# The Mesh Archetype

Berna L. Massingill

**Abstract**

A parallel program archetype aids in the development of reliable, efficient parallel applications with common computation/communication structures by providing development methods and code libraries specific to the structure. This document presents an archetype for *mesh computations.* It describes the common structure captured by the archetype abstraction, discusses a parallelization strategy for such computations, documents our code library to support this parallelization strategy, and presents a collection of example application programs.

## 1 Introduction: mesh computations

### 1.1 Overview

In a mesh computation:

- Data is based on an $N$-dimensional grid ($N = 1$, 2, or 3), with one or more variables per cell (grid point).
- Computation consists of some sequence of the following operations:
  - computing, for each cell, new values for one or more variables, based on old values of variables in that cell and nearby cells (neighbors, next-to-neighbors, etc.)
  - (optionally) reading in values for variables.
  - (optionally) writing out values of variables.
  - (optionally) computing global reductions over the whole grid — e.g., global maximum or sum of one of the variables.

  (Frequently the compute-new-values and reduction operations are performed repeatedly in a time-step loop.)

Figure 1 illustrates the basic operation — computing new values in terms of old values — in a 2-dimensional grid.

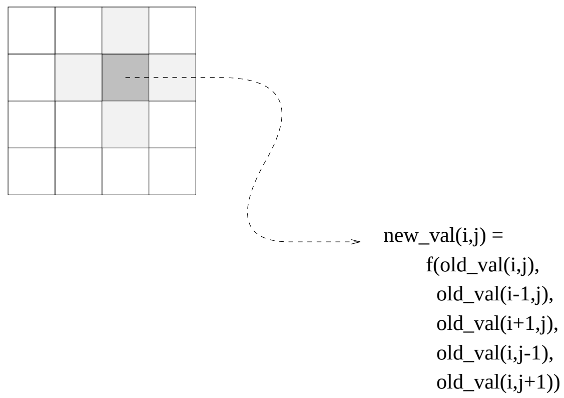


Figure 1: Basic mesh computation.

## 1.2 Parallelization strategy (distributed memory)

Mesh computations are readily parallelized for distributed-memory architectures using the following approach:

- The grid ($N$-dimensional) is distributed over an $N$-dimensional grid of processes.
- A separate (optional) host process is used for reads/writes involving a whole array.
- Non-grid variables — global constants and reduction variables, e.g. — are duplicated in each process.

### 1.2.1 Distributing the data

Figure 2 illustrates how data for a 2-dimensional mesh computation is distributed over a 2-dimensional grid of processes: The original (undistributed) 16-by-16 array is partitioned into contiguous subarrays (called *local sections*) and distributed among a 4-by-2 grid of processes.

Observe that now each element in the array has two sets of indices — its *global indices* (reflecting its position in the original undistributed array) and its *local indices* (reflecting its position in the local section). (For example, using the Fortran convention of starting array indices at 1, the shaded square has global indices (3,6) and local indices (1,2) in the grid process (2,1).)

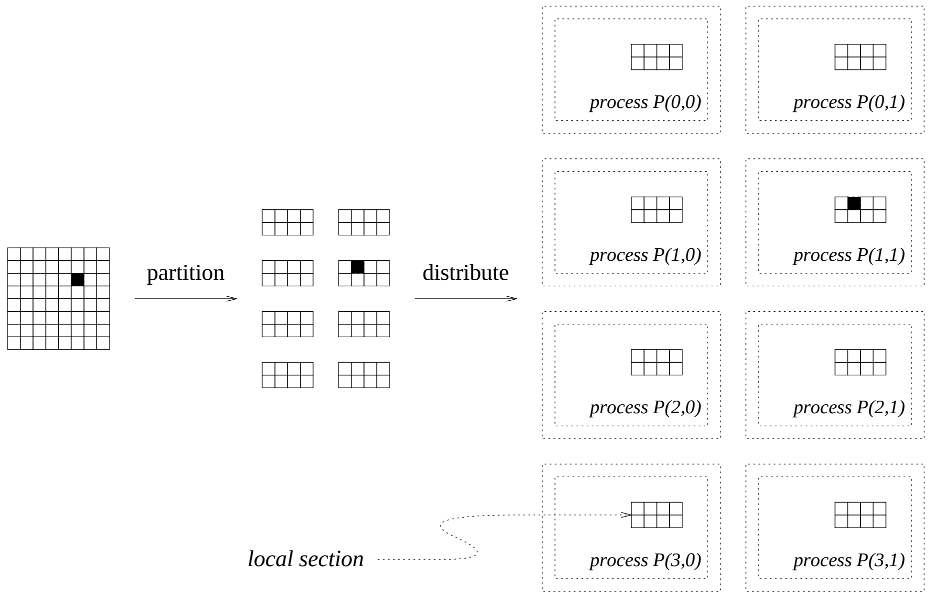


Figure 2: Distributing data.

### 1.2.2 Computing new values for grid points

Computation of new values for grid variables can now be done concurrently, with each grid process computing new values for the grid points in its local section. For a grid point in the interior of a local section, it is easy to see that the data required to compute its new values (old values at the point and neighboring points) is readily available locally, but for grid points on the boundary of a local section, the required data may reside in another process. This is handled by surrounding each local section with a *ghost boundary*, which is used as a buffer to contain a shadow copy of data from neighboring local sections. Figure 3 illustrates a local section plus ghost boundary. Note that the ghost boundary can have width greater than 1; if, for example, the computation of new values for a grid point requires values from points 2 grid cells away, then the ghost boundary should have width 2.

