Parallelism in LMDZ

A short survival guide for those who feel diagonally stuck in a parallel world

LMDZ course, December 17 2020

What is parallelism ?

• Simply based on dividing a task into many smaller ones handled separately but simultaneously among `workers'.
• e.g.: We want to translate the book *'Les Misérables'*, (V. Hugo, 1862, 5 cap. More than 1500 pages)

1. Assume we are all fluent in English and French
2. We divide the book in blocks of equal number of pages
3. Each one of us translates only his/her assigned pages independently of the others (except for possible borderline cases, e.g. a sentence starting at the end of a block and continuing in the next)
4. The translation of the whole book is finished once everyone is finished
5. There needs be some organizer who will assign tasks and be collecting all the translations to a single common final product.

• Parallel computing mimics this organisation.
• Making use of different cores, mainly with two paradigms: distributed memory and shared memory
• Technically, computations are spread over different cores.

Why go parallel ?

• To have simulations run faster by using multiple cores to share the workload, each working “as independently as possible” (i.e.: with the minimum communication/interaction with the other cores).

• To benefit from modern architectures (from laptops to supercomputers).

Which parallelism is implemented in LMDZ ?

• LMDZ is designed so that it can be compiled and run in serial (sequential) or parallel mode (in various forms, MPI, OpenMP, or mixed MPI/OpenMP, as will be discussed later).

• The implementation of the parallel modes has been thought of, and done (Yann Meurdesoif, LSCE, IPSL), so that it can easily benefit from all hardware platforms. The various aspects of parallelism in the code have moreover been coded as to be the least intrusive for users and developers.

MPI and OpenMP parallelism paradigms

MPI : Distributed Memory parallelism

• The code to be executed is replicated on all CPUs in as many processes.

• Each process runs independently and by default does not have access to the other processes' memory.

• Data is shared via a **message passing interface library** (OpenMPI, MPICH, etc.) which uses the interconnection network of the machine. Efficiency then essentially relies on the quality of the interconnection network. As a new set of subroutines and functions: CALL bcast, CALL gather, CALL scatter, ...

• The number of processes to use is chosen at runtime: mpirun -n 8 gcm.e

OpenMP : Shared memory parallelism

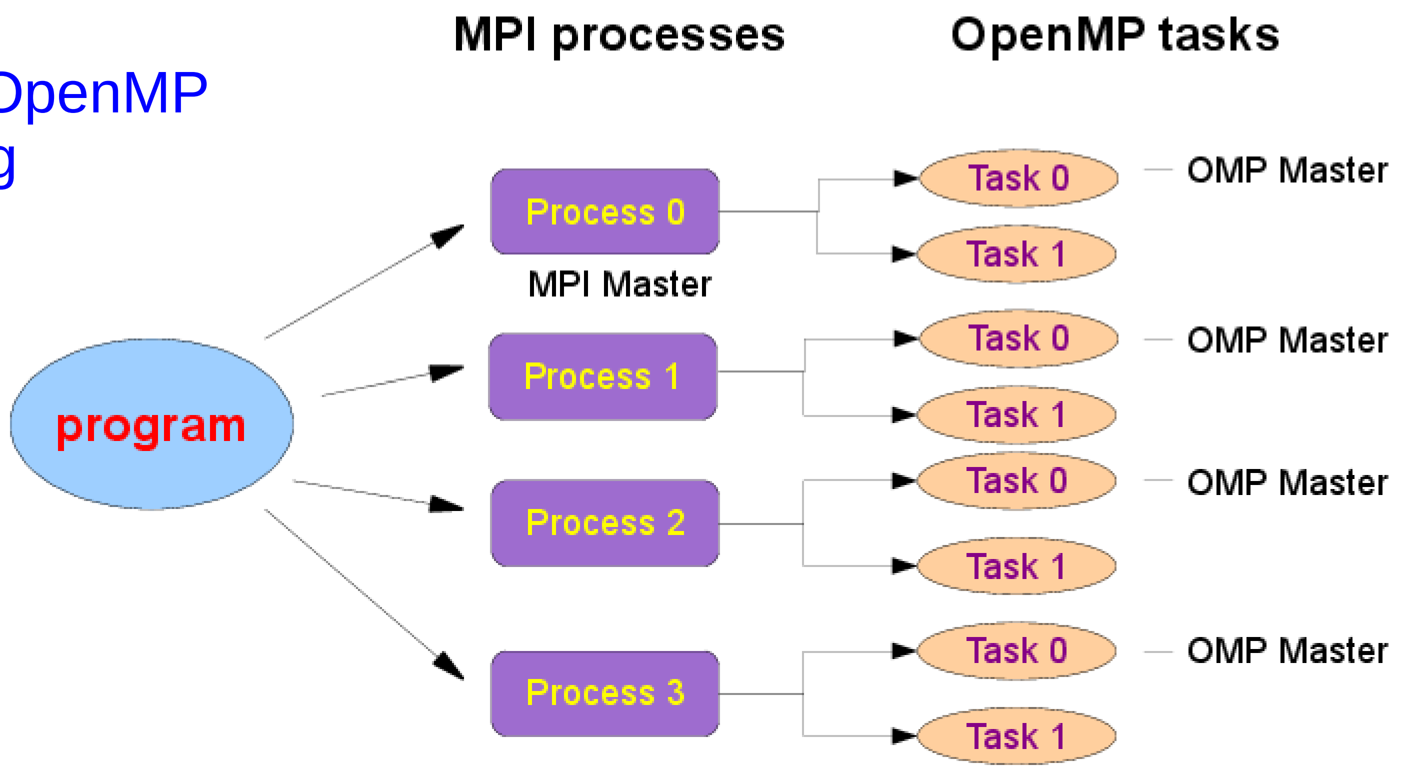
• This parallelism is based on the principle of multithreading. Multiple tasks (threads) run concurrently within a process.

