The Mesh-Spectral Archetype

Greg Davis and Berna L. Massingill

Abstract

A parallel program archetype [2] 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-spectral 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 tutorial example application programs.

1 Introduction

The mesh-spectral archetype is a combination and generalization of the mesh [5] and spectral [3] archetypes. In this section, we describe the data structures and operations on which this archetype is based and a strategy for parallelizing it.

1.1 Mesh-spectral computations

In a mesh-spectral computation:

- Data is based on 3-dimensional grids (arrays). A computation may contain multiple grids of different dimensions. Observe that 1- and 2-dimensional grids are special cases of 3-dimensional grids.
- Computation consists of some sequence of the following operations:
  - neighbor operations in which new values are computed for each point in a grid based on values at that point and nearby points.
  - row operations, in which new values are computed for each point in a grid based on values in the same row.
  - column operations, defined analogously.
  - reduction operations over a grid, e.g., global maximum.

Figures 1 and 2 illustrate two of these operations — a neighbor operation and a row operation respectively — in a 2-dimensional grid.

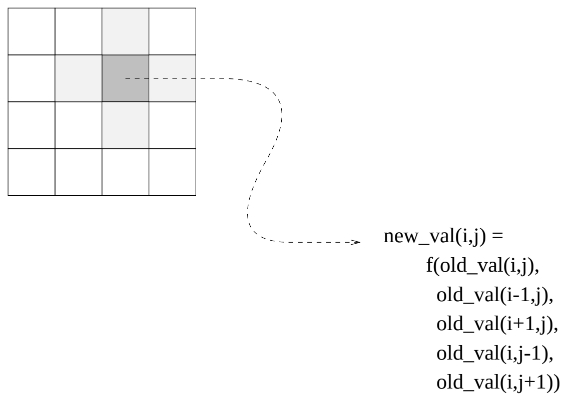

Figure 1: Neighbor operation.

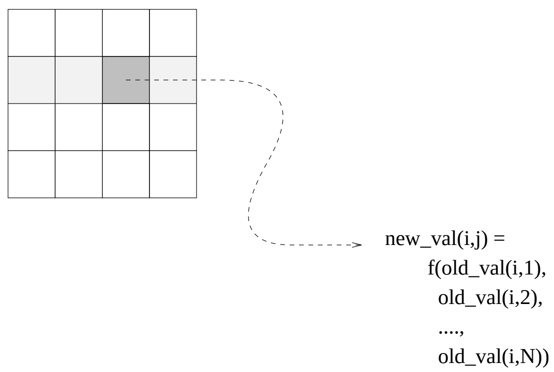

Figure 2: Row operation.

1.2 Parallelization strategy

Mesh-spectral computations are readily parallelized using the following approach:

- The overall structure of the computation is based on the SPMD (single program, multiple data) model; that is, it consists of some number N of processes all executing the same program, each on its own data.
- Each 3-dimensional data grid is distributed over the processes based on a 3-dimensional *process grid* of some or all of the N processes. It is usually the case that at any point in the computation all data grids will be distributed over the same process grid, but this is not required. Also, in the course of a computation, a data grid can be redistributed — i.e., the process grid over which it is distributed can be changed. §1.2.3 gives an example of such a computation.
- Non-grid variables — global constants and reduction variables, e.g. — are duplicated in each process.

Although this strategy is particularly applicable to distributed-memory architectures, it can be implemented on shared-memory architectures as well.

1.2.1 Distributing the data

Figure 3 illustrates how a 2-dimensional grid is distributed over a 2-dimensional process grid: The original (undistributed) 8-by-8 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 grid process (3,2).

Figure 4 illustrates two different distributions of a 8-by-8 array, one by rows and one by columns. The two shaded squares (global indices (2,7) and (7,4)) illustrate the different positions of the same data-grid element in different distributions.

1.2.2 Neighbor operations

Neighbor operations 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

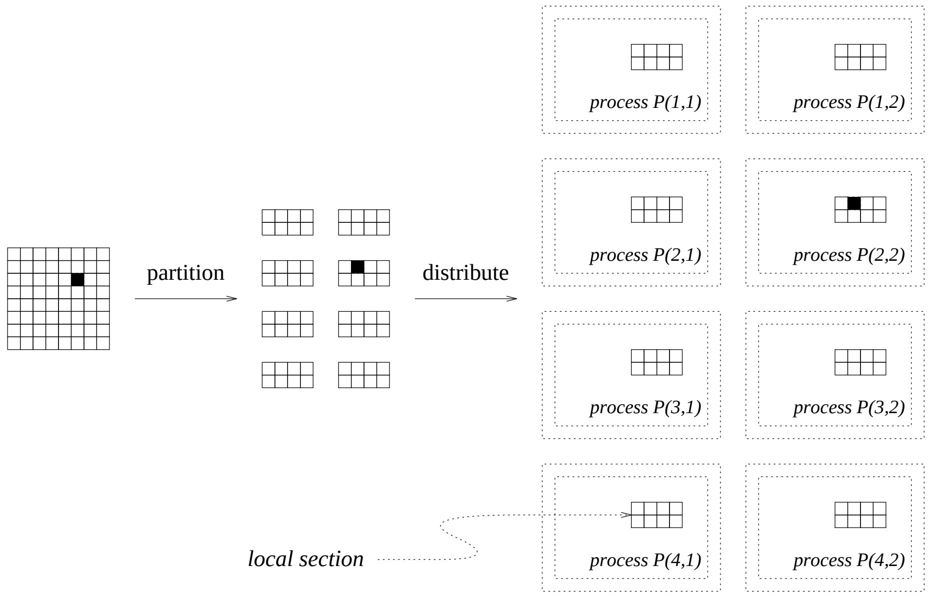

Figure 3: Distributing data by blocks.

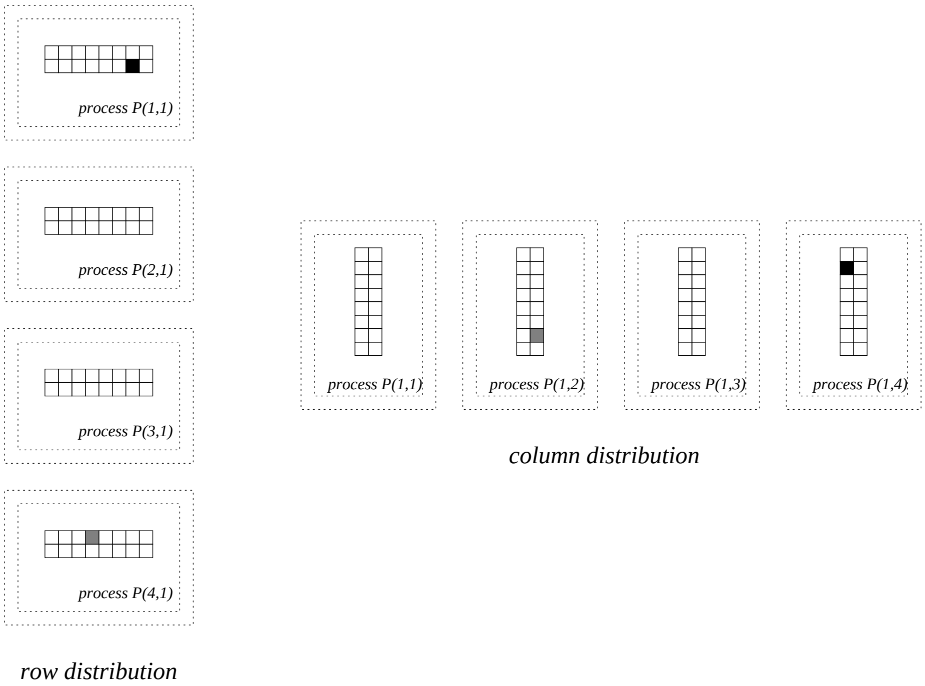

Figure 4: Distributing data by rows or columns.

with a *ghost boundary*, which is used as a buffer to contain a shadow copy of data from neighboring local sections. Figure 5 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. Values

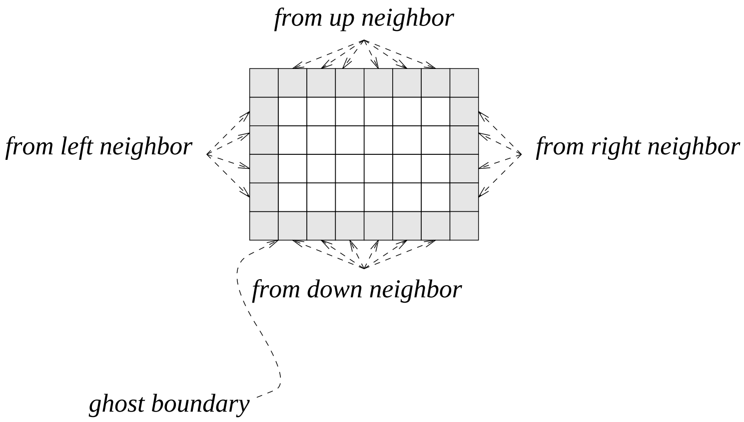

Figure 5: Local section with ghost boundary.

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 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 on the points in the local section shown in figure 5. (Both arrays mentioned by the formula – `newval` and `oldval` – have a local section of the form shown.) The code (in Fortran) would look something like that shown in figure 6.

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

1.2.3 Row or column operations

Row operations can be done just as they would be done in a sequential program, if the data is distributed by rows. Similarly, column operations are easily done if the data is distributed by columns. If the computation contains alternating

```
parameter (N=4)
parameter (M=6)
real newval(0:N+1, 0:M+1)
real oldval(0:N+1, 0:M+1)
....
call boundary_exchange(oldval)
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
```

Figure 6: Pseudocode for neighbor operation.

row and column operations (as is the case in *spectral-methods* computations), data is redistributed with *redistribution* operations. (For example, to perform a row and then a column operation, the data would initially be distributed by rows. The row operation would be performed, after which the data would be redistributed by columns and the column operation performed.)

As an example, consider performing an alternating series of row and column operations. Data is alternately distributed by rows and by columns (as illustrated in figure 4), with the same memory used to store the local sections for both distributions. The code (in Fortran) would look something like that shown in 7, (omitting for now the details of the arguments to the data redistribution routine):

1.2.4 Performing I/O

There are many approaches to performing I/O on a distributed grid, including:

- Sequential I/O from a single process. In this approach, a grid is distributed over a 1 by 1 by 1 process grid (i.e., gathered in a single process), and sequential I/O is performed from that process. A redistribution operation can be used to collect the data in one process before a write or scatter the data after a read. This approach is simple and portable, but it requires sufficient memory to collect the whole grid in one process, and it is not concurrent.

- "Blocked sequential" I/O from a single process. In this approach, a single process performs the required I/O on one local section at a time, communicating with other processes as needed. (I.e., in a write operation, the other processes send their local sections to the designated I/O process, which performs the desired file writes, while in a read operation the

```
parameter (N=8)
parameter (NP=4)
parameter (Nlocal=N/NP)
real data_by_rows(Nlocal, N)
real data_by_cols(N, Nlocal)
equivalence (data_by_rows, data_by_cols)
....
do k=1,NSTEPS
        do i=1,Nlocal
                call rowop(i,data_by_rows)
        enddo
        call redistribute_data(data_by_rows, < by columns >)
        do j=1,Nlocal
                call colop(j,data_by_cols)
        enddo
        call redistribute_data(data_by_cols, < by rows >)
enddo
```

Figure 7: Pseudocode for row/column operations.

designated I/O process performs the desired file reads and sends local sections to the other processes.) This approach is somewhat less simple but still portable, and it requires no additional memory, though it too is not concurrent.

- System-independent parallel I/O. In this approach, each process opens a separate file and performs I/O on its local section. In terms of simplicity, portability, and memory use, this approach is similar to the blocked sequential I/O described above. Whether it is truly concurrent is system-dependent.

- System-dependent parallel I/O. In this approach, processes use system-dependent routines to perform true parallel I/O. This approach requires no additional memory and is likely to be more concurrent than the approaches described above, but it is not portable, and its simplicity depends on the particular system routines used.

The mesh-spectral archetype provides explicit support for the second method (blocked sequential I/O), and the first and third methods can be readily implemented, as the example programs show. The fourth method can also be used with the archetype, but we do not address specifics, since it is not portable.

1.3 Parallelizing sequential code

Documentation for the mesh [5] and spectral [3] archetypes provides hints on parallelizing sequential code using the mesh and spectral archetypes respectively; the same methods can in many cases be applied to mesh-spectral computations as well.

2 Overview of the archetype implementation

To support this archetype, we provide a library of high-level routines that encapsulate the details of parallelism. This library facilitates more rapid development of applications. The current implementation is based on Fortran 77 (including a few of the most widely-supported extensions) and uses either Fortran M [4, 1] or MPI [6] for parallelism. Fortran M may be obtained via anonymous FTP from Argonne National Labs; many implementations of MPI are available. To use the archetype implementation, the application developer must provide:

- A main program, typically consisting primarily of a call to `spawn`.
- Parameter definitions, as described in §2.1.
- A subroutine `procmain` that will be executed by each process. This subroutine will typically consist of sequential code much like that for a sequential version of the application, interspersed with calls to the archetype library routines described in §3 to perform those operations specific to the parallel version.

2.1 User-specified and archetype-supplied parameters

The user must provide definitions for the following Fortran `PARAMETER`s:

- `MAXGRIDS` is the maximum number of data grids to be defined.
- `MAXPROCS` is the maximum number of processes to be spawned.
- `MAXLSIZE` is the maximum size of a local section — *in* `INTEGER`*-sized units.*
- `MAXREDUCE` is the maximum size of a vector to pass to the reduction or recursive-doubling subroutines (must be at least 3).

In the current implementation, these parameters are specified by modifying file `arch_uparms.h`. Additional Fortran `PARAMETER`s are defined in file `arch_parms.h`. It is recommended that users `INCLUDE` both files in all routines.

2.2 Grids and distributions

The archetype allows the user to define multiple grids and multiple distributions. Each grid is an object (in the sense of "object-oriented design") with the following attributes:

- dimensions (fixed).
- data type (fixed).
- distribution (can vary during the computation).

A distribution in turn consists of the following attributes:

- the dimensions of the process grid (which together with the dimensions of the grid determine the dimensions of each local section).
- the widths of the ghost boundaries (0 if no ghost boundaries are needed).
- whether ghost boundaries "wrap" during a boundary exchange — whether, e.g., the left ghost boundaries of the processes on the leftmost edge of the process grid are to be filled in with data from the right boundary of the processes on the rightmost edge of the process grid.
- whether the corners of the ghost boundaries (which correspond to data from diagonally-adjacent processes) are to be updated during a boundary exchange. (For some computations — e.g., the one in §1.2.2 — these corners are not used and so need not be updated.)

For example, figure 3 illustrates a grid with dimensions (8,8,1), distributed over a process grid with dimensions (4,2,1) and ghost boundaries of size 0.

Figure 8 illustrates the "wrap" and "corners" attributes: In this figure, a 4-by-4 data grid is distributed over a 4-by-4 process grid with ghost boundaries of size 1. Shaded boxes indicate which elements of the ghost boundary are updated during a boundary exchange operation.

Associated with each grid is an integer ID, its *grid number*, which is used to specify to the archetype library routines which grid is to be operated on. Valid grid numbers range from 1 to the user-defined parameter `MAXGRIDS`. A call to `set_mesh` for grid number `N` initializes the above-described attributes; subsequent calls to library routines for grid number `N` need not repeat this information. (For example, the only parameters to `boundary_exchange` are the local section and the grid number — information about the dimensions of the local section and the sizes of the ghost boundaries is supplied by the archetype implementation from its internal representation for grid `N`.) The distribution attributes of a grid are updated when a call to `redistribute_data` is made.

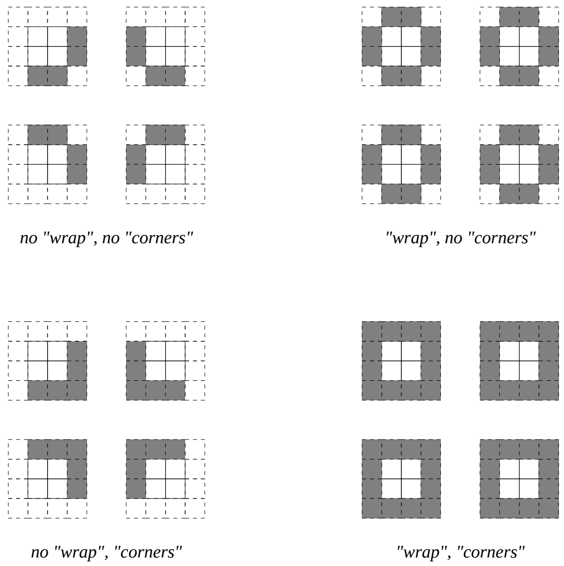

Figure 8: Effect of wrap and corners attributes.

