

Collective Communications

Introduction to Collective Communications

- Thus far all communications we have looked at have been point-to-point
 - Single sending matched to a single receiving process
- Now we will look at collective communications
 - Multiple processes involved in a single communication event
- Same result could be achieved using multiple point to point communications
- ...,but often less efficient
 - In collective communications information can be passed from processor to processor spreading the load

Broadcast using MPI_Bcast

```
int MPI_Bcast(  
    void *buffer,  
    int count,  
    MPI_Datatype datatype,  
    int root,  
    MPI_Comm comm )
```

- Simplest, but one of the most useful collective operations
- Sends the same data from one process to all other processes
 - Needs to be called at the same time by all processes
- MPI_Bcast is blocking
- Function definition looks like an MPI_Send, but the source, rather than the destination, is specified by root
 - The destination is every other process
- On root process root buffer is a pointer to the data to be sent
- On other processes buffer is a pointer to where the sent data is to be stored
 - After the operation buffer will contain the same data on all processes

MPI_Bcast example

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

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

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    srand(time(NULL)+id*10);

    int num_send = 1;
    int *data = new int[num_send];
```

```
    if (id == 0)
        for (int i = 0; i < num_send; i++)
            data[i] = rand();

    MPI_Bcast(data, num_send, MPI_INT, 0, MPI_COMM_WORLD);

    if (id == 0) cout << "Process 0 sent this data: ";
    else cout << "Process " << id << " received this data: ";

    for (int i = 0; i < num_send; i++)
        cout << "\t" << data[i];
    cout << endl;
    cout.flush();

    delete[] data;

    MPI_Finalize();
}
```

Gather and Scatter Communications

- Broadcast sends the same data to all the processes from a single process
 - Note that, in the previous example, `MPI_Bcast` is called on all processes at the same time. This is true of all collective operations.
- Sometimes you wish to send different data to each of the processes, but originating at a single process
 - This is known as a Scatter operation
- The reverse of a Scatter operation is a Gather operation
 - Gather involves sending data from all the processes and collecting it on a single process

MPI_Scatter

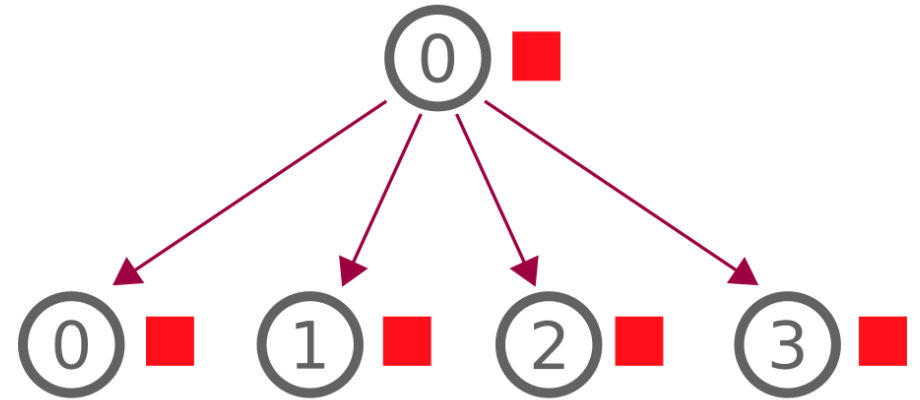
```
int MPI_Scatter(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```

- **send_data** is a pointer to the location of the source data
 - Note that this is only important on the **root** and can be a **nullptr** on other processes
- **send_count** is the number of items to be sent to each process, not the total amount data
- **recv_data** is a pointer to the location where the data is to stored
 - Cannot be null on root as it will also receive its share of the data
- **recv_count** is the number of items expected
 - Generally this would be the same as the **send_count**, but could be different if, for instance, you sent the data as bytes and received then as integers