• Each task essentially has (shared) **access to the global memory** of the process.

• Loops are parallelized using directives (!$OMP ... , which are included in the source code where they appear as comments) interpreted by the compiler.

• The number of OpenMP threads to used is set via an environment variable OMP_NUM_THREADS (e.g.: OMP_NUM_THREADS=4)

Hybrid MPI/OpenMP programming

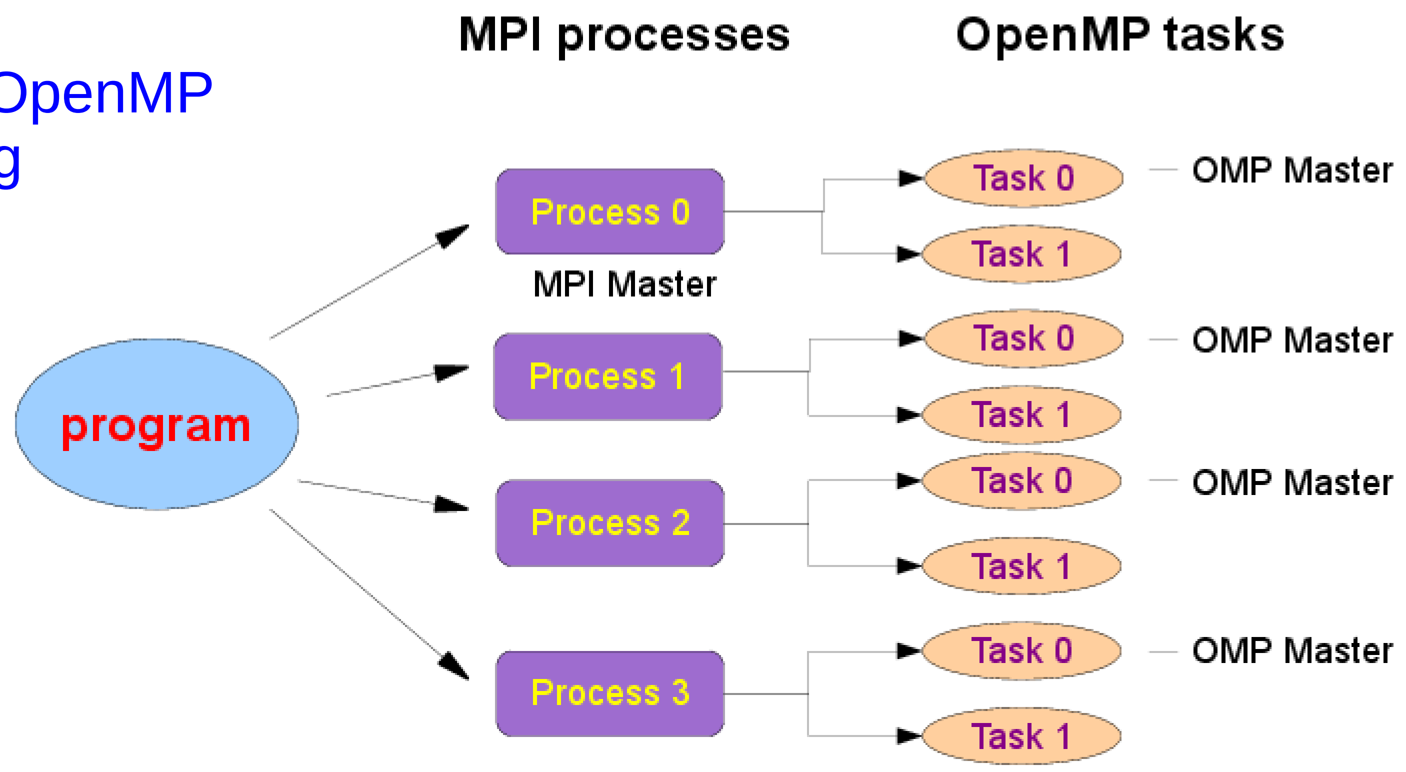


➢ Each MPI process launches OpenMP threads which have access to its global memory. (threads also have private memory for specific variables)

➢ In LMDZ, MPI and OpenMP are differently implemented to best fit requirements. The number of OpenMP threads per MPI process is fixed and remains the same throughout a simulation.

➢ No need to try to use more cores than available on your machine!

