MPI application notes

In this set of notes I will show you a series of applications that will use the MPI framework in order to build a distributed system. MPI is far more advanced than Java RMI and is a continually evolving standard. It will be possible for you to install the necessary MPI libraries on Windows/Linux/OSX and to be able to generate applications in the normal way. First we will start with some environment setup instructions for Windows and Linux. I'm not an OSX user but the instructions will be broadly similar to those for installing in Linux. All of these examples are programmed in C/C++ instead of Java.

For Windows: Get a copy of Visual Studio "Express 2013 for Windows Desktop" from the following link and install it. http://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx After this get the HPC 2012 pack from http://www.microsoft.com/en-us/download/details.aspx?id=41634 and install that too. You will now have all the tools installed that you will need to develop MPI programs

For Linux: Use your favorite package manager (synaptic, yum, pacman, emerge, etc) and install the following packages: gcc (your compiler), make (build system for building your applications) and openmpi (libraries and headers for using mpi programs). some Linux installations may have an openmpi-devel package (Fedora or Red Hat based distributions mainly) you should install that too if it is available, if you do not see this package then the openmpi package will have all the development stuff included.

Building an MPI application on windows:

1. in Visual Studio open the file menu and select new project.
2. select Templates->Visual C++ -> Win32 and select "Win32 Console Application". then give it a name and hit ok
3. select next, check empty project, and hit finish.
4. in the solution explorer, right click on "Source Files" then add then new item and you want a C++ file. call it Main.cpp
5. copy and paste the code from App001 into this new source file. you will notice a lot of errors because we have to set up some build paths
6. go to project then properties and in the configuration properties open up the C/C++ section. in the panel on the right cedit the additional include directories to include this path (assuming you installed in the default location) "C:\Program Files\Microsoft HPC Pack 2012\Inc"
7. go to the linker section of the project properties and select the input section. under additional dependancies add the following three: msmpi.lib msmpifec.lib msmpifmc.lib
8. in the general section of the linker section modify Additional Library Directories to use this path (if you are compiling on a 32-bit machine change amd64 to i386) "C:\Program Files\Microsoft HPC Pack 2012\Lib\amd64"
9. to run your program open a command line and navigate to your visual studio project folder and find your generated program. run the following command to generate 4 instances of your program to run together "mpiexec -n 4 <program_name_here>"

Building an MPI application on Linux:

1. create a new project that will use a make file in whatever IDE you are using. If you are

running from the commandline (using vim/emacs etc) you can create a make file easily enough makesure it is called Makefile (capital M must be there)

2. Give your make file the following code this is what you will use to build your application

```
# compiler for the source files
CXX=mpic++
# flags to pass to the compiler when compiling files
# enables debugging and disables optimisation
CXXFLAGS=-g -O0
# the list of object files that will compose your application
# these are basically the list of your .cpp files except with
# .cpp changed to .o
OBJECTS=mpitest.o
# the target application that we are trying to build
TARGET=mpitest

# the suffixes that this make file understands
.SUFFIXIES: .cpp .o

# the main target that will build the entire target
# note that the lines immediately after all: must start with
# exactly one tab stop
all: $(OBJECTS)
	$(CXX) -o $(TARGET) $(OBJECTS)

# rule to clean out all of our compiled files
clean:
	rm $(OBJECTS) $(TARGET)

# rule to convert a .cpp file into a .o file
.cpp.o:
	$(CXX) $(CXXFLAGS) -c $*.cpp
```

The reason we need the makefile is because we want to use the mpic++ compiler wrapper. it is far to much work to setup the standard g++ compiler which is why the wrapper is provided for us

3. create a new file called mpitest.cpp and give it the same code from App001
4. if you are using an IDE ask it to build the application using the makefile. if you are using a command line cd into the directory that has your makefile and source files and run the make command

5. run the command "mpiexec -n 4 mpitest" to see the mpi application running.

App 001: Basic Introduction to MPI

In this application we will introduce you to MPI programming. we will start with the 4 most common tasks that will occur in all MPI programs. The code that you see here is C++ based but as you will see it looks a lot like java but with a few minor differences

01) create a new source file and give it the following code (update your make file with details of the source file if necessary)

```cpp
/** simple program to test the MPI stuff to see if it works **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {

	// see if we can initialise the mpi library this is always the
	// first thing that we must do at the start of an MPI program
	MPI_Init(NULL, NULL);

	// one of the important tasks that we have to establish is how many processes are
	// in this MPI instance. this will determine who and what we have to communicate with
	int world_size;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	std::cout << "world size is: " << world_size << std::endl;

	// another important job is finding out which rank we are. we can use this rank
	// number to assign seperate jobs to different mpi units
	int world_rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
	std::cout << "world rank is: " << world_rank << std::endl;


	// before we can end our application we need to finalise everything so MPI can shut
	// down properly
	MPI_Finalize();

	// simple hello world just to verify that everything is working for us
	std::cout << "hello world" << std::endl;
	return 0;

}
```

A few things to note here

- For those of you who are used to java but not C++ note the following:
  - #include is equivelant to Java's import statement
  - int main(int argc, char** argv) is equivalent to Java's public static void main(String[] args)
  - char** represents a 2 dimensional array of characters that have unknown width and unknow height. C++ arrays do not have a .length property
  - &world_size this is to be read as "take the memory address of" the world_size variable, you will learn about pointers in time so don't worry about it for now

- All MPI programs must start with a single call to MPI_Init() we provide null values for both parameters to ask MPI to setup in it's default way
- All MPI programs must end with a single call to MPI_Finalize() to tell the MPI library to shut down and close all connections to other machines.
- Generally in an MPI program you will want to know how many other processors are in the group. this is why we call MPI_Comm_size(MPI_COMM_WORLD, &world_size)
    - the first argument is a default communication group that includes all MPI processes that are running the same program. all other communication groups are subsets of this group.
    - the second argument is where the number will be stored. this number will indicate how many units are running the same program
    - it is possible and will be shown in later examples make subsets of MPI_COMM_WORLD to simplify some methods of communication
- Generally in an MPI program you will also want to know what process rank you are in the group this is why we call MPI_Comm_rank with similar arguments to MPI_Comm_size
    - this provides the key to communication and task division in MPI. all MPI tasks for every processor is coded into the same application. to differenciate between different tasks for different processors you simply use if statements for different rank numbers. In most of the examples we will use a size of four but this can be as many processors as you wish.

