Chapter 11

Aspects of parallellisation

11.1 Basic principles

11.1.1 Implementation of MPI

With the implementation of MPI the work load is divided among a given number N_p of processes. MPI uses non-shared memory, i.e. each process does not share its own memory with the other processes. The global domain is divided in N_p horizontal sub-domains (no decomposition in the vertical). To solve the discretised equations in the horizontal, communications need to be set up between neighbouring domains. Communications are handled by the routines of the MPI (Message Passing Interface) library (MPI, 1995). Current implementation is Version 1.1.

Advantages are:

- The program runs faster.
- Internal memory is reduced when increasing N_p.
- Land areas can be removed from the global domain by taking a sufficiently large number of processes.

Disadvantages are:

- Increasing N_p decreases the work load per process but increases the number of communications and % of time spent in communications. A maximum effficiency will be attained when $N_p = N_{max}$ depending on the application. The effect is illustrated in Figure 11.1.

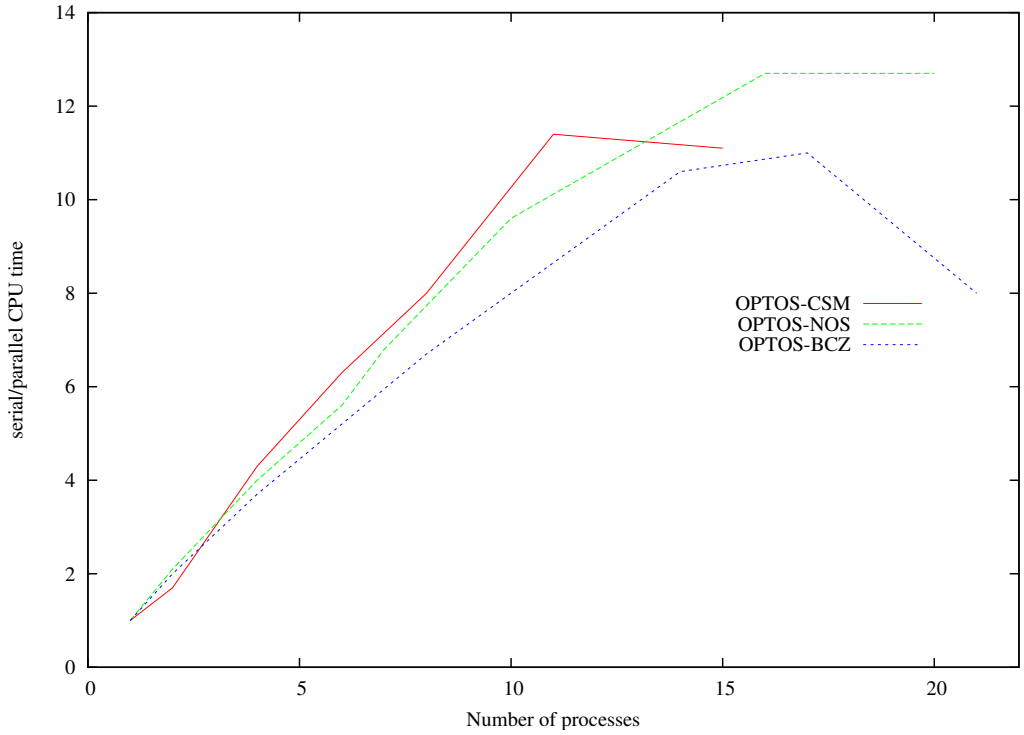

Figure 11.1: CPU efficiency, defined as the CPU time for a serial run divided by the one obtained for the parallel run, as function of the number of processes for the three ***optos*** test cases.

- Parallellisation is less efficient for 2-D (horizontally averaged) simulations and the mode-splitting scheme (vertical profiles reduce to one point, small time steps).

- Parallel code produces a slight overhead for serial applications.

MPI routine calls are only found in the file *comms_MPI.F90* [1]. Each MPI routine call (starting with MPI_) has a corresponding alias (starting with comms_). This allows a more efficient implementation of future MPI versions.

