Master in High Performance Computing

# OGSTM - BFM I/O IMPROVEMENTS

*Supervisors*:
Giorgio Bolzon,
Ivan Girotto,
Alberto Sartori,

*Candidate*:
Gianluca Coidessa

6th edition
2019–2020

## Acknowledgements

The research reported in this work was supported by OGS and CINECA under HPC-TRES program award number 2020 - 07.

# CONTENTS

# List of Abbreviations

OGSTM = OGS Transport Model

BFM = Biogeochemical Flux Model

OGCM = Ocean Global Circulation Model

NEMO = Nucleus (for) European Modeling (of) (the) Ocean

CMEMS Copernicus Marine Environment Monitoring Services

# OVERVIEW

This thesis project is part of a restyling work on the OGSTM - BFM model that is used to study the biogeochemical properties of Mediterranean Sea waters. OGS is one of the research centres that are involved in the Copernicus (European Union's Earth observation programme) project, being part of the 6$^{th}$ MSC (Monitoring and Forecasting Centres) which aim is to provide regular and systematic information about the physical state of the ocean and marine ecosystems for the Mediterranean Sea. OGS runs the OGSTM - BFM model (ocean numerical model) assimilating TAC (Thematic Data Assembly Centres) data to generate reanalyse (20 years in the past), analyse (today) and 10-days forecasts of the Mediterranean Sea. This thesis is aimed at the improvements of the outputs to enhance the dumping capability without affecting the total run time.

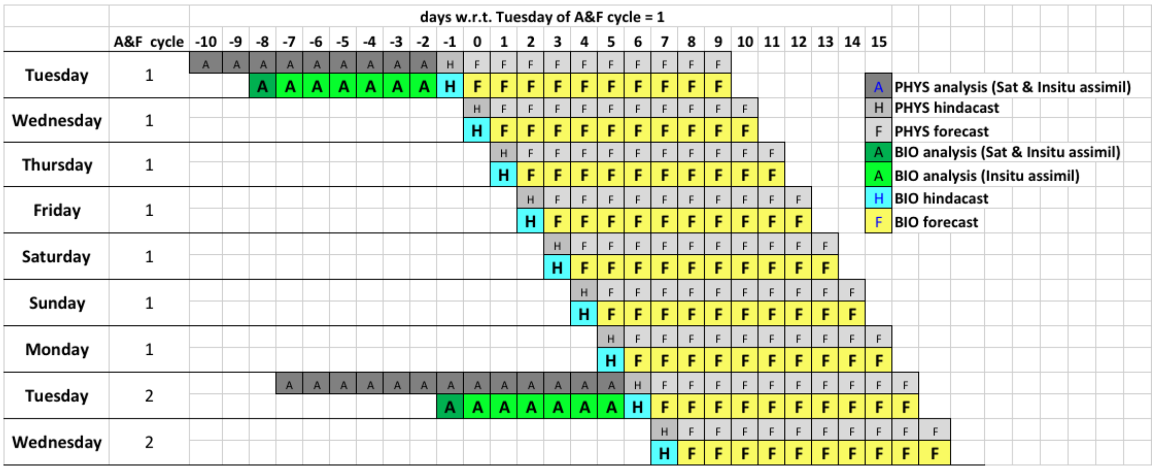
Before this thesis, the output of the model was serialised on a single processor. This way of handling I/O was time consuming as the new Copernicus project requests asked to save daily the model output (before was saved in part weekly and in part monthly).

The first aim of the thesis was to generate a distributed parallel I/O in which there are more writing processes that in parallel dump different output variables, in a way to scale down writing time. After the implementation of this paradigm the focus was directed to apply the same paradigm to the assimilation part (3DVAR) of model, as the daily assimilation had to be introduced (before was weekly).

# 1 INTRODUCTION

## 1.1Copernicus framework

The biogeochemical analysis and forecasts for the Mediterranean Sea at 1/24 degree are produced by means of the MedBFM model system (i.e. the physical-biogeochemical OGSTM-BFM model coupled with the 3DVarBio assimilation scheme). MedBFM model is run by OGS and uses as physical forcing the outputs of the NEMO-OceanVar model system (managed by CMCC). Seven days of analysis are produced weekly on Tuesday, with assimilation of surface chlorophyll concentration from satellite observations (provided by the CMEMS-OCTAC) and of vertical profiles of chlorophyll and nitrate from BGC-Argo floats (provided by CORIOLIS and LOV data centres). One day of hindcast and ten days of forecast are produced daily. The analysis and forecast products are released after completion of the Med-PHY workflow (Fig 1). On Tuesday, the workflow consists of 7 days of analysis (-8 to -2), one day of hindcast (-1) and 10 days of forecast (0 to 9). From Wednesday to Monday, the workflow consists of one day of hindcast and 10 days of forecast. The data assimilation cycle (Tuesday run) uses the satellite chlorophyll (i.e., a composite average in the range of ±3 days) at 12:00 UTC of the Monday of the previous week (day -8) and the in situ vertical profiles of chlorophyll and nitrate at 12:00 UTC from day -8 to day -2. On day -8, satellite and float assimilation is performed disjointedly.



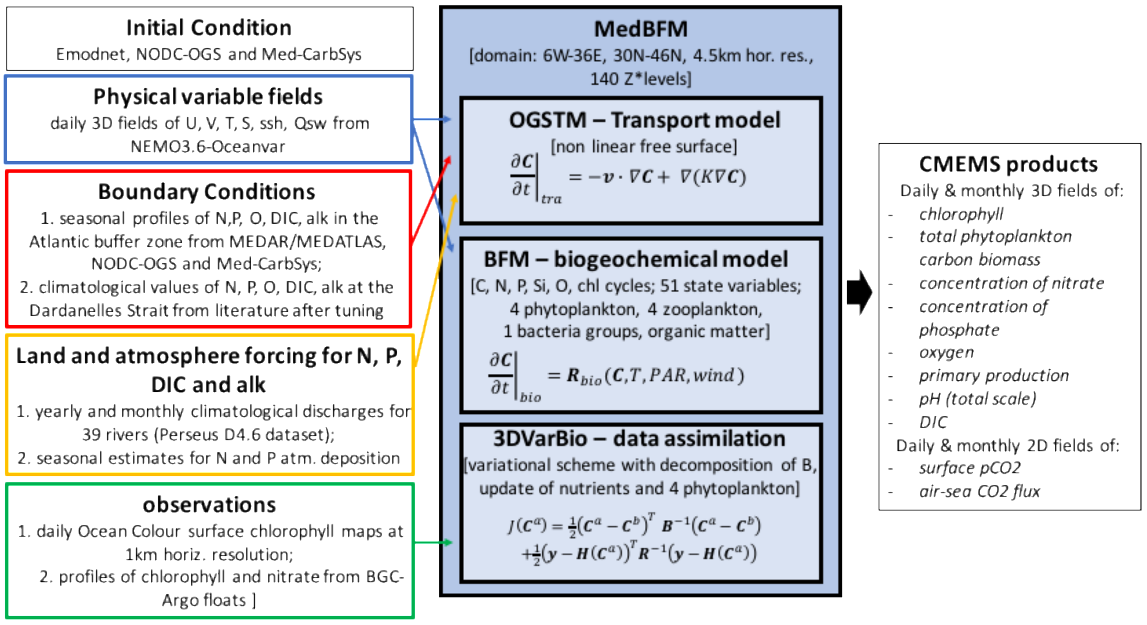
| | | days w.r.t. Tuesday of A&F cycle = 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A&F cycle | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | | |
| Tuesday | 1 | A | A | A | A | A | A | A | A | A | H | F | F | F | F | F | F | F | F | F | F | | | | | | | | |
| | | | | A | A | A | A | A | A | A | H | F | F | F | F | F | F | F | F | F | F | | | | | | | | |
| Wednesday | 1 | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | | | | |
| | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | | | | |
| Thursday | 1 | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | | | |
| | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | | | |
| Friday | 1 | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | | |
| | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | | |
| Saturday | 1 | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | |
| | | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | | |
| Sunday | 1 | | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | |
| | | | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | | |
| Monday | 1 | | | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | |
| | | | | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F | | |
| Tuesday | 2 | | | | A | A | A | A | A | A | A | A | A | A | A | A | A | H | F | F | F | F | F | F | F | F | F | F | |
| | | | | | | | | | | | A | A | A | A | A | A | A | H | F | F | F | F | F | F | F | F | F | F | |
| Wednesday | 2 | | | | | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F |
| | | | | | | | | | | | | | | | | | | | H | F | F | F | F | F | F | F | F | F | F |

| | |
|---|---|
| A | PHYS analysis (Sat & Insitu assimil) |
| H | PHYS hindcast |
| F | PHYS forecast |
| A | BIO analysis (Sat & Insitu assimil) |
| A | BIO analysis (Insitu assimil) |
| H | BIO hindcast |
| F | BIO forecast |

*Figure1: Analysis, hindcast and forecast scheme.*

## 1.2 Model Overview

MedBFM v3.1 consists of the coupled physical-biogeochemical OGSTM-BFM model and the 3DVarBio assimilation scheme (Salon et al., 2019; Lazzari et al., 2010, 2012, 2016; Cossarini et al., 2015; Teruzzi et al., 2014, 2018, 2019; Cossarini et al., 2019).



Initial Condition
Emodnet, NODC-OGS and Med-CarbSys
Physical variable fields
daily 3D fields of U, V, T, S, ssh, Qsw from NEMO3.6-Oceanvar
Boundary Conditions
1. seasonal profiles of N,P, O, DIC, alk in the Atlantic buffer zone from MEDAR/MEDATLAS, NODC-OGS and Med-CarbSys;
2. climatological values of N, P, O, DIC, alk at the Dardanelles Strait from literature after tuning
Land and atmosphere forcing for N, P, DIC and alk
1. yearly and monthly climatological discharges for 39 rivers (Perseus D4.6 dataset);
2. seasonal estimates for N and P atm. deposition
observations
1. daily Ocean Colour surface chlorophyll maps at 1km horiz. resolution;
2. profiles of chlorophyll and nitrate from BGC-Argo floats ]
MedBFM
[domain: 6W-36E, 30N-46N, 4.5km hor. res., 140 Z*levels]
OGSTM – Transport model
[non linear free surface]
$\left.\frac{\partial C}{\partial t}\right|_{tra} = -v \cdot \nabla C + \nabla(K\nabla C)$
BFM – biogeochemical model
[C, N, P, Si, O, chl cycles; 51 state variables; 4 phytoplankton, 4 zooplankton, 1 bacteria groups, organic matter]
$\left.\frac{\partial C}{\partial t}\right|_{bio} = R_{bio}(C, T, PAR, wind)$
3DVarBio – data assimilation
[variational scheme with decomposition of B, update of nutrients and 4 phytoplankton]
$J(C^a) = \frac{1}{2}(C^a - C^b)^T B^{-1}(C^a - C^b) + \frac{1}{2}(y - H(C^a))^T R^{-1}(y - H(C^a))$
CMEMS products
Daily & monthly 3D fields of:
- chlorophyll
- total phytoplankton carbon biomass
- concentration of nitrate
- concentration of phosphate
- oxygen
- primary production
- pH (total scale)
- DIC
Daily & monthly 2D fields of:
- surface pCO2
- air-sea CO2 flux

*Figure2: OGSTM – BFM scheme.*

The OGSTM-BFM (Figure 2) is designed with a transport model based on the OPA system and a biogeochemical reactor featuring the Biogeochemical Flux Model (BFM), while 3DVarBio is the data assimilation scheme for the correction of phytoplankton functional type and nutrient (i.e., nitrate and phosphate) variables using surface chlorophyll from satellite observations and vertical profiles of chlorophyll and nitrate from BGC-Argo floats.

