpolations
Asynchronous MPI
XIOS server 0
XIOS server M-1
'M' XIOS processes (XIOS scope)
Post-processing on server side. E.g. netCDF compression
System calls
output.nc
System file (scripts & output files)
iodef.xml
model
run_parallel

```
<file_definition>
  <file name="xios_output/3D" output_freq="3h">
    <field field_ref="t" operation= "instant" />
  </file>
</file_definition>
```

```
...
cp $foo_path/iodef.xml .
mkdir xios_output
lfs setstripe -c 4 xios_output
...
```

```
...
aprun -n 'N' ifsMASTER :
-n 'M' xios_server.exe
....
```

Figure 4.1: Scheme of the IFS-XIOS integration. It overviews how the different parts are interconnected.

If we focus on the structure of our code, we have followed the coding standards of IFS. They have different prefixes for file names, where in general "yom" is used for variables, "su" is used for setup routines and "c" is used for control routines. We have implemented three different files:

- *yomxios.F90*: it contains variables related to XIOS. For example the context handle, time step, data arrays, etc.
- *suxios.F90*: it contains all the routines needed to initialize XIOS from IFS. There are three public subroutines: *suxios_ini*, *suxios_fin* and *suxios_ctxt*.
- *cxios.F90*: it contains one public routine called *ifs_xios_send_fields* to send

data.

The development has been designed so that it is really simple to use and maintain. It is possible by calling the previous mentioned four public subroutines:

- *suxios_ini*: it is called at the beginning of IFS because MPI is initialized by XIOS.
- *suxios_fin*: it is called at the end of IFS when MPI is no needed anymore, because XIOS finalizes MPI.
- *suxios_ctxt*: it is called during the IFS setup. It performs the configuration of XIOS with the parameters of IFS.
- *ifs_xios_send_fields*: it is called at the end of each time step to send fields to XIOS.

The *ifs_xios_send_fields* subroutine is internally designed so that it has three different parts or steps. This three steps are: update calendar, NPROMA blocks gather and send fields. If IFS is running an output time step, all three steps will be sequentially executed; otherwise if IFS is running a non-output time step, only the update calendar step will be executed.

Figure 4.2 shows how these three steps are executed at the end of an output time step. For illustrative purposes, in this example all time steps perform output.

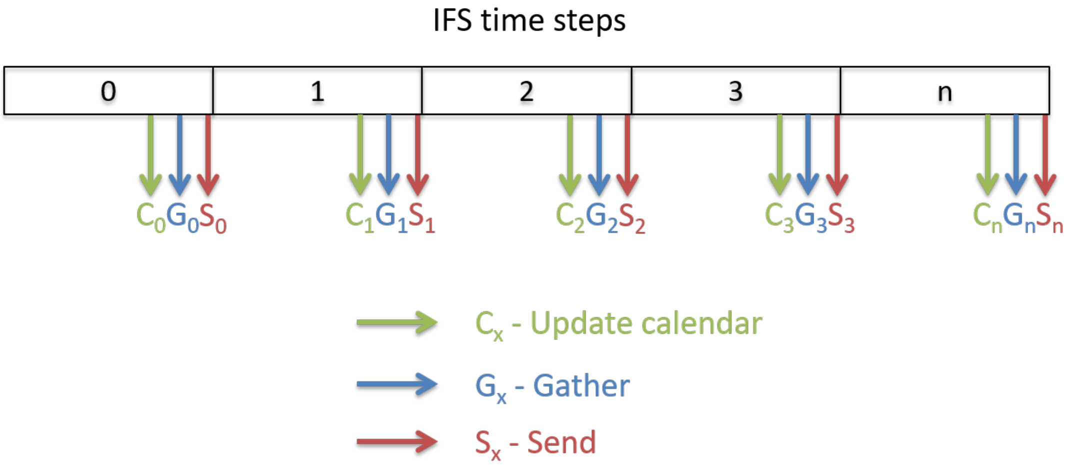


Figure 4.2: Output scheme developed for IFS: all three steps, update calendar, NPROMA blocks gather and send fields are sequentially executed at the end of and output time step

In the following sections we will explain more in detail how the previous files are implemented and the changes in the scripts. In Section 4.2 we explain the necessary steps to perform the XIOS setup from IFS, i.e., pass the IFS parameters that XIOS requests. It includes the design of both *yomxios.F90* and *suxios.F90* files. Then, in Section 4.3 we explain how we implemented the grid-point fields transfer. This is basically the design of the *cxios.F90* file. Finally, in Section 4.4 we explain the necessary changes in the IFS environment, including *model* and *run_parallel* scripts, to be able to run the whole integrated system.

4.2 XIOS setup

In order to set up XIOS to be fully operational, we have to follow a series of steps. First of all, XIOS must be initialized, which internally initializes MPI. Then, we have to setup the whole context. This part includes the calendar, which is very important to have both IFS and XIOS fully coordinated when the simulation progresses along the time. It also includes the geometry of IFS. This is the definition of the IFS grid, including the horizontal domain and the vertical axis. Thus, XIOS knows how received data from different independent processes has to be placed over the world. Finally, once the simulation finishes, we have to finalize XIOS, which internally also finalizes MPI.

4.2.1 Data and variables module

According to our development structure, we have a module in *yomxios.F90* that contains variables related to the XIOS context, calendar variables, identifiers for the axes and domains, and three data arrays that are used to send fields to XIOS. Code snippet 4.1 shows the yomxios module.

```
MODULE yomxios

USE PARKIND1, ONLY : JPIM, JPRB
USE xios

IMPLICIT NONE

SAVE

! XIOS context
TYPE(xios_context) :: context_handle
CHARACTER(len=3) :: model_name = ”ifs”
CHARACTER(len=3) :: ifs_context = ”ifs”

! Calendar management
TYPE(xios_date) :: time_origin
TYPE(xios_date) :: start_date
TYPE(xios_duration) :: time_step

! Axes definition
CHARACTER(len=12) :: model_axis_name = ”model_levels”

! Domains definition
CHARACTER(len=16) :: gaussian_domain_name=”reduced_Gaussian”

! Arrays to gather NPROMA blocks and send fields to XIOS
REAL(KIND=JPRB), ALLOCATABLE :: xios_gmv(:,:), xios_gmvs(:), xios_gfl(:,:)

END MODULE yomxios
```

Code snippet 4.1: Overview of the data and variables module

4.2.2 Initialization

The first step is to initialize XIOS and MPI. It is possible to initialize MPI through IFS or XIOS. According to the XIOS documentation, they recommend to initialize it through XIOS. This means that we have to ensure that IFS does not call *MPI_Init*.

As mentioned, XIOS will use the *MPI_COMM_WORLD* communicator for both components, it will use its own communicator, and it will return a local communicator to be used by IFS.

IFS is already prepared to work with *MPI_COMM_WORLD* or with a provided local communicator. This avoided us to adapt IFS to support provided communicators.

The mechanism to change the IFS communicator is through two variables: *LMPLUSERCOMM* and *MPLUSERCOMM*. Code snippet 4.2 shows how to do it. Setting up *LMPLUSERCOMM* to true, we are specifying to use a provided communicator, which will be provided with the *MPLUSERCOMM*. When we initialize XIOS, the subroutine *xios_initialize* returns a local communicator.

```fortran
! Enabling the usage of MPLUSERCOMM communicator, rather than MPI_COMM_WORLD
LMPLUSERCOMM = .TRUE.

! Initialization of XIOS and definition of the MPLUSERCOMM communicator to be used by IFS
CALL xios_initialize(model_name,return_comm=MPLUSERCOMM)
```

Code snippet 4.2: Mechanism to change IFS communicators

This code is part of the public *suxios_ini* subroutine.

4.2.3 Finalization

After the simulation, it is needed to finalize XIOS and MPI. Code snippet 4.3 shows how to do it. First of all, we deallocate some arrays that are used for sending fields to XIOS (it is explained in Section 4.3).

Then, we have to call *xios_context_finalize* and *xios_finalize* subroutines to finalize XIOS and MPI. Since in the initialization we specified to IFS to use a provided communicator, it will not call *MPI_Finalize*, which is called by XIOS.

```fortran
! Deallocating XIOS arrays
DEALLOCATE(xios_gmv)
DEALLOCATE(xios_gmvs)
DEALLOCATE(xios_gfl)

! Finalization of XIOS context
CALL xios_context_finalize()

! Finalization of XIOS and MPI
CALL xios_finalize()
```

Code snippet 4.3: MPI and XIOS finalization

This code is part of the public *suxios_fin* subroutine.

4.2.4 Context

One important step is the XIOS context setup. Code snippet 4.4 shows how to perform the setup. There are three first subroutines to start the IFS context definition scope, which is closed using the *xios_close_context_definition*. It contains three main private subroutines:

- *ifs_xios_set_calendar*: it is used to inform XIOS which type of calendar uses IFS, the start date of the simulation and the duration of the time step. Section 4.2.5 explains how it is implemented.

- *ifs_xios_set_axis* and *ifs_xios_set_domain*: both subroutines are used to set up the geometry of the model. It is essential to inform XIOS about the distribution of each one of the grid-points over the Earth, as well as how they are distributed between all IFS processes. Thus, XIOS can output valid data, where each grid-point has a longitude and a latitude over the Earth's surface. Section 4.2.6 explains how they are implemented.

After the context definition close, we allocate some arrays used to send fields to XIOS (the same arrays that we deallocated in Section 4.2.3, i.e., in finalization).

```fortran
! Context initialization
CALL xios_context_initialize(ifs_context, MPLUSERCOMM)
CALL xios_get_handle(ifs_context, context_handle)
CALL xios_set_current_context(context_handle)

! Date setting
CALL ifs_xios_set_calendar

! Definition of axes
CALL ifs_xios_set_axis(YDGEOMETRY)

! Definition of domains
CALL ifs_xios_set_domain(YDGEOMETRY)

! Close context definition
CALL xios_close_context_definition()

! Allocating XIOS arrays
ALLOCATE(xios_gmv(YDGEOMETRY%YRGEM%NGPTOT,YDGEOMETRY%
    ↪ YRDIMV%NFLEVG))
ALLOCATE(xios_gmvs(YDGEOMETRY%YRGEM%NGPTOT))
ALLOCATE(xios_gfl(YDGEOMETRY%YRGEM%NGPTOT,YDGEOMETRY%
    ↪ YRDIMV%NFLEVG))
```

Code snippet 4.4: Needed calls to set up the XIOS context

This code is part of the public *suxios_ctxt* subroutine.

4.2.5 Calendar

XIOS needs to know the type of calendar that IFS is using, in this case the Gregorian one. It is set up calling *xios_define_calendar*. Then, we set up the time origin and the start date using *xios_set_time_origin* and *xios_set_start_date* subroutines respectively. In this case, we use the same date and time for both. IFS stores the initial date using format AAAAMMDD in variable *NINDAT*, and the initial time in seconds in variable *NSSSSS*.

Finally, the last parameter to be set up is the time step that IFS is using. IFS stores the time step in a variable called *TSTEP*, and it is passed to XIOS using the *xios_set_timestep* subroutine. Code snippet 4.5 shows how the calendar setup is done.

```
INTEGER(KIND=JPIM) :: year, month, day, hours, minutes, seconds

year = NINDAT/10000
month = MOD(NINDAT/100, 100)
day = MOD(NINDAT, 100)
hours = NSSSSS/3600
minutes = (NSSSSS - hours*3600)/60
seconds = NSSSSS - hours*3600 - minutes*60

CALL xios_define_calendar(type="Gregorian")

! Time origin of the time axis. It will appear as meta-data attached to the time axis in the output file
CALL xios_set_time_origin(time_origin=xios_date(year, month, day, hours, minutes, seconds))

! Start date of the simulation for the current context
CALL xios_set_start_date(start_date=xios_date(year, month, day, hours, minutes, seconds))

! Updated date = start_date + NSTEP*YRRIP%TSTEP
time_step%second = YRRIP%TSTEP
CALL xios_set_timestep(time_step)
```

Code snippet 4.5: XIOS setup to use IFS calendar

This code is part of the private *ifs_xios_set_calendar* subroutine.

4.2.6 Geometry

One of the most difficult parts in setting up XIOS is the IFS geometry transfer to XIOS. Since we are working with a 3D grid, it consists of two parts: the 1D axis definition and the 2D domain definition.

Code snippet 4.6 shows how to perform the vertical axis definition. Using the loop, it basically builds and array with as many positions as the number of vertical levels. Each position of the array contains the number of the vertical level. IFS stores the number of vertical levels in variable *NFLEVG*.

Then, using the *xios_set_axis_attr* subroutine, we set up different attributes for the vertical axis: total number of vertical levels, the array containing the vertical levels, the units (we are working on sigma levels, so there are no units), and the direction of the axis, which is positive.

```
INTEGER(KIND=JPIM) :: i
REAL(KIND=JPRB) :: j
REAL(KIND=JPRB), ALLOCATABLE :: zML(:)

! Definition of model levels axis
ALLOCATE(zML(YDGEOMETRY%YRDIMV%NFLEVG))

j = 1.0
DO i = 1, YDGEOMETRY%YRDIMV%NFLEVG
  zML(i) = j
  j = j + 1.0
END DO

! Output all model levels
CALL xios_set_axis_attr(model_axis_name, n_glo=YDGEOMETRY%YRDIMV%
    ↪ NFLEVG, value=zML, unit="−", positive="up")

DEALLOCATE(zML)
```

Code snippet 4.6: Vertical axis definition

Code snippet 4.6 is part of the private *ifs_xios_set_axis* subroutine.

The second part is the domain definition. First of all, we need to build the local domain data (Code snippet 4.7) by iterating over all grid-points in the global domain to find and store in the *i_index* array which are the grid-points of the local domain. Thus, each IFS process will communicate to its XIOS client which are the local grid-points.

```
!
!* Local domain data
!
j = 0
DO i = 1, ni_glo
  IF (YDGEOMETRY%YRMP%NGLOBALPROC(i) == MYPROC) THEN
    j = j + 1
    ! XIOS requires indexing from 0
    i_index(j) = i − 1
  END IF
END DO
```

Code snippet 4.7: Local domain data setup

After that, there is a bigger loop (Code snippet 4.8) which is in charge of setting up the longitudes and latitudes of each one of the local *i_index* grid-points. They are stored in arrays *lonvalue_1d* and *latvalue_1d*. Furthermore, we also have to set up the boundaries for each grid-point. Boundaries are used to delimit the area that represents each grid-point. There are 4 corners for the area of each grid-point. The easiest way to determine the boundaries of each grid-point is using the middle point between two latitudes and the middle point between two longitudes. We have to be careful in the first and last latitudes, since there are not grid-points on the poles. Boundaries are stored in arrays *bounds_lon_1d* and *bounds_lat_1d*.

```
zrgauslat(0) = 90.0_JPRB
zrgauslat(ndglg+1) = −90.0_JPRB
zrgauslat(1:ndglg) = ASIN(YDGEOMETRY%YRCSGLEG%RMU(1:ndglg))∗(180.0
    ↪ _JPRB/(RPI))

DO i = 1, ni
  !
  !∗ Longitudes and latitudes for local domain grid−points (from radians to degrees)
  !
  latvalue_1d(i) = REAL(YDGEOMETRY%YRGSGEOM_NB%GELAT(i)∗(180.0_JPRB
      ↪ /RPI),JPRB)
  lonvalue_1d(i) = REAL(YDGEOMETRY%YRGSGEOM_NB%GELAM(i)∗(180.0
      ↪ _JPRB/RPI),JPRB)

  !
  !∗ Cells' boundaries for local domain grid−points
  !
  zdeltax = 0.5_JPRB∗360.0_JPRB/REAL(YDGEOMETRY%YRGEM%NLOENG(
      ↪ YDGEOMETRY%YRGSGEOM_NB%NGPLAT(i)))
  zdeltayup = 0.5_JPRB∗(zrgauslat(YDGEOMETRY%YRGSGEOM_NB%NGPLAT(i) −
      ↪  1) − zrgauslat(YDGEOMETRY%YRGSGEOM_NB%NGPLAT(i)))
  zdeltaydw = 0.5_JPRB∗(zrgauslat(YDGEOMETRY%YRGSGEOM_NB%NGPLAT(i) +
      ↪  1) − zrgauslat(YDGEOMETRY%YRGSGEOM_NB%NGPLAT(i)))

  IF (zrgauslat(YDGEOMETRY%YRGSGEOM_NB%NGPLAT(i) − 1) == 90.0_JPRB)
      ↪ zdeltayup = 2.0_JPRB∗zdeltayup
  IF (zrgauslat(YDGEOMETRY%YRGSGEOM_NB%NGPLAT(i) + 1) == −90.0_JPRB
      ↪ ) zdeltaydw = 2.0_JPRB∗zdeltaydw

  bounds_lon_1d(1,i) = lonvalue_1d(i) + zdeltax
  bounds_lat_1d(1,i) = latvalue_1d(i) + zdeltaydw
  bounds_lon_1d(2,i) = lonvalue_1d(i) + zdeltax
  bounds_lat_1d(2,i) = latvalue_1d(i) + zdeltayup
  bounds_lon_1d(3,i) = lonvalue_1d(i) − zdeltax
  bounds_lat_1d(3,i) = latvalue_1d(i) + zdeltayup
  bounds_lon_1d(4,i) = lonvalue_1d(i) − zdeltax
  bounds_lat_1d(4,i) = latvalue_1d(i) + zdeltaydw
END DO
```

Code snippet 4.8: Longitudes

Finally, all these arrays that we have build, are passed to XIOS using the *xios_set_domain_attr* subroutine. In addition, we send other parameters, such as the type of domain (Gaussian), the number of global grid-points and the number of local grid-points. We also specify that the local domain and the data domain have the same size, i.e., there will be no halos in the data arrays that we will send for each one of the fields. Code snippet 4.9 shows how it is done.

```fortran
! Define global domain and local domain index
CALL xios_set_domain_attr(gaussian_domain_name, type='gaussian', ni_glo=ni_glo,
     ↪ ibegin=0, ni=ni, i_index=i_index)
! Define local domain data
CALL xios_set_domain_attr(gaussian_domain_name, data_dim=1, data_ibegin=0, data_ni
     ↪ =ni)
! Define longitudes and latitudes for grid−point cells
CALL xios_set_domain_attr(gaussian_domain_name, lonvalue_1d=lonvalue_1d, latvalue_1d
     ↪ =latvalue_1d)
! Define cell's boundaries
CALL xios_set_domain_attr(gaussian_domain_name, nvertex=nvertex, bounds_lon_1d=
     ↪ bounds_lon_1d, bounds_lat_1d=bounds_lat_1d)

DEALLOCATE(i_index)
DEALLOCATE(lonvalue_1d, latvalue_1d)
DEALLOCATE(bounds_lon_1d, bounds_lat_1d)
DEALLOCATE(zrgauslat)
```

Code snippet 4.9: Horizontal domain definition

Code snippets 4.7, 4.8 and 4.9 are part of the private *ifs_xios_set_domain* subroutine.