[1]Sole exception is the MPI_abort call in *error_routines.F90*. Since MPI is based upon FORTRAN 77, MPI parameters are declared and defined in an "include" file instead of a module file. Instead of a USE a INCLUDE mpif.h statement has to be made inside the code, contrary to the programming rules stated in Section 8.1.1. This will be removed in future MPI implementations.

11.1.2 Principles of the parallel code

Variables (parameters and arrays) on a parallel grid can either be:

global defined on each sub-domain with the same value

local either defined on each sub-domain but with different values or defined on one or more (but not all) sub-domains

A specific aspect of parallellisation is that different effects are or may be procuced by the model code on different domains:

- A variable may be defined on one and non-defined on another domain.
- Arrays can be either allocated, deallocated or undefined on different domains.
- Local arrays usually have different shapes (or may be of size 0) on different domains.
- An IF statements may evaluate as .TRUE. on one or .FALSE. on another sub-domain if the expression contains local variables.

The basic rule for an efficient parallel code is that the work load is (as much as possible) equally spread among the processes. An important (but unevitable) exception is that output operations (except for log and error files) are restricted to one process, called the "master" process. On the other hand, all processes are allowed to read from the same file, avoiding to copy the input data from one particular "reader" process.

11.2 Domain decomposition

11.2.1 Definition

Process domains are arranged as a 2-D "parallel" grid with dimensions nprocsx and nprocsy. Domains, solely composed of land points, can be excluded from the grid so that nprocsx×nprocsy≥nprocs where nprocs is the number of "effective" processes. Each domain has a process id number, between 0 and nprocs-1, which is assigned by MPI, and two domain grid indices (i,j) where 1≤i≤nprocsx and 1≤j≤nprocsy. Dummy (land) domains are defined with a NULL process id (MPI_proc_null). Work load is as much as evenly partitioned between the sub-domains (since the sub-domains are not of the same size). Exception is the master process which is the only one with write access. Reading is performed by all processes (except defined otherwise by the user).

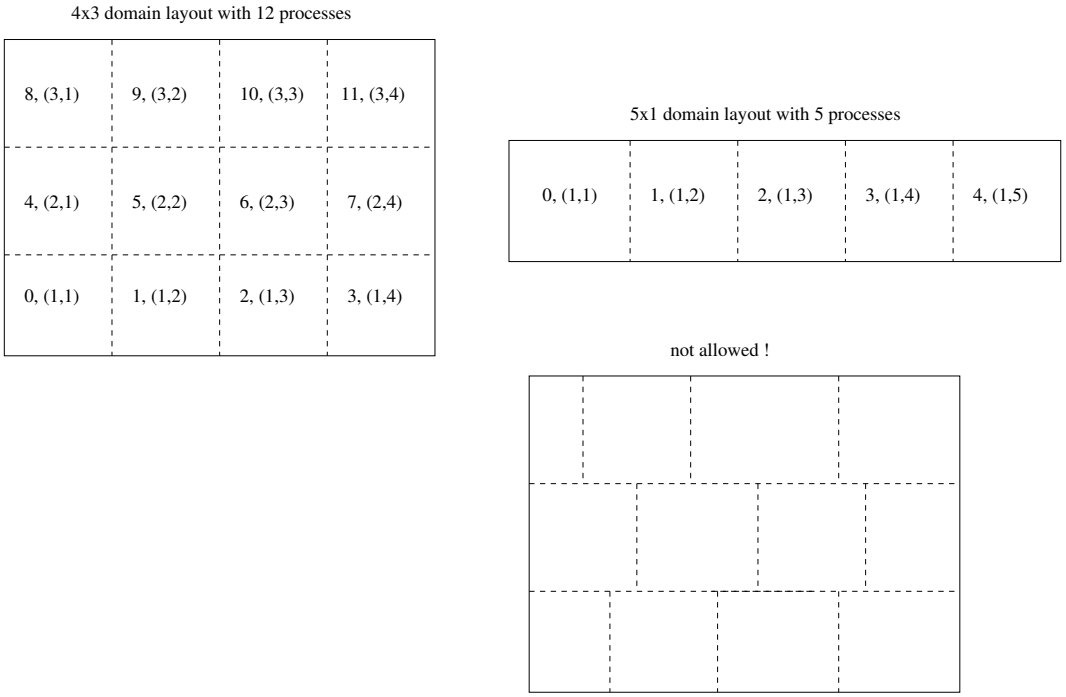

Figure 11.2: Examples of allowed and non-allowed domain decompositions. The first number is the process id, the two numbers in parentheses denote the domain indices.

