Message-Passing Tools for Structured Grid Communications

User's Guide
Version 2.0

Andrei Malevsky

Centre de Recherche en Calcul Appliqué
5160, boul. Décarie, bureau 400
Montréal, Québec H3X 2H9 CANADA

`malevsky@cerca.umontreal.ca`

Date published: May 13, 1997

Contents

1 What is the purpose of the MSG ?

Message-passing tools for Structured Grid communications (MSG) is a MPI-based library intended to simplify coding of data exchange within the FORTRAN 77 codes performing data transfers on distributed Cartesian grids. The FORTRAN 77 binding of the current version is essential since all the MSG routines assume the FORTRAN 77 arrays order (first index changes first). The main goal of the library is to conceal the explicit send/receive operations and provide means to place the boundary data into local arrays. The MSG tools can be employed by finite-difference, finite-volume, or finite-element codes which use Cartesian (tensor-product) grids. According to these discretization techniques, a computational domain is covered by a global grid. The global grid is divided between the processors, and the processors hold their local subgrids or subdomains. In the course of calculations, the processors exchange the information on the subdomain boundaries with their neighbors.

The MSG toolkit is intended to assist in efficient and uniform implementation of domain-decomposition based solvers on distributed memory platforms. It can be used with any other MPI-based library. The MSG data transfer routines operate within the scope of their own communicator (see [1]). The main objective of the library is to hide cumbersome operations needed to place boundary data into the faces of the local arrays during boundary exchange between subdomaines. Patterns of these data transfers depend on the connectivity of nodes of the global grid. They also depend on the connectivity of the nodes located on the global grid boundaries. For instance, a finite-difference discretization with the periodic boundary condition would require a "wrap-around" communication pattern where the last processor exchanges data with the first processor. A user of the MSG toolkit specifies the grid coordinates (indices of the global grid) of the halo regions, the dimensions of the global and local arrays, and the grid coordinates of the subdomains. Then a MSG setup routine automatically establishes the processor connectivity. In order to perform the boundary exchange efficiently, the MSG toolkit exploits the two features of message passing, the nonblocking message passing capabilities and the redundancy of communications. A nonblocking message passing means that a processor can send data into the network and continue to perform some work without waiting for the data to arrive to its destination. If the data transfer follows a repeated pattern, then this redundancy is exploited to minimize the latency of communications.

The necessity to rewrite existing sequential FORTRAN 77 codes in the MIMD style was the main reason behind the MSG project. The developers of these codes needed to have efficient data exchange tools while preserving their code and data structures without learning all the details of message-passing. The MSG library was employed by the distributed-memory version of the MC2 regional atmospheric model [2]. The use of MSG toolkit allowed to avoid an enormous amount of work needed to recode the inter-processor communications when the topology of data decomposition was modified.

2 Data transfers on distributed Cartesian grids.

Cartesian or tensor-product grids are common in finite-difference, spectral, and finite-volume schemes. A node of a tensor-product grid is defined by a vector of indices with one index per dimension. We assume here that the nodes of the global grid are divided between the processors in such a way that every node is assigned to one and only one processor. It does not prevent the MSG functions from being used within the framework of a domain-decomposition scheme employing overlapping subdomains. The nodes belonging to multiple subdomains would simply find themselves within the halo regions of the corresponding processors.

A topology of inter-processor communications is determined by the node connectivity. In the finite-difference applications the node connectivity is prescribed by a finite-difference stencil. A full (tensor-product) stencil means the full connectivity within a rectangular prism surrounding a node. A dimension of the rectangular prism is called "stencil width". The "widths" would determine the thicknesses of the halo regions needed to accommodate data from other processors.

Each processor holds its own subdomain. A subdomain is supposed to be a rectangular prism surrounded by a larger prism containing the halo which is to be filled with the other processors data. The coordinates of the corners of these prisms are used by the MSG setup routines to establish the patterns of data exchange. A partial (e.g. cross-shaped stencil) can not be described by means of a single rectangular prism. Nevertheless it can be defined as a superposition of several prisms. For example, a node in the second-order central finite difference approximation of the 2D Poisson operator needs information from the nodes located north, south, east, and west from it. This stencil is not full but can be described in two steps. The prism extended in the vertical direction would define the north-south data exchange pattern and the prism extended in the horizontal direction would define the east-west pattern.

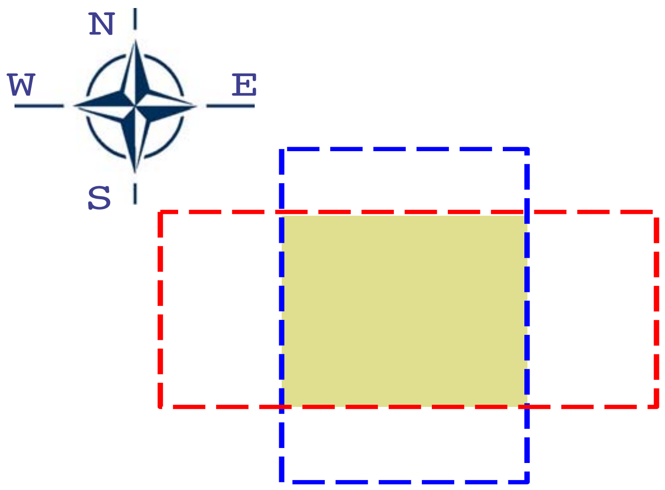

Figure 1: A cross-shaped area is obtained by superimposing two rectangles with one being extended in the north-south direction and another in east-west direction.

