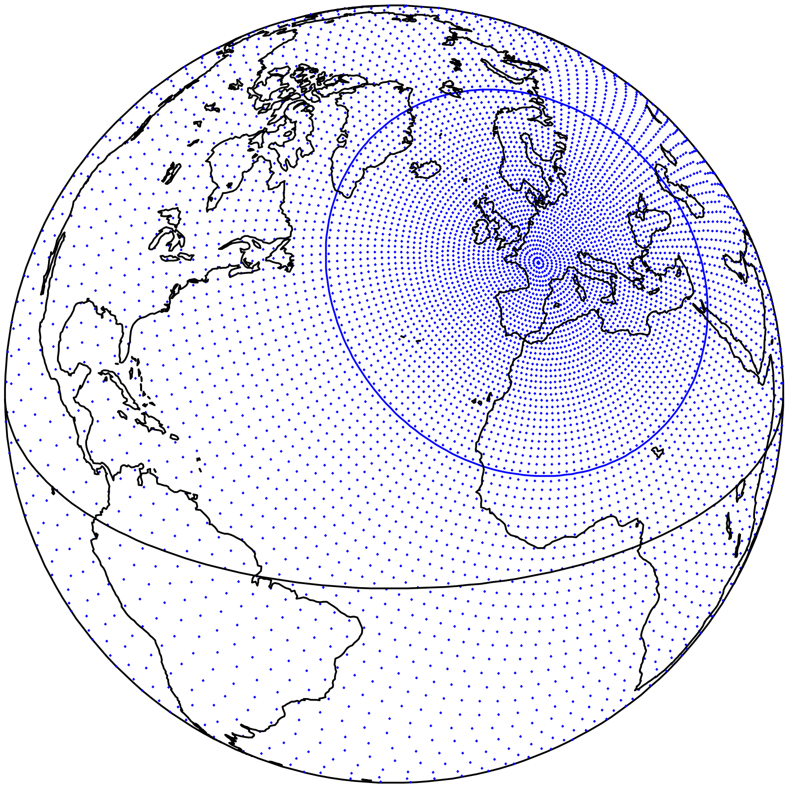
An introduction to the memory-distributed aspect of the code Arpege/Ifs/Aladin



R. El Khatib
METEO-FRANCE - CNRM/GMAP

December 6, 2002

Contents

List of Figures

1 Introduction

The architecture of ARPEGE/IFS/ALADIN software, which must be executed in real-time, is designed to run on high performance machines, which includes machines with a large number of processors.

The research of performance is base on the Amdal's law which roughly writes :

$$\mathrm{T = s + P/n}$$

where :
T denotes the elapsed CPU time,
s is the elapsed time spent in the serial part of the code,
P is the CPU time spent in the parallel part of the code
and n is the number of processors.

Obviously, the developer should design the code in a way that the serial part of the code would be as cheap as possible.

Beside of that, the parallel part of the code should be as efficient as possible, in order to take the maximum advantage of the machine performance.

For a long time, the machine constructors have developped *shared memory* machines : in such architecture each processor can access any part of the central memory.

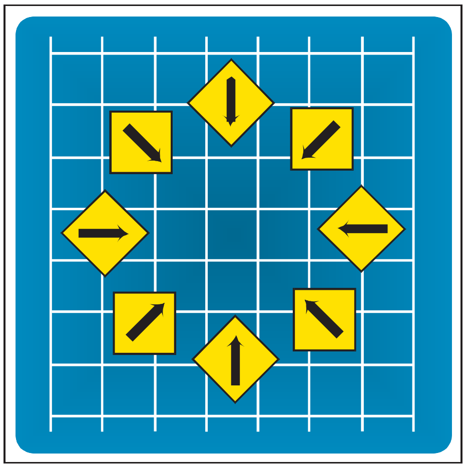


Figure 1: *Shared memory architecture : each processor (yellow item) can access any part of the central memory (blue area).*

With such architecture the constructors have been able to build computers with up to about 32 processors.

Unfortunately, and in contradiction with the law prediction, when the number of processors gets huge, the machine performance collapses, because the memory access conflicts get preponderant.