Some examples of simple domain decompositions are illustrated in Figure 11.2. Figure 11.3 shows an example of a decomposition where part of the land areas have been removed.

11.2.2 Local grid indexing system

The grid indexing system for a local sub-domain, shown in Figure 11.4, is practically the same as the one taken on the global grid (see Figure 5.1). Main differences are:

- The local grid dimensions are now ncloc and nrloc, instead of nc and nr.
- The most eastern column and most northern row are no longer composed of dummy land points (except when they are located at the edges of the global computational domain).

The local grid dimensions and indices are related to the global ones through the following definitions:

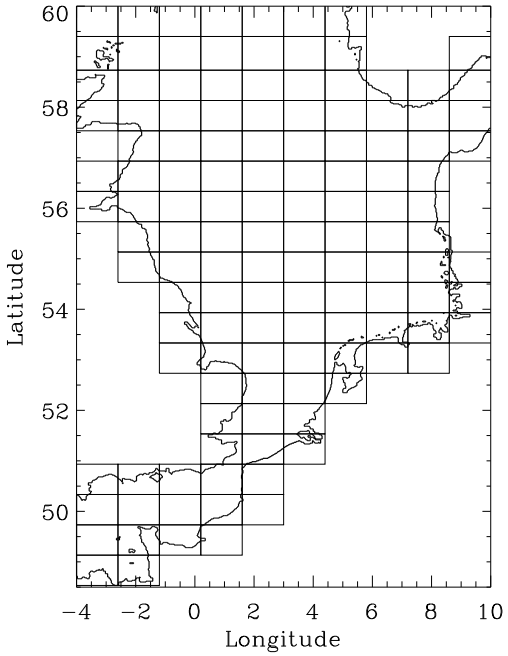

Figure 11.3: Domain decomposition for the North Sea area with 128 processes based on a 10×19 domain grid.

ncloc	X-dimension of the local domain (local variant of nc)
nrloc	Y-dimension of the local domain (local variant of nr)
nc1loc	global X-index of the leftmost column (local)
nc2loc	global X-index of the rightmost column (local)
nr1loc	global Y-index of the lowest row (local)
nr2loc	global Y-index of the highest row (local)
nc1procs(nprocs)	array with the values of nc1loc for each domain (global)
nc2procs(nprocs)	array with the values of nc2loc for each domain (global)
nr1procs(nprocs)	array with the values of nr1loc for each domain (global)
nr2procs(nprocs)	array with the values of nr2loc for each domain (global)

Global and local indices (at any node) are then related by

```
iglb = iloc + nc1loc - 1
jglb = jloc + nr1loc - 1
```

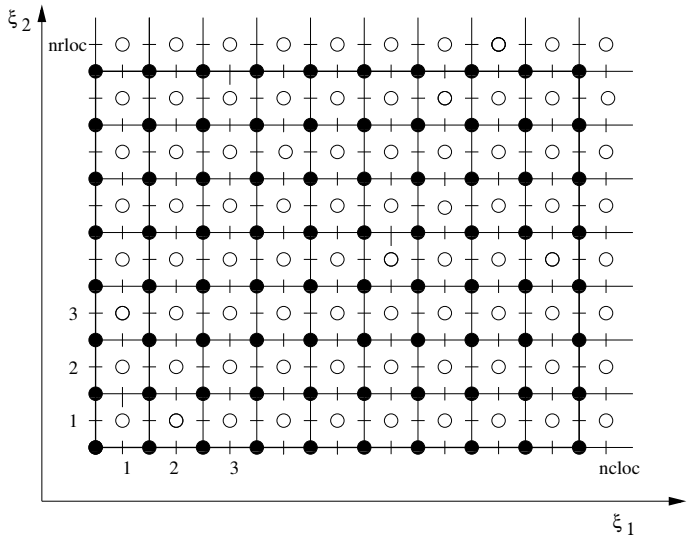

Figure 11.4: Horizontal layout of the computational grid on a local subdomain. Different nodes are indicated by the same symbols as in Figure 5.1.

