MPI Structures and MPIIO

Chris Brady

April 8, 2008

What are MPI types?

- New MPI datatypes, similar to MPI_REAL or MPI_FLOAT
- Usually represent subsections of arrays or similar

When should you use them?

- When sending array subsections (especially in C)
- **When using MPIIO**
- When you're using non-blocking communication

MPI_Type_commit

```
int MPI_Type_commit(MPI_Datatype *Datatype)

CALL MPI_TYPE_COMMIT(Datatype,ierr)
```

Description

Commits a created type to the MPI layer. Allows it to be used in MPI commands. Until this is called, use of an MPI derived datatype will fail.

MPI_Type_free

```
int MPI_Type_free(MPI_Datatype *Datatype)
```

```
CALL MPI_TYPE_FREE(Datatype,ierr)
```

Description

Frees a committed type. This type can then be recreated and recommitted. If Datatype is not a committed datatype then an error occurs.

- You have to commit a type using MPI_Type_commit before you can use it
- Once a type has been committed it can be used like one of the primitive MPI datatypes
- When you no longer need a type, delete it using MPI_Type_free
- The proceedure is the same for committing and freeing is the same for all types, whether MPI1 or MPI2
- Some commands, notably MPI_Reduce and MPI_Allreduce require additional code to work with MPI types, which will not be covered here

MPI_Type_contiguous

int MPI_Type_contiguous (int count, MPI_Datatype old_type, MPI_Datatype * new_type)

CALL MPI_TYPE_CONTIGUOUS (count, old_type,new_type, ierr)

Description

- Creates a type consisting of count adjacent copies of the type old_type.
- Not a very useful type of MPI type, but easy to understand

```
PROGRAM MPI_TYPE_TEST
    INTEGER :: new_type,ierr
    REAL, DIMENSION(100) :: Data

    !Put some data in the Data array here on rank 0
    !Create the type representing the array
    CALL MPI_TYPE_CONTIGUOUS(100, MPI_REAL, new_type,&
         ierr)
    CALL MPI_TYPE_COMMIT(new_type,ierr)

    CALL MPI_BCAST(Data, 1, new_type, 0, MPI_COMM_WORLD,&
         ierr)

    CALL MPI_TYPE_FREE(new_type,ierr)
END PROGRAM MPI_TYPE_TEST
```

```
int main(int argc, char** argv)
   {
     MPI_Datatype new_type;
     int error;
     float Data[100];

     error = MPI_Type_contiguous(100, MPI_FLOAT,
             &new_type);
     error = MPI_Type_commit(&new_type);
     error = MPI_Bcast(Data, 1, new_type, 0,
             MPI_COMM_WORLD);

   }
```

Other MPI types

- Possible to create other MPI types, including ones with arbitrarily complex structures
- The one that you will use most is MPI_Type_create_subarray
- This allows you to define a rectangular subarray of an array

MPI_Type_create_subarray

int MPI_Type_create_subarray (int ndims, int array_of_sizes[ ], int array_of_subsizes[ ], int array_of_starts[ ], int order, MPI_Datatype old_type, MPI_Datatype *new_type)

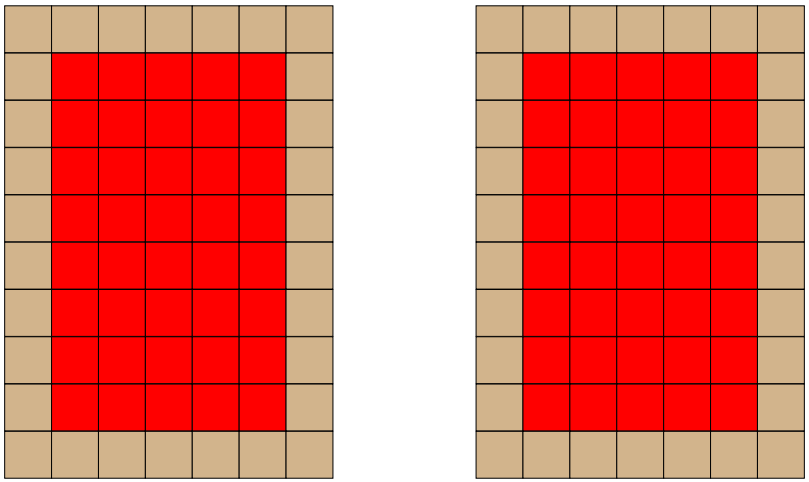
CALL MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes( ), array_of_subsizes( ), array_of_starts( ), order, old_type, new_type, ierr)