02) before the call to MPI finalize add in the following code, recompile and rerun

```
// we will try to send a message from rank zero that is just the rank number and we will ask
// the recieving process to print out that number
if(world_rank == 0) {
       MPI_Send(&world_rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
       MPI_Send(&world_rank, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
       MPI_Send(&world_rank, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
} else {
       int recieved_data = 2000;
       MPI_Recv(&recieved_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
       std::cout << "rank: " << world_rank << " recieved data from rank: " << recieved_data
<< std::endl;
}
```

A few things to note here:

- MPI_Send and MPI_Recv are the basic blocking send and recieve commands in MPI.
- In this example if a process has a world rank of zero it will send out a simple integer message with a value of zero to process ranks 1, 2, 3. these recievers must know that a message is coming. if you call MPI_Send but you not have an MPI_Recv to match it then your program will deadlock and progress will be impossible.
- MPI_Send has the following six parameters
    - the first parameter is the data that you wish to send. In this case we are sending a single integer which is the world rank of process zero. note that you must have the address of the variable and not the value itself.
    - the second parameter is the count of items to send. if you have more than one item to send at a time you specify the count here.
    - the third parameter is the datatype that is being communicated over the network. there are default basic datatypes but it is possible to define your own datatypes. we will show

    examples of this later.
    - the fourth parameter is the rank of the process to recieve the message.
    - the fifth parameter is the message tag. this gives you a method of distinguishing between different message types.
    - the sixth parameter is the communication group that the message is to be sent to. because our programs will be small we will use MPI_COMM_WORLD a lot.
- MPI_Recv has the same first six parameters as MPI_Send but with minor differences. the fourth parameter is now the process that you are recieving from not sending to.
    - the last parameter indicates the status of the message that has been recieved. for something as simple as this we will ignore the status information for now.
- later examples we will show how to do asynchronus communication

App 002 Master Slave communication distributed systems

In the previous example you saw a very simple example of how to communicate between processes. In this application we will show how to do some simple master slave type communication. We will have one master node and three slave nodes. The master node is the coordinator and tells the slaves what to do, while the slave nodes will wait until they recieve a message from the master will do that work and will then wait until the master tells them to shut down. The context for this example is each slave will generate an array of 100 numbers and will calculate the average of them all. when the master recieves all three averages it will calculate the overall average of them. This structure is very useful in map-reduce type tasks or any task where you need a single node to coordinate everything else.

01) clear out the code of your mpitest program and give it this shell code

```
/** simple program to test the MPI stuff to see if it works **/

/** includes **/
#include <iostream>
#include <cstdlib>
#include <mpi.h>

/** messages for communicating tasks **/
int COMPUTE_AVERAGE = 1;
int SEND_AVERAGE = 2;
int SHUTDOWN = 3;

/** the world rank and size that will be useful in many functions **/
int world_size;
int world_rank;


int main(int argc, char** argv) {

        // see if we can initialise the mpi library this is always the first thing that we
        // must do at the start of an MPI program
        MPI_Init(NULL, NULL);

        // one of the important tasks that we have to establish is how many processes are in
        // this MPI instance. this will determine who and what we have to communicate with
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        // another important job is finding out which rank we are. we can use this rank
        // number to assign seperate jobs to different mpi units
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);


        // before we can end our application we need to finalise everything so MPI can shut
        // down properly
        MPI_Finalize();

        // standard C/C++ thing to do at the end of main
        return 0;

}
```

02) add in the following code before the call to MPI_Finalise in the main function

```
// depending on which rank we are call the appropriate function
if(world_rank == 0)
	master();
else
	slave();
```

A few things to note here:

- this is the main reason why you have access to the world rank. it is a unique ID for each process in the group. thus you can assign specific tasks based on this number
- in this example as we know we will always have a node with rank 0 we generally use that as the rank for the master node all other nodes will be considered as slaves

03) add in this function just before the main function

```
void master(void) {
	// the total average of all the averages from the nodes
	float total_average = 0;
	// an average that we recieve from a node
	float average = 0;

	// ask all three nodes to compute an average
	for(int i = 1; i < world_size; i++)
		MPI_Send(&COMPUTE_AVERAGE, 1, MPI_INT, i, 0, MPI_COMM_WORLD);

	std::cout << "Master (0): told all slaves to compute" << std::endl;

	// ask all three nodes to send their average to us
	for(int i = 1; i < world_size; i++) {
		MPI_Send(&SEND_AVERAGE, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
		MPI_Recv(&average, 1, MPI_FLOAT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		total_average += average;
	}

	// take the average of averages and display the result
	std::cout << "Master (0): average result from all slaves is: " << total_average / 
(world_size - 1) << std::endl;

	// tell all the nodes to shutdown
	for(int i = 1; i < world_size; i++) {
		MPI_Send(&SHUTDOWN, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
	}

	std::cout << "Master (0): shutting down all slaves" << std::endl;

}
```

A few things to note here

- this is the function that will be used by the master node. In the first for loop this node sends a message to each of the slave nodes to compute the average of 100 random integers by sending a single integer that indicates an action
- the second for loop requests each node to send the computed average for it's generated list of 100 integers
  - note how it does a send and then looks for an immeiate recieve. this is the form of communication you must do when using blocking calls. Later on you will be able to split this in two by using asynchronus sends and recieves so you can reuse the thread while

MPI is busy handling communication.

- The third loop sends the shut down message which will cause all processes to finish executing.
- Note that it is entirely possible to have the master process do some processing and taking part in the task while the slave threads are computing. In this case the master process will become a coordinator.

04) add in this function just below the definition of the master function

```
void slave(void) {
	// the message type that we have recieved
	int message_type = 0;
	float average = 0;

	// keep looping until we recieve a shutdown message
	MPI_Recv(&message_type, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	while(message_type != SHUTDOWN) {
		if(message_type == COMPUTE_AVERAGE) {
			std::cout << "Slave (" << world_rank << "): calculating average" << 
std::endl;

			// get the average of 100 random numbers
			srand(world_rank);
			int sum = 0;
			for(int i = 0; i < 100; i++)
				sum += rand() % 10;
			average = sum / 100.f;

			std::cout << "Slave (" << world_rank << "): sum of 100 ints is " << sum 
<< std::endl;
			std::cout << "Slave (" << world_rank << "): average of 100 ints is " << 
average << std::endl;
			std::cout << "Slave (" << world_rank << "): calculated average" << 
std::endl;

		} else if(message_type == SEND_AVERAGE) {
			MPI_Send(&average, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);

			std::cout << "Slave (" << world_rank << "): sent average" << std::endl;
		}
		MPI_Recv(&message_type, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	}

	std::cout << "Slave (" << world_rank << "): going for shutdown" << std::endl;

}
```

A few things to note here:

- Here the slaves are constantly waiting for a new message from the master process until the shutdown message is sent
- note that if you are using a while loop you should recieve a message first before going any further. (good place to use a do while loop)
- everytime the slave recieves a message it will look at the message type and from that it will take the appropriate action.
- finally when the shutdown message is recieved then the loop will terminate and the function will be exited.

App003: App002 in a ring like structure.