Hybrid MPI/OpenMP programming



Different parallization approaches in the dynamics and physics

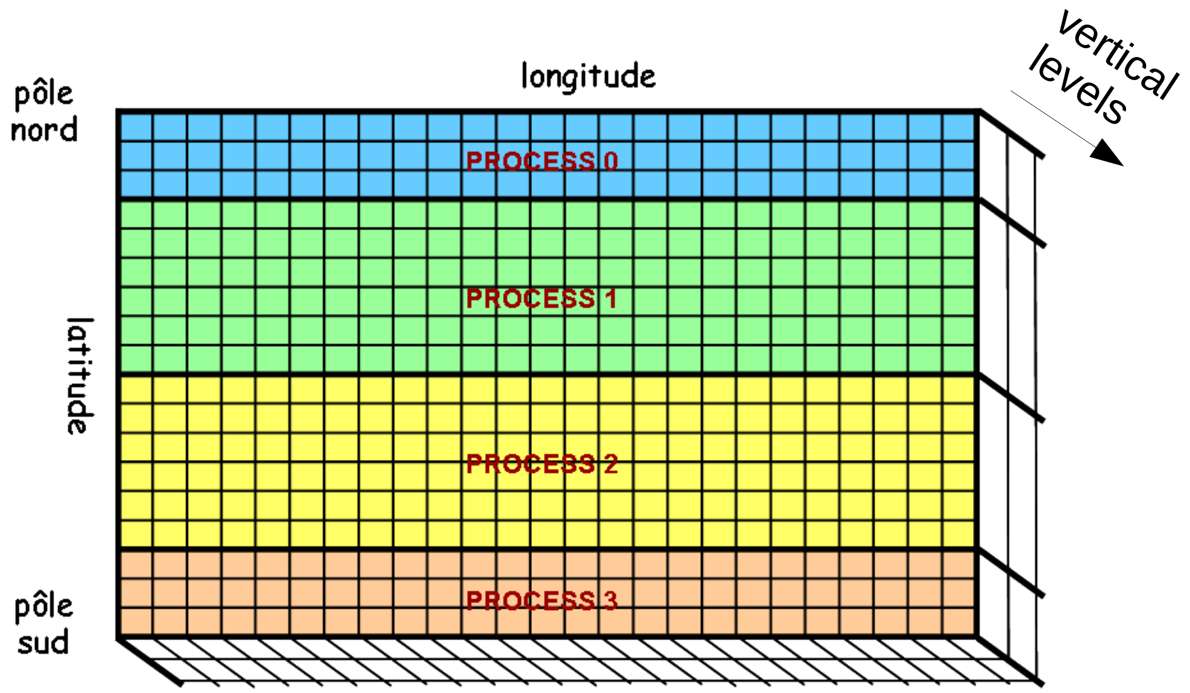
- In the dynamics

  Significant interactions between neighbouring meshes, and therefore numerous cases of data exchange and synchronizations. The subtler part of the parallelism in the code..

- In the physics

  There is **no interaction** between neighbouring columns of the atmosphere, which can easily be handled separately.

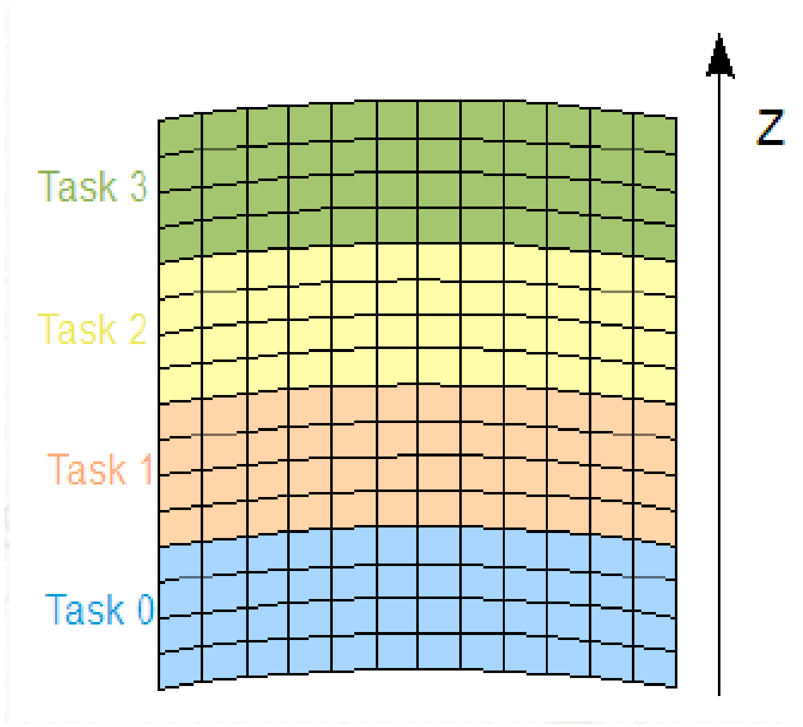
Parallelism in the dynamics



=> MPI tiling

- Tiling is by bands of latitude.
- A **minimum of 2 latitude bands per MPI process** is mandatory.
- But the work load is not the same for all latitudes (essentially because of the polar filter).
- Use option *adjust=y* (in *gcm.def*) to dynamically optimize (during the run) the band distribution of processes.
  - Run the GCM (in MPI mode only!) over at least a few thousand time steps to obtain a Bands_**x**x**_*prc.dat file.
  - Re-run the simulation using option *adjust=n* (with the Bands_* file in the run directory)
    NB: if there is no Bands_* file, the GCM creates one with a uniform balance between processes

