MPI Implementation

MPI Initialisation

- #include "mpi.h"
  - includes MPI library
- MPI_Init
  - Initialises MPI – defines communicator MPI_COMM_WORLD
- MPI_Comm_size
  - Returns total number of processes (defined at command line)
- MPI_Comm_rank
  - Returns rank (or process number) into proc_num **for this process**
- MPI_Finalize
  - Gracefully ends MPI
- Note - This program does not produce any output

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

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

int proc_num, nprocs;

MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

MPI_Comm_rank(MPI_COMM_WORLD,&proc_num);

MPI_Finalize();

}
```

Run with:
Mpiexec –n x a.out

where ‘x’ is the number of processes and will launch a.out ‘x’ number of times

Hello World

- Here a simple print statement is added ‘Hello World’, with proc_num
- All processes simultaneously execute the same code (SIMD)
- However, which processes executes this command first is non-deterministic
- From the output you can see that running the code on 4 processors can produce different output

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int proc_num, nprocs;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&proc_num);

printf ("Hello World from rank - %d \n", proc_num);

MPI_Finalize();

}
```

```
X:\MPI_Tutorial>mpiexec -host localhost -np 4 \\10.0.0.2\WorkDir\MPI_Tutorial\Test.exe
 Hello World from rank -          1
 Hello World from rank -          0
 Hello World from rank -          3
 Hello World from rank -          2
X:\MPI_Tutorial>RuntestMPI.bat

X:\MPI_Tutorial>mpiexec -host localhost -np 4 \\10.0.0.2\WorkDir\MPI_Tutorial\Test.exe
 Hello World from rank -          2
 Hello World from rank -          1
 Hello World from rank -          3
 Hello World from rank -          0
X:\MPI_Tutorial>
```

Race Conditions

If we add a second print statement of ‘Hello World Again’, in some cases some processes may execute this command before others have even executed the first.

This is known as race conditions and can have very bad consequences.

Multiple process simultaneously trying to write to write to a single file for example.

Sometimes it doesn’t matter or we want this i.e. asynchronous execution or in this case which ever process can sends to the screen buffer first.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int proc_num, nprocs;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&proc_num);

printf ("Hello World from rank - %d \n", proc_num);
printf ("Hello World Again from rank - %d \n", proc_num);

MPI_Finalize();
```

```
X:\MPI_Tutorial>mpiexec -host localhost -np 4 \\10.0.0.2\WorkDir\MPI_Tutorial\Test.exe
 Hello World from rank -          2
 Hello World from rank -          0
 Hello World from rank -          3
 Hello World from rank -          1
 Hello World Again from rank -          0
 Hello World Again from rank -          1
 Hello World Again from rank -          3
 Hello World Again from rank -          2
X:\MPI_Tutorial>RuntestMPI.bat

X:\MPI_Tutorial>mpiexec -host localhost -np 4 \\10.0.0.2\WorkDir\MPI_Tutorial\Test.exe
 Hello World from rank -          3
 Hello World from rank -          2
 Hello World from rank -          0
 Hello World Again from rank -          3
 Hello World from rank -          1
 Hello World Again from rank -          0
 Hello World Again from rank -          1
 Hello World Again from rank -          2
X:\MPI_Tutorial>_
```

Processes

Each process runs the same code.

Each process has it's own private memory area.

So for example if we specify a different value of a for each process.

Then the print command, which all processes execute gives a different value of a

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int proc_num, nprocs;
float a;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&proc_num);

if(proc_num == 0) a = 1.;
if(proc_num == 1) a = 2.;
if(proc_num == 2) a = 10.;
if(proc_num == 3) a = 15.;

printf ("%d %2f \n", proc_num, a);

MPI_Finalize();
}
```

```
2    10.00000
0    1.000000
1    2.000000
3    15.00000
X:\MPI_Tutorial>_
```

MPI_Send and MPI_Recv

The most basic MPI communications can be done with mpi_send and mpi_recv.

This is a blocking routine (more later), but for every mpi_send there has to be an equivalent mpi_recv.