The geometry of the 2D cross-shaped stencil is illustrated in the Figure 1. The cross-shaped area can be described by means of two superimposed rectangles. Each of these two rectangles can be specified by the coordinates of its four corners.

From the local viewpoint (for a processor holding its own subgrid) a node of the global grid can belong to one of the four following classes:

- internal nodes;
- local interface nodes;
- external interface nodes;
- nodes not connected to the processor.

We define below these classes.

Def. 1 *A node of the global grid is called internal to a processor (subdomain) if it is connected to its other internal nodes and its local interface nodes only.*

Def. 2 *A node of the global grid belongs to the local interface if it is connected to the internal nodes and the external interface nodes.*

Def. 3 *A node of the global grid belongs to the external interface if it is connected to the local interface nodes and is not connected to the internal nodes.*

Both the internal and the local interface nodes are local nodes. Local nodes compose a local subgrid. While the internal nodes lie in the interior of the subdomain the local interface nodes are connected to the outside world (to nodes of other subdomains). The external interface is also called a halo. The nodes of the halo region belong to the other processors but are connected to nodes assigned to the given processor. In the course of data exchange a processor sends out its local interface and receives the external interface.

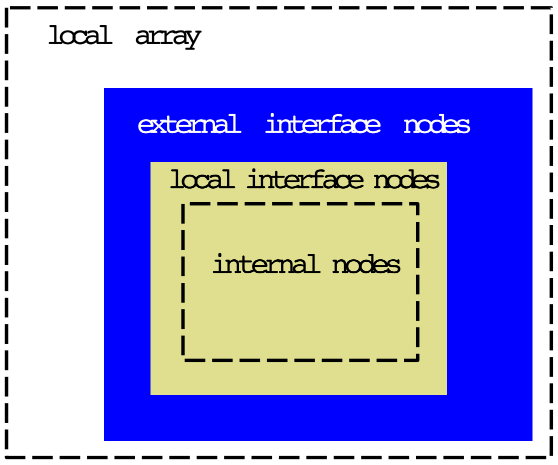

Figure 2: Local data structure (multidimensional array) must accommodate the local data and the external interface data (halo). The dimensions of local array can be the same or larger that those imposed by the dimensions of the local subgrid and by the thickness of the halo region.

We consider here only the decompositions of a Cartesian grid into rectangular prisms. Decompositions into subdomains of other shapes are not covered by the

MSG toolkit. Thus a local subgrid is a rectangle surrounded by the external interface (halo). A local data structure is illustrated in the Figure 2. The internal nodes form the area inside encircled by the local interface layer which is in turn surrounded by the halo. The local interface layer is sent out and the halo is filled with the values obtained from the neighboring processors. Both the local data and the external interface data must be stored within the local array. The local array can be of the same size or larger than the area contoured by the external interface layer. The number of dimensions of the local array can be the same or larger than the dimensionality of the global grid.

3 How to use the MSG functions within a code.

The MSG must be initialized before any calls to MSG functions are made. The function `MSG_enable` initializes the MSG toolkit. If the MPI has not been initialized before the call to `MSG_enable` the latter initializes the MPI. The call to `MSG_disable` disables the MSG toolkit. It does not replace the MPI clean-up function `MPI_Finalize` which must be executed after the MSG clean-up `MSG_disable`. A failure to call `MPI_Finalize` may result in a processor hangup.

A program employing MSG functions to perform an exchange of boundary information between the subdomains within a loop of iterations typically incorporates the following sequence of calls.

```
call MSG_tsetup(...)         -- sets up the tables of indices
...
do iter=1,niter
  ...
  call MSG_tbdx_send(...,ptrn,...) -- the first call opens
                                      a communication pattern
                                      indicated by ptrn and
                                      sends the boundary data
                                      to the network; each
                                      subsequent call with
                                      the same ptrn follows
                                      the same pattern
```

```
    ...
    ... some calculations involving the internal data only can
    ... be performed here while waiting for the boundary data
    ...
    call MSG_tbdx_receive(...,ptrn,...)  -- the boundary
                                            information
                                            is received

    ...
  enddo
  ...
  call MSG_tbdx_close(...,ptrn,...)      -- closes the channels
                                            associated with ptrn
```