2 Distributed Memory architecture

To solve the problems of memory access conflicts when the number of processors increases, the high performance computers of today are based on the so-called *distributed memory* architecture : such machines are divided in *nodes*.

- Each *node* (also called : "PE" for "Processor Element") is a piece of central memory with one processor.
- All the nodes are linked to one another through a communication system called the *crossbar*.

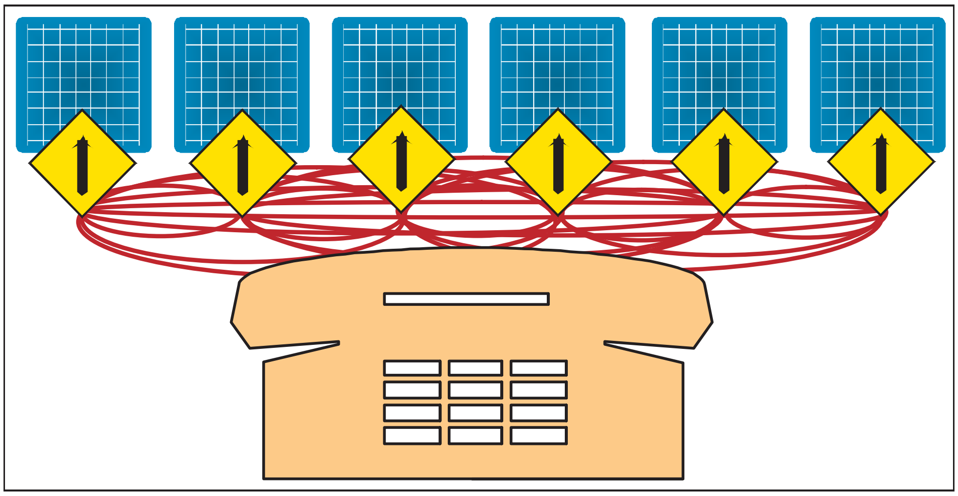


Figure 2: *Distributed Memory architecture : each processor (yellow item) belongs its own central memory (blue areas) ; all of them are linked through the crossbar (orange telephone).*

With such architecture, the number of nodes can reach 1000 or 2000, without being penalized by memory acceses since each processor belongs its own central memory.

In counterpart processors have to communicate with one another, and this has to be code by the programmer.

3 More about Distributed Memory architecture

This is a very short description of the architecture of the Fujitsu VPP serie :

Actually there are different kind of processors :

The primary PE (also named : PE #0) : this processor is unique ; it is connected with I/O devices and it is in charge of the machine management. This is the processor you usually connect to when you open a session on the machine.

The IOPEs : there is a few of them ; these processors are privileged since they are directly connected with I/O devices. They should be dedicated to the tasks which require a lot of I/O operations, like compilation or archived data access.

The secondary PEs : these are the ordinary processors ; they are undirectly connected to the I/O devices through the communication system ("crossbar") and the IOPEs.