Domain decomposition is uniquely defined by these last 4 "process" arrays. Two options are available, depending on the value of the switch iopt_MPI_partit:

1 Simple decomposition: the user specifies the values of two of the three parameters nprocs, nprocsx, nprocsy. The arrays are defined internally (see Section 14.3 for further details).

2 The arrays are defined externally by the user.

Other (global or local) parameters used in the model for parallel application are:

parallel_set	user-defined parameter to switch on/off the parallel mode (global)
idloc	local process id (local)
iprocloc	local process number = idloc+1 (local)
idmaster	Process id of the master process (global). Its value can be changed by the user. Default is 0.
idprocs(nprocs)	vector of local process ids (global)
master	.TRUE. if idloc equals idmaster, .FALSE. otherwise (local)
shared_read	user-defined parameter enabling reading by all processes if .TRUE. (global)

comm_work MPI communicator composed of all "working" processes (global)

iddomain(0:nprocsx+1,0:nprocsy+1) process id as function of domain grid indices (global)

11.3 Halos

Numerical discretisations in the horizontal and horizontal averaging may require the availability of values of a model grid array located at grid points within a neighbouring sub-domain. These values are calculated, not inside the domain itself, but within its neighbours. They are obtained by establishing MPI communications between the domain and its neighbours. Each of its neighbours sends an internal section of the array which is received and stored by the domain in one of the local array's halo sections (see Figure 11.5 and Figure 11.6).

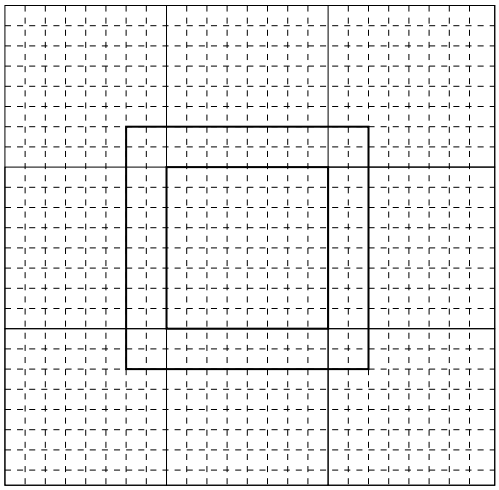

Figure 11.5: Illustration of an halo for an array defined on a local sub-domain. The halo of the sub-domain, located in the inner rectangle, is situated between the inner and outer rectangles.

A halo is created by adding extra columns and rows to the model array. Most arrays have halo sizes which are all equal to the parameter nhalo = 2, e.g.

```
uvel(1-nhalo:ncloc+nhalo,1-nhalo:nrloc+nhalo,nz)
```

Other arrays may have smaller halos or asymmetric halos, e.g.

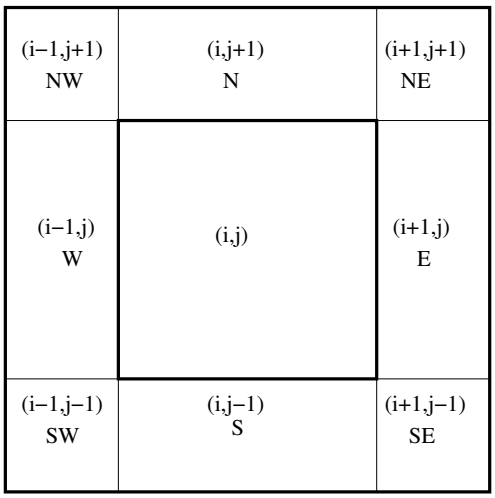

Figure 11.6: Partitioning of a sub-domain halo. Domain indices are given in parentheses.

```
gxcoord(0:ncloc+1,0:nrloc+1), atmpres(0:ncloc,0:nrloc)
```