4.2.7 *Iodef.xml* file

In order to complement the dynamic setup done through the XIOS API, we also had to implement the *iodef.xml* file to perform the static setup. It contains several sections that we will briefly mention.

The first one is the field definition (Code snippet 4.10). There are declared 12 3D fields and one 2D field: temperature, ozone, surface pressure, etc. They have a reference to the corresponding grid.

```
<field_definition level=”1” enabled=”.TRUE.” default_value=”1.e20” >
  <field_group id=”3D_fields” >
    <field id=”t” long_name=”Temperature” standard_name=”t” grid_ref=”
        ↪ model_levels” unit=”K” />
    <field id=”u” long_name=”U␣component␣of␣wind” standard_name=”u” grid_ref
        ↪ =”model_levels” unit=”m∗s−1” />
    <field id=”v” long_name=”V␣component␣of␣wind” standard_name=”v” grid_ref
        ↪ =”model_levels” unit=”m∗s−1” />
    <field id=”q” long_name=”Specific␣humidity” standard_name=”q” grid_ref=”
        ↪ model_levels” unit=”kg∗kg−1” />
    <field id=”vo” long_name=”Vorticity␣(relative)” standard_name=”vo” grid_ref
        ↪ =”model_levels” unit=”s−1” />
    <field id=”d” long_name=”Divergence” standard_name=”d” grid_ref=”
        ↪ model_levels” unit=”s−1” />
    <field id=”o3” long_name=”Ozone␣mass␣mixing␣ratio” standard_name=”o3”
        ↪ grid_ref=”model_levels” unit=”kg∗kg−1” />
    <field id=”cc” long_name=”Fraction␣of␣cloud␣cover” standard_name=”cc”
        ↪ grid_ref=”model_levels” unit=”(0−1)” />
    <field id=”crwc” long_name=”Specific␣rain␣water␣content” standard_name=”
        ↪ crwc” grid_ref=”model_levels” unit=”kg∗kg−1” />
    <field id=”cswc” long_name=”Specific␣snow␣water␣content” standard_name=”
        ↪ cswc” grid_ref=”model_levels” unit=”kg∗kg−1” />
    <field id=”clwc” long_name=”Specific␣cloud␣liquid␣water␣content”
        ↪ standard_name=”clwc” grid_ref=”model_levels” unit=”kg∗kg−1” />
    <field id=”ciwc” long_name=”Specific␣cloud␣ice␣water␣content” standard_name
        ↪ =”ciwc” grid_ref=”model_levels” unit=”kg∗kg−1” />
  </field_group>

  <field_group id=”2D_dynamical_fields” >
    <field id=”sp” long_name=”Surface␣pressure” standard_name=”sp” grid_ref=”
        ↪ surface_fields” unit=”Pa” />
  </field_group>
</field_definition>
```

Code snippet 4.10: Field definition used in *iodef.xml*

Then, there is the file definition (Code snippet 4.11). It contains two different files, *file_3D_bench* and *file_2D_bench*, which are used to output 3D and 2D fields respectively. Both files have an output frequency of 3 hours, they output instant values (no post-processing applied), they are output in multiple file mode, the frequency sampling of each field is 3 hours and files are forced to be flushed into the storage system at the end of 1 simulated day.

```
<file_definition type="multiple_file" format="netcdf4" par_access="collective" sync_freq="1d" min_digits="4" >
  <file id="file_3D_bench" name="xios_output/prognostic_3D_fields_benchmarking" output_freq="3h" output_level="10" enabled=".TRUE." >
    <field field_ref="t" name="temperature" freq_op="3h" operation="instant" />
    <field field_ref="u" name="u-wind" freq_op="3h" operation="instant" />
    <field field_ref="v" name="v-wind" freq_op="3h" operation="instant" />
    <field field_ref="q" name="specific_humidity" freq_op="3h" operation="instant" />
    <field field_ref="vo" name="vorticity" freq_op="3h" operation="instant" />
    <field field_ref="d" name="divergence" freq_op="3h" operation="instant" />
    <field field_ref="o3" name="ozone" freq_op="3h" operation="instant" />
    <field field_ref="cc" name="cloud_fraction" freq_op="3h" operation="instant" />
    <field field_ref="crwc" name="rain" freq_op="3h" operation="instant" />
    <field field_ref="cswc" name="snow" freq_op="3h" operation="instant" />
    <field field_ref="clwc" name="liquid_water" freq_op="3h" operation="instant" />
    <field field_ref="ciwc" name="ice_water" freq_op="3h" operation="instant" />
  </file>
  <file id="file_2D_bench" name="xios_output/prognostic_2D_fields_benchmarking" output_freq="3h" output_level="10" enabled=".TRUE." >
    <field field_ref="sp" name="surface_pressure" freq_op="3h" operation="instant" />
  </file>
</file_definition>
```

Code snippet 4.11: File definition used in *iodef.xml*

After that, there are the definitions of axes, domains and grids (Code snippet 4.12). We simply have one vertical axis, one reduced Gaussian domain, and two different grids: *model_levels* to represent 3D fields and *surface_fields* to represent 2D fields.

```
<axis_definition>
  <axis id=”model_levels” long_name=”vertical_model_levels” unit=”−” positive=”
      ↪ up” />
</axis_definition>

<domain_definition>
  <domain id=”reduced_Gaussian” long_name=”octahedral_reduced_Gaussian_grid”
      ↪ type=”gaussian” />
</domain_definition>

<grid_definition>
  <grid id=”surface_fields” description=”2D_dynamical_and_physical_output_fields” >
    <domain domain_ref=”reduced_Gaussian” />
  </grid>
  <grid id=”model_levels” description=”3D_output_fields_on_model_levels” >
    <domain domain_ref=”reduced_Gaussian” />
    <axis axis_ref=”model_levels” />
  </grid>
</grid_definition>
```

Code snippet 4.12: Axis, domain and grid definition used in *iodef.xml*

Finally, there are some variables which are used to set up XIOS from a computational point of view: size of the buffers, server or attached mode, info level, print performance reports, etc. Code snippet 4.13 shows the variables used.

```
<context id=”xios”>
  <variable_definition>
    <variable_group id=”buffer” >
      <variable id=”optimal_buffer_size” type=”string”>performance</variable>
      <variable id=”buffer_size_factor” type=”double”>1.0</variable>
    </variable_group>

    <variable_group id=”parameters” >
      <variable id=”using_server” type=”bool”>XIOS_USING_SERVER</variable>
      <variable id=”using_oasis” type=”bool”>false</variable>
      <variable id=”info_level” type=”int”>50</variable>
      <variable id=”print_file” type=”bool”>true</variable>
    </variable_group>
  </variable_definition>
</context>
```

Code snippet 4.13: XIOS variables definition used in *iodef.xml*

4.3 Grid-point fields transfer

Once the setup is done, at the end of each time step, we have to inform XIOS about the current time step and query if we have to send any field. If so, we will send the requested data.

Code snippet 4.14 shows how the time step is updated and how to process just one field, in this case a 3D GMV field which is temperature. First of all, we update the time step calling the *xios_update_calendar* subroutine. IFS stores the current time step in variable *NSTEP*.

After that, we query if the field that we want to send is active in the current time step. In the example, the subroutine *xios_field_is_active* is used to query if we have to send the temperature field.

If the field is active, we have to build an XIOS-style array before sending data. If we remember what we explained in Section 3.1.3, IFS uses a data structure that is not usual, having the following dimensions: *NPROMA*, *NFLEVG*, *NFIELDS* and *NGPBLKS*. This does not match with the XIOS data arrays, where we need to send an array with the following dimensions: longitude, latitude and vertical levels. When it uses unstructured or reduced Gaussian grids, XIOS merges the first two dimensions, so it needs and array with these two dimensions: uni-dimensional 2D domain and vertical levels. Therefore, we have to re-organize data of each field before sending it, which means to perform the NPROMA blocks gather.

To perform the gather, in Code snippet 4.14 we iterate over blocks of size *NPROMA*, until all the *NGPTOT* grid-points of the subdomain are processed. Then, for each block we have to iterate all the block grid-points for all *NFLEVG* vertical levels. We do not need to iterate over the total number of fields because we are gathering data of just one field.

Note that the gather is at the subdomain level, i.e., intra-node shared-memory, so MPI communications are not needed at all.

Once data is re-organized following the XIOS-style array, we can send it calling the *xios_send_field* subroutine.

```
INTEGER(KIND=JPIM) :: jstglo, jlev, icend, ibl, i, j

! Update calendar of XIOS with the current time step
CALL xios_update_calendar(NSTEP)

! GMV
! Temperature
IF (xios_field_is_active("t", .TRUE.)) THEN
  ! Array accessing optimized for GMV, not for xios_gmv
  DO jstglo = 1, YDGEOMETRY%YRGEM%NGPTOT, YDGEOMETRY%YRDIM%NPROMA
    icend = MIN(YDGEOMETRY%YRDIM%NPROMA,YDGEOMETRY%YRGEM%NGPTOT-jstglo+1)
    ibl = (jstglo-1)/YDGEOMETRY%YRDIM%NPROMA + 1
    DO jlev = 1, YDGEOMETRY%YRDIMV%NFLEVG
      i = jstglo
      DO j = 1, icend
        xios_gmv(i,jlev) = YDFIELDS%YRGMV%GMV(j,jlev,YDFIELDS%YRGMV%YT0%MT,ibl)
        i = i + 1
      END DO
    END DO
  END DO
  CALL xios_send_field("t",xios_gmv)
END IF
```

Code snippet 4.14: NPROMA blocks gather and send of temperature field

One important thing to highlight are the XIOS thread-safe calls. Since IFS uses OpenMP for intra-node parallelization and XIOS does not support threading, we must ensure that only one thread at each MPI process calls XIOS subroutines. Although in previous code snippets we have not used any kind of mechanism, in our development we actually used the OpenMP *SINGLE* construct. Code snippet 4.15 illustrates the mechanism that we followed to have thread-safe calls.

```
!$OMP SINGLE
! One single thread region
!$OMP END SINGLE
```

Code snippet 4.15: Mechanism to have thread-safe calls

4.4 Environment setup

Besides the source code development, it is really important to adapt the scripts in order to run IFS and XIOS. In the following sections we will give an overview of the main changes that were needed.

4.4.1 XIOS compilation

One of the challenges was the XIOS compilation. Since we are working in a Cray machine and the IFS 43r3 version and its libraries that we are using are only compiled with Cray, we had to compile XIOS using Cray as well. The problem is that there are not many Cray machines in Europe, so XIOS is almost always compiled with Intel or GNU. This probably implies that XIOS is not usually tested with Cray and as a consequence, there are some issues.

First of all, the parallel compilation does not work in Cray compilers due to different bugs, something that has been reported to the XIOS developers to solve the problem. For example, a parallel compilation with Intel using eight threads, it takes about five minutes, while a sequential Cray compilation it takes about eight hours.

The bigger issue was related with the version of the Cray Developer Toolkit (CDT). By default, it loads the version 17.03, which successfully compiled XIOS, but it always failed when passing the XIOS tests. We tried different compilation flags, but this did not solve the issue. At the end, we discovered that using an older version of the CDT, the 16.04 version, we were able to pass the XIOS tests.

However, we also had other minor issues, such as that by default, Cray compiler generates mod files in capital letters, but they were not recognized when linking with XIOS. We solved this issue by using the *-ef* flag, which generates *.mod* files in lowercase letters.

Finally, we used a conservative compilation for XIOS, i.e., using the *-O1* optimization flag, because we read that for some old Cray compilers there was a bug using the *-O3* flag.

4.4.2 Including and linking XIOS

XIOS needs to be included and linked like all libraries. This is not difficult, but it was needed to find the exact place between the large amount of IFS scripts.

Code snippet 4.16 shows how the include can be done.

```
VPATH := . $(VPATH) /perm/ms/spesiccf/c3xy/xios2/inc
```

Code snippet 4.16: XIOS including

Code snippet 4.17 shows how the link can be done. Note that we also include the *libcray-c++-rts* library, since XIOS is programmed in C++.

```
-L/opt/cray/cce/8.4.6/CC/x86-64/lib/x86-64 -lcray-c++-rts \
-L/perm/ms/spesiccf/c3xy/xios2/lib -lxios
```

Code snippet 4.17: XIOS linking

4.4.3 Parallel netCDF and HDF5 versions

Although IFS output is in GRIB format, it is linked through environment modules with netCDF and HDF5, libraries which have been covered in Section 2.2. IFS uses sequential versions of them, but depending on the XIOS configuration, it needs the netCDF and HDF5 parallel versions.

We changed the two lines of Code snippet 4.18 by the two lines of Code snippet 4.19.

```
LOAD_MODULE cray-netcdf/${_netcdf_version}
LOAD_MODULE cray-hdf5/${_hdf5_version}
```

Code snippet 4.18: Old netCDF and HDF5 modules

```
LOAD_MODULE cray−netcdf−hdf5parallel/${_netcdf_version}
LOAD_MODULE cray−hdf5−parallel/${_hdf5_version}
```

Code snippet 4.19: New netCDF and HDF5 modules

4.4.4 Model script

Previously, in Figure 3.12 we have seen that an IFS experiment has several tasks in its workflow. One of them is called *model*, which is in charge of running the model. The associated script is obviously called *model* and it also calls other scripts, like the *run_parallel* one, which is in charge of executing the IFS binary. The *run_parallel* script modification is explained in next Section 4.4.5.

However, before calling *run_parallel* we have to prepare the environment. Code snippet 4.20 shows some preliminary steps. First of all, there are two variables that will be substituted by XCdp. Remember that we said that through XCdp we can set up at run-time variables which are between percentage symbols. Then, we have to copy the *iodef.xml* file and replace the string *XIOS_USING_SERVER* to indicate if we will run XIOS in attached or server mode. We also have to copy the XIOS binary if it is run in server mode.

There is another important setup regarding the Lustre striping. Lustre [51] is an open-source parallel file system that supports many requirements of leadership class HPC simulation environments. Lustre stores a file in one or more Object Storage Target (OST) devices. By default, it uses one OST, but it is possible to divide the file into chunks that are stored in different OSTs. This is known as striping, and it is used for performance purposes, especially for really big files. ECMWF recommends to stripe a file of size more than a few hundred GBs.

According to our *iodef.xml* file, XIOS will store netCDF files into the *xios_output* directory, but depending on the configuration, files can be really huge. To be cautious, we apply an striping of 4 OSTs for each one of the netCDF files that will be stored in *xios_output*.

If we had not changed the number of OSTs per file, we could affect the performance of the whole system, including the rest of users that are performing I/O at that moment. This is because we could fill up the OST in the parallel file system where the big netCDF file would be in. When such OST runs out of space, it also affects other user's jobs.

```
xios_using_server=%XIOS_USING_SERVER:false%
xios_path=%XIOS_PATH%

cp $TROOT/$EXPVER/ifs−repo/ifs−source/scripts/gen/iodef.xml .
sed −i −− ”s/XIOS_USING_SERVER/$xios_using_server/g” ./iodef.xml

mkdir xios_output
lfs setstripe −c 4 xios_output

if [[ $xios_using_server = true ]] ; then
  cp ${xios_path}/bin/xios_server.exe .
fi
```

Code snippet 4.20: XIOS and Lustre setup in model script

Once all setup is done, it is possible to call *run_parallel* to execute the IFS binary. Code snippet 4.21 shows a simplified version of how to use *run_parallel*. If we want to run IFS and XIOS in MPMD mode, we need to pass to *run_parallel* a file that contains in different lines each one of the binaries and their arguments. Otherwise, we call *run_parallel* as usual, which means that IFS will use XIOS in attached mode.

```
if [[ $xios_using_server = true ]] ; then
  touch executables_file
  chmod u+x executables_file
  echo ”ifsMASTER␣−f␣h%FCLENGTH%␣−t␣$timestep␣−v␣ecmwf␣−e␣$EXPVER” >>
      ↪  executables_file
  echo ”xios_server.exe” >> executables_file
  run_parallel −m 2 executables_file 2>ifs.err >ifs.log
else
  run_parallel ifsMASTER −f h%FCLENGTH% −t $timestep −v ecmwf −e $EXPVER
      ↪ 2>ifs.err >ifs.log
fi
```

Code snippet 4.21: MPMD mode preliminary setup in model script

4.4.5 Supporting MPMD mode

The last major change that we did was in the *run_parallel* script to run IFS and XIOS in Multiple-Program Multiple-Data (MPMD) mode. This script was already prepared to run binaries in MPMD mode, but it seems that was never tested and it contained some errors. We fixed them and extended the script to be fully integrated with XIOS. We will show some code snippets that are useful to understand what we did.

At the beginning, we set up through XCdp several variables that are used by XIOS (Code snippet 4.22). They are related to the affinity of XIOS: number of XIOS servers, number of servers per node, number of server per Non-Uniform Memory Access (NUMA) socket within a node, etc. We also specify that we do not want threads, since XIOS does not use OpenMP.

```
xios_using_server=%XIOS_USING_SERVER:-1%

if [[ $xios_using_server = true ]]; then
  submit_total_tasks=%NPES_FC:-1%:%NPES_XIOS:-1%
  submit_tasks_per_node=$submit_tasks_per_node:%XIOS_TASKS_PER_NODE:-1%
  submit_tasks_per_numa_node=$submit_tasks_per_numa_node:%
      ↪ XIOS_TASKS_PER_NUMA_NODE:-1%
  submit_cpus_per_compute_unit=$submit_cpus_per_compute_unit:1
  omp_num_threads=$omp_num_threads:1
elif [[ $xios_using_server = false ]]; then
  submit_total_tasks=%NPES_FC:-1%
fi
```

Code snippet 4.22: XIOS variables setup

Another important variable is *MPICH_DMAPP_APP_IS_WORLD*, which by default is set to 0. When we were testing the MPMD mode, we were having a deadlock in the execution without any kind of error. We spent a lot of time trying to understand why it was happening. We fixed the error by using the setup of Code snippet 4.23. Cray MPICH internally uses DMAPP, and if we set *MPICH_DMAPP_APP_IS_WORLD* to 1, it uses MPMD for MPI, but treats each DMAPP application as if it is a distinct job. MPI ranks are globally contiguous and global MPI communication is possible.

```
export MPICH_DMAPP_APP_IS_WORLD=1
```

Code snippet 4.23: Environment variable to run in MPMD mode

Finally, in Code snippet 4.24 we show the command line that will execute both IFS and XIOS. This command line is the result of executing the code that we implemented in *run_parallel*. It has several parameters that determine the geometry of the parallel job.