Parallelism in the dynamics

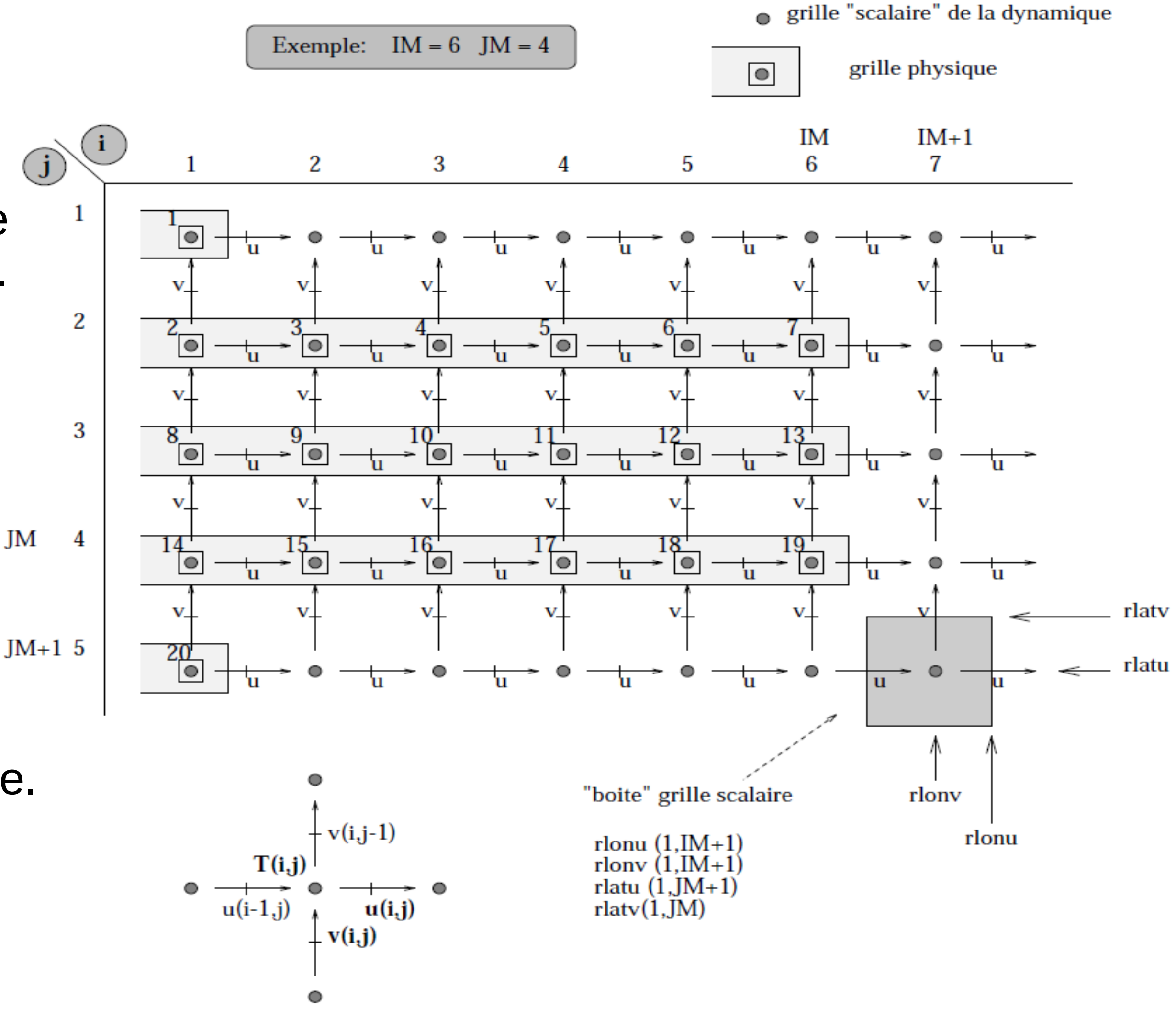


=> OpenMP spliting

- The split is done along the vertical levels only (the outermost loop in most computations).
- An estimate size of blocs to assign to each thread can be specified using option *omp_chunck*=... in gcm.def.
- In practice, target at least chunks of 4 or 5 vertical levels for each OpenMP task (an optimal compromise, but which may depend on the machine on which the code is run).

Parallelism in the physics

- The physics handles physical phenomena which interact within a single atmospheric column: radiation, convection boundary layer, etc.
- Individual columns of atmosphere **do not interact** with one another.
- The paralellization strategy is to distribute the colums of atmosphere over all cores.
- The physics grid : klon_glo geographic points over klev vertical levels.
  First node (1) => North pole,
  last node (klon_glo) => South pole.