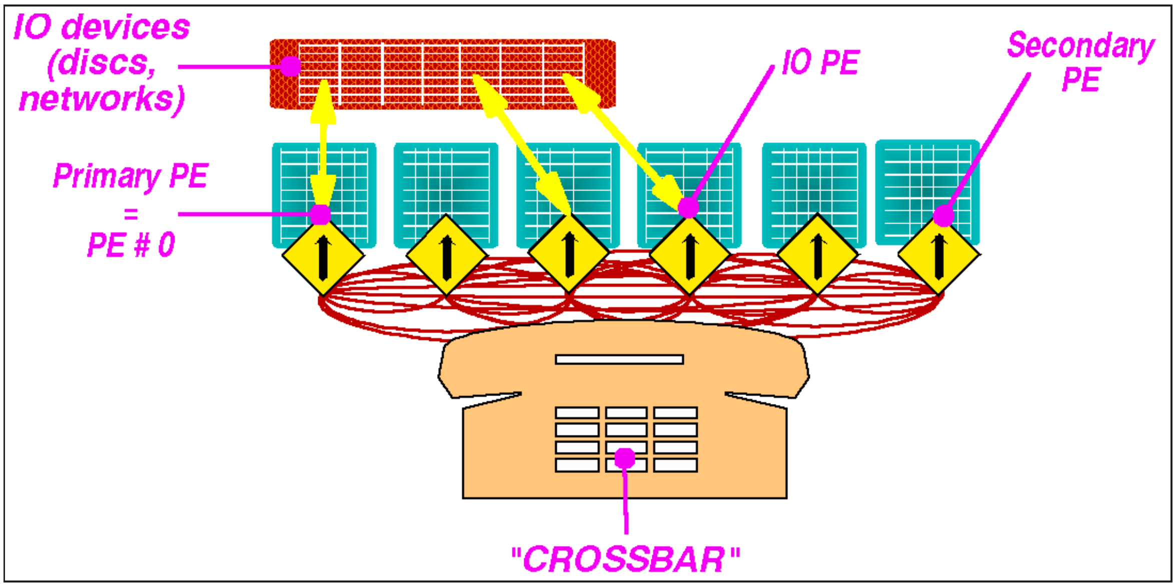


Figure 3: *Distributed Memory architecture : detailed figure (case of the Fujitsu VPP serie).*

Notices :

- In such machines the memory is larger and cheaper, . . . and also slower. Therefore, while programming on such machine, one should limit the I/O operations.
- Communications usually require some synchronization between the processors. Therefore the programmer should code optimized algorithm to limit the potential overcost due to the inter-processors communications.
- The calculation unit, which is usually a *scalar* processor unit, can contain (and this is the case of the Fujitsu VPP serie) a *vector* processor unit. In the incoming generation of high performance computers, the single processor of a node can be replaced by a cluster of scalar processors ; each of them is then called a *thread*. This kind of architecture requires more effort of programming for the developpers since some shared-memory code should be written. In principle this could be achieved with special directives to the compiler ("OPEN-MP" facility).

4 Multitasking

On shared-memory machine, we had adopted the *multitasking* programming method :

- One master-processor executes the *serial* code : setup, I/Os and synchronizations ; it scans all the parallel tasks but one, and attribute each of them to any work-free processor. Finally it runs itself a task.
- Each slave-processor is activated on a parallel task ; when its task is done it sends a signal, which helps for the synchornization and tells that it is free for another task.