The MSG library exploits the concept of communication channels. A communication pattern is a unique combination of channels to adjacent processors. A pattern is indicated by its number (integer variable `ptrn`). Only a limited number of communication patterns can be simultaneously opened. A communication pattern can be closed by the function `MSG_tbdx_close`. A channel is open when the first call to the send function `MSG_tbdx_send` with the corresponding pattern number `ptrn` is issued. The MSG send/receive functions `MSG_tbdx_send` and `MSG_tbdx_receive` are coded in the current version using the MPI repeated communication request functions `MPI_Send_init`, `MPI_Recv_init`, and `MPI_Start` [1]. The boundary data may take off at the moment the call to `MSG_tbdx_send` was issued. However a user should not touch the local and external interface segments until the function `MSG_tbdx_receive` is executed. In the meantime, some operations involving the internal data can be performed since the internal nodes do not need information from the other processors. The call to `MSG_tbdx_receive` ensures that the data have arrived and have been put into place. Calls to `MSG_tbdx_send` and `MSG_tbdx_receive` always go in pairs. The both calls must be made even when the processor only sends or only receives data.

4 Data transfer routines.

4.1 MSG_tbdx_send - send boundary information.

The function MSG_tbdx_send sends a boundary information to neighboring processors. It uses the index tables created at the setup stage and therefore must be preceded by a MSG setup routine. It is an asynchronous nonblocking routine and does not ensure that the information has actually arrived to the destination upon its completion. Neither the buffer nor the local interface segment of the local array should not be modified until the function MSG_tbdx_receive is executed. A call to MSG_tbdx_send must be issued even if the processor is only receiving data and does not send any data out since it is the function MSG_tbdx_send who opens both send and receive channels. The first call to MSG_tbdx_send with a specific communication pattern defined by the argument ptrn opens communication channels and then sends the boundary data through the channels. Once a pattern (channel) is established, the processors can communicate without extra handshakes. Every subsequent exchange with the same ptrn (through the same channel) will be performed faster. A processor may only send data or only receive information or perform a data exchange depending on the topology of grid decomposition. The MSG toolkit can be compiled to transfer single precision (-DSINGLE compiler switch gives FLOAT=REAL) or double precision (-DDOUBLE compiler switch gives FLOAT=REAL*8) floating point data.

Syntax

```
      subroutine MSG_tbdx_send(x, Buffer, NumAdjProc, Proc,
     &                         Ipr, Index, ptrn, ierror)
      integer NumAdjProc, ptrn, ierror
      integer Proc(NumAdjProc), Ipr(*), Index(*)
      FLOAT x(*), Buffer(*)
```

Arguments

- x - local array (input) of type FLOAT (can be REAL or REAL*8).
- Buffer - communication buffer (workspace). This array of type FLOAT must not be smaller than two times the largest of the faces of the subdomain in order to

accommodate the incoming external interface segment and the outgoing local interface segment.

- `NumAdjProc` - number of the adjacent processors (input) (integer). This argument must be preset by a setup routine.
- `Proc` - list of the adjacent processors (input) (integer array). This argument must be preset by a setup routine.
- `Ipr` - array of pointers to segments in the table of indices (input) (integer array). This argument must be preset by a setup routine. The size of this array must not be smaller than twice the number of the adjacent processors plus one.
- `Index` - array of indices needed to place the boundary data within the local array (input) (integer array). This argument must be preset by a setup routine. The size of this array must not be smaller than the overall size of the local interface data segment plus the size of the external interface data segment.
- `ptrn` - number of the communication pattern to use (input) (integer). It must be between 1 and the maximal allowed number of patterns `MAX_PATTERNS`. The maximal number of patterns is specified in the MSG compile time. If the allowed number of patterns is too small then the parameter `MAX_PATTERNS` in the file `geom_param_fort.h` should be increased and the MSG must be recompiled.
- `ierror` - return code (output) (integer):
  - 0 - successful completion;
  - -1 - an error occurred during an attempt to open a communication channel;
  - -2 - the specified number of communication pattern `ptrn` is zero or larger than the allowed maximum `MAX_PATTERNS`;
  - -3 - the number of adjacent processors is larger than the allowed maximum `MAX_PROCS` (this parameter is specified in the file `geom_param_fort.h`, and the MSG must be recompiled when there is a need to increase this parameter);

- `ierror > 0` corresponds to one of the errors within the MPI functions (see MPI error codes).

The entries in the index tables `NumAdjProc`, `Proc`, `Ipr`, `Index` should be set by a setup routine. Each communication pattern has its own tables.

4.2 `MSG_tbdx_receive` - receive boundary information.

The boundary exchange routines employ asynchronous nonblocking message-passing MPI functions. The routine `MSG_tbdx_send` sends the boundary data out to the network and does not wait for the confirmation of the arrival. Neither the buffer (specified by the `Buffer` argument) nor the local and external interface segments of the local array (specified by the `x` argument) should be modified before the corresponding call to `MSG_tbdx_receive` is issued. Operations involving the internal data only can be performed between the calls to `MSG_tbdx_send` and `MSG_tbdx_receive`. The routine `MSG_tbdx_receive` ensures that the data has actually arrived and has been placed into the appropriate location within the local array `x`. A call to `MSG_tbdx_receive` must be made to complete the data transfer even when the processor is only sending his data out without receiving anything.