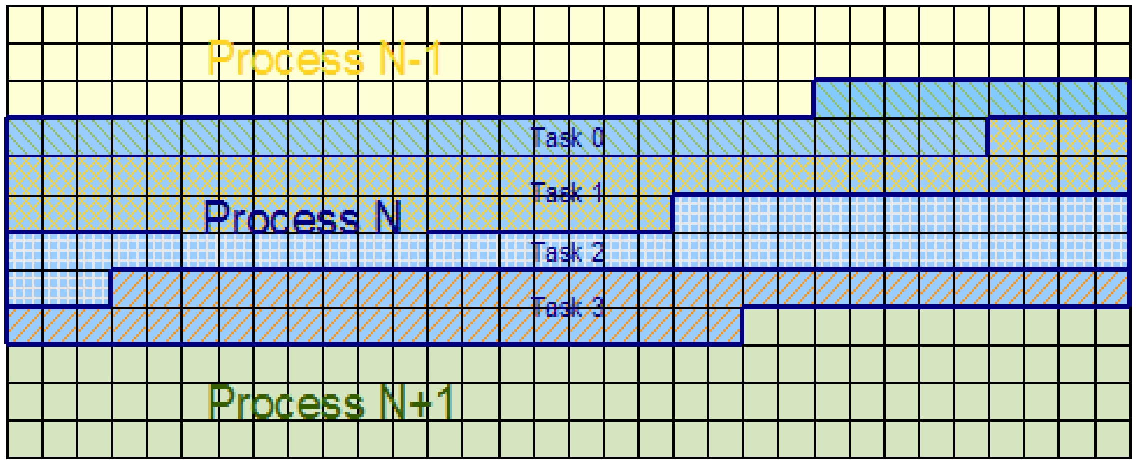


Parallelism in the physics

- The columns from the global domain are first distributed among the MPI processes.
  - The global domain : klon_glo columns of atmosphere

- The columns of each MPI domain are assigned to the OpenMP tasks assigned to the that process :
  - In each MPI domain : klon_mpi columns : Σ klon_mpi = klon_glo
  - In each OpenMP domain : klon_omp columns : Σ klon_omp = klon_mpi

- In practice, the size of the local domain klon is **an alias** of klon_omp (so as to behave exactly as when running the serial code).
  ➔ Never forget that klon varies from one core to another.



Time for an illustrative and interactive example

N.B: **There is a dedicated Tutorial about installing and using LMDZ in parallel**

Some LMDZ code linked to parallelism

Different parallelism control parameters:

- klon_glo, nbp_lon, nbp_lat, nbp_lev
- klon_mpi , klon_mpi_begin, klon_mpi_end, ii_begin, ii_end, jj_begin, jj_end, jj_nb, is_north_pole, is_south_pole, is_mpi_root, mpi_rank, mpi_size
- klon_omp, klon_omp_begin, klon_omp_end, is_omp_root, omp_size, omp_rank

Some general considerations in physics:

- For openMP, delacre all SAVE variables as:

  ```
  REAL, SAVE :: save_var
  !$OMP THREADPRIVATE(save_var)
  ```
- Allocation of variables with `klon' (real size within each core)

  ```
  ALLOCATE (myvar(klon))
  ```
- Neither North or Pole grid points are:

  ```
  myvar(1) or myvar(klon)
  ```

Data transferts:

- The transfer interfaces handle data of all the basic types : REAL, INTEGER, LOGICAL, CHARACTER(only for broadcast)
- The transfer interfaces moreover can handle fields of 1 to 4 dimensions

Data transfer in the physics (examples)

Broadcast : the master process duplicates its data to all processes and tasks.

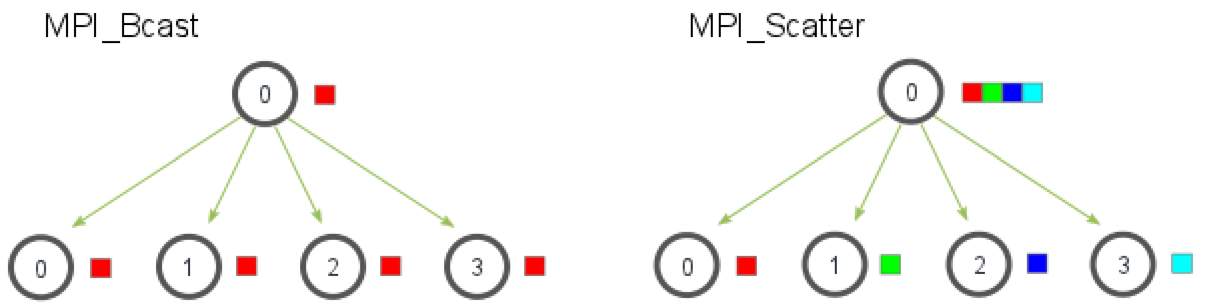
Independently of the variable's dimensions

CALL bcast(var)

Scatter : the master task has a field on the global grid (klon_glo) which is to be scattered to the local grids (klon).