2.3 Local data layout

Since there is no requirement that the dimensions of the process grid evenly divide the dimensions of the data grid, the actual amount of data on a particular process (referred to as the *data width*) may vary. The archetype library routines lay out data for a local section thus:

```
      dimension local_array(-igx+1:idwx+igx, -igy+1:idwy+igy,
     -                      -igz+1:idwz+igz)
```

where `igx`, `igy`, and `igz` are the sizes of the ghost boundaries, and `idwx`, `idwy`, and `idwz` are the data widths of the local section. Within this 3-dimensional array, data is arranged as follows:

- Elements `-igx+1..0` contain the left ghost boundary.
- Elements `1..idwx` contain the actual data.
- Elements `idwx+1..idwx+igx` contain the right ghost boundary.

(The other two dimensions are analogous.)

If the dimensions of the process grid evenly divide the dimensions of the grid, then all local sections have the same "shape" (dimensions of actual data), and the local sections can be declared with fixed dimensions. If not, the recommended approach is to declare local sections as 1-dimensional arrays of the maximum required size, and pass the array with its actual dimensions to a subroutine as an adjustable array argument. The maximum dimensions of a local section can be computed by dividing each dimension of the data grid by the corresponding dimension of the process grid and rounding up. Actual dimensions (data widths) can be obtained using routines `data_bounds` and `data_widths`. Many of the examples in §5 make use of this approach; in addition, the two Poisson-solver examples (§5.4 and §5.5) contrast the fixed-sizes approach and the adjustable-arrays approach.

Observe also that redistribution is in place, and so the layout of data in the local section storage for a grid depends on the current distribution of the grid. This can be handled via `EQUIVALENCE`, as in the pseudocode fragment in §1.2.3, or via adjustable arrays.

3 Library routines

The following routines are provided by the archetype. Their arguments are described using a Fortran-like syntax. Note that with the exception of the spawn routine and the utility routines, all of the following routines must be called on all spawned processes when called by one.

3.1 Initialization subroutines

3.1.1 Spawn subroutine

```
c-----------------------------------------------------------------------

      subroutine spawn(nprocs)

c     input variables:
c         integer nprocs

c-----------------------------------------------------------------------
```

This routine spawns `nprocs` processes and calls subroutine `procmain` on each. Subroutine `procmain`, provided by the user, has no arguments.

3.1.2 Grid initialization subroutine

```
c-----------------------------------------------------------------------

      subroutine set_mesh(ipdim,idim,ig,lw,corners,itype,ignum)

c     input variables:
c        integer ipdim(3), idim(3), ig(3), itype, ignum
c        logical lw(3), corners

c-----------------------------------------------------------------------
```

This routine initializes a grid with the following properties:

- `ipdim` are the dimensions of the process grid.
- `idim` are the dimensions of the data grid.
- `ig` are the sizes of the ghost boundaries (one for each dimension).
- For each dimension `i`, `lw(i)` indicates whether boundary exchanges should wrap around in this dimension, as discussed in §2.2 and illustrated by figure 8.
- `corners` indicates whether the corners of the ghost boundary must be updated during a boundary exchange, as discussed in §2.2 and illustrated by figure 8.
- `itype` is the type of the array, given by archetype-defined `PARAMETER`s `INTEGER_T`, `REAL_T`, or `DOUBLEPRECISION_T`.

Data for this grid is partitioned among the first `ipdim(1)*ipdim(2)*ipdim(3)` processes. The following constraints must be observed, else a run-time error occurs:

- `ipdim(1)*ipdim(2)*ipdim(3)` must be at most `nprocs` (the argument to `spawn`).
- `ipdim` and `idim` must be positive.
- `ipdim` must be no greater than `idim`.
- `igdim` must be nonnegative, and `idim/ipdim` must be at least `igdim`.

Variables `ipdim`, `ig`, `lw`, and `corners` are said to comprise the grid's distribution. These variables are saved by the archetype implementation for later use in archetype routines (e.g., boundary exchange or redistribution).

Note that this routine can be called from any legal state in the computation, not just at the beginning; however, it does not attempt to redistribute data, so it should be called only when the previous contents of the grid (if any) are no longer needed.

The examples in §5.3, §5.6, §5.4, and §5.5 illustrate the use of this routine, with assistance from an automatically-generated wrapper program described in §4.

3.2 Redistribution subroutine

```
c-----------------------------------------------------------------------

      subroutine redistribute_data(nipdim,nig,nlw,ncorners,
     * local_array,ignum)

c     input variables:
c         integer nipdim(3), nig(3), ignum
c         logical nlw(3), ncorners
c     input-output variables:
c         <defined-type> local_array(*)

c-----------------------------------------------------------------------
```

This subroutine redistributes the data for grid `ignum`. Data before and after the redistribution resides in the local section referenced by `local_array`, whose type is assumed to be the same as the `itype` defined for this grid when it was initialized (via `set_mesh`). The grid's previous distribution was established by a call to `set_mesh` or `redistribute_data`. Variables `nipdim`, `nig`, `nlw`, and `ncorners` specify its new distribution. These arguments are analogous to the similarly-named arguments to `set_mesh`. After the redistribution, grid `ignum` is distributed among the first `nip(1)*nip(2)*nip(3)` processes. The actual data for the grid is unchanged; all that changes is how it is distributed among processes. (Thus, a call to this routine generally results in interprocess communication.) The new distribution is saved by the archetype implementation, replacing the distribution established by a previous call to `set_mesh` or `redistribute_data`.

The examples in §5.3, §5.4, and §5.5 illustrate the use of this routine, with assistance from an automatically-generated wrapper program described in §4.

3.3 Boundary-exchange subroutine

```
c-----------------------------------------------------------------------

      subroutine boundary_exchange(local_array,ignum)

c     input variables:
c         integer local_array, ignum
c     input-output variables:
c         <defined-type> local_array(*)

c-----------------------------------------------------------------------
```

This routine performs a boundary exchange (with neighboring processes) to update the ghost boundaries of grid `ignum`. `local_array` represents the local section of the grid. Whether the boundaries of the process grid "wrap" and whether the corners of the ghost boundaries are updated is controlled by the grid's distribution (variables `lw` and `corners` respectively, set during the most recent call to `set_mesh` or `redistribute_data`). These attributes are discussed in §2.2 and illustrated by figure 8.

The examples in §5.4 and §5.5 illustrate the use of this routine.

3.4 Reduction subroutines

3.4.1 Generic reduction subroutines

A generic reduction subroutine is provided, from which users can build custom reduction operations as needed. In addition, specific reduction subroutines covering common operations are provided.

```
c-----------------------------------------------------------------------

      subroutine global_reduction(reduceproc,ni,nr,ndp,i_in,
     * r_in,dp_in,i_out,r_out,dp_out,ignum)

c     input variables:
c         external reduceproc
c         integer ni, nr, ndp, ignum
c         integer i_in(ni)
c         real r_in(nr)
c         double precision dp_in(ndp)
c     output variables:
c         integer i_out(ni)
c         real r_out(nr)
c         double precision dp_out(ndp)

c-----------------------------------------------------------------------
```

`global_reduction` performs a global reduction on input arrays `i_in`, `r_in`, and `dp_in` to yield output arrays `i_out`, `r_out`, and `dp_out`. That is, each process contributes a local value for each of the three input arrays, and these local values are combined pairwise using subroutine `reduceproc`, described below, to produce global values for the output arrays. Only those processes over which grid `ignum` is distributed are involved. If `ignum` is -1, all processes are involved.

The reduction subroutine has the following form:

```
c-----------------------------------------------------------------------

      subroutine reduceproc(ni,nr,ndp,i_in1,i_in2,r_in1,r_in2,
     $    dp_in1,dp_in2,i_out,r_out,dp_out)

c     input variables:
c         integer ni, nr, ndp
c         integer i_in1(ni), i_in2(ni)
c         real r_in1(nr), r_in2(nr)
c         double precision dp_in1(ndp), dp_in2(ndp)
c     output variables:
c         integer i_out(ni)
c         real r_out(nr)
c         double precision dp_out(ndp)

c-----------------------------------------------------------------------
```

This routine combines (`i_in1`, `r_in1`, `dp_in1`) and (`i_in2`, `r_in2`, `dp_in2`) to produce (`i_out`, `r_out`, `dp_out`). The order in which the arguments are combined is not specified, so if the operation defined by this routine is not commutative and associative (as is the case, for example, with floating-point addition), users should be careful that results do not depend on a particular order of combination.

The example in §5.7 illustrates the use of the very similar generic recursive-doubling routine.

3.4.2 Specific reduction subroutines

```
c-----------------------------------------------------------------------

      subroutine global_max_dp(dpvin,lin,dpvout,lout,ignum)
      subroutine global_max_r(rvin,lin,rvout,lout,ignum)
      subroutine global_max_i(ivin,lin,ivout,lout,ignum)

      subroutine global_min_dp(dpvin,lin,dpvout,lout,ignum)
      subroutine global_min_r(rvin,lin,rvout,lout,ignum)
      subroutine global_min_i(ivin,lin,ivout,lout,ignum)

      subroutine global_sum_dp(dpvin,dpvout,ignum)
      subroutine global_sum_r(rvin,rvout,ignum)
      subroutine global_sum_i(ivin,ivout,ignum)

      subroutine global_product_dp(dpvin,dpvout,ignum)
      subroutine global_product_r(rvin,rvout,ignum)
      subroutine global_product_i(ivin,ivout,ignum)

c     input variables:
c         integer lin(3), ignum
c         double precision dpvin
c         real rvin
c         integer ivin
c     output variables:
c         integer lout(3)
c         double precision dpvout
c         real rvout
c         integer ivout

c-----------------------------------------------------------------------
```

These routines perform the reductions suggested by their names. `global_max_dp` takes as input a local double-precision maximum `dpvin` and its indices `lin` and produces as output a global double-precision maximum `dpvout` and its indices `lout`. The reduction operates only over those processes over which grid `ignum` is distributed (over all processes if `ignum` is -1). The other procedures are analogous.

The examples in §5.4 and §5.5 illustrate the use of these routines.

3.4.3 Generic recursive-doubling subroutine

Recursive doubling, described in detail in [8], is a form of reduction in which the reduction operation is not necessarily commutative. The mesh-spectral archetype provides a generic recursive-doubling subroutine, from which users can build custom recursive-doubling operations as needed.

```
c-----------------------------------------------------------------------

      subroutine recursive_doubling(doubleproc,ni,nr,ndp,i_in,
     * r_in,dp_in,i_out,r_out,dp_out,ignum)

c     input variables:
c         external doubleproc
c         integer ni, nr, ndp, ignum
c         integer i_in(ni)
c         real r_in(nr)
c         double precision dp_in(ndp)
c     output variables:
c         integer i_out(ni)
c         real r_out(nr)
c         double precision dp_out(ndp)

c-----------------------------------------------------------------------
```

`recursive_doubling` performs recursive doubling on input arrays `i_in`, `r_in`, and `dp_in` to yield output arrays `i_out`, `r_out`, and `dp_out`. That is, each process contributes a local value for each of the three input arrays, and these local values are combined using recursive doubling with subroutine `doubleproc`, described below, to produce values for the output arrays. Only those processes over which grid `ignum` is distributed are involved. If `ignum` is -1, all processes are involved. The number of processes involved must be a power of 2.

The recursive-doubling subroutine has the following form:

```
c-----------------------------------------------------------------------

      subroutine doubleproc(ni,nr,ndp,i_in1,i_in2,r_in1,r_in2,
     $    dp_in1,dp_in2,i_out,r_out,dp_out)

c     input variables:
c         integer ni, nr, ndp
c         integer i_in1(ni), i_in2(ni)
c         real r_in1(nr), r_in2(nr)
c         double precision dp_in1(ndp), dp_in2(ndp)
c     output variables:
c         integer i_out(ni)
c         real r_out(nr)
c         double precision dp_out(ndp)

c-----------------------------------------------------------------------
```

This routine combines (i_in1, r_in1, dp_in1) and (i_in2, r_in2, dp_in2) to produce (i_out, r_out, dp_out).

The example in §5.7 illustrates the use of this routine.

3.5 Input/output subroutines

As described in §1.2.4, several forms of I/O are possible with the archetype, but specific support is provided only for "block sequential" I/O via the following routines.

```
c-----------------------------------------------------------------------

      subroutine write_mesh(writeproc,ignum,iunit,local_array)

c     input variables:
c         external writeproc
c         integer ignum,iunit
c         <defined-type> local_array(*)

      subroutine read_mesh(readproc,ignum,iunit,local_array)

c     input variables:
c         external readproc
c         integer ignum,iunit
c     output variables:
c         <defined-type> local_array(*)

c-----------------------------------------------------------------------
```

Subroutine `write_mesh` has the effect of using `writeproc` to write data from grid `ignum` to a sequential file, identified by Fortran unit number `iunit`. Array `local_array` contains the local section of the grid. Only one process (identified by archetype-defined `PARAMETER IOPROC`) actually performs writes; the other processes send their local sections to this process, which writes them out. Thus, only the process whose process number (as returned by subroutine `local_pos`, described in §3.7) is `IOPROC` should open the output file. The user-supplied subroutine `writeproc` controls exactly which data is written out and in what format; it has the following form:

```
c-----------------------------------------------------------------------

      subroutine writeproc(ignum,iunit,ipx,ipy,ipz,
     * ixlo,ixhi,iylo,iyhi,izlo,izhi,nxdw,nydw,nzdw,local_array)

c     input variables:
c         integer ignum,iunit
c         integer ipx,ipy,ipz
c         integer ixlo,ixhi,iylo,iyhi,izlo,izhi
c         integer nxdw,nydw,nzdw
c     output variables:
c         <defined-type> local_array(ixlo:ixhi,iylo:iyhi,izlo:izhi)

c-----------------------------------------------------------------------
```

This subroutine writes data from `local_array` onto Fortran unit number `iunit`. The grid number, `ignum`, and the process-grid indices, `(ipx, ipy, ipz)`, are provided so that `writeproc` can, if desired, write out only selected portions of the grid. The dimensions of the local section are given by arguments `ixlo`, `ixhi`, `iylo`, `iyhi`, and `izlo`, `izhi`; its data widths are given by `nxdw`, `nydw`, and `nzdw`, both as described in §2.3. Observe that each local section is written out contiguously; thus, in a grid distributed by (cubical) blocks, output data consists of data from block (1,1,1), then from (2,1,1), etc.

Subroutine `read_mesh` is analogous. Again, only one process actually performs reads, so only that process (identified by the process number `IOPROC`) should open the input file. The user-specified procedure for `read_mesh` has the following form:

```
c-----------------------------------------------------------------------

      subroutine readproc(ignum,iunit,ipx,ipy,ipz,
     * ixlo,ixhi,iylo,iyhi,izlo,izhi,nxdw,nydw,nzdw,local_array)

c     input variables:
c         integer ignum,iunit
c         integer ipx,ipy,ipz
c         integer ixlo,ixhi,iylo,iyhi,izlo,izhi
c         integer nxdw,nydw,nzdw
c         <defined-type> local_array(ixlo:ixhi,iylo:iyhi,izlo:izhi)

c-----------------------------------------------------------------------
```

Observe that data for each local section should be contiguous in the input file; thus, input data should consist of all data for block (1,1,1), then for (2,1,1), etc.

The example in §5.3 illustrates the use of these routines. In addition, the examples in §5.4 and §5.5 illustrate sequential I/O, and the example in §5.2 illustrates the use of system-independent parallel I/O, both as described in §1.2.4.