Syntax

```
      subroutine MSG_tbdx_receive(x, Buffer, NumAdjProc, Proc,
     &                            Ipr, Index, ptrn, ierror)
      integer NumAdjProc, ptrn, ierror
      integer Proc(NumAdjProc), Ipr(*), Index(*)
      FLOAT x(*), Buffer(*)
```

Arguments

The routine `MSG_tbdx_receive` has the same arguments as the routine `MSG_tbdx_send` (see Section 4.1). It may produce the following return codes (in `ierror`):

- 0 - successful completion;
- `ierror > 0` corresponds to one of the errors within the MPI functions (see MPI error codes).

4.3 MSG_tbdx_close - close a communication pattern.

The routine MSG_tbdx_close closes the communication pattern indicated by the argument ptrn. All the arguments are the same as those of the routines MSG_tbdx_send and MSG_tbdx_receive. Communication channels are limited resources, and it is advisable to deallocate a communication pattern if it is not needed anymore. The maximal number of patterns which can be simultaneously activated is given by the parameter MAX_PATTERNS at the MSG compile time in the file geom_param_fort.h. A pattern with the same number ptrn can be later reopened. A communication pattern should always be closed first by calling MSG_tbdx_close and then reopened if either a geometry of local array or a processor topology has changed. A previously closed channel reopens while making the first request (MSG_tbdx_send) with the same ptrn.

Syntax

```
      subroutine MSG_tbdx_close(x, Buffer, NumAdjProc, Proc,
     &                          Ipr, Index, ptrn, ierror)
      integer NumAdjProc, ptrn, ierror
      integer Proc(NumAdjProc), Ipr(*), Index(*)
      FLOAT x(*), Buffer(*)
```

Arguments

The routine MSG_tbdx_close has the same arguments as the routine MSG_tbdx_send (see Section 4.1). It may produce the following return codes (in ierror):

- 0 - successful completion;
- ierror > 0 corresponds to one of the errors within the MPI functions (see MPI error codes).

5 Setup routines.

The index tables must be set by the MSG setup routines before the actual communications calls are issued. These tables will be later used by the MSG_tbdx_send and MSG_tbdx_receive routines. The setup routines do not send any messages. They are entirely local. The routine MSG_tsetup sets up the tables for the simple case when the local array consists only of the local data plus the overlapping region (halo) at the sides of the local data segment. For the simple setup, the widths of the halo regions on all the processors must all be the same. A more general case can be handled by the detailed setup routine `MSG_tp_setup`.

The setup routines treat the global grid as always being three dimensional (except in the case when the work is being done on a slice of an array as described in the Section 7.1). If the grid has less than three dimensions then the remaining dimensions should be set to 1.

5.1 `MSG_tsetup` - simple table setup.

The simple table setup function `MSG_tsetup` attempts to divide the global grid equally between the processors which form a grid of processors. If a dimension of the global computational grid is not a multiple of the processor grid's dimension then the remaining nodes of the computational grid will be assigned to the last processor along this axis. The three dimensions of the global grid have to be specified in the array `GlobalGridSize`. A grid of processors must be described by the argument `ProcGridSize` (integer array of size 3). It is assumed that the processors are numbered according to the FORTRAN 77 order (fist index changes first). All the three dimensions of the computational grid along with the three dimensions of the processor grid must be given as arguments to `MSG_tsetup`. If the dimensionality of the problem is less than 3 then the remaining entries in `GlobalGridSize` and `ProcGridSize` should be set to 1.

The routine `MSG_tsetup` assumes that the dimensions of the local array are equal to the sizes of local data segment plus the equal size halos on both sides. The global grid dimensions in Figure 3 are `GlobalGridSize=10,10,1`, the processors grid dimensions are `ProcGridSize=2,2,1`, and the halo widths are `HaloWidth=1,1,0`. The halos (dark shaded areas in Figure 3) will be filled by the neighbors. The grid shown here is periodic in both directions and therefore `IfPeriodic=1,1,0`.

Syntax

```
      subroutine MSG_tsetup(NumProc, MyProc, ptrn, GlobalGridSize,
     &                      ProcGridSize, HaloWidth, IfPeriodic,
     &                      NumAdjProc, Proc, Ipr, Index,
     &                         LocalAxisSize, LocalAxisPos, ierror)
      integer NumProc, MyProc, NumAdjProc, ptrn, ierror
      integer LocalAxisSize, LocalAxisPos
      integer ProcGridSize(3), HaloWidth(3), IfPeriodic(3), GlobalGridSize(3)
      integer Proc(NumAdjProc), Ipr(*), Index(*)
```