The first dimension of the global field must be klon_glo, and the one of the local field must be klon

CALL scatter(field_glo,field_loc)



Scatter2D : same as Scatter except that the global field is defined on a 2D grid of : nbp_lon x nbp_lat.

The first and second dimensions of the global field must be (nbp_lon,nbp_lat), and the first dimension of the local field must be klon

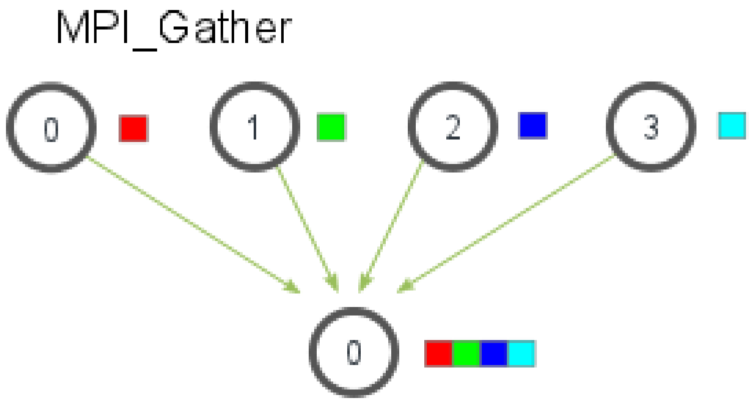
CALL scatter2D(field2D_glo,field1D_loc)

Figures from: http://mpitutorial.com/mpi-scatter-gather-and-allgather/
Note however that in LMDZ the Scatter/Gather process is coded across MPI and OpenMP.

Gather : a field defined on the local grids (klon) is gathered on the global grid of the master process (klon_glo).

The first dimension of the global field must be klon_glo, and the one of the local field must be klon

CALL gather(field_loc,field_glo)



Gather2D : gather data on the 2D grid of the master process.

CALL gather2D(Field1D_loc,Field2D_glo)

Figures from: http://mpitutorial.com/mpi-scatter-gather-and-allgather/
Note however that in LMDZ the Scatter/Gather process is coded across MPI and OpenMP.

Illustrative example of load data from a file, (simplified) extracted from phylmd/read_map2D.F90

```
USE dimphy
USE netcdf
USE mod_grid_phy_lmdz
USE mod_phys_lmdz_para
...

REAL, DIMENSION(nbp_lon,nbp_lat) :: var_glo2D
REAL, DIMENSION(klon_glo)        :: var_glo1D
REAL, DIMENSION(klon)            :: varout

! Read variable from file. Done by master process MPI and master thread OpenMP
  IF (is_mpi_root .AND. is_omp_root) THEN
     NF90_OPEN(filename, NF90_NOWRITE, nid)
     NF90_INQ_VARID(nid, varname, nvarid)

     start=(/1,1,timestep/)
     count=(/nbp_lon,nbp_lat,1/)
     NF90_GET_VAR(nid, nvarid, var_glo2D,start,count)
     NF90_CLOSE(nid)

     ! Transform the global field from 2D to 1D
     CALL grid2Dto1D_glo(var_glo2D,var_glo1D)
  ENDIF

! Scatter global 1D variable to all processes
  CALL scatter(var_glo1D, varout)
```

Writing output IOIPSL files and rebuilding the results

- Each MPI process writes data for its domain in a distinct file. One thus obtains as many files histmth_00XX.nc files as processes were used for the simulation.

- The domain concerned by a given IOIPSL file is defined with a call to histbeg, which is encapsulated in histbeg_phy (module iophy.F).

- Data is gathered on the master (rank 0) OpenMP task for each process. Each MPI process then calls the IOIPSL routine histwrite, which is encapsulated in histwrite_phy (module iophy.F).