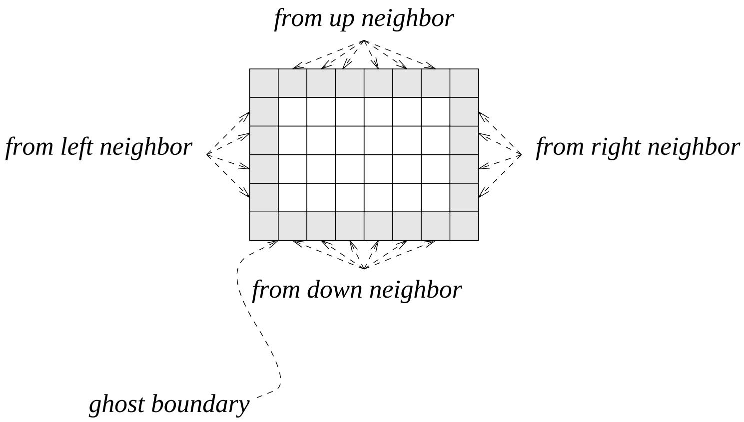


Figure 3: Local section with ghost boundary.

Values for array elements in the ghost boundaries are filled in via a *boundary exchange* operation, in which grid processes send data from the boundaries of their local sections to the ghost boundaries of their neighbors. Once this is done, values for all points in a local section (possibly excluding those on the boundary of the global array) can be computed exactly as one would compute them in a sequential program, with the ghost boundaries supplying neighboring values for points on the boundaries of the local sections.

As an example, consider performing the computation shown in figure 1 to the points in the local section shown in figure 3. (Both arrays mentioned by the formula – `newval` and `oldval` – have a local section of the form shown.) The code would look something like this:

```
parameter (N=6)
parameter (M=4)
real newval(0:N+1, 0:M+1)
real oldval(0:N+1, 0:M+1)
.... boundary exchange ....
do i=1,N
do j=1,M
        newval(i,j) = f(oldval(i-1,j), oldval(i+1,j),
                        oldval(i,j-1), oldval(i,j+1))
enddo
enddo
```

Observe that no special handling is required for points on the boundaries of the local section (`i` = 1 or `N`, `j` = 1 or `M`); values for `oldval` at neighboring points come from the ghost boundaries, which have been filled in by a boundary exchange operation.

#### 1.2.3 Performing I/O

If the computation requires reading a whole grid-based array from a sequential file, this can be done by using a separate host process to read the data into an undistributed copy of the array and then distributing the data from the host process to the grid processes. Writing a whole grid-based array is analogous. (Alternatively, each grid process can read/write its own local section from/to its own file, which is often more efficient but not as straightforward.)

Reading or writing non-grid-based variables (global constants, reduction variables, etc.) can also be accomplished using the host process: Global constants are read into the host process and then broadcast to the grid process, while reduction variables are computed by all processes acting together and then written by the host process.

#### 1.2.4 Structuring the parallel program

These operations lead to the interprocess communication structure shown in figure 4: Grid processes communicate with their neighbors and with the host process.

### 1.3 Sequential vs. parallel

Given the parallelization strategy described in the previous section, a parallel program to accomplish a particular mesh computation closely resembles its sequential counterpart, except that the work has been partitioned between a host process and a number of essentially identical grid processes:

**Computing new values for grid-based variables.**

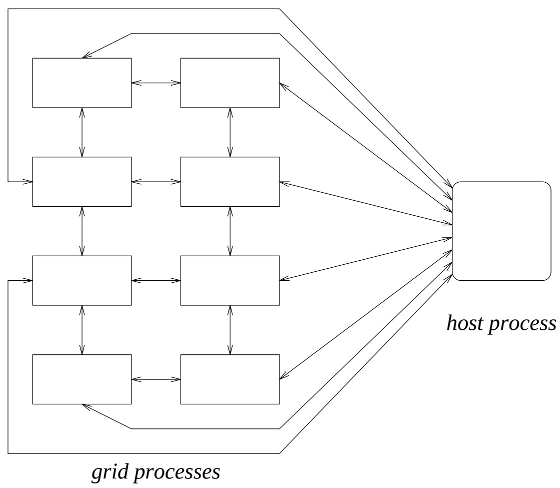


Figure 4: Interprocess communication.

**Sequential program** loops over the whole grid. Points on the boundary may be treated differently from interior points.

**Host process** does nothing.

**Grid processes** first ensure that ghost boundaries to be used as input contain current values (via a boundary-exchange operation), then each loop over a local section. Because of the presence of the ghost boundaries, no special handling is required for points on "internal" boundaries (points that are on the boundary of a local section but that do not correspond to points on the boundary of the whole array). If points on the boundary of the whole array require different treatment, this is handled by grid processes that contain part of the boundary.

**Reading values into a grid-based variable.**

**Sequential program** reads into a whole array, e.g. from a file.

The parallel program may take several approaches. The most straightforward makes use of the host process:

**Host process** reads into its array and then participates in redistribution operation that distributes array values over the process grid.

**Grid processes** participate in redistribution operation.

An alternative approach reads data directly into the grid processes:

**Host process** does nothing.

**Grid processes** each read from a separate sequential file. Each file contains data for one local section.

**Writing values from a grid-based variable.**

**Sequential program** writes a whole array, e.g. to a file.

The parallel program may take several approaches. The most straightforward makes use of the host process:

**Host process** participates in redistribution operation that collects array values from the process grid and then writes from its array.

**Grid processes** participate in redistribution operation.

An alternative approach writes data directly from the grid processes:

**Host process** does nothing.

**Grid processes** each write to a separate sequential file. Each file contains data for one local section.

**Reading values into a duplicated (non-grid) variable.**

**Sequential program** reads data (global constants, e.g.) from a file.

**Host process** reads data in the same way the sequential program would and then participates in a broadcast operation to copy the data to the grid processes.

**Grid processes** participate in a broadcast operation to obtain data from the host process.

**Writing values from a duplicated (non-grid) variable.**

**Sequential program** writes data (results of a reduction operation, e.g.) to a file.

**Host process** writes data exactly as the sequential program does. (Usually, the variable whose value is to be written has the same value in all processes — either because it is a global constant or because it is the result of a reduction operation, as described below.)

**Grid processes** do nothing.

**Performing a reduction operation.**

**Sequential program** performs the reduction, often by looping over the whole array.

**Host process** participates in the reduction operation — without, however, supplying data — and receives the result.

**Grid processes** participate in the reduction operation, supplying data and receiving the result. (E.g., to compute a global maximum, each grid process computes a local maximum, and then all processes (host and grid) participate in a reduction operation, after which all processes have the resulting global maximum.)