The West/East/South/North sizes are usually given by a 4-element vector. The halo sizes of the 3 arrays above are respectively

```
(nhalo,nhalo,nhalo,nhalo), (1,1,1,1), (1,0,1,0)
```

Halo sizes of a model array are the same on all sub-domains. The corresponding global arrays (i.e. defined over the whole domain) are declared with the same halo sizes as their local counterparts. Halos of sub-domains which are located at the edge of the (full) computational domain, extend to the outside of the computational domain. These outside points are taken as dummy land points belonging to a dummy outside sub-domain with NULL process id.

11.4 Communications

11.4.1 Send and receive in MPI

A MPI communication consist of a send operation on one process (say `A`) and a receive operation on another process (say `B`). COHERENS uses only so-called "blocking" send and receive operations. The implementation in COHERENS is based upon the following communication modes:

1. Synchronous send.

   - A sends a message with the data to B.
   - B receives the message and the data, sends a message back to A and completes next.
   - A receives the message from B and completes.

   This is the most robust mode.

2. Bufffered send.

   - A sends a message to B, sends the data to a buffer, and completes next.
   - B receives the message, retrieves the data from the buffer and completes next.

   This mode is useful for transfering a large amount of data.

3. Standard send.

   - The operation is performed either synchronously or in buffered mode. The choice is made internally by MPI .
   - Less robust than synchronous mode but usually more efficient.

Only the standard and synchronous mode are implemented in COHERENS.

1. Syntax of a standard send

```
SUBROUTINE MPI_send(buf,count,datatype,dest,tag,comm,ierror)
```

<type>, INTENT(IN), DIMENSION(*) :: buf	initial address of the send buffer
INTEGER, INTENT(IN) :: count	number of elements in the send buffer
INTEGER, INTENT(IN) :: datatype	type of data in the send buffer (e.g. MPI_REAL for real data)
INTEGER, INTENT(IN) :: dest	rank (process id) of the destination process
INTEGER, INTENT(IN) :: tag	message tag
INTEGER, INTENT(IN) :: comm	communicator (usually comm_work)
INTEGER, INTENT(OUT) :: ierror	returned error code.

2. Syntax of a synchronous send

```
SUBROUTINE MPI_ssend(buf,count,datatype,dest,tag,comm,ierror)
```

where the arguments have the same meaning as before.

3. Syntax of a receive operation

```
SUBROUTINE MPI_recv(buf,count,datatype,source,tag,comm,&
                  & status,ierror)
```

<type>, INTENT(OUT), DIMENSION(*) :: buf	initial address of the receive buffer
INTEGER, INTENT(IN) :: count	number of elements in the receive buffer
INTEGER, INTENT(IN) :: datatype	type of the data in the receive buffer (.e.g. MPI_INT for integer data)
INTEGER, INTENT(IN) :: source	rank (process id) of the source process
INTEGER, INTENT(IN) :: tag	message tag
INTEGER, INTENT(IN) :: comm	communicator (usually comm_work)
INTEGER, INTENT(OUT), STATUS(MPI_STATUS_SIZE) :: status	return status
INTEGER, INTENT(OUT) :: ierror	returned error code.

In the COHERENS code, send and receive operations are performed using the comms_send_* and comms_recv_* routines defined in *comms_MPI.F90*.

11.4.2 Sort of communications

The following sort of communications are used in the program: copy, distribute, combine, combine all, exchange, collect.

1. Copy operations.
   - Copies (the same) data from a root (usually the master) process to all other processes.
   - The operation involves N_p-1 sends from the root and 1 receive at each other process.
   - This is called a "one-to-all" operation and therefore asymmetric.

- Used to copy data read by the root process from a data file (not needed if shared_read = .TRUE.).