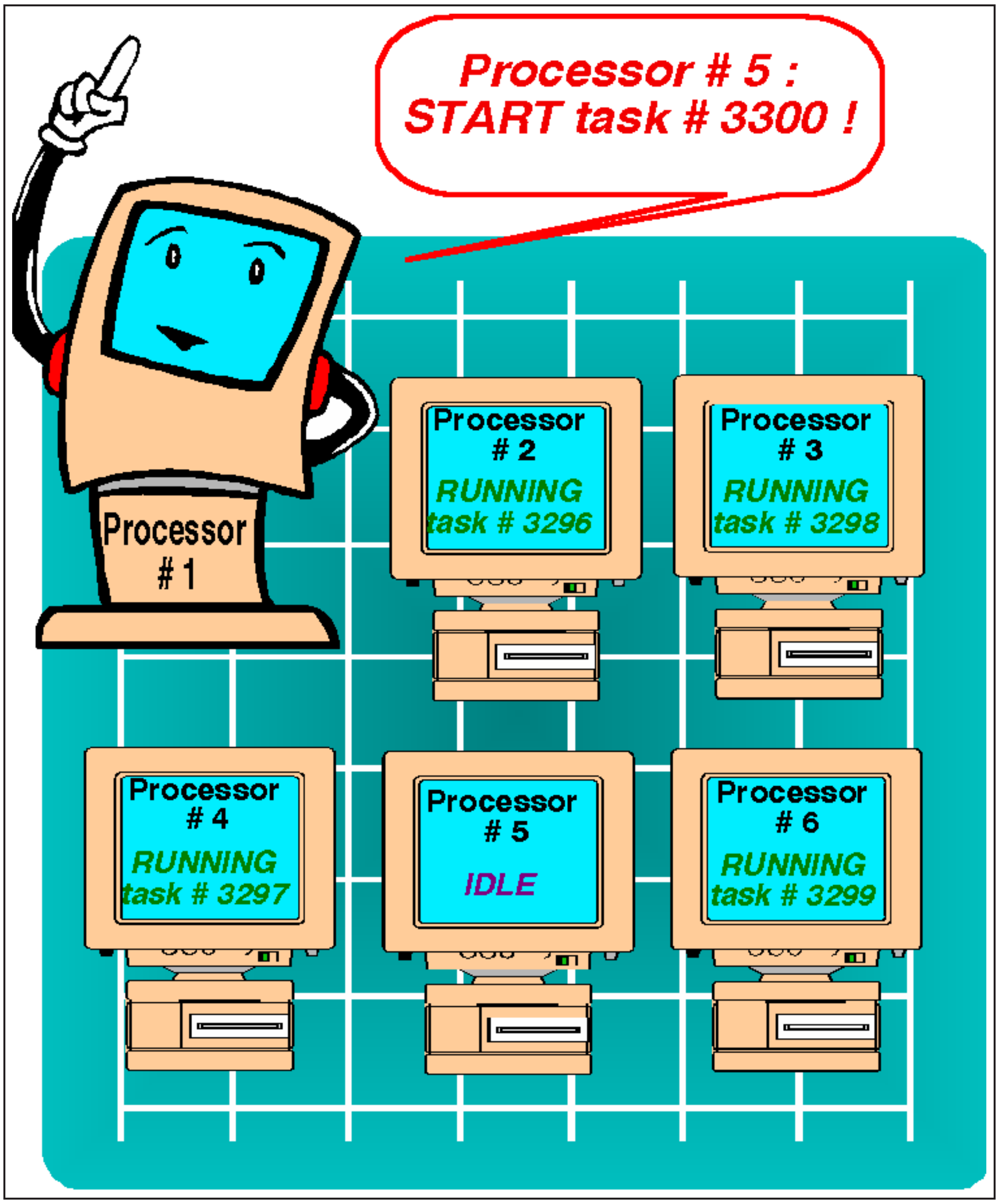


Figure 4: *Multitasking mechanism.*

Notice : the data in central memory is partitioned so that each task works on dedicated data.

5 Message Passing

On distributed-memory machine, we have adopted the *message passing* programming method :

- Each node runs through the whole code (serial or parallel parts), but on *dedicated* data.
- Each node is supposed to be independent from the others, except when it needs data dedicated to another processor ; in that case *communications* are involved.
- All nodes may not do the same jobs : for instance, when communications take place, one node is the sender to another node which is the reciever. Another example of these differenciated works concerns the I/Os :
  - we usually reserve one processor to gather data in order to write out a data file ;
  - several processors can share the work of reading and distributing input data ;
  - anyhow each node will perform computations on its dedicated data.