If the computation does not perform whole-grid reads or writes using the host process, then it is possible to parallelize it without a host process; in that case, the actions performed by the host process in the above descriptions are instead performed by one of the grid processes, which is singled out as the "designated I/O process".

# 2 Archetype implementation

Several implementations (program skeletons and code libraries) exist for the mesh archetype, including:

- Fortran M (1D, 2D, 3D, and 3D without a host process).
- Fortran plus NX (2D and 2D without a host process).

In general, the user-level specifications of all implementations are the same. Where they differ, we describe here the Fortran M 3D implementation. (Fortran M [2, 1] is a Fortran variant available via anonymous FTP from Argonne National Labs.)

## 2.1 Parameter definitions

The application programmer supplies `PARAMETER` definitions for the following:

- The dimensions of the ($N$-dimensional) grid.
- The width of the ghost boundary. (This width cannot be larger than the size of a local section.)
- The dimensions of the process grid (i.e., how many grid processes are to be used).

This is done by modifying the archetype-supplied `INCLUDE` file `mesh_uparms.h`; see comments in the file for additional information.

The archetype uses these definitions to declare additional `PARAMETER`s:

- The dimensions of a local section.
- Index ranges for a local section, including ghost boundaries.

These `PARAMETER`s should be used to work with grid-based data — for declarations, loop bounds, etc. See comments in the `INCLUDE` file `mesh_parms.h` for more information.

**Example**

For example, to distribute a grid of dimensions 1000 by 1000 by 1000 over a process grid of dimensions 2 by 2 by 2 (8 grid processes in all), for a computation in which new values for grid variables require values at nearest-neighbor points only, the application programmer would define:

- Grid dimensions `NX=1000`, `NY=1000`, and `NZ=1000`.
- Ghost boundary `NGHOST=1`.
- Process grid dimensions `XPROCS=2`, `YPROCS=2`, and `ZPROCS=2`.

## 2.2 Data declarations

The archetype defines a `PROCESS COMMON` block (data common *to the process* — each process's block is distinct) for data to be used by its subroutines. These variables are available to the application programmer; useful variables include:

- Process type — host or grid — and process indices (for grid processes). (The process type variable is replaced in the no-host-process implementations with a logical variable indicating whether this process is the "designated I/O process".)
- Actual size of local data — may be different from the local-dimension `PARAMETER` if the number of processes does not evenly divide the size of the grid.

See comments in `INCLUDE` file `mesh_common.h` for more information.

The application programmer defines all application data, including grid-based data. Dimensions of grid-based arrays should be declared using the archetype-supplied size parameters mentioned above.

**Example**

Continuing the example of the preceding section, grid data can be declared thus:

- In the host process, the whole array is declared:

  ```
  REAL MYDATA(NX, NY, NZ)
  ```

  Observe that *this declaration is only needed if array* `MYDATA` *is to be read/written from the host process.*

- In a grid process, a local section (including ghost boundaries) is declared:

  ```
  REAL MYDATA(IXLO:IXHI, IYLO:IYHI, IZLO:IZHI)
  ```

(PARAMETERs IXLO, IXHI, etc. are computed by the archetype implementation from the user-specified grid dimensions, ghost boundary size, and process grid dimensions.)

Indices for the local section exclusive of ghost boundaries are (1:NXlsize, 1:NYlsize, 1:NZlsize).

### 2.3 Program skeleton

The archetype provides a program skeleton, including a main program to set up the grid of processes; the user supplies the body of two programs:

- The program to execute in the host process (not required for the no-host-process implementations).
- The program to execute in the grid processes.

and can provide additional subroutines as needed for the application. The user can also define COMMON blocks; data in COMMON blocks is typically declared using archetype-supplied dimensions, as described above. For Fortran M implementations of the archetype, COMMON blocks should be declared as PROCESS COMMON blocks to indicate that each process is to have a unique copy of the COMMON block.

Here is the program skeleton, with comments showing what the user must supply (flagged USER SUPPLIES). Note that for no-host-process implementations, the two subroutines hostmain and gridmain are replaced by a single subroutine procmain.

```
C=======================================================================
C=======================================================================
C
C       sample program p0 -- dummy (no user-supplied code)
C
C=======================================================================
C=======================================================================

C=======================================================================
C
C       host process-main program
C
C=======================================================================

        subroutine hostmain

        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'
```

```
C=======USER SUPPLIES common block(s), declarations, body of program
C       include '?'
C=======end of USER SUPPLIES
        end

C=======================================================================
C
C       grid process-main program
C
C=======================================================================

        subroutine gridmain

        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'

C=======USER SUPPLIES common block(s), declarations, body of program
C       include '?'
C=======end of USER SUPPLIES

        end

C=======================================================================
C
C       additional routines
C
C=======================================================================

C=======USER SUPPLIES additional routines (optional)
C       routines should generally begin with:
C       include 'mesh_uparms.h'
C       include 'mesh_parms.h'
C       include 'mesh_common.h'
C       include(s) for user common blocks, if any
C=======end of USER SUPPLIES
```

### 2.4 Subroutines and functions

The archetype supplies many subroutines and functions: to perform the needed collective communication operations, to perform reduction operations, to aid in managing global and local indices, etc.

In current implementations, these routines are provided as source file `mesh_lib.FM` (for Fortran M) or `mesh_lib.F` (for other versions). Comment headers in the source file provide a detailed specification for each routine, in the form of a precondition (what must hold before the routine is called) and a postcondition (what will hold after the routine completes). The remainder of this section lists the available routines by category and comments on suggested usage.