The OGSTM 4.0 transport model is a modified version of the OPA 8.1 transport model (Foujols et al., 2000), which resolves the advection, the vertical diffusion and the sinking terms of the tracers (biogeochemical variables). The meshgrid is based on 1/24° longitudinal scale factor and on 1/24°cos(φ) latitudinal scale factor. The vertical meshgrid accounts for 141 vertical z-levels (125 active in the Mediterranean domain): 35 in the first 200 m depth, 60 between 200 and 2000 m, 30 below 2000 m. The temporal scheme of OGSTM is an explicit forward time scheme for the advection and horizontal diffusion terms, whereas an implicit time step is adopted for the vertical diffusion.

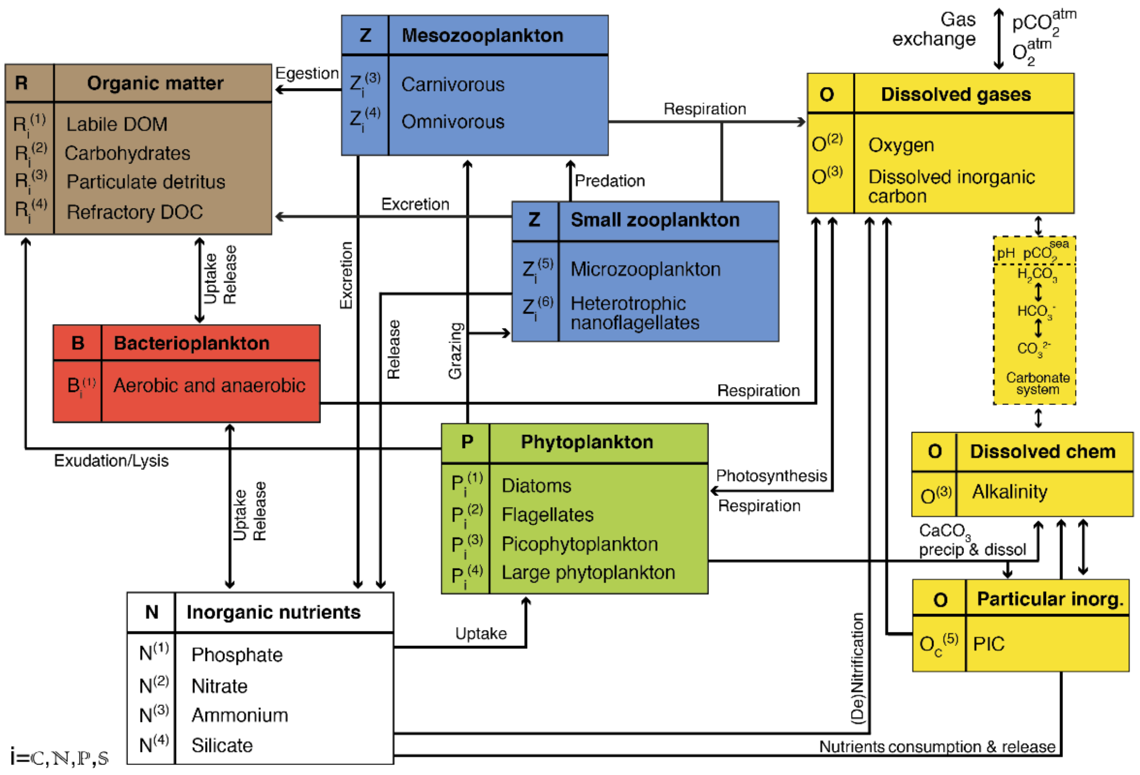


*Figure 3: BFM scheme.*

BFMv5 model (i.e., the official version released by www.bfm-community.eu) describes the biogeochemical cycles of 4 chemical compounds: carbon, nitrogen, phosphorus and silicon through the dissolved inorganic, living organic and non-living organic compartments (Figure 3).

The data assimilation of the surface chlorophyll concentration and of the vertical insitu profiles of chlorophyll and nitrate is performed through a variational scheme (3DVarBio) during the 7 days of analysis of the Tuesday run of Fig. 1 (see details on 3DVarBio in Teruzzi et al., 2014, 2018, 2019 and Cossarini et al., 2019). The operational workflow of the analysis run (the Tuesday row in Fig. 1) consists of a sequence of seven days of assimilation: the satellite surface chlorophyll map (i.e., a composite average in the range of ±3 days) is assimilated at 12:00 UTC of the previous Monday (i.e., day - 8) and the insitu vertical profiles of chlorophyll and nitrate are assimilated at 12:00 UTC from day -8 to day -2. A pre-processing quality control is applied prior of the assimilation

## 1.3 HPC configuration and setup

The model is written in Fortran (90 standard) and it is parallelized in distributed memory using MPI. With reference to the CMEMS analysis and forecasting product, in its standard configuration the model runs on 10 computing nodes of Galileo (CINECA), using a total of 360 cores (36 cores/node). Domain decomposition minimizes the land/water ratio for the MPI processes domain cells, in order to optimize the load balancing. An example of domain decomposition, for a smaller number of processes, is provided in fig. 4:

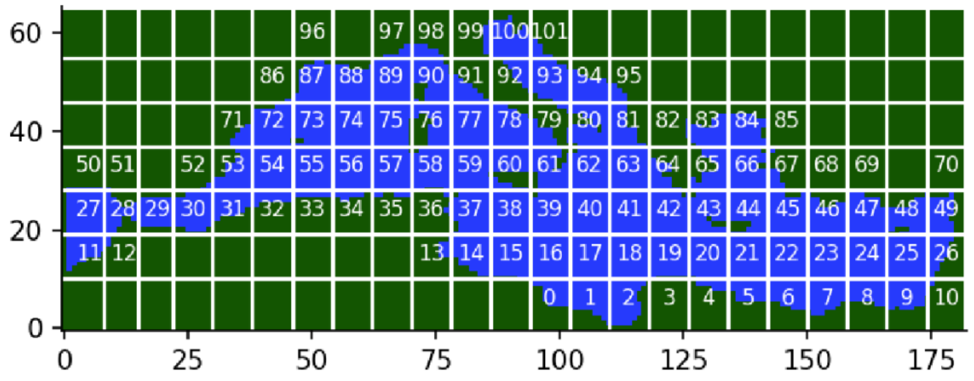


*Figure 4: Example of domain decomposition for 102 processes.*

All the tests have been done on Galileo supercomputer (CINECA). Galileo is an IBM NeXtScale cluster (architecture: Linux Infiniband cluster and Network: Intel OmniPath (100Gb/s) high-performance network). The nodes used are reserved nodes for OGS test and analysis. The nodes are of the type 1022 (Intel Broadwell), characterised by 2 x 18-cores Intel Xeon E5-2697 v4 at 2.30 GHz and 128 GB/node.

## 1.4 Workflow

To run OGSTM – BFM model we have different configurations that have different mesh grids. From here when is written ¼ mesh we are referring to a mesh grid based on 1/4° longitudinal scale factor and on 1/4°cos(φ) latitudinal scale factor (TEST 1/4), while 1/16 is referred to a mesh grid of 1/16° longitudinal scale factor and on 1/16°cos(φ) latitudinal scale factor (TEST 1/16) and 1/24 is referred to a mesh grid of 1/24° longitudinal scale factor and on 1/24°cos(φ) latitudinal scale factor (TEST 1/24). A more refined mesh grid leads to a longer computational time and to bigger output files. That is why we use the TEST 4 as implementing test case, with the more debugging work. After that we went to the TEST 16 and finally to the TEST24. The general workflow is depicted in figure 5.

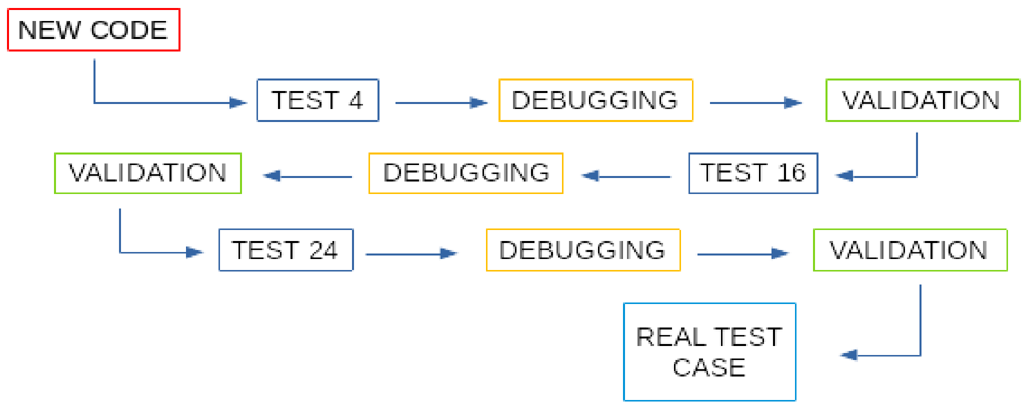


*Figure 5: Project workflow scheme.*

In the everyday work, TEST 4 is used for debugging while the other two are for production. As we had to measure the performances of the code relates to the different test cases, TEST 4 was run with 1, 3 and 5 nodes, TEST 16 and TEST 24 with 5, 10 and 20 nodes each.

| | 1 NODE | 3 NODES | 5 NODES | 10 NODES | 20 NODES |
|---|---|---|---|---|---|
| TEST 4 | X | X | X | | |
| TEST 16 | | | X | X | X |
| TEST 24 | | | X | X | X |
| PRODUCTION CASE | | | | | X |

*Table 1: Scheme of number of Nodes used to run tests.*

To validate the output results we use the *md5sum* function. Before all the tests we have run the old code (code before the thesis) and we have stored the results. As the *md5sum* function gives a unique alphanumeric string to each single file, similar to a digital fingerprint, if the new output files have the same *md5sum* string as the old code files, they are identical: through the new code we are obtaining the same results as the previous code.

## 1.5 Model outputs

OGSTM – BFM is a complex numerical model characterised by different output and different types of files that has to be dumped.

The output core is defined by the output variables of the model, i.e. the results of each run. Inside the model there are two main groups of variables:

- Passive tracer: 51 biogeochemical variables that the model needs to know to evolve; they are the basis of the model and are defined as state variables.

- Diagnostic variables: 122 biogeochemical variables that the model calculates during the run (101 3d-variables + 11 2d-variables).

To run the simulations, the end user needs to pass as input all the 51 passive tracers and can define which output variables wants to obtain (obviously state variables are mandatory).

Output is categorised in different group:

- RESTARTS: state variable mandatory for the functionality of the model plus some diagnostics; when restarts are written, 51 double precision state variable plus 51 double precision backups average values of state variables plus 13 double precision diagnostic variables are saved.

- AVE FREQ 1 variables: 17 single precision state variables plus 13 single precision diagnostic variables are saved in high frequency: it means that their saving happens more frequently during the run (that was implemented to save time in the output parts: to monitor the model we don't need to dump all state variables, but we can choose which ones).

- AVE FREQ 2 variables: 51 single precision state variables plus 13 single precision diagnostic variables. We are saving all state variables plus diagnostic variables we want as output.

- DA variables: data assimilation variables that the process of data assimilation needs to work.

