Creating MPI Datatypes

Why create your own MPI datatype?

- When using MPI it is best to do as few communications as possible
  - Rather send all required data in a single large communication than in a number of smaller communications
- If the data is continuous and all of the same type (e.g. an array) this is easy to achieve
- Often you want to send discontinuous information and/or variables of a mixture of types (e.g. only some of the member variables in an object)
- We can achieve this by making our own MPI variable types to encompass a number of simpler MPI types (or even other types that we have created ourself)

Demonstrate with an example

- Creating an MPI variable to send some of the member variables within an object

```
#include <mpi.h>
#include <iostream>
#include <locale>

using namespace std;

int id, p;

class my_class
{
public:
	int I1, I2;
	int var_not_to_send;
	double D1;
	/*Note that I am using a C string rather than, e.g.std::string
	This is because the data in std::string is dynamically allocated and
	does not have a fixed relationship to the rest of the data in the class*/
	char S1[50];

	static void buildMPIType();
	static MPI_Datatype MPI_type;
};

MPI_Datatype my_class::MPI_type;
```

```
void my_class::buildMPIType()
{
	int block_lengths[4];
	MPI_Aint displacements[4];
	MPI_Aint addresses[4], add_start;
	MPI_Datatype typelist[4];

	my_class temp;

	typelist[0] = MPI_INT;
	block_lengths[0] = 1;
	MPI_Get_address(&temp.I1, &addresses[0]);

	typelist[1] = MPI_INT;
	block_lengths[1] = 1;
	MPI_Get_address(&temp.I2, &addresses[1]);

	typelist[2] = MPI_DOUBLE;
	block_lengths[2] = 1;
	MPI_Get_address(&temp.D1, &addresses[2]);

	typelist[3] = MPI_CHAR;
	block_lengths[3] = 50;
	MPI_Get_address(&temp.S1, &addresses[3]);

	MPI_Get_address(&temp, &add_start);
	for (int i = 0; i < 4; i++) displacements[i] = addresses[i] - add_start;

	MPI_Type_create_struct(4, block_lengths, displacements, typelist, &MPI_type);
	MPI_Type_commit(&MPI_type);
}
```

Continuing the example

```
int main(int argc, char *argv[])
{
	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);

	my_class::buildMPIType();

	my_class data;

	if (id == 0)
	{
		data.I1 = 6;
		data.I2 = 3.0;
		data.D1 = 10.0;
		data.var_not_to_send = 25;
		string str_temp = "My test string";
		//Copy the data from the std:: string into the C string - safer than using strcpy
		copy(str_temp.begin(), str_temp.end(), data.S1);
	}

	MPI_Bcast(&data, 1, my_class::MPI_type, 0, MPI_COMM_WORLD);

	cout << "On process " << id << " I1=" << data.I1 << " I2=" << data.I2 << " D1= " << data.D1 << " S1=" << data.S1 << ". The unsent variable is " << data.var_not_to_send << endl;
	MPI_Type_free(&my_class::MPI_type);
	MPI_Finalize();
}
```

What are we trying to do?

- When we have sent data previously we have sent a pointer to that data
- We still want to send a pointer to indicate which data is to be sent, but as the data can be discontinuous and/or of different types, we need to create a list of the data types and sizes to be sent together with the offset of that data's memory location from the pointer
  - This must be done in a generic fashion (i.e. the set of offsets must be the same for every object of that type)
    - We can't do it for, for instance, pointers stored within an object as the offset for the data pointed to will be different for each object. The same is true for STL containers as their data is dynamically allocated and therefore does not have a consistent relative position in memory
  - Later we will look at creating temporary MPI variable types to get around problems like this

How do we do this?

- We create a temporary object of the data we are trying to make the MPI datatype for:
  - The offsets of the member variables will be the same for all objects of the same class
  - my_class temp;
- We then store information about the individual variables that will make up the MPI datatype
  - Its MPI datatype - typelist[3] = MPI_CHAR;
  - The number of continuous variables of that type - block_lengths[3] = 50;
- We then get the pointer to that variable (stored in an MPI friendly format)
  - MPI_Get_address(&temp.D1, &addresses[2]);
  - Note that this is not yet the offset that we actually need, but rather the raw address

How do we do this? (continued)

- Once we have gathered this data for all the member variables that we wish to send using this MPI datatype we can calculate the offsets in the memory location
  - Obtain the memory location of the beginning of the object - MPI_Get_address(&temp, &add_start);
  - We then subtract this from all the addresses to get their offsets –

    offsets[i] = addresses[i] - add_start;
- Once we have all this information we can use it to create the structure for the MPI_datatype (MPI_my_class in this example)
  - MPI_Type_create_struct(4, block_lengths, offsets, typelist, & my_class::MPI_type);
- Once the structure for the datatype has been created it must be committed before it can be used in any communications
  - MPI_Type_commit(& my_class::MPI_type);