Binaries are separated by a colon and their arguments are passed after the name. For example, IFS has several arguments, while XIOS does not have anyone. Then, each binary has its own aprun parameters: the number of processes (-*n*), the number of processes per node (-*N*), the number of processes per NUMA socket (-*S*), the number of cores per each process (-*d*) and the number of logical threads, i.e., hyperthreading (-*j*). In addition, there are global parameters for both binaries, such as CPU binding (-*cc*), amount of memory per each process (-*m*) and the NUMA memory affinity (-*ss*).

Note that for IFS, we also set the number of OpenMP threads.

```
aprun -cc cpu -m8000h -n 702 -N 12 -S 6 -j 2 -d 6 -ss env
    ↪ OMP_NUM_THREADS=6 ./ifsMASTER -f h240 -t 600.0 -v ecmwf -e b0s8 :
    ↪ -cc cpu -n 10 -N 1 -j 1 -d 1 ./xios_server.exe
```

Code snippet 4.24: Example of an aprun command running IFS and XIOS

Chapter 5

Performance analysis and optimization

After the integration, the next step is to study the computational performance to detect potential bottlenecks and optimize them if possible.

The methodology that we used to analyze the performance and optimize the code is illustrated in Figure 5.1.

Firstly, we measure the execution time to use it as a reference. Then, using a profiling and/or tracing tool, we detect any possible bottleneck. Following up, we apply the proper optimization technique, and we also verify that the results are correct.

After that, we measure the execution time again, and depending on the time improvement of the optimization, we definitely integrate it or we discard it. If we discard the optimization, we try to find a better optimization, while if we integrate the optimization, we have to choose the next step: if the overall speedup including all the optimization is good enough, we end up the process; otherwise, we start a new iteration by performing a new performance analysis.

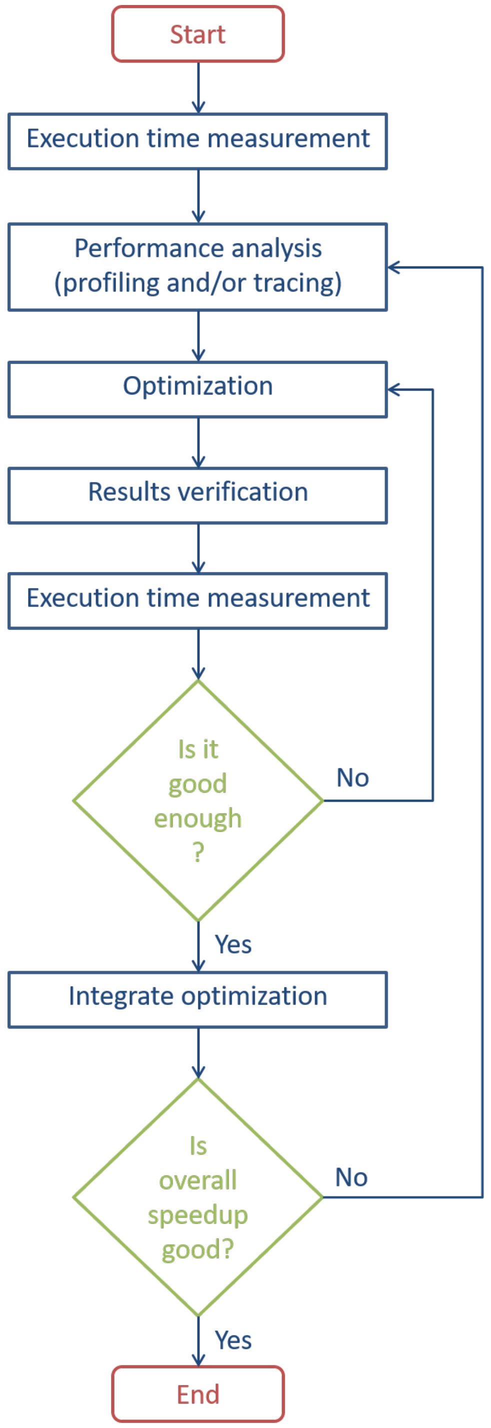


Figure 5.1: Performance analysis and optimization flowchart

Before explaining the optimization techniques that we performed, it is better to give some execution times about the IFS-XIOS integration. It is very important to keep in mind that all the performance numbers that appear in this master's thesis are using the IFS configuration explained in later Section 6.1. In summary, we use: the T1279 octahedral reduced Gaussian grid with 137 verticals levels; the forecast has a length of 10 days with a time step of 600 seconds and outputs at a frequency of 3 hours; IFS uses 702 MPI processes with 6 OpenMP threads per process, which gives a total of 4212 cores.

The total execution time of IFS and XIOS without any optimization takes 7773 seconds for a total of 1440 time steps, being this total execution time the summation of all time steps plus a period of initialization and finalization of the model. To have a reference and changing only the I/O scheme, the sequential output scheme takes 9054 seconds, the MF I/O server takes 7535 seconds, and the same execution disabling the output takes 7356 seconds.

If we focus on the execution of one time step, the execution times are of the order of less than 10 seconds. One of the IFS log files reports the execution time per each time step. Code snippet 5.1 illustrates a simplified output from time step 311 to 321. The 7th column indicates the individual execution time. Regular time steps last about 4.7 seconds, while one out of six time steps last about 6.7 seconds. The slower ones are in charge of also executing the radiation (it is not computed at each time step due to computational cost reasons).

```
12:24:21  0AAA00AAA  STEPO   311   27.918   27.918   4.680   164:25
12:24:25  0AAA00AAA  STEPO   312   28.082   28.082   4.702   164:54
12:24:32  0AAA00AAA  STEPO   313   40.167   40.167   6.734   165:34
12:24:37  0AAA00AAA  STEPO   314   27.714   27.714   4.646   166:01
12:24:41  0AAA00AAA  STEPO   315   28.526   28.526   4.788   166:30
12:24:46  0AAA00AAA  STEPO   316   28.086   28.086   4.714   166:58
12:24:51  0AAA00AAA  STEPO   317   27.938   27.938   4.679   167:26
12:24:55  0AAA00AAA  STEPO   318   27.370   27.370   4.592   167:53
12:25:02  0AAA00AAA  STEPO   319   39.994   39.994   6.708   168:33
12:25:07  0AAA00AAA  STEPO   320   28.826   28.826   4.826   169:02
12:25:12  0AAA00AAA  STEPO   321   28.034   28.034   4.701   169:30
```

Code snippet 5.1: IFS log file with non-output time steps

Furthermore, in Code snippet 5.2 we provide specific execution times of our developed code and included in the IFS execution. These times are obtained using the GSTATS profiling tool, briefly explained in following Section 5.1. There are three profiled subroutines, SUXIOS_INI, SUXIOS_CTXT and CXIOS, which corresponds to our public *suxios_ini*, *suxios_ctxt* and *ifs_xios_send_fields* subroutines respectively. Note that *suxios_fin* does not appear, although we tried to profile it. It could be related to the fact that it finalizes MPI and as a consequence GSTATS is not able to properly finish the profiling.

The time that these three subroutines take is not considerable: initialization takes 0.5 seconds, context setup takes 9.5 seconds, and the NPROMA blocks gather and send of fields take 84.8 seconds. In relative terms, they sum 1.24% of the total execution time (last column).

```
ROUTINE      CALLS SUM(s)  AVE(ms) STDDEV(ms)  MAX(ms) SUMB(s) FRAC(%)
SUXIOS_INI       1    0.5    526.5        0.0    526.5     0.0    0.01
SUXIOS_CTXT      1    9.5   9473.2        0.0   9473.2     0.0    0.12
CXIOS         1441   84.8     58.9      240.3   1158.2    15.2    1.11
```

Code snippet 5.2: Profiling analysis with GSTATS of the non-optimized IFS-XIOS integration

In Section 5.1, we will briefly explain the tools that we used to study the computational performance.

After that, we will explain the different optimization techniques that we used (5.2, 5.3, 5.4) and the reason why we used them.

Finally, in Section 5.5 we explain three additional optimization techniques that we did not implement due to the following reasons:

- It was not possible to use it.
- It was implemented by the compiler using the proper compilation flag.
- It is enabled by default, but it was interesting to see what happens if not used.

5.1 Tools

We basically used two kind of tools: profiling and tracing. They are explained in the following two sections.

5.1.1 Profiling

Profiling is the analysis of the application's behavior using information gathered as the program executes in order to determine which parts need to be optimized. To achieve this goal, a profiling tool takes into account aspects such as execution time per subroutine, execution time per line of code (this implies to know which instructions have more cost), which are the dependencies between functions (it builds a tree with calls), the number of times that each function has been called, etc. We can get all this data without looking at the source code or even without having to instrument it.

One popular tool is gprof, which offers simplicity. However, since we are not interested in profiling the entire code, we used the GSTATS timers. IFS has its own low overhead timers called GSTATS. They are really simple to use: declare an identifier of the region that you want to instrument, and add two calls, at the beginning and at the end of the region.

At the end of the execution, GSTATS generates a profile for each instrumented region: number of executed times, the average, the aggregated, the maximum, standard deviation, etc.

5.1.2 Tracing

Tracing is the process of recording event-based performance data along the execution of a program. Using a viewer we can see the behavior of the application in our machine, focusing on hardware counters, communication patterns or memory hierarchy. The tools used to trace the model were Extrae and Paraver, which are open-source and developed by the BSC Performance Tools group [52]:

- Extrae is a package used to instrument the code automatically and/or manually through its API. It generates Paraver trace-files with hardware counters, MPI messages and other information for a post-mortem analysis. It can be downloaded and installed in any HPC facility.
- Paraver is a trace browser that can show a global visual qualitative perspective of the application behavior for later focus on the bottlenecks with a detailed quantitative analysis. The tool allows to create views with any parameter recorded and points to a region of a code, making process of modification easier.

Figure 5.2 shows an example of a trace, which in this case is a regular time step from an IFS-XIOS execution. A trace has the timeline on the x axis and the MPI processes on the y axis.

Along the timeline, some (or many) events happen, which can be related to MPI calls, cache misses, Million Instructions Per Second (MIPS) and many other performance metrics. These events can be chosen with Paraver, but it depends on the configuration of Extrae, i.e., we have to set up in an XML file the events that we want to generate.

The trace of the Figure 5.2 shows MPI call events, where each color represents an MPI function. Figure 5.3 illustrates in detail each one of the possible colors, i.e., possible MPI functions. Note that the first color, light blue, it is actually not an MPI function, but it represents computation (outside MPI).

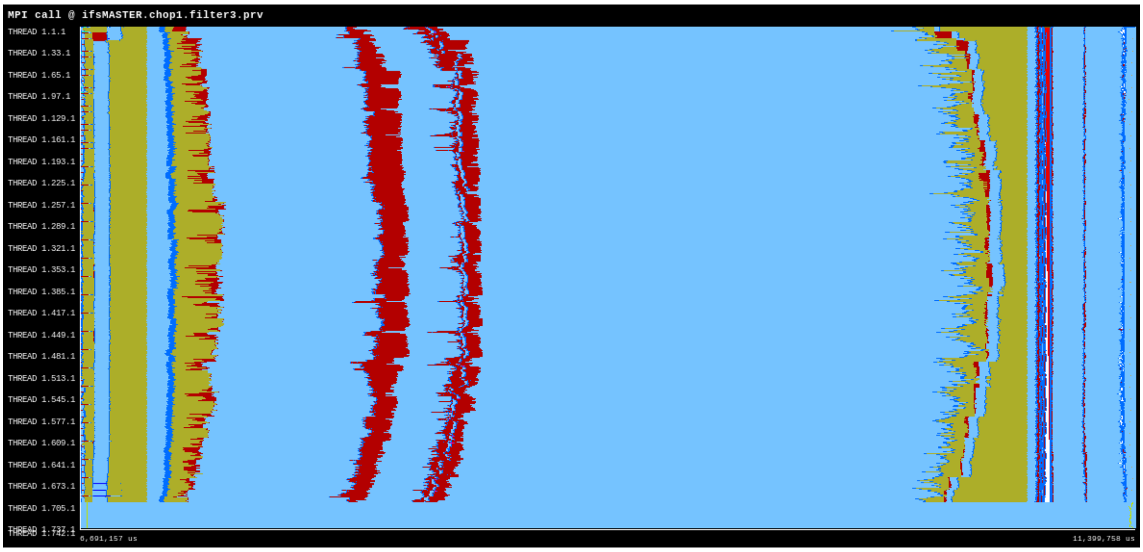
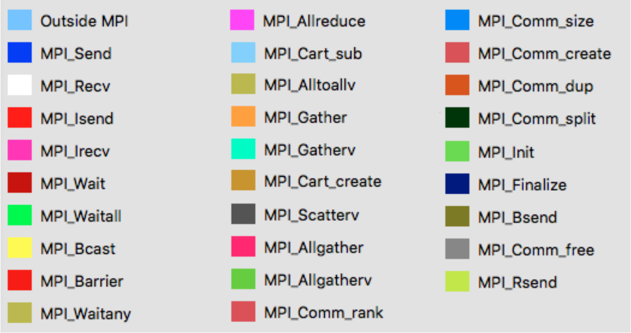


Figure 5.2: Example of a trace with MPI call events. It shows a complete regular time step of an IFS-XIOS execution. Timeline is on the x axis and MPI processes on the y axis.



🟦 Outside MPI
🟦 MPI_Send
⬜ MPI_Recv
🟥 MPI_Isend
🟪 MPI_Irecv
🟥 MPI_Wait
🟩 MPI_Waitall
🟨 MPI_Bcast
🟥 MPI_Barrier
🟨 MPI_Waitany
🟪 MPI_Allreduce
🟦 MPI_Cart_sub
🟨 MPI_Alltoallv
🟧 MPI_Gather
🟩 MPI_Gatherv
🟧 MPI_Cart_create
⬛ MPI_Scatterv
🟥 MPI_Allgather
🟩 MPI_Allgatherv
🟥 MPI_Comm_rank
🟦 MPI_Comm_size
🟥 MPI_Comm_create
🟧 MPI_Comm_dup
🟩 MPI_Comm_split
🟩 MPI_Init
🟦 MPI_Finalize
🟫 MPI_Bsend
⬛ MPI_Comm_free
🟨 MPI_Rsend

Figure 5.3: Legend with the colors representation of the MPI calls used by Paraver

Although other metrics have been evaluated too, the figures showed in this document are traces related to MPI call events because it is enough for the performance analyses and optimization techniques that we used.

5.2 Threading with OpenMP

One of the optimization was to parallelize the NPROMA blocks gather using OpenMP threads. As we explained in Section 4.3, the gather was necessary because the requested XIOS arrays have different dimensions with regard the IFS ones, so it was necessary to build those new arrays.

According to the profiling analysis with GSTATS timers, the gather does not represent too much time. Code snippet 5.3 shows in detail the gather of the ciwc field. It takes about 5.8 ms, so the optimization would not have a large impact in the execution time for these tests. However, while the OpenMP master thread is performing the gather, the rest of threads are idle, so this operation is actually inefficient. In addition, it could become a bottleneck for future experiments demanding more computational power. Thus, it is better to use all the threads.

```
ROUTINE      CALLS SUM(s) AVE(ms) STDDEV(ms) MAX(ms) SUMB(s) FRAC(%)
ciwc_GATHER   80    0.5     5.8      0.9        6.5     0.0     0.01
```

Code snippet 5.3: Profiling analysis with GSTATS of a field gather

We followed the parallelization strategy of Code snippet 5.4. It contains the gather and send of two fields, specific humidity and temperature, in order to understand how we try to overlap the send of one field with the gather of the next one. This is extended to many other fields using the same strategy, but showing two fields is enough for illustrative purposes.

Note that now, the condition of each IF is a logical variable, since we have performed the call to the *xios_field_is_active* subroutine and stored the condition before the OpenMP parallel region. This is because XIOS subroutines are not thread-safe.

We start by parallelizing the gather using an *OMP DO* directive, where the granularity of chunks is an NPROMA block. This strategy is beneficial for OpenMP, since IFS data arrays were designed to that end. We explained the benefits of this design in Section 3.1.3.

After that, once we have an XIOS-style array, we can send it to XIOS. Only one of the threads will perform the send, while the others will perform the gather of the next field. This overlapping is possible because we use the *NOWAIT* clause in the *SINGLE* construct.

```
!$OMP PARALLEL PRIVATE(jstglo,icend,ibl,jlev)

! GFL - Specific humidity
IF (q) THEN
  !$OMP DO SCHEDULE(DYNAMIC)
  DO jstglo = 1, YDGEOMETRY%YRGEM%NGPTOT, YDGEOMETRY%YRDIM%NPROMA
    icend = MIN(YDGEOMETRY%YRDIM%NPROMA,YDGEOMETRY%YRGEM%NGPTOT-jstglo+1)
    ibl = (jstglo-1)/YDGEOMETRY%YRDIM%NPROMA + 1
    DO jlev = 1, YDGEOMETRY%YRDIMV%NFLEVG
      xios_gfl(jstglo:jstglo+icend-1,jlev) = YDFIELDS%YRGFL%GFL(1:icend,jlev,YGFL%YQ%MP,ibl)
    END DO
  END DO
  !$OMP END DO
  !$OMP SINGLE
  CALL xios_send_field(”q”,xios_gfl)
  !$OMP END SINGLE NOWAIT
END IF

! GMV - Temperature
IF (t) THEN
  !$OMP DO SCHEDULE(DYNAMIC)
  DO jstglo = 1, YDGEOMETRY%YRGEM%NGPTOT, YDGEOMETRY%YRDIM%NPROMA
    icend = MIN(YDGEOMETRY%YRDIM%NPROMA,YDGEOMETRY%YRGEM%NGPTOT-jstglo+1)
    ibl = (jstglo-1)/YDGEOMETRY%YRDIM%NPROMA + 1
    DO jlev = 1, YDGEOMETRY%YRDIMV%NFLEVG
      xios_gmv(jstglo:jstglo+icend-1,jlev) = YDFIELDS%YRGMV%GMV(1:icend,jlev,YDFIELDS%YRGMV%YT0%MT,ibl)
    END DO
  END DO
  !$OMP END DO
  !$OMP SINGLE
  CALL xios_send_field(”t”,xios_gmv)
  !$OMP END SINGLE NOWAIT
END IF
! .
! . Gather and send many other fields
! .
!$OMP END PARALLEL
```

Code snippet 5.4: Parallelization strategy of the NPROMA blocks gather

This optimization slightly improves the performance as it can be seen in results

Section 6.3 (see Figures 6.2 and 6.3). The execution time is reduced from 7773 second to 7705 seconds. It is important to highlight that depending on the distribution of cores between MPI and OpenMP, we could exploit much more this optimization. For example, if we have a configuration where we use more cores for OpenMP threads and less cores for MPI processes, we would have less subdomains, but much bigger. Thus, the gather with just one thread would be much more significant, because subdomains would have much more data to be gathered.