In the table below, output savings for one-year production reanalysis run at 1/24 are defined.

| | n° times/year saving | N° D precision state vars | N°D precision BKP vars | N° D precision diag. vars | N° S precision state vars | N° S precision diag. vars | N° S precision Da vars |
|---|---|---|---|---|---|---|---|
| RESTARTS | 36 | 51 | 51 | 13 | | | |
| AVE 1 | 365 | | | | 21 | 13 | |
| AVE 2 | 53 | | | | 51 | 13 | |
| DA | 53 | | | | | | 19 |

*Table 2: Output saving scheme for one-year production reanalysis run at 1/24 (S =single precision, D = double precision, vars = variables, diag. = diagnostic) .*

The dumping of each group of output variables is handled by a specific routine:

- *trcdit.f90*: dumps AVE 1 and AVE 2 state variables in single precision while AVE 2 backups state variables are dumped in double precision.

- *diadump.f90*: dumps diagnostic AVE 1 and AVE 2 variables in single precision plus backup diagnostic variables in double precision.

- *trcwri.f90*: dumps RESTARTS variables in double precision.

- *trcwri_DA.f90*: dumps variables for data assimilation in single precision.

- *trcdia.f90*: is the routine that controls the activation of the routines described before.

All the routines described before were redefined by this thesis.

# 2. OLD VERSION

## 2.1 Code analysis

Before analysing the code, it is important to define how variables were stored during computing time. Each processor stored its part of data variables in the node local memory. To dump variables, all processor parts had to be put together to create the final variable matrix that had to be dumped. So, inside each processor local memory is stored a matrix which dimension is Nvariables * local_i * local_j *z, where i is the latitude index, j is the longitude index and z is the depth index. As dimension are not the same for each processor, the local matrix is defined with dimension Nvariables * local_i_max * local_j_max * z to be sure that all data fit inside the matrix. Only the processor that will dump variables has defined in its local memory the I/O matrix of dimension Nvariables * i * j * z that can contain all data variables for the whole simulation grid.

The paradigm used to dump outputs was very simple: variables were dumped serially by the processor 0 that collect all data variables by variables from all other processors. All dumping routines were structured in the same way:

1) In the first part processor 0 implemented its local indexes and copy its own local matrix in the I/O matrix.

2) In the second part through a loop of dimension 1 to number of processes, processor 0 receives (*MPI_recv*) by each single process the local indexes of the process and its part of the matrix to copy inside the I/O matrix. Processor 0 unpacks data and copies them inside the I/O matrix.

3) In the third part is defined the procedure of the other processors: the packing of the data to send and the sending of the data (local indexes and local matrix) through *MPI_send*.

4) After all the collecting, the processor 0 dump the variable.

Code example (from *trcdit.f90*):

```
“ …
        DO jn=1,jptra !  # Master loop where jptra is the number of variables
*************** START COLLECTING DATA ****************************
**PART 1
        if(myrank == 0) then
******* myrank 0 sets indexes of tot matrix where to place its own part************
        iPd     = nldi
        iPe     = nlei
        Pd     = nldj
```

```
…..
***** START ASSEMBLING ***  myrank 0 puts its tracer part in the tot matrix******
tottrnIO(:,totjstart:totjend,totistart:totiend)=
traIO_HIGH(:,reljstart:reljend,relistart:reliend,jn_high)
….
do idrank = 1,mpi_glcomm_size-1
```
**PART 2**
```
! **************  myrank 0 is receiving from the others their buffer  ****
           call  MPI_RECV(jpi_rec,1,mpi_integer,idrank,  1,mpi_comm_world,  status,  ierr)  !*  first  info
to know where idrank is working
           call MPI_RECV(jpj_rec,1,mpi_integer, idrank, 2,mpi_comm_world, status, ierr)
…..
******* myrank 0 sets indexes of tot matrix where to place buffers of idrank
        irange    = iPe - iPd + 1
        jrange    = jPe - jPd + 1
**** ASSEMBLING *** myrank 0 puts in tot matrix buffer received by idrank
        do ji =totistart,totiend
        i_contribution   = jpk*jpj_rec*(ji-1-totistart+ relistart)
        do jj =totjstart,totjend
        j_contribution = jpk*(jj-1-totjstart+ reljstart)
        do jk =1, jpk
        ind = jk + j_contribution + i_contribution
        tottrnIO(jk,jj,ji)= bufftrn(ind)
….
```
**PART 3**
```
else  ! IF LABEL 1,  if(myrank == 0)
! **** work of the other ranks
! ****** 1. load  inf buffer their IO matrices
        if (FREQ_GROUP.eq.2) then
        do ji =1 , jpi
        i_contribution= jpk*jpj * (ji - 1 )
        do jj =1 , jpj
        j_contribution=jpk*(jj-1)
        do jk =1 , jpk
        ind =  jk + j_contribution + i_contribution
        bufftrn   (ind)= traIO( jk,jj,ji,jn)
….
! ******  2.send buffer to myrank 0
        call MPI_SEND(jpi  , 1,mpi_integer, 0, 1, mpi_comm_world,ierr)
        call MPI_SEND(jpj  , 1,mpi_integer, 0, 2, mpi_comm_world,ierr)
……
************* END COLLECTING DATA  *****************
```
**PART 4**
```
! *********** START WRITING **************************
 if(myrank == 0) then ! IF LABEL 4
        if (IsBackup) then
        CALL      WRITE_AVE_BKP(bkpname,var,datefrom,      dateTo,tottrnIO,ave_counter,      deflate_ave,
deflate_level_ave)
        else
…“
```

## 2.2 Profiling

Hereafter are reported the time for each routine single call (time is given by the sum of writing time, communication time and calculation time). Routine time is correlated to the number of variables to dump and to their precision as reported in the table.

| | N° | Double precision (sec) | Single precision (sec) |
|---|---|---|---|
| *trcdit.f90* | 21 | | 11 .3 |
| | 51 | 42.1 | 26 .2 |
| *diadump.f90* | 13 | 11.1 | 6.5 |
| *trcwri,f90* | 51 | 42,7 | |
| *trcwri_DA.f90* | 19 | 15.90 | 9.7 |

*Table 3: I/O routines timing.*

Hereafter is reported the pie plot that shows the total I/O time of the model for one year reanalysis 1/24 simulation.

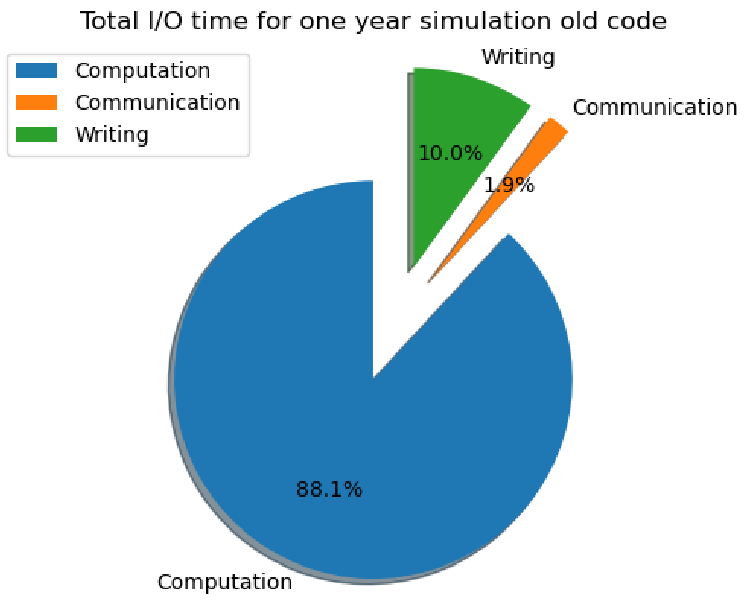


*Figure 6: Total I/O time of the model for one-year reanalysis 1/24 simulation.*

## 2.3 Issues

It is possible to see from the code how in each routine there are some parts that are repeated many times. Position indexes are proper of each process and are defined in the starting part of the model. Then we have a lot of *Mpi_send* and *Mpi_receive* that are not optimised.

Despite these issues I/O time seems not to be so critical at the moment as I/O is about 15% of total simulation timing. But future project needs will take to a dramatic situation: we will need to save variables every two hours at least for AVE FREQ 1 and daily for what concerns AVE FREQ 2 leading to the situation exposed in the graph below.

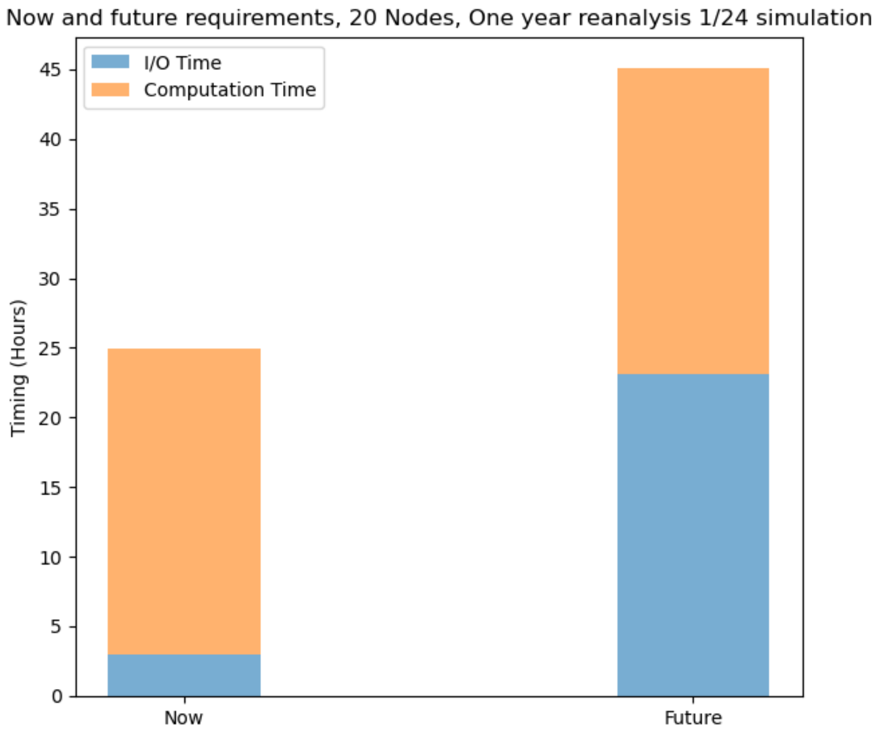


*Figure 7: Reanalysis 1/24 simulation timing, now and future.*

It is possible to see how I/O time will be greater than computational time.

# 3. NEW I/O PARADIGM

## 3.1 Initial optimisation

First of all, we started from trying to optimise the code. We wrote a new routine, *mpi_gatherv_info.f90*, that initially trace all the position indexes of each processors and through different *Mpi_gather* send all of them to rank 0, I.e. is the writing rank. This operation has the effect to clean the code from repeated parts and take some advantages in communication time.Inside each I/O routine we have a series of *MPI_send* (from all processors) and *MPI_receive* (in the rank 0) used to send the variables local matrix to the rank 0 for the dumping of variables. We substituted this amount of *MPI_send/receive* with a single *MPI_gather* that is better optimised. At this stage we didn't touch the routine structures.

