Using MPI Derived Datatypes to Pass Non-Contiguous Information
From Using MPI by Gropp, Lusk, and Skjellum

In this appendix we briefly discuss some details of C and Fortran that interact with MPI.

E.1 Arrays in C and Fortran

This section discusses the layout of Fortran arrays in memory and talks very briefly about how implicit "reshapes" of arrays are handled in Fortran 77. All arrays are stored in memory according to some rule, defined by the language, that says how to map the indices of an array reference such as `a(i,j,k,l)` into a computer's memory. Understanding and exploiting this rule is important in creating efficient representations of data.

E.1.1 Column and Row Major Ordering

Many discussions of the differences between Fortran and C arrays refer to "column" and "row" major ordering. These terms come from looking at a two-dimensional array as representing a matrix. To understand this, consider the $m \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}$$

If we use the Fortran declaration

```
real a(m,n)
```

and represent the matrix element a_{ij} with `a(i,j)`, then *column major ordering*, used in the Fortran language, means that the elements are stored by columns; that is, they are stored in the order `a(1,1)`, `a(2,1)`, ..., `a(m,1)`, `a(1,2)`, ..., `a(m,n)`. *Row major ordering*, used in the C language, means that the elements are stored by rows; that is, `a[0][0]`, `a[0][1]`, ..., `a[0][n-1]`, `a[1][0]`, ..., `a[m-1][n-1]`. We have used Fortran and C array notation here to emphasize how each is stored in memory.

E.1.2 Meshes vs. Matrices

While a matrix-based view of arrays is very common and often helpful, a different view of arrays is more natural for many applications. For instance, the 2-d Poisson example in Chapter 4. The solution to the problem had a natural representation as

a function $u(x, y)$. We solved this on a discrete mesh of points (x_i, y_j) and used the Fortran element `u(i,j)` to represent $u(x_i, y_j)$. While this approach seems entirely natural, consider how this appears:

```
u(1,m)     u(2,m)     ···   u(n,m)
u(1,m-1)   u(2,m-1)   ···   u(n,m-1)
  ⋮          ⋮         ⋮     ⋮
u(1,1)     u(2,1)     ···   u(n,1)
```

Viewed this way, the *rows* are stored together! What is going on?

The real story is that Fortran arrays are (always) stored so that, when we look at how the elements are placed in memory, we see that the first index varies most rapidly. In fact, the rule for mapping a Fortran array into memory is quite simple: If the array is declared as `A(N1,N2,...)`, then `A(I1,I2,...)` is the `(I1-1) + N1*((I2-1) + N2*(...))`th element (starting from zero).

The rule for C is the opposite; the *last* index varies most rapidly. When considering the special case of two dimensions, and looking at the arrays as representing matrices, we see how these lead to the row- and column-major interpretations.

E.1.3 Higher Dimensional Arrays

Once we know how arrays are laid out in the computer's memory, we can use that information to design ways to access sections of multidimensional arrays, planes with one coordinate constant, for example. If we wish to send a plane of data out of a 3-d array, we can form three different datatypes, one for each coordinate direction.

For concreteness, consider a Fortran array dimensioned as

```
double precision a(nx,ny,nz)
```

We will consider two different situations. In the first, we wish to send an entire face of this 3-D rectangular array. In the second, we wish to send only a rectangular part of the face. We will see that exploiting the knowledge of how data is laid out in memory will allow use to use a single `MPI_Type_vector` call for the first situation, while in the second situation (involving a part of a face), we will need to build a derived type from another derived type.

This is a good place to mention that most MPI implementations do not understand Fortran-90 array sections. That is, you should not pass a part of an array using, for example, `a(3:10,19:27,4)`. Instead, you should pass the first element that you wish to send; for example, `a(3,19,4)`. In some Fortran environments,

even this may not work. In that case, the MPI-2 standard [100], which covers the issues of MPI and Fortran in detail, should be consulted.

A prototype version of an MPI implementation that worked with HPF programs, including handling array sections, is described in [41].