- **Warning**: what is mentioned above is only true for **outputs in the physics**; it is also possible to make some outputs in the dynamics (triggered via ok_dyn_ins and ok_dyn_ave in *run.def*), but for these, data is moreover gathered on the master process so that there is only one file on output (which is a major bottleneck, performance-wise) => should only be used for debugging.

Writing output IOIPSL files and rebuilding the results

Once the simulation finished, one must gather the data in a single file.
This requires using the rebuild utility:

```
rebuild -o histmth.nc histmth_00*.nc
```

- rebuild is a utility distributed with IOIPSL
  See « How to install IOIPSL and the rebuild utility» in the LMDZ website FAQ (http://lmdz.lmd.jussieu.fr/utilisateurs/faq-en)

- If using the install_lmdz.sh script then the rebuild utility is automatically built and placed in the “modipsl/bin” subdirectory; Look at script reb.sh in the BENCH32x32x39 test case:

```
modipsl=`pwd | sed -n -e 's/modipsl.*.$/modipsl/p'`
file=$1

...

$modipsl/bin/rebuild -o $file.nc ${file}_0*.nc
if [ -f $file.nc ] ; then rm -f ${file}_0*nc ; fi
```

The XIOS library (there is a dedicated tutorial)

- Next generation of the output library (IOIPSL not upgraded any more, will become depreciated).

  https://forge.ipsl.jussieu.fr/ioserver

- Having installed the XIOS library (reference versions on global IPSL account on Ada and Curie), compile LMDZ using the "-io xios" option:

```
makelmdz_fcm -mem -parallel [mpi|omp|mpi_omp] -io xios .....
```

- Output is managed via `xml' files
- Set flag "ok_all_xml=y" in config.def in order to control the ouputs in the X**.nc files.

- With XIOS, output files can be generated as single files (no need to rebuild output files), by setting type="one_file" par_access="collective" parameters in the attributes of the "file" definition in the xml:

```
<file_definition type="one_file" par_access="collective" ...... >
```

To summarize

► In the physics, as long as there is no communication between columns, you can develop and modify code “as if in serial”. Only mandatory requirement (for OpenMP): variables which have a SAVE attribute have to be declared as !$OMP THREADPRIVATE.

- ➔ Do take the time to check the correct integration of modifications! Results should be identical (bitwise) when the number of processes or OpenMP threads is changed (at least when compiling in 'debug' mode).

► In the dynamics, parallelism is much more intrinsic; one should really take the time to understand the whole system before modifying any line of code.

► One can compile in any of the following parallel modes: mpi, omp or mpi_omp

```
makelmdz_fcm -mem -parallel [mpi|omp|mpi_omp] .....
```

► A run should use as many cores as possible, without forgetting that the maximum number of MPI processes = number of nodes along the latitude / 2 and that it is usually best to use 1 OpenMP task for every 4 or 5 points along the vertical.

► To optimize the workload among different MPI processes, run a first month with *adjust=y* in run.def. And then use the obtained bands_resol_Xprc.dat files for the following simulations.

Mixed bag of thoughts, advice and comments

- To run on a « local » machine (typically a multicore laptop):
  - An MPI library must be installed, and the « arch » files must be correspondingly modified to compile the model: 'makelmdz_fcm -arch local [-mem] ...'
  - It is always best to be able to use as much memory as possible: ulimit -s unlimited
  - It is also important to reserve enough private memory for OpenMP tasks: export OMP_STACKSIZE=200M
  - Use 'mpirun -np n ...' to run with n processes, and 'export OMP_NUM_THREADS=m' to use m OpenMP tasks
  - Some examples and advice are given here (in English and French): http://lmdz.lmd.jussieu.fr/utilisateurs/guides/lmdz-parallele-sur-pc-linux-en

- To run on clusters (Climserv, Ciclad, ...) and machines of supercomputing centres (IDRIS, CCRT,..):
  - Check the centre's documentation to see how to specify the number of MPI processes, OpenMP tasks, local limitations (memory, run time) for batch submission of jobs, etc.
  - Ask your favourite IPSL colleagues for some advice and their job headers on these machines.