Code snippet 5.5 shows the profiling analysis of the field ciwc with GSTATS timers using the optimization. The average time is reduced from 5.8 ms to 1.2 ms. Note that this is also extensible to all other fields that we need to output.

```
ROUTINE       CALLS SUM(s) AVE(ms) STDDEV(ms) MAX(ms) SUMB(s) FRAC(%)
ciwc_GATHER    80     0.1     1.2        0.2       2.3     0.0     0.00
```

Code snippet 5.5: Profiling analysis with GSTATS of an optimized gather

5.3 Optimized compilation of XIOS

The following optimization was the compilation of XIOS. The compilation of XIOS using the *-O1* flag should be improved in order to take advantage of the optimization introduced by the compiler automatically, where the use of *-O2* or *-O3* are the options recommended by default. However, as we explained in Section 4.4.1 we had a lot of issues to use them.

We were analyzing the performance reports of XIOS and we realized that it was spending a considerably amount of time for just writing data, especially on client side. Code snippet 5.6 shows that IFS spent about 132 seconds in executing XIOS client code, while Code snippet 5.7 shows that XIOS servers spent about 3196 seconds in processing events.

We know that enabling post-processing leads to an increase of the XIOS execution time. Depending on the post-processing filter used, this post-processing is performed on client side or on server side. For example, horizontal domain interpolations are performed on client side, while data compression is performed on server side. However, since in our tests there is not post-processing at all, the execution time for XIOS seems to be too much high. For this reason, we thought that it would be interesting to try to improve the XIOS compilation.

```
-> report :  Performance report : Whole time from XIOS init and
   ↪ finalize: 7681.68 s
-> report :  Performance report : total time spent for XIOS :
   ↪ 132.715 s
-> report :  Performance report : time spent for waiting free
   ↪ buffer : 3.80519 s
-> report :  Performance report : Ratio : 0.0495359 %
```

Code snippet 5.6: XIOS report on client side

```
-> report : Performance report : Time spent for XIOS : 7681.68
-> report : Performance report : Time spent in processing events :
   ↪  3196.4
-> report : Performance report : Ratio : 41.6107%
```

Code snippet 5.7: XIOS report on server side

As we mentioned in Section 4.4.1, we read that for some old Cray compilers there was a bug using the -*O3* flag. However, this bug may have been solved in new versions, so once updated, we tried to use -*O3*. Interestingly, XIOS compiled and tests successfully passed.

When we tried this new optimized version with IFS, we experienced an improvement in the execution time as it can be seen in results Section 6.3 (see Figures 6.2 and 6.3). The execution time is reduced from 7705 seconds to 7629 seconds. Code snippet 5.8 shows that the execution time on client side is reduced from 132 seconds to 40 seconds. On server side (Code snippet 5.9), same effect occurs and execution time is reduced from 3196 seconds to 1382 seconds.

```
-> report :  Performance report : Whole time from XIOS init and
    ↪ finalize: 7562.36 s
-> report :  Performance report : total time spent for XIOS :
    ↪ 40.3018 s
-> report :  Performance report : time spent for waiting free
    ↪ buffer : 0.463693 s
-> report :  Performance report : Ratio : 0.00613159 %
```

Code snippet 5.8: XIOS report on client side once optimized

```
-> report : Performance report : Time spent for XIOS : 7562.37
-> report : Performance report : Time spent in processing events :
    ↪  1382.16
-> report : Performance report : Ratio : 18.2768%
```

Code snippet 5.9: XIOS report on server side once optimized

This optimization will be especially useful when post-processing is used, since it typically requires a lot of computation.

5.4 Overlapping computation and communication

The last optimization that we implemented was an overlapping between the computation of IFS and the MPI communications to XIOS.

We mentioned that non-output time steps have execution times of about 4.7 seconds (regular time step) and about 6.7 seconds (time step with radiation) as showed in Code snippet 5.1. However, we realized that in an output time step, there is a slight increase in the execution time of the three following time steps. Code snippet 5.10 illustrates this effect: time step 324 performs output through XIOS, however, the time steps which experience an increase are 325, 326 and 327. This increase actually has many variability among similar output time steps, which means that sometimes is negligible and sometimes is really noticeable.

Nevertheless, in general we can observe this pattern in the following three time steps of an output time step.

```
12:24:55 0AAA00AAA STEPO 318 27.370 27.370 4.592 167:53
12:25:02 0AAA00AAA STEPO 319 39.994 39.994 6.708 168:33
12:25:07 0AAA00AAA STEPO 320 28.826 28.826 4.826 169:02
12:25:12 0AAA00AAA STEPO 321 28.034 28.034 4.701 169:30
12:25:16 0AAA00AAA STEPO 322 27.770 27.770 4.655 169:58
12:25:21 0AAA00AAA STEPO 323 27.690 27.690 4.654 170:26
12:25:26 0AAA00AAA STEPO 324 27.854 27.854 4.679 170:53
12:25:33 0AAA00AAA STEPO 325 42.771 42.771 7.158 171:36
12:25:38 0AAA00AAA STEPO 326 30.114 30.114 5.044 172:06
12:25:43 0AAA00AAA STEPO 327 30.870 30.870 5.181 172:37
12:25:48 0AAA00AAA STEPO 328 27.874 27.874 4.682 173:05
```

Code snippet 5.10: IFS log file with one output time step

In order to understand what was happening, we traced the execution with Extrae. Figure 5.4 illustrates three time steps, where the first one is performing output at the end. The major part of processes of the trace are the IFS ones, while some processes on the bottom are the XIOS ones.

In this trace, IFS is using the output scheme that we explained in Chapter 4 and showed in Figure 4.2. This scheme sequentially executes at the end of an output time step these three steps: update calendar, NPROMA blocks gather and send fields. If it is not an output time step, it only performs the update calendar step.

The trace reveals an interesting point: at the beginning of the next time step after an output time step (second time step in Figure 5.4), there is an increase in the communications among IFS processes. They are basically performing an *MPI_Waitany* and an *MPI_Alltoallv*, but they seem to be delayed according to the time step where this should be done.

The problem is related to the moment where some fields are sent to XIOS. Although they are sent using asynchronous communications to overlap with computation, at the beginning of the time step IFS is not performing computation, but synchronous communication. This leads IFS to wait for XIOS asynchronous communications and its own synchronous communications. IFS communications have to wait because nodes and network are already occupied by XIOS communications.

In addition, at the beginning of the third time step there is also a delay in the *MPI_Waitany* and *MPI_Alltoallv* execution, though at the end of the second time step no field is sent at all. So we concluded that when executing the update calendar after an output time step, XIOS performs costly communications that occupy the network resources which cannot be used by IFS at that moment.

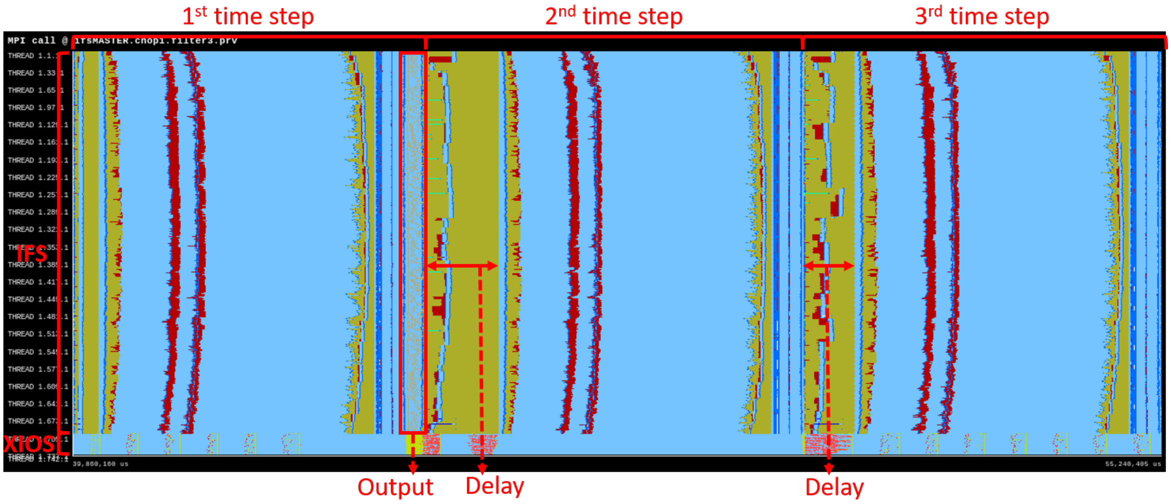


Figure 5.4: Trace with three IFS time steps

Then, knowing that update calendar and send fields steps can potentially collide with IFS communications, we considered to re-order them along the execution time to truly overlap XIOS communication with IFS computation. According to Figure 5.4, at each time step there is a huge area (light blue) where IFS is only performing computation. This region is in charge of executing the psychics of IFS. Ideally, it is a perfect place to perform the XIOS communications.

Our new scheme with re-ordered steps is illustrated in Figure 5.5: the update calendar step (C_x) is prior called at the beginning of the physics in the same time step; the NPROMA blocks gather (G_x) is the only step that remains at the end of the same time step; and the send fields step (S_x) is delayed to the next time step at the beginning of the physics, called before the update calendar of the next time step (C_{x+1}).

Once all time steps are performed, we may need to execute the send fields step (S_n) of the last time step, something that it is negligible for the final execution time.

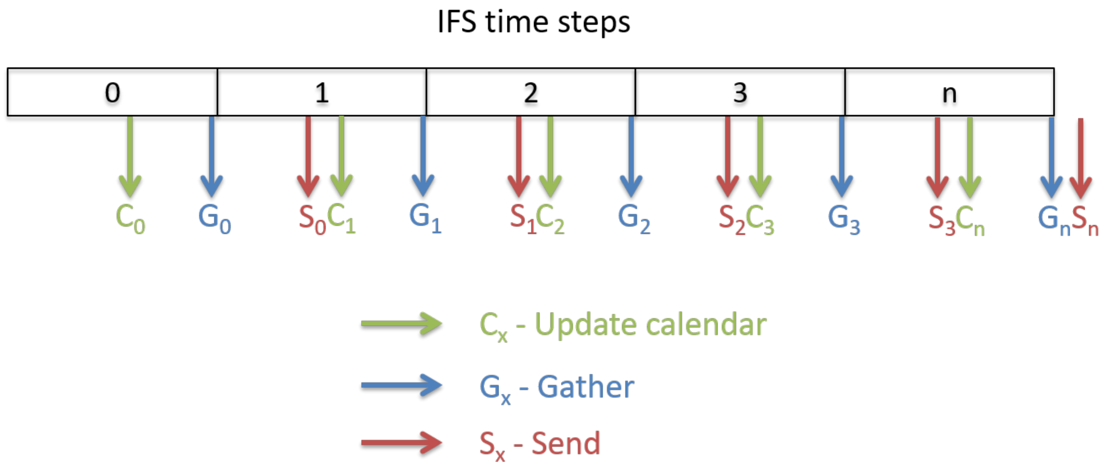


Figure 5.5: Optimized output scheme developed for IFS. The three needed steps to output data are split: the update calendar is performed in the middle of the time step; the NPROMA blocks gather is maintained at the end of the time step; and the send fields is delayed to the next time step

The approach to implement the first scheme was to use just one subroutine called *ifs_xios_send_fields*, which contained the three essential steps: update calendar, NPROMA blocks gather and send fields.

The implementation of the new scheme is not difficult: we split the three steps into three different subroutines:

- *ifs_xios_calendar*: it contains the call to the *xios_update_calendar* subroutine and all the calls to *xios_field_is_active* for each one of the fields to individually store in logical variables if they need to be sent or not.
- *ifs_xios_gather_fields*: depending on the values of the logical variables, this subroutine will perform the NPROMA blocks gather of the fields that need

to be sent.

- *ifs_xios_send_fields*: depending on the values of the logical variables, this subroutine will send the fields which are active.

These three public subroutines are placed in the proper places of the IFS code so that the new scheme is fully implemented.

According to the results that are shown in Section 6.3 (see Figures 6.2 and 6.3), it is the optimization that gives more improvement in the total execution time: it is reduced from 7629 seconds of the previous optimization to 7507 seconds.

This new scheme improves the execution time of the three time steps that follow an output time step. Code snippet 5.11 shows this behaviour: time step 324 performs output, but it almost does not affect the execution time of time steps 325, 326 and 327.

```
12:27:45 0AAA00AAA STEPO   318   26.926   26.926   4.514   162:23
12:27:52 0AAA00AAA STEPO   319   38.414   38.414   6.441   163:01
12:27:56 0AAA00AAA STEPO   320   27.054   27.054   4.535   163:28
12:28:01 0AAA00AAA STEPO   321   27.030   27.030   4.534   163:55
12:28:05 0AAA00AAA STEPO   322   26.882   26.882   4.502   164:22
12:28:10 0AAA00AAA STEPO   323   27.394   27.394   4.607   164:50
12:28:15 0AAA00AAA STEPO   324   27.142   27.142   4.549   165:17
12:28:21 0AAA00AAA STEPO   325   39.310   39.310   6.579   165:56
12:28:26 0AAA00AAA STEPO   326   28.318   28.318   4.755   166:24
12:28:31 0AAA00AAA STEPO   327   28.686   28.686   4.813   166:53
12:28:35 0AAA00AAA STEPO   328   26.990   26.990   4.527   167:20
```

Code snippet 5.11: IFS log file with one output time step once optimized

If we trace this new scheme, Figure 5.6 indicates that there is no delay in the *MPI_Waitany* and *MPI_Alltoallv* operations at the beginning of an IFS time step. It is visible how IFS executes the NPROMA blocks gather at the end of the first time step and the call to send fields at the beginning of the physics in the second time step.

However, it is also important to remark that now there is a delay at the end of time steps two and three, but it is less significant than the delay of the first scheme.

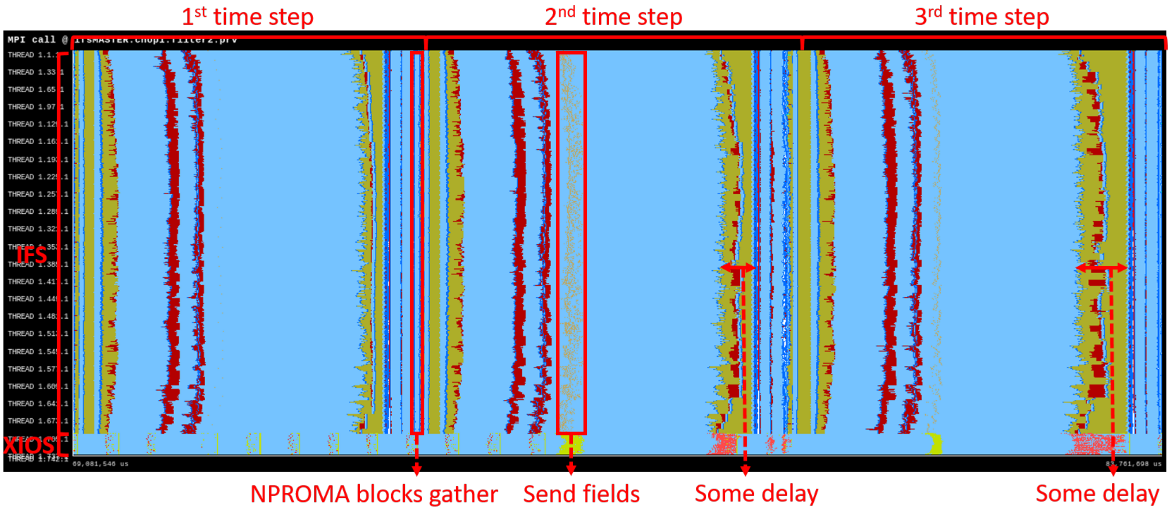


Figure 5.6: Trace with three IFS time steps once optimized

Figure 5.7 shows a comparison of the two output schemes. Traces of Figures 5.4 and 5.6 are compared using the same elapsed time. It is visible how the three time steps of the trace that uses the optimized output scheme (below) finish earlier than the other three time steps, the ones that use the non-optimized scheme.

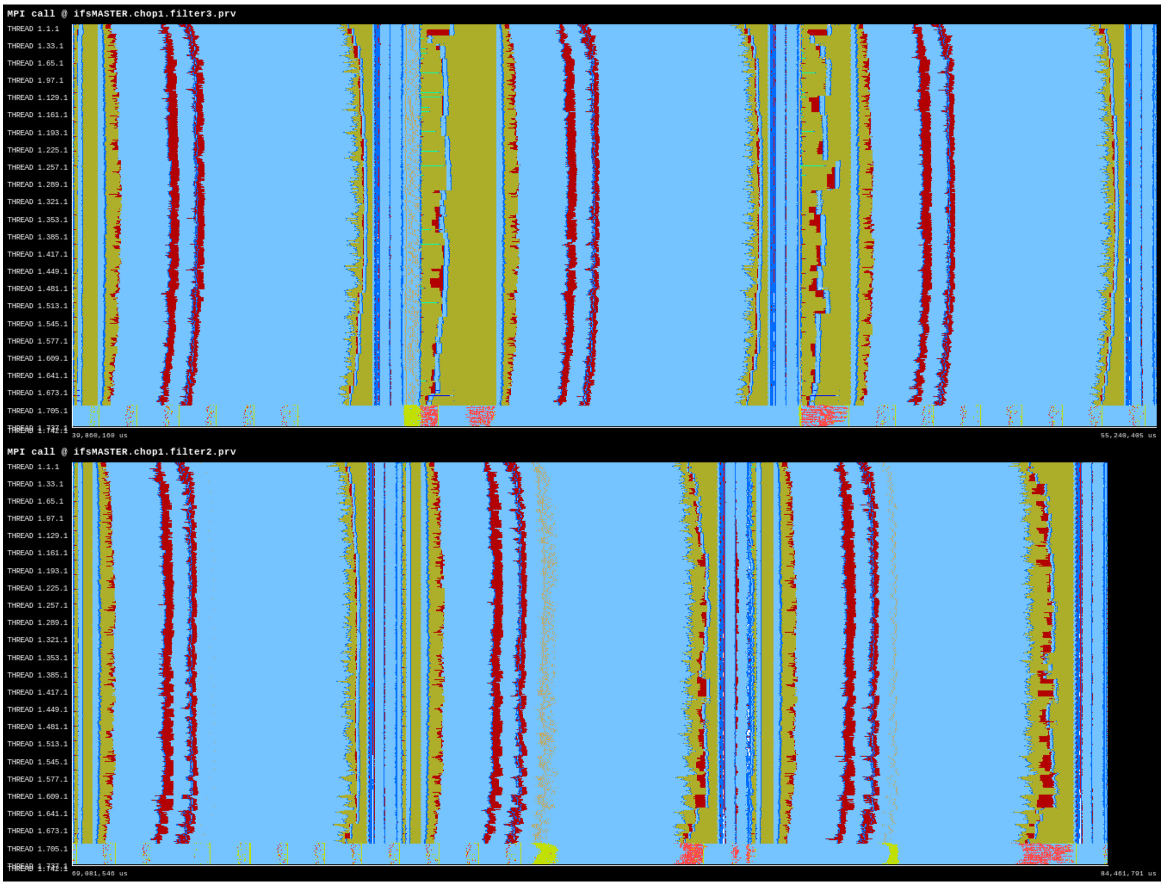


Figure 5.7: Trace comparison using the same elapsed time between the non-optimized overlapping (above) and the optimized overlapping (below). The three time steps of the optimized trace finish earlier than the three ones of the non-optimized trace

5.5 Explored options

We also considered three additional optimization techniques, but as we explained at the beginning of this chapter, we did not implement them.