3.6 Broadcast subroutine

A single combined broadcast subroutine is provided. One intended use of this subroutine is to allow a single process to read, e.g., constants from an input file and broadcast them to all other processes.

```
c-----------------------------------------------------------------------

      subroutine broadcast(ni,nr,ndp,iv,rv,dpv,ignum)

c     input variables:
c         integer ignum, ni, nr, ndp
c     input/output variables:
c         integer iv(ni)
c         real rv(nr)
c         double precision dpv(ndp)

c-----------------------------------------------------------------------
```

`broadcast` operates as follows: Arrays `iv`, `rv`, and `dpv` are copied from the process identified by process number `IOPROC` to the remaining processes. The corresponding sizes `ni`, `nr`, and `ndp` are input variables in all processes; the arrays themselves are input variables in process `IOPROC` and output variables in the other processes. Only those processes over which grid `ignum` is distributed are involved. If `ignum` is -1, all processes are involved.

The examples in §5.4 and §5.5 illustrate the use of this routine.

3.7 Utility subroutines

A variety of utility subroutines and functions are provided; see the example programs in §5 for typical uses.

```
c-----------------------------------------------------------------------

      subroutine pack3(i,j,k,iarray)

c     input variables:
c         integer i,j,k
c     output variables:
c         integer iarray(3)

      subroutine unpack3(i,j,k,iarray)

c     input variables:
c         integer iarray(3)
c     output variables:
c         integer i,j,k

      subroutine pack3l(li,lj,lk,larray)

c     input variables:
c         logical li,lj,lk
c     output variables:
c         logical larray(3)

      subroutine unpack3l(li,lj,lk,larray)

c     input variables:
c         logical larray(3)
c     output variables:
c         logical li,lj,lk

c-----------------------------------------------------------------------
```

Subroutine `pack3` packs three integers into an array; subroutine `unpack3` unpacks an array into three integers. Subroutines `pack3l` and `unpack3l` are analogous.

```
c-----------------------------------------------------------------------

      subroutine local_pos(ipxyz,procnum,ignum)

c     input variables:
c         integer ignum
c     output variables:
c         integer ipxyz(3), procnum

c-----------------------------------------------------------------------
```

This routine returns, for the process in which it is called, its coordinates `ipxyz` in the process grid for grid `ignum` and its process number `procnum`. If this process is not used in the current distribution of grid `ignum`, it returns (-1, -1, -1) for `ipxyz`. The process number `procnum` is independent of the current distribution of the grid. `ignum` is assumed to be a valid grid number or -1; in the latter case, (-1, -1, -1) is returned in `ipxyz`.

```
c-----------------------------------------------------------------------

      integer function numprocs(ignum)

c     input variables:
c         integer ignum

c-----------------------------------------------------------------------
```

This routine returns the number of processes over which grid `ignum` is distributed, or the total number of processes spawned if `ignum` is -1. `ignum` is assumed to be a valid grid number or -1.

```
c-----------------------------------------------------------------------

      subroutine proc_to_xyz(procnum,ipxyz,ignum)

c     input variables:
c         integer procnum, ignum
c     output variables:
c         integer ipxyz(3)

c-----------------------------------------------------------------------
```

This routine returns, for the process specified by `procnum`, its coordinates `ipxyz` in the process grid for grid `ignum`. If this process is not used in the current distribution of grid `ignum`, it returns (-1, -1, -1) for `ipxyz`. `ignum` is assumed to be a valid grid number or -1; in the latter case, (-1, -1, -1) is returned in `ipxyz`.

```
c-----------------------------------------------------------------------

      subroutine xyz_to_proc(ipxyz,procnum,ignum)

c     input variables:
c         integer ipxyz(3), ignum
c     output variables:
c         integer procnum

c-----------------------------------------------------------------------
```

This routine returns the process number procnum for the process whose coordinates in the process grid for grid ignum are ipxyz, or (-1, -1, -1) if the coordinates are invalid. ignum is assumed to be a valid grid number or -1; in the latter case, (-1, -1, -1) is returned in ipxyz.

```
c-----------------------------------------------------------------------

      subroutine global_to_local(ijk,ipxyz,ijkloc,ignum)

c     input variables:
c         integer ijk(3), ignum
c     output variables:
c         integer ipxyz(3), ijkloc(3)

c-----------------------------------------------------------------------
```

Subroutine global_to_local takes global indices ijk for grid ignum and returns the corresponding local indices (ipxyz, ijkloc), where ipxyz are the process-grid coordinates and ijkloc are indices into the local section. Both output arguments are set to (-1, -1, -1) if the input argument is invalid (outside the dimensions of the grid). ignum is assumed to be a valid grid number.

```
c-----------------------------------------------------------------------

      subroutine local_to_global(ipxyz,ijkloc,ijk,ignum)

c     input variables:
c         integer ipxyz(3), ijkloc(3), ignum
c     output variables:
c         integer ijk(3)

c-----------------------------------------------------------------------
```

Subroutine local_to_global takes process-grid indices ipxyz and local-section indices ijkloc for grid ignum and returns the corresponding global indices ijk, or (-1, -1, -1) if the input arguments are invalid. ignum is assumed to be a valid grid number.

```
c-----------------------------------------------------------------------

      subroutine intersect(ipxyz,iglb1,iglb2,iloc1,iloc2,empty,ignum)

c     input variables:
c         integer ipxyz(3)
c         integer iglb1(3), iglb2(3), ignum
c     output variables:
c         integer iloc1(3), iloc2(3)
c         logical empty

c-----------------------------------------------------------------------
```

Subroutine `intersect` computes, for grid `ignum`, the intersection of a range of global indices (`iglb1:iglb2`) with the indices of the local section for process `ipxyz`. `empty` indicates whether this intersection is empty; if it is `.FALSE.`, `iloc1:iloc2` is the range of local indices corresponding to the intersection. (If `empty` is `.TRUE.`, `iloc1` and `iloc2` are undefined.) `ignum` and `ipxyz` are assumed to be valid.

This routine is intended for use in computing loop bounds, particularly in situation in which it may not be desirable to loop over the whole local section. The examples in §5.4 and §5.5 illustrate the use of this routine.

```
c-----------------------------------------------------------------------

      subroutine data_bounds(ipxyz,lowxyz,highxyz,ignum)

c     input variables:
c         integer ipxyz(3)
c         integer ignum
c     output variables:
c         integer lowxyz(3), highxyz(3)

c-----------------------------------------------------------------------
```

This routine returns the bounds of the data for grid `ignum` actually contained on the local process given by `ipxyz`, including ghost boundaries, as described in §2.3. `ignum` and `ipxyz` are assumed to be valid.

These bounds are appropriate for use in passing local sections as adjustable array arguments, as illustrated in the examples in §5.4 and §5.6.

```
c-----------------------------------------------------------------------

      subroutine data_widths(ipxyz,bdry,isizexyz,ignum)

c     input variables:
c         integer ipxyz(3)
c         logical bdry
c         integer ignum
c     output variables:
c         integer isizexyz(3)

c-----------------------------------------------------------------------
```

This routine returns the widths of the data for grid `ignum` actually contained on the local process given by `ipxyz`, as described in §2.3. Ghost boundaries are included if and only if `bdry` is `.TRUE.`. `ignum` and `ipxyz` are assumed to be valid.

The examples in §5.3 and §5.6 illustrate some uses of this routine.

```
c-----------------------------------------------------------------------

      integer function max_lsize(ignum)

c     input variables:
c         integer ignum

c-----------------------------------------------------------------------
```

This routine returns the maximum size of a local section for grid `ignum`, including ghost boundaries. `ignum` is assumed to be valid.

A suggested use of this routine is to verify that an array to be used as a local section is indeed big enough.

```
c-----------------------------------------------------------------------

      subroutine gridinfo(ipdims,igdims,ighost,lwrap,lcorners,
     * igtype,ignum)

c     input variables:
c         integer ignum
c     output variables:
c         integer ipdims(3), igdims(3), ighost(3), igtype
c         logical lwrap(3), lcorners

c-----------------------------------------------------------------------
```

This routine returns all information about the current distribution for grid `ignum` — dimensions of the process grid (`ipdims`), dimensions of the data grid (`igdims`), sizes of ghost boundaries (`ighost`), whether boundaries are to wrap during a boundary exchange (`lwrap`), whether corners are to be included in a boundary exchange (`lcorners`), and data type (`igtype`). `ignum` is assumed to be valid.

```
c-----------------------------------------------------------------------

      subroutine uniqname(ipxyz,inname,outname)

c     input variables:
c         integer ipxyz(3)
c         character*(*) inname
c     output variables:
c         character*(*) outname

c-----------------------------------------------------------------------
```

This routine is intended to be used to generate a unique filename for each process in a process grid. It appends the character representation of `ipxyz` (intended to be process-grid coordinates) to `inname` to produce `outname`. `ipxyz` is assumed to be valid.

```
c-----------------------------------------------------------------------

      subroutine uniqnamep(iproc,inname,outname)

c     input variables:
c         integer iproc
c         character*(*) inname
c     output variables:
c         character*(*) outname

c-----------------------------------------------------------------------
```

This routine is intended to be used to generate a unique filename for each process, independent of process grid. It appends the character representation of `iproc` (intended to be a process number) to `inname` to produce `outname`. `iproc` is assumed to be valid.

The example in §5.2 illustrates a use of this routine.

4 Utility program to generate parameter declarations

4.1 Motivation

It is frequently useful in developing Fortran application programs to declare constant values via `PARAMETER` declarations, often in a header file to be `INCLUDE`'d in each routine. For mesh-spectral archetype applications, these include:

- Grid numbers (for which a `PARAMETER` declaration provides a symbolic name).
- Grid dimensions.
- Process grid dimensions.
- Maximum dimensions of local sections — which can be calculated in the `PARAMETER` declarations from grid and process grid dimensions, thereby reducing the chance of error.

If there are many grids and/or distributions, it can be tedious to prepare such a header file by hand. Also, it can be tedious to package these constants into the arrays of size 3 required as arguments to `set_mesh` and `redistribute_data`. (The rationale for using arrays of dimensions rather than individual dimensions as arguments to these two routines was to keep the length of the argument list to a manageable size, at the cost of perhaps requiring extra processing to package the various dimensions into arrays of size 3.)

Therefore, the archetype implementation includes a standalone utility program, called `header.exe`, that takes as input a set of definitions of grids and distributions and produces:

- A header file containing `PARAMETER` declarations for grid and distribution attributes. Names of `PARAMETER`s are formed from user-supplied identifiers for grids and distributions.
- A file containing a wrapper program for `set_mesh` and `redistribute_data`; this wrapper program takes advantage of the generated `PARAMETER` declarations and thus requires fewer and simpler arguments than `set_mesh` and `redistribute_data`.

4.2 Program input and output

`header.exe` prompts the user for the following inputs:

- Identifiers for the 3 dimensions (typically `x`, `y`, and `z`).
- Number of grids.

- For each grid:
  - Short character identifier (to be used in generating PARAMETER names).
  - Dimensions.
- Number of distributions.
- For each distribution (process grid):
  - Short character identifier (to be used in generating PARAMETER names).
  - Dimensions of process grid.
  - Widths of ghost boundaries.
  - Whether the boundaries wrap in a boundary exchange.
  - Whether corners are updated in a boundary exchange.
- Whether to generate PARAMETER declarations for local section dimension bounds. (This is only useful when the dimensions of the process grid divide the dimensions of the grid evenly; see the example in §5.5 for how these PARAMETERs are used.)
- Names for the two files created (see below).
- Additional lines to be included in the PARAMETER-declarations (header) file.

It generates two files, with names as supplied by the user, one containing the PARAMETER declarations and one containing the wrapper program to be used for calling `set_mesh` and `redistribute_data`. The examples in §5.3, §5.4, §5.5, and §5.6 present sample executions of `header.exe` and the automatically-generated files.

It is anticipated that users may prefer not to rerun `header.exe` interactively every time they want to make a small change in a grid's or distribution's attributes. Since the program is set up to simply read from standard input, it can readily be executed in batch mode with input coming from a file. Users may capture interactive input for later reuse in batch mode in several ways (e.g., using Unix commands `script` or `tee`), or they may use the following procedure:

1. Copy and edit the provided template input file `template.header_in`, which contains condensed versions of program prompts and example input.
2. Use provided script file `makeheader.sh` to strip out the prompts and use the result as input to `header.exe`.

5 Example programs

This section presents several tutorial example programs that illustrate how to use the various features of the mesh-spectral archetype. The following observations apply to most of the examples:

- The utility program `header.exe`, described in §4, is used to generate `INCLUDE` files containing `PARAMETER` declarations, and these declarations are used to specify sizes of grids, process grids, and local sections.
- As described in §2.3, local sections are frequently declared as 1-dimensional arrays and then passed to subroutines as adjustable array arguments with their actual dimensions.
- Examples are presented in their Fortran M versions; MPI versions are for the most part identical except that they do not include a `processors` declaration in the main program.

5.1 "Hello, world"

5.1.1 Program specification.

In this program, each process simply writes a "hello, world" message with its process number to standard output.

5.1.2 Program code (file `hello.FM`)

Observe the following:

- The main program contains a Fortran M statement (`processors (MAXPROCS)` indicating the maximum number of processes to be started. This is required for archetype-based applications using the Fortran M implementation.
- The executable code consists of a main program that calls `spawn` and a subroutine `procmain` to execute in each spawned process, as described in §2.
- No distributed grids are used in this application.
- Each process obtains its process number through a call to `local_pos`.

```
c---------------------------------------------------------------------
c---------------------------------------------------------------------
c
c     example of mesh/spectral archetype
c
c     'hello, world'
```

```
c
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c     main program
c
c-----------------------------------------------------------------------

      program main
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      integer procs
      parameter (procs=4)

      processors (MAXPROCS)
      call spawn(procs)
      end

c-----------------------------------------------------------------------
c
c     process-main program
c
c-----------------------------------------------------------------------

      subroutine procmain()
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'

      integer procxyz(3), procnum

      call local_pos(procxyz,procnum,-1)
      print*, 'hello, world, from process ', procnum

      end

c-----------------------------------------------------------------------
```

5.2 "Hello, world" (variant)

5.2.1 Program specification.

In this program, each process opens a separate file and writes to it a "hello, world" message with its process number. This is an example of the "system-independent parallel I/O" described in §1.2.4.

5.2.2 Program code (file hello_files.FM)

Observe the following:

- This program is identical to the previous example, except for the use of file output rather than standard (terminal-directed) output.
- Each process generates a uniquely-named file by calling `uniqnamep` and writes output to that file.

```
c----------------------------------------------------------------------
c----------------------------------------------------------------------
c
c     example of mesh/spectral archetype
c
c     'hello, world' to files
c
c----------------------------------------------------------------------
c----------------------------------------------------------------------

c----------------------------------------------------------------------
c
c     main program
c
c----------------------------------------------------------------------

      program main
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      integer procs
      parameter (procs=4)

      processors (MAXPROCS)
      call spawn(procs)
      end

c----------------------------------------------------------------------
c
c     process-main program
c
c----------------------------------------------------------------------

      subroutine procmain()
      implicit none
      include 'arch_uparms.h'
```

```
      include 'arch_parms.h'

      integer IUNIT
      parameter (IUNIT=50)
      integer procxyz(3), procnum
      character*80 outfile

      call local_pos(procxyz,procnum,-1)
      call uniqnamep(procnum,'hello_files.OUT_',outfile)
      open (IUNIT, file=outfile, status='unknown', access='sequential',
     -    form='formatted')
      write (IUNIT, *) 'hello, world, from process ', procnum
      close (IUNIT)

      end

c----------------------------------------------------------------------
```

5.3 2-dimensional FFT

5.3.1 Program specification.

This program performs a 2-dimensional FFT in place, as described in [7] and many other references. Performing the 2-dimensional FFT is accomplished by performing a 1-dimensional FFT on each row of a 2-dimensional array and then performing a 1-dimensional FFT on each column of the resulting 2-dimensional array.

5.3.2 Program code (file `fft.FM`)

Observe the following:

- The actual FFTs are performed using a sequential subroutine library (written by Clive Temperton, originally for vector machines but portable to other architectures as well) that allows the computation of FFTs on a set of vectors with a single subroutine call. Comments in `fft.FM` describe the calling sequence for the FFT library subroutines; code for the library itself is not shown.

- In order to perform the FFT computations using this sequential subroutine library, the program employs two distributions for the 2-dimensional array on which the FFT is to be performed: a distribution by rows (which, for each row, puts all data in a single process, allowing it to be processed with a sequential FFT subroutine), and a distribution by columns (which has the same effect on columns). Data is initially distributed by rows; after the FFT-by-rows is performed, it is redistributed by columns and the FFT-by-columns is performed.