Arguments

- `NumProc` - total number of subdomains (processors) (input) (integer).
- `MyProc` - my processor's number (from 1 to `NumProc`) (input) (integer).
- `ptrn` - this integer argument has been left for compatibility with the previous version of MSG. It is neither used nor modified.
- `GlobalGridSize` - dimensions of the global grid (input) (integer array of size 3).
- `ProcGridSize` - dimensions of the grid of processors (input) (integer array of size 3).
- `HaloWidth` - widths of the halos (input) (integer array of size 3).
- `IfPeriodic` - indicates the type of boundary conditions (input) (integer array of size 3). If `IfPeriodic(axis)=1` then the last processor along the axis `axis` will exchange data with the first processor (periodic or "wrap around" boundary condition). `IfPeriodic(axis)=0` indicates that the axis `axis` is not periodic.
- `NumAdjProc` - number of the adjacent processors (output) (integer).
- `Proc` - list of the adjacent processors (output) (integer array of the size `NumAdjProc` at least).
- `Ipr` - table of pointers to the segments within the index table (output) (integer array of size `2*NumAdjProc+1` at least).
- `Index` - table of indices (output) (integer array of the size of the local interface data segment plus the external interface data segment).

- `LocalAxisSize` - integer (input), must be 1 unless data is moved within a slice (see Section 7.1).
- `LocalAxisPos` - integer (input), must be 1 unless data is moved within a slice (see Section 7.1).
- `ierror` - return code (output) (integer):
  - 0 - successful completion;
  - -5 - wrong number of processors specified (`ProcGridSize` does not conform to `NumProc`).

The routine `MSG_tsetup` finds the adjacent processors. Their number is returned in `NumAdjProc`. It also sets the tables `Proc`, `Ipr`, `Index` to be used by the MSG send/receive routines.

5.2 `MSG_tp_setup` - detailed table setup.

The detailed table setup function `MSG_tp_setup` is more flexible than the simple table setup function `MSG_tsetup` described in the Section 5.1. It allows a detailed control over array layouts. Local arrays must include but are not limited to the local data segments and the external interface data segments. The sizes of local data segments can vary from a processor to a processor, but the decomposition of the global grid must preserve its tensor product nature.

It should be noted that the external interface segments together with the local data must form a rectangular prism. In this section, this rectangular prism is referred as an *active data segment*. Thus the shaded areas in Figure 4 must have a rectangular shape as they do. A user can place the active data segment anywhere within the local array. Its position within the local array must specified by the three offsets given in `ActDataStart`. The complete information on all the other processors data structures must be known to a processor in order to establish its connectivity and to set its index tables needed for data transfer. The numbering of the processors does not have to follow the FORTRAN 77 order anymore. The processors must form a 3D grid but can be numbered without any particular order.

Syntax

```
 subroutine MSG_tp_setup(LocalArraySize, ActDataStart,
&                        GlobalCoordLocalData, GlobalCoordActData,
&                        NumProc, MyProc, NumAdjProc, Proc, Ipr,
&                        Index, LocalAxisSize, LocalAxisPos, ierror)
 integer NumProc, MyProc, NumAdjProc, ierror
 integer LocalArraySize(3, NumProc), ActDataStart(3, NumProc)
 integer GlobalCoordLocalData(2, 3, NumProc),
&        GlobalCoordActData(2, 3, NumProc)
 integer Proc(NumAdjProc), Ipr(*), Index(*)
```

Arguments

- `LocalArraySize` - dimensions of the local array (input) (integer array of size 3).
- `ActDataStart` - three indices of the first element of the active data segment within the local array (input) (integer array of size 3). This argument indicates the position of the active data segment within the local array. An index may vary from 1 to `LocalArraySize(axis)`.
- `GlobalCoordLocalData` - global grid coordinates of the local data segments for all the processors (input) (integer array of dimensions `2, 3, NumProc`). The value of `GlobalCoordLocalData(1, axis, i)` gives the global index of the first element of the local data segment (rectangular prism) for the axis `axis` and the processor `i`. The value of `GlobalCoordLocalData(2, axis, i)` gives the global index of the last element of local data. If the dimensionality is less than 3 then all the entries of `GlobalCoordLocalData` for the remaining axes must be set to 1.
- `GlobalCoordActData` - global grid coordinates of the active (local + external interface) data segments for all the processors (input) (integer array of dimensions `2, 3, NumProc`). This array has the same format as `GlobalCoordLocalData`.
- `NumProc` - total number of subdomains (processors) (input) (integer).
- `MyProc` - my processor's number (from 1 to `NumProc`) (input) (integer).

- `NumAdjProc` - number of the adjacent processors (input/output) (integer). On the input, `NumAdjProc` signifies the number of neighbors already included in the list for a given communication pattern. Upon the return, it gives the updated number of adjacent processors. A nonzero input value of `NumAdjProc` would indicate that the tables are being appended (see Section 7.3).

- `Proc` - list of the adjacent processors (input/output) (integer array of the size `NumAdjProc` at least). If the input value of `NumAdjProc` is not zero then the list `Proc` will be appended.

- `Ipr` - table of pointers to the segments within the index table (input/output) (integer array of size `2*NumAdjProc+1` at least). If the input value of `NumAdjProc` is not zero then this table will be appended.

- `Index` - table of indices (input/output) (integer array of the size of the local interface data segment plus the external interface data segment). If the input value of `NumAdjProc` is not zero then this table will be appended.