*CAUTION:* Care must be taken to ensure that the host process and the grid processes coordinate properly, because some subroutines (those for moving data to and from the host process and those for performing global reductions) must be executed by *all* processes in order to function correctly. One way to ensure this is to give the two programs (host and grid) the same overall structure, differentiating between them only when necessary. For example, in a program with a time-step loop, the grid program might look like this:

```
        do m = 1, NSTEPS
C               local computation
                call mesh_update_bdry(mydata)
                do i = 1, NXlocal
                do j = 1, NYlocal
                do k = 1, NZlocal
                        .... computation ....
                enddo
                enddo
                enddo
C               move to host and print
                call mesh_GtoH_grid(mydata)
C               (no printing in grid process)
        enddo
```

while the host program would look like this:

```
        do m = 1, NSTEPS
C               local computation
C               (no computation in host process)
C               move to host and print
                call mesh_GtoH_host(mydata_host)
                .... print ....
        enddo
```

### 2.4.1 Subroutines to redistribute data

These subroutines copy a grid-based array from the host process (undistributed form) to the grid processes (distributed form) and vice versa:

```
subroutine mesh_HtoG_host(host_array)
real host_array(....)
subroutine mesh_HtoG_grid(grid_array)
real grid_array(....)

subroutine mesh_GtoH_host(host_array)
real host_array(....)
subroutine mesh_GtoH_grid(grid_array)
real grid_array(....)
```

Observe that there are two subroutines for each operation, one that executes in the host process and one that executes in the grid processes.

*NOTE:* These subroutines are not available in the no-host-process implementations of the archetype.

### 2.4.2 Subroutines to exchange boundary data

Current implementations provide a single routine to update the ghost boundaries for a distributed array:

```
subroutine mesh_update_bdry(array)
real array(....)
```

This subroutine updates the ghost boundaries of `array` (by exchanging boundary information with neighboring processes). The corners of the ghost boundaries are updated, but ghost boundaries that correspond to the boundaries of the global array are not. (Observe that the width of the ghost boundaries — i.e., the width of the boundary to exchange — is specified by the user via `PARAMETER NGHOST`.)

*NOTE:* Some implementations also include alternative boundary-exchange operations, including:

- A boundary exchange operation that does not update the corners of the ghost section — this is potentially somewhat faster.
- A boundary exchange operation that treats the grid as a torus rather than a mesh — i.e., fills in ghost boundaries for processes on the boundaries of the process grid with values from processes on the opposite border. (E.g., the left ghost boundary of a process on the left edge of the process grid contains values from the right boundary of the corresponding process on the right edge of the process grid.)

Refer to comments in the implementation code for details.

### 2.4.3 Subroutines for global reductions

Current implementations have several subroutines for performing reduction operations:

```
C       maximum (integer)
        subroutine mesh_merge_int_max(isize, int_in, int_out)
        integer int_in(isize), int_out(isize)

C       maximum absolute value (real)
        subroutine mesh_merge_real_maxabs(isize, real_in, real_out)
        real real_in(isize), real_out(isize)

C       sum (real)
        subroutine mesh_merge_real_sum(isize, real_in, real_out)
        real real_in(isize), real_out(isize)

C       synchronize only
        subroutine mesh_synch
```

Additional subroutines may be provided in future implementations, depending on feedback from users. Users are also welcome to write their own reduction subroutines by scavenging code from the provided routines. (E.g., a subroutine for computing a global minimum is almost identical to a subroutine for computing a global maximum.)

### 2.4.4 Subroutines for broadcasting data

These subroutines broadcast data from the host process (or, for no-host-process implementations, the designated I/O process) to the grid processes. They would typically be used during initialization to set global constants — the host process would read the constants from a file and then broadcast them to the grid processes.

```
        subroutine mesh_bcast_int(isize, int_array)
        integer int_array(isize)

        subroutine mesh_bcast_real(isize, real_array)
        real real_array(isize)
```

### 2.4.5 Subroutines for parallel I/O

Current implementations do not contain support for parallel I/O operations, since at this time there is little consensus about how to support parallel I/O in a portable way. It is not difficult, however, for applications to perform a simple type of parallel I/O using the following approach. Data can be written as follows:

1. Each grid process opens a uniquely-named file. (The filename should be generated using the subroutine `fname`, described below.)
2. Each grid process writes its local section to its file using standard Fortran I/O in whatever format the user chooses.
3. If necessary, after the program completes, a separate utility program combines the separate files (each containing data from a single local section) into a single file containing data from the whole array.

Reading data is analogous; if necessary, a separate utility program splits a single file into separate files for the individual grid processes.

In addition, one of our implementations contains a set of routines for "portable parallel I/O", which take the above approach but use machine-independent formats for disk data. Two formats are provided, compressed and uncompressed. Additional utility programs are required to translate this machine-independent format to a standard Fortran I/O format.

These subroutines and utility programs are briefly described below. For more information, communicate with the author of this document (berna@cs.caltech.edu) or Rajit Manohar (rajit@cs.caltech.edu).

**Subroutines**

- Subroutines to open parallel I/O files:

```
subroutine mesh_openwrite(filename, ifd)
subroutine mesh_openread(filename, ifd)
subroutine mesh_copenwrite(filename, ifd)
subroutine mesh_copenread(filename, ifd)
```

- Subroutines to write to parallel I/O files:

```
subroutine mesh_cwrite(ifd, array)
subroutine mesh_write(ifd, array)
```

- Subroutines to read from parallel I/O files:

```
subroutine mesh_cread(ifd, array)
subroutine mesh_read(ifd, array)
```

- Subroutines to close parallel I/O files:

```
subroutine mesh_cclose(ifd)
subroutine mesh_close(ifd)
```

**Utility programs**

- Program `split` splits a single file (compressed or uncompressed format) into files for use by the parallel I/O read subroutines.
- Program `merge` merges files generated by the parallel I/O write routines (compressed or uncompressed format) into a single file.
- Programs to convert from standard formats to the machine-independent compressed or uncompressed format are in work.

#### 2.4.6 Utility subroutines and functions

Current implementations include an assortment of utility routines:

- Subroutines to convert process type and process indices to process number, and vice versa. (These routines identify each process with a unique single number, which is sometimes useful.)

  ```
  integer function iprocnum(iptype, ipx, ipy, ipz)
  subroutine pxpypz(iproc, iptype, ipx, ipy, ipz)
  ```

  In no-host-process implementations, the argument `iptype` is omitted.

