MPI Mechanic

May 2004

Provided by ClusterWorld for
Jeff Squyres
cw.squyres.com

www.clusterworld.com

JEFF SQUYRES

How to Succeed in Datatypes Without Really Trying

A novice asked the master: "I have a program that sometimes runs and sometimes aborts. I have followed the rules of programming, yet I am totally baffled. What is the reason for this?"

The master replied: "You are confused because you do not understand MPI. The rules of programming are transitory; only MPI is eternal. Therefore you must contemplate MPI before you receive enlightenment."

"But how will I know when I have received enlightenment?" asked the novice.

"Your program will then run correctly," replied the master.

The Story So Far

This month we'll discuss MPI datatypes. As the name implies, datatypes are used to represent the format and layout of the data that is being sent and received. MPI datatypes are extremely flexible — so much so that they are likely to be quite confusing to the beginner. When properly used, MPI datatypes can both provide good communications performance as well as reduce application complexity.

Typed Messages

Although MPI allows untyped messages, most applications utilize *typed* messages, meaning that the MPI implementation is aware of the format and layout of the message being sent or received.

Specifically, a message is simply a dense sequence of zeros and ones. Only when it is assigned a *format* and *layout* at its source or destination buffer does the sequence have any meaning. MPI allows the structure of the buffer to be arbitrary. A buffer is described with a *type map* — an abstract sequence of types paired with corresponding displacements that describes the format and layout of a buffer. A *datatype* is an instance of a type map. Since messages are typically described with stream-like qualities, they are described with a *type signature* — a sequence of datatypes (but no displacements). Hence, a given type signatures may correspond to many different datatypes (type maps).

Putting it all together: a buffer with a given type map is sent with a corresponding datatype, creating a message with a corresponding type signature. The receiver provides a buffer and datatype (implying a specific type map) which directly corresponds to the type signature that the will be used to place the message in the target buffer.

Message Data Format

The canonical example of formatting differences is the "endian" problem, which refers to the order in which bits are stored in memory for multi-byte values (e.g., a four byte integer). So-called "big endian" systems write the least significant byte (lsb) at the highest memory position; "little endian" systems write the lsb at the lowest memory position. Listing One is a sample C program that shows the difference.

The program is fairly simple — assign a value into an unsigned integer and the display the actual values stored in memory (assuming a four-byte integer). Running this program on an Intel x86 machine

LISTING ONE

Showing Data Format in Memory

```
1 #include <stdio.h>
2 int main(int argc, char* argv[]) {
3   unsigned int i = 258;
4   unsigned char *a = (unsigned char *) &i;
5   printf("%x %x %x %x\n", a[3], a[2], a[1], a[0]);
6   return 0;
7 }
```

LISTING TWO

Building a 2D Matrix Datatype

```
1 double my_array[10][10];
2 MPI_Datatype row, array;
3 MPI_Type_vector(1, 10, 1, MPI_DOUBLE, &row);
4 MPI_Type_commit(&row);
5 MPI_Type_vector(1, 10, 1, row, &array);
6 MPI_Type_commit(&row);  <-- This should be &array
7 /* ...fill my_array... */
8 MPI_Bcast(my_array, 1, array, 0, MPI_COMM_WORLD);
```

(little endian) results in: "0 0 1 2", whereas running this program on a Sun SPARC machine (big endian) results in: "2 1 0 0".

MPI implementations capable of handling heterogeneous environments will automatically perform endian translation of typed messages. For example, when sending the integer value of 1,234 from an MPI process on a Sun SPARC, if it is received on a corresponding MPI process on an Intel x86 machine, MPI will do the appropriate byte swapping to ensure that the received value is 1,234.

Another data format issue is the size of a given type. The C type `double`, for example, may have different sizes on different machines (e.g., four bytes or eight bytes). Indeed, sometimes even different compilers on the same machine will have different data sizes for the same type. The behavior of an MPI implementation when faced with mismatched data sizes varies; most implementations will either upcast/truncate (depending on whether small data is being received into a large buffer or vice versa) or abort.

Watch Out For That Pointer, Eugene!

Care should be taken when sending pointer values from one process to another. Although MPI will ensure to transport the pointer value correctly to the target process (it's just an integer, after all), it may have no meaning at the destination since pointers in one process' virtual memory may have no relation to addresses in another process' memory.

There are limited scenarios where this is useful (e.g., echoing pointers back in ACKs), but consider yourself warned: be very sure of what you are doing when sending pointers between processes.

Simple Data Format Example

An MPI message is described in terms of number of elements and the type of each element (as opposed to total number of bytes). MPI contains pre-built definitions of many datatypes intrinsic to C, C++, and Fortran. These datatypes can be used to send and receive both variables and arrays. For example:

```
MPI_Status status;
int values[10];
MPI_Recv(value,
  10,
  MPI_INT,
  src,
  tag,
  comm,
  &status);
```

This code fragment will receive ten integers into the `values` array. Any endian differences will be automatically handled by MPI (if the implementation is capable of it).

Message Data Layout

Although messages are dense streams of zeros and ones, the buffers where they originate and terminate need not be contiguous — they may not even share the same data format and layout. Specifically, the type maps used on the sender and receiver may be different — as long as the type signatures used to send and receive the message are the same, the data will be transferred properly.

