MPI Mechanic

June 2004

Provided by ClusterWorld for
Jeff Squyres
cw.squyres.com

ClusterWorld™
REDEFINING HIGH PERFORMANCE COMPUTING

www.clusterworld.com

BRIAN BARRETT

Return of the MPI Datatypes

Jeff is off this month, supposedly writing his dissertation. Personally, I think he's procrastinating — doing "research" in Disneyland, hiking the Himalayas, working on MPI-3 or some other *academic* endeavor. In the meantime, hello, I'm Brian — I'll be your host this month. We have lots of flavors on tap here at the House of MPI, including the new, Atkins-friendly, low-carb `MPI_TYPE_CREATE_RESIZED`.

A Quick Datatype Review

Last month we examined basic MPI datatypes. Datatypes provide necessary information to the MPI library about data format and location. As we saw last month, MPI provides both basic datatypes (`MPI_INT`) and the ability to create more advanced user-defined datatypes. MPI can use the type information to perform any format conversion, such as endian or size, necessary to communicate between two peers. Datatypes also simplify sending C structures or arrays of elements.

This month, we expand on our datatype coverage. Without knowledge of the basics of MPI datatypes, this month may be more difficult than the previous articles to follow. So find last month's magazine and read the basics of datatypes before getting started. In addition to performance benefits from letting the MPI do packing and unpacking, datatypes can simplify an application and help ensure messages are received correctly.

How to Avoid Datatypes

Despite last month's article and the remainder of this article, there are times where using user-defined datatypes are not the best option. Legacy applications may require explicitly buffers for sending, as was common with libraries before MPI. Data layout and size may be dynamic during execution of the application, which makes defining datatypes difficult. For these situations, MPI provides the ability to explicitly pack noncontiguous data into user provided buffers using `MPI_PACK`, with `MPI_UNPACK` for unpacking. Listing One shows an example of using `MPI_PACK` to send the structure used last month, rather than creating a matching type.

How MPI stores the data is implementation defined, including the ability to add internally useful meta-data. It is possible that the data added to `buf` is larger than the size of `struct my_struct`. `MPI_PACK_SIZE` is provided to determine the maximum buffer size needed to pack the given data.

Using `MPI_PACK` avoids determining offsets and creating datatypes. Packing allows the same buffer to be sent multiple times, reducing the workload on MPI. On the other hand, pack forces a memory copy into the user provided buffer when MPI may not have needed to do any packing. Packing is also error-prone, as MPI often does not check that data is unpacked in the same order it was packed.

Sending Columns of a Matrix

In C, sending a row of a matrix is easy, as the row is stored in consecutive bytes of memory. A column is more difficult, as the row must be traversed before arriving at the next element in the column. This space is often called the *stride*.

LISTING ONE

Building a Buffer Using MPI_Pack

```
 1 struct my_struct {
 2   int int_value[10];
 3   double average;
 4   char debug_name[MAX_NAME_LEN];
 5   int flag;
 6 };
 7 void send_data(struct my_struct data,
                 MPI_Comm comm, int rank) {
 8   char buf[BUFSIZE];
 9   int pos = 0;
10   MPI_Pack(&data.int_value, 10, MPI_INT,
             buf, BUFSIZE, &pos, comm);
11   MPI_Pack(&data.average, 1, MPI_DOUBLE,
             buf, BUFSIZE, &pos, comm);
12   MPI_Pack(&data.debug_name,MAX_NAME_LEN,
             MPI_CHAR, buf, BUFSIZE, &pos, comm);
13   MPI_Pack(&data.flag, 1, MPI_INT, buf,
             BUFSIZE, &pos, comm);
14   MPI_Send(buf, pos, MPI_PACKED, rank, 0, comm);
15 }
```

Without user-defined datatypes, there are two ways to send a column to another process: send each element individually or pack the elements into an array by hand. The code below shows how to avoid the hassle by creating an MPI datatype.

```
double buf[10][12];
MPI_Datatype column;
MPI_type_vector(10,
                1, 12,
                MPI_DOUBLE,
                &column);
MPI_Type_commit(&column);
MPI_Send(buf[2], 1,
         column, 0,
         0, MPI_COMM_WORLD);
```

In the listing above, the type is committed and immediately used. Once committed, the datatype can be reused throughout the program. By adjusting the index in the `MPI_SEND`, any column in the matrix can be sent. Not only is the number of lines of code required to send a column using user-defined datatypes smaller than if packed the buffer by hand, an MPI implementation has the option to avoid packing the data before sending. Some communication channels allow "vectored sends," meaning the ability to send from many data locations and receive into many data locations.

Why not MPI_BYTE

MPI provides the datatype `MPI_BYTE` to represent a byte of memory. The MPI implementation will not perform any datatype conversion on the buffer. So why not use `MPI_BYTE` and avoid all the complexity of datatypes?

Using `MPI_BYTE` prevents MPI from performing any data conversion (as discussed in last month's article). Data padding and alignment issues, normally completely hidden from the user, must be taken into account. In C, this is generally not a problem because C programmers are used to dealing with padding issues. However, Fortran generally does a good job of handling padding and alignment behind the scenes. Using `MPI_BYTE` forces dangerous assumptions about the sizes of various datatypes.