5.5.1 Vectorization with SIMD instructions

One of the key points in current x86 machines is the use of vector instructions, also known as Single Instruction, Multiple Data (SIMD) instructions. We thought that they would be really important for performing the NPROMA blocks gather, in order to speed up the whole process.

However, before any kind of implementation, we checked the assembly code to know if compiler was using them. Effectively, Cray compiler was vectorizing the gather, as it is shown in Code snippet 5.12. At the beginning, it uses a series of *vmovaps* instructions to move data from memory to vector registers (*xmm*). Then, this data is moved from vector registers to the stack (*rsp*). Finally, once all data is in the stack, Cray compiler calls the *xios_send_field_r8_2d* subroutine using the *callq* instruction.

```
21482846: c6 84 24 e0 00 00 00 movb $0x74,0xe0(%rsp)
2148284d: 74
2148284e: c5 f8 28 05 6a 55 89 vmovaps 0x2489556a(%rip),%xmm0 #45d17dc0 <$data_init$yomxios_+0x440>
21482855: 24
21482856: c5 f8 28 0d 52 55 89 vmovaps 0x24895552(%rip),%xmm1 #45d17db0 <$data_init$yomxios_+0x430>
2148285d: 24
2148285e: c5 f8 28 15 3a 55 89 vmovaps 0x2489553a(%rip),%xmm2 #45d17da0 <$data_init$yomxios_+0x420>
21482865: 24
21482866: c5 f8 28 1d 22 55 89 vmovaps 0x24895522(%rip),%xmm3 #45d17d90 <$data_init$yomxios_+0x410>
2148286d: 24
2148286e: c5 f8 28 25 0a 55 89 vmovaps 0x2489550a(%rip),%xmm4 #45d17d80 <$data_init$yomxios_+0x400>
21482875: 24
21482876: c5 f8 28 2d f2 54 89 vmovaps 0x248954f2(%rip),%xmm5 #45d17d70 <$data_init$yomxios_+0x3f0>
2148287d: 24
2148287e: c5 f8 29 84 24 50 01 vmovaps %xmm0,0x150(%rsp)
21482885: 00 00
21482887: c5 f8 29 8c 24 40 01 vmovaps %xmm1,0x140(%rsp)
2148288e: 00 00
21482890: c5 f8 29 94 24 30 01 vmovaps %xmm2,0x130(%rsp)
21482897: 00 00
21482899: c5 f8 29 9c 24 20 01 vmovaps %xmm3,0x120(%rsp)
214828a0: 00 00
214828a2: c5 f8 29 a4 24 10 01 vmovaps %xmm4,0x110(%rsp)
214828a9: 00 00
214828ab: c5 f8 29 ac 24 00 01 vmovaps %xmm5,0x100(%rsp)
214828b2: 00 00
214828b4: 48 89 e0 mov %rsp,%rax
214828b7: 48 05 e0 00 00 00 add $0xe0,%rax
214828bd: 48 89 84 24 d0 00 00 mov %rax,0xd0(%rsp)
214828c4: 00
214828c5: 48 c7 84 24 d8 00 00 movq $0x1,0xd8(%rsp)
214828cc: 00 01 00 00 00
214828d1: 48 8b bc 24 d0 00 00 mov 0xd0(%rsp),%rdi
214828d8: 00
214828d9: 48 89 e6 mov %rsp,%rsi
214828dc: 48 81 c6 00 01 00 00 add $0x100,%rsi
214828e3: ba 01 00 00 00 mov $0x1,%edx
214828e8: e8 53 2a 4f 03 callq 24975340 <xios_send_field_r8_2d$idata_>
```

Code snippet 5.12: Vectorization of the NPROMA blocks gather

5.5.2 Memory affinity

The majority of HPC machines use Non-Uniform Memory Access (NUMA) nodes. This type of computers are characterized by having more than one processor, which share memory without uniform access time. This means that the whole memory is distributed across the processors, where each processor locally owns a part of the memory. However, processors are able to access other processors' local memory, but more slowly.

Cray XC40 nodes have two NUMA sockets (or processors), each one with 18 cores and its own local memory. When running a parallel job with aprun, it is possible to specify if sockets are allowed to allocate memory from each other or not. The *-ss* flag indicates that a core or a thread can allocate only the memory local to its assigned NUMA socket.

In theory, using this flag it should give more computational performance, since slow crossed memory allocations between sockets are not allowed. This type of tests are known as memory affinity. Our idea was to test and potentially use the *-ss* optimal option. However, by default IFS uses the *-ss* flag to run, so this optimization was already in use.

Nevertheless, we thought that it would be useful to see if *-ss* was certainly optimizing the run of IFS, so we repeated the experiment removing this flag. We observed that the total execution time dramatically increased by 20%. Thus, we proved that the *-ss* flag to achieve NUMA affinity is mandatory for this kind of applications.

5.5.3 Derived MPI Datatypes

We have seen that IFS-style arrays and XIOS-style arrays are different, so we needed to perform an NPROMA blocks gather before sending fields to XIOS. We occurred that using derived MPI Datatypes could be possible to avoid the gather.

The idea of using derived MPI Datatypes [53] is to describe the layout of data structures in memory. This means that they are used to describe how data is stored: stride between elements, how many contiguous elements per block and the block count. It is possible to describe vectors, structs, etc. Thus, the NPROMA blocks gather would not be needed.

Nevertheless, we were not able to use this technique because we do not directly work with MPI subroutines when we send data to XIOS. The only way is by calling the *xios_send_field* subroutine, which is from the XIOS API. Although this cannot be done, it could be suggested to the XIOS developers to add it as a new feature in the future.

Chapter 6

Evaluation

In this chapter we will evaluate the results of the development and the optimization techniques. In Section 6.1 we explain the IFS configuration that we used to perform the benchmark tests. Then, in Section 6.2 we show the two metrics that are used to measure the computational performance. Finally, in Section 6.3 we explain all the different tests that we performed.

6.1 IFS configuration

It is really important to choose a proper IFS configuration to benchmark our development, since we want to be as close as possible to real experiments. Thus, we could have an exact idea of the benefits of using XIOS.

We will compare and discuss in Section 6.3 the different output schemes: sequential ouput, MF I/O server, XIOS server (different optimized versions) and also no output, to know which is the maximum achievable speedup.

Table 6.1 shows a summary of the basic IFS parameters that are used in the configuration to compare all the output schemes. We use the octahedral reduced Gaussian grid in high resolution (see Section 3.1.2), which is the same that ECMWF uses for its operational forecast. We also use 702 MPI processes with 6 OpenMP threads for each one, which gives a total of 4212 processing elements. This distribution of processing elements is recommended by ECMWF, because it offers a good balance between performance and efficiency. Moreover, we do not change neither the number of MPI processes nor the number of OpenMP threads because it would be a scalability study, but it is not part of this work. In order to evaluate the computational performance of our development, it is preferable to focus on the number of processes of XIOS and the output size.

We almost used all the default values that prepIFS sets up when a new experiment is created. We just changed a few parameters, especially the ones related to performance and output.

Basic IFS parameters	
Grid type	Cubic octahedral reduced Gaussian
Horizontal resolution	1279
Vertical resolution	137
Forecast length	10 days
Time step	600 seconds
Compile environment	cdt/16.04
NPROMA size	16
Output frequency	3 hours
2D prognostic fields	1
3D prognositc fields	12
Diagnostics	No
Forecast tasks (MPI processes)	702
OpenMP threads per forecast task	6
Hyperthreading	Yes

Table 6.1: Summary of the IFS configuration

The parameters that change between output schemes are the following:

- Sequential output: we do not need additional processes for I/O servers. We do not want to write data using the Fields Data Base (FDB).
- MF I/O server: we have to specify the number of dedicated processes for I/O, which the recommended amount by ECMWF is 18. We do want to

write data using FDB.

- XIOS: we have to specify the number of dedicated processes for I/O, which the recommended amount is analyzed in later Section 6.3.1. We cannot use FDB, so we disable it.

The output is characterized by being quite large in size: we are outputting data at a frequency of 3 hours, while in the operational forecast is at 6 hours. Thus, we are able to stress the schemes and prove that XIOS works well. In any case, there are real configurations where a frequency of 3 hours could be used.

We output the following 12 3D prognostic fields:

- U component of wind
- V component of wind
- Temperature
- Divergence
- Vorticity
- Specific humidity
- Cloud fraction
- Ozone
- Specific rain water content (rain)
- Specific snow water content (snow)
- Specific cloud liquid water content (liquid water)
- Specific cloud ice water content (ice water)

In addition, we also output this 2D prognostic field:

- Surface pressure

It is important to be aware of the output size of this particular configuration because it is considerably large and it is directly related to the time used for the output process. However, it depends on the data format:

- GRIB format: files' size is about 770 GB.
- NetCDF format: files' size is about 3.2 TB.

This difference in size between both formats is due to compression. While GRIB files are compressed, netCDF files do not have any kind of compression. However, we could enable the netCDF files compression in the XIOS configuration, which would not affect the overall performance because it is done on server side. But, we do not do that because we are only evaluating I/O aspects.

Finally, in order to evaluate the results, for each case we make the average of three executions. It is true that when measuring I/O the sampling should be bigger, about 10 runs, because the variability considerably grows. Nevertheless, we do not do it because of two reasons. The first one is because the I/O variability is not reflected in the total execution time, since I/O servers have enough time to "buffer" it. The second reason is because we are running huge experiments, so we cannot afford wasting such amount of CPU hours.

6.2 Metrics

In order to compare and evaluate the two IFS output schemes against the different optimization versions of our development, we will use two basic metrics in computer sciences: execution time and speedup.

We measured the execution time as the elapsed time from the beginning of the IFS-XIOS execution to completion.

The speedup measures the relative performance between two systems solving the same problem. Systems can be of any type: hardware systems, software systems, etc. In our case, we compare different output schemes for IFS, which always uses the configuration explained in previous Section 6.1.

We define the speedup as:

$$S = \frac{T_{baseline}}{T_{optimized}}$$

Where:

- S is the speedup.
- $T_{baseline}$ is the execution time of the case that we take as a reference.
- $T_{optimized}$ is the execution time of cases that we want to know the relative improvement against the reference case.

6.3 Results

In this section we will explain all the different tests that we performed to measure our development. First of all, in Section 6.3.1 we will show how the optimal number of XIOS servers change in function of the output size. After that, in Section 6.3.2 we perform a comparison between all the output schemes that we previously mentioned. In Section 6.3.3 we perform the same type of comparison, but adding the time needed to convert GRIB to netCDF files. Finally, in Section 6.3.4 we perform a comparison test by adding to non-XIOS schemes the same number of cores and nodes that XIOS uses.

Figures use some abbreviations for XIOS: v1, v2, v3 and v4. They include the different optimization versions which are in increasing order. For example, v3 includes v1 and v2. The list is the following:

- XIOS v1: non-optimized version. It is only the IFS-XIOS integration development. It corresponds to Chapter 4.
- XIOS v2: it includes the optimization of the NPROMA blocks gather parallelization with OpenMP threads. It corresponds to Section 5.2.
- XIOS v3: it includes the optimized compilation of XIOS. It corresponds to Section 5.3.

- XIOS v4: it includes the overlapping computation and communication optimization. It corresponds to Section 5.4.

6.3.1 Optimal number of XIOS servers

In this test we execute IFS with XIOS under the same conditions, only changing the number of fields to be output. The purpose is to find the minimum number of XIOS servers needed that do not increase the total execution time depending on the output size. This means that we are finding the optimal number of XIOS servers for different number of fields to be output.

XIOS servers consume a considerable amount of memory to maintain the value of variables needed in some post-processing operations. This means that depending on the volume of the set of fields, more or less memory is allocated by XIOS servers. This is solved using the memory of more or less nodes of the supercomputer. Additionally, the more servers are working at the same time, the more fields can be processed in parallel, avoiding a bottleneck during the output process. In Figure 6.1 we use three different output sizes to find the corresponding optimal numbers: 12 3D fields, 6 3D fields and 1 3D field. When we output 12 fields, we need 10 servers, with 6 fields we need 5 servers and with 1 field we just need 1 server. It is visible that the number of needed servers scales in function of the output size.

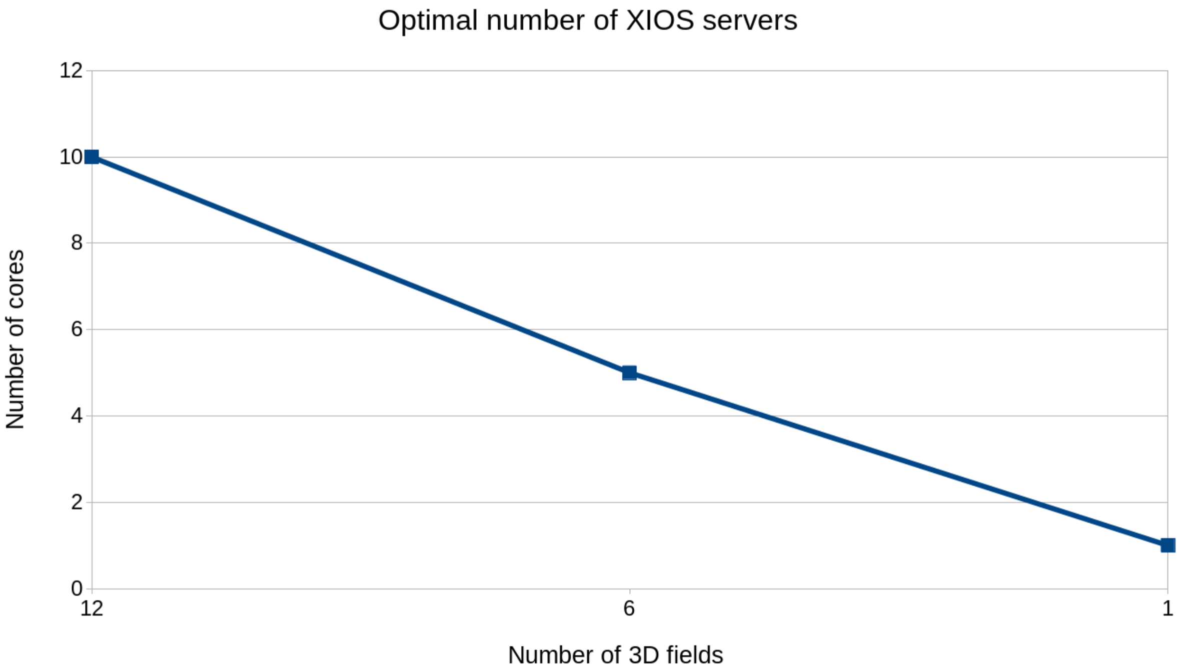


Figure 6.1: Optimal number of XIOS servers in function of the output size

There is an important point to mention. For this test we are placing one server per node, because if we try to fit those number of cores in less nodes, the execution crashes due to lack of memory. Thus, we can also say that Figure 6.1 shows the number of nodes needed in function of the output size. However, note that if we increase the number of XIOS servers per node, but keeping the needed amount of nodes, the execution successfully finishes spending the same elapsed time.

Therefore, this figure suggests that XIOS is memory sensitive. As we have explained, XIOS consumes a lot of memory because it accumulates data on server side just in case it is needed to apply time integration filters, i.e., post-processing over a period of time.

6.3.2 Comparison test between different output schemes

In this test we compare the different output schemes that we have available: IFS sequential output, MF I/O server, different IFS-XIOS optimized versions and IFS with no output. The fact that I/O schemes are compared with the no output case is to know the maximum achievable speedup because it does not have I/O overhead at all. It is important to remind that all executions are done using 702 MPI processes with 6 OpenMP threads each one. In addition, we are going to show results that take into account the whole execution of IFS-XIOS, this is, from the beginning to the end of the execution.

Figure 6.2 shows the execution time of the different schemes. The first column is the sequential scheme, which takes 9054 seconds. It is considerably slower than the rest of schemes. The second one, the MF I/O server, takes 7519 seconds. It is a good time compared to no output, which takes 7356 seconds.

The next four columns are the different XIOS versions. It is visible how each optimization improves the previous one. The XIOS v1 is only the IFS-XIOS integration, which takes 7773 seconds. It is faster than the sequential output, but slower than the MF I/O server. The next one, XIOS v2, parallelizes the NPROMA blocks gather using OpenMP threads. This optimization avoids that only the OpenMP master thread works while the rest of threads are idle. It takes 7705 seconds, which is a reduction of 68 seconds regarding the XIOS v1.

After that, the XIOS v3 uses the optimized compilation of XIOS. It is very important to reduce both client and server execution time, especially if it is necessary to run post-processing, which typically implies a lot of computation. The execution time is improved by 76 seconds with regard the XIOS v2, taking 7629 seconds.

Finally, the most optimized version, XIOS v4, re-orders the three necessary steps to perform output: update calendar, NPROMA blocks gather and send fields. The send fields step is performed in the IFS physics computation, taking advantage that there are not communications at all. Thus, it is possible to achieve better overlapping between XIOS communication and IFS computation. The execution time takes 7507 seconds, 122 seconds faster than the previous XIOS v3. It is also slightly faster than the MF I/O server (12 seconds), and much more than the sequential output (1547 seconds). In addition, it is only 151 seconds slower than no output.