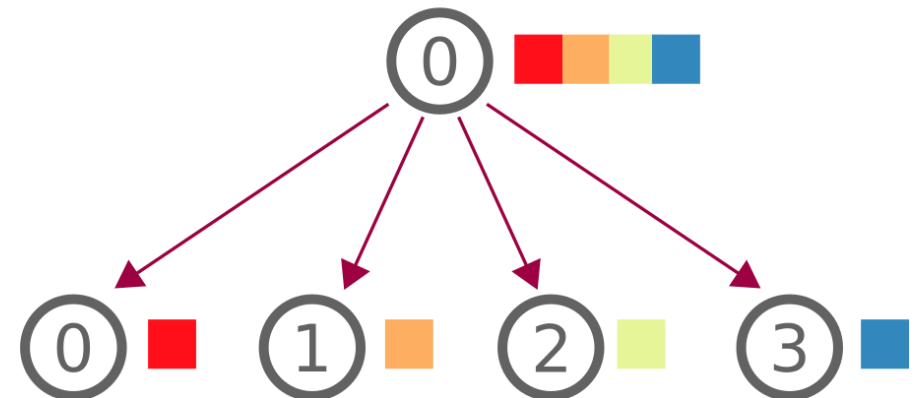
The structure of `send_data`

- With `MPI_Scatter`, the data in `send_data` is sent to processes in the order of the process id
- If `send_count` is greater than one then the first `send_count` items are sent to process zero, the second `send_count` items to process one etc.
- The size of `send_data` should therefore be `send_count` times the total number of processes
 - If you wish to scatter an amount of data not exactly divisible by the number of processes you may need to pad the data
- Note that the data for the root is also copied into `recv_data` on the root

MPI_Bcast



MPI_Scatter



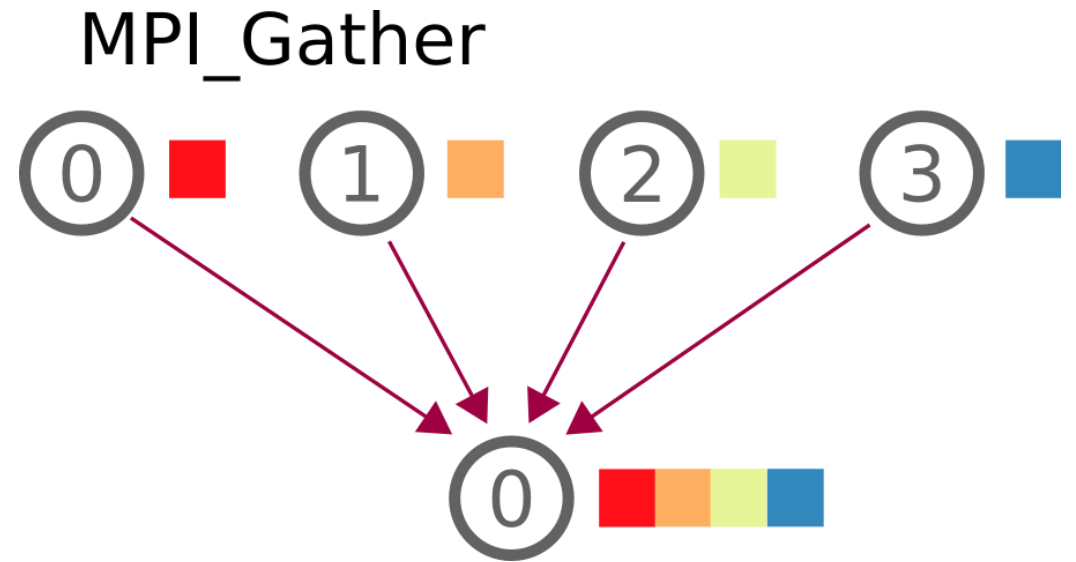
MPI_Gather

```
int MPI_Gather(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```

- The parameters required for `MPI_Gather` are identical to that for `MPI_Scatter`
- The big difference is that now the `send_data` has only got `send_count` items, while `recv_data` will need space for `recv_count` times the number of processes of data
 - This time it is the `recv_data` that can be `NULL` (or `nullptr`) on processes other than the root

The structure of `recv_data`

- As `MPI_Gather` is essentially the reverse operation of `MPI_Scatter`, the structure of the data in `recv_data` will be similar to that in `send_data` from `MPI_Scatter`
 - `recv_data` will have the data sent from process zero stored in the first `recv_count` elements, the data for process one stored in the next `recv_count` elements etc.



An example using MPI_Scatter and MPI_Gather

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

double calc_ave(double *list, int num)
{
    double total = 0.0;
    for (int i = 0; i < num; i++)
        total += list[i];
    return total / num;
}

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

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    srand(time(NULL) + id * 10);

    double *send_data1 = NULL, send_data2;
    double *recv_data1 = NULL, *recv_data2 = NULL;
    int num_sendrecv = 20;
```

```

if (id == 0)
{
    send_data1 = new double[num_sendrecv*p];
    for (int i = 0; i < num_sendrecv*p; i++)
        send_data1[i] = ((double)rand() / (double)RAND_MAX)*100.0;
}

recv_data1 = new double[num_sendrecv];

MPI_Scatter(send_data1, num_sendrecv, MPI_DOUBLE, recv_data1, num_sendrecv, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);

send_data2 = calc_ave(recv_data1, num_sendrecv);

delete[] recv_data1;    //Can be called on all processes as it is NULL where not used
delete[] send_data1;

if (id == 0)
    recv_data2 = new double[p];

MPI_Gather(&send_data2, 1, MPI_DOUBLE, recv_data2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

if (id == 0)
{
    cout << "The following averages were calculated for each set of data:" << endl;
    for (int i = 0; i < p; i++)
        cout << "\t" << i << " " << recv_data2[i] << endl;
}

delete[] recv_data2;
MPI_Finalize();