It is not necessary to write all algorithms in a master-slave type structure. In this example we will replicate the same task however we will use a ring structure of 4 processes connected in this order 0 -> 1 -> 2 ->3 -> 0. process zero will act as the coordinator it will inform process 1 to start processing. process 1 will inform process 2 to start processing and so on. This method takes advantage of the parallel/concurent nature of a distributed system a bit better than the master-slave method. when the results are computed process zero will ask process 1 to total the sum by passing it's sum to process one. this will continue until process zero recieves the total sum and it will then ask the other processes to shut themselves down. this kind of structure is very useful in matrix multiplication tasks

01) create a new project with a single source file

02) add in the following shell code

```
/** simple program to test the MPI stuff to see if it works **/

/** includes **/
#include <iostream>
#include <cstdlib>
#include <mpi.h>

/** messages for communicating tasks **/
int COMPUTE_AVERAGE = 1;

/** the world rank and size that will be useful in many functions **/
int world_size;
int world_rank;

int main(int argc, char** argv) {

	// see if we can initialise the mpi library this is always the first thing that we
	// must do at the start of an MPI program
	MPI_Init(NULL, NULL);

	// one of the important tasks that we have to establish is how many processes are in
	// this MPI instance. this will determine who and what we have to communicate with
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	// another important job is finding out which rank we are. we can use this rank
	// number to assign seperate jobs to different mpi units
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// compute the average of 400 numbers by using ring like communication. whoever is
	// node zero should be given the job of coordinator
	if(world_rank == 0)
		coordinator();
	else
		computeAverage();

	// before we can end our application we need to finalise everything so MPI can shut
	// down properly
	MPI_Finalize();

	// standard C/C++ thing to do at the end of main
	return 0;

}
```

A few things to note here:

- note that we have removed two of the message types. This is because we will instruct each node to start computation and after that we will follow a strict method for this task. Node that we set a single node as coordinator and all other nodes are helpers.

02) add the following function above the main function

```
void coordinator(void) {
	int message;

	// tell the next node that we have to start computing an average
	MPI_Send(&COMPUTE_AVERAGE, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

	// we will expect the last node to send us a message to compute our average. get it
	// and ignore it
	MPI_Recv(&message, 1, MPI_INT, world_size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

	// compute our average and print out the results
	srand(world_rank);
	int sum = 0;
	for(int i = 0; i < 100000; i++)
		sum += rand() % 10;
	float average = sum / 100000.f;

	std::cout << "coordinator 0 sum is: " << sum << std::endl;
	std::cout << "coordinator 0 average is: " << average << std::endl;

	// ask node one to compute it's average by sending a single floating point value this
	// will add in it's average and pass to node 2 etc
	// when we get the result back we will have the full average
	MPI_Send(&average, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
	MPI_Recv(&average, 1, MPI_FLOAT, world_size - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

	// print out the average by dividing by the number of nodes and exit
	std::cout << "coordinator 0 total average is: " << average / world_size << std::endl;

}
```

A few things to note here:

- this looks similar to the master function from the previous program. however in this case we are also performing computation with this node. first the coordinator must instruct all other nodes in the ring to start computation. it does this by sending a message to node 1. when all nodes have been notified the coordinator recieves the compute message from the last node in the ring.
- at this point the coordinator knows that all nodes are processing and therefore can start processing itself.
- when computation is finished the coordinator must get the total average from all nodes. again we use ring like communication here. we send the average onto the next node which will put add its average.
- at the end when all nodes have added the average the coordinator will recieve a message from the last node with the fully compiled average

03) add the following function above the main function

```
void computeAverage(void) {
	int message;
	float current_average;

	// wait till we get the compute message after this tell the next node to start
	// computing an average before computing our average
	MPI_Recv(&message, 1, MPI_INT, (world_rank + world_size - 1) % world_size, 0, 
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	MPI_Send(&COMPUTE_AVERAGE, 1, MPI_INT, (world_rank + 1) % world_size, 0, 
MPI_COMM_WORLD);
	srand(world_rank);
	int sum = 0;
	for(int i = 0; i < 100000; i++)
		sum += rand() % 10;
	float average = sum / 100000.f;

	// print out infomation about our sum and average
	std::cout << "node " << world_rank << " sum is: " << sum << std::endl;
	std::cout << "node " << world_rank << " average is: " << average << std::endl;

	// at some point we will get a message that will contain a current total of the
	// overall average. take that value add our average
	// to it and pass it on to the next node
	MPI_Recv(&current_average, 1, MPI_FLOAT, (world_rank + world_size - 1) % world_size, 
0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	current_average += average;
	std::cout << "node " << world_rank << " current average is " << current_average << 
std::endl;
	MPI_Send(&current_average, 1, MPI_FLOAT, (world_rank + 1) % world_size, 0, 
MPI_COMM_WORLD);

}
```

A few things to note here:

- Note that the big difference between this method and the slave of the last method is that we have no for loops or if statements. there are no decisions necessary. All coorination is handled by the sending and the recieving of messages.
- At the very start of the function we wait for the compute average message to be recieved. Once recieved we immediately pass it onto the next node in the ring and start computing.
- After the average has been computed each node knows that it will then recieve a single floating point number that represents a total average from the previous node each node will add it's average to this total and pass it onto the next node.
- after this the node will exit and finish.

App 004: The Hypercube topology

In this application we will do an addition task similar to the previous three applications. However in this application we will show a hypercube topology. a hypercube is a 4 or more dimensional extension of a cube. each edge has exactly two nodes regardless of direction. This may seem like a complex structure or topology for a network or communication, however it is efficient at propogating data and messages or for collecting messages or data from the network. A restriction of this network is that there must be a number of nodes that is an exact power of two. it is recommended that you run this with 8 or 16 nodes

01) start with a fresh cpp file and add in the following main method.

```
int main(int argc, char** argv) {

	// see if we can initialise the mpi library this is always the first thing that we
	// must do at the start of an MPI program
	MPI_Init(NULL, NULL);

	// one of the important tasks that we have to establish is how many processes are in
	// this MPI instance. this will determine
	// who and what we have to communicate with
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	// another important job is finding out which rank we are. we can use this rank
	// number to assign seperate jobs to different
	// mpi units
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// depending on our rank we may be the coordinator or a participant
	if(world_rank == 0)
		coordinator();
	else
		participant();

	// before we can end our application we need to finalise everything so MPI can shut
	// down properly
	MPI_Finalize();

	// standard C/C++ thing to do at the end of main
	return 0;

}
```

02) add in the following helper methods above the main method

```
// function to caluclate the power of 2 that we are using. in the hyper cube case we need
this to figure out
// how many directions of communication we have
int hypercubePower(int size) {
	unsigned int power = 1, j = 0;
	for(; power < size; power = power << 1, j++);
	std::cout << j << std::endl;
	return j;
}

// initialises an array with the given value and size
void initArray(int *to_init, int value, unsigned int size) {
	for(unsigned int i = 0; i < size; i++)
		to_init[i] = value;
}

// function to calulate our most significant power
```

```
int mostSignficantPower(int rank) {
	int power = 1;

	// to calculate our most significant power keep multiplying by two until we find a
number
	// that is bigger than our rank then go one step back
	while(power < rank)
		power <<= 1;
	power >>= 1;

	// return the power when we are finished
	return power;
}

// function that determines the communication directions based on the given rank
void computeDirections(int *comms_offset, int rank, int hypercube_power) {
	// go through each power if we have a 0 bit we must add, a 1 bit we must subtract
	for (int i = 1, j = 0; j < hypercube_power; i <<= 1, j++) {
		if (rank & i)
			comms_offset[j] = -i;
		else
			comms_offset[j] = i;
	}
}

// function to calculate the sum of 100 random integers and return it
int randomSum(void) {
	int sum = 0;
	for(unsigned int i = 0; i < 100; i++)
		sum += rand() % 10;
	return sum;
}
```

A few things to note here:

- some of these functions are necessary in a hypercube topology you need to know the exact power of the number of nodes in the topology
- the most significant power is used to determine which directions each node will send messages in and which directions it will recieve messages for both distribution and reduction tasks
- each node can communicate with one and only one node in each direction which will be determined by the individual bits in their rank. if the bit is one the communicating node had that bit set to zero and vice versa. this is why we calculate an offset for each communication direction.