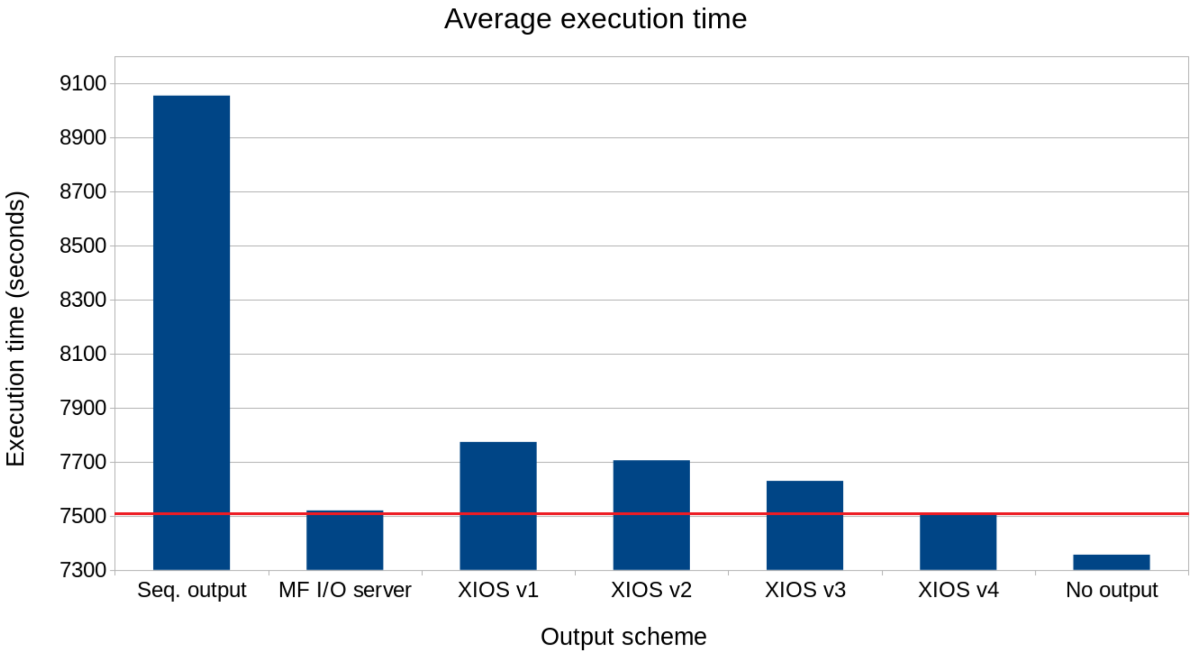


Figure 6.2: Comparison of the execution time of different output schemes

Figure 6.3 shows the speedup of the same previous comparison. The maximum achievable speedup is 1.231x, i.e., a 23.1% faster than the sequential output. Both MF I/O server and XIOS v4 execution time improvements are 20.4% and 20.6% respectively.

Comparing the sequential output (9054 seconds) and the no output tests (7356 seconds), it is easy to check that the output process requires an average of 1698 seconds, which represents a 23% more of the execution time of IFS. However, thanks to the most optimized version of the IFS-XIOS integration, this time is reduced from 1698 to 151 seconds, which represents only a 2% of the execution time of IFS, achieving that almost all the output process is hidden with the IFS computation.

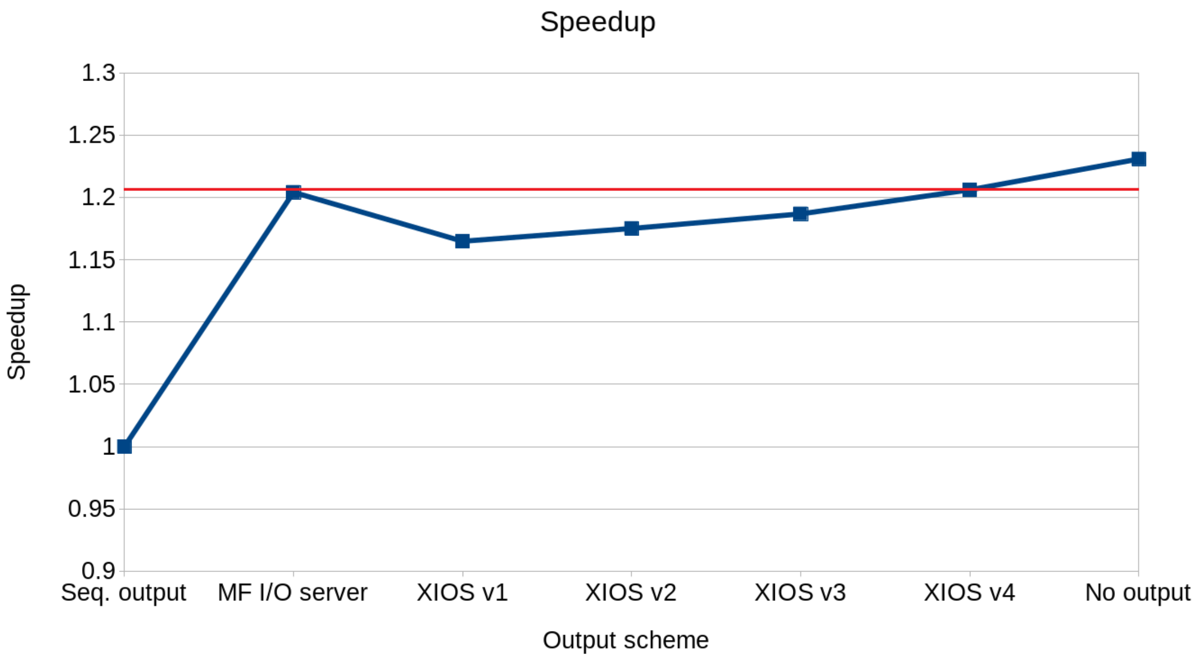


Figure 6.3: Comparison of the speedup of different output schemes

6.3.3 Comparison test adding the GRIB to netCDF post-processing

This test is the same as the previous one (Section 6.3.2), but with the additional cost of converting GRIB files to netCDF files. This cost is only added in sequential ouput and MF I/O server schemes, since they are the ones that write in GRIB format. Note that we do not include the no output case, which does not have neither GRIB format nor netCDF format, since it does not generate data.

This test is useful to know the potential benefit of using XIOS to avoid the costly GRIB to netCDF post-processing.

Figure 6.4 shows a comparison between the different output schemes, where the time needed to convert GRIB files to netCDF files is taken into account. The post-processing time needed for the conversion is really huge, about 13680 seconds (3.8 hours). It is much more slower than the simulation itself.

The execution time of the sequential output scheme is increased from 9054 seconds to 22734 seconds, and for the MF I/O server from 7519 seconds to 21199 seconds. It represents an increase of 151.1% and 181.9% respectively.

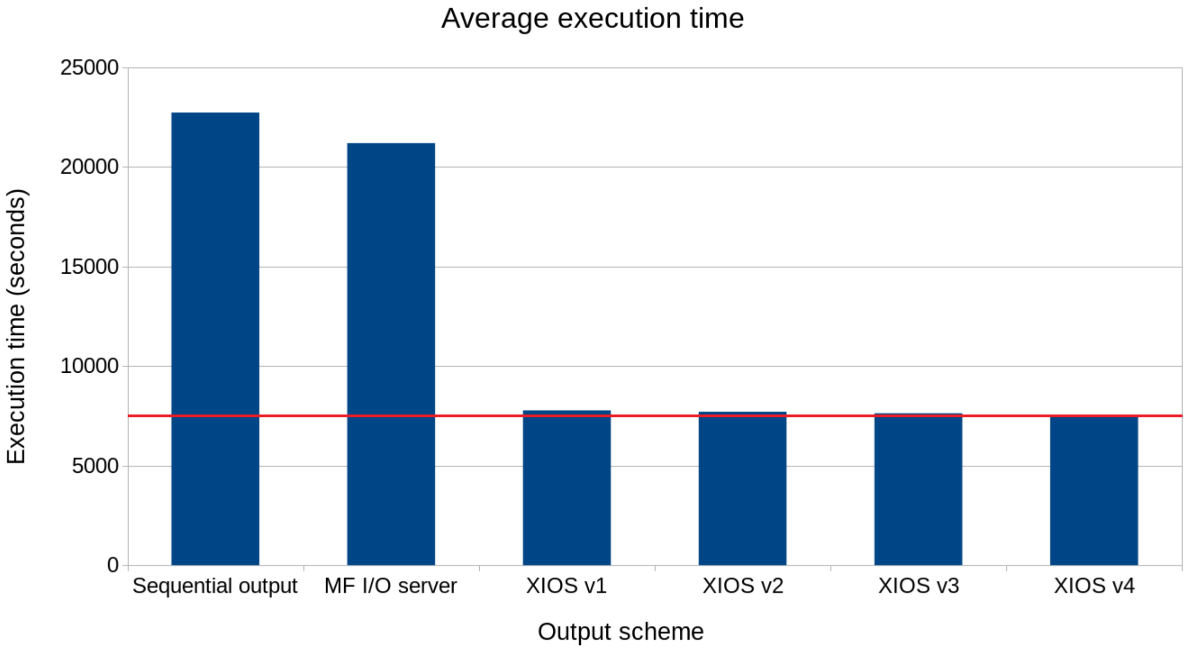


Figure 6.4: Comparison of the execution time of different output schemes, where both sequential output and MF I/O server have the additional cost of converting GRIB to netCDF files

Figure 6.5 shows the speedup of the same previous comparison. It is considerable the speedups that XIOS achieves in all its optimization versions: 2.92x in v1, 2.95x in v2, 2.97x in v3 and 3.02x in v4.

It is really evident that if we take into account both model and post-processing tasks, the benefit of using XIOS compared to the current I/O schemes is huge.

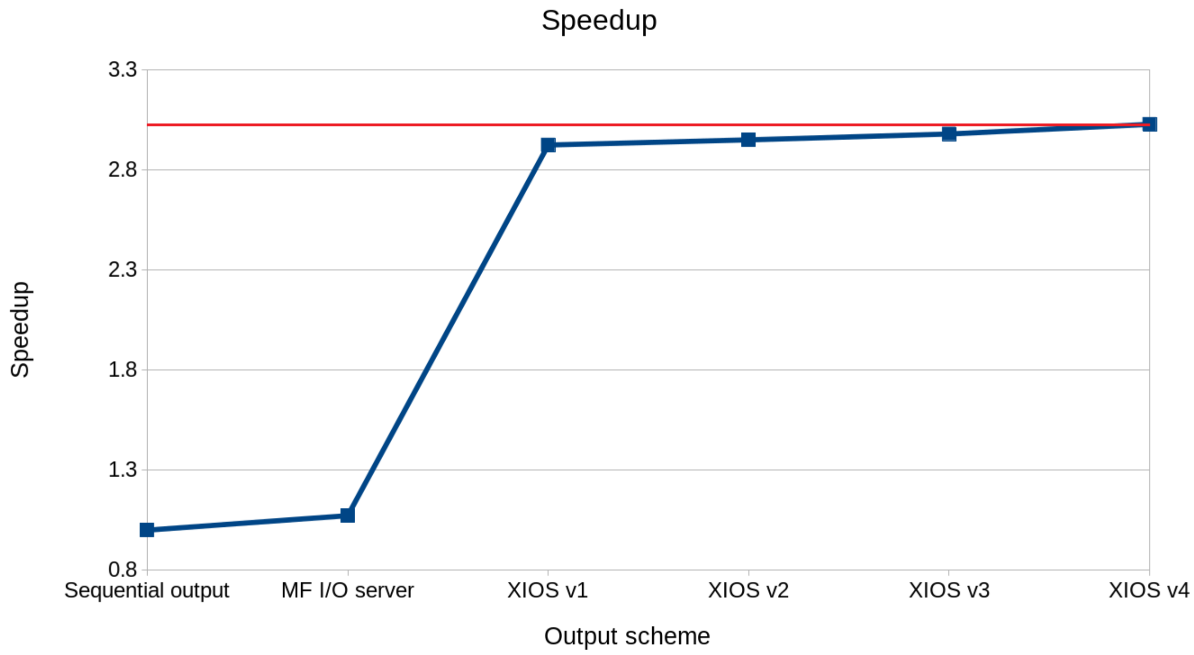


Figure 6.5: Comparison of the speedup of different output schemes, where both sequential output and MF I/O server have the additional cost of converting GRIB to netCDF files

6.3.4 Comparison test with additional computational resources

Finally, we perform a comparison with additional computational resources between the sequential output scheme and the most optimized version of IFS-XIOS, i.e., XIOS v4. The idea is to use in both cases, firstly, the same amount of cores, and secondly, the same amount of nodes. Thus, we guarantee that we compare cases using the same amount of computational resources. Furthermore, at the end there is the same comparison test, but also taking into account the GRIB to netCDF conversion cost for the sequential output scheme.

It is important to keep in mind that each Cray XC40 node has 36 physical cores, so using hyperthreading 72 threads. In our case, we are running 12 MPI processes per node, where each one has 6 OpenMP threads. Thus, we are completely fulfilling nodes.

We always take as a reference the optimal amount of XIOS servers (and nodes) that will be added to the other cases. Since we are outputting 12 3D fields, we use 10 XIOS servers spread along 10 nodes. Therefore, those additional resources used by XIOS, are added to the sequential output scheme as follows: if using the same amount of cores, we use 10 additional cores to run IFS ($702 + 10 = 712$). On the contrary, if using the same amount of nodes, we use 120 additional cores to run IFS ($702 + 120 = 822$).

Note that we do not include the MF I/O server. This is because of two reasons. Firstly, if we want to use 10 cores as dedicated I/O processes, the execution crashes since there are not enough resources. The minimum amount is 18, the default value. And secondly, if we want to use 10 nodes to place I/O processes, we cannot set it up because I/O processes are internally distributed by IFS. It is not possible to choose the geometry of the parallel job through aprun.

Figure 6.6 shows the execution time comparison of the two evaluated schemes. The first case is the sequential output. It has a considerable improvement using the same amount of nodes, where execution time is reduced from 9054 seconds to 8282 seconds. The second case is the XIOS v4, where the execution time is always kept to 7507 seconds, because it is the reference case regarding the amount of resources.

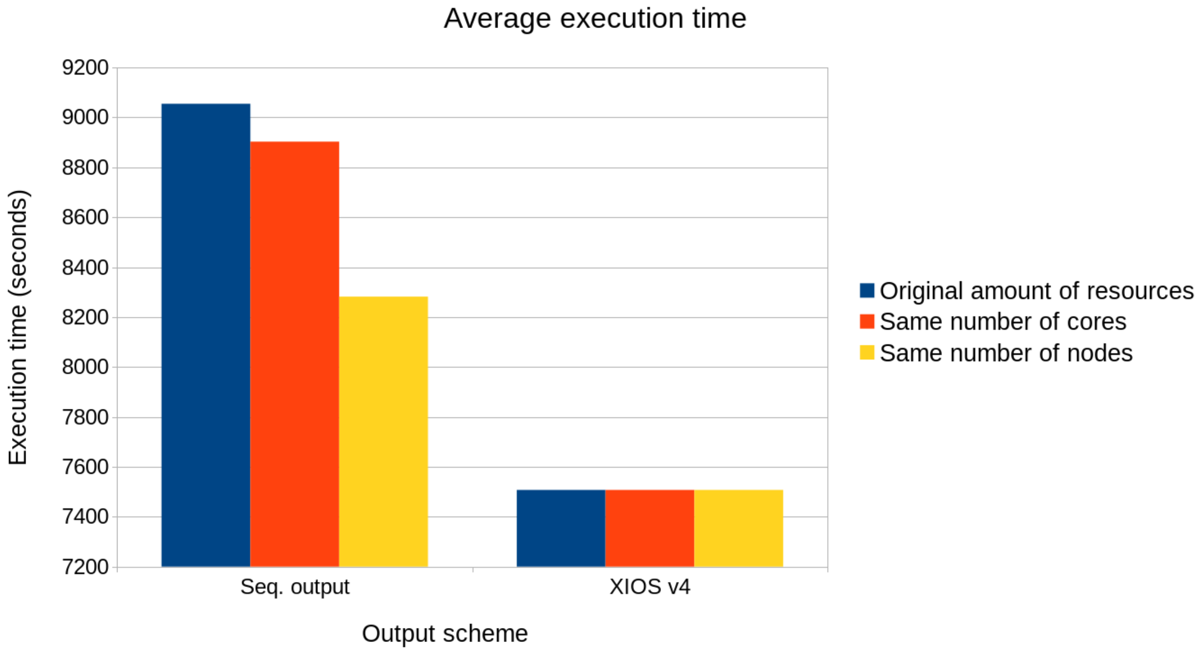


Figure 6.6: Comparison test with additional computational resources of the execution time of two different output schemes. In blue they are using the original amount of resources, in orange they are using the same amount of cores, and in yellow they are using the same amount of nodes

From previous Figure 6.6 we compute the speedup of Figure 6.7. It illustrates how the XIOS v4 scheme achieves a lower speedup when the sequential output scheme uses more computational resources.

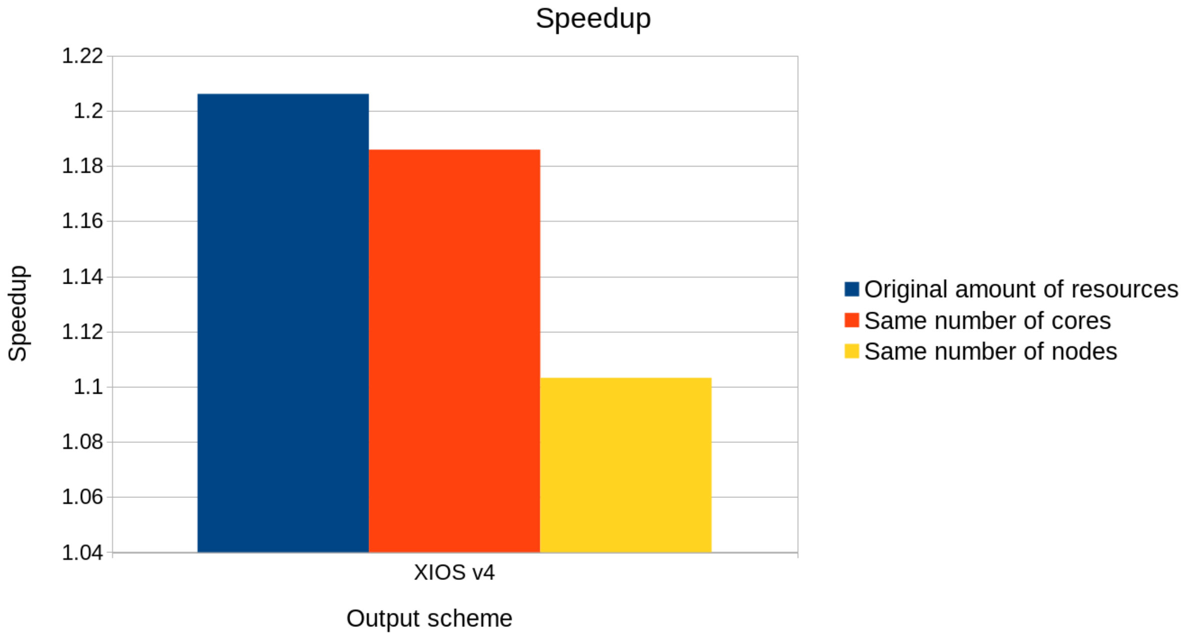


Figure 6.7: Speedup of XIOS v4 against the sequential output scheme with additional computational resources. In blue the original amount of resources, in orange the same amount of cores, and in yellow the same amount of nodes

Figure 6.8 illustrates the same execution time comparison of previous Figure 6.6, but also taking into account the execution time of converting GRIB to netCDF files. In this scenario, the benefit of using more computational resources for the sequential output scheme is less significant, because the increase added by the format conversion is really big.