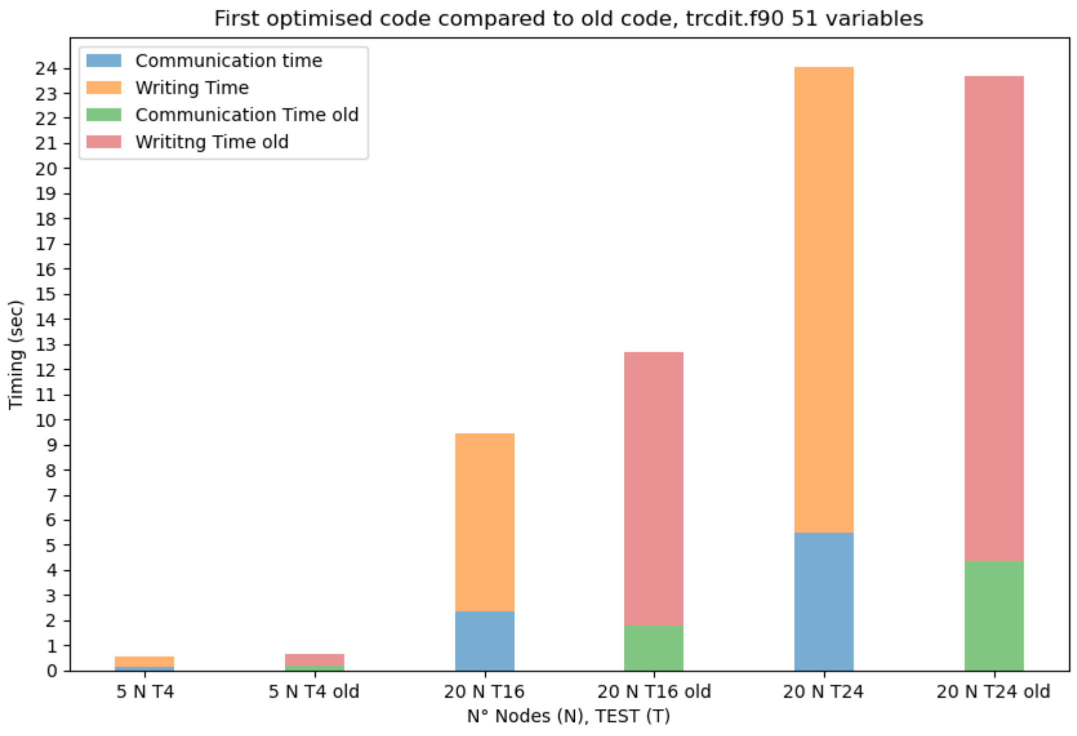


*Figure 7: First optimisation results compared to old code.*

As it is possible to see in figure 7, this first optimisation led to a good result for what concerns simulations at the lowest level of mesh grid accuracy, while for real simulation case as TEST 24 didn’t take any advantage. Timing was measured for *trcdit.f90* that is the more demanding routine as it  dumps 51 variables. We didn’t take any advantages as the amount of data to exchange was heavy ans as the *MPI_gather* exchange a fix amount of data: we are exchanging more data then we needed.

## 3.2 New I/O, how?

All these considerations and results led to the definition of a new I/O paradigm.

The new paradigm relies on the possibility that the file system support parallel writing from different nodes. So the idea was to distribute variables to dump to each node. But how?

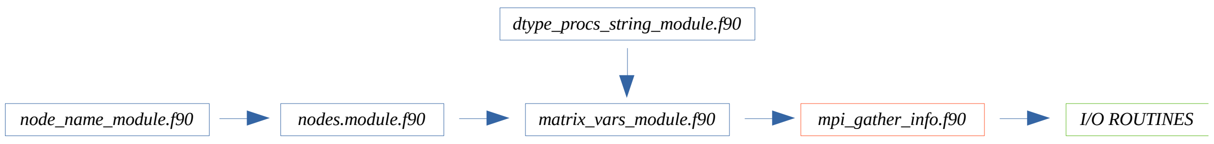


*Figure 8: Pre I/O-routines workflow.*

### 3.2.1 Pre I/O modules

The new system has to be machine independent and plastic: it has to adapt to I/O needs of different end users. So first of all we implemented a new module called *nodes_module.f9*0 that defines how many nodes are used for the ongoing simulation, which are the nodes and how many processors per node are used. All processors run a *node_name_module.f90* module that defines the name of the node they are part of. After that in the first part of the *nodes_module.f9*0 through an *MPI_gather* all processors send to processor 0 their local array containing the name of the node. As MPI processes are in order of rank, processor 0 will have all the situation mapped. Processor 0 working on the total array of names defines how many nodes are used for the simulation and how processors are distributed between nodes. After that, processor 0 defines an array of integers that collect the first processor of each node: that will be the writing process of each node. So as it is ordered through MPI processes, the first slot of the array contains the writing processor of nodes 1, the second slot contains the writing procs of the second node and so on. After that mapping processor 0 sends the two arrays (arrays of nodes and array of writing procs) to all processors. From this point all processors know how many nodes are involved in the model run and which is the writing processor of each node.

From *dtype_procs_string_module.f90*:

```
“module dtype_procs_string_module
  TYPE processor_string
     CHARACTER (LEN = 20) :: var_name
  END TYPE processor_string
end module dtype_procs_string_module”
```

From *node_name_module.f90*:

```
“...
 INTEGER :: lengt
        CHARACTER*(MPI_MAX_PROCESSOR_NAME) local_array
        CONTAINS
!------------------------------------------------------------------------------
        SUBROUTINE NODE_NAME_FILL()
        INTEGER :: IERROR
        CALL MPI_GET_PROCESSOR_NAME(local_array, lengt, IERROR)
        END SUBROUTINE NODE_NAME_FILL
…”
```

From *nodes_module.f90*:

```
“…
CALL MPI_GATHER(local_array, lengt,MPI_CHAR, total_array,lengt,MPI_CHAR, 0, MPI_COMM_WORLD,
IERROR)
***rank 0 define how many nodes are used
        IF (myrank == 0) THEN
                nodes = 1
                p=1
                k=2
                DO i=2, mpi_glcomm_size
                        IF (i==1) THEN
                               ! write(*,*)
                        END IF
                        DO j=1, i
                                IF (total_array(i) == total_array(j)) THEN
                                        EXIT
                                END IF
                        END DO
                        IF (i==j) THEN
                                nodes = nodes + 1
                        END IF
                END DO
…
DO i=1, mpi_glcomm_size - 1
   CALL MPI_Send(nodes,1,MPI_INT,i,4,MPI_COMM_WORLD,IERROR)
END DO
…
```

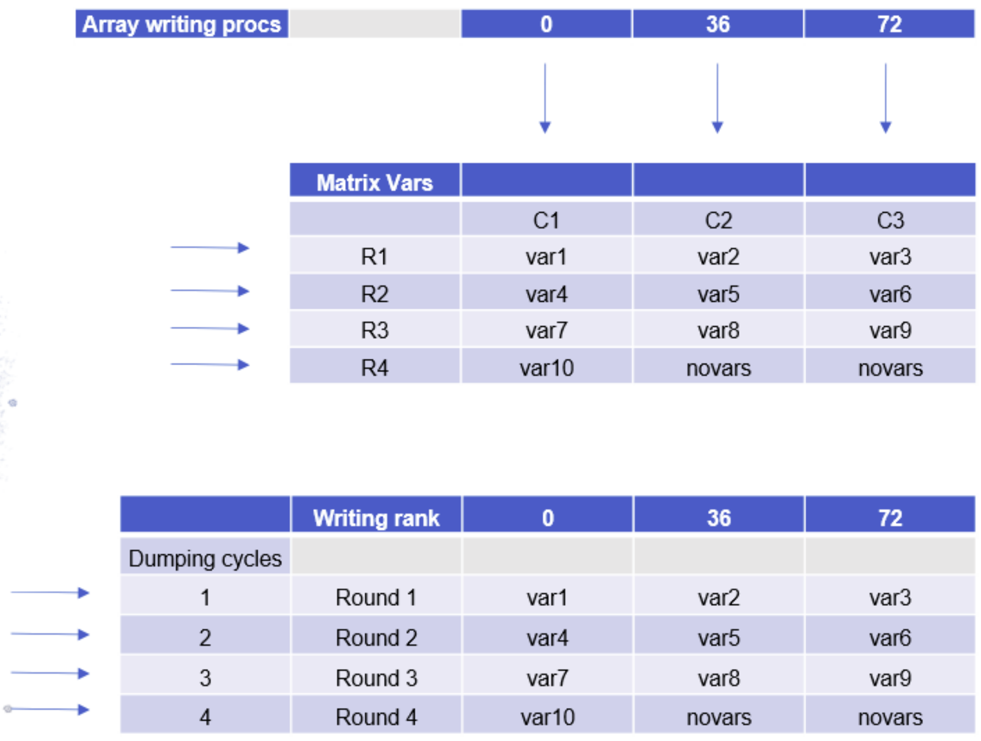
*****rank 0 define the number and who are writing procs**

```
 writing_procs(1) = 0
 DO i=2, mpi_glcomm_size
 IF (nodes==1) THEN
!if the number of node is one break, no sense to calculate, avoid problems
  EXIT
 ELSE IF (total_array(p) /= total_array(i)) THEN
 writing_procs(k)= i-1
 p=i
 k=k+1
```

```
ELSE
CYCLE
END IF
END DO
!rank 0 send to all ranks writing_procs
DO i=1, mpi_glcomm_size - 1
CALL MPI_Send(writing_procs,nodes,MPI_INT,i,3,MPI_COMM_WORLD,IERROR)
                END DO
…
 IF (myrank >0) THEN
  CALL MPI_Recv(nodes,1,MPI_INT,0,4,MPI_COMM_WORLD,MPI_STATUS_IGNORE, IERROR)
  ALLOCATE(writing_procs(nodes))
 CALL MPI_Recv(writing_procs,nodes,MPI_INT,0,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE, IERROR)
 DO k=1, nodes
write (*,*) 'writing procs position is ', k, writing_procs(k)
 END DO
 END IF
…”
```

Now we have to define which are the variables that will be dumped. We defined a *matrix_vars_module.f90* module inside which through different routines are defined the matrices of variables to dump according to a precise scheme. First of all, different string matrix are defined, as we have different type of variables in different processes, and each matrix will contain the name of variables to be dumped (we will have a matrix for variables in high freq, one for variables in low freq, one for diagnostic variables and so on). We have defined through a *dtype_procs_string_module.f90* module a dtype of fix dimension that will contain the name of variables. So, matrices will be filled with string of character dtypes. The common scheme to define the matrix is the same: first are calculated the matrix dimension. Rows are defined through the division of the total number of variables to dump and the number of nodes: if the division module is different from 0, we add a row to the total n umber of rows. Column numbers is related to the number of nodes: there will be as column as the number of nodes and therefore the number of writing processors (figure 9). The idea is simple, each number of column is related to the number of writing processor: column number one is referred to the first slot of the writing array (contains all ranks of writing processors), so first column referred to processor 0 of the first node, second column referred to processor 0 of second node and so on. It is a simple way of defining which processor will dump the variable, but it is effective and easy to control.

