

# Parallelism in LMDZ

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

LMDZ course, December 11th 2013

# 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

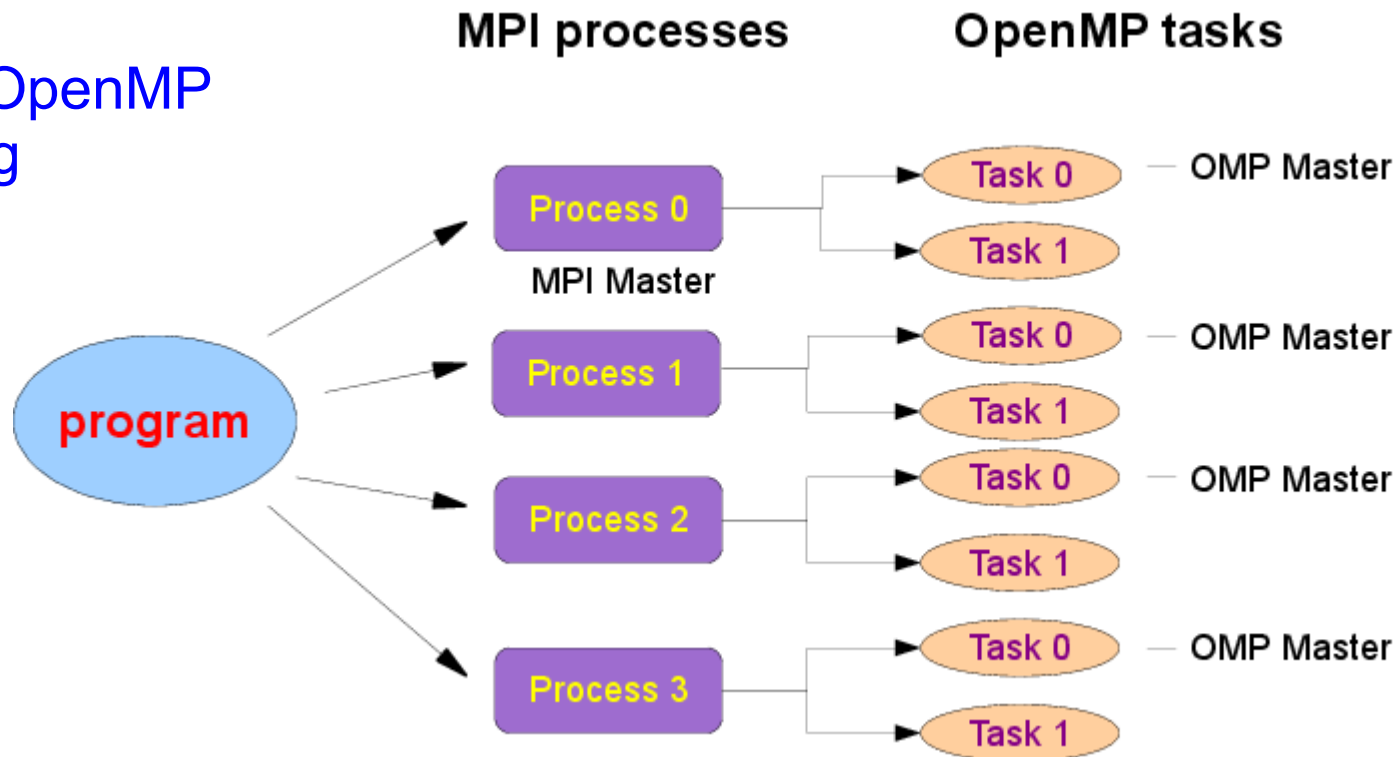
## MPI : Distributed Memory parallelism

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

## OpenMP : Shared memory parallelism

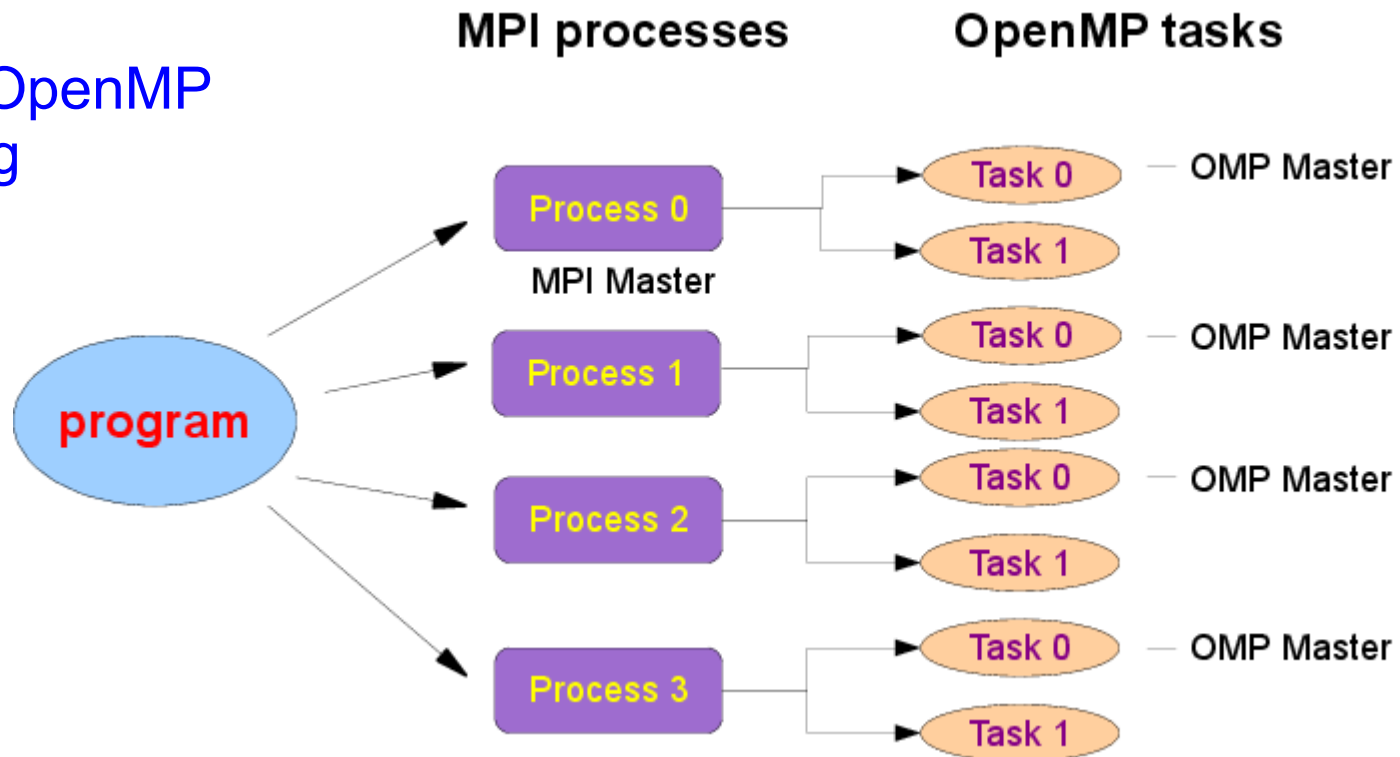
- This parallelism is based on the principle of multithreading. Multiple tasks (threads) run concurrently within a process.
- Each task essentially has (shared) **access to the global memory** of the process.
- **Loops are parallelized using directives** (!OMP ... , which are included in the source code 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.