2. Distribute operations.

   - Copies (i.e. distributes) the local parts of a global model array from the root process to all other sub-domains. Each local section may or may not contain the array's local halo parts.
   - The operation involves N_p-1 sends from the root and 1 receive at each other process.
   - This is called a "one-to-all" operation and therefore asymmetric.
   - Used to set up initial conditions on each local domain or to distribute surface data in case the surface data grid coincides with the model grid. Distribute operations are redundant if shared_read = .TRUE..

3. Combine operations.

   - Copies (i.e. combines) a local array from each sub-domain to a corresponding section of a global array on the root process. Halos are not included.
   - The operation involves N_p-1 receives at the root and 1 send from each other process.
   - This is called a "all-to-one" operation and therefore asymmetric.
   - Various versions are implemented:
     - Combination of local "full" model arrays into a global "full" array.
     - Combination of local subsections of model arrays into a global array.
     - Combination of local irregular (station) data into a global array.
     - Combination of local open boundary arrays into a global boundary array.
   - Used for the construction of global arrays, obtained from local model arrays, local regular or irregular sub-arrays. The global arrays are mostly intended for output, the calculation of global array sums, array maxima and minima.

4. Combine-all operations.

- The same as the combine operation except that the global data are made available to all processes.
- The operation involves N_p-1 sends and N_p-1 receives on each process.
- This is called a "all-to-all" operation and therefore symmetric. Note that work load is (about) the same for each sub-domain so that the CPU time for a combine-all is about the same as for a corresponding combine operation.

5. Exchange operations.

   - Send sub-sections of local model arrays to a corresponding section within the halo of each neighbouring domain. Receives data from each neighbouring domain and stores these data in one of its own halo sections.
   - Since each sub-domain has 8 neighbours (including dummy domains outside the computational domain), the operation requires in general 8 sends and 8 receives on each sub-domain. However, this number can be reduced by specifying the halo sections for which an exchange is needed or when component(s) of the halo size vector is (are) zero.
   - The operation is symmetric.
   - Exchange operations are an essential part of the parallel code and are mainly for implementation of numerical algorithms for horizontal advection and diffusion, and for horizontal averaging.

6. Collect operations.

   - Stores all local arrays into one global array with an extra dimension of size nprocs.
   - The operation involves N_p-1 sends and N_p-1 receives on each domain.
   - This is a "all-to-all" operation and therefore symmetric.
   - The operation is not frequently used.

Remarks:

- The root process is by default the master process.
- The communication routines are programmed using the MPI_send and MPI_recv routines either in standard or synchronous mode.

- There are options to use MPI collective calls instead for some operations: MPI_bcast for copy and MPI_allgather for collect operations.

- For exchange operations there is an alternative option to use the MPI_sendrecv utility routine which combine a send and a receive operation into one call.

11.4.3 Implementation

The main difficulty in programming MPI communications is to prevent so-called dead locks, i.e. a send/receive operation cannot terminate because one of the processes is engaged in another communication which cannot terminate as well and so on. The problem does not arise for the asymmetric "one-to-all" and "all-to-one" operations since each process is engaged in either a send or a receive operation but not both. The problem is more severe for "all-to-all" and exchange operations. Implementation of these two type of operations is discussed below.

11.4.3.1 all-to-all operations

Firstly, an order of operations is defined for each process via a $2N_p \times N_p$ array. The elements are either -1, 0, 1, 2. Taking the example of N_p =4, the array is defined as follows (with easy extension for a general value of N_p):

$$\begin{pmatrix} 0 & 2 & 2 & 2 \\ 1 & 0 & 2 & 2 \\ 1 & 1 & 0 & 2 \\ 1 & 1 & 1 & 0 \\ -1 & 1 & 1 & 1 \\ 2 & -1 & 1 & 1 \\ 2 & 2 & -1 & 1 \\ 2 & 2 & 2 & -1 \end{pmatrix}$$

where each column (except the first) is obtained by shifting the previous one donwwards by 1 position. The first column represents the type and order of the communications performed by process 0, the second by process 1 and so on. The numbers present the kind of the operations to be performed:

-1: Dummy operation. Nothing is done.