- `LocalAxisSize` - integer (input), must be 1 unless data is moved within a slice (see Section 7.1).

- `LocalAxisPos` - integer (input), must be 1 unless data is moved within a slice (see Section 7.1).

- `ierror` - return code (output) (integer):

  - 0 - successful completion;

  - -1 - number of adjacent processors is larger than the allowed maximun `MAX_PROCS` (this parameter is specified in the file `geom_param_fort.h`, and the MSG must be recompiled when there is a need to increase this parameter);

  - -6 - the active data segment is not within the local array (`ActDataStart` is less than 1 or larger than `LocalArraySize`).

The input to boundary exchange routines is illustrated in Figure 5 for a single dimension. Multidimensional data structures are rectangular prisms or tensor product of 1D intervals. If the beginning of the local array coincides with the first element of the active segment (`GlobalCoordActData(1,axis,i)`) then `ActDataStart(axis,i)=1`. If the left external interface data segment is empty (no halo) then the active data segment starts with the local data, and `GlobalCoordActData(1,axis,i)` is equal to `GlobalCoordLocalData(1,axis,i)`. The same situation is possible on the right end of the local array. If the subdomain has a periodic left boundary of the width `h` and the index of global grid ranges from 1 to `N` for this axis then `GlobalCoordActData(1,axis,i)` equals to `N-h+1`. If the subdomain has a `h`-wide periodic right boundary and the index of global grid starts with 1 then `GlobalCoordActData(2,axis,i)=h`. It is assumed that a processor can have only one right and one left neighbors. This condition is equivalent to the fact that a halo should be smaller that the corresponding neighbor's local data segment.

6 Syntax of the auxiliary functions.

6.1 `MSG_enable` - initialize the MSG.

The function `MSG_enable` initializes the MSG. It must precede any other MSG calls except the routine `MSG_set_comm_parent` which can be used to change the default parent communicator (see Section 6.3). It can be called instead of or after the MPI initialization function (`MPI_Init`). If the MPI has not been initialized before the call to `MSG_enable` the latter initializes the MPI.

Syntax

```
subroutine MSG_enable(MyProc, NumProc)
integer MyProc, NumProc
```

Arguments

- `MyProc` - my processor number (can be from 1 to `NumProc`) (output) (integer).
- `NumProc` - the number of processors in the system (output) (integer).

The function `MSG_enable` returns in `NumProc` the size of the group of processors associated with the parent communicator. The default parent communicator is

MPI_COMM_WORLD [1] which involves all the processors. The MSG routines use their own communicator created by MSG_enable.

6.2 MSG_disable - quit the MSG.

The function MSG_disable disconnects the MSG. It frees the communicator associated with the MSG routines. MSG_disable does not replace the MPI clean-up function MPI_Finalize which must be called after MSG_disable. The failure to call MPI_Finalize may result in a hangup or create a zombie process.

Syntax

```
subroutine MSG_disable(ierror)
integer ierror
```

Arguments

- ierror - return code produced by the MPI function MPI_COMM_FREE (output) (integer).

6.3 MSG_set_comm_parent - set the parent communicator.

If the MSG is called within the scope of a communicator different from default communicator MPI_COMM_WORLD then a call to the function MSG_set_comm_parent should be issued before the MSG initialization (MSG_enable).

Syntax

```
subroutine MSG_set_comm_parent(comm)
integer comm
```

Arguments

- comm - parent communicator (input) (integer).

6.4 MSG_myproc - get my processor's number.

The function MSG_myproc returns the number of my processor. According to the MSG, the processors are numbered starting with 1.

Syntax

```
integer function MSG_myproc()
```

6.5 MSG_nproc - get the number of processors in the system.

The function MSG_nproc returns the number of processors allocated for the MSG or the number of processors in the group associated with the MSG communicator.

Syntax

```
integer function MSG_nproc()
```

7 Advanced features of MSG.

7.1 Moving information within a slice of a local array.

The MSG data transfer and setup functions are able to exchange information within a slice of a distributed grid. This option was added to handle data structures intended to enhance the performance of local arithmetic of the cache-based microprocessors not featuring an interleaved memory. In this circumstances, it may be beneficial to stack several variables participating in local arithmetic operations on the top of each other to form a grid with the first axis being local to a processor. This data structure would enhance data locality thus reducing the number of cache misses. The FORTRAN 77 order is assumed again, and this local axis must be the first axis of the local array.

A data structure where the first axis is local is shown in Figure 6. The data transfer is taking place within a plane perpendicular to the local axis. The dimension of the local axis (`LocalAxisSize`) and the position of the slice within this axis (`LocalAxisPos`) should be given as arguments to the MSG setup functions.