## Hybrid MPI/OpenMP programming



## Different parallelization approaches in the dynamics and physics

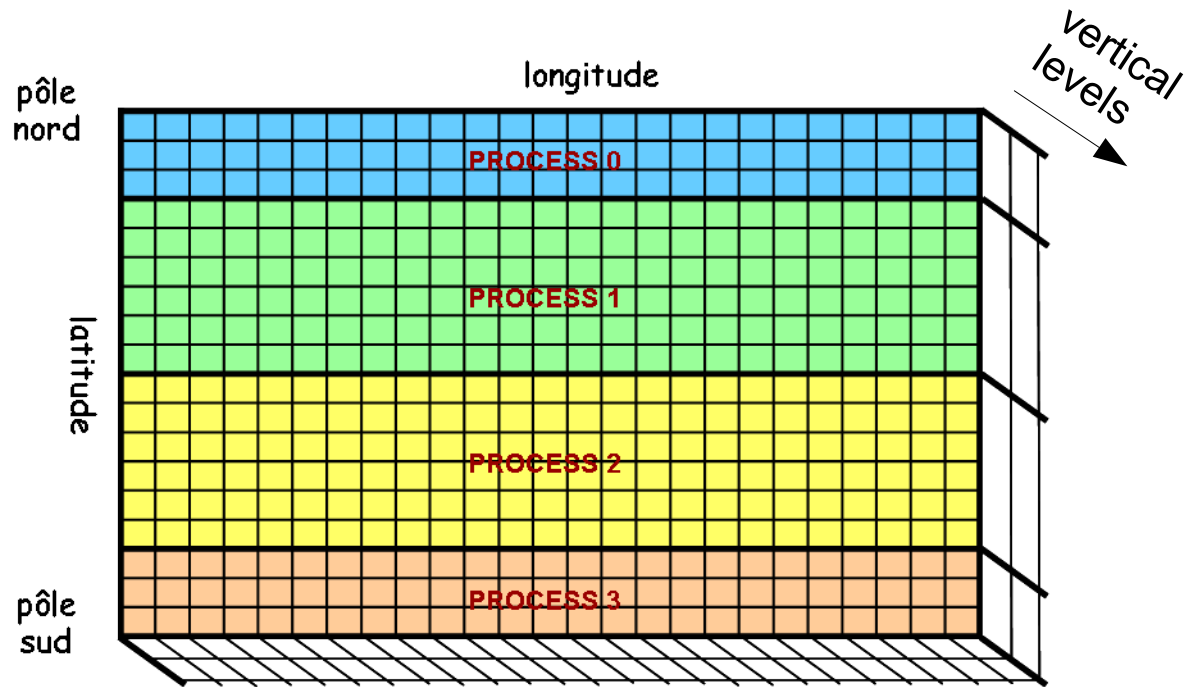
### ➤ In the dynamics

Short time steps; many 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

longer time steps; no interaction between neighbouring columns of the atmosphere.