Description

- Creates a type which represents an ndimsD subarray of an ndimsD array
- array_of_sizes, array_of_starts and array_of_subsizes are arrays with ndims elements
- array_of_sizes describes the extents of the whole array in each direction
- array_of_starts describes the offset for the starting position of the subarray in each direction
- array_of_subsizes describes the extents of the subarray in each direction
- As before, all lengths are given in multiples of old_type

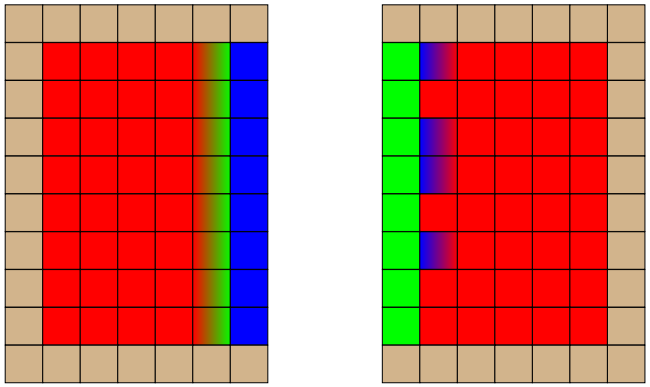
Weaknesses of MPI_Type_create_subarray

- At first sight, MPI_Type_create_subarray looks like it solves all problems, but it has problems all of its own
- array_of_sizes must be the same on every process
- array_of_subsizes must be the same on every process
- Therefore, MPI_Type_create_subarray is simply to do uniform, even subdivision of an array, with an identical fraction of the array being referred to on each processor
- In fact, it's main purpose is in MPIIO, where it is used to represent the subsection of a global array held by each processor
- It does still work in communication, although it's not quite as useful as it might be.



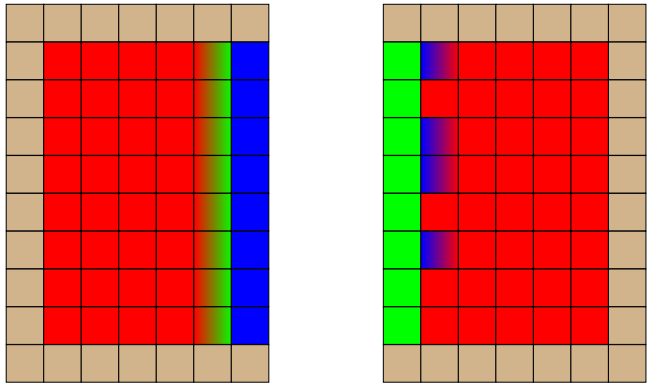
2D domain decomposed hydrocode

- Hydrocode decomposed onto 2 processors
- Brown ghost cells must be populated from "real" cells on adjacent processors



2D 2nd-order domain decomposed hydrocode

- Want MPI types which correspond both to the source cells (gradient shaded) and the destination cells (solid non-red cells)
- Two transactions (send from left to right, send from right to left)
- Consider only one sending to right, receiving from left



2D 2nd-order domain decomposed hydrocode

- Assume the array starts with (0,0) top left cell
- Cell to be source are (5, 1:8)
- Cells to be destination are (0, 1:8)

```
INTEGER :: ierr, type_sendtoright, type_recvfromleft
INTEGER, DIMENSION(2) :: sizes = (/7, 10/)

subsizes = (/1, 8/)
starts = (/5, 1/)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, &
     starts, MPI_ORDER_FORTRAN, MPI_REAL, &
     type_sendtoright, ierr)

subsizes = (/1, 8/)
starts = (/0, 1/)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, &
     starts, MPI_ORDER_FORTRAN, MPI_REAL, &
     type_recvfromleft, ierr)
```

```
MPI_Datatype type_sendtoright, type_recvfromleft ;
int sizes[2] = {7, 10};

subsizes = {1, 8}
starts = {5, 1}
error = MPI_Type_create_subarray(2, sizes, subsizes,
        starts, MPI_ORDER_C, MPI_FLOAT,
        &type_sendtoright);
error = MPI_Type_commit(&type_right_recv);

subsizes = {1, 4}
starts = {0, 1}
error = MPI_Type_create_subarray(2, sizes, subsizes,
        starts,MPI_ORDER_C, MPI_FLOAT,
        &type_recvfromleft);
```

```
CALL MPI_SENDRECV(Data,1,type_sendtoright,right,tag,&
     Data,1,type_recvfromleft, left, tag,&
     MPI_COMM_WORLD, status, ierr)
```

```
error = MPI_Sendrecv(Data,1,type_sendtoright,right,
    tag,Data,1,type_recvfromleft, left, tag,
    MPI_COMM_WORLD, status)
```