This allows the sending and receiving of arbitrary data structures from one process to another. When used properly, this flexibility can dramatically reduce overhead, latency, and application complexity.

For example, consider a C structure that contains a `int` and an `double`. Assume that both sender and receiver have the same sizes for both types, but the sender aligns `double` on eight byte boundaries

LISTING THREE

Building an MPI Datatype for a C Structure

```
1 struct my_struct {
2   int int_values[10];
3   double average;
4   char debug_name[MAX_NAME_LEN];
5   int flag;
6 };
7 void make_datatype(MPI_Datatype *new_type) {
8   struct my_struct foo;
9   int i, counts[4] = { 10, 1, MAX_NAME_LEN, 1 };
10  MPI_Datatype types[4] = { MPI_INT, MPI_DOUBLE,
    MPI_CHAR, MPI_INT };
11  MPI_Aint disps[4];
12  MPI_Address(foo.int_values, &disps[0]);
13  MPI_Address(&foo.average, &disps[1]);
14  MPI_Address(foo.debug_name, &disps[2]);
15  MPI_Address(&foo.flag, &disps[3]);
16  for (i = 3; i >= 0; --i)
17    disps[i] -= disps[0];
18  MPI_Type_struct(4, counts, disps, types, new_type);
19  MPI_Type_commit(new_type);
20 }
```

and the receiver aligns `double` on four byte boundaries. The overall size of the structure will be different between the two, as will the placement of the data in memory — even though the values will be the same. When used with appropriate datatypes, MPI will automatically read the message from the source buffer using the sender's layout and write it to the destination buffer using the destination's layout.

Vector Layout Example

MPI provides functions for building common types of data representation: contiguous data, vectors, and indexed data. Listing Two shows making a nested vector datatype that describes a 2D matrix.

Line 3 builds a vector to describe a single row — this is 1 set of 10 `double` instances with a stride of 1. Line 5 builds a second datatype describing 1 set of 10 rows with a stride of 1 — effectively the entire 2D array. Note that MPI requires *committing* a datatype with `MPI_TYPE_COMMIT` before it can be used (lines 4 and 6). Once the `array` datatype is committed, it is used to broadcast the array on line 8. The `array` type can also be used with any `other` communication function, such as `MPI_SEND`, `MPI_RECV`, etc.

Indexed datatypes can be built with the `MPI_TYPE_INDEXED` function (not described here).

C Structure Layout Example

Although the MPI vector and index interfaces can build complex and useful datatypes, by definition, they cannot describe arbitrary data layouts. The `MPI_TYPE_STRUCT` function allows the specification of arbitrary type maps.

The use of `MPI_TYPE_STRUCT` is admittedly fairly clunky — it is necessary to specify arrays of field offsets, counts, and datatypes. Listing Three shows a program constructing an MPI datatype for a complex C structure (note that several shortcuts were taken for space reasons).

Note the use of the type `MPI_Aint` on line 11. An `MPI_Aint` is defined as an integer guaranteed to be large enough to hold an address. C programmers are likely to be confused by lines 12-15: the `MPI_ADDRESS` function is equivalent to assigning the address of the first argument to the second arguments. The purpose of having `MPI_ADDRESS` is twofold: 1) avoid ugly casting, and 2) provide "pointer-like" semantics in Fortran. Lines 16-17 are intended to subtract off the base address of the structure, effectively converting the `disps` array to hold relative displacements instead of absolute addresses.

The `MPI_TYPE_STRUCT` call on line 18 creates a datatype from the type map components, and it is committed on line 19. The `new_type` datatype can now be used to send instances of `struct my_struct`.

One Message vs. Many Messages

A common knee-jerk reaction to the complexity of MPI datatypes is "I'll just send my structure in a series of messages rather than construct a datatype." Consider the C structure show in Listing Two.

The contents of `my_struct` could easily be sent in four separate messages. But consider the actual cost of sending four messages instead of constructing an MPI datatype and sending one message: although the same amount of data will be sent, you'll likely be sending at least four times the overhead for the same amount of data, causing the overall efficiency ratio to drop. Specifically, most MPI implementations send a fixed amount of overhead for each message. Hence, you'll be sending that fixed overhead three more times than is necessary. Additionally, you may incur up to four times the latency before the entire dataset is received at the destination.

If this structure is sent frequently in your application (e.g., once every iteration), or if you have to send arrays of this structure, the extra overhead and latency can quickly add up and noticeably degrade performance.

Instead, it is frequently better to make an appropriate datatype and send the entire structure in a single message. Indeed, entire arrays of the structure can also be sent in a single message, dramatically increasing efficiency as compared to sending four messages for each structure instance in the array.

Resources

The Tao of Programming
by Geoffrey James. ISBN 0931137071.

MPI Forum
- http://www.mpi-forum.org/

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

Where to Go From Here?

This column has really only touched on some parts of MPI datatypes — there's more to MPI datatypes than meets the eye. Stay tuned for next month's column where we'll discuss more features and issues with datatypes.

Jeff Squyres is a research associate at Indiana University and is the lead developer for the LAM implementation of MPI. Reach him at jsquyres@lam-mpi.org