After that, we populate matrix with a string of type "novars_input" that will be useful in the future double check when variables will be dumped. After this last operation we populate matrices with the name of variables to dump. To know the name of each variable associated we have to pass through a frequency table that map the number of variables and its name related to an input file of the model (*namelist.passivetrc*) in which the end user defines how many variables will be used in the model run, which of them will be state variables and diagnostic variables and which will be high and low frequency. So at the ending of the *matrix_vars_module* all processors have stored the number of nodes, writing processors and all the matrices of variables to dump.



| Array writing procs | | 0 | 36 | 72 |
|---|---|---|---|---|

| Matrix Vars | | | |
|---|---|---|---|
| | C1 | C2 | C3 |
| R1 | var1 | var2 | var3 |
| R2 | var4 | var5 | var6 |
| R3 | var7 | var8 | var9 |
| R4 | var10 | novars | novars |

| | Writing rank | 0 | 36 | 72 |
|---|---|---|---|---|
| Dumping cycles | | | | |
| 1 | Round 1 | var1 | var2 | var3 |
| 2 | Round 2 | var4 | var5 | var6 |
| 3 | Round 3 | var7 | var8 | var9 |
| 4 | Round 4 | var10 | novars | novars |

*Figure 9: Matrix variables and array writing processor example (3 Nodes, with one wr procs. Per node, 10 variables to dump).*

From *matrix_vars_module.f90*:

“…

```
IF (MOD(jptra,nodes)==0)THEN
                matrix_state_2_row = (jptra/nodes)
        ELSE
                matrix_state_2_row = (jptra/nodes) + 1
        END IF
…
DO i=1,matrix_state_2_row
                DO j=1,matrix_col
                        matrix_state_2(i,j)%var_name = novars
                END DO
        END DO
…
DO i=1,matrix_state_2_row
                DO j=1,matrix_col
                        IF (counter==jptra)THEN
                                EXIT
                        ELSE
                                matrix_state_2(i,j)%var_name = ctrcnm(counter+1)
                                counter=counter + 1
                        END IF
                END DO
        END DO
```

```
…
SUBROUTINE DIA_MATRIX_VARS()

        INTEGER :: i

        !high freq dia 3d
        jptra_dia_high_wri = 0
        DO i =1, jptra_dia
                IF (diahf(i).eq.1 .and. diaWR(i) == 1) then
                        jptra_dia_high_wri = jptra_dia_high_wri + 1
                END IF
        ENDDO

        ALLOCATE (highfreq_table_dia_wri(jptra_dia_high_wri))

        jptra_dia_high_wri = 0
        DO i =1, jptra_dia
                IF (diahf(i).eq.1 .and. diaWR(i) == 1) then
                        jptra_dia_high_wri = jptra_dia_high_wri + 1
                        highfreq_table_dia_wri(jptra_dia_high_wri) = i
                END IF
        END DO
…"
```

At this level assignment processes are completed. The model now runs the *mpi_gather_info.f90* module an all its routines. Inside this module are implemented all the routines that are fundamental for the new I/O implementation: first the allocation part of all the array and buffers that will be used for the parallel part. Then the definition of a boolean WRITING_RANK_WR for all processes that is initialised to false. After a quick loop, every rank compared his number with the integers stored in the writing processor array: if the the rank is included in the ones of the array the boolean is set to true. This simple loop permits to activate which processes are the writing ones. At this point through a series of *MPI_gather*, all processes send to writing ones all the informations they need to dump variables: local indexes, proper of each rank, are stored inside each writing processor. Now all writing processors can define how many data will be transferred from each processor calculating the *sendount* and *jpdispl_count* of each processor data transfer, and then storing them in dedicated total arrays.

From *mpi_gather_info.f90*:

```
"…
LOGICAL :: WRITING_RANK_WR
SUBROUTINE INIT_MPI_GATHER_INFO()
        WRITING_RANK_WR = .FALSE.
        CALL ALLOCATE_MPI_GATHER_INFO()
        !gather(send+recv from each rank, stored in array of each indices)
        call mppsync()
        DO wr_procs=1, nodes
```

```
                CALL    MPI_GATHER(    jpi,    1,    MPI_INTEGER,    jpi_rec_a,1,MPI_INTEGER,
writing_procs(wr_procs), MPI_COMM_WORLD, IERROR)
                CALL   MPI_GATHER(   jpj,   1,   MPI_INTEGER,   jpj_rec_a,   1,MPI_INTEGER,
writing_procs(wr_procs), MPI_COMM_WORLD, IERROR)
                CALL   MPI_GATHER(   nimpp,   1,   MPI_INTEGER,   istart_a,   1,MPI_INTEGER,
writing_procs(wr_procs), MPI_COMM_WORLD, IERROR)
…
sendcount = jpi * jpj * jpk
        sendcount_2d = jpi * jpj
        if(WRITING_RANK_WR)then
                cont = 0
                DO loop_ind = 1, mpi_glcomm_size
                        jprcv_count(loop_ind) = jpi_rec_a(loop_ind) * jpj_rec_a(loop_ind) *
jpk
                        jpdispl_count(loop_ind) = cont
                        cont = cont + jprcv_count(loop_ind)
                end DO
…"
```

Making a recap, at this point of the run, all writing processors have stored all the information they need for the I/O parallel routine, while all processes has stored the information of variables and writing processors.

### 3.2.2 I/O routines

I/O routines have been modified with the same structure. They have been implemented in two main parts. The first part is the one that is run by all processors while the second one is related to writing processors only. I/O routines have a main loop that is related to the number of cycling dump. Each row of a variables matrix is an I/O cycle dump. So the main architecture has an outer loop that goes from one to the number of rows (n° of dumping cycles) and an inner loop that goes from one to the number of variables to dump (number of columns = number of nodes = number of writing processor).

The first part is divided in two parts too: the first part is the loop through which all processors pack their data from local matrix to a buffer that will be send by an *MPI_gatherv* (second part) to the respective writing processor. The packing is different if a a variable belongs to high or low frequency group. The *Mpi_gatherv* is the same for all 3d variables, while it has different data for 2d diagnostic variables: all these data comes from the before *mpi_gather_info module*. All buffers are collected in a bigger one in the referred writing rank.

After the collecting part (part 1) there is the code block proper of writing processors. Before dumping the variable, they have to unpack the buffer that contains all local buffers of all processors into a 3d matrix that reflect the initial domain decomposition. Then the writing procedure will be call. All matrix variables are written in netcdf 4, and while backups and restarts are written in double precision, all other outputs are casted to single precision to save time and space.

Now it is possible to understand why we call it a parallel I/O paradigm, as we don't have the classical parallel writing, in which a defined number of processors dump down its own part of the same file, but we have a parallel dumping of complete variables by each writing processor. E.g.: if we have 20 nodes, with 1 writing processor per node, we have for each dumping loop 20 processor that dump down 20 variables (one each) contemporary: that permits to scale the writing time as will be shown in the next pages.

Below, the used code is reported from the *trcdit.f90*.

```
"…
**FIRST PART
!!ALL PROCESSORS
DUMPING_LOOP: DO jv = 1, n_dumping_cycles
 DO ivar = 1 , nodes  !number of variables for each round corresponds to the number of nodes
 writing_rank = writing_procs(ivar)
!!PACKING PART
 IF (COUNTER_VAR > JPTRA)then
  EXIT
 else if (COUNTER_VAR_HIGH > JPTRA_HIGH)then
  EXIT
 ELSE
  if (FREQ_GROUP.eq.2) then
  do ji =1 , jpi
   i_contribution= jpk*jpj * (ji - 1 )
    do jj =1 , jpj
     j_contribution=jpk*(jj-1)
     do jk =1 , jpk
      ind =  jk + j_contribution + i_contribution
       bufftrn (ind)= traIO( jk,jj,ji,counter_var)
     enddo
    enddo
  enddo
 else ! FREQ_GROUP.eq.1
...
 counter_var = counter_var + 1
  if (FREQ_GROUP.eq.1) counter_var_high = counter_var_high + 1
 !GATHERV TO THE WRITING RANK
  CALL  MPI_GATHERV(bufftrn,  sendcount,  MPI_DOUBLE_PRECISION,  bufftrn_TOT,  jprcv_count, jpdispl_count, MPI_DOUBLE_PRECISION, writing_rank, MPI_COMM_WORLD, IERR)
  END IF
 END DO
**PART 2
! *************** COLLECTING DATA *****************************
 IF (WRITING_RANK_WR)then
  writing_rank_init_time = MPI_Wtime()
  ind_col = (myrank / n_ranks_per_node)+1
  if (FREQ_GROUP.eq.2) then
   var_to_store = matrix_state_2(jv,ind_col)%var_name
  else
   var_to_store = matrix_state_1(jv,ind_col)%var_name
  end if
  IF (var_to_store == "novars_input")then
   EXIT
  ELSE
 DO idrank = 0,mpi_glcomm_size-1
  ! ******* WRITING RANK sets indexes of tot matrix where to place buffers of idrank
  irange    = iPe_a(idrank+1) - iPd_a(idrank+1) + 1
  jrange    = jPe_a(idrank+1) - jPd_a(idrank+1) + 1
  totistart = istart_a(idrank+1) + iPd_a(idrank+1) - 1
  totiend   = totistart + irange - 1
  totjstart = jstart_a(idrank+1) + jPd_a(idrank+1) - 1
  totjend   = totjstart + jrange - 1
  relistart = 1 + iPd_a(idrank+1) - 1
  reliend   = relistart + irange - 1
  reljstart = 1 + jPd_a(idrank+1) - 1
  reljend   = reljstart + jrange - 1
  ! **** ASSEMBLING *** WRITING RANK  puts in tot matrix buffer received by idrank
  do ji =totistart,totiend
   i_contribution   = jpk*jpj_rec_a(idrank+1)*(ji-1-totistart+ relistart)
   j_contribution = jpk*(jj-1-totjstart+ reljstart)
   do jk =1, jpk
    ind = jk + j_contribution + i_contribution
    tottrnIO(jk,jj,ji)= bufftrn_TOT(ind+jpdispl_count(idrank+1))
    enddo
   enddo
  enddo
 END DO
 output_file_nc = DIR//'ave.'//datemean//'.'//trim(var_to_store)//'.nc'
 bkpname = DIR//'ave.'//datemean//'.'//trim(var_to_store)//'.nc.bkp'
  if (IsBackup) then
  CALL      WRITE_AVE_BKP(bkpname,var_to_store,datefrom,      dateTo,tottrnIO,elapsed_time,
deflate_ave, deflate_level_ave)
   else
 CALL   WRITE_AVE(output_file_nc,var_to_store,datefrom,   dateTo,   tottrnIO,   deflate_ave,
deflate_level_ave)
  endif
 END IF
 END IF
END DO DUMPING_LOOP
...”
```

As it is possible to see in the code, there are two points of control in the code to avoid the possibility to go outnumber of variables to dump and to write void files. The first control is made by counting all the inner loops, to be sure that variables that will be sent are in the correct number (`IF (COUNTER_VAR > JPTRA) EXIT`). The second control is made at writing processors level. The control is made by checking that the variable name is not equal to the string “*no_vars_to store*” (`IF (var_to_store == "novars_input")then EXIT`) that was initialised in the previous routine: it avoids the possibility of writing void files.

All I/O routines are structured as the one shown before.

### 3.2.3 Writing multiprocessors per node.

As we have declared in the previous section, the new I/O structure relies on the fact that the file system can handle parallel writing by all nodes contemporary. Till this moment we have spoken for a single writing processor per node. As we wanted to test all possibilities we have upgraded the code to manage multiprocessors writing per node. The main differences are in the *nodes_module .f90* and in the *matrix_vars_module.f90*. To avoid lots of changes in the code, we have maintained the *writing_procs* array as the final array that has stored the rank of all writing nodes. We have changed the previous *writing_procs* array in a *writing_procs_base* array and we have add a loop to define the new *writing_procs* array with multiprocessors per node, simply declaring a bigger array (array dimension = nodes * n* of procs per node) and adding the new ranks after the first one of each node. To make tests we have decided to have all the processors in the same socket inside each node. After that we have corrected all the code parts in which we have loop regarding nodes number, from nodes to *nodes*n°of procs per node*. To better handle the number of processor per node, the declaration is not hard coded but defined into an input file (*namelist.init*).

