Advanced Message Passing in MPI

Using MPI Datatypes with Opaque C++ Types

Paul Preney, OCT, M.Sc., B.Ed., B.Sc.

preney@sharcnet.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

September 17, 2014

Abstract

When one is using arrays of fundamental types such as doubles, using MPI is reasonably straight-forward. When one needs to use MPI to transmit complicated data structures, pointers, and other opaque types whose internals may be not known by the programmer, using MPI becomes significantly more difficult. The MPI standard has facilities to dynamically define new message types in order to pass such between nodes using `MPI_Datatype` along with a number of functions to register and deregister such types. This talk will introduce how to properly use `MPI_Datatype` to transmit non-trivial, custom opaque data structures between MPI nodes using C++. Since using such MPI calls is rather low-level, the talk will also introduce how to exploit the features of C++ to more easily accomplish the same at a higher-level.

Presentation Overview

Table of Contents

What is MPI?

Message-**P**assing **I**nterface:

- is a *de facto* standard dating back to 1994. [3]
- is used to write portable code for parallel computers within a distributed memory context.
- has language bindings for Fortran and C.
  - **NOTE:** The C++ language bindings were removed in MPI v3.0. [6, §16.2, p.596]
- enables compute nodes to efficiently pass messages to one another.

History of MPI Features

Briefly these are the features associated with each version of the MPI standard:

- **v1.x** [4]
  - Supports two-way communications: point-to-point, broadcast, reduce, scatter, gather, etc.
  - Supports “Derived Datatypes” which enable nodes to define at run-time the the structure of messages sent and/or received.
- **v2.x** [5]
  - Added one-sided communications (put, get, and accumulate) and synchronization methods.
  - Added the ability to spawn new processes at run-time.
  - Added parallel I/O support.
- **v3.0** [6]
  - Added Fortran 2008 bindings.
  - Added new one-sided communication operations.
  - Extended support for non-blocking collectives.

MPI Derived Datatypes

The focus of this talk is on using **MPI Derived Datatypes** with **message-passing** operations. [4, §3.12] [5, §4] [6, §4]

Without loss of generality the only operations we will be concerned with are `MPI_Send()` and `MPI_Recv()`. [4, §3]

- Know that all communications operations in MPI also have an `MPI_Datatype` argument.

Also without loss of generality, all of the MPI code in this talk will assume the sender is node 0 and the receiver is node 1.

- You are free and encouraged to use more nodes in your programs!

MPI_Send

`MPI_Send(buf,count,type,dest,tag,comm)` is a blocking send operation whose arguments are defined as follows: [6, §3]

Argument	In/Out	Description
buf	IN	starting address of send buffer
count	IN	number of elements in send buffer
type	IN	`MPI_Datatype` of each send buffer element
dest	IN	node rank id to send the buffer to
tag	IN	message tag
comm	IN	communicator

When called, `MPI_Send` transmits `count` elements in `buf` all of type `type` to node `dest` with the label `tag`.

The buffer is assumed to have been sent after the call returns.

MPI_Recv

`MPI_Recv(buf,count,type,src,tag,comm,status)` is a blocking receive operation whose arguments are defined as follows: [6, §3]

Argument	In/Out	Description
buf	OUT	starting address of receive buffer
count	IN	number of elements in receive buffer
type	IN	`MPI_Datatype` of each buffer element
src	IN	node rank id to receive the buffer from
tag	IN	message tag
comm	IN	communicator
status	OUT	status object

When called, `MPI_Recv` receives up to `count` elements in `buf` all of type `type` from node `src` with the label `tag`.

Up to `count` buffer elements can be stored.

Table of Contents

MPI_Datatype

MPI uses instances of a special type called `MPI_Datatype` to represent the types of messages being sent or received.

The MPI standard defines a set of predefined `MPI_Datatype`s that map to C's fundamental types as well as Fortran types. Some of these mappings for C are: [6, §3.2]

MPI_Datatype Name	C Type
`MPI_C_BOOL`	**`_Bool`**
`MPI_CHAR`	**`char`** (treated as text)
`MPI_UNSIGNED_CHAR`	**`unsigned char`** (treated as an integer)
`MPI_SIGNED_CHAR`	**`signed char`** (treated as an integer)
`MPI_INT`	**`signed int`**
`MPI_DOUBLE`	**`double`**
`MPI_LONG_DOUBLE`	**`long double`**
`MPI_C_DOUBLE_COMPLEX`	**`double _Complex`**

Registering New MPI_Datatypes

One can register new `MPI_Datatype`s using any of the functions described in [4, §3.12], [5, §4], and [6, §4]. Of these, these are the most important in this presentation:

Function	Purpose	Memory Organization
`MPI_Type_commit`	registers type	n/a
`MPI_Type_free`	deregisters type	n/a
`MPI_Type_create_struct`	makes new type	like a C **`struct`**

Use: First *register* the new `MPI_Datatype`, then *commit* it so it can be used, and when done, *deregister* it to free its associated resources.