Sending data with our new type

- We can send them using this MPI datatype like we would any other MPI variable
  - E.g. MPI_Bcast(&data, 1, my_class::MPI_type, 0, MPI_COMM_WORLD);
  - …or MPI_Send(&data, 1, my_class::MPI_type, i, tag_num, MPI_COMM_WORLD);

- Types should be freed once they are no longer needed
  - MPI_Type_free(&my_class::MPI_type);
  - Not very important in this example, but not doing so when repeatedly using a temporary MPI datatype to send data will result in memory leaks and potential crashes
  - Note that an MPI type can be freed as soon as the communication has been set up and does not need to wait until the communication has actually completed
    - Useful when setting up temporary MPI types for use with non-blocking communications

Do Worksheet 4 Exercise 1

Other Uses

- Other times you might want to create a “permanent” MPI datatype is when you are wanting to send the same set of discontinuous data multiple times.
- A common example of this is when you want to send a column of data from a 2D array
  - Assuming you are in row-column order then all the data on a single row is continuous, but all the data on a single column is discontinuous
  - The grid for e.g. a simulation will typically exist in the same memory location/s for the entire simulation and therefore it is useful to set up MPI datatypes for the parts of the grid that you need to communicate to other processes

Do Worksheet 4 Exercise 2

Creating temporary MPI datatypes

- Creating an object for a class is not dissimilar to having MPI_datatypes for more primitive variables and can then be used in the same way
- Sometimes, though, you want to send a set of unrelated information as it is still best to send it at the same time
  - To do this we can create an that MPI_datatype that includes all the separate variables
  - These variables might not have a fixed relationship to one another in memory (e.g. memory that is reallocated as it changes size – includes most stl containers)
    - Can’t simple create the type once and forget about it

An example – Sending disparate data

- In this example assume that the class and associated functions from the previous example exist:

```
my_class my_data[10];
int value_top, value_bottom;

void Send_Data()
{
	int block_lengths[3];
	MPI_Aint addresses[3];
	MPI_Datatype typelist[3];

	MPI_Datatype MPI_Temp;

	typelist[0] = my_class::MPI_type;
	block_lengths[0] = 10;
	MPI_Get_address(my_data, &addresses[0]);

	typelist[1] = MPI_INT;
	block_lengths[1] = 1;
	MPI_Get_address(&value_top, &addresses[1]);

	typelist[2] = MPI_INT;
	block_lengths[2] = 1;
	MPI_Get_address(&value_bottom, &addresses[2]);

	MPI_Type_create_struct(3, block_lengths, addresses, typelist, &MPI_Temp);
	MPI_Type_commit(&MPI_Temp);

	MPI_Bcast(MPI_BOTTOM, 1, MPI_Temp, 0, MPI_COMM_WORLD);

	MPI_Type_free(&MPI_Temp);
}
```

```
int main(int argc, char *argv[])
{
	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);

	my_class::buildMPIType();

	if (id == 0)
	{
		for (int i = 0; i < 10; i++)
		{
			my_data[i].I1 = 6+i*25;
			my_data[i].I2 = 3-i*4;
			my_data[i].D1 = 10.0+31.*i;
			my_data[i].var_not_to_send = 25;
			string str_temp = "My test string";
			copy(str_temp.begin(), str_temp.end(), data.S1);
		}

		value_top = 16;
		value_bottom = 5;
	}

	Send_Data();

	cout << "On process " << id << endl;
	for (int i = 0; i < 10; i++)
		cout << "\t" << i << ": I1=" << my_data[i].I1 << " I2=" << my_data[i].I2 << " D1= " << my_data[i].D1 << " S1=" <<
			my_data[i].S1 << ". The unsent variable is " << my_data[i].var_not_to_send << endl;
	cout << "\ttop value: " << value_top << "\tbottom value: " << value_bottom << endl;

	MPI_Type_free(&my_class::MPI_type);
	MPI_Finalize();
}
```

Sending disparate data

- Similar to creating an MPI type for a class, but some differences
  - Don’t subtract the location of the beginning of the object – The data being sent is not related in terms of memory location
  - When sending the data the pointer to the data to be sent (or received) is now the bottom of memory - MPI_BOTTOM
- Our temporary data structure includes an MPI type that we have created ourselves
- Note that in this example the variables for sending and receiving are the same – This need not be the case – All that is required is that the order of the data and the size of the data on the sending and receiving side be the same

Do Worksheet 4 Exercise 3

Writing to File when using MPI and Post-Processing

Writing data from multiple nodes

- Data can be transferred back to a single node and written as a single file
  - Fine for smaller amounts of output data
  - Can become very expensive in communication terms if there is a lot of output data and/or data needs to be frequently outputted (e.g. outputting lots of data at multiple timesteps)
  - We will look at the MPI implementation of this using MPI_file_write (and its variants) to handle multiple writes to the same file from different processes
- One alternative is to have each process write its own data to file
  - Still need to still be transferred to a single destination, but often to a file server rather than one of the other nodes (which would still need to write it somewhere)
  - Need to carry out post-processing to combine the data from the multiple files
  - Can result in a very large number of small files, which might make you unpopular with the systems administrator!