- The input/output data for the FFT is a 2-dimensional array of single-precision complex numbers. The mesh-spectral archetype does not directly support COMPLEX data types, but since single-precision complex numbers are the same size as double-precision reals, it is possible to declare the required grid as double precision.

- Utility program header.exe, described in §4, is used to generate two files:

  - fft_header.h, containing PARAMETER declarations for the grid and the two distributions.
  - fft_wrap.h, containing a wrapper program for archetype routines set_mesh and redistribute_data.

  The PARAMETERs defined in fft_header.h are used to declare program variables trigsnx, trigsny, and lclsize_dg1. The wrapper program in fft_wrap.h, wrapper, is used to invoke both set_mesh and redistribute_data. §5.3.3 describes inputs to header.exe in more detail, and §5.3.4 and §5.3.5 show its output.

- I/O is performed using archetype routines read_mesh and write_mesh. These subroutines perform "blocked sequential" I/O as described in §1.2.4; all data is read from or written to a single file, with the archetype routines handling the required communication, and the actual reading or writing of data performed by user-supplied routines readproc and writeproc. Observe that since the actual I/O is performed using these user-supplied routines, the format of the data is under the user's control — this program uses unformatted (binary) I/O to write out the whole local section in row-major order, while a different program might choose a different format or a different order.

```
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------
c
c     example of mesh/spectral archetype
c
c     2D FFT
c
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c     main program
c
c-----------------------------------------------------------------------
```

```
      program main
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'fft_header.h'

      processors (MAXPROCS)
      call spawn(procs)
      end

c-----------------------------------------------------------------------
c
c     process-main program
c
c-----------------------------------------------------------------------

      subroutine procmain()
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'fft_header.h'

      external readproc, writeproc

      integer iunit, ounit, procnum, procxyz(3), datasize(3)
c
c     arguments for cfft99:
c         cfft99(a,work,trigs,ifax,inc,jump,n,lot,isign)
c
c         a contains input/output of FFT (a set of vectors)
c         work is a work array
c         trigs, ifax contain constants for cfft99 (generated by
c             subroutine cftfax)
c         inc is stride between vector elements
c         jump is between successive vectors
c         n is size of each vector
c         lot is how many vectors
c         isign indicates forward/inverse transform
c
      integer ifaxnx(13), ifaxny(13)
      real trigsnx(2*nx_dg1), trigsny(2*ny_dg1)
      integer inc, jump, n, lot

c     local section of distributed grid -- type is really complex,
c         declared double precision for compatibility with archetype
      double precision arr(lclsize_dg1)
      complex work(lclsize_dg1)
```

```
      iunit=11
      ounit=12

c
c     set up grid and read input data from 'fft.IN'
c
      call local_pos(procxyz,procnum,-1)
      call wrapper(id_dg1,id_row,DOUBLEPRECISION_T,arr,SETMESH)
      if (procnum.eq.IOPROC) then
          open(unit=iunit, file='fft.IN', form='unformatted')
          open(unit=ounit, file='fft.OUT', form='unformatted')
      endif
      call read_mesh(readproc, id_dg1, iunit, arr)

c
c     initialize for fft
c
      call cftfax(nx_dg1, ifaxnx, trigsnx)
      call cftfax(ny_dg1, ifaxny, trigsny)

c
c     do fft by rows
c
      inc = nxlcl_dg1_row
      jump = 1
      n = ny_dg1
      call local_pos(procxyz,procnum,id_dg1)
      call data_widths(procxyz,.false.,datasize,id_dg1)
      lot = datasize(1)
      call cfft99(arr, work, trigsny, ifaxny, inc, jump, n, lot, -1)

c
c     redistribute data
c
      call wrapper(id_dg1,id_col,DOUBLEPRECISION_T,arr,REDISTRIBUTEDATA)

c
c     do fft by columns
c
      inc = 1
      jump = nx_dg1
      n = nx_dg1
      call local_pos(procxyz,procnum,id_dg1)
      call data_widths(procxyz,.false.,datasize,id_dg1)
      lot = datasize(2)
      call cfft99(arr, work, trigsnx, ifaxnx, inc, jump, n, lot, -1)
```

```
c
c     redistribute data
c
      call wrapper(id_dg1,id_row,DOUBLEPRECISION_T,arr,REDISTRIBUTEDATA)

c
c     write output data to 'fft.OUT'
c
      call write_mesh(writeproc, id_dg1, ounit, arr)

      if (procnum.eq.IOPROC) then
          close(iunit)
          close(ounit)
      endif
      return
      end

c-----------------------------------------------------------------------
c
c     subroutine for reading input data (one local section at
c         a time)
c
c-----------------------------------------------------------------------

      subroutine readproc(ignum, iunit, ipx, ipy, ipz, ixlo, ixhi,
     -     iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, rarr)
      implicit none

      integer ignum, iunit, x, y,z
      integer ipx, ipy, ipz
      integer ixlo, ixhi, iylo, iyhi, izlo, izhi
      integer nxlcl, nylcl, nzlcl

      complex rarr(ixlo:ixhi, iylo:iyhi, izlo:izhi)

      do x=1,nxlcl
         do y=1,nylcl
            do z=1,nzlcl
               read(iunit, end=10) rarr(x, y, z)
            enddo
         enddo
      enddo
 10   return
      end

c-----------------------------------------------------------------------
```

```
c
c     subroutine for writing input data (one local section at
c         a time)
c
c-----------------------------------------------------------------------

      subroutine writeproc(ignum, ounit, ipx, ipy, ipz, ixlo, ixhi,
     -     iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, warr)
      implicit none

      integer ignum, ounit
      integer ipx, ipy, ipz, x, y, z
      integer ixlo, ixhi, iylo, iyhi, izlo, izhi, izlo
      integer nxlcl, nylcl, nzlcl

      complex warr(ixlo:ixhi, iylo:iyhi, izlo:izhi)

      do x=1,nxlcl
         do y=1,nylcl
            do z=1,nzlcl
               write(ounit) warr(x, y, z)
            enddo
         enddo
      enddo
      return
      end

c-----------------------------------------------------------------------
c
c     automatically-generated wrapper program to pack arrays
c         and call set_mesh or redistribute_data
c
c-----------------------------------------------------------------------

      include 'fft_wrap.h'

c-----------------------------------------------------------------------
```

5.3.3 Generating declarations and wrapper (file `fft.header_in`)

The following input to `makeheader.sh` is used to generate header and wrapper-program files for this example. Observe the following:

- This application requires one data grid and two distributions, one by rows and one by columns. Input to `header.exe` includes a "name" for each grid and distribution, which is used to generate `PARAMETER`s.

```
>>What are dimension 1, 2, 3 called?
```

```
x
y
z
>>How many data grids are there?
1
>>For each data grid:
>>  What is its name?
dg1
>>  What are its dimensions?
4
4
1
>>End of 'for each data grid'
>>How many processes are there?
2
>>How many different distributions are there?
2
>>For each distribution:
>>  What is its name?
col
>>  What are the dimensions of the process grid?
1
2
1
>>  For each of the 3 dimensions, what is the width of the ghost boundary?
0
0
0
>>  For each of the 3 dimensions, does it wrap?  (y/n)
n
n
n
>>  Do you need to include corners in boundary exchanges? (y/n)
n
>>  What is its name?
row
>>  What are the dimensions of the process grid?
2
1
1
>>  For each of the 3 dimensions, what is the width of the ghost boundary?
0
0
0
>>  For each of the 3 dimensions, does it wrap?  (y/n)
n
n
```

```
n
>>  Do you need to include corners in boundary exchanges? (y/n)
n
>>End of 'for each distribution'
>>Do you want parameters for dimension bounds for local sections? (y/n)
>>(These are only useful when process grid dimensions divide
>>    data grid dimensions evenly.)
n
>>What is the name of the header file to be created?
fft_header.h
>>Enter additional lines for the header file; '.' to stop:
.
>>What is the name of the subroutine file to make?
fft_wrap.h
```

5.3.4 Generated declarations (file `fft_header.h`)

Observe the following:

- `PARAMETER`s that pertain to the data grid, independent of distribution, are identified by the suffix the user gave as its "name", `dg1`, including its grid number, `id_dg1`. One of these `PARAMETER`s, `lclsize_dg1`, computes the maximum size of a local section of grid `dg1` under all distributions.
- `PARAMETER`s that pertain to a distribution, independent of the grid it is applied to, are identified by the suffix the user gave as its "name", `col` or `row`.
- `PARAMETER`s that pertain to a combination of grid and distribution are identified by both suffixes, e.g., the maximum local section size for grid `dg1` under distribution `col` is `lclsize_dg1_col`.

```
c----------------------------------------------------------------------
c
c     parameter declarations
c
c----------------------------------------------------------------------

c----------------------------------------------------------------------
c
c     constants (always present)
c
      logical SETMESH, REDISTRIBUTEDATA
      parameter (SETMESH=.TRUE.)
      parameter (REDISTRIBUTEDATA=.FALSE.)

c----------------------------------------------------------------------
c
```

```
c     grid IDs, dimensions, number of processes (supplied by you)
c
      integer procs
      parameter (procs = 2)

      integer id_dg1
      parameter (id_dg1 = 1)
      integer nx_dg1
      parameter (nx_dg1 = 4)
      integer ny_dg1
      parameter (ny_dg1 = 4)
      integer nz_dg1
      parameter (nz_dg1 = 1)

c-----------------------------------------------------------------------
c
c     col distribution:
c
c     distribution ID, parameters for col
c     (supplied by you)
c
      integer id_col
      parameter (id_col = 1)
      integer nprocx_col
      parameter (nprocx_col = 1)
      integer nprocy_col
      parameter (nprocy_col = 2)
      integer nprocz_col
      parameter (nprocz_col = 1)
      integer nghostx_col
      parameter (nghostx_col = 0)
      integer nghosty_col
      parameter (nghosty_col = 0)
      integer nghostz_col
      parameter (nghostz_col = 0)
      logical lwrapx_col
      parameter (lwrapx_col = .FALSE.)
      logical lwrapy_col
      parameter (lwrapy_col = .FALSE.)
      logical lwrapz_col
      parameter (lwrapz_col = .FALSE.)
      logical lcorner_col
      parameter (lcorner_col = .FALSE.)
c
c     maximum local section dimensions for col, without ghost boundary
c     (computed)
c
```

```
      integer nxlcl_dg1_col
      parameter (nxlcl_dg1_col =
     -      (nx_dg1 + nprocx_col -1) / nprocx_col)
      integer nylcl_dg1_col
      parameter (nylcl_dg1_col =
     -      (ny_dg1 + nprocy_col -1) / nprocy_col)
      integer nzlcl_dg1_col
      parameter (nzlcl_dg1_col =
     -      (nz_dg1 + nprocz_col -1) / nprocz_col)
c
c     1d sizes of local sections for col
c     (computed)
c
      integer lclsize_dg1_col
      parameter (lclsize_dg1_col =
     -      (nxlcl_dg1_col + 2*nghostx_col) *
     -      (nylcl_dg1_col + 2*nghosty_col) *
     -      (nzlcl_dg1_col + 2*nghostz_col))

c----------------------------------------------------------------------
c
c     row distribution:
c
c     distribution ID, parameters for row
c     (supplied by you)
c
      integer id_row
      parameter (id_row = 2)
      integer nprocx_row
      parameter (nprocx_row = 2)
      integer nprocy_row
      parameter (nprocy_row = 1)
      integer nprocz_row
      parameter (nprocz_row = 1)
      integer nghostx_row
      parameter (nghostx_row = 0)
      integer nghosty_row
      parameter (nghosty_row = 0)
      integer nghostz_row
      parameter (nghostz_row = 0)
      logical lwrapx_row
      parameter (lwrapx_row = .FALSE.)
      logical lwrapy_row
      parameter (lwrapy_row = .FALSE.)
      logical lwrapz_row
      parameter (lwrapz_row = .FALSE.)
      logical lcorner_row
```

```
      parameter (lcorner_row = .FALSE.)
c
c     maximum local section dimensions for row, without ghost boundary
c     (computed)
c
      integer nxlcl_dg1_row
      parameter (nxlcl_dg1_row =
     -      (nx_dg1 + nprocx_row -1) / nprocx_row)
      integer nylcl_dg1_row
      parameter (nylcl_dg1_row =
     -      (ny_dg1 + nprocy_row -1) / nprocy_row)
      integer nzlcl_dg1_row
      parameter (nzlcl_dg1_row =
     -      (nz_dg1 + nprocz_row -1) / nprocz_row)
c
c     1d sizes of local sections for row
c     (computed)
c
      integer lclsize_dg1_row
      parameter (lclsize_dg1_row =
     -      (nxlcl_dg1_row + 2*nghostx_row) *
     -      (nylcl_dg1_row + 2*nghosty_row) *
     -      (nzlcl_dg1_row + 2*nghostz_row))

c---------------------------------------------------------------------
c
c     maximum and combined sizes
c     (computed)
c
c     maximum computed using the formula:
c     max(i,j) = (j*(j/i)+i*(i/j)) / ((i/j)+(j/i))
c

c
c     maximum size of local sections for dg1
c
      integer i1
      parameter (i1 = lclsize_dg1_col)
      integer i2
      parameter (i2 = lclsize_dg1_row)
      integer i3
      parameter (i3 =
     -      (i2*(i2/i1) + i1*(i1/i2)) /
     -      ((i1/i2) + (i2/i1)) )
      integer lclsize_dg1
      parameter (lclsize_dg1 = i3)
c
```

```
c     1d size of combined local sections
c     (computed)
c
      integer lclsize
      parameter (lclsize =
     -      lclsize_dg1 )

c-----------------------------------------------------------------------
c
c     additional declarations (supplied by you)
c

c-----------------------------------------------------------------------
```

5.3.5 Generated wrapper program (file fft_wrap.h)

Subroutine wrapper hides some of the details of calling subroutines set_mesh and redistribute_data. Arguments to wrapper are:

- dg — the data grid ID, as generated in the header file (id_dg1 in this example).
- pg — the distribution ("process grid") ID, also as generated in the header file (id_row or id_col in this example).
- type — the data type; INTEGER_T, REAL_T, or DOUBLEPRECISION_T.
- local — the local section of the data grid.
- flag — which archetype routine to call; SETMESH for set_mesh or REDISTRIBUTEDATA for redistribute_data.

```
      subroutine wrapper(dg,pg,type,local,flag)
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'fft_header.h'
      integer dg,pg,type
      integer local(*)
      integer ip(3), id(3),ig(3)
      logical lw(3),corners,flag
      goto (100), dg
  100 call pack3(nx_dg1, ny_dg1, nz_dg1, id)
      goto 9998
 9998 goto (1, 2), pg
    1 call pack3(nprocx_col, nprocy_col, nprocz_col, ip)
      call pack3(nghostx_col, nghosty_col, nghostz_col, ig)
      call pack3l(lwrapx_col, lwrapy_col, lwrapz_col, lw)
```

```
      corners = lcorner_col
      goto 9999
    2 call pack3(nprocx_row, nprocy_row, nprocz_row, ip)
      call pack3(nghostx_row, nghosty_row, nghostz_row, ig)
      call pack3l(lwrapx_row, lwrapy_row, lwrapz_row, lw)
      corners = lcorner_row
      goto 9999
 9999 if (flag) then
            call set_mesh(ip, id, ig, lw, corners, type, dg)
      else
            call redistribute_data(ip, ig, lw, corners, local, dg)
      endif
      end
```

5.4 Poisson solver

5.4.1 Program specification.

This example is based on the discussion of the Poisson problem in [8]. The program finds a numerical solution to the 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 discretization and Jacobi iteration; i.e., by discretizing the problem domain and applying the following operation to all interior points until convergence is reached:

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

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.

5.4.2 Program code (file `poisson.FM`)

Observe the following:

- As in the FFT example in §5.3, utility program `header.exe` is used to generate files containing `PARAMETER` declarations and a wrapper program for archetype routines `set_mesh` and `redistribute_data`. §5.4.3 describes inputs to `header.exe` in more detail, and §5.4.4 and §5.4.5 show its output.

- The program makes use of one 2-dimensional data grid, `uk`, and also declares a local variable `ukp1` to hold a temporary copy of the new values during the update operation.

- The program begins with the data grid collected in a single process; it performs initialization in that process and then distributes the values over a 2-dimensional process grid. At the end of the computation, the data is once again collected in a single process for output.