MPI_Type_commit and MPI_Type_free

`MPI_Type_commit(type)` registers `type` so that it can be used with MPI communications functions.

`MPI_Type_free(type)` deregisters `type` when it no longer needs to be used with MPI communications functions.

MPI_Type_create_struct

`MPI_Type_create_struct(count,blocklens,displacements,types,newtype)` constructs a new `MPI_Datatype` instance whose memory representation is a sequence of blocks where:

- each block has a corresponding length provided in the array `blocklens`,
- each block has a corresponding displacement from the startng address of the buffer provided in the array `displacements`,
- each block has a corresponding `MPI_Datatype` provided in the array `types`,

The new `MPI_Datatype` is stored in `newtype`.

MPI_Type_create_struct (con't)

```
struct simple { int i; double d[3]; } v;

constexpr std::size_t num_members = 2;
int lengths[num_members] = { 1, 3 };
MPI_Aint offsets[num_members] = {
  offsetof(simple, i), offsetof(simple, d) };
MPI_Datatype types[num_members] = { MPI_INT, MPI_DOUBLE };
MPI_Datatype simple_type;
MPI_Type_struct(num_members, lengths, offsets, types,
  simple_type);
MPI_Type_commit(simple_type);

// In sender on node 0...
MPI_Send(&v, 1, simple_type, 1, 0, MPI_COMM_WORLD);

// In receiver on node 1...
MPI_Status s;
MPI_Recv(&v, 1, simple_type, 0, 0, MPI_COMM_WORLD, &s);
```

Table of Contents

A Problem!

Many types are opaque in terms of their memory layouts.

- Do you *really* know the exact memory layout of a given `struct`, `class`, or `union`?
- If not then you cannot pass the *address-of* a `struct`, `class`, or `union` variable to an MPI C call that assumes a specific memory layout!

A Problem! (con't)

Many types don't have “standard layout”.

- Standard layout is required to meaningfully pass `struct`, `class`, and `union` variables to other languages by relying on its memory layout.
- A type is **not** in *standard layout* if:
  - it has non-`static` members that are not in *standard layout*,
  - it has one or more `virtual` functions,
  - it has `virtual` base classes,
  - it has non-*standard layout* base classes,
  - it has more than one type of access control (e.g., `public`, `protected`, `private`) for data members, and,
  - some other conditions.

The term *standard layout* is defined in the C++ standard. [2, §9].

A Problem! (con't)

MPI calls require knowledge of variables' memory layouts.

- These calls are incompatible with non-standard layout types.

MPI calls do not support variable-length objects except for arrays.

- So how can one easily send and receive variables with types like `std::string`, `std::vector<std::string>`, etc.?

Unsure About Standard Layout?

This C++11 code can be used determine if a type has standard layout:

```
// With g++ use -std=c++11 option.
#include <iostream>
#include <type_traits>

struct A { int i; double d[3]; };
struct B { public: int i; private: double d[3]; };

int main()
{
  std::cout
    << "A:␣" << std::is_standard_layout<A>::value << '\n' // 1
    << "B:␣" << std::is_standard_layout<B>::value << '\n' // 0
  ;
}
```

Handling Variable-Length Objects

Only using the MPI functions previously discussed, there is a simple way to handle variable-length objects:

- Create a **`struct`** with an integer member representing the length that precedes the variable-length object.

This allows one now to easily send/receive those objects:

1. First send/receive the length.
2. If receiving ensure there is sufficient space to hold the object.
3. Finally send/receive the string data.

Let's consider `std::string`...

Handling std::string

Conceptually this is the type needed to be registered with MPI to handle `std::string`:

```
// For conceptual purposes only...
struct mpi_sendrecv
{
  unsigned length_;
  char str_[length_];
};
```

However this is not needed since MPI already can handle an array of **`char`**!

Handling std::string (con't)

To send a `std::string`, this is all that is needed:

```
void send(
  std::string const& str,
  int dest, int tag, MPI_Comm comm
)
{
  unsigned len = str.size();
  MPI_Send(&len, 1, MPI_UNSIGNED, dest, tag, comm);
  if (len != 0)
    MPI_Send(str.data(), len, MPI_CHAR, dest, tag, comm);
}
```

Handling std::string (con't)

Receiving a `std::string` is trickier since `std::string` has no member function that returns a non-const `char` array.