- Subroutines to convert from global indices to local indices (plus process-grid indices), and vice versa.

  ```
  subroutine ilocal(ii, ipx, iiloc)
  subroutine jlocal(jj, ipy, jjloc)
  subroutine klocal(kk, ipz, kkloc)
  subroutine iglobal(ipx, iiloc, ii)
  subroutine jglobal(ipy, jjloc, jj)
  subroutine kglobal(ipz, kkloc, kk)
  ```

- Subroutines to compute the intersection of a range of global indices with the local intersection. (This is useful in determining, e.g., whether the local section contains part of a boundary and if so what the local indices are.)

  ```
        subroutine xintersect(iglob1, iglob2, iloc1, iloc2,
       -         lempty)
        subroutine yintersect(jglob1, jglob2, jloc1, jloc2,
       -         lempty)
        subroutine zintersect(jglob1, jglob2, kloc1, kloc2,
       -         lempty)
  ```

- A subroutine to append a unique-to-each-process suffix to a filename. (This allows each process to access a different file.)

```
      subroutine fname(iptype, ipx, ipy, ipz, inname,
     -          outname)
```

  In no-host-process implementations, the argument `iptype` is omitted.

# 3 Compiling and linking

**Fortran M version**

This section describes how to compile and link a program based on the Fortran M implementation. It assumes that the Fortran M compiler (`fm`) [2, 1] is installed.

To compile and link a program based on the Fortran M implementation:

- Compile the "main program file" — the file containing the archetype-supplied program skeleton plus the user-supplied code for the programs that execute in the host and grid processes.
- Compile any additional source files (e.g., for user-written subroutines called from the host and grid programs).
- Compile the archetype library routines (`mesh_lib.FM`).
- Link the results of the above compilations.

For example, to compile the skeleton program `p0.FM` shown earlier:

```
fm -c p0.FM
fm -c mesh_lib.FM
fm -o p0.exe p0.o mesh_lib.o
```

Observe that the library routines and the main program file must be recompiled if any of the `PARAMETER`s in `mesh_uparms.h` are changed — this is because of the static nature of Fortran 77, on which Fortran M is based.

**Fortran M version with parallel I/O**

To include the optional parallel I/O, compile all the provided C routines (`.c` files) with a C compiler and include the resulting `.o` files in the above link. See the sample `Makefile` for an example. The utility programs `split` and `merge` must be compiled separately.

### Other implementations

Compilation is similar to that for Fortran M, but using the appropriate compiler and/or message-passing library. See the sample `Makefile` for an example.

# 4 Short example application programs

This section presents some short, simple example applications illustrating use of the archetype and its library routines.

## 4.1 "Hello, world"

This example is a simple "hello, world" program. Recall that in addition to providing code for the host and grid processes, the user must also modify the archetype `INCLUDE` file `mesh_uparms.h` to provide the desired values for the dimensions of the process grid (`XPROCS`, `YPROCS`, and `ZPROCS`), the dimensions of the data grid (`NX`, `NY`, and `NZ`), and the size of the ghost boundary (`NGHOST`). In this example, we show both the user-supplied code and the complete archetype-supplied `INCLUDE` file `mesh_uparms.h`.

### 4.1.1 Code for host and grid processes

```
C=====================================================================
C=====================================================================
C
C       sample program p1 -- "hello":
C               each process prints "hello" message
C
C=====================================================================
C=====================================================================

C=====================================================================
C
C       host process-main program
C
C=====================================================================

        subroutine hostmain

        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'

C=======USER-SUPPLIED common block(s), declarations, body of program

        print*, 'hello from host process'
```

```
C=======end of USER-SUPPLIED code
        end

C=======================================================================
C
C       grid process-main program
C
C=======================================================================

        subroutine gridmain

        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'

C=======USER-SUPPLIED common block(s), declarations, body of program

C       iprocx, iprocy, iprocz are defined in mesh_common.h
        print*, 'hello from grid process at ', iprocx, iprocy, iprocz

C=======end of USER-SUPPLIED code

        end
```

### 4.1.2 Archetype file mesh_uparms.h

```
C=======================================================================
C
C       user-set parameters for mesh-computation template
C
C=======================================================================

C
C       The mesh computation is based on the idea of a 3D grid of
C       processes.  Parameters XPROCS, YPROCS, and ZPROCS give the
C       dimensions of that grid.
C
        integer XPROCS, YPROCS, ZPROCS
        parameter (XPROCS=2)
        parameter (YPROCS=2)
        parameter (ZPROCS=2)

C
C       Arrays to be distributed among grid processes are assumed
C       to have dimensions NX by NY by NZ.  mesh_parms.h
C       defines parameters NXlsize, NYlsize, and NZlsize as
```

```
C       dimensions for local sections of such arrays, based on
C       parameters NX, NY, NZ, and the process grid sizes defined
C       above.
C
C       CAUTION:
C
C       The dimensions of the process grid need not evenly divide
C       the dimensions of the grid.  (E.g., XPROCS need not divide
C       NX.)  However, to use the X dimension as an example:
C       The template assumes that the X-dimension of the grid
C       is broken up into XPROCS sections, of which all but the
C       last are of size NXlsize (= ceiling(NX/XPROCS)).  The
C       size of the last section is given by:
C               NX - (XPROCS-1)*NXlsize
C       You must avoid setting NX and XPROCS such that this
C       quantity is non-positive.  (Example:  If NX=5 and XPROCS=4,
C       then NXlsize=2, and the above quantity is -1, which is not
C       valid.)
C

        PARAMETER (NX=8,NY=6,NZ=4)


C
C       In addition, distributed arrays may need "ghost boundaries"
C       to hold values from neighboring processes -- a ghost boundary
C       of size N is appropriate for array X if computation for a
C       particular cell (of X or some other array value) depends on
C       the values of X in N neighboring cells in each direction.
C       Parameter NGHOST gives the size of such ghost boundaries.
C       mesh_parms.h uses this parameter to define parameters
C       IXLO, IXHI, IYLO, IYHI, IZLO, and IZHI, which can be used
C       to dimension arrays with ghost boundaries -- e.g.,
C       X(IXLO:IXHI,IYLO:IYHI,IZLO:IZHI).
C
C       CAUTION:
C
C       The width of the ghost boundaries must not exceed the size
C       of the local sections.
C
        PARAMETER (NGHOST=1)
```

## 4.2 1D heat equation

In this example, the goal is to solve the 1D heat diffusion equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$

using the approximation:

$$\frac{U(x_i, t_{k+1}) - U(x_i, t_k)}{\Delta t} = \frac{U(x_{i+1}, t_k) - 2U(x_i, t_k) + U(x_{i-1}, t_k)}{\Delta x^2}$$

A sequential program for this computation is straightforward. It maintains two copies of variable $U$, one for the current time step (`uk`) and one for the next time step (`ukp1`). At each time step, it computes the values of `ukp1` based on the values of `uk`. Observe that the two boundary points are handled differently — they maintain a constant value.