- Values for stepsize (`H`) and convergence tolerance `TOL`) are to be read at runtime from standard input, so the program performs this read in the designated I/O process (as described in §6.2) and then uses `broadcast` to copy their values to the other processes.

- Space for each local section is declared using the `PARAMETER` `lclsize_dg1`, which is computed in automatically-generated header file `poisson_header.h` to be the maximum space required for a local section in either of the distributions. The local section is then passed to the routines that operate on it (`initgrid`, `computegrid`, and `printgrid`) as an adjustable array argument, using the actual dimensions appropriate to the current distribution, as obtained from `data_bounds`.

- The bulk of the computation is performed in subroutine `computegrid`. The basic computation is a "neighbor operation", as described in §1.2.2; the parallel version uses several archetype features and routines:

  - New values are to be computed only for points in the interior of the grid, so `intersect` is used to compute the intersection of this region with the local section, and the resulting values are used as loop bounds.
  - Computation of new values for each grid point is a "neighbor operation", as described in §1.2.2, so ghost boundaries are used to contain values from neighboring processes, and these boundaries are refreshed at each iteration using `boundary_exchange`.
  - Whether convergence has occurred depends on the global maximum of `abs(ukp1(i,j) - uk(i,j))`, so each process computes a local maximum for this expression, and then `global_max_dp` is used to find the maximum over all processes.

```
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------
c
c     example of mesh/spectral archetype
c
c     Poisson solver
```

```
c
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c     main program
c
c-----------------------------------------------------------------------

      program main
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_header.h'

      processors (MAXPROCS)
      call spawn(procs)
      end

c-----------------------------------------------------------------------
c
c     process-main program
c
c-----------------------------------------------------------------------

      subroutine procmain()
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_header.h'

      integer procnum, idummy(3)
      integer procnum_one, procxyz_one(3)
      integer procnum_blk, procxyz_blk(3)
      integer low(3), high(3)

      double precision uk(lclsize_dg1)
      double precision ukp1(lclsize_dg1)

      double precision F, G
      external F, G

      double precision diffmax
      integer nsteps

c
```

```
c     read and broadcast constants
c
      call local_pos(idummy,procnum,-1)
      if (procnum .eq. IOPROC) then
          print*, 'enter stepsize'
          read*, H
          print*, 'enter tolerance'
          read*, TOL
          print*, 'enter interval to check for convergence'
          read*, NCHECK
          print*, 'enter maximum number of steps'
          read*, MAXSTEPS
      endif
      call broadcast(NIC,NRC,NDPC,iconst,rconst,dconst,-1)

c
c     set up grid (undistributed)
c
      call wrapper(id_dg1,id_one,DOUBLEPRECISION_T,uk,SETMESH)
      call local_pos(procxyz_one,procnum_one,id_dg1)

c
c     initialize undistributed grid
c
      if (procnum_one .eq. 1) then
          call initgrid(nx_dg1, ny_dg1, uk)
      endif

c
c     distribute
c
      call wrapper(id_dg1,id_blk,DOUBLEPRECISION_T,uk,REDISTRIBUTEDATA)
      call local_pos(procxyz_blk,procnum_blk,id_dg1)

c
c
c     perform computation
c
      call data_bounds(procxyz_blk, low, high, id_dg1)
      call computegrid(low(1),high(1),low(2),high(2),uk,ukp1,
     -    diffmax,nsteps)

c
c     redistribute to collect all data in one process again
c
      call wrapper(id_dg1,id_one,DOUBLEPRECISION_T,uk,REDISTRIBUTEDATA)
```

```
c
c     print results
c
      if (procnum_one .eq. 1) then
          print*, 'NX, NY = ', nx_dg1, ny_dg1
          print*, 'H = ', H
          print*, 'tolerance = ', TOL
          call Fprint
          call Gprint

          if (diffmax .le. TOL) then
              print*, 'convergence occurred in ', nsteps, ' steps'
          else
              print*, 'no convergence in ', MAXSTEPS, ' steps; ',
     -            'max. difference ', diffmax
          endif
          call printgrid(nx_dg1, ny_dg1, uk)
      endif

      end

c---------------------------------------------------------------------
c
c     initialize grid
c
c---------------------------------------------------------------------

      subroutine initgrid(nx, ny, uk)
      integer nx, ny
      double precision uk(nx, ny)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_header.h'

      double precision F, G
      external F, G

      integer i, j

c
c     interior points
c
      do i = 2, nx-1
      do j = 2, ny-1
          uk(i,j) = F(i,j,nx,ny,H)
```

```
      enddo
      enddo
c
c     boundary points
c
      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

      end

c----------------------------------------------------------------------
c
c     print grid data
c
c----------------------------------------------------------------------

      subroutine printgrid(nx, ny, uk)
      integer nx, ny
      double precision uk(nx, ny)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_header.h'

      integer i, j

      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     do grid computation
c
c----------------------------------------------------------------------
```

```
      subroutine computegrid(ixlo,ixhi,iylo,iyhi,uk,ukp1,diffmax,nsteps)

      integer ixlo,ixhi,iylo,iyhi
      double precision uk(ixlo:ixhi,iylo:iyhi)
      double precision ukp1(ixlo:ixhi,iylo:iyhi)
      double precision diffmax
      integer nsteps

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_header.h'

      double precision F, G
      external F, G

      double precision diff, difflocal
      integer glbstart(3), glbend(3)
      integer lclstart(3), lclend(3)
      integer ipxyz(3), iproc
      integer istart, iend, jstart, jend, kstart, kend
      integer locmaxin(3), locmaxout(3)
      logical iempty(3)
      integer i, j, k

c     compute loop bounds
      call local_pos(ipxyz,iproc,id_dg1)
      call pack3(2,2,1,glbstart)
      call pack3(nx_dg1-1,ny_dg1-1,1,glbend)
      call intersect(ipxyz,glbstart,glbend,lclstart,lclend,iempty,
     -    id_dg1)
      call unpack3(istart,jstart,kstart,lclstart)
      call unpack3(iend,jend,kend,lclend)

c     loop until convergence
      diffmax = TOL + 1.0

      do k = 1,MAXSTEPS

c         refresh ghost boundaries
          call boundary_exchange(uk,id_dg1)
c         compute new values
          do i = istart, iend
          do j = jstart, jend
              ukp1(i,j) = 0.25*(H*H*F(i,j,nx_dg1,ny_dg1,H)
     -            + uk(i-1,j) + uk(i,j-1)
```

```
     -            + uk(i+1,j) + uk(i,j+1) )
          enddo
          enddo
c         every NCHECK-th step, recompute convergence test
          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 global_max_dp(difflocal,locmaxin,diffmax,locmaxout,
     -            id_dg1)
          endif
c         copy new values to old values
          do i = istart, iend
          do j = jstart, jend
              uk(i,j) = ukp1(i,j)
          enddo
          enddo

c         check for convergence
          nsteps = k
          if (diffmax .le. TOL) go to 1000

      enddo

1000  return
      end

c-----------------------------------------------------------------------
c
c     functions for Poisson equation
c
c-----------------------------------------------------------------------

      double precision function F(i,j,nx,ny,h)
      implicit none
      integer i, j, nx, ny
      double precision h

      F = 0.0
      end

c-----------------------------------------------------------------------
```

```
      subroutine Fprint
      implicit none
      print*, 'F(i,j) = 0.0'
      end

c-----------------------------------------------------------------------

      double precision function G(i,j,nx,ny,h)
      implicit none
      integer i, j, nx, ny
      double precision h

      G = (i+j)*h
      end

c-----------------------------------------------------------------------

      subroutine Gprint
      implicit none
      print*, 'G(i,j) = (i+j)*H'
      end

c-----------------------------------------------------------------------
c
c     automatically-generated wrapper program to pack arrays
c         and call set_mesh or redistribute_data
c
c-----------------------------------------------------------------------

      include 'poisson_wrap.h'

c-----------------------------------------------------------------------
```

5.4.3 Generating declarations and wrapper (file poisson.header_in)

The following input to makeheader.sh is used to generate header and wrapper-program files for this example. Observe the following:

- This application requires one data grid and two distributions, one that collects all data on one process and one that distributes it among processes. Input to header.exe includes a "name" for each grid and distribution, which is used to generate PARAMETERs.
- Lines declaring the user-defined common block const are to be incorporated into the generated header file.

```
>>What are dimension 1, 2, 3 called?
x
```

```
y
z
>>How many data grids are there?
1
>>For each data grid:
>>  What is its name?
dg1
>>  What are its dimensions?
10
10
1
>>End of 'for each data grid'
>>How many processes are there?
4
>>How many different distributions are there?
2
>>For each distribution:
>>  What is its name?
one
>>  What are the dimensions of the process grid?
1
1
1
>>  For each of the 3 dimensions, what is the width of the ghost boundary?
0
0
0
>>  For each of the 3 dimensions, does it wrap?  (y/n)
n
n
n
>>  Do you need to include corners in boundary exchanges? (y/n)
n
>>  What is its name?
blk
>>  What are the dimensions of the process grid?
2
2
1
>>  For each of the 3 dimensions, what is the width of the ghost boundary?
1
1
0
>>  For each of the 3 dimensions, does it wrap?  (y/n)
n
n
n
```

```
>>  Do you need to include corners in boundary exchanges? (y/n)
n
>>End of 'for each distribution'
>>Do you want parameters for dimension bounds for local sections? (y/n)
>>(These are only useful when process grid dimensions divide
>>    data grid dimensions evenly.)
n
>>What is the name of the header file to be created?
poisson_header.h
>>Enter additional lines for the header file; '.' to stop:
c     stepsize
      double precision H
c     convergence criterion
      double precision TOL
c     how often to check for convergence
      integer NCHECK
c     maximum number of steps
      integer MAXSTEPS
c     common block for constants
      integer NIC, NRC, NDPC
      parameter (NIC=2)
      parameter (NRC=0)
      parameter (NDPC=2)
      double precision dconst(NIC)
      real rconst
      integer iconst(NDPC)
      common /consts/ dconst, iconst
      equivalence (H, dconst(1))
      equivalence (TOL, dconst(2))
      equivalence (NCHECK, iconst(1))
      equivalence (MAXSTEPS, iconst(2))
.
>>What is the name of the subroutine file to make?
poisson_wrap.h
```

5.4.4 Generated declarations (file poisson_header.h)

Observe the following:

- PARAMETERs are declared using the naming conventions described in §5.3.4.
- The user-supplied lines defining common block const are included in the generated file.

```
c----------------------------------------------------------------------
c
c     parameter declarations
c
```

```
c---------------------------------------------------------------------

c---------------------------------------------------------------------
c
c     constants (always present)
c
      logical SETMESH, REDISTRIBUTEDATA
      parameter (SETMESH=.TRUE.)
      parameter (REDISTRIBUTEDATA=.FALSE.)

c---------------------------------------------------------------------
c
c     grid IDs, dimensions, number of processes (supplied by you)
c
      integer procs
      parameter (procs = 4)

      integer id_dg1
      parameter (id_dg1 = 1)
      integer nx_dg1
      parameter (nx_dg1 = 10)
      integer ny_dg1
      parameter (ny_dg1 = 10)
      integer nz_dg1
      parameter (nz_dg1 = 1)

c---------------------------------------------------------------------
c
c     one distribution:
c
c     distribution ID, parameters for one
c     (supplied by you)
c
      integer id_one
      parameter (id_one = 1)
      integer nprocx_one
      parameter (nprocx_one = 1)
      integer nprocy_one
      parameter (nprocy_one = 1)
      integer nprocz_one
      parameter (nprocz_one = 1)
      integer nghostx_one
      parameter (nghostx_one = 0)
      integer nghosty_one
      parameter (nghosty_one = 0)
      integer nghostz_one
      parameter (nghostz_one = 0)
```

```
      logical lwrapx_one
      parameter (lwrapx_one = .FALSE.)
      logical lwrapy_one
      parameter (lwrapy_one = .FALSE.)
      logical lwrapz_one
      parameter (lwrapz_one = .FALSE.)
      logical lcorner_one
      parameter (lcorner_one = .FALSE.)
c
c     maximum local section dimensions for one, without ghost boundary
c     (computed)
c
      integer nxlcl_dg1_one
      parameter (nxlcl_dg1_one =
     -      (nx_dg1 + nprocx_one -1) / nprocx_one)
      integer nylcl_dg1_one
      parameter (nylcl_dg1_one =
     -      (ny_dg1 + nprocy_one -1) / nprocy_one)
      integer nzlcl_dg1_one
      parameter (nzlcl_dg1_one =
     -      (nz_dg1 + nprocz_one -1) / nprocz_one)
c
c     1d sizes of local sections for one
c     (computed)
c
      integer lclsize_dg1_one
      parameter (lclsize_dg1_one =
     -      (nxlcl_dg1_one + 2*nghostx_one) *
     -      (nylcl_dg1_one + 2*nghosty_one) *
     -      (nzlcl_dg1_one + 2*nghostz_one))

c----------------------------------------------------------------------
c
c     blk distribution:
c
c     distribution ID, parameters for blk
c     (supplied by you)
c
      integer id_blk
      parameter (id_blk = 2)
      integer nprocx_blk
      parameter (nprocx_blk = 2)
      integer nprocy_blk
      parameter (nprocy_blk = 2)
      integer nprocz_blk
      parameter (nprocz_blk = 1)
      integer nghostx_blk
```

```
      parameter (nghostx_blk = 1)
      integer nghosty_blk
      parameter (nghosty_blk = 1)
      integer nghostz_blk
      parameter (nghostz_blk = 0)
      logical lwrapx_blk
      parameter (lwrapx_blk = .FALSE.)
      logical lwrapy_blk
      parameter (lwrapy_blk = .FALSE.)
      logical lwrapz_blk
      parameter (lwrapz_blk = .FALSE.)
      logical lcorner_blk
      parameter (lcorner_blk = .FALSE.)
c
c     maximum local section dimensions for blk, without ghost boundary
c     (computed)
c
      integer nxlcl_dg1_blk
      parameter (nxlcl_dg1_blk =
     -      (nx_dg1 + nprocx_blk -1) / nprocx_blk)
      integer nylcl_dg1_blk
      parameter (nylcl_dg1_blk =
     -      (ny_dg1 + nprocy_blk -1) / nprocy_blk)
      integer nzlcl_dg1_blk
      parameter (nzlcl_dg1_blk =
     -      (nz_dg1 + nprocz_blk -1) / nprocz_blk)
c
c     1d sizes of local sections for blk
c     (computed)
c
      integer lclsize_dg1_blk
      parameter (lclsize_dg1_blk =
     -      (nxlcl_dg1_blk + 2*nghostx_blk) *
     -      (nylcl_dg1_blk + 2*nghosty_blk) *
     -      (nzlcl_dg1_blk + 2*nghostz_blk))

c----------------------------------------------------------------------
c
c     maximum and combined sizes
c     (computed)
c
c     maximum computed using the formula:
c     max(i,j) = (j*(j/i)+i*(i/j)) / ((i/j)+(j/i))
c

c
c     maximum size of local sections for dg1
```

```
c
      integer i1
      parameter (i1 = lclsize_dg1_one)
      integer i2
      parameter (i2 = lclsize_dg1_blk)
      integer i3
      parameter (i3 =
     -      (i2*(i2/i1) + i1*(i1/i2)) /
     -      ((i1/i2) + (i2/i1)) )
      integer lclsize_dg1
      parameter (lclsize_dg1 = i3)
c
c     1d size of combined local sections
c     (computed)
c
      integer lclsize
      parameter (lclsize =
     -      lclsize_dg1 )

c-----------------------------------------------------------------------
c
c     additional declarations (supplied by you)
c
c     stepsize
      double precision H
c     convergence criterion
      double precision TOL
c     how often to check for convergence
      integer NCHECK
c     maximum number of steps
      integer MAXSTEPS
c     common block for constants
      integer NIC, NRC, NDPC
      parameter (NIC=2)
      parameter (NRC=0)
      parameter (NDPC=2)
      double precision dconst(NIC)
      real rconst
      integer iconst(NDPC)
      common /consts/ dconst, iconst
      equivalence (H, dconst(1))
      equivalence (TOL, dconst(2))
      equivalence (NCHECK, iconst(1))
      equivalence (MAXSTEPS, iconst(2))

c-----------------------------------------------------------------------
```

5.4.5 Generated wrapper program (file poisson_wrap.h)

Subroutine wrapper hides some of the details of calling subroutines set_mesh and redistribute_data. Arguments to wrapper are as described in §5.3.5.

```
      subroutine wrapper(dg,pg,type,local,flag)
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_header.h'
      integer dg,pg,type
      integer local(*)
      integer ip(3), id(3),ig(3)
      logical lw(3),corners,flag
      goto (100), dg
 100  call pack3(nx_dg1, ny_dg1, nz_dg1, id)
      goto 9998
9998  goto (1, 2), pg
   1  call pack3(nprocx_one, nprocy_one, nprocz_one, ip)
      call pack3(nghostx_one, nghosty_one, nghostz_one, ig)
      call pack3l(lwrapx_one, lwrapy_one, lwrapz_one, lw)
      corners = lcorner_one
      goto 9999
   2  call pack3(nprocx_blk, nprocy_blk, nprocz_blk, ip)
      call pack3(nghostx_blk, nghosty_blk, nghostz_blk, ig)
      call pack3l(lwrapx_blk, lwrapy_blk, lwrapz_blk, lw)
      corners = lcorner_blk
      goto 9999
9999  if (flag) then
            call set_mesh(ip, id, ig, lw, corners, type, dg)
      else
            call redistribute_data(ip, ig, lw, corners, local, dg)
      endif
      end
```

5.5 Poisson solver (variant)

5.5.1 Program specification.

This program performs the same calculation as the previous example (§5.4.1, differing only in the implementation details discussed below.