Instead use a `std::vector<char>` as a receiving area and then copy that into the `std::string`:

```
void recv(std::string& str, int src, int tag, MPI_Comm comm)
{
  unsigned len;
  MPI_Status s;
  MPI_Recv(&len, 1, MPI_UNSIGNED, src, tag, comm, &s);

  if (len != 0) {
    std::vector<char> tmp(len);
    MPI_Recv(tmp.data(), len, MPI_CHAR, src, tag, comm, &s);
    str.assign(tmp.begin(), tmp.end());
  } else
    str.clear();
}
```

SHARCNET™

Handling std::vector

If what is stored in `std::vector` is a fundamental type, then the code is almost identical to `std::string`. The send code is:

```
void send(
  std::vector<int> const& vec,
  int dest, int tag, MPI_Comm comm
)
{
  unsigned len = vec.size();
  MPI_Send(&len, 1, MPI_UNSIGNED, dest, tag, comm);
  if (len != 0)
    MPI_Send(vec.data(), len, MPI_INT, dest, tag, comm);
}
```

Handling std::vector (con't)

and the receive code is:

```
void recv(std::vector<int>& vec, int src, int tag, MPI_Comm comm)
{
  unsigned len;
  MPI_Status s;
  MPI_Recv(&len, 1, MPI_UNSIGNED, src, tag, comm, &s);

  if (len != 0) {
    vec.resize(len);
    MPI_Recv(vec.data(), len, MPI_INT, src, tag, comm, &s);
  } else
    vec.clear();
}
```

However when what is stored is *not* a fundamental type, one *may* want the type to be registered.

Registering Standard Layout Types

Just as one can create new types in C and C++ using `struct`, `class`, or `union`, MPI permits the definition of new **derived datatypes** [6, §4] for messages.

Suppose one needs to handle messages in the form of this *fixed-length standard layout* structure:

```
struct example
{
  int x;
  int y;
  double vec[3];
};
```

Registering Standard Layout Types (con't)

The `example` structure can be registered as follows:

```
#include <cstddef> // For offsetof macro

MPI_Datatype register_mpi_type(example const&) {
  constexpr std::size_t num_members = 3;
  int lengths[num_members] = { 1, 1, 3 };

  MPI_Aint offsets[num_members] = { offsetof(example, x),
    offsetof(example, y), offsetof(example, vec) };
  MPI_Datatype types[num_members] = { MPI_INT, MPI_INT,
    MPI_DOUBLE };

  MPI_Datatype type;
  MPI_Type_struct(num_members, lengths, offsets, types, &type);
  MPI_Type_commit(&type);
  return type;
}
```

Registering Standard Layout Types (con't)

Thus given a deregistration function:

```
void deregister_mpi_type(MPI_Datatype type)
{
  MPI_Type_free(&type);
}
```

Registering Standard Layout Types (con't)

One can now easily write a send function:

```
void send(
  example const& e,
  int dest, int tag, MPI_Comm comm
)
{
  MPI_Datatype type = register_mpi_type(e);
  MPI_Send(&e, 1, type, dest, tag, comm);
  deregister_mpi_type(type);
}
```

Registering Standard Layout Types (con't)

and a receive function:

```
void recv(
  example const& e,
  int src, int tag, MPI_Comm comm
)
{
  MPI_Status s;
  MPI_Datatype type = register_mpi_type(e);
  MPI_Recv(&e, 1, type, src, tag, comm, &s);
  deregister_mpi_type(type);
}
```

Registering Standard Layout Types (con't)

Which allows one to easily handle sending `std::vector<example>`s:

```
void send(
  std::vector<example> const& ve,
  int dest, int tag, MPI_Comm comm
)
{
  unsigned len = ve.size();
  MPI_Send(&len, 1, MPI_UNSIGNED, dest, tag, comm);

  if (len != 0) {
    MPI_Datatype type = register_mpi_type(&ve[0]);
    MPI_Send(ve.data(), len, type, dest, tag, comm);
    deregister_mpi_type(type);
  }
}
```

Registering Standard Layout Types (con't)

and receiving `std::vector<example>`s:

```
void recv(
  std::vector<example> const& ve,
  int src, int tag, MPI_Comm comm
)
{
  unsigned len; MPI_Status s;
  MPI_Recv(&len, 1, MPI_UNSIGNED, src, tag, comm, &s);

  if (len != 0) {
    ve.resize(len);
    MPI_Datatype type = register_mpi_type(&ve[0]);
    MPI_Recv(ve.data(), len, type, src, tag, comm, &s);
    deregister_mpi_type(type);
  } else
    ve.clear();
}
```

Table of Contents

Switching Containers

There were reasons an explicit length was sent and received in the previous examples:

- It allows all *variable-length, homogeneous container types* to have the *identical* send-receive message structure.
- Since their message structures are identical, the send and receive data container types *don't have to match:* they only need to contain the same type.

Suppose one sends a `std::vector<example>`.

Now consider how one might receive it into a `std::list<example>`...