2D domain decomposed hydrocode

- In practice, there is no speed benefit to using the MPI types to describe the ghost cells
- This may change because if more codes start to use MPI types in this way then there will be an effort to improve performance
- In C, this style may make code easier to read because it removes all the subarray copying code
- The routine really comes into its own when using MPIIO

MPI Structures and MPIIO

Chris Brady

April 8, 2008

MPIIO Advantages

- Improved output speed in large parallel environments
- For LUSTRE based filesystems, you WILL NOT get acceptable performance from using many small files.
- Output to a single file for any number of processors (easily)

MPIIO Disadvantages

- Syntax is not exactly like either C or FORTRAN IO (although similar in concept)
- Can be slower on desktop machines
- Produces C type binary output, which can't be read in using standard Fortran90 or F77 (although most modern F90 compilers support FORM="binary")

MPIIO Concepts

- In most senses MPIIO is the same as conventional binary IO
- There are commands to open and close files, read and write data and move file pointers
- There are things called file views which describe the layout of data across processors
- There are some commands to help with writing simple data layouts more easily than using file views, but they will not be covered.
- File views are described using MPI types

MPI_File_open

int MPI_File_open (MPI_Comm Comm, char *filename, int amode, MPI_Info info, MPI_File *mpi_fh)

CALL MPI_FILE_OPEN (Comm, filename, amode, info, mpi_fh, ierr)

Description

- Opens a file using MPIIO and returns a file handle mpi_fh. As usual in Fortran mpi_fh is of type INTEGER
- This is a collective operation. All processes must have the same amode and the filename must reference the same file (does not have to be the same filename)

Description

- Choosing whether opening the file for reading or writing is via constants passed as part of amode as normal
- info is used to pass additional parameter, which will generally vary from system to system. You create MPI_Info objects using the MPI_Info_ commands. Most generally, you can use MPI_INFO_NULL to open the file generically.

MPI_File_open modes

- MPI_MODE_RDONLY - Open for reading
- MPI_MODE_RDWR - Open for reading and writing
- MPI_MODE_WRONLY - Open for writing
- MPI_MODE_CREATE - Create the file if it does not exist,
- MPI_MODE_EXCL - Throw error if file exists
- MPI_MODE_DELETE_ON_CLOSE - Delete the file when its closed
- MPI_MODE_UNIQUE_OPEN - Throw error if file opened anywhere else
- MPI_MODE_SEQUENTIAL - Sequential mode (tapes etc.)
- MPI_MODE_APPEND - Set initial position of all file pointers to end of file

MPI_File_close

int MPI_File_close (MPI_File *mpi_fh)

CALL MPI_FILE_CLOSE (mpi_fh)

Description

- Closes and frees the file handle mpi_fh
- File handles must be closed before MPI is finalized. Otherwise behaviour is undefined.

MPI_File_write

int MPI_File_write (MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

CALL MPI_FILE_WRITE (mpi_fh, buf, count, datatype, status, ierr)

Description

- Writes to the file pointed to by the handle mpi_fh using the individual file pointer
- This is a non collective operation and can be run on any subset of processors requested
- As with normal POSIX IO, the individual filepointer is moved as data is written

MPI_File_write_all

int MPI_File_write_all (MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

CALL MPI_FILE_WRITE_ALL (mpi_fh, buf, count, datatype, status, ierr)

Description

- Writes to the file pointed to by the handle mpi_fh using the individual file pointer
- This is a collective operation and all the processors in the MPI_Comm given to MPI_File_open must call MPI_File_write_all or the code will lock.