From *nodes_module.f90:*

```
"…
 writing_procs_base(1) = 0
                DO i=2, mpi_glcomm_size
                        IF (nodes==1) THEN
                                !if the number of node is one break, no sense to calculate,
avoid problems
                                EXIT

                        ELSE IF (total_array(p) /= total_array(i)) THEN
                                writing_procs_base(k)= i-1
                                p=i
                                k=k+1
                        ELSE
                                CYCLE
                        END IF
                END DO
…
if (num_of_wr_procs_perNODE == 1) then
                        ALLOCATE(writing_procs(nodes))
                        do i=1, nodes
                                writing_procs(i) = writing_procs_base(i)
                        end do
                else
                        ALLOCATE (writing_procs(nodes*num_of_wr_procs_perNODE))
                        cycle_cont = 1
                        DO i=1,nodes
                                IF (nodes==1) THEN
                                        !if the number of node is one break, no sense to
                                        !calculate, avoid problems
                                        EXIT
                                ELSE
                                        tot_cycle_cont = i - 1
                                        do j=1, num_of_wr_procs_perNODE
                                                if(j<(num_of_wr_procs_perNODE/2) + 1)then
                                                        writing_procs(cycle_cont)         =
writing_procs_base(i) + j - 1
                                                        cycle_cont = cycle_cont + 1
                                                else
                                                        writing_procs(cycle_cont)         =
writing_procs_base(i) + j - (num_of_wr_procs_perNODE/2) + 19
                                                        cycle_cont = cycle_cont + 1
                                                end if
                                                !tot_cycle_cont = tot_cycle_cont + 1
                                        end do
                                END IF
                        END DO
                end if
…”
```

From *matrix_vars_module.f90:*

“…

```
IF (MOD(jptra,nodes*num_of_wr_procs_perNODE)==0)THEN
                matrix_state_2_row = (jptra/(nodes*num_of_wr_procs_perNODE))
        ELSE
                matrix_state_2_row = (jptra/(nodes*num_of_wr_procs_perNODE)) + 1
        END IF
...”
```

From *trcdit.f90*:

“...

```
DUMPING_LOOP: DO jv = 1, n_dumping_cycles
                DO ivar = 1 , nodes*num_of_wr_procs_perNODE!
…
do i=1, nodes*num_of_wr_procs_perNODE
                                if(myrank == writing_procs(i))then
                                        ind_col=i
                                        !write(*,*)       'myrankis',       myrank,'indcol
is',ind_col,'writing_proc is', writing_procs(i),'iis', i, 'jvis', jv
                                        exit
                                end if
                        end do
…”
```

# 4. RESULTS AND IMPROVEMENTS

To show the obtained results we have decided to report *trcdit.f90* graphics both for single and double precision, referred to AVE_2 variables, as it is the more demanding routine, dumping 51 variables. To show general improvements we will show the graphics related to real production case at 1/24 mesh grid, that is the case in which before the thesis we would like to obtain the best results. Multiprocessor tests have been done for the TEST 24 only, with 2 and 4 writing ranks per node.

## 4.1 TEST 4

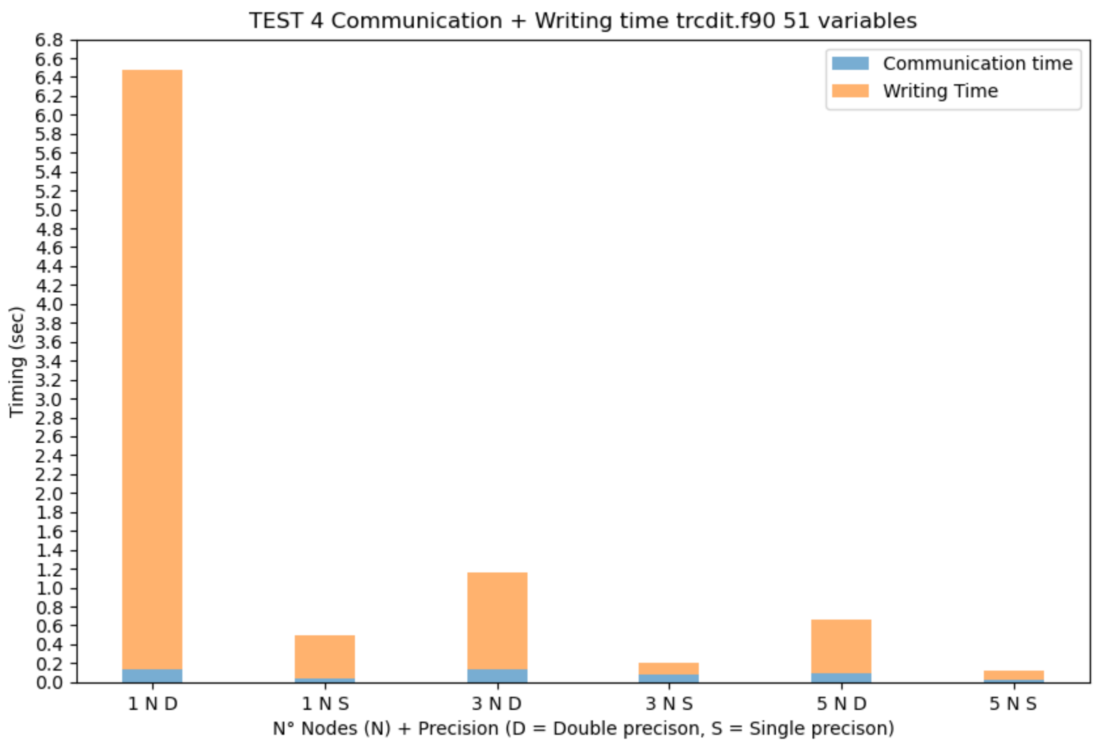


*Figure 10: TEST 4 results.*

It is possible to see how writing time decreases with the incrementing of the number of nodes. Writing in double precision is more costly in timing than writing in single precision. Communication time is similar in all the tests and decreases slightly with increasing number of nodes.

## 4.2 TEST 16

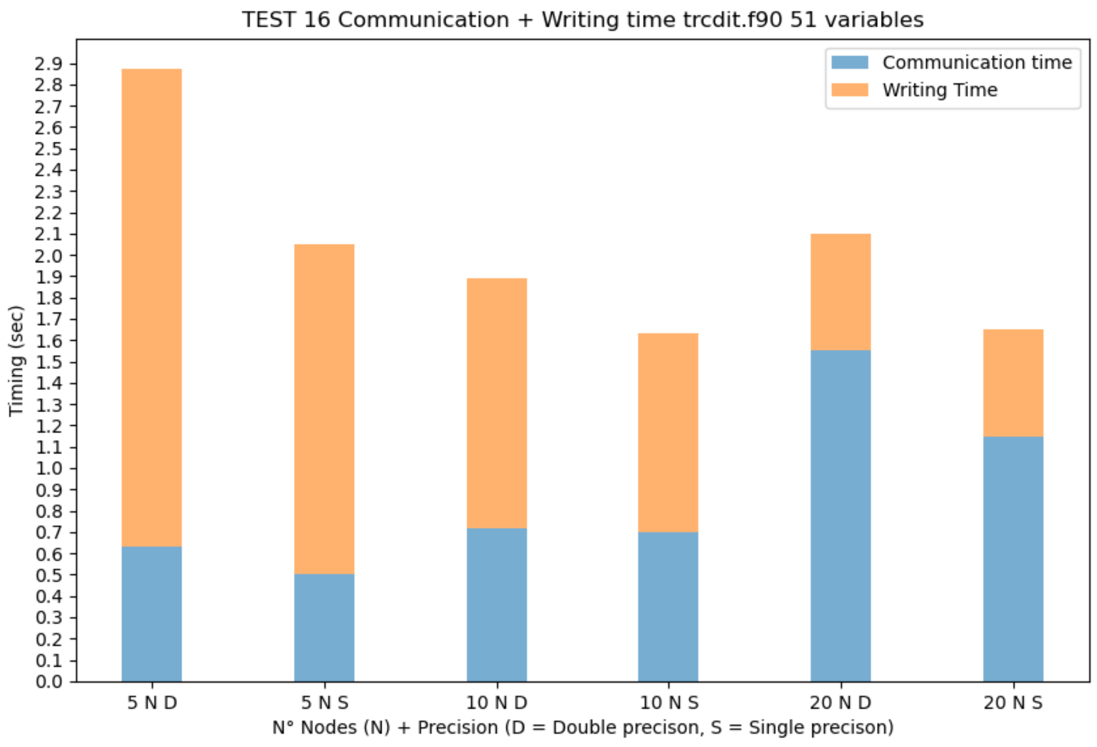


*Figure 11: TEST 16 results.*

In the TEST 16 the pattern is similar to TEST 4. Increasing the number of nodes leads to a decrease in the writing time but an increase of communication time. Output files are heavier, and this is reflected both in the writing and communication time. Obviously, double precision is more demanding than single precision.

## 4.3 TEST 24

TEST 24 is the most important from a production point of view. In the figures are reported both the cases for single and multiprocessors writing ranks per node.

For the single writing rank per node cases, it is possible to see the same pattern as the tests before. Increasing the number of nodes takes to an important decrease in writing time: time passes from 7.2 to 2.2 seconds for double precision and from 5.12 to 1.45 seconds for single precision. Communication time increases with the increasing of the number of nodes. In the heavier configuration, with 20 nodes, communication time is equivalent (double precision case) or greater (single precision time) then writing time, becoming a new issue to solve.

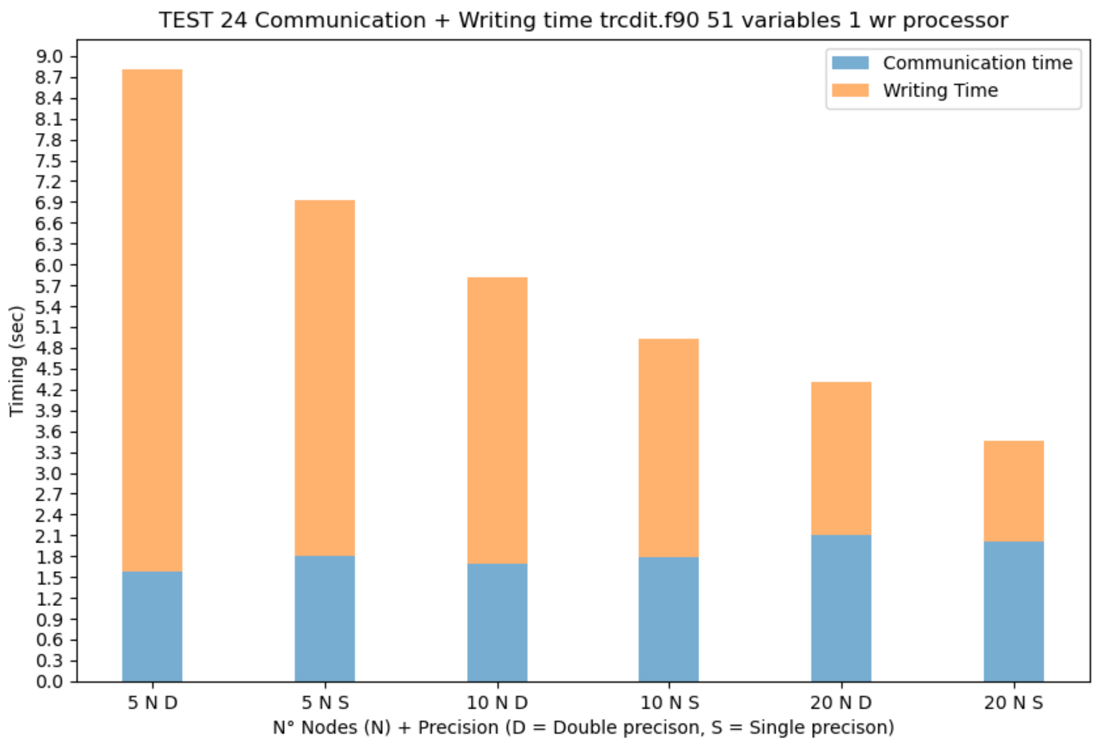


*Figure 12: TEST 24 1 writing processor per node results.*

For the multiprocessors cases the pattern is the same but communication time increases much more: it becomes 2 or 3 times bigger then writing time in the 20 nodes tests. This behaviour that was expected shows how it is not worthy to have more then one writing processors per node in the above cases. Time gained in dumping variables is lost with the increase of communication time. But, if we observe the numbers about the 10 nodes multiprocessor test cases compared to the similar 20 nodes case (20 nodes case with 1 writing processor per node is equivalent to the 10 nodes case with 2 writing processors per node), we can see that we gain in writing time with equivalent communication time.