Receiving std::list<example>

One method to send a `std::list<example>` is:

```
void recv(
  std::list<example> const& le,
  int src, int tag, MPI_Comm comm
)
{
  // Receive everything into a vector...
  std::vector<example> tmp;
  recv(tmp, src, tag, comm);

  // And assign it to the list...
  le.assign(tmp.begin(), tmp.end());
}
```

Receiving std::list<example> (con't)

Here's another (exception unsafe wrt MPI comm.) method:

```
void recv(
  std::list<example> const& le,
  int src, int tag, MPI_Comm comm
)
{
  unsigned len; MPI_Status s;
  MPI_Recv(&len, 1, MPI_UNSIGNED, src, tag, comm, &s);

  if (len != 0) {
    example tmp;
    for (unsigned i=0; i != len; ++i) {
      recv(tmp, src, tag, comm);
      le.push_back(tmp);
    }
  } else
    le.clear();
}
```

Sending std::list<example>

Similarly here's an method to send a `std::list<example>`:

```
void send(
 std::list<example> const& le,
 int dest, int tag, MPI_Comm comm
)
{
 unsigned len = le.size();
 MPI_Send(&len, 1, MPI_UNSIGNED, dest, tag, comm);

 for (
  std::list<example>::const_iterator i=le.begin(),
   iEnd=le.begin();
  i != iEnd;
  ++i
 )
  send(*i, dest, tag, comm);
}
```

Sending std::list<example> (con't)

And an alternate method:

```
void send(
  std::list<example> const& le,
  int dest, int tag, MPI_Comm comm
)
{
  // Copy everything into a vector...
  std::vector<example> tmp(le.begin(), le.end());

  // and send it...
  send(tmp, dest, tag, comm);
}
```

Sending std::list<example> (con't)

Q. Does MPI allows one to send something as a single send operation and receive it component-by-component using multiple receive operations?

A. Yes! The received parts must match the definition of the whole send.

Sending std::list<example> (con't)

Q. Does MPI allows one to send something component-by-component using multiple send operations and to receive it as a single receive operation?

A. Yes! The sent parts must match the definition of the whole receive.

NOTE: This is effectively what allows the `std::list<example>` functions sending/receiving element-by-element to be able to interoperate with the earlier `std::vector<example>` functions!

Handling Opaque Types

An opaque type is a type where the memory layout is **not** known.

All that can be done is either:

- define and register a suitable `struct` to send/receive such, or,
- send/receive all object state component-wise.

Examples include the earlier codes handling `std::string`, `std::vector`, and `std::list`.

- *Thinking* the layout is X is *not* the same as the documentation for such saying it is!
- Even with `std::array` you must call `.data()` to access (the documented part) of its internal layout.
- `std::string`, `std::array`, and `std::vector` are all "special" in the sense the C++ standard defines the layout with in terms of what `.data()` returns.

SHARCNET™

Table of Contents

Closing Advice

Advice:

- It is better to have slower correct code than fast incorrect code.
- Always remember in C++ exceptions can be thrown.
  - Where appropriate use the RAII (Resource Acquisition is Initialization) design pattern to ensure resources are cleaned up if an exception occurs. [7, §5.2, §13.3] [8, §19.5]
- Design your code to be exception-safe with respect to non-atomic MPI communications.
  - You don't want a node waiting for data that will *never* be sent because an exception occurred!
- Write higher-level, possibly overloaded functions to make it easier to handle all types —not just opaque ones!
  - e.g., `send()` and `recv()` in this presentation.

Closing Comments

If you are writing code using `MPI_Datatype` it is worth downloading and reading the appropriate sections in the appropriate MPI standard. [4, §3.12] [5, §4] [6, §4]

Boost's MPI library provides a high-level interface to `MPI_Datatype`. [1]

- Boost.MPI internally uses MPI's `MPI_PACK` to send and receive data.

How to use `MPI_PACK` was not discussed in this presentation.

- If you are curious about this, read the appropriate MPI standard's section on "Derived Datatypes". [4, §3.12] [5, §4] [6, §4]

Questions & Thank You

Questions.

Thank you for attending this presentation!

References

[1] Boost.org, ed. *Boost.MPI*. 2014-09-13. URL: http://www.boost.org/doc/libs/1_56_0/doc/html/mpi.html.

[2] ISO/IEC. *Information technology – Programming languages – C++*. ISO/IEC 14882-2011, IDT. Geneva, Switzerland, 2012.

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 1.0. 1994-05-05. URL: http://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps.

[4] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 1.3. 2008-05-30. URL: http://www.mpi-forum.org/docs/mpi-1.3/mpi-report-1.3-2008-05-30.pdf.

References (con't)

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 2.2. 2009-09-04. URL: `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 3.0. 2012-09-21. URL: `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[7] B. Stroustrup. *The C++ Programming Language*. 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2013, p. 1346.

[8] B. Stroustrup. *Programming: Principles and Practice Using C++*. 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2014, p. 1274.