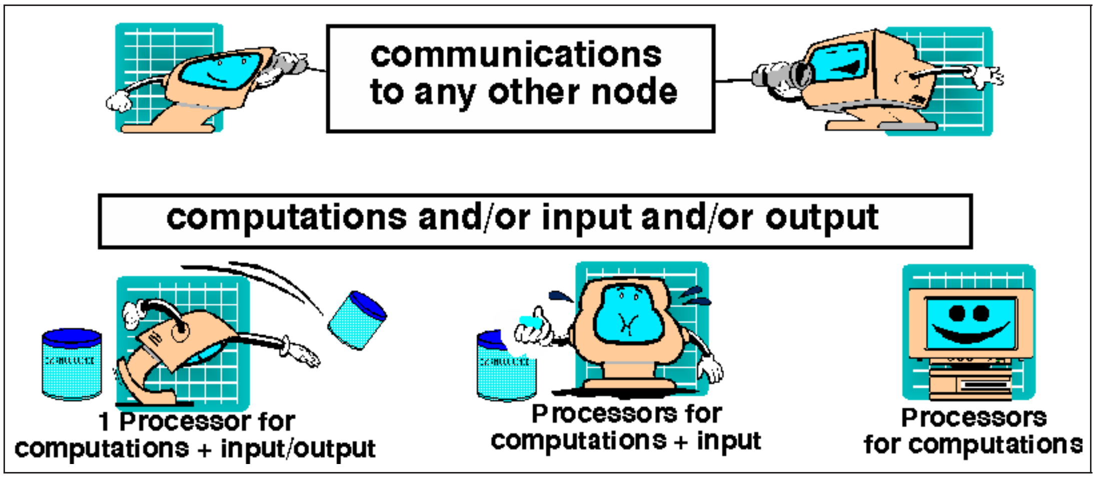


Figure 5: *Message Passing mechanism.*

Consequently the code will include conditional blocks on the processors.

6 Spectral data distribution

The spectral data are distributed per waves, in such a way that the numbers of spectral coefficients treated by every node will be as balanced as possible.

While in multitasking each task would be the treatement of a wave , in message passing each node is in charge of a subset of waves. Furthermore, the transformations from spectral space to gridpoint space and vice-versa implies a data transposition between the nodes and thus, communications.

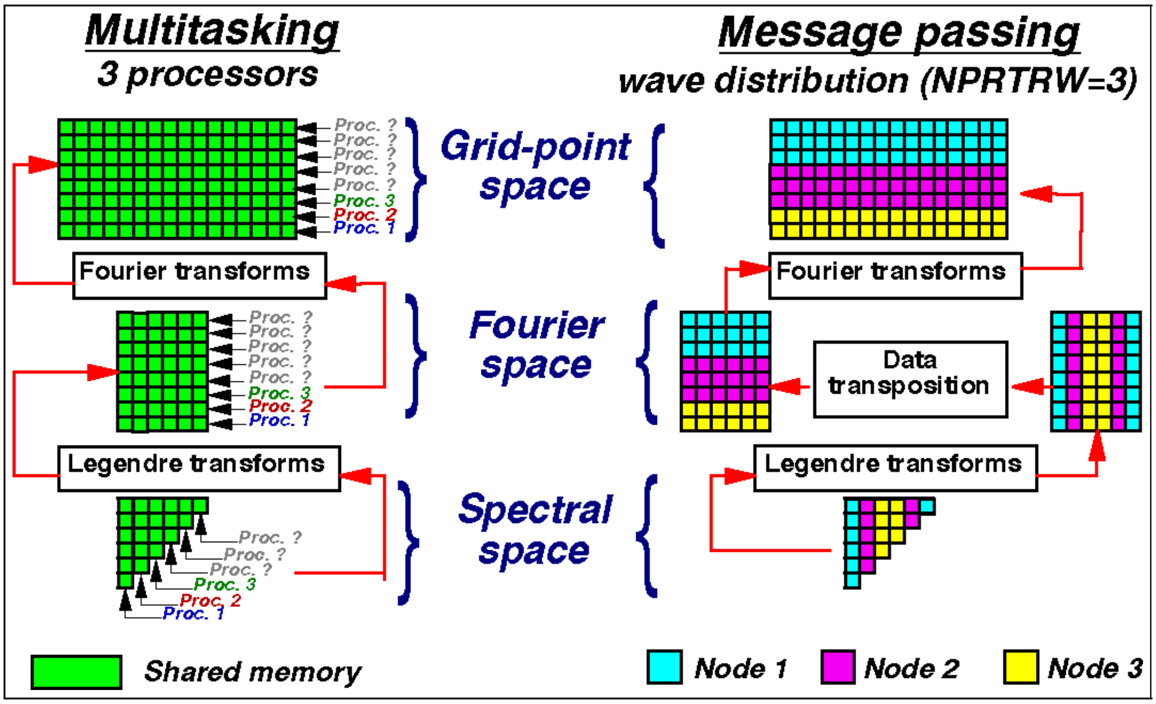


Figure 6: *Comparison between spectral data parallelization and spectral data distribution.*

7 Gridpoint data distribution

The most natural data distribution in gridpoint space for the model is a distribution per packets of latitudes (controlled by the so-called variable `NPRGPNS` for "North-South" distribution).