5.5.2 Program code (file poisson_fixed.FM)

Observe the following:

- This program is identical to the version in §5.4, except that it does not make use of adjustable array arguments to pass the local section to the routines that operate on it. Instead, each of these routines (`initgrid`, `computegrid`, and `printgrid`) declares the local section (its argument) using sizes obtained from the automatically-generated header file `poisson_fixed_header.h`. Observe that this approach works only if the dimensions of the process grid evenly divide the dimensions of the data grid (which is the case in this example but need not be true in general).

```
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------
c
c     example of mesh/spectral archetype
c
c     Poisson solver
c
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c     main program
c
c-----------------------------------------------------------------------

      program main
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_fixed_header.h'

      processors (MAXPROCS)
      call spawn(procs)
      end

c-----------------------------------------------------------------------
c
c     process-main program
c
c-----------------------------------------------------------------------

      subroutine procmain()
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_fixed_header.h'

      integer procnum, idummy(3)
      integer procnum_one, procxyz_one(3)
```

```
      double precision uk(lclsize_dg1)
      double precision ukp1(lclsize_dg1)

      double precision F, G
      external F, G

      double precision diffmax
      integer nsteps

c
c     read and broadcast constants
c
      call local_pos(idummy,procnum,-1)
      if (procnum .eq. IOPROC) then
          print*, 'enter stepsize'
          read*, H
          print*, 'enter tolerance'
          read*, TOL
          print*, 'enter interval to check for convergence'
          read*, NCHECK
          print*, 'enter maximum number of steps'
          read*, MAXSTEPS
      endif
      call broadcast(NIC,NRC,NDPC,iconst,rconst,dconst,-1)

c
c     set up grid (undistributed)
c
      call wrapper(id_dg1,id_one,DOUBLEPRECISION_T,uk,SETMESH)
      call local_pos(procxyz_one,procnum_one,id_dg1)

c
c     initialize undistributed grid
c
      if (procnum_one .eq. 1) then
          call initgrid(uk)
      endif

c
c     distribute
c
      call wrapper(id_dg1,id_blk,DOUBLEPRECISION_T,uk,REDISTRIBUTEDATA)

c
c     perform computation
c
      call computegrid(uk,ukp1,diffmax,nsteps)
```

```
c
c     redistribute to collect all data in one process again
c
      call wrapper(id_dg1,id_one,DOUBLEPRECISION_T,uk,REDISTRIBUTEDATA)

c
c     print results
c
      if (procnum_one .eq. 1) then
          print*, 'NX, NY = ', nx_dg1, ny_dg1
          print*, 'H = ', H
          print*, 'tolerance = ', TOL
          call Fprint
          call Gprint

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

      end

c----------------------------------------------------------------------
c
c     initialize grid
c
c----------------------------------------------------------------------

      subroutine initgrid(uk)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_fixed_header.h'

      double precision uk(nx_dg1, ny_dg1)

      double precision F, G
      external F, G

      integer i, j
```

```
c
c     interior points
c
      do i = 2, nx_dg1-1
      do j = 2, ny_dg1-1
          uk(i,j) = F(i,j,nx_dg1,ny_dg1,H)
      enddo
      enddo
c
c     boundary points
c
      do j = 1, ny_dg1
          uk(1,j) = G(1,j,nx_dg1,ny_dg1,H)
          uk(nx_dg1,j) = G(nx_dg1,j,nx_dg1,ny_dg1,H)
      enddo
      do i = 2, nx_dg1-1
          uk(i,1) = G(i,1,nx_dg1,ny_dg1,H)
          uk(i,ny_dg1) = G(i,ny_dg1,nx_dg1,ny_dg1,H)
      enddo

      end

c-----------------------------------------------------------------------
c
c     print grid data
c
c-----------------------------------------------------------------------

      subroutine printgrid(uk)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_fixed_header.h'

      double precision uk(nx_dg1, ny_dg1)

      integer i, j

      do i = 1, nx_dg1
          if (nx_dg1 .gt. 10) print*, ' '
          print 9999, (uk(i,j), j = 1, ny_dg1)
      enddo
9999  format(10F8.4)

      end
```

```
c-----------------------------------------------------------------------
c
c     do grid computation
c
c-----------------------------------------------------------------------


      subroutine computegrid(uk,ukp1,diffmax,nsteps)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_fixed_header.h'

      double precision uk(ixlo_dg1_blk:ixhi_dg1_blk,
     -    iylo_dg1_blk:iyhi_dg1_blk)
      double precision ukp1(ixlo_dg1_blk:ixhi_dg1_blk,
     -    iylo_dg1_blk:iyhi_dg1_blk)
      double precision diffmax
      integer nsteps

      double precision F, G
      external F, G

      double precision diff, difflocal
      integer glbstart(3), glbend(3)
      integer lclstart(3), lclend(3)
      integer ipxyz(3), iproc
      integer istart, iend, jstart, jend, kstart, kend
      integer locmaxin(3), locmaxout(3)
      logical iempty(3)
      integer i, j, k

c     compute loop bounds
      call local_pos(ipxyz,iproc,id_dg1)
      call pack3(2,2,1,glbstart)
      call pack3(nx_dg1-1,ny_dg1-1,1,glbend)
      call intersect(ipxyz,glbstart,glbend,lclstart,lclend,iempty,
     -    id_dg1)
      call unpack3(istart,jstart,kstart,lclstart)
      call unpack3(iend,jend,kend,lclend)

c     loop until convergence
      diffmax = TOL + 1.0

      do k = 1,MAXSTEPS
```

```
c         refresh ghost boundaries
          call boundary_exchange(uk,id_dg1)
c         compute new values
          do i = istart, iend
          do j = jstart, jend
              ukp1(i,j) = 0.25*(H*H*F(i,j,nx_dg1,ny_dg1,H)
     -            + uk(i-1,j) + uk(i,j-1)
     -            + uk(i+1,j) + uk(i,j+1) )
          enddo
          enddo
c         every NCHECK-th step, recompute convergence test
          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 global_max_dp(difflocal,locmaxin,diffmax,locmaxout,
     -            id_dg1)
          endif
c         copy new values to old values
          do i = istart, iend
          do j = jstart, jend
              uk(i,j) = ukp1(i,j)
          enddo
          enddo

c         check for convergence
          nsteps = k
          if (diffmax .le. TOL) go to 1000

      enddo

1000  return
      end

c----------------------------------------------------------------------
c
c     functions for Poisson equation
c
c----------------------------------------------------------------------

      double precision function F(i,j,nx,ny,h)
      implicit none
      integer i, j, nx, ny
```

```
      double precision h

      F = 0.0
      end

c----------------------------------------------------------------------

      subroutine Fprint
      implicit none
      print*, 'F(i,j) = 0.0'
      end

c----------------------------------------------------------------------

      double precision function G(i,j,nx,ny,h)
      implicit none
      integer i, j, nx, ny
      double precision h

      G = (i+j)*h
      end

c----------------------------------------------------------------------

      subroutine Gprint
      implicit none
      print*, 'G(i,j) = (i+j)*H'
      end

c----------------------------------------------------------------------
c
c     automatically-generated wrapper program to pack arrays
c         and call set_mesh or redistribute_data
c
c----------------------------------------------------------------------

      include 'poisson_fixed_wrap.h'

c----------------------------------------------------------------------
```

5.5.3 Generating declarations and wrapper (file poisson_fixed.header_in)

The following input to `makeheader.sh` is used to generate header and wrapper-program files for this example. Observe the following:

- Input for this program is identical to that for the preceding example (§5.4), except that it requests additional `PARAMETER`s giving the dimension bounds (lower and upper index limits) for the local sections. These `PARAMETER`s are then used to declare the dimensions of the local sections in subroutine `computegrid`.

```
>>What are dimension 1, 2, 3 called?
x
y
z
>>How many data grids are there?
1
>>For each data grid:
>>  What is its name?
dg1
>>  What are its dimensions?
10
10
1
>>End of 'for each data grid'
>>How many processes are there?
4
>>How many different distributions are there?
2
>>For each distribution:
>>  What is its name?
one
>>  What are the dimensions of the process grid?
1
1
1
>>  For each of the 3 dimensions, what is the width of the ghost boundary?
0
0
0
>>  For each of the 3 dimensions, does it wrap?  (y/n)
n
n
n
>>  Do you need to include corners in boundary exchanges? (y/n)
n
>>  What is its name?
blk
>>  What are the dimensions of the process grid?
2
2
1
>>  For each of the 3 dimensions, what is the width of the ghost boundary?
1
1
```

```
0
>>  For each of the 3 dimensions, does it wrap?  (y/n)
n
n
n
>>  Do you need to include corners in boundary exchanges? (y/n)
n
>>End of 'for each distribution'
>>Do you want parameters for dimension bounds for local sections? (y/n)
>>(These are only useful when process grid dimensions divide
>>    data grid dimensions evenly.)
y
>>What is the name of the header file to be created?
poisson_fixed_header.h
>>Enter additional lines for the header file; '.' to stop:
c     stepsize
      double precision H
c     convergence criterion
      double precision TOL
c     how often to check for convergence
      integer NCHECK
c     maximum number of steps
      integer MAXSTEPS
c     common block for constants
      integer NIC, NRC, NDPC
      parameter (NIC=2)
      parameter (NRC=0)
      parameter (NDPC=2)
      double precision dconst(NIC)
      real rconst
      integer iconst(NDPC)
      common /consts/ dconst, iconst
      equivalence (H, dconst(1))
      equivalence (TOL, dconst(2))
      equivalence (NCHECK, iconst(1))
      equivalence (MAXSTEPS, iconst(2))
.
>>What is the name of the subroutine file to make?
poisson_fixed_wrap.h
```

5.5.4 Generated declarations (file `poisson_fixed_header.h`)

Observe the following:

- This file is identical to the one generated for the preceding example (§5.4), except that it includes additional `PARAMETER`s for each combination of data grid and distribution that define the dimension bounds (lower and upper index limits, including ghost boundaries) for the corresponding local section — for example, the `x` dimension bounds for data grid `dg1` and distribution `blk` are `ixlo_dg1_blk` and `ixhi_dg1_blk`.

```
c-----------------------------------------------------------------------
c
c     parameter declarations
c
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c     constants (always present)
c
      logical SETMESH, REDISTRIBUTEDATA
      parameter (SETMESH=.TRUE.)
      parameter (REDISTRIBUTEDATA=.FALSE.)

c-----------------------------------------------------------------------
c
c     grid IDs, dimensions, number of processes (supplied by you)
c
      integer procs
      parameter (procs = 4)

      integer id_dg1
      parameter (id_dg1 = 1)
      integer nx_dg1
      parameter (nx_dg1 = 10)
      integer ny_dg1
      parameter (ny_dg1 = 10)
      integer nz_dg1
      parameter (nz_dg1 = 1)

c-----------------------------------------------------------------------
c
c     one distribution:
c
c     distribution ID, parameters for one
c     (supplied by you)
c
      integer id_one
      parameter (id_one = 1)
      integer nprocx_one
      parameter (nprocx_one = 1)
      integer nprocy_one
      parameter (nprocy_one = 1)
      integer nprocz_one
```

```
      parameter (nprocz_one = 1)
      integer nghostx_one
      parameter (nghostx_one = 0)
      integer nghosty_one
      parameter (nghosty_one = 0)
      integer nghostz_one
      parameter (nghostz_one = 0)
      logical lwrapx_one
      parameter (lwrapx_one = .FALSE.)
      logical lwrapy_one
      parameter (lwrapy_one = .FALSE.)
      logical lwrapz_one
      parameter (lwrapz_one = .FALSE.)
      logical lcorner_one
      parameter (lcorner_one = .FALSE.)
c
c     maximum local section dimensions for one, without ghost boundary
c     (computed)
c
      integer nxlcl_dg1_one
      parameter (nxlcl_dg1_one =
     -      (nx_dg1 + nprocx_one -1) / nprocx_one)
      integer nylcl_dg1_one
      parameter (nylcl_dg1_one =
     -      (ny_dg1 + nprocy_one -1) / nprocy_one)
      integer nzlcl_dg1_one
      parameter (nzlcl_dg1_one =
     -      (nz_dg1 + nprocz_one -1) / nprocz_one)
c
c     local section dimension bounds for one, with ghost boundary
c     (computed)
c     (only useful if process grid dimensions divide
c         data grid dimensions evenly)
c
      integer ixlo_dg1_one
      parameter (ixlo_dg1_one = 1-nghostx_one)
      integer ixhi_dg1_one
      parameter (ixhi_dg1_one = nxlcl_dg1_one + nghostx_one)
      integer iylo_dg1_one
      parameter (iylo_dg1_one = 1-nghosty_one)
      integer iyhi_dg1_one
      parameter (iyhi_dg1_one = nylcl_dg1_one + nghosty_one)
      integer izlo_dg1_one
      parameter (izlo_dg1_one = 1-nghostz_one)
      integer izhi_dg1_one
      parameter (izhi_dg1_one = nzlcl_dg1_one + nghostz_one)
```

```
c
c     1d sizes of local sections for one
c     (computed)
c
      integer lclsize_dg1_one
      parameter (lclsize_dg1_one =
     -      (nxlcl_dg1_one + 2*nghostx_one) *
     -      (nylcl_dg1_one + 2*nghosty_one) *
     -      (nzlcl_dg1_one + 2*nghostz_one))

c----------------------------------------------------------------------
c
c     blk distribution:
c
c     distribution ID, parameters for blk
c     (supplied by you)
c
      integer id_blk
      parameter (id_blk = 2)
      integer nprocx_blk
      parameter (nprocx_blk = 2)
      integer nprocy_blk
      parameter (nprocy_blk = 2)
      integer nprocz_blk
      parameter (nprocz_blk = 1)
      integer nghostx_blk
      parameter (nghostx_blk = 1)
      integer nghosty_blk
      parameter (nghosty_blk = 1)
      integer nghostz_blk
      parameter (nghostz_blk = 0)
      logical lwrapx_blk
      parameter (lwrapx_blk = .FALSE.)
      logical lwrapy_blk
      parameter (lwrapy_blk = .FALSE.)
      logical lwrapz_blk
      parameter (lwrapz_blk = .FALSE.)
      logical lcorner_blk
      parameter (lcorner_blk = .FALSE.)
c
c     maximum local section dimensions for blk, without ghost boundary
c     (computed)
c
      integer nxlcl_dg1_blk
      parameter (nxlcl_dg1_blk =
     -      (nx_dg1 + nprocx_blk -1) / nprocx_blk)
      integer nylcl_dg1_blk
```

```
      parameter (nylcl_dg1_blk =
     -      (ny_dg1 + nprocy_blk -1) / nprocy_blk)
      integer nzlcl_dg1_blk
      parameter (nzlcl_dg1_blk =
     -      (nz_dg1 + nprocz_blk -1) / nprocz_blk)
c
c     local section dimension bounds for blk, with ghost boundary
c     (computed)
c     (only useful if process grid dimensions divide
c         data grid dimensions evenly)
c
      integer ixlo_dg1_blk
      parameter (ixlo_dg1_blk = 1-nghostx_blk)
      integer ixhi_dg1_blk
      parameter (ixhi_dg1_blk = nxlcl_dg1_blk + nghostx_blk)
      integer iylo_dg1_blk
      parameter (iylo_dg1_blk = 1-nghosty_blk)
      integer iyhi_dg1_blk
      parameter (iyhi_dg1_blk = nylcl_dg1_blk + nghosty_blk)
      integer izlo_dg1_blk
      parameter (izlo_dg1_blk = 1-nghostz_blk)
      integer izhi_dg1_blk
      parameter (izhi_dg1_blk = nzlcl_dg1_blk + nghostz_blk)

c
c     1d sizes of local sections for blk
c     (computed)
c
      integer lclsize_dg1_blk
      parameter (lclsize_dg1_blk =
     -      (nxlcl_dg1_blk + 2*nghostx_blk) *
     -      (nylcl_dg1_blk + 2*nghosty_blk) *
     -      (nzlcl_dg1_blk + 2*nghostz_blk))

c----------------------------------------------------------------------
c
c     maximum and combined sizes
c     (computed)
c
c     maximum computed using the formula:
c     max(i,j) = (j*(j/i)+i*(i/j)) / ((i/j)+(j/i))
c

c
c     maximum size of local sections for dg1
c
      integer i1
```

```
      parameter (i1 = lclsize_dg1_one)
      integer i2
      parameter (i2 = lclsize_dg1_blk)
      integer i3
      parameter (i3 =
     -      (i2*(i2/i1) + i1*(i1/i2)) /
     -      ((i1/i2) + (i2/i1)) )
      integer lclsize_dg1
      parameter (lclsize_dg1 = i3)
c
c     1d size of combined local sections
c     (computed)
c
      integer lclsize
      parameter (lclsize =
     -      lclsize_dg1 )

c-----------------------------------------------------------------------
c
c     additional declarations (supplied by you)
c
c     stepsize
      double precision H
c     convergence criterion
      double precision TOL
c     how often to check for convergence
      integer NCHECK
c     maximum number of steps
      integer MAXSTEPS
c     common block for constants
      integer NIC, NRC, NDPC
      parameter (NIC=2)
      parameter (NRC=0)
      parameter (NDPC=2)
      double precision dconst(NIC)
      real rconst
      integer iconst(NDPC)
      common /consts/ dconst, iconst
      equivalence (H, dconst(1))
      equivalence (TOL, dconst(2))
      equivalence (NCHECK, iconst(1))
      equivalence (MAXSTEPS, iconst(2))

c-----------------------------------------------------------------------
```