Numbered File Names

- File names need to be numbered according to the processor that created it and (potentially) the output/iteration/timestep number
  - You don’t want to overwrite data from other processes and trying to have different processes write to the same file can result in either errors or serious blocking issues
- Using a string to create the file name and open the file (where num is an output step number and id is the rank of the process):

```
string fname;
fname = "output" + to_string(num) + "_" + to_string(id) + ".dat";
f1.open(fname.c_str(), ios_base::out);
```

Writing files using MPI

- MPI includes its own functionality for reading and writing files
  - Quite a broad topic that we will only touch the surface of
- One useful feature is the ability to write data from multiple processes to a single file either in a ordered or unordered format
  - Most, but not all file systems will support this
  - Will be slower than writing data to individual files on the local drive as it requires network communications

Example writing data to a file

- In this example each file will write a string to a shared file
  - Note that this is just a code snippet

```
MPI_File mpi_file;
string fname("test.txt");

if (MPI_File_open(MPI_COMM_WORLD, fname.c_str(), MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &mpi_file) != MPI_SUCCESS)
{
		cout << "Error opening file!" << endl;
		exit(0);
}

string string_out;

string_out = string("This data has been written by ") + to_string(id) + string("\n");

MPI_File_write_ordered(mpi_file, string_out.c_str(), (int)string_out.length(), MPI_CHAR, MPI_STATUS_IGNORE);

MPI_File_close(&mpi_file);
```

Opening MPI Files

- In MPI files are handled in a similar way to how they are done in C
  - File handles are used instead of the more C++ way of using streams

```
MPI_File mpi_file;
```

- MPI_File is the file handle variable type that is used to address specific files
- The file to be opened, e.g.:

```
MPI_File_open(MPI_COMM_WORLD, fname.c_str(), MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &mpi_file);
```

- This specific example will create a new file for writing to only

MPI_File_open

```
int MPI_File_open(
MPI_Comm comm,
char *filename,
int mode,
MPI_Info info,
MPI_File *file_handle)
```

- filename is the name of the file to be opened
  - This can include paths etc.
- comm is the communicator
  - We can stick with MPI_COMM_WORLD
- mode is the file access mode to use
  - Can be combined using a binary or (|)
- info stores information about the file that is being opened
  - Can be ignored using MPI_INFO_NULL
- file_handle is the pointer to the file handle that is used for the subsequent file writing
- The function will return a status
  - I recommend checking this status - MPI_SUCCESS means that the file has been successfully opened

File modes

MPI_MODE_RDONLY	Read only
MPI_MODE_RDWR	Reading and writing
MPI_MODE_WRONLY	Write only
MPI_MODE_CREATE	Create the file if it does not exist
MPI_MODE_EXCL	Error if creating file that already exists
MPI_MODE_DELETE_ON_CLOSE	Delete file on close
MPI_MODE_UNIQUE_OPEN	File will not be opened elsewhere
MPI_MODE_SEQUENTIAL	File will only be accessed sequentially
	Can't move backwards and forwards in the file
MPI_MODE_APPEND	Add data to end of an existing file

MPI_File_write_ordered

```
int MPI_File_write_ordered(
MPI_File file_handle,
void *buf,
int count,
MPI_Datatype datatype,
MPI_Status *status)
```

- file_handle is the pointer to the file handle that is used for the subsequent file writing
- buf is a pointer to the data to send
- count is the number of items to write
- datatype is the MPI data type to be written to the file
- The function will return a status
  - I recommend checking this status - MPI_SUCCESS means that the file has been successfully opened
- Note that this function is very similar to an MPI_Send in terms of the parameters that it expects

MPI_File_write functions

- There are a number of different types of MPI file write (and equivalent read) functions that can be used
  - MPI_File_write – Writes to the file, but each process must set the location for writing
    - By default the file pointer will point at the beginning of the file
    - By default each processor will therefore tend to write over the data of other processors
      - Each processor needs to move its file pointer to an appropriate location to stop this happening
    - Used in combination with MPI_File_seek to move the file pointer to the desired location
    - Note that this writing need not be done simultaneously on each processor
  - MPI_File_write_at – Write data to a specified location in the file
    - Roughly equivalent to using MPI_File_seek followed by MPI_File_write, but slightly safer
    - Note that this writing need not be done simultaneously on each processor
  - MPI_File_write_ordered – The data written is ordered by the processor order
    - Probably the easiest way of writing data
    - …,but requires simultaneous writing and can therefore be less efficient as every process needs to wait for the slowest process before writing can occur
    - This is writing to a shared file pointer and requires simultaneous writes (or reads) from all processes
      - This involves a blocking collective communication

File Writing in MPI general points

- The most efficient way to write files in MPI is for each process to write its own files
  - Files collated or post-processed after the simulation completes

- Sometimes useful to use a single file across all processes
  - Infrequent small file writes such as to a status log file
  - Initial reading of input data for different processes

Do Worksheet 4 Exercise 4