MPI_File_read

int MPI_File_read (MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

CALL MPI_FILE_READ (mpi_fh, buf, count, datatype, status, ierr)

Description

- Reads from the file pointed to by the handle mpi_fh using the individual file pointer
- This is a non collective operation and can be run on any subset of processors requested
- As with normal POSIX IO, the individual filepointer is moved as data is read

MPI_File_read_all

int MPI_File_read_all (MPI_File mpi_fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

CALL MPI_FILE_READ_ALL (mpi_fh, buf, count, datatype, status, ierr)

Description

- Reads from the file pointed to by the handle mpi_fh using the individual file pointer
- This is a collective operation and all the processors in the MPI_Comm given to MPI_File_open must call MPI_File_write_all or the code will lock.

```
float Data[100];
MPI_File mpi_fh;

// Writes the array Data using collective IO
error = MPI_File_open(MPI_COMM_WORLD,"out.dat",
        MPI_MODE_WRONLY | MPI_MODE_CREATE,
        MPI_INFO_NULL, &mpi_fh);
error = MPI_File_write_all(mpi_fh, Data, 100,
        MPI_FLOAT,&status);
error = MPI_File_close(&mpi_fh);
```

```
REAL, DIMENSION(100) :: Data
INTEGER :: mpi_fh, ierr
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

CALL MPI_FILE_OPEN(MPI_COMM_WORLD,"out.dat",
     MPI_MODE_WRONLY + MPI_MODE_CREATE, &
     MPI_INFO_NULL, mpi_fh, ierr)
CALL MPI_FILE_WRITE_ALL(mpi_fh, Data, 100,&
      MPI_REAL, status, ierr)
CALL MPI_FILE_CLOSE(mpi_fh)
```

MPI_File_seek

int MPI_File_seek (MPI_File mpi_fh, MPI_Offset offset, int whence)

CALL MPI_FILE_SEEK (mpi_fh, offset, whence)

Description

- Moves the indiviual file pointer on a processor by a distance offset from point whence
- MPI_SEEK_SET - offset from start of file
- MPI_SEEK_CUR - offset from current positions of file pointer
- MPI_SEEK_END - offset from end of file
- In Fortran, MPI_Offset is of type INTEGER(KIND = MPI_OFFSET_KIND) rather than INTEGER

Other useful MPIIO operations

- MPI_File_read_at & MPI_File_write_at - Read or write at specific offset rather than current file pointer
- MPI_File_read_at_all & MPI_File_write_at_all - Collective read or write at specific offset rather than current file pointer
- MPI_File_iread & MPI_File_iwrite - Non blocking read and write
- MPI_File_delete - Deletes a named file.

What are file views?

- If you wish to have more control over where a given processor will write it's data then you have to use file views.
- File views use MPI types to describe the section of the global data that the current processor has, and it's location in the final file on disk
- Makes it easier to write multidimensional arrays into a single file

MPI_File_set_view

int MPI_File_set_view (MPI_File mpi_fh, MPI_Offset offset, MPI_Datatype etype, MPI_Datatype filetype, char* datarep, MPI_Info info)

CALL MPI_FILE_SET_VIEW (mpi_fh, offset, etype, filetype, datarep, info, ierr)

Description

- Sets the file view on file handle mpi_fh
- offset is the offset from the start of the file at which to apply the file view
- datarep is a string describing the data representation, usually "native"

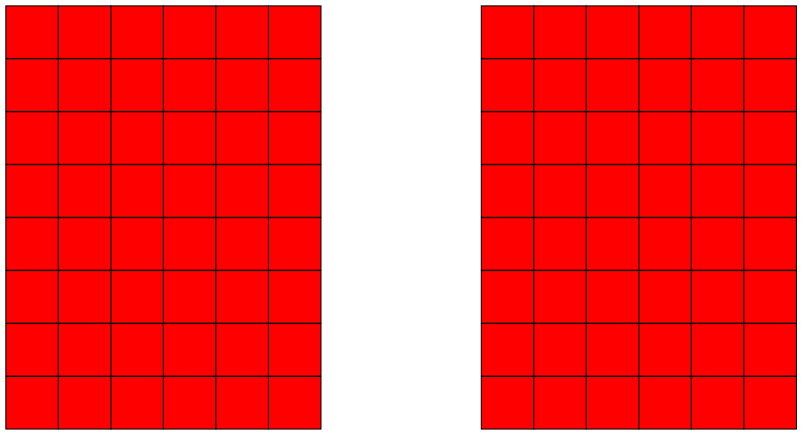
MPI_File_set_view

int MPI_File_set_view (MPI_File mpi_fh, MPI_Offset offset, MPI_Datatype etype, MPI_Datatype filetype, char* datarep, MPI_Info info)

