Creating MPI Variable Types

Why create your own MPI variable type?

- 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 our self)

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;

MPI_Datatype MPI_my_class;

class my_class
{
public:
	int I1, I2;
	int var_not_to_send;
	double D1;
	char S1[50];

	static void buildMPIType();
};
```

```
void my_class::buildMPIType()
{
	int block_lengths[4];
	MPI_Aint offsets[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++) offsets[i] = addresses[i] - add_start;

	MPI_Type_create_struct(4, block_lengths, offsets, typelist, &MPI_my_class);
	MPI_Type_commit(&MPI_my_class);
}
```

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;
		data.D1 = 10.0;
		data.var_not_to_send = 25;
		strncpy(data.S1, "My test string", 50);
	}

	MPI_Bcast(&data, 1, MPI_my_class, 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(&MPI_my_class);
	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 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
        - 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 type 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[2] = MPI_DOUBLE;
  - The number of continuous variable of that type - block_lengths[2] = 1;
- 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

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, &MPI_my_class);
- Once the structure for the datatype has been created it must be committed before it can be used in any communications
  - MPI_Type_commit(&MPI_my_class);

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, MPI_my_class, 0, MPI_COMM_WORLD);
  - …or MPI_Send(&data, 1, MPI_my_class, i, tag_num, MPI_COMM_WORLD

- Types should be freed once they are no longer needed
  - MPI_Type_free(&MPI_my_class);
  - 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

Do Worksheet 4 Exercise 1

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 at 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 may 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] = MPI_my_class;
        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;
                        strncpy(my_data[i].S1, "My test string", 50);
                }

                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(&MPI_my_class);
        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 ourself
- 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 2

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 small 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 data at multiple timesteps)
- 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

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 stream to create the file name and open the file:

```
stringstream fname;
fstream f1;
fname << "output" << num << "_" << id << ".dat";
f1.open(fname.str().c_str(), ios_base::out);
```