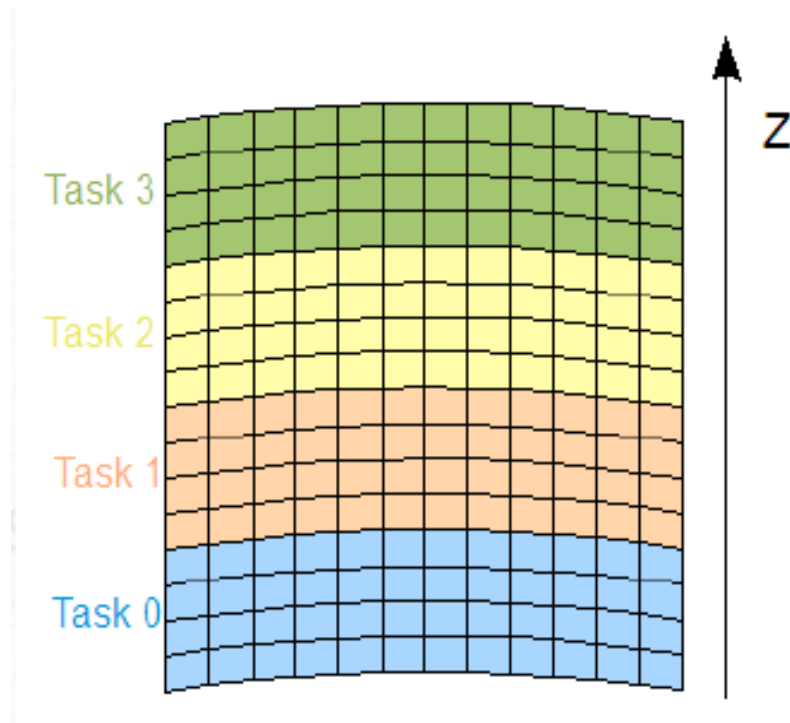
# Parallelism in the dynamics



## => MPI tiling

- Tiling is by bands of latitude.
- A minimum of 3 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