CALL MPI_FILE_SET_VIEW (mpi_fh, offset, etype, filetype, datarep, info, ierr)

Description

- info contains additional information, will usually be MPI_INFO_NULL
- etype is the basic datatype being written to the file
- filetype is (normally) a derived datatype which describes the layout of the data for the current processor on the disk
- This will be much clearer with an example



2D domain decomposed hydrocode

- Return to the hydrocode on two processors
- We now want to only write the real cells, so ignore the ghost cells
- The global array is 12 x 8, each subarray is 6 x 8
- Use MPI_Type_create_subarray to create a representation of the layout

```
REAL, DIMENSION(6,8) :: Data_local
INTEGER, DIMENSION(2) :: sizes
INTEGER, DIMENSION(2) :: subsizes
INTEGER, DIMENSION(2) :: starts

sizes = (/12, 8/)
subsizes = (/6, 8/)
IF (rank == 0) starts = (/0, 0/)
IF (rank == 1) starts = (/6, 0/)

CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes,&
     starts, MPI_ORDER_FORTRAN, MPI_REAL,&
      new_type, ierr)
```

```
int error;
float Data_local[6][8];
int sizes[2] ;
int subsizes[2];
int starts[2];

sizes = {12, 8};
subsizes = {6, 8};
if (rank == 0) starts = {0, 0}
if (rank == 1) starts = {6, 0}

error = MPI_Type_create_subarray(2, sizes, subsizes,
        starts, MPI_ORDER_C, MPI_FLOAT, &new_type);
```

2D domain decomposed hydrocode

- Note that now, have created a subarray of the GLOBAL 12 x 8 array
- This simple code uses if statements for the start points, in general you have to work this out algorithmically
- This works as a file view because it defines the type to have the full extent of the whole array, with each processor seing a "hole" over the parts of the array that it doesn't own. Therefore, when the file view is set, the processor will only write it's own part of the array
- This is the general approach in MPIIO views. Create a type which is the size of the full array when finally written out and then put "holes" in all the places that the current processor doesn't have data for

```
int rank;
MPI_File mpi_fh;

// Writes the array Data_local using a view
error = MPI_File_open(MPI_COMM_WORLD,"out.dat",
        MPI_MODE_WRONLY | MPI_MODE_CREATE,
        MPI_INFO_NULL, &mpi_fh);
error = MPI_File_set_view(mpi_fh, 0, MPI_FLOAT,
        new_type, "native", MPI_INFO_NULL);
error = MPI_File_write_all(mpi_fh, Data_local,
        6*8, MPI_FLOAT, &status);
error = MPI_File_close(&mpi_fh);
```

```
INTEGER :: rank
INTEGER :: mpi_fh, ierr
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

CALL MPI_FILE_OPEN(MPI_COMM_WORLD,"out.dat",&
     MPI_MODE_WRONLY + MPI_MODE_CREATE, &
     MPI_INFO_NULL, mpi_fh, ierr)
CALL MPI_FILE_SET_VIEW(mpi_fh, 0, MPI_REAL, &
     new_type, "native", MPI_INFO_NULL, ierr)
CALL MPI_FILE_WRITE_ALL(mpi_fh, Data_local, &
     6*8, MPI_REAL, status,ierr)
CALL MPI_FILE_CLOSE(mpi_fh)
```

2D domain decomposed hydrocode

- This code will generate a single file called "out.dat" which will contain the single 12x8 array exactly as if it was written by a single serial process
- Note that the actual MPI_File_write_all command is unchanged by the file view and is exactly as would be expected if you were just writing the local array
- Note that once the view is applied, the different processors no longer need to interact using the shared file pointer, and file writing is done using MPI_File_write_all
- Once a view is applied, it is retained for all future writes and if you want to write in a different way, you have to set a new view

Final notes

- MPIIO greatly increases IO perfomance on parallel filesystems
- Parallel filesystems are already a part of most large HPC systems, and will be part of any future large HPC system
- Using MPIIO involves describing the layout of data on your processors using an MPI type
- On each processor create an MPI type which corresponds to the fraction of the data that the current processor controls
- Make sure that the type you create describes the extents of the whole array