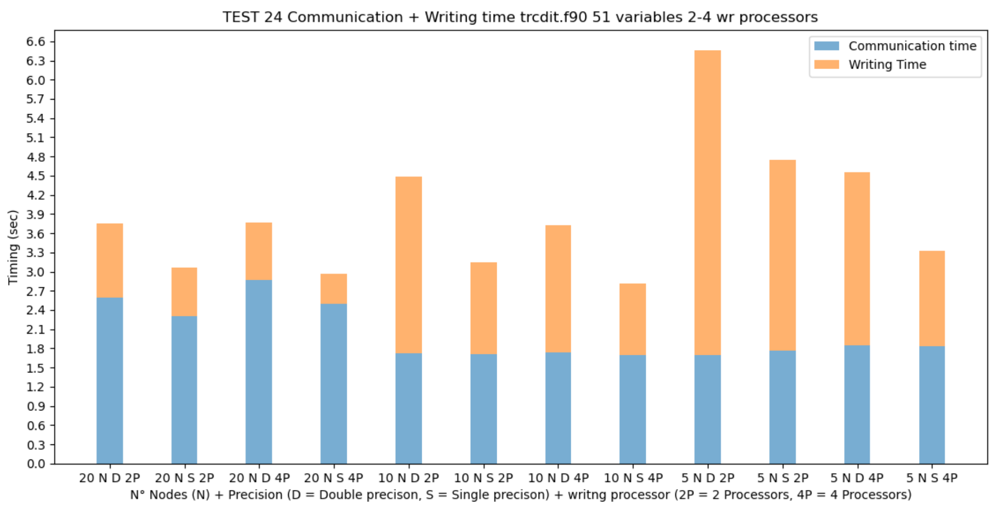


*Figure 13: TEST 24, 2 - 4 writing processors per node results.*

## 4.4 General Improvements

To have a better description of general improvements, we have calculated writing time speed ups and total speed ups. To obtain speed ups we have divided the old code timings by the new timings. For what concerns multiprocessor we have used the total number of nodes as reference.

Here after are reported the graphs for writing speed ups and total speed ups.

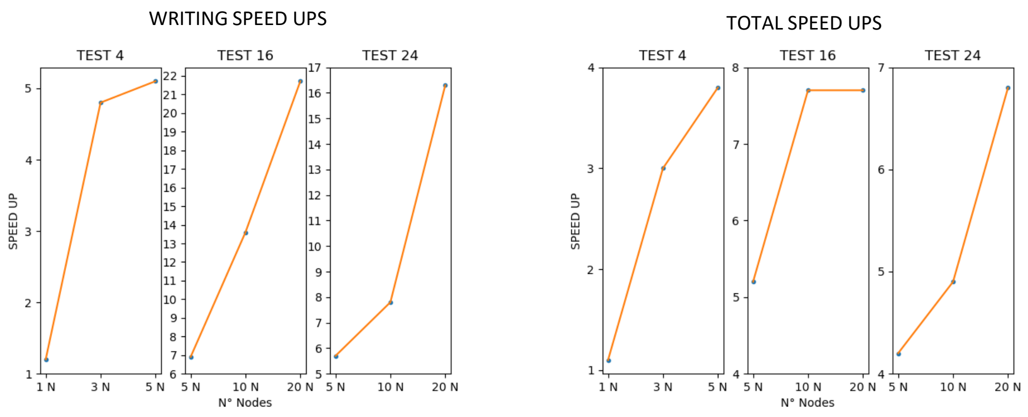


*Figure 14: Single processor speedups trcdit.f90, 51 variables.*

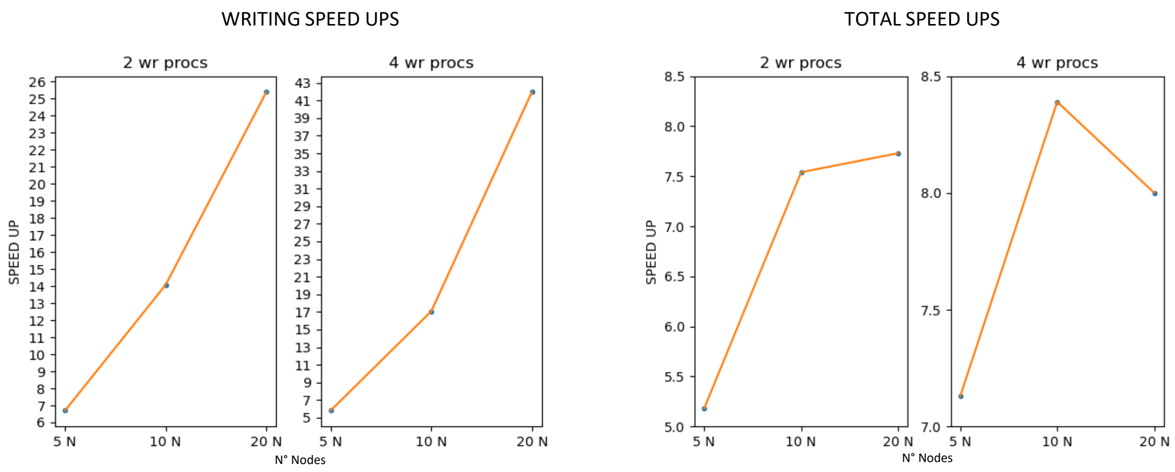


*Figure 15: Single processor speedups trcdit.f90, 51 variables*

Is it possible to see how the best total speed up was obtained by TEST 24 with 10 nodes and 4 processor per node. Similar values are obtained also for TEST 24 with 20 nodes and 2 processor per node and TEST 16 with 10 and 20 nodes. These results are obtained by the balance between a decreasing writing time and an increasing communication time. If we watch the tests from a data exchange point of view, we have done some test of weak scaling taking into account only the 5 nodes case. It is possible to see how increasing the output file dimension, the time scale from TEST 4 to TEST 16 but decrease with TEST 24.

## 4.5 Real test case

As all the new code went in production in the last months of the thesis, we would like to report some data. We decided to use the 20 nodes configuration with one writing processor per node, because it was the most solid from a new code point of view and because we need the greatest number of nodes for computational reasons. We are now reporting data for a reanalysis case ran with 20 nodes, for a simulation of one year. Timing is reported in seconds and is referred to the total time for all variables involved:

| | | | | Double precision | Double precision | Double precision | Single precision | Single precision | Single precision |
|---|---|---|---|---|---|---|---|---|---|
| | Type | Saving time | Annually | State variable | Average bkp vars | Diagnostic bkp | State variable | Diagnostic variable | 3D var ass |
| N° | Restarts | 3/month | | 51 | 51 | 13 | 0 | 0 | 0 |
| Writing | | | 36 | 1.68 | 2.2 | 0.43 | 0 | 0 | 0 |
| Communication | | | 36 | 1.83 | 3.83 | 0.39 | | | |
| Pak/unpack | | | 36 | 0.122 | 0.125 | 0.2 | | | |
| N° | Ave freq 1 | daily | | 0 | 0 | 0 | 21 | 13 | 0 |
| Writing | | | 365 | | | | 0.92 | 0.44 | |
| Communication | | | 365 | | | | 1.56 | 0.49 | |
| Pak/unpack | | | 365 | | | | 0.17 | 0.25 | |
| N° | Ave freq 2 | Weekly | | 0 | 0 | 0 | 51 | 13 | 0 |
| Writing | | | 53 | | | | 1.45 | 0.42 | |
| Communication | | | 53 | | | | 3.8 | 0.39 | |
| Pak/unpack | | | 53 | | | | 0.4 | 0.2 | |
| N° | DA restarts | Weekly | | 0 | 0 | 0 | 0 | 0 | 19 |
| Writing | | | 53 | | | | | | 0.54 |
| Communication | | | 53 | | | | | | 1.41 |
| Pak/unpack | | | 53 | | | | | | 0.14 |

*Table 4: 20 nodes 1/24 reanalysis simulation run partial timings.*

| RESTARTS | Type | Timing |
|---|---|---|
| | Writing | 155.16 |
| | Communication | 217.8 |
| | Pak/unpack | 16.092 |
| AVE FREQ 1 | | |
| | Writing | 496.4 |
| | Communication | 748.25 |
| | Pak/unpack | 153.3 |
| AVE FREQ 2 | | |
| | Writing | 99.11 |
| | Communication | 222.07 |
| | Pak/unpack | 31.8 |
| DA RST | | |
| | Writing | 28.62 |
| | Communication | 74.73 |
| | Pak/unpack | 7.42 |

*Table 5: 20 nodes 1/24 reanalysis simulation run total timings.*

We compared total timing of new code and old code respect to the reanalysis set-up and for the future requests. In the table all data are summarised.

In the case of the current reanalysis production, it is possible to see how I/O times decrease from 185.07 minutes to 37.51 minutes, taking the total amount time of simulation from 25 hours to 22.5. Considering that we were running with 20 nodes and 36 cores per node, we saved 1170 hours of computational time [ (185.07 – 37.51)/60 * 720] for one simulation year.

| | NEW CODE | OLD CODE |
|---|---|---|
| Writing | 779.29 | 9174.86 |
| Communication | 1262.85 | 1725.9 |
| Pak/unpack | 208.612 | 203.6 |
| Total Time in sec | 2250.752 | 11104.36 |
| Total Time in minutes | 37.51 | 185.07 |
| Speed up | 4.93 | |
| TIME Future requisites | 20410.702 | 87811.04 |
| Time in hours | 5.66 | 24.39 |

*Table 6: Comparing old code and new code total timings.*

Taking account of the future requisites from the Copernicus project, we have passed, theoretically, from 24.39 I/O hours to 5.66 saving 13.485 computational hours.

## 4.6 New paradigm issues and optimisation

After optimising the writing procedure, the new issue was the communication time. We know that our implementation has a weak point in the *MPI_gatherv* inside each I/O procedure. As a matter of fact *MPI_gatherv* are of blocking type: to make the next call each mpi call has to wait for the previous one. That was not a problem with lower number of nodes, but becomes an issue increasing the number of nodes. We didn’t focus on rewriting this part because after the implementation of the new code, a new bottleneck had to be solved: 3dvar optimisation. We will speak about this part in the next part of the present thesis.

Initially we were thinking to bufferize I/O and trying to do it asynchronously from computation time. We were thinking to I/O nodes, but bandwidth problems would rise. One possibility was the RAMFS, saving data into Ram and then creating a second routine that at run time will transfer data to the storage area of the cluster. We will see if these ideas will be possible and practicable.

# 5 DATA ASSIMILATION OPTIMISATION (3DVAR)

## 5.1 Overview

As explained in the thesis introduction part, assimilation (3dvar) is an important part of the model: it permits the model correction with real data taken from satellites and floaters. 3Dvar optimisation was the goal of MHPC 2015 – 2016 Pierluigi di Cerbo thesis, sponsored by OGS. 3Dvar is ran as a library inside the OGSTM - BFM model, as it was developed and tested as an independent part.

After the 2016 implementation Assimilation scheme was computed on 1 node. Optimisation studies showed that the best option was to run 3dvar with 22 processors because of memory bound problems. That set up was enough till this year. As a matter of fact in 2020 we started to use new floaters data. The main difference relies on the fact that new floaters have bigger depth levels leading to bigger files: we have memory bound problems again. We were forced to run 3dvar with 9 processors maximum to avoid problem of memory bound. Another issue was the fact that we have to run assimilation daily instead of weekly, leading to bigger computational time.