The distribution balance can be improved by allowing a latitude splitting, so that the numbers of gridpoints treated by each node will be all very close to one another. This is particularly important when reduced (gaussian) grid is involved.

Another possibility consists in distributing the gridpoints per packets of longitudes (controlled by the so-called variable `NPRGPEW` for "East-West" distribution) ; this is a part of the "B-level" distribution (see next subsection).

All these distribution strategies can be combined.

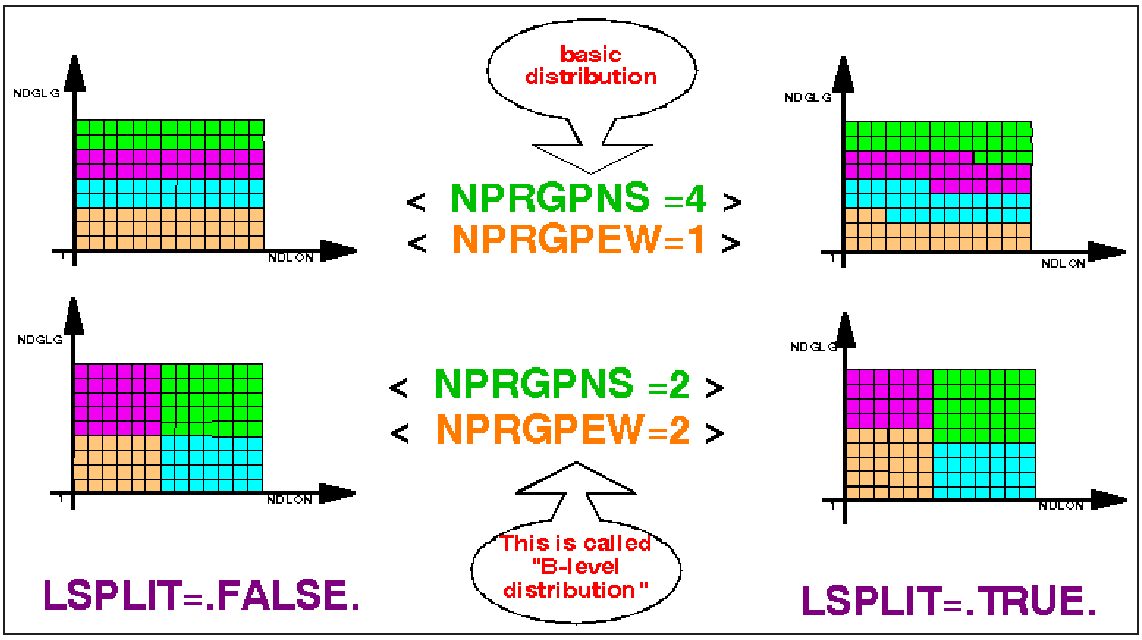


Figure 7: *Four examples of gridpoint data distribution*

8 "B-level" distribution

What is happening when the number of processors becomes huge ?

Let us consider the following example : a spectral truncation T298 linear grid with 41 vertical levels, 300 latitude rows and 600 longitude rows (reduced grid).

In spectral space the wave distribution ("A"-level distribution) limits the number of processors which can be used to the truncation value, that is 298 in the current example.
In this space the "B-level" distribution consists in the distribution of the vertical model levels. Combining both distributions, and in the current example, we would then be able to use up to 298 times 41 (=12218) processors.
However nice, this distribution is difficult to achieve because then the data distributed among the processor is not homogeneous (for instance, only one set of "A" processors will contain the surface pressure).

In gridpoint space the North-South distribution ("A"-level distribution) limits the number of processors which can be used to the number of latitudes, that is 300 in the current example.
In this space the "B-level" distribution consists in the distribution of the longitude rows (East-West distribution). Combining both distributions, and in the current example, we would then be able to use up to the total number of gridpoints, that is (for this reduced grid) 126596 processors.