03) add in the following coordinator method

```
// the coordinator of the application
void coordinator(void) {
	// we need to establish what power is used in the hypercube this tells us how many
communication directions we have
	int power = hypercubePower(world_size);
	std::cout << power << std::endl;
	// an array that will house the various offsets that we need to connect with the
appropriate nodes in each direction
	int comms_offsets[16];
	initArray(comms_offsets, 0, 16);

	// we need to determine our communication directions based on the bits in our rank
	computeDirections(comms_offsets, world_rank, power);
	for(unsigned int i = 0; i < power; i++)
```

```
            std::cout << "rank " << world_rank << " offset " << i << ": " <<
comms_offsets[i] << std::endl;

        // to start the computation we need to distribute a message for starting the
computation in this case we will send a
        // single integer that will represent the dimension that we need to communicate on.
this will state which channel we
        // need to send on
        int message = 0;
        for (unsigned int i = 0; i < power; i++) {
            MPI_Send(&i, 1, MPI_INT, world_rank + comms_offsets[i], 0, MPI_COMM_WORLD);
            std::cout << "rank " << world_rank << "send message to rank " << world_rank +
comms_offsets[i] << std::endl;
        }

        // calculate the average of 100 random numbers
        srand(world_rank);
        int sum = randomSum();
        float average = sum / 100.f;
        std::cout << "rank " << world_rank << " average is: " << average << std::endl;

        // send a message to all the other nodes to tell them to sum up their averages
        for (unsigned int i = 0; i < power; i++) {
            MPI_Send(&i, 1, MPI_INT, world_rank + comms_offsets[i], 0, MPI_COMM_WORLD);
            std::cout << "rank " << world_rank << "send total message to rank " <<
world_rank + comms_offsets[i] << std::endl;
        }

        // we need to get the averages from each direction
        float total_average = average;
        for (int message = power - 1; message >= 0; message--) {
            MPI_Recv(&average, 1, MPI_FLOAT, world_rank + comms_offsets[message], 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            std::cout << "rank " << world_rank << " recieved total from " << world_rank +
comms_offsets[message] << std::endl;
            total_average += average;
        }

        // print out the overall average
        std::cout << "rank " << world_rank << " total and average: " << total_average << " "
<< total_average / world_size << std::endl;

}
```

A few things to note here:

- as you can see from the method above communicating in a hyper cube even with a simple additional task it can be quite complex however the upside of this is that the propogation of a message from originator to all nodes is O(log N) instead of O(N) in a master/slave or ring topology.
- note that the message we send for coordination is based on the power as this will determine the next communication direction.
- In this example node zero will send a message to nodes 1,2, and 4

04) add in the following function

```
// a participant in the application
void participant(void) {
        // we need to establish what power is used in the hypercube this tells us how many
communication directions we have
        int power = hypercubePower(world_size);
        std::cout << power << std::endl;
```

```
	// an array that will house the various offsets that we need to connect with the
appropriate nodes in each direction
	int comms_offsets[16];
	initArray(comms_offsets, 0, 16);

	// we need to determine our communication directions based on the bits in our rank
	computeDirections(comms_offsets, world_rank, power);
	for (unsigned int i = 0; i < power; i++)
		std::cout << "rank " << world_rank << " offset " << i << ": " <<
comms_offsets[i] << std::endl;

	// we need to recieve a message to start computation from another node and then
propogate it on further
	int message = 0;
	MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	for (message++; message < power; message++) {
		MPI_Send(&message, 1, MPI_INT, world_rank + comms_offsets[message], 0,
MPI_COMM_WORLD);
		std::cout << "rank " << world_rank << " send message to rank " << world_rank +
comms_offsets[message] << std::endl;
	}

	// calculate the average of 100 random numbers
	srand(world_rank);
	int sum = randomSum();
	float average = sum / 100.f;
	std::cout << "rank " << world_rank << " average is: " << average << std::endl;

	// we need to get a message that will tell us to total our average get this message
and retain it
	// send a message to all the other nodes to tell them to sum up their averages
	MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
	for (int i = message + 1; i < power; i++) {
		MPI_Send(&i, 1, MPI_INT, world_rank + comms_offsets[i], 0, MPI_COMM_WORLD);
		std::cout << "rank " << world_rank << " sent total message to rank " <<
world_rank + comms_offsets[i] << std::endl;
	}

	// we need to get the averages from each direction
	float total_average = average;
	for (int i = power - 1; i > message; i--) {
		MPI_Recv(&average, 1, MPI_FLOAT, world_rank + comms_offsets[i], 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		std::cout << "rank " << world_rank << " recieved total from " << world_rank +
comms_offsets[i] << std::endl;
		total_average += average;
	}

	// print out the overall average
	std::cout << "rank " << world_rank << " total and average: " << total_average << " "
<< total_average / world_size << std::endl;

	// send our average onto the the next node
	std::cout << "rank " <<  world_rank << " " << comms_offsets[message] << std::endl;
	MPI_Send(&total_average, 1, MPI_FLOAT, world_rank + comms_offsets[message], 0,
MPI_COMM_WORLD);
	std::cout << "rank " << world_rank << " sent total average to rank " << world_rank +
comms_offsets[message] << std::endl;

}
```

A few things to note here:

- Again this looks more complex than before but the communication speedup is worth the complexity.

- at first we wait to recieve a message from any node when that message is recieved it will denote which direction the communication came from. This will determine the amount of sends each node must do. A higher number indicates less sending. A number equal to the power will indicate no sending.
- computation works as before for the summation of the 100 numbers and then a similar communication message is sent through the network.
- After this the totals must be sent back to the coordinator. this works in the opposite way to sending the communication message. the totals are added and propigated back to the coordinator. This takes advantage of multiple communication links and acts like a reverse flood.

App 005 using MPI's inbuilt broadcasting mechanism

With the previous applications we have shown a need to communicate simple messages using the MPI send and MPI recv commands. This is such a common operation in MPI applications that there is a special function reserved for broadcasting a message to all other units in a communicator. This application shows the most basic example of communicating a message to each and every rank in the communicator. note that the if statement is not required around the Bcast function. MPI will automatically determine which node is broadcasting and which nodes are recieving.