To solve the problem faster, we have thought to implement the same paradigm used for the I/O parallel distribution to the 3dvar scheme. In this moment we don't have time to implement a new 3dvar scheme, as it will take months. We applied the distribution scheme to 3dvar, not for output parts, as 3dvar an output part handled with PETSc libraries to write netcdf 4 files in parallel. We adjusted routines that handle the I/O parallel distribution for 3dvar.

## 5.2 Code analysis

We focused in the 3var routines that constructed the communicator that defined which rank were involved in the assimilation part.

Here after the code from *ogstm_mpi.f90*:

```
“...
SUBROUTINE mynode
      INTEGER :: ierr
#ifdef ExecDA
      PetscErrorCode :: stat
#endif
#ifdef key_mpp_mpi
      CALL mpi_comm_rank(mpi_comm_world,myrank,ierr)
      CALL mpi_comm_size(mpi_comm_world,mpi_glcomm_size,ierr)
      call parlec ! in order to read DA_Nprocs
#ifdef ExecDA
      if(myrank .lt. DA_Nprocs) then
        call MPI_Comm_split(MPI_COMM_WORLD, DA_Nprocs, myrank, Var3DCommunicator, ierr)
        PETSC_COMM_WORLD = Var3DCommunicator
        call PetscInitialize(PETSC_NULL_CHARACTER,stat)
        CHKERRQ(stat)
      else
        call MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, myrank, Var3DCommunicator, ierr)
      endif
#endif !ExecDA
#else
      mpi_glcomm_size = 1
      myrank = 0
#endif
END SUBROUTINE
…"
```

It is possible to see how, through the *parlec call*, number of processors for data assimilation was imported inside the routine my node (from *namelist.init* input file). At this point of the code, all processors whose rank was minor then the number of processors declared, are collected inside the Var3DCommunicator that handle the assimilation scheme inside the OGSTM – BFM.

## 5.3 Profiling

3dvar ia code that works to convergence with a cost function that is the core part of the system. As every run could be different from the other, we blocked the simulation using the same time simulation window and the same satellite file. Running the code before the change we have done with this thesis, the old code for a single assimilation finished in 420 seconds (7 minutes). In a real case, the single assimilation procedure was done weekly: for 1 year of simulation, DA accounted for 7*4(weeks)*12(months) /60 = 5.6 hours (to be added to the 22 ours of model computation time). With the new requirements we would have 7*7(daily)*4(weeks)*12(months)/60 = 39.2 hours for data assimilation only. That would be a great amount of time dramatically increasing the computational hours.

## 5.4 New routine

We have written a new routine *3d_var_MP.f90* that handle the distribution of the cores between nodes. We started from the writing_procs array of the I/O part, in which are declared the first rank of each node, and after that we implemented a new array of dimension Nodes*n° of DA procs per node in which one we store the ranks of processor that we want to use for the data assimilation. The distribution is made symmetric for each socket of a node. The end user through the namelist.init input file can control the total number of DA procs and the number of processors per node. We have implemented three possibilities: the first one is when the end user decides to use only one node, all the DA processors will be on the same node and in number of 9. The second possibility is when DA_procs* nodes < Da_total procs: we defined that will be 5 processors per node scaling from the total number. The third possibility is when DA_procs > nodes*DA_procs_per_node: from the input id defined how many processors per node and how many nodes. We have declared a boolean

V3D_VAR_PARALLEL for all processors: processors that are selected for data assimilation change the boolean value from false to true, so they have been activated for data assimilation procedures: that will be the control point for DA routines.

From *3d_var_MP.f90*:
"…
```
if(nodes == 1) then
                case_selection = 1
        else if (nodes*TREd_procs_per_node < DA_Nprocs) then
                case_selection = 2
        else
                case_selection = 3
        end if

        counter_3d_procs = 1
        SELECT CASE (CASE_SELECTION)
                CASE(1)
                        ALLOCATE(TREd_procs_per_node_array(max_procs_per_one_node))
                        do k=1, max_procs_per_one_node
                                TREd_procs_per_node_array(k) = k - 1
                        end do
                        DO i=1, 9
                                write(*,*) '3d_var_proc is',TREd_procs_per_node_array(i)

                        END DO
                        do i=1,max_procs_per_one_node
                                IF(MYRANK == TREd_procs_per_node_array(i)) then
                                        V3D_VAR_PARALLEL = .true.
                                END IF
                        end do
                CASE(2)
                        ALLOCATE(TREd_procs_per_node_array(nodes*TREd_procs_per_node))
                        do i=1, nodes
                                if(counter_3d_procs > nodes*TREd_procs_per_node) then
                                        exit
                                else

TREd_procs_per_node_array(counter_3d_procs)=writing_procs(i)
                                        counter_3d_procs = counter_3d_procs + 1
                                        do j=1, TREd_procs_per_node -1
                                                if(counter_3d_procs                        >
nodes*TREd_procs_per_node)then
                                                        exit
                                                else

TREd_procs_per_node_array(counter_3d_procs)=writing_procs(i)+ j
                                                        counter_3d_procs = counter_3d_procs
+ 1
                                                end if
                                        end do
                                end if
                        end do
                        DO i=1, counter_3d_procs - 1
                                write(*,*) '3d_var_procis',TREd_procs_per_node_array(i)
                        END DO
                        do i=1,nodes*TREd_procs_per_node
                                IF(MYRANK == TREd_procs_per_node_array(i)) then
                                        V3D_VAR_PARALLEL = .true.
                                END IF
                        end do
                CASE(3)
                        ALLOCATE (TREd_procs_per_node_array(DA_Nprocs))
                        DO i=1, NODES
                                if(counter_3d_procs > DA_Nprocs)then
                                        exit
                                else

TREd_procs_per_node_array(counter_3d_procs)=writing_procs(i)
                                        counter_3d_procs = counter_3d_procs + 1
                                        do j=1, TREd_procs_per_node -1
                                                if(counter_3d_procs > DA_Nprocs) then
                                                        exit
                                                else

TREd_procs_per_node_array(counter_3d_procs)=writing_procs(i)+ j
                                                        counter_3d_procs = counter_3d_procs
+ 1
                                                end if
                                        end do
                                end if
                        end do
                        DO i=1, counter_3d_procs - 1
                                write(*,*)'3d_var_procis',TREd_procs_per_node_array(i)
                        END DO
                        do i=1,DA_Nprocs
                                IF(MYRANK == TREd_procs_per_node_array(i)) then
                                        V3D_VAR_PARALLEL = .true.
                                END IF
                        end do
                END SELECT
!PG parts from ogstm_mpi
        if(V3D_VAR_PARALLEL) then
                SELECT CASE (CASE_SELECTION)
                        CASE(1)
                                call                         MPI_Comm_split(MPI_COMM_WORLD,
max_procs_per_one_node,myrank,Var3DCommunicator, ierror)
                        CASE(2)
                                call
MPI_Comm_split(MPI_COMM_WORLD,nodes*TREd_procs_per_node,myrank,Var3DCommunicator, ierror)
                        CASE(3)
                                call
MPI_Comm_split(MPI_COMM_WORLD,DA_nprocs,myrank,Var3DCommunicator,ierror)
                END SELECT

                PETSC_COMM_WORLD = Var3DCommunicator
                call PetscInitialize(PETSC_NULL_CHARACTER,stat)
                CHKERRQ(stat)
        else
                call MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED,myrank,Var3DCommunicator,
ierror)
        endif
…”
```

## 5.5 Improvements

We have done a lot of tests to identify the best solution between nodes and number of processors per node. The weak part of the code is the communication between processors: there are 32 *MPI_all_to_all* that are called at every step of the convergence run. Increasing the number of nodes and of processors increase dramatically the communication time intra – nodes and inter – nodes. We have obtained the best results with a setup of 4 processor per node and 13 nodes, running data assimilation procedures in 155 seconds (2.5 minutes). The speedup was: 420/155 = 2.7 times. In the real case, the single assimilation procedure is done weekly: for 1 year of simulation, DA accounted for 2.5*4(weeks)*12(months) /60 = 2 hours (to be added to the 22 ours of model computation time). With the new requirements we would have 2.5*7(daily)*4(weeks)*12(months)/60 = 14 hours for data assimilation. Speedup is not so big, but is enough to save a lot of computational hours: 5.6 – 2 = 4.6 * 720 procs = 3312 computational hours for a year simulation at present requirements (39.2 -14 = 25.2 * 720 procs = 18144 computation hours for one year simulation with future requirements).

## 5.6 Issues

As we have presented, communication time is the weak part with the cost function too. Future intentions are to change the data assimilation procedure to cut the computational time, but we need a lot of time to do the work. Maybe a new MHPC thesis?

# 6. FUTURE WORK

We have found new objectives to make the OGSTM – BFM model more efficient and less costly from the computational hours point of view.

We would like to implement a parallel reading of input files.

We would like to optimise better the data assimilation part.

We would like to optimise better I/O.

All these ideas have to be studied and defined.

# 7. REFERENCES

Cossarini, G., Lazzari, P., Solidoro, C., 2015. Spatiotemporal variability of alkalinity in the Mediterranean Sea. Biogeosciences, 12(6), 1647-1658.

Lazzari, P., Teruzzi, A., Salon, S., Campagna, S., Calonaci, C., Colella, S., Tonani, M., Crise, A. 2010. Pre-operational short-term forecasts for the Mediterranean Sea biogeochemistry. Ocean Science, 6, 25-39.

Lazzari, P., Solidoro, C., Ibello, V., Salon, S., Teruzzi, A., Béranger, K., Colella, S., and Crise, A., 2012. Seasonal and inter-annual variability of plankton chlorophyll and primary production in the Mediterranean Sea: a modelling approach. Biogeosciences, 9, 217-233.

Lazzari, P., Solidoro, C., Salon, S., Bolzon, G., 2016. Spatial variability of phosphate and nitrate in the Mediterranean Sea: a modelling approach. Deep Sea Research I, 108, 39-52.

Salon, S.; Cossarini, G.; Bolzon, G.; Feudale, L.; Lazzari, P.; Teruzzi, A.; Solidoro, C., and Crise, A. (2019) Novel metrics based on Biogeochemical Argo data to improve the model uncertainty evaluation of the CMEMS Mediterranean marine ecosystem forecasts. Ocean Science, 15, pp.997–1022. DOI: https://doi.org/10.5194/os-15-997-2019.

Teruzzi, A., Dobricic, S., Solidoro, C., Cossarini, G. 2014. A 3D variational assimilation scheme in coupled transport biogeochemical models: Forecast of Mediterranean biogeochemical properties, Journal of Geophysical Research, doi:10.1002/2013JC009277.

Teruzzi, A., Bolzon, G., Salon, S., Lazzari, P., Solidoro, C., Cossarini, G., 2018. Assimilation of coastal and open sea biogeochemical data to improve phytoplankton simulation in the Mediterranean Sea. Ocean Modelling, 132, 46-60.

Teruzzi, A., Di Cerbo, P., Cossarini, G., Pascolo, E., Salon, S., 2019. Parallel implementation of a data assimilation scheme for operational oceanography: the case of the MedBFM model system, submitted to Computers & GeosciencesTeruzzi, A., Bolzon, G., Salon, S., Lazzari, P., Solidoro, C., and Cossarini, G. (2018). Assimilation of coastal and open sea biogeochemical data to improve phytoplankton simulation in the Mediterranean sea. Ocean Modelling, 132:46–60.