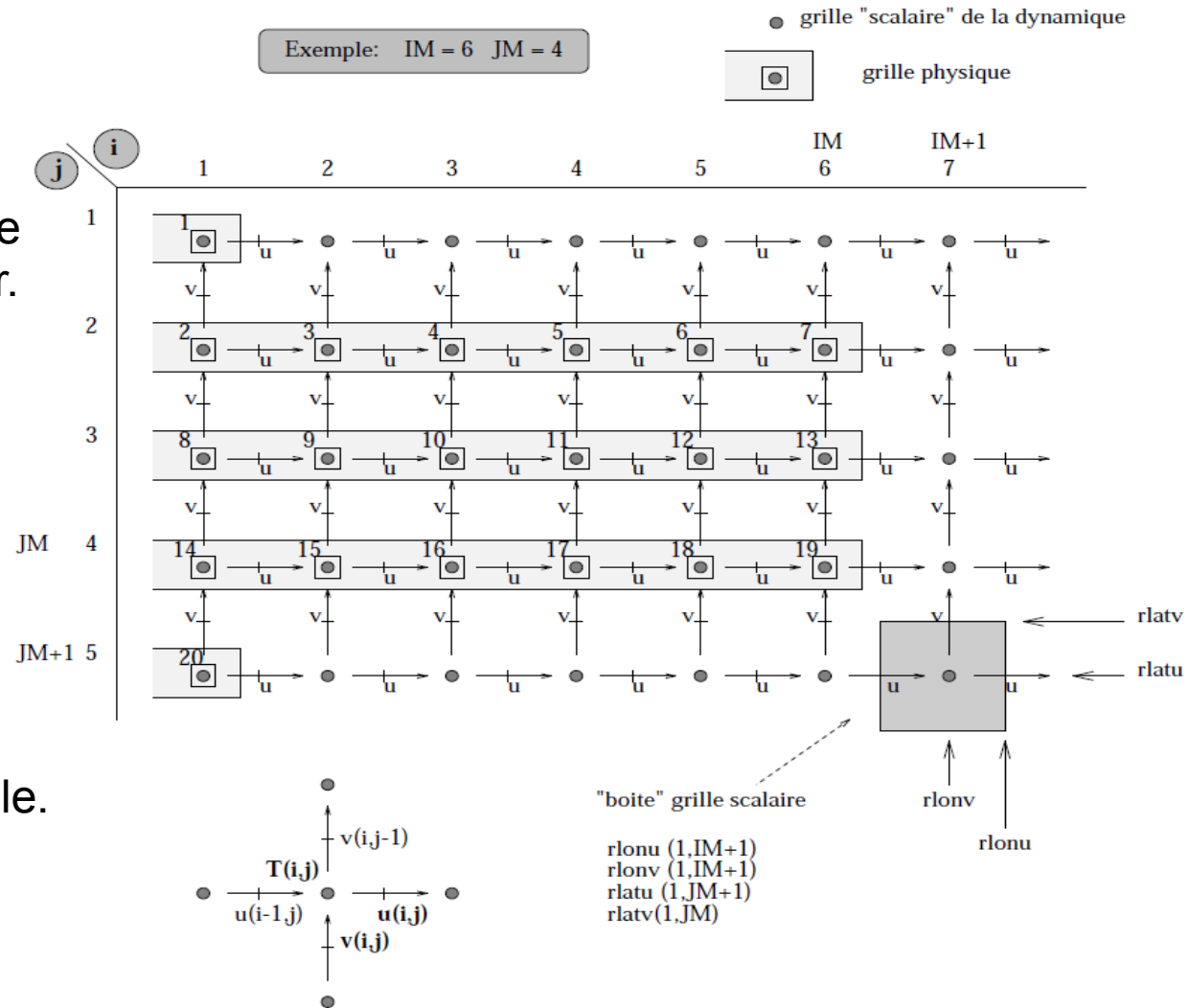


## => OpenMP split

- The split is done along the vertical levels only (the outermost loop in most computations).
- An indicate size of blocs to assign to each thread can be specified using option `omp_chunk=...` in `gcm.def`.
- In practice, target 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

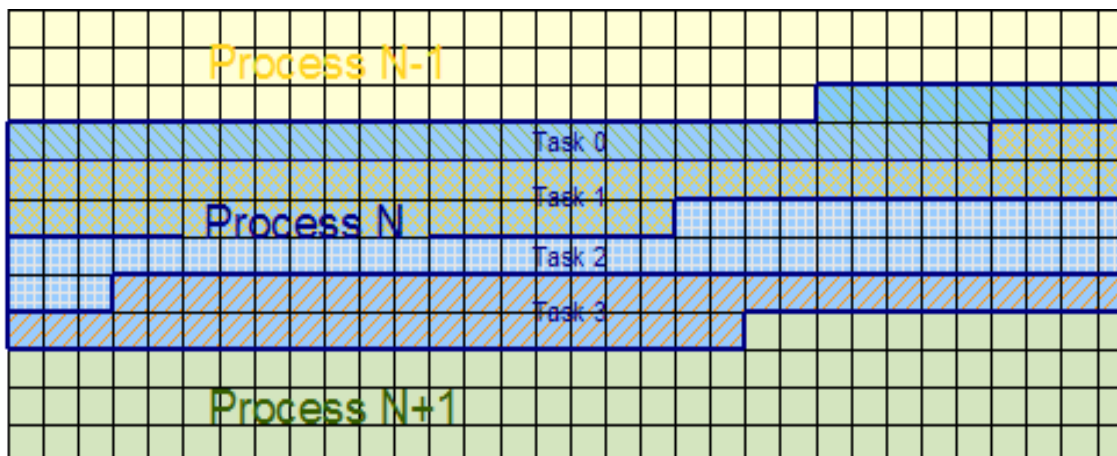
- The physics handles physical phenomena which interact within a single atmospheric column: radiation, convection boundary layer, etc.
- Individual columns of atmosphere **do not interact** with one another.
- The parallelization strategy is to **distribute the columns of atmosphere over all cores**.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





# Parallelism in the physics

- The columns from the global domain are first distributed among the MPI processes.
  - The global domain :  $k_{lon\_glo}$  columns of atmosphere
- The columns of each MPI domain are assigned to the OpenMP tasks assigned to the that process :
  - In each MPI domain :  $k_{lon\_mpi}$  columns :  $\sum k_{lon\_mpi} = k_{lon\_glo}$
  - In each OpenMP domain :  $k_{lon\_omp}$  columns :  $\sum k_{lon\_omp} = k_{lon\_mpi}$
- In practice, the size of the local domain  **$k_{lon}$  is an alias of  $k_{lon\_omp}$**  (so as to behave exactly as when running the serial code).
  - Never forget that  $k_{lon}$  varies from one core to another.