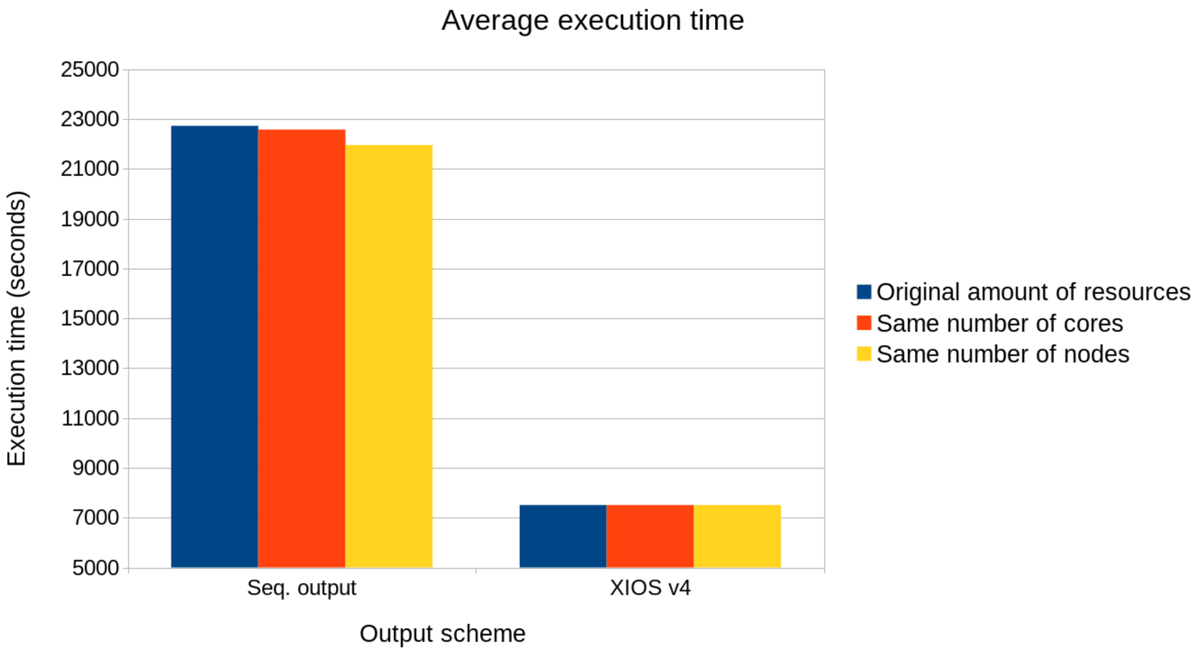


Figure 6.8: Execution time comparison test with additional computational resources and GRIB to netCDF conversion cost of two different output schemes. In blue they are using the original amount of resources, in orange they are using the same amount of cores, and in yellow they are using the same amount of nodes

Figure 6.9 shows the speedup computed from the previous figure, where it is evident that XIOS offers a considerable improvement with regard to the sequential output scheme. The speedup of XIOS v4 is 3.00x using the same amount of cores and 2.92x using the same amount of nodes.

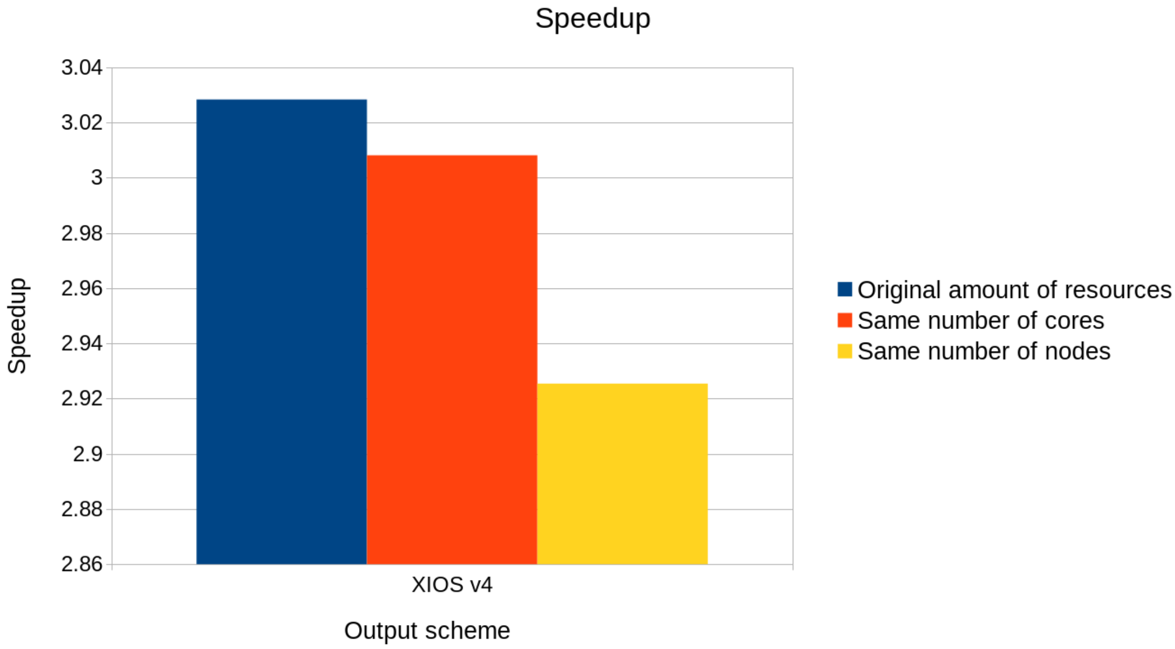


Figure 6.9: Speedup of XIOS v4 against the sequential output scheme with additional computational resources and GRIB to netCDF conversion cost. In blue the original amount of resources, in orange the same amount of cores, and in yellow the same amount of nodes

Chapter 7

Conclusions and future work

7.1 Conclusions

In this master's thesis we have integrated XIOS into IFS with a twofold objective: reduce the total execution time of the model and reduce the workflow's critical path by using online post-processing. In addition, we had a secondary objective which was to increase the usability of IFS by using an easier output configuration file compared to the current approach.

In order to achieve the objectives, we have presented a development which is characterized by being very easy to use. IFS initializes, sets up and finalizes XIOS through three subroutines. Then, we have designed a scheme to write data at the end of each output time step. This output scheme is made of three steps: update calendar, NPROMA blocks gather and send fields. This first development already improved the execution time with regard the sequential output scheme, despite of not being optimized. The execution time takes 7773 seconds, while the sequential output takes 9054 seconds, which adds a 23% of time to the execution of IFS.

After that, we have applied several optimization techniques to improve the bottlenecks that we detected in the performance analysis.

In the first optimization we have used OpenMP threads to parallelize the NPROMA blocks gather. Although it does not give a large improvement, it is important to not have threads in idle, while only works the master thread. This optimization would be especially beneficial when having larger subdomains with less MPI processes and more OpenMP threads per MPI process to ensure a good scalability for any configuration of future developments. The execution time is reduced 68 seconds with regard the first development.

The second optimization is an improvement in the XIOS compilation. After sorting out all the issues that we were having, we achieved to compile XIOS using the *-O3* optimization flag. Not surprisingly, there is an improvement in the execution time, which is reduced by 76 seconds regarding the previous optimization. This proves that it is important to compile external libraries using the best optimization flags of the compiler. Although this could seem trivial, a non-optimized compilation of external libraries could be a bottleneck for the scalability of a model.

The last optimization is the most beneficial in terms of computational performance. We have used more sophisticated tools such as Extrae and Paraver to find what could be improved. We have realized that there was not a real overlap in the computation and communication of IFS and XIOS respectively, so we have designed a new output scheme to re-arrange the steps and achieve an improved overlap. In this new scheme, the three steps of the first output scheme are split and placed along the IFS time step. This optimization takes 7507 seconds, improving the execution time by 122 seconds. Thanks to this last optimization, it is even slightly faster than the MF I/O server, which takes 7519 seconds. Furthermore, it is important to mention that this optimization, which also includes the previous ones, is only 151 seconds slower than IFS without outputting data at all, which represents only a 2% of the execution time of IFS. Within 151 seconds, IFS outputs through XIOS 3.2 TB of data. This optimization proves that the use of asynchronous communications to overlap with computation is sometimes not enough. In some cases, an additional study is needed to detect in which areas computation and communication can be effectively overlapped, taking into account other communication stages along the complete execution.

We have also considered other optimization techniques such as vectorization with

SIMD instructions, memory affinity and derived MPI Datatypes. However, we have not implemented them due to different reasons: the vectorization of the code that we would have implemented, it is already done by the compiler; the memory affinity improvement is already in use in IFS (however, we have proved that its use is very important for the performance of the execution); and the derived MPI Datatypes are not possible to be used, since data is sent to XIOS through Fortran subroutines.

When we also take into account the post-processing task to compare performance results, the benefit of using XIOS becomes enormous. In both sequential output and MF I/O server schemes is necessary to convert GRIB files to netCDF files in the post-processing task, which takes 13680 seconds (3.8 hours). The execution time is increased up to 22734 seconds in the sequential output and 21199 seconds in the MF I/O server. Our most optimized version is a 202% faster than the sequential output scheme and a 182% faster than the MF I/O server. Thus, we have implemented an scalable development that will address the I/O issue that is limiting the computational performance of IFS within EC-Earth. It represents a first step in EC-Earth that will considerably reduce the total execution time of large experiments, as well as it will increase the efficiency. This does not only imply to save thousands of computing hours, but also storage space because we will only store processed data ready to be used, instead of a huge amount of temporary raw data.

We have actively worked, discussed and collaborated with two European institutions, ECMWF and NLeSC, to succeed in the completion of this master's thesis. ECMWF is interested in this work because they will distribute OpenIFS with XIOS as an optional I/O scheme. They also showed interest to use XIOS for IFS in its seasonal forecasts, taking into account the benefits of avoiding the GRIB to netCDF files conversion.

Furthermore, the EC-Earth community is very interested in this work, because it is one of the key developments that need to be implemented for a future major release of EC-Earth. Some of the institutions involved in this community are BSC, KNMI, SMHI, etc.

7.2 Future work

In order to keep improving this work, there are several tasks that will be gradually and sequentially implemented in the future by different institutions:

- It is needed to implement vertical interpolations, but ensuring a user-friendly output configuration file. The problem it that XIOS has some online post-processing features that are (still) not available. This will be solved by re-using code from FullPos, but ensuring that the XIOS output configuration file does not lose usability.
- All the work done for IFS, will be ported to OpenIFS.
- At that time, XIOS will be already integrated with OpenIFS and NEMO, so it will be necessary to adapt EC-Earth components to generate on-line diagnostics through XIOS.
- Port to GPUs the XIOS source code that performs costly computations, such as diagnostics or interpolations.

Acronyms

ADIOS ADaptable I/O System.

AOGCM Atmosphere-Ocean Global Circulation Model.

API Application Programming Interface.

Arpege Action de Recherche Petite Echelle Grande Echelle.

BSC Barcelona Supercomputing Center.

CAM Community Atmosphere Model.

CDI Climate Data Interface.

CDI-pio CDI with parallel I/O.

CDT Cray Developer Toolkit.

CeCILL CEA CNRS INRIA Logiciel Libre.

CESM Community Earth System Model.

CFIO Climate Fast Input/Output.

CMIP Coupled Model Intercomparison Project.

CMOR Climate Model Output Rewriter.

CPU Central Processing Unit.

DART Decoupled and Asynchronous Remote Transfers.

DKRZ German Climate Computing Centre.

ECMWF European Centre for Medium-Range Weather Forecasts.

ESM Earth System Modelling.

FDB Fields Data Base.

GPU Graphic Processing Unit.

GRIB General Regularly-distributed Information in Binary form.

H2020 Horizon 2020.

HDF Hierarchical Data Format.

HPC High Performance Computing.

HPCF High Performance Computing Facility.

I/O Input/Output.

IFS Integrated Forecast System.

ILP Instruction Level Parallelism.

IPSL Institute Pierre Simon Laplace.

KNMI Koninklijk Nederlands Meteorologisch Instituut.

MF Météo-France.

MIPS Million Instructions Per Second.

MPG Max-Planck-Institute for Meteorology.

MPI Message Passing Interface.

MPMD Multiple-Program Multiple-Data.

NEMO Nucleus for European Modelling of the Ocean.

netCDF Network Common Data Format.

NLeSC Netherlands eScience Center.

NUMA Non-Uniform Memory Access.

NWP Numerical Weather Prediction.

OpenMP Open Multi-Processing.

OST Object Storage Target.

PIO Parallel I/O library.

POSIX Portable Operating System Interface.

PRIMAVERA PRocess-based climate sIMulation: AdVances in high-resolution modelling and European climate Risk Assessment.

SIMD Single Instruction, Multiple Data.

SMHI Swedish Meteorological and Hydrological Institute.

SMS Supervisor Monitor Scheduler.

SURFEX Surface Externalisée.

WAM WAve Model.

XIAS XIOS Interface for Arpege-climat and Surfex.

XIOS XML Input/Output Server.

XML Extensible Markup Language.

References

[1] Efecan Poyraz, Heming Xu, and Yifeng Cui. “Application-specific I/O Optimizations on Petascale Supercomputers”. In: *ICCS 2014. 14th International Conference on Computational Science*. Vol. 29. Elsevier, Jan. 2014, pp. 910–923. DOI: `10.1016/J.PROCS.2014.05.082`.

[2] C. Prodhomme, L. Batté, F. Massonnet, P. Davini, O. Bellprat, V. Guemas, F. J. Doblas-Reyes, C. Prodhomme, L. Batté, F. Massonnet, P. Davini, O. Bellprat, V. Guemas, and F. J. Doblas-Reyes. “Benefits of Increasing the Model Resolution for the Seasonal Forecast Quality in EC-Earth”. In: *Journal of Climate* 29.24 (Dec. 2016), pp. 9141–9162. ISSN: 0894-8755. DOI: `10.1175/JCLI-D-16-0117.1`.

[3] Hisashi Yashiro, Koji Terasaki, Takemasa Miyoshi, and Hirofumi Tomita. “Performance evaluation of a throughput-aware framework for ensemble data assimilation: the case of NICAM-LETKF”. In: *Geoscientific Model Development* 9.7 (2016), pp. 2293–2300. ISSN: 1991-9603. DOI: `10.5194/gmd-9-2293-2016`.

[4] Adrian Jackson, Fiona Reid, Joachim Hein, Alejandro Soba, and Xavier Saez. “High Performance I/O”. In: *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, Feb. 2011, pp. 349–356. ISBN: 978-1-4244-9682-2. DOI: `10.1109/PDP.2011.16`.

[5] Xavier Yepes-Arbós, M. C. Acosta, Kim Serradell, Alicia Sanchez Lorente, and Francisco J. Doblas-Reyes. *Simulation-based performance analysis of EC-Earth 3.2.0 using Dimemas*. Tech. rep. 2017, p. 30.

[6] Xavier Yepes-Arbós, M. C. Acosta, Kim Serradell, Alicia Sanchez Lorente, and Francisco J. Doblas-Reyes. “Simulation-based performance analysis of

EC-Earth 3.2.0 using Dimemas". In: *EGU General Assembly 2017*. Vienna: Copernicus GmbH, 2017.

[7] Xavier Yepes-Arbós, M. C. Acosta, Kim Serradell, Oriol Mula-Valls, and Francisco J. Doblas-Reyes. *Scalability and performance analysis of EC-Earth 3.2.0 using a new metric approach (Part II)*. Tech. rep. 2016, p. 56.

[8] Mario C. Acosta, Xavier Yepes-Arbós, Kim Serradell, Alicia Sanchez Lorente, and Francisco J. Doblas-Reyes. *Performance study of OpenIFS : towards a more efficiently scalable model*. 2017.

[9] Zhuo Liu, Bin Wang, Teng Wang, Yuan Tian, Cong Xu, Yandong Wang, Weikuan Yu, Carlos A. Cruz, Shujia Zhou, Tom Clune, and Scott Klasky. "Profiling and Improving I/O Performance of a Large-Scale Climate Scientific Application". In: *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*. IEEE, July 2013, pp. 1–7. ISBN: 978-1-4673-5775-3. DOI: `10.1109/ICCCN.2013.6614174`.

[10] S.R.M. Barros, D. Dent, L. Isaksen, G. Robinson, G. Mozdzynski, and F. Wollenweber. "The IFS model: A parallel production weather code". In: *Parallel Computing* 21.10 (1995), pp. 1621–1638. ISSN: 0167-8191. DOI: `10.1016/0167-8191(96)80002-0`.

[11] Wilco Hazeleger, Camiel Severijns, Tido Semmler, Simona Ştefănescu, Shuting Yang, Xueli Wang, Klaus Wyser, Emanuel Dutra, José M. Baldasano, Richard Bintanja, Philippe Bougeault, Rodrigo Caballero, Annica M. L. Ekman, Jens H. Christensen, Bart van den Hurk, Pedro Jimenez, Colin Jones, Per Kållberg, Torben Koenigk, Ray McGrath, Pedro Miranda, Twan Van Noije, Tim Palmer, José A. Parodi, Torben Schmith, Frank Selten, Trude Storelvmo, Andreas Sterl, Honoré Tapamo, Martin Vancoppenolle, Pedro Viterbo, and Ulrika Willén. "EC-Earth: A Seamless Earth-System Prediction Approach in Action". In: *Bulletin of the American Meteorological Society* 91.10 (Oct. 2010), pp. 1357–1363. ISSN: 0003-0007. DOI: `10.1175/2010BAMS2877.1`.

[12] PRIMAVERA Wiki. *PRIMAVERA*. URL: `http://proj.badc.rl.ac.uk/primavera` (visited on 04/12/2018).

[13] PCMDI - Program For Climate Model Diagnosis and Intercomparison. *CMIP - Coupled Model Intercomparison Project - Overview.* URL: `https://cmip.llnl.gov` (visited on 01/08/2018).