5.5.5 Generated wrapper program (file poisson_fixed_wrap.h)

Subroutine wrapper hides some of the details of calling subroutines set_mesh and redistribute_data. Arguments to wrapper are as described in §5.3.5.

```
      subroutine wrapper(dg,pg,type,local,flag)
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'poisson_fixed_header.h'
      integer dg,pg,type
      integer local(*)
      integer ip(3), id(3),ig(3)
      logical lw(3),corners,flag
      goto (100), dg
  100 call pack3(nx_dg1, ny_dg1, nz_dg1, id)
      goto 9998
 9998 goto (1, 2), pg
    1 call pack3(nprocx_one, nprocy_one, nprocz_one, ip)
      call pack3(nghostx_one, nghosty_one, nghostz_one, ig)
      call pack3l(lwrapx_one, lwrapy_one, lwrapz_one, lw)
      corners = lcorner_one
      goto 9999
    2 call pack3(nprocx_blk, nprocy_blk, nprocz_blk, ip)
      call pack3(nghostx_blk, nghosty_blk, nghostz_blk, ig)
      call pack3l(lwrapx_blk, lwrapy_blk, lwrapz_blk, lw)
      corners = lcorner_blk
      goto 9999
 9999 if (flag) then
            call set_mesh(ip, id, ig, lw, corners, type, dg)
      else
            call redistribute_data(ip, ig, lw, corners, local, dg)
      endif
      end
```

5.6 Multigrid prototype

5.6.1 Program specification.

This example is a prototype for multigrid applications, based on the discussion of multigrid methods in [8]. It implements the parallel equivalent of the pseudocode shown in figure 9. (Observe that this pseudocode omits all details of data structures.)

5.6.2 Program code (file multigrid.FM)

Observe the following:

```
c
c     perform relaxation on grid levels, from fine to coarse
c
      do i = NLEVELS, 2, -1
          call relax(<grid for level i>)
          call restrict(<grid for level i>, <grid for level i-1>)
      enddo
c
c     solve coarsest grid
c
      call solve(<grid for level 1>)
c
c     interpolate solution to finer grids
c
      do i = 2, NLEVELS
          call interpolate(<grid for level i>, <grid for level i+1>)
      enddo
```

Figure 9: Pseudocode for multigrid prototype.

- As in the FFT example in §5.3, utility program `header.exe` is used to generate files containing `PARAMETER` declarations and a wrapper program for archetype routines `set_mesh` and `redistribute_data`. §5.6.3 describes inputs to `header.exe` in more detail, and §5.6.4 and §5.6.5 show its output.
- In this application, only one distribution is required, but the data grids to be distributed vary in size, from fine to coarse. Each process has a local section for each grid.
- Since Fortran 77 lacks record/structure and pointer constructs, data for local sections is concatenated into a single 1-dimensional array, `local_grids`. Variable `istart` gives the starting indices of the individual grids within this array; since the semantics of Fortran 77 is to pass by reference, the i-th grid can be passed by referencing `local_grids(istart(i))`. Further, local sections of individual grids are passed to subroutines `interpolate`, `restrict`, `relax`, and `solve` as adjustable array arguments, as in the Poisson example in §5.4.

```
c----------------------------------------------------------------------
c----------------------------------------------------------------------
c
c     example of mesh/spectral archetype
c
c     simplified prototype multigrid
c
```

```
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c     main program
c
c-----------------------------------------------------------------------

      program main
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'multigrid_header.h'

      processors (MAXPROCS)
      call spawn(procs)
      end

c-----------------------------------------------------------------------
c
c     process-main program
c
c-----------------------------------------------------------------------

      subroutine procmain()
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'multigrid_header.h'

c     storage for all grid levels
      double precision local_grids(lclsize)

      integer iproc, ipxyz(3)
      integer ilo(3,NLEVELS)
      integer ihi(3,NLEVELS)
      integer igrid_id(NLEVELS)
      integer istart(NLEVELS)
      integer lvl

c     set up grid IDs
      igrid_id(1) = id_lv1
      igrid_id(2) = id_lv2
      igrid_id(3) = id_lv3
      igrid_id(4) = id_lv4
```

```
c     set up 'pointers' to grid levels
      istart(1) = 1
      istart(2) = 1+lclsize_lv1_blk
      istart(3) = 1+lclsize_lv1_blk+lclsize_lv2_blk
      istart(4) = 1+lclsize_lv1_blk+lclsize_lv2_blk+lclsize_lv3_blk

c     set up grids and local bounds
      do lvl = 1, NLEVELS
          call wrapper(igrid_id(lvl),id_blk,DOUBLEPRECISION_T,
     -        local_grids(istart(lvl)),SETMESH)
          call local_pos(ipxyz,iproc,igrid_id(lvl))
          call data_bounds(ipxyz,ilo(1,lvl),ihi(1,lvl),igrid_id(lvl))
      enddo

c
c     perform relaxation on grid levels, from fine to coarse
c
      do lvl = NLEVELS, 2, -1
          call relax(ilo(1,lvl),ihi(1,lvl),
     -        ilo(2,lvl),ihi(2,lvl),
     -        ilo(3,lvl),ihi(3,lvl),
     -        igrid_id(lvl),
     -        local_grids(istart(lvl)))
          call restrict(
     -        ilo(1,lvl),ihi(1,lvl),
     -        ilo(2,lvl),ihi(2,lvl),
     -        ilo(3,lvl),ihi(3,lvl),
     -        igrid_id(lvl),
     -        local_grids(istart(lvl)),
     -        ilo(1,lvl-1),ihi(1,lvl-1),
     -        ilo(2,lvl-1),ihi(2,lvl-1),
     -        ilo(3,lvl-1),ihi(3,lvl-1),
     -        igrid_id(lvl-1),
     -        local_grids(istart(lvl-1)))
      enddo
c
c     solve coarsest grid
c
      call solve(ilo(1,1),ihi(1,1),
     -    ilo(2,1),ihi(2,1),
     -    ilo(3,1),ihi(3,1),
     -    igrid_id(1),
     -    local_grids(istart(1)))
c
c     interpolate solution to finer grids
c
      do lvl = 2, NLEVELS
```

```
          call interpolate(
     -        ilo(1,lvl-1),ihi(1,lvl-1),
     -        ilo(2,lvl-1),ihi(2,lvl-1),
     -        ilo(3,lvl-1),ihi(3,lvl-1),
     -        igrid_id(lvl-1),
     -        local_grids(istart(lvl-1)),
     -        ilo(1,lvl),ihi(1,lvl),
     -        ilo(2,lvl),ihi(2,lvl),
     -        ilo(3,lvl),ihi(3,lvl),
     -        igrid_id(lvl),
     -        local_grids(istart(lvl)))
      enddo
      end

c-----------------------------------------------------------------------
c
c     interpolation routine
c
c-----------------------------------------------------------------------

      subroutine interpolate(ixlo1,ixhi1,iylo1,iyhi1,izlo1,izhi1,
     -    igrid_id1, local_grid1,
     -    ixlo2,ixhi2,iylo2,iyhi2,izlo2,izhi2,
     -    igrid_id2, local_grid2)

      integer ixlo1,ixhi1,iylo1,iyhi1,izlo1,izhi1
      integer igrid_id1
      integer local_grid1(ixlo1:ixhi1,iylo1:iyhi1,izlo1:izhi1)
      integer ixlo2,ixhi2,iylo2,iyhi2,izlo2,izhi2
      integer igrid_id2
      integer local_grid2(ixlo2:ixhi2,iylo2:iyhi2,izlo2:izhi2)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'multigrid_header.h'

      integer iproc,ipxyz(3)
      integer isizes1(3),isizes2(3)
      integer nxlocal1,nylocal1,nzlocal1,nxlocal2,nylocal2,nzlocal2

      call local_pos(ipxyz,iproc,igrid_id1)
      call data_widths(ipxyz,.false.,isizes1,igrid_id1)
      call unpack3(nxlocal1,nylocal1,nzlocal1,isizes1)
      call data_widths(ipxyz,.false.,isizes2,igrid_id2)
      call unpack3(nxlocal2,nylocal2,nzlocal2,isizes2)
```

```fortran
      call boundary_exchange(local_grid1,igrid_id1)

      print*, 'doing interpolation in process ', ipxyz,
     -    ' from gridsize ', isizes1(1), ' to gridsize ', isizes2(1)

      end

c-----------------------------------------------------------------------
c
c     restriction routine
c
c-----------------------------------------------------------------------

      subroutine restrict(ixlo1,ixhi1,iylo1,iyhi1,izlo1,izhi1,
     -    igrid_id1, local_grid1,
     -    ixlo2,ixhi2,iylo2,iyhi2,izlo2,izhi2,
     -    igrid_id2, local_grid2)

      integer ixlo1,ixhi1,iylo1,iyhi1,izlo1,izhi1
      integer igrid_id1
      integer local_grid1(ixlo1:ixhi1,iylo1:iyhi1,izlo1:izhi1)
      integer ixlo2,ixhi2,iylo2,iyhi2,izlo2,izhi2
      integer igrid_id2
      integer local_grid2(ixlo2:ixhi2,iylo2:iyhi2,izlo2:izhi2)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'multigrid_header.h'

      integer iproc,ipxyz(3)
      integer isizes1(3),isizes2(3)
      integer nxlocal1,nylocal1,nzlocal1,nxlocal2,nylocal2,nzlocal2

      call local_pos(ipxyz,iproc,igrid_id1)
      call data_widths(ipxyz,.false.,isizes1,igrid_id1)
      call unpack3(nxlocal1,nylocal1,nzlocal1,isizes1)
      call data_widths(ipxyz,.false.,isizes2,igrid_id2)
      call unpack3(nxlocal2,nylocal2,nzlocal2,isizes2)

      call boundary_exchange(local_grid1,igrid_id1)

      print*, 'doing restriction in process ', ipxyz,
     -    ' from gridsize ', isizes1(1), ' to gridsize ', isizes2(1)

      end
```

```
c----------------------------------------------------------------------
c
c     relaxation routine (iterative solve)
c
c----------------------------------------------------------------------

      subroutine relax(ixlo,ixhi,iylo,iyhi,izlo,izhi,
     -    igrid_id, local_grid)

      integer ixlo,ixhi,iylo,iyhi,izlo,izhi
      integer igrid_id
      integer local_grid(ixlo:ixhi,iylo:iyhi,izlo:izhi)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'multigrid_header.h'

      integer iproc,ipxyz(3)
      integer isizes(3)
      integer nxlocal,nylocal,nzlocal
      integer ix,iy,iz

      call local_pos(ipxyz,iproc,igrid_id)
      call data_widths(ipxyz,.false.,isizes,igrid_id)
      call unpack3(nxlocal,nylocal,nzlocal,isizes)

      call boundary_exchange(local_grid,igrid_id)

      print*, 'doing relaxation in process ', ipxyz,
     -    ' for gridsize ', nxlocal

      do ix = 1, nxlocal
      do iy = 1, nylocal
      do iz = 1, nzlocal
          local_grid(ix,iy,iz) = 0.0
      enddo
      enddo
      enddo
      end

c----------------------------------------------------------------------
c
c     coarse-level solve
c
c----------------------------------------------------------------------
```

```fortran
      subroutine solve(ixlo,ixhi,iylo,iyhi,izlo,izhi,
     -    igrid_id, local_grid)

      integer ixlo,ixhi,iylo,iyhi,izlo,izhi
      integer igrid_id
      integer local_grid(ixlo:ixhi,iylo:iyhi,izlo:izhi)

      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'multigrid_header.h'

      integer iproc,ipxyz(3)
      integer isizes(3)
      integer nxlocal,nylocal,nzlocal
      integer ix,iy,iz

      call local_pos(ipxyz,iproc,igrid_id)
      call data_widths(ipxyz,.false.,isizes,igrid_id)
      call unpack3(nxlocal,nylocal,nzlocal,isizes)

      call boundary_exchange(local_grid,igrid_id)

      print*, 'doing solve in process ', ipxyz,
     -    ' for gridsize ', nxlocal

      do ix = 1, nxlocal
      do iy = 1, nylocal
      do iz = 1, nzlocal
          local_grid(ix,iy,iz) = 0.0
      enddo
      enddo
      enddo
      end

c-----------------------------------------------------------------------
c
c     automatically-generated wrapper program to pack arrays
c         and call set_mesh or redistribute_data
c
c-----------------------------------------------------------------------

      include 'multigrid_wrap.h'

c-----------------------------------------------------------------------
```

5.6.3 Generating declarations and wrapper (file multigrid.header_in)

The following input to makeheader.sh is used to generate header and wrapper-program files for this example. Observe the following:

- This application requires only one distribution, but several data grids, ranging from fine to coarse. Input to header.exe includes a "name" for each grid and distribution, which is used to generate PARAMETERs.
- Lines declaring the user-defined PARAMETER NLEVELS are to be incorporated into the generated header file.

```
>>What are dimension 1, 2, 3 called?
x
y
z
>>How many data grids are there?
4
>>For each data grid:
>>  What is its name?
lv1
>>  What are its dimensions?
4
4
4
>>  What is its name?
lv2
>>  What are its dimensions?
8
8
8
>>  What is its name?
lv3
>>  What are its dimensions?
16
16
16
>>  What is its name?
lv4
>>  What are its dimensions?
32
32
32
>>End of 'for each data grid'
>>How many processes are there?
8
>>How many different distributions are there?
```

```
1
>>For each distribution:
>>  What is its name?
blk
>>  What are the dimensions of the process grid?
2
2
2
>>  For each of the 3 dimensions, what is the width of the ghost boundary?
1
1
1
>>  For each of the 3 dimensions, does it wrap?  (y/n)
n
n
n
>>  Do you need to include corners in boundary exchanges? (y/n)
y
>>End of 'for each distribution'
>>Do you want parameters for dimension bounds for local sections? (y/n)
>>(These are only useful when process grid dimensions divide
>>    data grid dimensions evenly.)
n
>>What is the name of the header file to be created?
multigrid_header.h
>>Enter additional lines for the header file; '.' to stop:
c     number of grid levels
      integer NLEVELS
      parameter (NLEVELS=4)
.
>>What is the name of the subroutine file to make?
multigrid_wrap.h
```