# Some code parameters linked to parallelism

## Global grid : module mod\_grid\_phy\_lmdz

- `klon_glo` : number of horizontal nodes of the global domain (1D grid)
- `nbp_lon` : number of longitude nodes (2D grid) = `iim`
- `nbp_lat` : number of latitude nodes (2D grid) = `jjm+1`
- `nbp_lev` : number of vertical levels = `klev` or `llm`

## MPI grid : module mod\_phys\_lmdz\_mpi\_data

- `klon_mpi` : number of nodes in the MPI local domain.
- `klon_mpi_begin` : start index of the domain on the 1D global grid.
- `klon_mpi_end` : end index of the end of the domain on the 1D global grid.
- `ii_begin` : longitude index of the beginning of the domain (2D global grid).
- `ii_end` : longitude index of the end of the domain (2D global grid).
- `jj_begin` : latitude index of the beginning of the domain (2D global grid).
- `jj_end` : latitude index of the end of the domain (2D global grid).
- `jj_nb` : number of latitude bands = `jj_end-jj_begin+1`
- `is_north_pole` : `.true.` If the process includes the North pole.
- `is_south_pole` : `.true.` If the process includes the south pole.
- `is_mpi_root` : `.true.` If the process is the MPI master.
- `mpi_rank` : rank of the MPI process.
- `mpi_size` : total number of MPI processes.

## Some code parameters linked to parallelism

### OpenMP grid : module mod\_phys\_lmdz\_mpi\_data

- `klon_omp` : number of nodes in the local OpenMP domain.
- `klon_omp_begin`: beginning index of the OpenMP domain within the MPI domain.
- `klon_omp_end` : end index of the OpenMP domain.
- `is_omp_root` : .true. If the task is the OpenMP master thread.
- `omp_size` : number of OpenMP threads in the MPI process.
- `omp_rank` : rank of the OpenMP thread.

# Coding constraints in the physics

- Nothing specific to worry about change (compared to serial case) if :
  - There is no interaction between atmospheric columns
  - No global or zonal averages need be computed
  - There are no input or output files to read or write
- One mandatory thing to enforce for OpenMP : **all variables declared as SAVE** or in « common » blocks must be protected by an **!\$OMP THREADPRIVATE** clause, for instance :

```
REAL, SAVE :: save_var  
!$OMP THREADPRIVATE(save_var)
```

- **Arrays must be allocated with the local size klon**, since klon may vary from a core to another; for instance :  
    ALLOCATE (myvar(klon))

**Warning!** Note that in the general case, myvar(1) or myvar(klon) are neither North nor South poles.

- Use the logical variables, **is\_north\_pole** et **is\_south\_pole** if a specific treatment for the poles is required

## Data transfer in the physics

- **Care is required if :**
  - there is some interaction between atmospheric columns
  - some zonal or global averages need be computed
  - files must be read or written

=> Then, some transfers between MPI processes and OpenMP tasks will be required:

- All **transfer routines are encapsulated** and transparently handle communications between MPI processes and OpenMP tasks.
- All are include in module : [mod\\_phys\\_lmdz\\_transfert\\_para](#)

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 (continued)

**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)

**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)

**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)

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

CALL gather2D(Field1D\_loc,Field2D\_glo)

## Some examples

### Reading physics parameters from files `physiq.def` and `config.def`

- Done in `conf_phys.F90`
- The master task read the value ( call `getin('VEGET',ok_veget_omp)` within an `!$OMP MASTER` clause) and then duplicates it to the other tasks (after the `!$OMP END MASTER` , there is an `ok_veget = ok_veget_omp` )

### Reading the initial state for the physics : `startphy.nc`

- Done in `phyetat0.F`
- The master task of the master process (`if(is_mpi_root.and.is_omp_root)` or equivalently `if(is_master)`) reads the field on the global grid and then scatters it to the local grids using the `scatter` routine.
- Encapsulated in the `get_field` routine of the `iostart` module.

### Writing the final state for the physics : `restartphy.nc`

- Done in `phyredem.F`
- Encapsulated in routine `put_field` of the `iostart` module, `put_field` first does a `gather` to collect all the local fields into a global field. Then the master task of the master process writes the field in the restart file `restartphy.nc`.