[14] Yann Meurdesoif, A Caubel, R Lacroix, J Dérouillat, and M H Nguyen. *XIOS Tutorial.* 2016.

[15] Rajeev Thakur. *Parallel I/O and MPI-IO.*

[16] Yu-Heng Tseng and Chris Ding. “Efficient Parallel I/O in Community Atmosphere Model (CAM)”. In: *The International Journal of High Performance Computing Applications* 22.2 (May 2008), pp. 206–218. ISSN: 1094-3420. DOI: `10.1177/1094342008090914`.

[17] Luis Kornblueh, Deike Kleberg, Uwe Schulzweida, Mathias Pütz, Christoph Pospiech, Thomas Jahns, Moritz Hanke, Jörg Behrens, and Mathis Rosenhauer. *Parallel I/O for Earth System Modelling.* 2012.

[18] Kui Gao, Wei-keng Liao, Arifa Nisar, Alok Choudhary, Robert Ross, and Robert Latham. “Using Subfiling to Improve Programming Flexibility and Performance of Parallel Shared-file I/O”. In: *2009 International Conference on Parallel Processing.* IEEE, Sept. 2009, pp. 470–477. ISBN: 978-1-4244-4961-3. DOI: `10.1109/ICPP.2009.68`.

[19] Yinlong Zou, Wei Xue, and Shenshen Liu. “A case study of large-scale parallel I/O analysis and optimization for numerical weather prediction system”. In: *Future Generation Computer Systems* 37 (July 2014), pp. 378–389. ISSN: 0167-739X. DOI: `10.1016/J.FUTURE.2013.12.039`.

[20] Moritz Hanke. *Parallel I/O: Review on information on techniques for parallel I/O in High Performance Computing.* 2010.

[21] Matthieu Haefele. *Parallel I/O for High Performance Computing.* 2011.

[22] Message Passing Interface Forum. *MPI-2 : Extensions to the Message-Passing Interface.* Tech. rep. 2003, p. 370.

[23] Ramses van Zon. *MPI-IO.* 2013.

[24] HDF5 Group. *HDF5.* URL: `https://support.hdfgroup.org/HDF5` (visited on 01/02/2018).

[25] Unidata. *Network Common Data Form (NetCDF).* URL: `https://www.unidata.ucar.edu/software/netcdf` (visited on 01/02/2018).

[26] Parallel netCDF Wiki. *Network Common Data Form (NetCDF)*. URL: `https://trac.mcs.anl.gov/projects/parallel-netcdf` (visited on 01/02/2018).

[27] John M. Dennis, Jim Edwards, Ray Loy, Robert Jacob, Arthur A. Mirin, Anthony P. Craig, and Mariana Vertenstein. "An Application Level Parallel I/O Library for Earth System Models". In: *The International Journal of High Performance Computing Applications* 26.1 (Feb. 2012), pp. 43–53. ISSN: 1094-3420. DOI: `10.1177/1094342011428143`.

[28] Moritz Hanke, Joachim Biercamp, Carlos Osuna Escamilla, Thomas Jahns, Deike Kleberg, Paul Selwood, and Steve Mullerworth. *Deliverable 7.3 – Reference implementations of Parallel I/O and of I/O Server*. Tech. rep. 2013.

[29] Github NCAR. *Parallel I/O library (PIO)*. URL: `http://ncar.github.io/ParallelIO` (visited on 10/24/2017).

[30] Chen Jin, Scott Klasky, Stephen Hodson, Weikuan Yu, Jay Lofstead, Hasan Abbasi, Karsten Schwan, Matthew Wolf, Wei-keng Liao, Alok Choudhary, Manish Parashar, Ciprian Docan, and Ron Oldfield. "Adaptive IO System (ADIOS)". In: *Cray Users Group (CUG) Workshop*. 2008, pp. 1–8.

[31] Norbert Podhorszki. *Using the Adaptable I/O System (ADIOS)*. 2014.

[32] Jay Lofstead, Scott Klasky, Hasan Abbasi, and Karsten Schwan. *ADaptable IO System (ADIOS) for Scientific Codes*. 2008.

[33] Max-Planck-Institut für Meteorologie. *CDI*. URL: `https://code.mpimet.mpg.de/projects/cdi` (visited on 10/24/2017).

[34] Eric Maisonnave, Irina Fast, Thomas Jahns, Joachim Biercamp, Stéphane Sénési, Yann Meurdesoif, and Uwe Fladrich. *CDI-pio & XIOS I/O servers compatibility with HR climate models*. Tech. rep. 2017, p. 19.

[35] X. M. Huang, W. C. Wang, H. H. Fu, G. W. Yang, B. Wang, and C. Zhang. "A fast input/output library for high-resolution climate models". In: *Geoscientific Model Development* 7.1 (Jan. 2014), pp. 93–103. ISSN: 1991-9603. DOI: `10.5194/gmd-7-93-2014`.

[36] Yann Meurdesoif, M H Nguyen, R Lacroix, A Caubel, O Abramkina, Y Wang, and J Dérouillat. *XIOS and I/O Where are we?* 2017.

[37] Stéphane Sénési. *XIAS - XIOS Interface for Arpege-climat and Surfex*. Tech. rep. 2016, p. 17.

[38] ECMWF. *Atmospheric dynamics.* URL: `https://www.ecmwf.int/en/research/modelling-and-prediction/atmospheric-dynamics` (visited on 10/23/2017).

[39] ECMWF. *Atmospheric physics.* URL: `https://www.ecmwf.int/en/research/modelling-and-prediction/atmospheric-physics` (visited on 10/23/2017).

[40] ECMWF. *Modelling and Prediction.* URL: `https://www.ecmwf.int/en/research/modelling-and-prediction` (visited on 01/24/2018).

[41] Sarah N. Collins, Robert S. James, Pallav Ray, Katherine Chen, Angie Lassman, and James Brownlee. "Grids in Numerical Weather and Climate Models". In: *Climate Change and Regional/Local Responses.* Ed. by Yuanzhi Zhang and Pallav Ray. InTech, 2013. Chap. 4, pp. 111–128. ISBN: 978-953-51-1132-0. DOI: `10.5772/55922`.

[42] Paul Dando. *Changes to ECMWF's grids in 2016.* Reading, 2015.

[43] Sylvie Malardel, Nils Wedi, Willem Deconinck, Michail Diamantakis, Christian Kühnlein, George Mozdzynski, Mats Hamrud, and Piotr Smolarkiewicz. "A new grid for the IFS". In: *ECMWF* 146 (Jan. 2016), pp. 23–28. DOI: `10.21957/zwdu9u5i`.

[44] ECMWF. *IFS Documentation – Cy43r1 Operational implementation 22 Nov 2016. Part VI: Technical and Computational Procedures.* Tech. rep. 2016, pp. 1–227.

[45] Nils P. Wedi. *PrepIFS - User Guide.* URL: `http://www.prism.enes.org/Software/WSS/prepifs/prepIFSUserGuide/prepIFSUserGuide.html` (visited on 03/19/2018).

[46] Yann Meurdesoif. *XIOS User Guide.* Tech. rep. 2017, pp. 1–51.

[47] Yann Meurdesoif. *XIOS Fortran Reference Guide.* Tech. rep. 2017, pp. 1–52.

[48] George J. Boer, Douglas M. Smith, Christophe Cassou, Francisco Doblas-Reyes, Gokhan Danabasoglu, Ben Kirtman, Yochanan Kushnir, Masahide Kimoto, Gerald A. Meehl, Rym Msadek, Wolfgang A. Mueller, Karl E. Taylor, Francis Zwiers, Michel Rixen, Yohan Ruprich-Robert, and Rosie Eade. "The Decadal Climate Prediction Project (DCPP) contribution to CMIP6". In: *Geoscientific Model Development* 9.10 (Oct. 2016), pp. 3751–3777. ISSN: 1991-9603. DOI: `10.5194/gmd-9-3751-2016`.

[49] Brian Eaton, Jonathan Gregory, Bob Drach, Karl Taylor, Steve Hankin, Jon Blower, John Caron, Rich Signell, Phil Bentley, Greg Rappa, Heinke Höck, Alison Pamment, Martin Juckes, and Martin Raspaud. *NetCDF Climate and Forecast (CF) Metadata Conventions*. Tech. rep. 2009, p. 156.

[50] ECMWF. *Supercomputer*. URL: `https://www.ecmwf.int/en/computing/our-facilities/supercomputer` (visited on 01/24/2018).

[51] Lustre. *About the Lustre File System*. URL: `http://lustre.org/about` (visited on 04/06/2018).

[52] BSC. *BSC tools*. URL: `https://tools.bsc.es` (visited on 02/28/2018).

[53] *MPI Data Types*. 2012.

Appendix A

Grid-point decomposition variables

Variable	**Arrays dimensions and description**
LSPLITLAT	(1:NDGENL) Logical indicating whether a given row on the ”A” set is split with another ”A” set.
MYFRSTACTLAT	Scalar The first latitude row (global index) on this ”A” set (1..NDGLG) (Equivalent to NFRSTLAT(MYSETA))
MYLATS	(1:NDGENL) The latitude row (global index) a given row on this ”A” set corresponds to.
MYLSTACTLAT	Scalar The last latitude row (global index) on this ”A” set (1..NDGLG) (Equivalent to NLSTLAT(MYSETA))

MYPROC	Scalar Logical processor ID (1..NPROC). Note, processor numbering does not follow the normal Fortran array ordering (row first), but instead runs in a column first order, so processor ”1” is in the North Western corner, processor ”2” in to the South of this and so on
MYSETA	Scalar Which ”A” set (North-South) this processor is in (1..NPRGPNS)
MYSETB	Scalar Which ”B” set (East-West) this processor is in (1..NPRGPEW)
NDGENL	Scalar Number of latitude rows handled by this ”A” set
NFRSTLAT	(1..NPRGPNS) The first latitude row (global index) for a given ”A” set. (1..NDGLG)
NFRSTLOFF	Scalar Offset of the first latitude row (global index). (Equivalent to MYFRSTACTLAT-1)
NLSTLAT	(1:NPRGPNS) The last latitude row (global index) for a given ”A” set. (1..NDGLG)
NPTRFRSTLAT	(1:NPRGPNS) Index of the first latitude strip on the given ”A” set. (Used for indexing NSTA and NONL arrays)
NPTRLAT	(1:NDGLG) Index of the first latitude strip of the given global latitude. (Used for indexing NSTA and NONL arrays)

NPTRLSTLAT	(1:NPRGPNS) Index of the last latitude strip on the given ”A” set. (Used for indexing NSTA and NONL arrays)
NSTA	(1:NDGLG+NPRGPNS-1, 1:NPRGPEW) Number of grid points from Greenwich meridian at the start of the given latitude strip on the given ”B” set. Counting starts at 1, so for a grid point at the start of a row (i.e. on the meridian) NSTA(Index,1)=1
NONL	(1:NDGLG+NPRGPNS-1, 1:NPRGPEW) Number of grid points on this latitude strip within my ”B” set

Table A.1: Variables describing the grid-point decomposition (Adapted from [44])

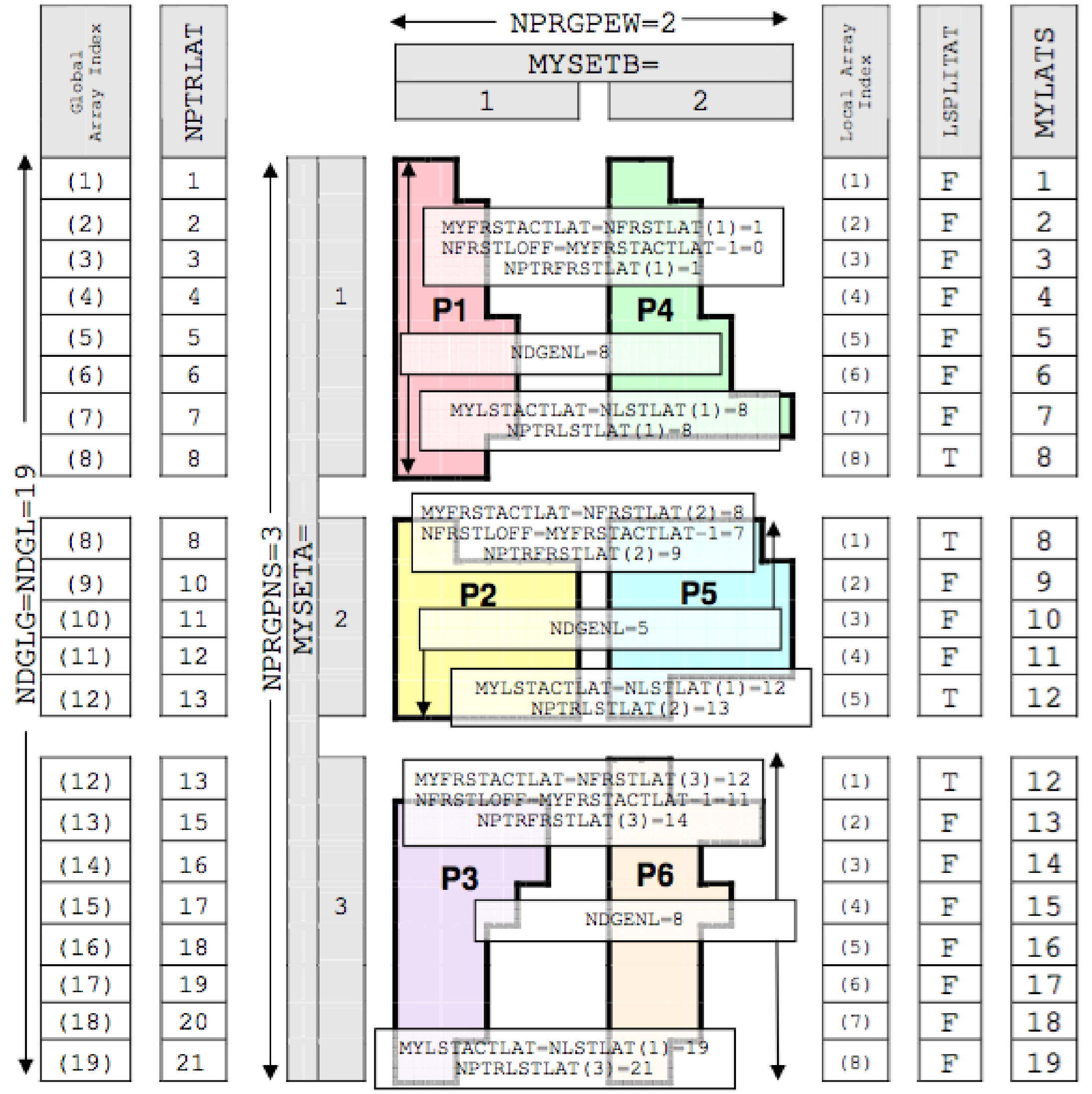


Figure A.1: Variables describing the grid-point decomposition (Reproduced from [44])

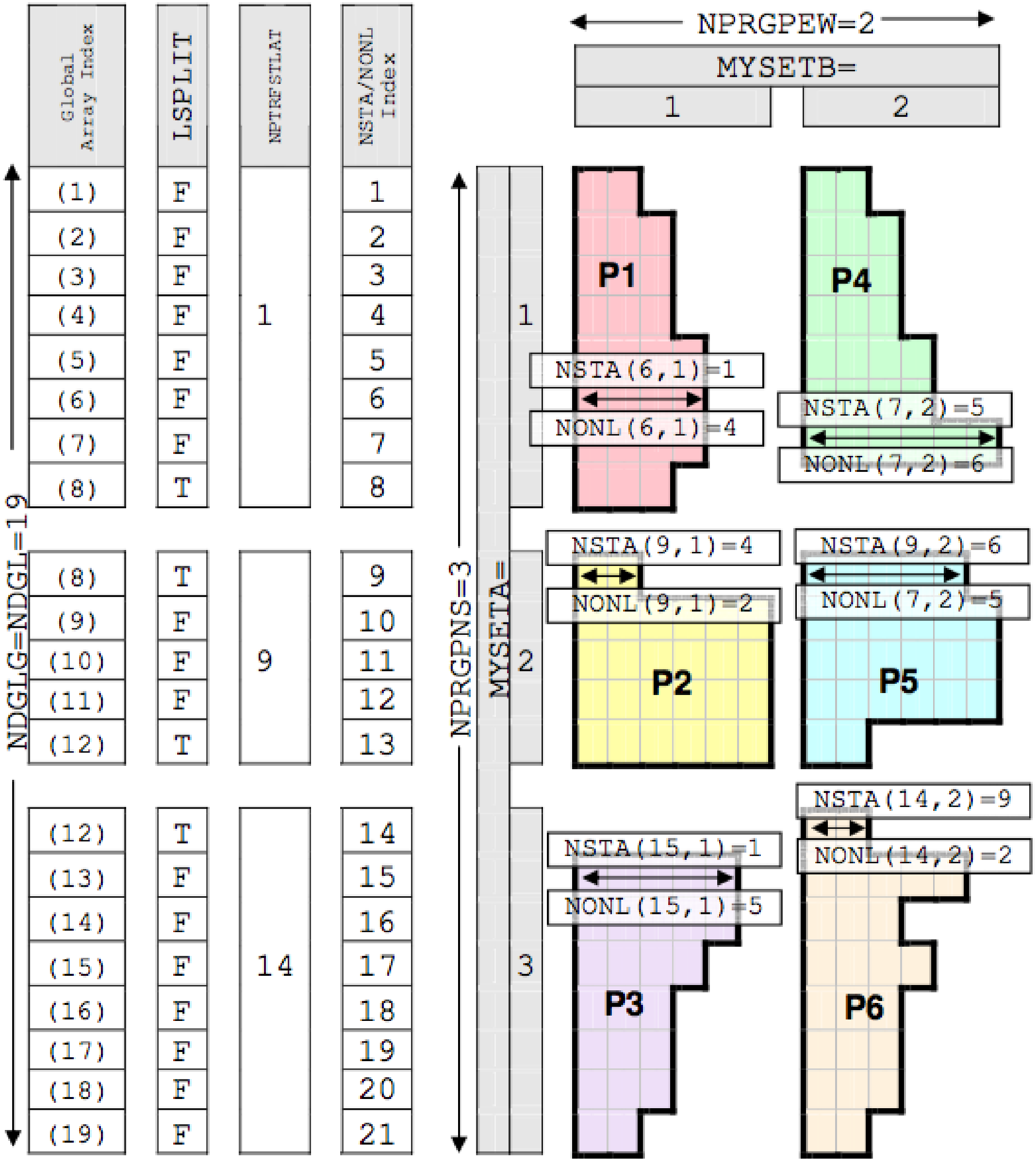


Figure A.2: NSTA and NONL arrays in the grid-point decomposition (Reproduced from [44])