01) start with a fresh MPI project and create a single source file

02) put this code in your source file

```
/** simple program to show how the MPI broadcast function works **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {

	// initialise the MPI library
	MPI_Init(NULL, NULL);

	// determine the world size
	int world_size;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	// determine our rank in the world
	int world_rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// print out the rank and size
	std::cout << "rank " << world_rank << " size " << world_size << std::endl;

	// 03 code goes here

	// finalise the MPI library
	MPI_Finalize();
}
```

03) add this code in place of the comment for 03 code

```
if(world_rank == 0) {
	// broadcast a message to all the other nodes
	int message = 0xDEADBEEF;
	MPI_Bcast(&message, 1, MPI_INT, 0, MPI_COMM_WORLD);
	std::cout << "rank 0 broadcasting " << message << std::endl;
} else {
	// recieve a message from the root
	int message = 0;
	MPI_Bcast(&message, 1, MPI_INT, 0, MPI_COMM_WORLD);
	std::cout << "rank " << world_rank << " recieved broadcast of " << message << 
std::endl;
}
```

A few things to note here:

- The if else statement is not necessary here. the only reason that it is used is to show that MPI_Bcast determines who is sending and recieving by itself.
- MPI_Bcast takes 5 arguments

- the first is the buffer to send from or recieve into
- the second is the count of items to send
- the third is the type of data to send
- the fourth is the root of this communication. if the root matches the world rank MPI_Bcast will initiate a send. otherwise it will initiate a recieve.
- the final argument is the communicator through which this message will be sent.

App006: using MPI_Reduce to total up a sum across all nodes in a communicator

In the examples we have seen so far we have had to communicate our results back to the root node using a combination of sends and recieves. This is also a very common task within MPI that there is a seperate function for doing these reduce operations. In this example we show 4 notes with a single sum communicating to the root node to generate a total sum.

01) start with a fresh project

02) add in this base code to your main source file

```
/** simple program to show how the MPI broadcast function works **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {

	// initialise the MPI library
	MPI_Init(NULL, NULL);

	// determine the world size
	int world_size;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	// determine our rank in the world
	int world_rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// print out the rank and size
	std::cout << "rank " << world_rank << " size " << world_size << std::endl;

	// code for 03 goes here

	// finalise the MPI library
	MPI_Finalize();
}
```

03) add in this code in place of the comment marked "code for 03 goes here"

```
// lets imagine we have a local sum in each node that is 5 and we want to reduce that to a
// single value in the root node that is to be displayed to the user
int our_sum = 5;
int total_sum = 0;
MPI_Reduce(&our_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
std::cout << "rank " << world_rank << " our_sum, total_sum: " << our_sum << ", " <<
total_sum << std::endl;
```

A few things to note here:

- note that there is no decision to be made on who sends or recieves the data. MPI_Reduce like MPI_Bcast will automatically determine who is the sender and who is the reciever
- MPI_Reduce requires 7 arguments
  - The first is the local variable that is to be reduced across all nodes. All nodes must implement this variable

- The second is the destination variable that will contain the reduced value. All nodes must implement this variable but only the root node will store a value in this variable.
- Thrid and Fourth are the count and type as before,
- The fifth argument is the reduce operation that is to be performed on all nodes. sum is one such built in operation but there are other built in operations like min, max, bitwise and/or/not, logical and/or/not etc.
- The sixth argument is the root of the operation. this functions in the same manner as MPI_Bcast.
- The seventh argument is the communicator on which the operation is performed

App007: using MPI_Barrier to force synchronisation during a computation

At certian points during a computation you may require that all nodes finish a certian set of processing tasks before they can move onto the next task. In this case you need to force all nodes to synchronise before they can continue. This is achieved by the MPI_Barrier function which will wait for all nodes to synchronise before allowing all nodes to continue working.

01) start with a new project

02) in your main source file add in the following base code

```
/** simple program to show how the MPI broadcast function works **/

/** includes **/
#include <iostream>
#include <mpi.h>
#include <cstdlib>

int main(int argc, char** argv) {

	// initialise the MPI library
	MPI_Init(NULL, NULL);

	// determine the world size
	int world_size;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	// determine our rank in the world
	int world_rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// print out the rank and size
	std::cout << "rank " << world_rank << " size " << world_size << std::endl;



	// finalise the MPI library
	MPI_Finalize();
}
```

03) add in the following code in place of the comment marked "code for 03 goes here"

```
// to show off the mpi barrier command we will simulate an operation across many nodes by
// implementing a random sleep time we will get all nodes to synchronise before printing out
// a final message those of you using *nix systems should remove the leading underscore from
// the call to the sleep command
srand(world_rank);
int sleep_time = rand() % 5;
std::cout << "node " << world_rank << " sleeping for " << sleep_time << " seconds." <<
std::endl;
_sleep(sleep_time * 1000);
std::cout << "node " << world_rank << " exiting sleep and synchronising" << std::endl;

MPI_Barrier(MPI_COMM_WORLD);
std::cout << "node " << world_rank << "synchronised" << std::endl;
```

A few things to note here:

- here we pick a random sleep time for all nodes between 0 and 5 seconds. this is used to simulate different workloads on each node. when you run this application you will note that all nodes will sleep and synchronise before the final message is displayed
- MPI_Barrier forces synchronisation. it accepts a single parameter which is a communicator. The barrier will force all nodes into a cold sleep until all nodes in the communicator have made the MPI_Barrier call. MPI_Barrier will then release all nodes to start computing again. hence this is why you see the synchronised message at the end of computation.

App008: An example of a standard MPI computation application

In previous applications we showed the basics of communication and synchronisation with MPI. However, most MPI applications will send out data from one process to the rest perform some computation and then will reduce those results to a single unit. In this example we show how to distribute data to individual nodes do the computation and collect the result by using the reduce command.

01) start with a fresh MPI project

02) add in the following shell code to your main source file

```
/** simple program to show how the MPI broadcast function works **/

/** includes **/
#include <iostream>
#include <mpi.h>

// function that will implement the coordinator job of this application
void coordinator(int world_size) {

}

// function that will implement the participant job of this applicaiton
void participant(int world_rank, int world_size) {

}

int main(int argc, char** argv) {

	// initialise the MPI library
	MPI_Init(NULL, NULL);

	// determine the world size
	int world_size;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	// determine our rank in the world
	int world_rank;
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// print out the rank and size
	std::cout << "rank " << world_rank << " size " << world_size << std::endl;

	// if we have a rank of zero then we are the coordinator. if not we are a participant
	// in the task
	if(world_rank == 0)
		coordinator(world_size);
	else
		participant(world_rank, world_size);

	// finalise the MPI library
	MPI_Finalize();
}
```

03) add in the following code to the coordinator function

```
std::cout << "coordinator rank 0 starting " << std::endl;

// generate 100000 random integers and store them in an array
int values[100000];
for(unsigned int i = 0; i < 100000; i++)
       values[i] = rand() % 10;

// determine the size of each partition by dividing 100000 by the world size
// it is impertative that the world_size divides this evenly
int partition_size = 100000 / world_size;
std::cout << "coordinator rank 0 partition size is " << partition_size << std::endl;

// broadcast the partition size to each node so they can setup up memory as appropriate
MPI_Bcast(&partition_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
std::cout << "coordinator rank 0 broadcasted partition size" << std::endl;

// send out a partition of data to each node
for(unsigned int i = 1; i < world_size; i++) {
       MPI_Send(values + partition_size * i, partition_size, MPI_INT, i, 0, MPI_COMM_WORLD);
       std::cout << "coordinator rank 0 sent partition to rank " << i << std::endl;
}

// generate an average for our partition
int total = 0;
for(unsigned int i = 0; i < partition_size; i++)
       total += values[i];
float average = (float) total / partition_size;
std::cout << "coordinator rank 0 average is " << average << std::endl;

// call a reduce operation to get the total average and then divide that by the world size
float total_average = 0;
MPI_Reduce(&average, &total_average, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
std::cout << "total average is " << total_average / world_size << std::endl;
```