## Illustrative example of load data from a file, (simplified) extract 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>)
- In the IDRIS and CCRT supercomputing centres, **rebuild** is available to all, along with other common tools :

### **IDRIS**

Ada/Adapp :     /smphome/rech/psl/rpsl035/bin/rebuild

### **CCRT**

Curie :         /home/cont003/p86ipsl/X64/bin/rebuild

# 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: `'makeImdz_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 `'mpiexec -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://Imdz.lmd.jussieu.fr/utilisateurs/guides/Imdz-parallele-sur-pc-linux-en>
- **To run on clusters** (Climserv, Ciclad, Gnome, ...) **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.
  - Some information on appropriate job headers for some of the machines widely used at IPSL is gathered here (in French):  
<https://forge.ipsl.jussieu.fr/igcmg/wiki/IntegrationOpenMP>

## Illustrative Example of a purely MPI job (loadleveler) on Ada (IDRIS)

> cat Job\_MPI

```
# #####  
# ##      ADA IDRIS      ##  
# #####  
# @ job_name = test  
# @ job_type = parallel  
# @ output = $(job_name).$(jobid)  
# @ error =  $(job_name).$(jobid)  
# Number of MPI processes to use  
# @ total_tasks = 32  
# Maximum (wall clock) run time hh:mm:ss  
# @ wall_clock_limit = 0:30:00  
# Default maximum memory per process is 3.5Gb, but one  
# can ask for up to 7.0Gb with less than 64 processes  
# @ as_limit = 3.5gb  
# End of job header  
# @ queue  
  
poe ./gcm.e > gcm.out 2>&1
```

> llsubmit Job\_MPI

## Illustrative Example of a mixed MPI/OpenMP job (loadleveler) on Ada

```
> cat Job_MPI_OMP
```

```
# #####  
# ##      ADA IDRIS      ##  
# #####  
# @ job_name = test  
# @ job_type = parallel  
# @ output = $(job_name).$(jobid)  
# @ error =  $(job_name).$(jobid)  
# Number of MPI processes to use  
# @ total_tasks = 16  
# Number of OpenMP threads per MPI process  
# @ parallel_threads = 4  
# Maximum (wall clock) run time hh:mm:ss  
# @ wall_clock_limit = 0:30:00  
# Maximum memory per process is 3.5Gb x parallel_threads if more  
# than 64 processes; 7.0Gb x parallel_threads otherwise  
# @ as_limit = 14.0gb  
# End of job header  
# @ queue  
  
# Set the private stack memory for each thread  
export OMP_STACKSIZE=200M  
  
poe ./gcm.e > gcm.out 2>&1
```

```
> llsubmit Job_MPI_OMP
```

# libIGCM

- The **modips/libIGCM** environment manages IPSL models (LMDZOR, LMDZORINCA, IPSLCM5, LMDZREPR, ORCHIDEE\_OL) as a platform which simplifies extraction, installation and setting up simulations on the supercomputers “traditionally” used at IPSL; documentation available on

<http://forge.ipsl.jussieu.fr/igcmg/wiki/platform>

- The distributed job headers and a few parameters may still need be modified (e.g. memory requirements and maximum allowed run time).
- Recombination of the output files (rebuild) is automatically done.
- There are regularly **modips/libIGCM courses** (organized by Josefine Ghattas & Anne Cozic) ; sometimes in French and sometimes in English.

<http://forge.ipsl.jussieu.fr/igcmg/wiki/platform/formation>

# 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**  
`makeImdz_fcm -parallel [mpi|omp|mpi_omp] [-mem] .....`
- ▶ A run should use as many cores as possible, without forgetting that the **maximum number of MPI processes = number of nodes along the latitude / 3** 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.
- ▶ Rebuild the output files once the run is over : **rebuild -o histmth.nc histmth\_00\*.nc**