5.6.4 Generated declarations (file `multigrid_header.h`)

Observe the following:

- `PARAMETER`s are declared using the naming conventions described in §5.3.4. `PARAMETER lclsize` computes the total local section size for all declared grids.
- The user-supplied lines defining `PARAMETER NLEVELS` are included in the generated file.

```
c----------------------------------------------------------------------
c
c     parameter declarations
c
```

```fortran
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c     constants (always present)
c
      logical SETMESH, REDISTRIBUTEDATA
      parameter (SETMESH=.TRUE.)
      parameter (REDISTRIBUTEDATA=.FALSE.)

c-----------------------------------------------------------------------
c
c     grid IDs, dimensions, number of processes (supplied by you)
c
      integer procs
      parameter (procs = 8)

      integer id_lv1
      parameter (id_lv1 = 1)
      integer nx_lv1
      parameter (nx_lv1 = 4)
      integer ny_lv1
      parameter (ny_lv1 = 4)
      integer nz_lv1
      parameter (nz_lv1 = 4)
      integer id_lv2
      parameter (id_lv2 = 2)
      integer nx_lv2
      parameter (nx_lv2 = 8)
      integer ny_lv2
      parameter (ny_lv2 = 8)
      integer nz_lv2
      parameter (nz_lv2 = 8)
      integer id_lv3
      parameter (id_lv3 = 3)
      integer nx_lv3
      parameter (nx_lv3 = 16)
      integer ny_lv3
      parameter (ny_lv3 = 16)
      integer nz_lv3
      parameter (nz_lv3 = 16)
      integer id_lv4
      parameter (id_lv4 = 4)
      integer nx_lv4
      parameter (nx_lv4 = 32)
      integer ny_lv4
      parameter (ny_lv4 = 32)
```

```
      integer nz_lv4
      parameter (nz_lv4 = 32)

c-----------------------------------------------------------------------
c
c     blk distribution:
c
c     distribution ID, parameters for blk
c     (supplied by you)
c
      integer id_blk
      parameter (id_blk = 1)
      integer nprocx_blk
      parameter (nprocx_blk = 2)
      integer nprocy_blk
      parameter (nprocy_blk = 2)
      integer nprocz_blk
      parameter (nprocz_blk = 2)
      integer nghostx_blk
      parameter (nghostx_blk = 1)
      integer nghosty_blk
      parameter (nghosty_blk = 1)
      integer nghostz_blk
      parameter (nghostz_blk = 1)
      logical lwrapx_blk
      parameter (lwrapx_blk = .FALSE.)
      logical lwrapy_blk
      parameter (lwrapy_blk = .FALSE.)
      logical lwrapz_blk
      parameter (lwrapz_blk = .FALSE.)
      logical lcorner_blk
      parameter (lcorner_blk = .TRUE.)
c
c     maximum local section dimensions for blk, without ghost boundary
c     (computed)
c
      integer nxlcl_lv1_blk
      parameter (nxlcl_lv1_blk =
     -      (nx_lv1 + nprocx_blk -1) / nprocx_blk)
      integer nylcl_lv1_blk
      parameter (nylcl_lv1_blk =
     -      (ny_lv1 + nprocy_blk -1) / nprocy_blk)
      integer nzlcl_lv1_blk
      parameter (nzlcl_lv1_blk =
     -      (nz_lv1 + nprocz_blk -1) / nprocz_blk)
      integer nxlcl_lv2_blk
      parameter (nxlcl_lv2_blk =
```

```
     -      (nx_lv2 + nprocx_blk -1) / nprocx_blk)
      integer nylcl_lv2_blk
      parameter (nylcl_lv2_blk =
     -      (ny_lv2 + nprocy_blk -1) / nprocy_blk)
      integer nzlcl_lv2_blk
      parameter (nzlcl_lv2_blk =
     -      (nz_lv2 + nprocz_blk -1) / nprocz_blk)
      integer nxlcl_lv3_blk
      parameter (nxlcl_lv3_blk =
     -      (nx_lv3 + nprocx_blk -1) / nprocx_blk)
      integer nylcl_lv3_blk
      parameter (nylcl_lv3_blk =
     -      (ny_lv3 + nprocy_blk -1) / nprocy_blk)
      integer nzlcl_lv3_blk
      parameter (nzlcl_lv3_blk =
     -      (nz_lv3 + nprocz_blk -1) / nprocz_blk)
      integer nxlcl_lv4_blk
      parameter (nxlcl_lv4_blk =
     -      (nx_lv4 + nprocx_blk -1) / nprocx_blk)
      integer nylcl_lv4_blk
      parameter (nylcl_lv4_blk =
     -      (ny_lv4 + nprocy_blk -1) / nprocy_blk)
      integer nzlcl_lv4_blk
      parameter (nzlcl_lv4_blk =
     -      (nz_lv4 + nprocz_blk -1) / nprocz_blk)
c
c     1d sizes of local sections for blk
c     (computed)
c
      integer lclsize_lv1_blk
      parameter (lclsize_lv1_blk =
     -      (nxlcl_lv1_blk + 2*nghostx_blk) *
     -      (nylcl_lv1_blk + 2*nghosty_blk) *
     -      (nzlcl_lv1_blk + 2*nghostz_blk))

      integer lclsize_lv2_blk
      parameter (lclsize_lv2_blk =
     -      (nxlcl_lv2_blk + 2*nghostx_blk) *
     -      (nylcl_lv2_blk + 2*nghosty_blk) *
     -      (nzlcl_lv2_blk + 2*nghostz_blk))

      integer lclsize_lv3_blk
      parameter (lclsize_lv3_blk =
     -      (nxlcl_lv3_blk + 2*nghostx_blk) *
     -      (nylcl_lv3_blk + 2*nghosty_blk) *
     -      (nzlcl_lv3_blk + 2*nghostz_blk))
```

```
      integer lclsize_lv4_blk
      parameter (lclsize_lv4_blk =
     -      (nxlcl_lv4_blk + 2*nghostx_blk) *
     -      (nylcl_lv4_blk + 2*nghosty_blk) *
     -      (nzlcl_lv4_blk + 2*nghostz_blk))

c----------------------------------------------------------------------
c
c     maximum and combined sizes
c     (computed)
c
c     maximum computed using the formula:
c     max(i,j) = (j*(j/i)+i*(i/j)) / ((i/j)+(j/i))
c

c
c     maximum size of local sections for lv1
c
      integer lclsize_lv1
      parameter (lclsize_lv1 = lclsize_lv1_blk)
c
c     maximum size of local sections for lv2
c
      integer lclsize_lv2
      parameter (lclsize_lv2 = lclsize_lv2_blk)
c
c     maximum size of local sections for lv3
c
      integer lclsize_lv3
      parameter (lclsize_lv3 = lclsize_lv3_blk)
c
c     maximum size of local sections for lv4
c
      integer lclsize_lv4
      parameter (lclsize_lv4 = lclsize_lv4_blk)
c
c     1d size of combined local sections
c     (computed)
c
      integer lclsize
      parameter (lclsize =
     -      lclsize_lv1 +
     -      lclsize_lv2 +
     -      lclsize_lv3 +
     -      lclsize_lv4 )

c----------------------------------------------------------------------
```

```
c
c     additional declarations (supplied by you)
c
c     number of grid levels
      integer NLEVELS
      parameter (NLEVELS=4)

c-----------------------------------------------------------------------
```

5.6.5 Generated wrapper program (file `multigrid_wrap.h`)

Subroutine `wrapper` hides some of the details of calling subroutines `set_mesh` and `redistribute_data`. Arguments to `wrapper` are as described in §5.3.5.

```
      subroutine wrapper(dg,pg,type,local,flag)
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'
      include 'multigrid_header.h'
      integer dg,pg,type
      integer local(*)
      integer ip(3), id(3),ig(3)
      logical lw(3),corners,flag
      goto (100, 200, 300, 400), dg
  100 call pack3(nx_lv1, ny_lv1, nz_lv1, id)
      goto 9998
  200 call pack3(nx_lv2, ny_lv2, nz_lv2, id)
      goto 9998
  300 call pack3(nx_lv3, ny_lv3, nz_lv3, id)
      goto 9998
  400 call pack3(nx_lv4, ny_lv4, nz_lv4, id)
      goto 9998
 9998 goto (1), pg
    1 call pack3(nprocx_blk, nprocy_blk, nprocz_blk, ip)
      call pack3(nghostx_blk, nghosty_blk, nghostz_blk, ig)
      call pack3l(lwrapx_blk, lwrapy_blk, lwrapz_blk, lw)
      corners = lcorner_blk
      goto 9999
 9999 if (flag) then
            call set_mesh(ip, id, ig, lw, corners, type, dg)
      else
            call redistribute_data(ip, ig, lw, corners, local, dg)
      endif
      end
```

5.7 Recursive doubling example

5.7.1 Program specification.

This program is a somewhat contrived example illustrating the use of the recursive doubling subroutine. Each of the 4 processes contains a group of arrays (integer, real, and double-precision); the goal of the program is to "reverse" this data, so that process i contains the data originally found in process $4 - (i - 1)$. This is easily accomplished using a recursive doubling scheme in which the operation that combines local values with neighbor values simply copies the neighbor values.

5.7.2 Program code (file `doubling.FM`)

Observe the following:

- No distributed grids are used in this application; each process defines data for its group of arrays, initializes it, and then calls `recursive_doubling` to perform the recursive doubling. Since the recursive doubling is to performed over all processes (rather than only over those pertaining to a particular distributed grid), the final argument to `recursive_doubling` is -1.
- User-supplied subroutine `revproc` defines the desired operation for the recursive doubling. Execution of `recursive_doubling` repeatedly combines values from "neighboring" processes (refer to [8], for example, for a discussion of recursive doubling) using `revproc`.

```
c----------------------------------------------------------------------
c----------------------------------------------------------------------
c
c     example of mesh/spectral archetype
c
c     recursive doubling example:  reverse arrays
c
c----------------------------------------------------------------------
c----------------------------------------------------------------------

c----------------------------------------------------------------------
c
c     main program
c
c----------------------------------------------------------------------

      program main
      implicit none
      include 'arch_uparms.h'
```

```
      include 'arch_parms.h'
      integer procs
      parameter (procs=4)

      processors (MAXPROCS)
      call spawn(procs)
      end

c-----------------------------------------------------------------------
c
c     process-main program
c
c-----------------------------------------------------------------------

      subroutine procmain()
      implicit none
      include 'arch_uparms.h'
      include 'arch_parms.h'

      integer NI, NR, NDP
      parameter (NI=2)
      parameter (NR=1)
      parameter (NDP=1)
      integer idata(NI), ireverse(NI)
      real rdata(NR), rreverse(NR)
      double precision dpdata(NDP), dpreverse(NDP)
      integer iproc, ipxyz(3), i
      external revproc

c     set input data and print
      call local_pos(ipxyz,iproc,-1)
      do i = 1, ni
          idata(i) = 100*iproc+i
      enddo
      do i = 1, nr
          rdata(i) = 1000*iproc+i
      enddo
      do i = 1, ndp
          dpdata(i) = 10000*iproc+i
      enddo
      print*, 'input data for proc ', iproc, ' is ',
     -    idata, rdata, dpdata

c     do recursive doubling on all processes
      call recursive_doubling(revproc,NI,NR,NDP,idata,rdata,dpdata,
     -    ireverse,rreverse,dpreverse,-1)
```

```
c     print results
      print*, 'results for proc ', iproc, ' are ',
     -     ireverse, rreverse, dpreverse
      end

c----------------------------------------------------------------------
c
c     recursive-doubling subroutine (called by recursive_doubling())
c
c     copy second argument to output -- when applied in a recursive
c         doubling scheme, the effect is to reverse the order of the
c         input
c
c----------------------------------------------------------------------

      subroutine revproc(ni,nr,ndp,i_in1,i_in2,r_in1,r_in2,
     $        dp_in1,dp_in2,i_out,r_out,dp_out)

      integer ni, nr, ndp
      integer i_in1(ni), i_in2(ni), i_out(ni)
      real r_in1(nr), r_in2(nr), r_out(nr)
      double precision dp_in1(ndp), dp_in2(ndp), dp_out(ndp)

      include 'arch_uparms.h'
      include 'arch_parms.h'
      implicit none

      integer i

      do i = 1, ni
          i_out(i) = i_in2(i)
      enddo
      do i = 1, nr
          r_out(i) = r_in2(i)
      enddo
      do i = 1, ndp
          dp_out(i) = dp_in2(i)
      enddo
      end

c----------------------------------------------------------------------
```

6 Implementation specifics

As noted in §2, the current implementation is based on Fortran 77 (including a few of the most widely-supported extensions — long variable names, lower-case source, and the do/enddo construction) and uses either Fortran M ([4, 1], available via anonymous FTP from Argonne National Labs) or MPI ([6], widely available) for parallelism.

6.1 Compiling and linking programs

6.1.1 Fortran M version

To compile and link a program based on the Fortran M implementation, assuming the Fortran M compiler fm is installed:

- Compile the user-supplied main program file.
- Compile any additional source files (e.g., for user-written subroutines called from the process-main subroutine).
- Compile the archetype library routines (arch_lib.FM).
- Link the results of the above compilations.

For example, to compile example program hello.FM shown earlier:

```
fm -c hello.FM
fm -c arch_lib.FM
fm -o hello.exe hello.o arch_lib.o
```

Observe that the library routines and the main program file must be recompiled if any of the parameters in arch_uparms.h are changed.

6.1.2 Fortran-plus-MPI version

To compile and link a program based on the Fortran-plus-MPI implementation, assuming MPI is installed:

- Compile the user-supplied main program file.
- Compile any additional source files (e.g., for user-written subroutines called from the process-main subroutine).
- Compile the archetype library routines (arch_lib.f).
- Link the results of the above compilations.

For example, to compile an MPI version of example program hello.f:

```
mpif77 -c hello.f
mpif77 -c arch_lib.f
mpif77 -o hello.exe hello.o arch_lib.o
```

Observe that the library routines and the main program file must be recompiled if any of the parameters in arch_uparms.h are changed.

6.2 Hints and tips

- In the Fortran M implementation (but not in the Fortran-plus-MPI implementation), the process identified by process number `IOPROC` has access to the original program's standard input, as illustrated in the examples in §5.4 and §5.5.
- Many routines are implemented only for grids of type `DOUBLEPRECISION_T`. Extending the implementation to support other types should be straightforward.
- Converting the implementation to use some other Fortran-compatible parallel language or library should be a straightforward matter:
  - Modify `spawn` to use the appropriate mechanism for starting processes.
  - Replace Fortran M sends and receives with the appropriate mechanism for deterministic point-to-point message-passing. (E.g., for NX or PVM, one would use tagged messages.)

  (In fact this approach was used to produce the MPI version from the reference Fortran M implementation.)

7 Obtaining the code

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

- Fortran M implementation (`FM3D.tar.gz`).
- Fortran-plus-MPI implementation (`MPI3D.tar.gz`).

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

- Required source files.
- A sample `Makefile`.
- Source for the utility program `header.exe` described in §4.
- Source and related files for the example programs described in §5.

Acknowledgements

The authors thank Lena Petersen, for implementing the 2-dimensional FFT example; John Langford, for implementing the utility program `header.exe`; and Mani Chandy and Dan Meiron, who direct the scientific archetypes project.

References

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

[2] K. M. Chandy. Concurrent program archetypes. In *Proceedings of the Scalable Parallel Library Conference*, 1994.

[3] G. Davis. Spectral methods for scientific computing. Available via http://www.etext.caltech.edu/Implementations/Spectral/spectral.html.

[4] 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.

[5] B. L. Massingill. The mesh archetype. Technical Report CS-TR-96-25, California Institute of Technology, 1996. Also available via http://www.etext.caltech.edu/Implementations/Mesh/meshdoc.ps or http://www.etext.caltech.edu/Implementations/Mesh/meshdoc.html.

[6] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.

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