An equivalent parallel program using the mesh archetype is not much more complicated:

- The declaration of the size of the grid (`NX`) is moved to `INCLUDE` file `mesh_uparms.h`. Parameter `NGHOST` is 1 for this application, since `ukp1` depends on values of `uk` at most one step away. Parameter `XPROCS` is chosen by the user.

- Grid-based variables `uk` and `ukp1` are distributed among grid processes. Observe the use of archetype-supplied constants `IXLO` and `IXHI` to declare the size of the local sections — these constants give the proper dimensions of the local section, including ghost boundaries, while allowing the indices of the local section proper (exclusive of ghost boundaries) to range from 1 to `NXlsize`.

- The whole array is initialized in the host process and then copied to the grid processes (routines `mesh_HtoG_host` and `mesh_HtoG_grid`). (It could also be initialized directly in the grid processes; this approach was chosen for simplicity.)

- At each time step, `mesh_update_bdry` is called to update the ghost boundaries before they are used in the grid computation. The special handling for the (global) boundary points is provided by using archetype library routine `xintersect` to determine which points in the local section are in the interior of the global array (global indices 2 through `NX - 1`). The grid values are then copied back to the host process for printing (routines `mesh_GtoH_host` and `mesh_GtoH_grid`).

Observe that the code executed by the host and grid processes has the same high-level structure — both execute the time-step loop, for example. This ensures that proper synchronization is maintained.

### 4.2.1 Sequential program

```
C=======================================================================
C=======================================================================
```

```
C
C       example program heat:
C               explicitly solves the 1D diffusion equation
C
C=======================================================================
C=======================================================================

        program heat

C=======grid size
        integer NX
        parameter (NX=100)

        integer NSTEPS
        parameter (NSTEPS=100)

C=======grid variables
        real uk(1:NX), ukp1(1:NX)

        dx = 1.0/NX
        dt = 0.5*dx*dx

C=======initialization
        do i= 2,NX-1
           uk(I)=0.0
        enddo
        uk(1)=1.0
        uk(NX)=1.0

C=======time step loop
        do k=1,NSTEPS
C===========grid computation
            do i=2,NX-1
                ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
            enddo
            do i=2,NX-1
                uk(i)=ukp1(i)
            enddo
C===========sequential output
            print*, 'timestep ', k
            print 25,(uk(I),I=1,NX)
25          format(4X,E15.5)
        enddo

        end

C=======================================================================
```

### 4.2.2 Parallel program (user-supplied code)

```
C=======================================================================
C=======================================================================
C
C       example program heat:
C               explicitly solves the 1D diffusion equation
C
C=======================================================================
C=======================================================================

C=======================================================================
C
C       host process-main program
C
C=======================================================================

        subroutine hostmain

        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'

C=======grid size
C       moved to archetype file mesh_uparms.h
C       integer NX
C       parameter (NX=100)
        integer NSTEPS
        parameter (NSTEPS=100)

C=======grid variables (whole array in host)
        real uk(1:NX)

        dx = 1.0/NX
        dt = 0.5*dx*dx

C=======initialization
        do i=2,NX-1
           uk(i)=0.0
        enddo
        uk(1)=1.0
        uk(NX)=1.0
        call mesh_HtoG_host(uk)

C=======time step loop
        do k=1,NSTEPS
C===========grid computation (grid processes only)
C===========sequential output
```

```
            call mesh_GtoH_host(uk)
            print*, 'timestep ', k
            print 25,(uk(I),I=1,NX)
25          format(4X,E15.5)
        enddo
        end

C====================================================================
C
C       grid process-main program
C
C====================================================================

        subroutine gridmain

        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'

C=======grid size
C       moved to archetype file mesh_uparms.H
C       integer NX
C       parameter (NX=100)
        integer NSTEPS
        parameter (NSTEPS=100)

C=======grid variables (local sections)
        real uk(IXLO:IXHI),ukp1(IXLO:IXHI)

        dx = 1.0/NX
        dt = 0.5*dx*dx

C=======initialization
        call mesh_HtoG_grid(uk)

C=======time step loop
        do k=1,NSTEPS
C===========grid computation
            call mesh_update_bdry(uk)
            call xintersect(2,NX-1,istart,iend,iempty)
            do i=istart,iend
                ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
            enddo
            do i=istart,iend
                uk(i)=ukp1(i)
            enddo
C===========sequential output (I/O in host process only)
```

```
        call mesh_GtoH_grid(uk)
    enddo
    end

C=======================================================================
```

## 4.3 2D Poisson solver

In this example, the goal is to solve the 2D Poisson problem:

$$-\frac{\partial^2 U}{\partial x^2} - \frac{\partial^2 U}{\partial y^2} = f(x, y)$$

with Dirichlet boundary condition

$$u(x, y) = g(x, y)$$

using Jacobi iteration; i.e., by discretizing the problem domain and applying the following operation to all interior points until convergence is reached:

$$4u^{(k+1)}_{(i,j)} = h^2 f_{i,j} + u^{(k)}_{(i-1,j)} + u^{(k)}_{(i+1,j)} + u^{(k)}_{(i,j-1)} + u^{(k)}_{(i,j+1)}$$

(This example is based on the discussion of the Poisson problem in [3]).