```

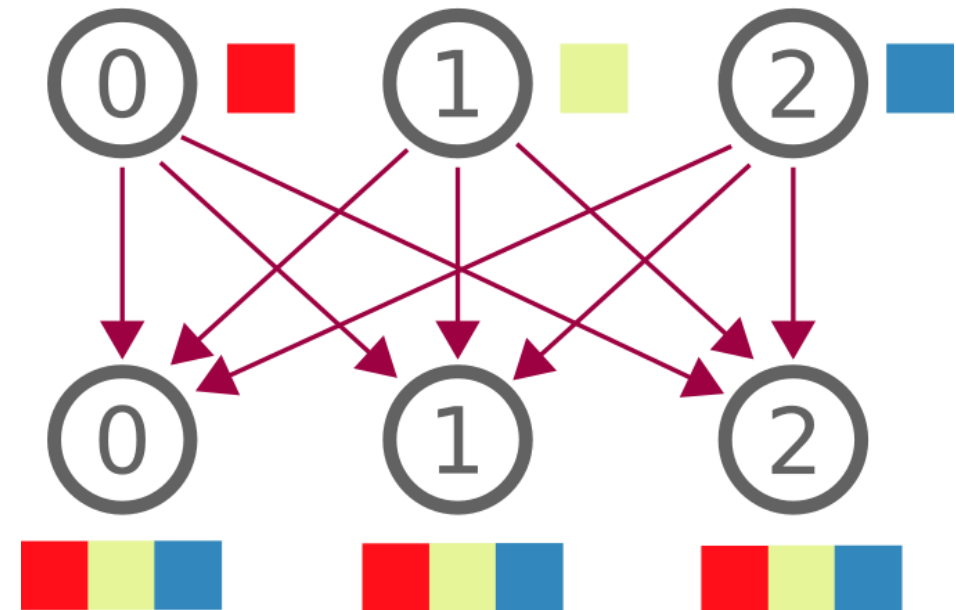
Do Worksheet 3 Exercise 1

MPI_Allgather

```
int MPI_Allgather(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    MPI_Comm communicator)
```

- MPI_Allgather is very closely related to MPI_Gather
 - The difference is that the `recv_data` ends up on all processes
 - A `root` is therefore not required