A few things to note here:

- In this example the root node will generate the data in this case it is randomly generated but in the majority case it will be stored in files that are to be read and transfered over the network
- note that there is a division on the total data to generate equal size partitions that are independant of each other
- the root broadcasts the partition size to all the nodes in the communicator this is to ensure that the other nodes can allocate the necessary memory for the values it will recieve.
- note that in the send command the first argument is values + partition_size * i. note that the ampersand is missing. this is because values is an array and not a basic variable. arrays do not require the preciding ampersand. the addition of partition_size * i is to ensure that each node recieves the correct data.
- as before the calculation of the average and the reduction of the values work the same

04) add in the following code to the participant function

```
std::cout << "participant rank " << world_rank << " starting" << std::endl;

// get the partition size from the root and allocate memory as necessary
int partition_size = 0;
MPI_Bcast(&partition_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
std::cout << "participant rank " << world_rank << " recieved partition size of " <<
partition_size << std::endl;
```

```cpp
// allocate the memory for our partition
int *partition = new int[partition_size];

// recieve the partition from the root
MPI_Recv(partition, partition_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
std::cout << "participant rank " << world_rank <<  " recieved partiton from root" <<
std::endl;

// generate an average for our partition
int total = 0;
for(unsigned int i = 0; i < partition_size; i++)
       total += partition[i];
float average = (float) total / partition_size;
std::cout << "participant rank " << world_rank << " average is " << average << std::endl;

// call a reduce operation to get the total average and then divide that by the world size
float total_average = 0;
MPI_Reduce(&average, &total_average, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// as we are finished with the memory we should free it
delete partition;
```

A few things to note here:

- the first operation is to determine the size of the data that we are recieving. hence the call to MPI_Bcast
- after this we can allocate the appropriate array size before recieving the values. again because partition is an array we omit the & in the MPI_Recv command
- the rest of the operations for generating the average and reducing the values proceed as before
- note for those of you coming from Java in C/C++ we are responsible for cleaning up our memory there is no garbage collector for performance reasons therefore when we are finished with allocated memory we must call a delete on the memory we allocated.

App009: using the MPI_Scatter operation to distribute partitions of data to each node in a communicator

In the previous example we had a partition of data that needed to be distributed among all nodes in the MPI cluster. In most cases if you have an array that needs to be distributed in partitions among different nodes. As this is a common operation in MPI there is a function dedicated to this operation. this function is called MPI_Scatter. and it works in a similar way to MPI_Bcast and MPI_Reduce.

01) start with a new MPI project

02) in your source file add the following code

```
/** simple program to show how the mpi scatter operation works **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
	// initialise the MPI libary
	MPI_Init(NULL, NULL);

	// determine the world size and the world rank
	int world_size, world_rank;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// arrays that we need for communicating data
	int *total_array = new int[40];
	int *partition = new int[40 / world_size];

	// if we are process zero then create an array of 40 random integers
	if(world_rank == 0) {
		std::cout << "total array start-------- " << std::endl;
		for(unsigned int i = 0; i < 40; i++) {
			total_array[i] = rand() % 10;
			std::cout << total_array[i] << ", ";
			if(i % 10 == 9) std::cout << std::endl;
		}
		std::cout << "total array end-------- " << std::endl;
	}

	// run the scatter operation and then display the contents of all 4 nodes
	MPI_Scatter(total_array, 40 / world_size, MPI_INT, partition, 40 / world_size,
MPI_INT, 0, MPI_COMM_WORLD);
	std::cout << "rank " << world_rank << " partition: ";
	for(unsigned int i = 0; i < 40 / world_size; i++)
		std::cout << partition[i] << ", ";
	std::cout << std::endl;

	// we are done with the MPI library so we must finalise it
	MPI_Finalize();

	// clear up our memory before we finish
	delete total_array;
	delete partition;
}
```

A few things to note here:

- In this example a 40 element array of random integers is created by rank 0 that is to be partitioned and distributed to all ranks in the communicator.
- we print out the array to begin with to show all of the values that are in the generated array to check that the scatter operation works as expected
- All ranks must participate in the MPI_Scatter command including the rank that is scattering the data to other nodes. The MPI_Scatter command expects 8 arguments
    - The first argument is the array that is to be scattered amongst the ranks in the communicator.
    - The second argument is the number of items that should be distributed to each rank
    - The third is the type of data that is being distributed
    - the node that serves as the root will reference these three arguments while non-root nodes will reference the next three
    - The fourth, fifth and sixth arguments follow the same format as the first three but this time they describe the buffer that is recieving the data. again a location, count and data type that is being recieved
    - the last two arguments specify the root of the scatter and also which communicator that this scatter is functioning on.

App010: using the MPI_Gather command to recieve partitions of data from all nodes in a communicator

In the previous example we saw how to send individual partitions of data from a single root node to multiple child nodes. It is not uncommon for MPI applications to send data out to multiple nodes and expect to recieve partitions of data back. One such example would be computing a series of eigenvalues for a matrix. The example we show here is the opposite of App009 we have a set of nodes generating 10 random numbers and the root will gather all 4 partitions and will assemble them in a single array

01) start with a new MPI project

02) give your source file the following code

```
/** simple program to show how the mpi scatter operation works **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
	// initialise the MPI libary
	MPI_Init(NULL, NULL);

	// determine the world size and the world rank
	int world_size, world_rank;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// arrays that we need for communicating data
	int *total_array = new int[40];
	int *partition = new int[40 / world_size];

	// all processes should generate a partition of data to be communicated to the node
that is gathering
	// and print it out to console
	srand(world_rank);
	std::cout << "rank " << world_rank << " numbers: ";
	for(unsigned int i = 0; i < 40 / world_size; i++) {
		partition[i] = rand() % 10;
		std::cout << partition[i] << ", ";
	}
	std::cout << std::endl;

	// run the gather operation and print out the total partition that was recieved
	MPI_Gather(partition, 40 / world_size, MPI_INT, total_array, 40 / world_size,
MPI_INT, 0, MPI_COMM_WORLD);
	if(world_rank == 0) {
		std::cout << "total data--------" << std::endl;
		for(unsigned int i = 0; i < 40; i++) {
			std::cout << total_array[i] << ", ";
			if(i % 10 == 9)
				std::cout << std::endl;
		}
		std::cout << std::endl;
	}


	// we are done with the MPI library so we must finalise it
```

```
        MPI_Finalize();

        // clear up our memory before we finish
        delete total_array;
        delete partition;
}
```

A few things to note here:

- As stated this example is similar to the previous application but here we are gathering multiple partitions of data together
- All processes will generate their own individual partition of data before taking part in the gather operation
- Note that MPI gather takes the same number of arguments and the same types for each argument as MPI_Scatter
- be careful though the size to send must match the size to recieve as the root node is expecting this size from all nodes that are taking part in the gather. if you specify the total size of the partition in the recieve your MPI application will crash and will be shut down.