A sequential program for this computation is straightforward. It maintains two copies of variable $u$, one for the current iteration (`uk`) and one for the next iteration (`ukp1`). At each iteration, it computes the values of `ukp1` based on the values of `uk`. Observe that the boundary points are handled differently — they maintain a constant value. Every `NCHECK` steps, the maximum of $|u^{(k+1)}_{(i,j)} - u^{(k)}_{(i,j)}|$ is computed to check for convergence.

An equivalent parallel program using the mesh archetype is not much more complicated:

- The declarations of the dimensions of the grid (`NX` and `NY`) are moved to `INCLUDE` file `mesh_uparms.h`. Parameter `NGHOST` is 1 for this application, since `ukp1` depends on values of `uk` at most one step away. Parameters `XPROCS` and `YPROCS` are chosen by the user.

- Grid-based variables `uk` and `ukp1` are distributed among grid processes. Observe the use of archetype-supplied constants to declare their dimensions.

- The whole grid is initialized in the host process and then copied to the grid processes (routines `mesh_HtoG_host` and `mesh_HtoG_grid`). (It could also be initialized directly in the grid processes; this approach was chosen for simplicity.)

- At each iteration, `mesh_update_bdry` is called to update the ghost boundaries before they are used in the grid computation. The special handling for the (global) boundary points is provided by using archetype library routines `xintersect` and `yintersect` to determine which points in the local section are in the interior of the global grid. Computing a global maximum for the convergence test is accomplished by computing a local maximum in each grid process and then calling archetype library routine `mesh_merge_real_maxabs` to find the global maximum. (After calling `mesh_merge_real_maxabs`, every process, including the host process, has the value of the global maximum.)
- When convergence is reached (or `MAXSTEPS` iterations have been performed), grid values are copied back to the host process for printing (routines `mesh_GtoH_host` and `mesh_GtoH_grid`).

Observe that the code executed by the host and grid processes has the same high-level structure — both execute the main loop, for example, including the convergence test. This ensures that proper synchronization is maintained.

### 4.3.1 Sequential program

```
C=====================================================================
C=====================================================================
C
C       example program poisson:
C               uses Jacobi relaxation to solve the Poisson equation
C
C=====================================================================
C=====================================================================

        program poisson

C=======grid size
        integer NX
        integer NY
        parameter (NX=10)
        parameter (NY=10)

        real H
        parameter (H=0.05)
C       how often to check for convergence
        integer NCHECK
        parameter (NCHECK=10)
C       convergence criterion
        real TOL
        parameter (TOL=0.00001)
C       maximum number of steps
```

```
        integer MAXSTEPS
        parameter (MAXSTEPS=1000)

        external F
        external G

C=======grid variables
        real uk(1:NX,1:NY), ukp1(1:NX,1:NY)

        real diff, diffmax

C=======initialization
C       interior points
        do i = 2, NX-1
        do j = 2, NY-1
                uk(i,j) = F(i,j,NX,NY,H)
        enddo
        enddo
C       boundary points
        do j = 1, NY
                uk(1,j) = G(1,j,NX,NY,H)
                uk(NX,j) = G(NX,j,NX,NY,H)
        enddo
        do i = 2, NX-1
                uk(i,1) = G(i,1,NX,NY,H)
                uk(i,NY) = G(i,NY,NX,NY,H)
        enddo

C=======main loop (until convergence)
        diffmax = TOL + 1.0

        do k=1,MAXSTEPS

C=======grid computation (compute new values)
        do i = 2, NX-1
        do j = 2, NY-1
                ukp1(i,j) = 0.25*(H*H*F(i,j,NX,NY,H)
     -                  + uk(i-1,j) + uk(i,j-1)
     -                  + uk(i+1,j) + uk(i,j+1) )
        enddo
        enddo
C=======convergence test (global max)
        if (mod(k,NCHECK) .eq. 0) then
                diffmax = 0.0
                do i = 2, NX-1
                do j = 2, NY-1
                        diff = abs(ukp1(i,j) - uk(i,j))
```

```
                        if (diff .gt. diffmax) diffmax = diff
                enddo
                enddo
        endif
C=======grid computation (copy new values to old values)
        do i = 2, NX-1
        do j = 2, NY-1
                uk(i,j) = ukp1(i,j)
        enddo
        enddo

C=======convergence check
        if (diffmax .le. TOL) go to 1000

        enddo

C=======sequential output
1000    continue
        print*, 'NX, NY = ', NX, NY
        print*, 'H = ', H
        print*, 'tolerance = ', TOL
        call Fprint
        call Gprint

        if (diffmax .le. TOL) then
                print*, 'convergence occurred in ', k, ' steps'
        else
                print*, 'no convergence in ', MAXSTEPS, ' steps; ',
     -                  'max. difference ', diffmax
        endif

        do i = 1, NX
                if (NX .gt. 10) print*, ' '
                print 9999, (uk(i,j), j = 1, NY)
        enddo
9999    format(10F8.4)

        end

C=======================================================================

        real function F(i,j,nx,ny,h)
        integer i, j, nx, ny
        real h

        F = 0.0
        end
```

```
        subroutine Fprint
        print*, 'F(i,j) = 0.0'
        end

C========================================================================

        real function G(i,j,nx,ny,h)
        integer i, j, ny, ny
        real h

        G = (i+j)*h
        end

        subroutine Gprint
        print*, 'G(i,j) = (i+j)*H'
        end

C========================================================================
```