MPI_Allgather



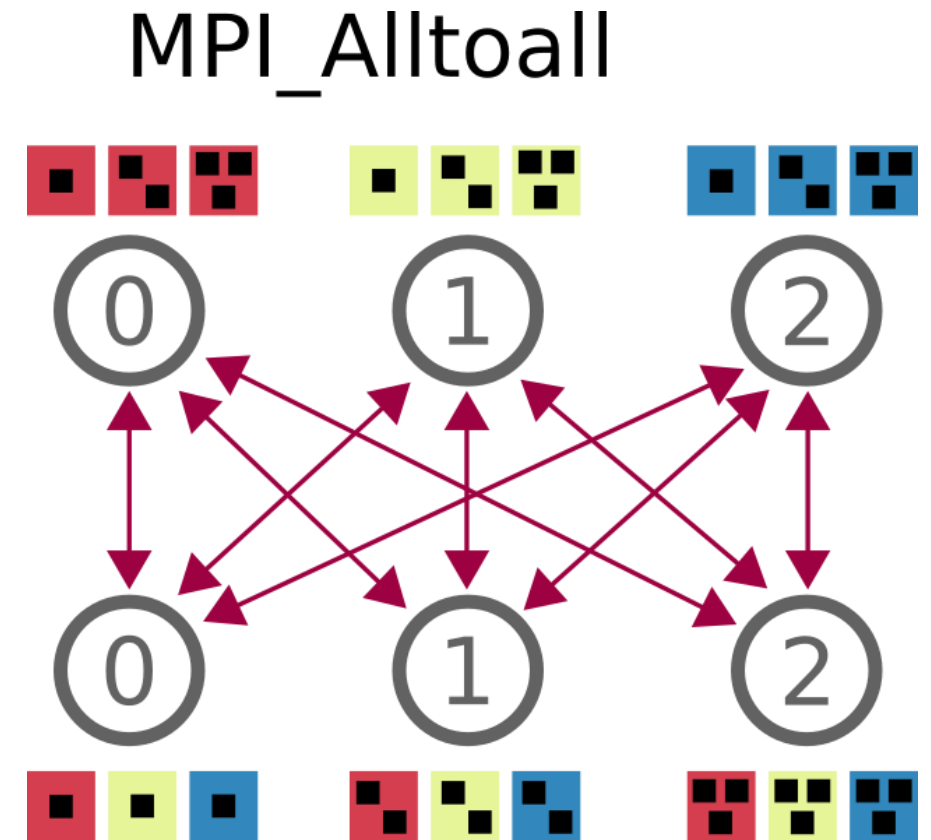
MPI_Alltoall

```
int MPI_Alltoall(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    MPI_Comm communicator)
```

- The parameters in `MPI_Alltoall` are the same as those for `MPI_Allgather`, but what it does is somewhat different

MPI_Alltoall

- Unlike in `MPI_Allgather`, where each process receives the same data, in `MPI_Alltoall` each process sends a different piece of data to each other process
 - The total size of both `send_data` and `recv_data` must be equal to the count times the number of processes
 - Achieves the same thing as the previous lecture's everyone communicating with everyone example, but is more efficient



Transferring data from everyone to everyone

- MPI_Alltoall

```
double *send_data = new double[p*num_send];  
double *recv_data = new double[p *num_send];
```

```
//Set values of send_data
```

```
MPI_Alltoall(send_data , num_send, MPI_DOUBLE, recv_data, num_send,  
             MPI_DOUBLE, MPI_COMM_WORLD);
```

- MPI_Isend and MPI_Irecv

```
double *send_data = new double[p*num_send];  
double *recv_data = new double[p *num_send];
```

```
//Set values of send_data
```

```
for (int i=0;i<p;i++)  
    if (i != id)  
    {  
        MPI_Irecv(&recv_data[i* num_send], num_send, MPI_DOUBLE, i,  
                  tag_num, MPI_COMM_WORLD, &request[cnt]);  
        cnt++;  
    }  
    else for (int j=0;j< num_send;j++)  
        recv_data[i*numsend+j] = send_data[i*numsend+j] ;
```

```
for (int i = 0; i<p; i++)  
    if (i != id)  
    {  
        MPI_Isend(&send_data[i* num_send], num_send, MPI_DOUBLE, i,  
                  tag_num, MPI_COMM_WORLD, &request[cnt]);  
        cnt++;  
    }  
}
```

```
//Potentially do work here while waiting
```

```
MPI_Waitall(cnt, request, MPI_STATUS_IGNORE);
```