0: The process "communicates" with itself, the local data are stored directly into the global array.

1: Send operation.

2: Receive operation.

Secondly, the source or destination of each communication needs to be defined. The process ids are taken from the following vector (of size $2N_p$): $(0, 1, 2, 3, 0, 1, 2, 3)^T$. Replacing the one and twos in the previous array by respectively S and R, and inserting the process ids into each column of the previous array, all communications can be symbolically presented by the matrix

$$\begin{pmatrix} 0 & R0 & R0 & R0 \\ S1 & 0 & R1 & R1 \\ S2 & S2 & 0 & R2 \\ S3 & S3 & S3 & 0 \\ -1 & S0 & S0 & S0 \\ R1 & -1 & S1 & S1 \\ R2 & R2 & -1 & S2 \\ R3 & R3 & R3 & -1 \end{pmatrix}$$

where the first column aplies to process 0, the second column to process 1 and so on. The numbers 0 and -1 have the same meaning as before, Si means send to process i and Ri receive from process i. For example, process 2 (third column) performs the following operations : receive from process 0, receive from 1, internal storage, send to 3, send to 0, send to 1, nothing, receive from 3. It can easily be shown that the order of communications, defined in this way, can never produce a dead lock, since for each send/receive the destination/source process will always be available to receive/send the data.

11.4.3.2 exchange operations

Consider as example a 4×4 decomposition which is arranged in a chessboard pattern as shown in Figure 11.7.

Each domain has to communicate with its 8 neighbours. Dummy domains (with process id MPI_proc_null) are added outside the computational domain (not shown). Each communication has a specified direction (W, E, S, N, SW, NE, NW, SE, as shown in Figure 11.6). The 8 sends and receives are organised in 16 steps. The first four deal with W/E communications.

1: Black sends W, white receives E.

2: Black receives E, white sends W.

3: Black sends E, white receives W.

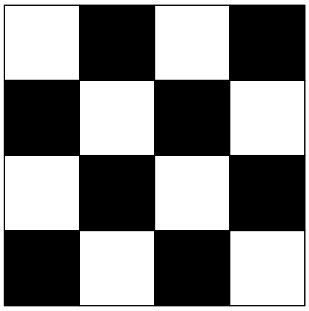

Figure 11.7: Communication pattern for exchange operations.

4: Black receives W, white sends E.

5-8: The same now for S/N directions.

9-12: The same now for SW/NE directions.

13-16: The same now for SE/NW directions.

The actual number of communications depends on the sizes of the halo. For example, an array with halo size (0,2,2,2) has no western halo so that steps 3, 4, 11, 12, 13, 14 become unnecessary. There is also an option foreseen in the exchange call to skip all corner communications (steps 9 to 16).

11.4.3.3 program routines for communications

The following generic communication routines have been implemented in the code. For a discussion of the FORTRAN syntax see Section 31.17.

collect_vars	collect operations
combine_mod	combine operations on full model grid arrays
combine_obc	combine operations on open boundary arrays
combine_stats	combine operations on irregular (stations) arrays
combine_submod	combine operations on sub-sections of full model grid arrays
copy_chars	copy operation on character data
copy_vars	copy operation on numerical data
distribute_mod	distribute operation on model grid arrays
exchange_mod	exchange operation on model grid arrays.

11.5 Local versus global array indexing

Since the parallel decomposition uses non-shared memory, arrays exist only on a local basis. In some cases, one needs to store all these local components into some global array. The question is how to relate a local array element with the corresponding element in the global array. The answer is so-called index mapping, which maps local array indices into the corresponding global ones.

For arrays defined on the model grid, this mapping has a simple form and follows from the definition of the domain decomposition itself. Assume that (iloc,jloc) are the local and (iglb,jglb) its corresponding global indices. They are related by

```
iglb = iloc + nc1loc - 1
jglb = jloc + nr1loc - 1
```

The solution is less obvious for arrays not indexed by positions on the model grid. Consider the example shown in Figure 11.8. The domain is decomposed in 4 sub-domains.