App011: creating a new communicator in MPI

In the examples thus far we have defaulted to using the single provided communicator MPI_COMM_WORLD. However for many MPI tasks it is useful to divide communications into seperate groups. each of these groups is a communicator but all communicators are subsets of MPI_COMM_WORLD. This example will show you how to make subsets of MPI_COMM_WORLD.

01) start with a new MPI project

02) give your source file the following shell code

```
/** simple program to show how the mpi scatter operation works **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
	// initialise the MPI libary
	MPI_Init(NULL, NULL);

	// determine the world size and the world rank
	int world_size, world_rank;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// code for 03 goes here

	// code for 04 goes here

	// code for 05 goes here

	// code for 06 goes here

	// we are done with the MPI library so we must finalise it
	MPI_Finalize();
}
```

03) add in the following code in place of the comment marked "code for 03 goes here"

```
// creating a communicator in MPI requires a bit of work but the first task is to extract a
// handle for the world communicator as all user defined communicators are subsets of this
// overall communicator
MPI_Group world_group;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
```

A few things to note here:

- building a communicator requires the use of MPI_Group. one of the first tasks you do in creating a new communicator is to retrieve the communicator associated with MPI_COMM_WORLD through the use of the MPI_Comm_group command.

04) add in the following code in place of the comment marked "code for 04 goes here"

```
// we need to determine the world rank numbers that will be in the new communicator. in this
// example we will divide our group into two dynamically
int *group_ranks = new int[world_size / 2];
if(world_rank < world_size / 2) {
	for(int i = 0; i < world_size / 2; i++)
		group_ranks[i] = i;
} else {
	for(int i = world_size / 2, j = 0; i < world_size; i++, j++)
		group_ranks[j] = i;
}

// print out the group ranks
std::cout << "rank " << world_rank << " group ranks: ";
for(int i = 0; i < world_size / 2; i++)
	std::cout << group_ranks[i] << ", ";
std::cout << std::endl;
```

A few things to note here:

- here we are dividing our world ranks into two seperate groups. we have a group for the lower half of ranks (the if statement) and a group for the upper half of ranks (the else statement)
- the code you see here will work for any even number of processes that you provide

05) add in the following code in place of the comment marked "code for 05 goes here"

```
// create a new group out of the ranks and then create the communicator with that group
MPI_Group new_group;
MPI_Group_incl(world_group, world_size / 2, group_ranks, &new_group);
MPI_Comm sub_comm;
MPI_Comm_create(MPI_COMM_WORLD, new_group, &sub_comm);
```

A few things to note here:

- using the ranks subsets that were created in 04 we can generate a new MPI_Group which in turn can be used to create a new communicator.
- MPI_Group_incl takes in a list of ranks to include in a new group. this function expects 4 arguments
  - the first is the MPI_Group that you are making a subset of
  - the second is the number of ranks that will be in this new group
  - the third is the individual ranks that make up this group.
  - the fourth is the MPI_Group where this group is initialised and stored
- Once the new group is created we can initialise our new communicator by using the MPI_Comm_create function which expects three arguments
  - the first argument is the communicator that we are taking a subset of.
  - the second argument is the group of ranks that will form the subset
  - the third argument is the location of where to initialise and store the communicator.

06) add in the following code in place of the comment marked "code for 06 goes here"

```
// get our rank and size in the new communicator and print that out
int new_rank, new_size;
MPI_Comm_size(sub_comm, &new_size);
MPI_Comm_rank(sub_comm, &new_rank);
std::cout << "rank " << world_rank << " sub comm size and rank: " << new_size << " " <<
new_rank << std::endl;

// send a simple broadcast message on the new communicator with the world rank
int message = world_rank;
MPI_Bcast(&message, 1, MPI_INT, 0, sub_comm);
std::cout << "world rank " << world_rank << " sub comm message is " << message << std::endl;

// delete all user defined communicators and groups
MPI_Comm_free(&sub_comm);
MPI_Group_free(&new_group);

// any memory we allocated must be deallocated
delete group_ranks;

// we are done with the MPI library so we must finalise it
MPI_Finalize();
```

A few things to note here:

- note that communication on sub communicators work in the exact same way as using MPI_COMM_WORLD
- all communication functions that you have used so far will work with these new communicators
- be aware that a process gets a new rank in each communicator it is part of. this rank is unique to this communicator.
- note that at the end of your MPI application you are required to free your communicators and groups before finalising your applicaiton.

App012: using MPI_Scatterv to send different lengths of data to different nodes

Sometimes in an MPI application you will need to send different sizes of data to different nodes. Scatterv is a variant of the normal scatter that will enable us to do this. However, the downside of this approach is that you need to setup an array with the offsets of each partition with a seperate array detailing the length of each partition

01) start with a fresh mpi project

02) add in a main function that looks like this

```
int main(int argc, char **argv) {
	// initialise the MPI libary
	MPI_Init(NULL, NULL);

	// determine the world size and the world rank
	int world_size, world_rank;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// decide which task this node is taking on depending on the rank
	if(world_rank == 0)
		coordinator(world_rank, world_size);
	else
		participant(world_rank, world_size);

	// we are done with the MPI library so we must finalise it
	MPI_Finalize();
}
```

03) add in this function above the main function for printing out the data of an array this will be used in some of the other functions later

```
// function that takes in an integer array and prints it out to console
void printArrayToConsole(int *to_print, int print_size) {
	for(int i = 0; i < print_size; i++) {
		std::cout << to_print[i] << ", ";
		if(i % 10 == 9)
			std::cout << std::endl;
	}
	std::cout << std::endl;
}
```

04) add in the following participant function just above the main function

```
// task for the participant
void participant(int world_rank, int world_size) {
	// take part in the scatter to get our partition size
	int partition_size;
	MPI_Scatter(NULL, 0, MPI_INT, &partition_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
	std::cout << "rank " << world_rank << " psize: " << partition_size << std::endl;

	// allocate memory for our partition
	int *partition = new int[partition_size];

	// take part in the scatterv to get our data
	MPI_Scatterv(NULL, NULL, NULL, MPI_INT, partition, partition_size, MPI_INT, 0, 
MPI_COMM_WORLD);

	// print out the numbers that we have obtained in the scatter
	printArrayToConsole(partition, partition_size);

	// delete our partition as we are finished with it
	delete partition;
}
```

A few things to note here:

- to get the scatterv to work each node in the communicator should know what size of data it is recieving. this is why we have a normal scatter to begin as this is distributing the size of each partition to each node.
    - in this case we set the first two arguments to NULL and 0 because the reciever is not sending any data.
- Once each node knows the size of its partition we can take part in the scatterv
    - like the normal scatter the first 3 arguments are set to null because we are only recieving data. arguments 5 to 9 specify where to store the data, how much, what type who the sender is and what communicator this is working on
- note that we are using the scatterv function here. Normally a node that is a reciever in a scatter will set the first two arguments to NULL and 0 because the reciever is not sending data.