7.2 Using the same index tables for multiple arrays.

The MSG functions always try to avoid unnecessary memory moves. By default, they try to place data directly in the appropriate positions within the local array whenever it is possible. If a boundary data segment is contiguous in processor's memory then it can be given to the MPI send/receive routines in place with no buffering. The data transfer function keep track of the address of the beginning of the segment. In this situation, any other array would require its own communication pattern. Although it is possible sometimes to use the same communication pattern and for several distributed arrays having the same shape. In order to transfer the data correctly in this situation, the MSG must be compiled with the `-DBUFFER_ALWAYS` switch. This compile time switch indicates that incoming and outgoing data will always be buffered, and it will be safe to employ the same pattern indicated by the argument `ptrn` to `MSG_tbdx_send` and `MSG_tbdx_receive` for multiple arrays. It means that different arrays can be given as arguments (argument `x`) to `MSG_tbdx_send` with the same communication pattern `ptrn` (see Section 4.1). The `-DBUFFER_ALWAYS` option would save some time at the moment when communication channels are being opened. It would, however, slow down the send/receive routines due to extra transfers in the local memory.

The same set of index tables `Proc`, `Ipr`, `Index` (see Section 4.1) can always be employed for arrays of the same shape. A distributed array is considered to be of the same shape as another array if their local data structures are identical. That means the arrays must have the same dimensions and must be distributed in the same way. It is not necessary to call a MSG setup routine several times to set up the index tables `Proc`, `Ipr`, `Index` for distributed arrays having the same layout (shape). The address of an array becomes known to the MSG only when a communication pattern is opened by issuing a call to `MSG_tbdx_send`. The address is not tied to the index tables `Proc`, `Ipr`, `Index`.

7.3 Appending index tables for already existing communication patterns.

An existing communication pattern can be appended. An information can be added to the tails of the index tables `Ipr` and `Index`, and to the end of the list of adjacent processors `Proc` (see Section 5). This option is needed to describe a situation when the halos do not form a rectangular prism, as for example in the case of a cross-shaped stencil. If a geometry of the external interface area can be defined as a superposition of several rectangular prisms then the communication pattern can be formed by calling the detailed setup function `MSG_tp_setup` (see Section 5.2) several times giving each time coordinates of the corners of one of these prisms. A nonzero input value of the argument `NumAdjProc` to `MSG_tp_setup` would indicate that the communication pattern is being appended.

7.4 MSG compile time switches.

The MSG routines can be compiled with the following compile time switches.

- `-DBUFFER_ALWAYS` indicates that the data are always buffered and it is safe to apply the same communication pattern to multiple arrays.
- `-DDEBUG` disables the MPI calls within the MSG routines. The MSG functions create the index tables but the data transfer is not performed. This option can be used to debug.
- `-DSINGLE` (default option) indicates that the arrays are of single precision `REAL`.
- `-DDOUBLE` indicates that the arrays are of double precision.
- `-D_SUN` switch (default option) is used to link FORTRAN 77 and C modules when names of FORTRAN 77 routines are converted to lowercase letters with an underscore symbol at the end (Sun and Silicon Graphics machines).
- `-D_CRAY` switch is used to link FORTRAN 77 and C modules when names of FORTRAN 77 routines are converted to uppercase letters (CRAY T3D and T3E).

- `-D_CSPP` switch is used to link FORTRAN 77 and C modules when names of FORTRAN 77 routines are converted to lowercase letters (Convex Exemplar).

References

[1] Gropp, W., Lusk, E., and Skjellum, A. 1994 *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA.

[2] Thomas, S. J., A. V. Malevsky, M. Desgagne, R. Benoit, P. Pellerin, and M. Valin 1997 *Massively Parallel Implementation of the Mesoscale Compressible Community Model*, submitted to Parallel Computing.

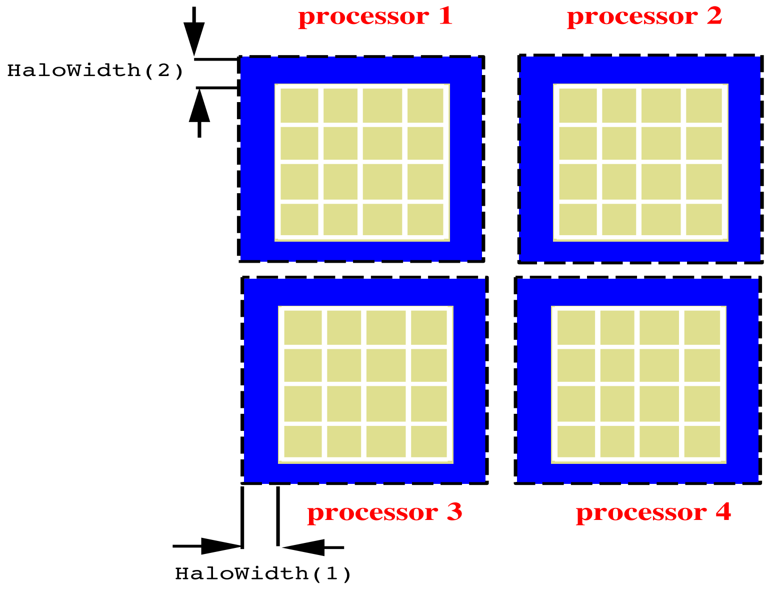

Figure 3: A 2D Cartesian grid is divided between four processors. Each processors holds the same data structure, and the embedding local array consists of the local data segment and the external interface data segment (halo) only. The halo has the darker shade. The local data segment has the lighter shade. The darker area will be filled with the data from the neighboring processors. The width of the halo region for each of the three spatial dimensions `axis=1,2,3` is specified by the value of `HaloWidth(axis)`. This decomposition is suitable for the simple table setup routine `MSG_tsetup`.