LISTING TWO

Using Parts of a Structure

```
 1 struct vehicle {
 2   double position[3];
 3   double destination[3];
 4   double velocity[3];
 5   double fuel;
 6 }
 7 struct vehicle cars[10];
 8 MPI_Datatype tmp_car_type, car_type;
 9 int i, counts[2]={ 3, 3 };
10 MPI_Datatype types[2]={ MPI_DOUBLE, MPI_DOUBLE };
11 MPI_Aint disps[2];
12 MPI_Address(&cars[0].position, &disps[0]);
13 MPI_Address(&cars[0].velocity, &disps[1]);
14 disps[1] -= disps[0];
15 MPI_Type_struct(2, counts, disps, types,
                  &tmp_car_type);
16 MPI_Type_create_resized(tmp_car_type, 0,
                          sizeof(struct vehicle),
                          &car_type);
16 MPI_Type_commit(&car_type);
...
17 MPI_Send(cars, 10, car_type, ...);
```

Send Only What Is Needed

Thus far, we have looked at ways to send simple datatypes, an entire matrix, parts of a matrix, and an entire structure. It is also possible to send only part of a structure. Listing Two provides an example of sending selected elements of a structure using datatypes. For example, in a simple traffic simulation, a local vehicle may only need to know the position and velocity of a remote vehicle. Locally, fuel and destination are also tracked.

The basics of creating the `tmp_car_type` datatype should look familiar from last month's article. The calls to `MPI_ADDRESS` are used to find the relative displacements between `position` and `velocity`. Of course, there is some space between the two arrays for the `destination` field. MPI will only send the two fields desired, ignoring the empty space. As long as `MPI_ADDRESS` is used to find offsets, holes in the middle of a structure definition will be taken care of "magically" by MPI.

The call to `MPI_TYPE_CREATE_RESIZED` is new to this example and is used to fill in the gaps for MPI. Based on the information given to MPI when `tmp_car_type` is created, the structure appears to end after the final element of the `velocity`

field. If an array of `tmp_car_type` is sent, MPI will look for the start of the second structure directly after the last element of the velocity array (accounting for padding, of course). Instead of finding the first element of the position array for the second structure, it will find the `fuel` element from the first struct. `MPI_TYPE_CREATE_RESIZED` provides the lower bound and extent of the datatype.

Both terms will be discussed in the next section, but the extent in this example is the true size of the vehicle structure.

Lower Bounds, Upper Bounds, and Extents

The vehicle example introduced one of the most confusing parts of datatypes: bounds and extents. Every datatype has a lower bound, upper bound, and extent. The lower bound is the offset from the start of the user buffer to the start of the first datatype entry for the buffer. In the vehicle example above, if the `fuel` entry was first instead of last, MPI would need to know that it should skip over the `fuel` entry to find the `destination` entry. In this case, the lower bound would be `sizeof(double)`. `MPI_ADDRESS` can be used to compute the lower bound, similar to how offsets between datatype entries are found. The lower bound can either be adjusted using `MPI_TYPE_CREATE_RESIZED` or `MPI_TYPE_LB`.

The upper bound is end of the last element in a datatype, plus any required padding. In any array, the next entry begins directly after the upper bound of the current entry. The extent is the size of the datatype, or the upper bound minus the lower bound.

Although the datatype's upper bound can be set using `MPI_TYPE_UB`, it is often much easier and less error-prone to set the extent using `MPI_TYPE_CREATE_RESIZED`.

One-off Datatypes

In each of the datatype examples presented thus far, an instance of a structure is used to determine addresses of each element in the datatype. The addresses are then used to determine offsets to use in the datatype. The resulting datatype can be used to describe any instance of the same structure. However, there are some instances where a "one-off" datatype is created to describe a structure that will only exist once. In these cases, determining addresses to find offsets, only to use the offsets to recompute the addresses is wasteful.

MPI provides the constant `MPI_BOTTOM` to for instances where computing offsets will be wasteful.

```
MPI_Send(MPI_BOTTOM, 5,
  custom_type,...)
```

The MPI will still have to do some offset math in order to find the elements in the entire array. `MPI_BOTTOM` can be tempting, as it saves a couple of lines of code. However, `MPI_BOTTOM` should generally be avoided. One of the advantages of datatypes is that they can be reused to avoid errors in user applications. If absolute addresses are used with `MPI_BOTTOM`, it is not possible to reuse the datatype in a generic way.

Resources

MPI Forum (including the MPI-1 and MPI-2 specification documents): *www.mpi-forum.org*

NCSA MPI tutorial: *webct.ncsa.uiuc.edu:8900/public/MPI*

Common Pitfalls and Misconceptions

One common misconception with MPI datatypes is that they are slow. Early in the life of MPI, using MPI datatypes to pack messages was often slower than packing the data by hand. Datatype performance has been and continues to be an active area of research, allowing datatype implementations to achieve much higher performance. Some MPI implementations are even capable of doing scatter/gather sends and receives, completely eliminating the need to pack messages for transfer. In short, poor datatype performance is generally a thing of the past, and it's getting better every day.

MPI provides a huge, often overwhelming, number of options when working with datatypes. Although it is often tempting to use the predefined datatypes and avoid complexity, proper use of datatypes can reduce errors and improve performance. Using a complex datatype removes the problem of ensuring the correct order of sends and receives to move a structure piecemeal.

Where to Go From Here?

This column provides a number of examples of using datatypes to their full potential. The resources listed in the side bar present even more examples of utilizing datatypes to simplify applications. Next month, we will move on to any implementor's favorite subject: common mistakes in using MPI and how to avoid them.

Brian Barrett is a parallel systems analyst at the Information Sciences Institute, University of Southern California and a developer on the LAM implementation of MPI. He can be reached at brbarret@lam-mpi.org