Sending an Entire Face. To send the elements `a(1:nx,1:ny,k)` for some value of `k`, we can do not even need a datatype, since this selects `nx*ny` contiguous memory locations, starting at `a(1,1,k)`. However, it can be convenient to have a datatype for each face; we can construct the datatype for this face with

```
call MPI_TYPE_CONTIGUOUS( nx * ny, MPI_DOUBLE_PRECISION, &
                          newz, ierror )
```

The next face to send is `a(1:nx,j,1:nz)`. This is a vector: there are `nx` elements in each block, there are `nz` blocks, and the blocks are separated by a stride of `nx*ny`. The datatype representing full faces in the $x-z$ plane is

```
call MPI_TYPE_VECTOR( nz, nx, nx * ny, MPI_DOUBLE_PRECISION, &
                      newy, ierror )
```

Finally, consider the $y-z$ face `a(i,1:ny,1:nz)`. There are `ny*nz` elements, each of size 1, separated by `nx`. To see this, remember that the formula for the locations of the elements of `a(i,j,k)` is `offset + (i-1) + nx * ((j-1) + ny * (z-1))`. The value of j runs from 1 to ny, and k runs from 1 to nz. Thus the elements are

$$
\begin{aligned}
&\mathit{offset} + (i-1) + 0 \\
&\mathit{offset} + (i-1) + nx \\
&\quad \ldots \\
&\mathit{offset} + (i-1) + nx * (ny-1) \\
&\mathit{offset} + (i-1) + nx * (0 + ny * 1) \\
&\quad \ldots \\
&\mathit{offset} + (i-1) + nx * ((ny-1) + ny * (nz-1))
\end{aligned}
$$

Note that the element at `a(i,ny,k)` is `nx` elements from the element at `a(i,1,k+1)`, as of course are all the elements `a(i,j,k)` are `nx` elements from `a(i,j+1,k)`. Thus, we can use the vector type

```
call MPI_TYPE_VECTOR( ny * nz, 1, nx * ny, &
                      MPI_DOUBLE_PRECISION, newx, ierror )
```

These examples show the power of the `blockcount` argument in the MPI vector datatype creation calls.

Sending a Partial Face. If instead of sending the entire face of the cube, we want to send or receive the elements `a(sx:ex,sy:ey,k)` for some value of `k`, we can define a vector datatype `newz`:

```
call MPI_TYPE_VECTOR( ey-sy+1, ex-sx+1, nx, &
                      MPI_DOUBLE_PRECISION, newz, ierror )
call MPI_TYPE_COMMIT( newz, ierror )
call MPI_SEND( a(sx,sy,k), 1, newz, dest, tag, comm, ierror )
```

To understand this, we need only look at the discussion of arrays above. For the elements `a(sx:ex,sy:ey,k)`, we see that the data consists of `ey-sy+1` groups ("columns") of `ex-sx+1` elements ("rows"); the rows are contiguous in memory. The stride from one group to another (i.e., from `a(sx,j,k)` to `a(sx,j+1,k)`) is just `nx` double precision values. Note that this is an example of a vector type with a block count that is different from one.

Similarly, to send or receive the elements `a(sx:ex,j,sz:ez)` we can use

```
call MPI_TYPE_VECTOR( ez-sz+1, ex-sx+1, nx*ny, &
                      MPI_DOUBLE_PRECISION, newy, ierror )
call MPI_TYPE_COMMIT( newy, ierror )
```

The explanation for this is the same as for `a(sx:ex,sy:ey,k)` except that the stride between elements is `nx*ny` double precision values.

The final case, to send or receive the elements `a(i,sy:ey,sz:ez)` requires a little more work because we can not use the blocklength argument in `MPI_Type_vector`. In this case, we take advantage of MPI's ability to form a datatype from an MPI datatype. First, we for a datatype for the elements `a(i,sy:ey,k)`:

```
call MPI_TYPE_VECTOR( ey-sy+1, 1, nx, &
                      MPI_DOUBLE_PRECISION, newx1, ierror )
```

(We do not need to commit `newx1` because we will not be using it in a communication operation.) Next, we form a vector of these types. Since the stride between these elements is probably not an integral multiple of `ey-sy+1`, we use `MPI_Type_hvector`; this routine measures the stride in bytes.

```
call MPI_TYPE_EXTENT( MPI_DOUBLE_PRECISION, sizeof, ierror )
call MPI_TYPE_HVECTOR( ez-sz+1, 1, nx*ny*sizeof, &
                       newx1, newx, ierror )
call MPI_TYPE_COMMIT( newx, ierror )
```

An approach similar to that in Section 5.4 can be used to generate more general datatypes.