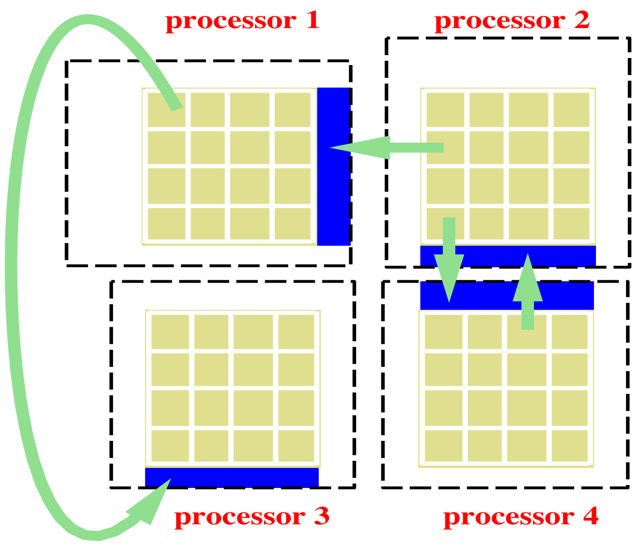

Figure 4: A 2D Cartesian grid is divided between four processors. The local arrays (contoured by dashed lines) have different dimensions. They confine the local data segments and the external interface data segments (halos). The halo has the darker shade. The local data segment has the lighter shade. The both shaded areas form the active data segment. The arrows indicate how the halos will be filled. This structure requires a detailed setup.

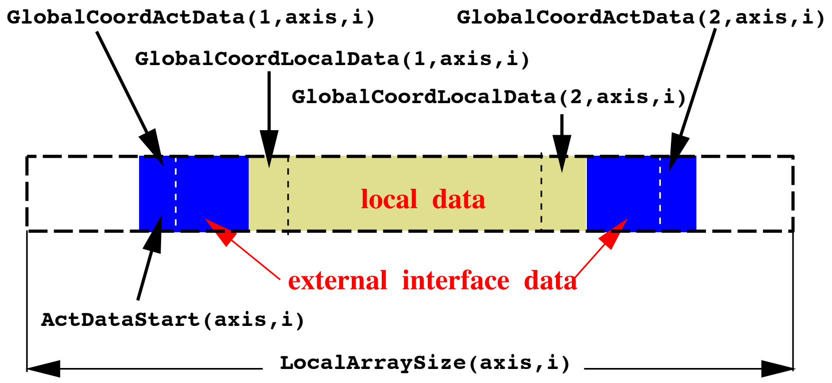
GlobalCoordActData(1,axis,i)
GlobalCoordActData(2,axis,i)
GlobalCoordLocalData(1,axis,i)
GlobalCoordLocalData(2,axis,i)
local data
external interface data
ActDataStart(axis,i)
LocalArraySize(axis,i)

Figure 5: Arguments for the detailed setup routine `MSG_tp_setup` describe local data structures for all the processors. A 1D data structure (along the axis `axis`) for the processor `i` is depicted here. The global indices of the first and the last elements of the local data segment are given by `GlobalCoordLocalData(1,axis,i)` and `GlobalCoordLocalData(2,axis,i)`. The global indices of the first and the last elements of the active (local + external interface) data segment are given by `GlobalCoordActData(1,axis,i)` and `GlobalCoordActData(2,axis,i)`. `ActDataStart(axis,i)` indicates the position of the active data segment within the local array of length `LocalArraySize(axis,i)`.

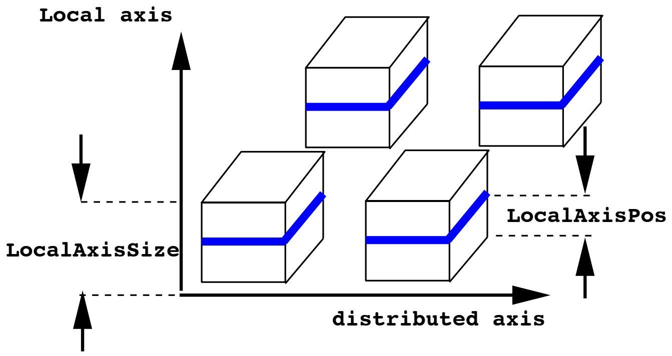

Figure 6: The MSG functions can transfer information within a slice of a distributed grid. Then the first axis must be local to a processor. The other three axes can be distributed. Here the first axis of a 3D grid is local and the remaining two axes are distributed between the processors. The information is moved within the plane perpendicular to the local axis. The dimension of the local axis is given by `LocalAxisSize` and the position of the plane is given by `LocalAxisPos`.