In conclusion, *the "B-level" distribution enables us to reach the limits of the most powerful computers of today.*

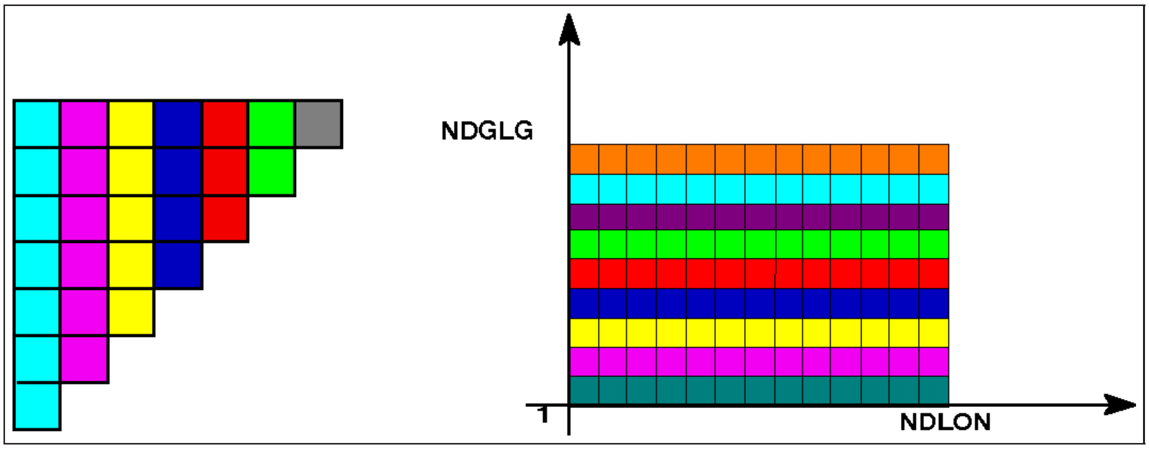


Figure 8: *Illustration showing the limits of the "A"-level" distribution (1 color per processor) in spectral space (left) and gridpoint space (right)*

9 Consequences in the code

Developping a distributed code, the programmers have to cope with a new scope of variables : the *local* ones and the *global* ones.

The local variables are those which have a local definition in the code.

The global variables are those which have a global definition in the code.

Typically, a global variable will have the same value on each node, while a local variable may have a different value from one node to another.

Another consequence of the distribution on the code is the emergence of new control keys to tune the distribution.

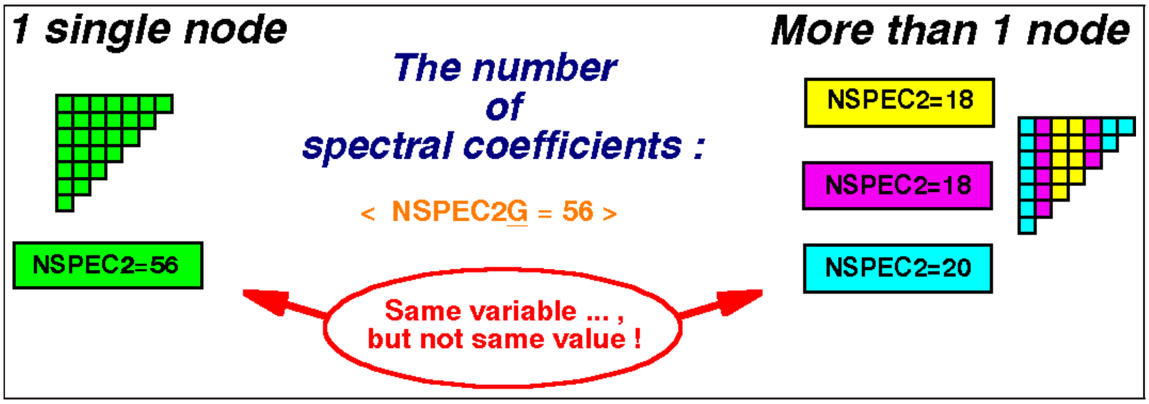


Figure 9: *Example of local versus global variable : the total number of spectral coefficients in the model is a global variable, while the number of spectral coefficients treated by a processor is local ; in this example, this local variable may have or may not have the same value as another processor.*

10 Complications in the code

Sometimes, things are getting hard !
For instance the case of latitude splitting is particularly tricky in a spectral model.