05) add in the following coordinator above the participant function

```
// task for the coordinator
void coordinator(int world_rank, int world_size) {
	// generate a group of 40 numbers randomly
	int *num_array = new int[40];
	for(unsigned int i = 0; i < 40; i++)
		num_array[i] = rand() % 10;
	printArrayToConsole(num_array, 40);

	// make a partition length array with even divisions then randomly offset it by 1
	int *length_array = new int[world_size];
	for(unsigned int i = 0; i < 4; i++) {
		length_array[i] = (40 / world_size);
		if(i % 2)
			length_array[i]--;
		else
			length_array[i]++;
	}
	printArrayToConsole(length_array, 4);

	// make an offsets array by using the displacements array
	int *offsets_array = new int[world_size];
	offsets_array[0] = 0;
	for(unsigned int i = 1; i < 4; i++)
		offsets_array[i] = offsets_array[i-1] + length_array[i-1];
	printArrayToConsole(offsets_array, 4);

	// send each node their individual pariition size and allocate memory for our own 
partition
	int partition_size;
	MPI_Scatter(length_array, 1, MPI_INT, &partition_size, 1, MPI_INT, 0, 
MPI_COMM_WORLD);
	std::cout << "rank 0 psize: " << partition_size << std::endl;
	int *partition = new int[partition_size];

	// use the scatterv to send the data to each node
	MPI_Scatterv(num_array, length_array, offsets_array, MPI_INT, partition, 
partition_size, MPI_INT, 0, MPI_COMM_WORLD);

	// print out the numbers that we have obtained in the scatter
	printArrayToConsole(partition, partition_size);

	// delete our partition as we are finished at it
	delete partition;
	delete num_array;
	delete length_array;
	delete offsets_array;
}
```

A few things to note here:

- the first two loops set up random data and some uneven partition lengths
    - the partition lengths determine how much data to send to each node we must have a length for each and every node
- the following loop sets up offsets in the data to determine where in the array to start sending data to each node.
    - this combined with the lengths array will determine what each and every node will recieve in terms of data
    - for example if we have lengths of 11,9,11,9 and offsets of 0,11,20,31 this will send 11 items to rank zero starting from offset 0. rank 1 will get 9 items of data starting from offset 11 etc etc
- the first of the scatters sends the length of each partition to the appropriate node in the communicator so they can set up their partitions correctly and expect the right size of data
- the scatterv is next the first three arguments are the most important for the root these specify the data that is to be sent to all nodes, the length of data to be sent to each node (must be an array) and the offsets of each partition in the data (must be in an array)

App013 using an MPI_Gatherv to get different sizes of data from all nodes in a communicator

In this example we will show a similar action to App012. Instead of scattering different sizes of data we will gather in different sizes of data. the arguments to the gatherv function are similar to scatterv except this time all nodes are sending and only the root is recieving.

01) start with a new MPI project

02) add in the following main function

```
int main(int argc, char **argv) {
	// initialise the MPI libary
	MPI_Init(NULL, NULL);

	// determine the world size and the world rank
	int world_size, world_rank;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

	// decide which task this node is taking on depending on the rank
	if(world_rank == 0)
		coordinator(world_rank, world_size);
	else
		participant(world_rank, world_size);

	// we are done with the MPI library so we must finalise it
	MPI_Finalize();
}
```

03) add in the following function above the main function

```
// function that takes in an integer array and prints it out to console
void printArrayToConsole(int *to_print, int print_size) {
	for(int i = 0; i < print_size; i++) {
		std::cout << to_print[i] << ", ";
		//if(i % 10 == 9)
		//	std::cout << std::endl;
	}
	std::cout << std::endl;
}
```

04) add in the following participant function

```
// task for the participant
void participant(int world_rank, int world_size) {
	// we need to generate some random data
	int partition_size = (40 / world_size);
	if(world_rank % 2 == 0)
		partition_size++;
	else
		partition_size--;
	int *partition = new int[partition_size];
	srand(world_rank);
	for(unsigned int i = 0; i < partition_size; i++)
		partition[i] = rand() % 10;
	std::cout << "rank " << world_rank << " data";
	printArrayToConsole(partition, partition_size);
	// take part in the gather to let the root know how much data to expect. as a 
participant we
	// dont need to send any data hence arguments 4 and 5 are null and zero
	MPI_Gather(&partition_size, 1, MPI_INT, NULL, 0, MPI_INT, 0, MPI_COMM_WORLD);

	// take part in the gatherv to send all of our data to the root
	MPI_Gatherv(partition, partition_size, MPI_INT, NULL, NULL, NULL, MPI_INT, 0, 
MPI_COMM_WORLD);

	// clean up any data we have allocated
	delete partition;
}
```

A few things to note here:

- each node generates a different partition of data of a different size
- before the master can gather all of this data it must first find out how much data each node is sending. this is why we use the first normal gather
- the gatherv afterwards sets the data that each node will send to the root (defined by the first four arguments)
    - like before with the scatterv the participant is not recieving data which is why the next three arguments are set to null.

05) add in the following coordinator function

```
// task for the coordinator
void coordinator(int world_rank, int world_size) {
	// we need to generate some random data as we are node zero
	int partition_size = (40 / world_size) + 1;
	int *partition = new int[partition_size];
	int *num_array = new int[40];
	for(unsigned int i = 0; i < partition_size; i++)
		partition[i] = rand() % 10;
	std::cout << "rank 0 data";
	printArrayToConsole(partition, partition_size);

	// we need to gather the length of each partition that each node has
	int *lengths_array = new int[world_size];

	// we need to generate the offsets for each partition before we take part in the 
gather
	MPI_Gather(&partition_size, 1, MPI_INT, lengths_array, 1, MPI_INT, 0, 
MPI_COMM_WORLD);
	std::cout << "rank 0 sizes recieved: ";
	printArrayToConsole(lengths_array, world_size);
	int *offsets_array = new int[world_size];
	offsets_array[0] = 0;
	for(unsigned int i = 1; i < 4; i++)
		offsets_array[i] = offsets_array[i-1] + lengths_array[i-1];

	// gather the data from the nodes
	MPI_Gatherv(partition, partition_size, MPI_INT, num_array, lengths_array, 
offsets_array, MPI_INT, 0, MPI_COMM_WORLD);

	// display the data
	std::cout << "rank 0 recieved data: " << std::endl;
	printArrayToConsole(num_array, 40);

	// clean up any memory we have allocated
	delete lengths_array;
	delete partition;
```

```
        delete num_array;
}
```

A few things to note here:

- similar to App012 the root needs to figure out how much data each node is going to send which is why we need the first gather. from this data we need to determine the offsets for where to store the data in our array.
- the gatherv that follows defines what data the root is going to send to itself (first 4 arguments) and where to store all gathered data next 3 arguments
    - the 3 arguments must all be array types that specify the storage area, the amount data to be recieved from each node and where to store each set of data in the array itself.