### 4.3.2 Parallel program (user-supplied code)

```
C========================================================================
C========================================================================
C
C       example program poisson:
C               uses Jacobi relaxation to solve the Poisson equation
C
C========================================================================
C========================================================================

C========================================================================
C
C       host process-main program
C
C========================================================================

        subroutine hostmain
        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'

C=======grid size
C       moved to archetype file mesh_uparms.h
C       integer NX
C       integer NY
C       parameter (NX=10)
C       parameter (NY=10)
```

```
      real H
      parameter (H=0.05)
C     how often to check for convergence
      integer NCHECK
      parameter (NCHECK=10)
C     convergence criterion
      real TOL
      parameter (TOL=0.00001)
C     maximum number of steps
      integer MAXSTEPS
      parameter (MAXSTEPS=1000)

      external F
      external G

C=======grid variables (whole array in host)
      real uk(1:NX,1:NY), ukp1(1:NX,1:NY)

      real diff, difflocal, diffmax

C=======initialization
C     interior points
      do i = 2, NX-1
      do j = 2, NY-1
              uk(i,j) = F(i,j,NX,NY,H)
      enddo
      enddo
C     boundary points
      do j = 1, NY
              uk(1,j) = G(1,j,NX,NY,H)
              uk(NX,j) = G(NX,j,NX,NY,H)
      enddo
      do i = 2, NX-1
              uk(i,1) = G(i,1,NX,NY,H)
              uk(i,NY) = G(i,NY,NX,NY,H)
      enddo
C     move to grid
      call mesh_HtoG_host(uk)

C=======main loop (until convergence)
      diffmax = TOL + 1.0

      do k=1,MAXSTEPS

C=======grid computation (compute new values; in grid only)
C=======convergence test (global max)
```

```
        if (mod(k,NCHECK) .eq. 0) then
C               (computation in grid only)
                call mesh_merge_real_maxabs(1, difflocal, diffmax)
        endif
C=======grid computation (copy new values to old values; in grid only)

C=======convergence check
        if (diffmax .le. TOL) go to 1000

        enddo

C=======sequential output
1000    continue
        call mesh_GtoH_host(uk)
        print*, 'NX, NY = ', NX, NY
        print*, 'H = ', H
        print*, 'tolerance = ', TOL
        call Fprint
        call Gprint

        if (diffmax .le. TOL) then
                print*, 'convergence occurred in ', k, ' steps'
        else
                print*, 'no convergence in ', MAXSTEPS, ' steps'
                print*, 'no convergence in ', MAXSTEPS, ' steps; ',
     -                  'max. difference ', diffmax
        endif

        do i = 1, NX
                if (NX .gt. 10) print*, ' '
                print 9999, (uk(i,j), j = 1, NY)
        enddo
9999    format(10F8.4)

        end

C======================================================================
C
C       grid process-main program
C
C======================================================================

        subroutine gridmain
        include 'mesh_uparms.h'
        include 'mesh_parms.h'
        include 'mesh_common.h'
```

```
C=======grid size
C       moved to archetype file mesh_uparms.h
C       integer NX
C       integer NY
C       parameter (NX=10)
C       parameter (NY=10)

        real H
        parameter (H=0.05)
C       how often to check for convergence
        integer NCHECK
        parameter (NCHECK=10)
C       convergence criterion
        real TOL
        parameter (TOL=0.00001)
C       maximum number of steps
        integer MAXSTEPS
        parameter (MAXSTEPS=1000)

        external F
        external G

C=======grid variables (local sections)
        real uk(IXLO:IXHI,IYLO:IYHI), ukp1(1:NXlsize,1:NYlsize)

        real diff, difflocal, diffmax

        logical iempty, jempty

C=======initialization (in host only)
        call mesh_HtoG_grid(uk)

C=======main loop (until convergence)

C       compute loop bounds
        call xintersect(2, NX-1, istart, iend, iempty)
        call yintersect(2, NY-1, jstart, jend, jempty)

        diffmax = TOL + 1.0

        do k=1,MAXSTEPS

C=======grid computation (compute new values)
        call mesh_update_bdry(uk)
        do i = istart, iend
        do j = jstart, jend
                ukp1(i,j) = 0.25*(H*H*F(i,j,NX,NY,H)
```

```
     -                   + uk(i-1,j) + uk(i,j-1)
     -                   + uk(i+1,j) + uk(i,j+1) )
        enddo
        enddo
C=======convergence test (global max)
        if (mod(k,NCHECK) .eq. 0) then
                difflocal = 0.0
                do i = istart, iend
                do j = jstart, jend
                        diff = abs(ukp1(i,j) - uk(i,j))
                        if (diff .gt. difflocal) difflocal = diff
                enddo
                enddo
                call mesh_merge_real_maxabs(1, difflocal, diffmax)
        endif
C=======grid computation (copy new values to old values)
        do i = istart, iend
        do j = jstart, jend
                uk(i,j) = ukp1(i,j)
        enddo
        enddo

C=======convergence check
        if (diffmax .le. TOL) go to 1000

        enddo

C=======sequential output (I/O in host only)
1000    continue
        call mesh_GtoH_grid(uk)

        end

C========================================================================

        real function F(i,j,nx,ny,h)
        integer i, j, nx, ny
        real h

        F = 0.0
        end

        subroutine Fprint
        print*, 'F(i,j) = 0.0'
        end

C========================================================================
```

```
      real function G(i,j,nx,ny,h)
      integer i, j, ny, ny
      real h

      G = (i+j)*h
      end

      subroutine Gprint
      print*, 'G(i,j) = (i+j)*H'
      end

C======================================================================
```

# 5 How to obtain the code

The following files are available via Web page http://www.etext.caltech.edu/Implementations/Mesh/:

- 1D Fortran M implementation (`FM1D.tar.gz`).
- 2D Fortran M implementation (`FM2D.tar.gz`).
- 2D Fortran M implementation with portable parallel I/O (`FM2D_parIO.tar.gz`).
- 3D Fortran M implementation (`FM3D.tar.gz`).
- 3D Fortran M implementation without host process (`FM3D_nohost.tar.gz`).
- 2D Fortran-plus-p4 implementation (`P4_2D.tar.gz`).
- 2D Fortran-plus-NX implementation (`NX2D.tar.gz`).
- 2D Fortran-plus-NX implementation without host process (`NX2D_nohost.tar.gz`).

Each file contains the tar'd and gzip'd form of a directory containing:

- Required source files.
- A sample `Makefile`.
- (Fortran M implementations) A directory of sample programs, containing source code and sample output.

## References


[1] Argonne National Laboratory. The Fortran M programming language. Available via http://www.mcs.anl.gov/fortran-m/FM.html.

[2] I. T. Foster and K. M. Chandy. FORTRAN M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[3] E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.