MPI_Reduce

- `MPI_Reduce` is a very useful collective operation which shares some similarity to `MPI_Gather` in that data is sent from all the processes and collated on `root`
- Its big difference is that the data is not stored separately, but combined using an operation `op`
 - This is also why there is a single `count` and `datatype` as the operation must involve a single type
 - The operation is applied separately to each of the items in the data
- `MPI_Allreduce` is similar to `MPI_Reduce` except that the data is received on all processes
 - A `root` is therefore not required

```
int MPI_Reduce(  
    void* send_data,  
    void* recv_data,  
    int count,  
    MPI_Datatype datatype,  
    MPI_Op op,  
    int root,  
    MPI_Comm communicator)
```

Reduce operations in MPI (MPI_Op op)

- There are a large number of different reduce operations that are available:

MPI_MAX: Stores the maximum value

MPI_MIN: Stores the minimum value

MPI_SUM: Stores the sum of the values

MPI_PROD: Stores the product of the values

MPI_LAND: Carries out a logical “and” on all the values

MPI_BAND: Carries out a bit-wise “and” on all the values

MPI_LOR: Carries out a logical “or” on all the values

MPI_BOR: Carries out a bit-wise “or” on all the values

MPI_LXOR: Carries out a logical “xor” on all the values

MPI_BXOR: Carries out a bit-wise “xor” on all the values

MPI_MAXLOC: Gives a maximum value and its location (i.e. it returns a pair of values)

MPI_MINLOC: Same as **MPI_MAXLOC**, but based on a minimum value

Modifying the previous example: MPI_Reduce

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

double calc_ave(double *list, int num)
{
    double total = 0.0;
    for (int i = 0; i < num; i++)
        total += list[i];
    return total / num;
}

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

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    srand(time(NULL) + id * 10);
```

```
    double *send_data1 = NULL, send_data2;
    double *recv_data1 = NULL, recv_data2;
    int num_sendrecv = 20;

    if (id == 0)
    {
        send_data1 = new double[num_sendrecv*p];
        for (int i = 0; i < num_sendrecv*p; i++)
            send_data1[i] = ((double)rand() / (double)RAND_MAX)*100.0;
    }

    recv_data1 = new double[num_sendrecv];

    MPI_Scatter(send_data1, num_sendrecv, MPI_DOUBLE, recv_data1, num_sendrecv, MPI_DOUBLE, 0,
                MPI_COMM_WORLD);

    send_data2 = calc_ave(recv_data1, num_sendrecv);

    delete[] recv_data1;    //Can be called on all processes as it is NULL where not used
    delete[] send_data1;

    MPI_Reduce(&send_data2, &recv_data2, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (id == 0)
        cout << "The average of all the data is " << recv_data2 / p << endl;

    MPI_Finalize();
}
```

Do Worksheet 3 Exercise 2 and 3

Non-blocking collectives

- You could use the non-blocking version of the collective operation, which allows you to do things like continuing working while waiting for them to complete
 - We will not specifically look at non-blocking collectives here, but their difference to the blocking version is similar to the difference between, for instance, `MPI_Send` and `MPI_Isend`
 - They also require a request variable and are guaranteed to have completed once `MPI_Wait` has been called
 - They can also be checked for completion using `MPI_Test`
- The non-blocking equivalents all have an I in their name. E.g.
 - Non-blocking `MPI_Bcast` is `MPI_Ibcast`
 - Non-blocking `MPI_Gather` is `MPI_Igather` etc.
- The parameters they take in and their usage are identical to their non-blocking equivalents except for the extra `MPI_Request*` variable

Do Worksheet 3 Exercise 4

In the next lecture we will look at some other things that may be required for writing MPI programs