Anyhow, there are two main recommendations to assess :

KEEP THE DISTRIBUTION SPIRIT IN MIND ! You should really think *distributed* ; think and act as if you were yourself a . . . processor !

BEWARE OF THE PAST OF THE CODE ! This is the most confusing aspect of the exercise : there are still pieces of codes and variables which refer to the old shared-memory code. Furthermore the code can look even more tricky since it has been developped at a time in such a way that both shared and distributed memory codes had to co-exist. Let us hope that things will get cleaner in the future !

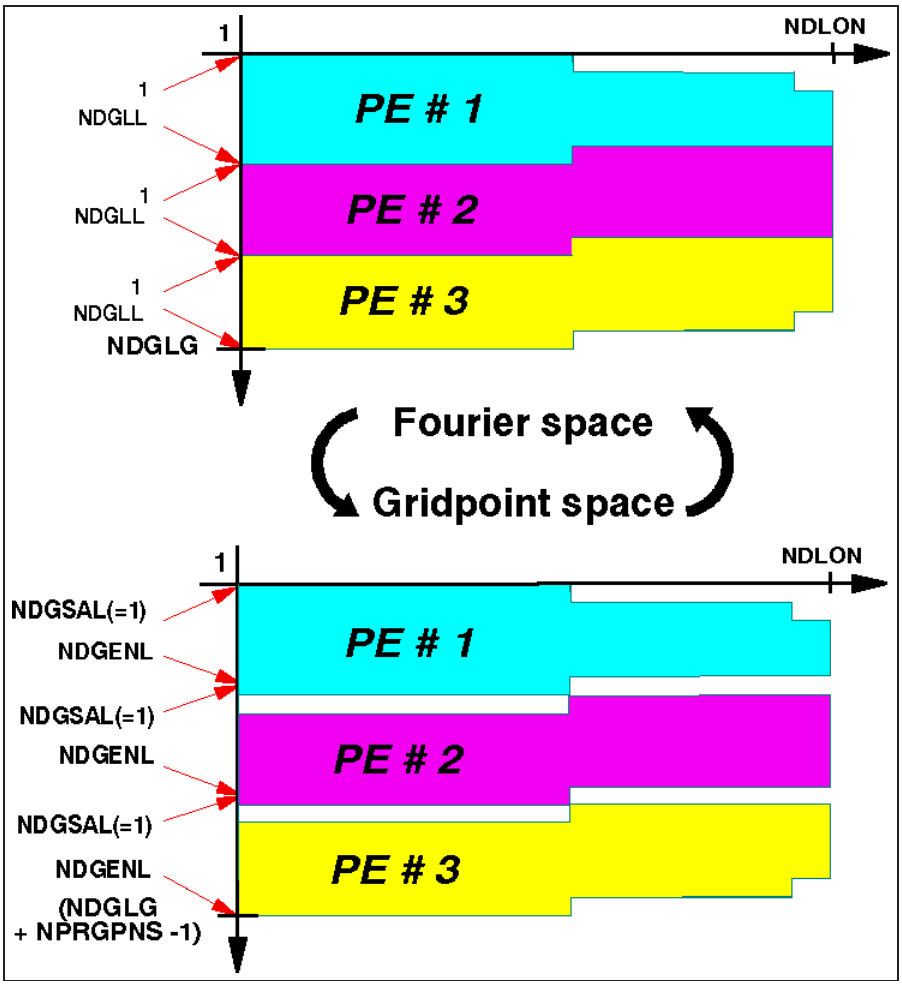


Figure 10: *The tricky case of latitude splitting : while loops go from 1 to NDGLL in Fourier space, they go from NDGSAL=1 to NDGENL (different from NDGLL) in gridpoint space ; furthermore the corresponding global variables NDGSAG and NDGENG have different meaning in shared and distributed memory code. Hence a pointer array here is dimensioned (NDGSAG:NDGENG+NPRGPNS-1) to both fit the shared and distributed memory code, while it could now be dimensioned just (NDGLG+NPRGPNS-1).*