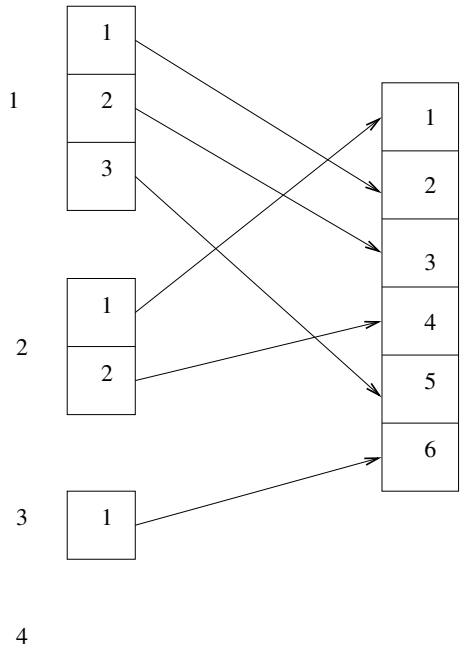

Figure 11.8: Example of index mapping between global and local arrays.

There are 6 data points in total of which 3 are located in the domain with process number 1, 2 in domain 2, 1 in domain 3 and none in domain

4. The 3 elements of domain 1 map into elements (2,3,5) of the global array, the 2 elements of domain 2 into (1,4), the element from domain 3 into global element 6. No mapping is obviously needed for domain 4. The mapping can be programmed as follows

```
INTEGER :: iproc, maxstats, nostatsglb, nostatsloc
INTEGER, DIMENSION(nprocs) :: nostatsprocs
INTEGER, DIMENSION(:,:) :: lstatsprocs

nostatsglb = 6
nostatsprocs = (/3,2,1,0/)
iproc_110: DO iproc = 1,nprocs
   IF (idloc.EQ.idprocs(iproc)) THEN
      nostatsloc = nostatsprocs(iproc)
   ENDIF
ENDDO iproc_110
maxstats = MAXVAL(nostatsprocs)
ALLOCATE (lstatsprocs(nprocs,maxstats))
lstatsprocs(1,1:3) = (/2,3,5/)
lstatsprocs(2,1:2) = (/1,4/)
lstatsprocs(3,1) = 6
```

where nostatsloc, nostatsglb are the local and global number of data points, nostatsprocs an array with the number of data points per process and lstatsprocs the array defining the index mapping. If, for example, the data points represent stations used for output, the local data must be combined first into the global array using the index mapping array. The situation is more complex in practice since the number of points per domain are unknown initially. Usually, these points have corresponding geographical positions which allows to determine their distribution over the different sub-domains.

The second example is related to the indexing of open boundary locations. The following definitions are made

nobu	number of global U-open boundary points (global)
nobuloc	number of local U-open boundary points (local)
westobu(nobu)	.TRUE. (.FALSE.) for U-open boundary points at western (eastern) boundaries (global)
iobuloc(nobuloc)	local X-index of U-open boundary points on the local grid (local)
jobuloc(nobuloc)	local Y-index of U-open boundary points on the local grid (local)

indexobu(nobuloc)	global indexes of local U-boundary points (local)
indexobuprocs(nobu,nprocs)	global indexes of local U-boundary points per domain (global)

The parameter nobuloc and the last 3 arrays are defined in routine open_boundary_arrays (file *Grid_Arrays.F90*). Consider the following code

```
iiloc_110: DO iiloc=1,nobuloc
   i = iobuloc(iiloc); j = jobuloc(iiloc)
   ii = indexobu(iiloc)
   IF (westobu(ii)) THEN
      ....
   ENDIF
ENDDO iiloc_110
```

The IF statement determines whether a local open boundary points is located at a western or eastern boundary.

Index mapping is used in the program for local versus global indexing of:

- open boundary arrays
- output arrays of model data on an irregular grid
- the positions of the open boundary points belonging to nested sub-grids