The developer has to tell determine what and when to send

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Input Parameters
buf
initial address of send buffer (choice)
count
number of elements in send buffer (nonnegative integer)
datatype
datatype of each send buffer element (handle)
dest
rank of destination (integer)
tag
message tag (integer)
comm
communicator (handle)

int MPI_Recv(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Status * status)

Output Parameters
buf
initial address of send buffer (choice)
buf
Status object (Status)

Input Parameters
count
number of elements in send buffer (nonnegative integer)
datatype
datatype of each send buffer element (handle)
dest
rank of destination (integer)
tag
message tag (integer)
comm
communicator (handle)

Data Types

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Blocking Send and Receive

In this example i and j are initialised as 0 on both processes

i is then changed to 1 on process 0

The mpi_send and mpi_recv is linked by the if statements.

i from process 0 is sent to process 1

j from process 1 is received from process 0.

Printing i and j before and after the send and receive you can see that j becomes 1 on process 1

Output order ranks for i,j can change.

Order of Sent and Receive will not due to blocking (synchronisation). I.e. both process wait for each other at the mpi_send and mpi_recv subroutines.

```
i=0;
j=0;

if(proc_num == 0) i = 1;

printf ("Rank - %d %d %d \n", proc_num,i,j);

if(proc_num == 0) {
printf ("%d Sending\n", proc_num);
MPI_Send(&i,1,MPI_INT,1,0,MPI_COMM_WORLD);
}

if(proc_num == 1) {
MPI_Recv(&j,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
printf ("%d Recieved\n", proc_num);
}

printf ("Rank - %d %d %d \n", proc_num,i,j);
```

```
X:\MPI_Tutorial>mpiexec -host localhost -np 2
 Rank -          1          0          0
 Rank -          0          1          0
           0 Sent
           1 Recieved
 Rank -          1          0          1
 Rank -          0          1          0
X:\MPI_Tutorial>RuntestMPI.bat

X:\MPI_Tutorial>mpiexec -host localhost -np 2
 Rank -          1          0          0
 Rank -          0          1          0
           0 Sent
           1 Recieved
 Rank -          1          0          1
 Rank -          0          1          0
X:\MPI_Tutorial>RuntestMPI.bat

X:\MPI_Tutorial>mpiexec -host localhost -np 2
 Rank -          0          1          0
 Rank -          1          0          0
           0 Sent
           1 Recieved
 Rank -          1          0          1
 Rank -          0          1          0
X:\MPI_Tutorial>
```

Deadlock

If we comment out the mpi_send call, then the process hangs

Process 1 sits at mpi_recv, but with no message to recv as mpi_send does not occur it waits and so the whole code waits at the next blocking call

Programs hangs

```
if(proc_num == 0) {
printf ("%d Sending\n", proc_num);
\\MPI_Send(&i,1,MPI_INT,1,0,MPI_COMM_WORLD);
}

if(proc_num == 1) {
MPI_Recv(&j,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
printf ("%d Recieved\n", proc_num);
}
```

```
X:\MPI_Tutorial>mpiexec -host localhost -np 2 \\10.0.0.2\WorkDir\MPI_Tutorial\Test.exe
           0 Sent
[mpiexec@MSC-84602] Sending Ctrl-C to processes as requested
[mpiexec@MSC-84602] Press Ctrl-C again to force abort
Terminate batch job (Y/N)? y

X:\MPI_Tutorial>RuntestMPI.bat

X:\MPI_Tutorial>mpiexec -host localhost -np 2 \\10.0.0.2\WorkDir\MPI_Tutorial\Test.exe
           0 Sent
```

Ping pong example

Process 0 and 1 initialised with rank of their neighbour and -1 and 1 for a 'flag'.

Flag changes between -1 and 1 on each process, but they are always different

Send a recv is based on flag value so processes take it in turn sending and receiving

```
1 Sending      1
0 Recieved     1
0 Sending      2
1 Recieved     2
1 Sending      3
0 Recieved     3
0 Sending      4
1 Recieved     4
1 Sending      5
0 Recieved     5
0 Sending      6
1 Recieved     6
1 Sending      7
0 Recieved     7
0 Sending      8
1 Recieved     8
1 Sending      9
0 Recieved     9
0 Sending     10
1 Recieved    10
```

```
ping_pong_count = 0;
if(proc_num == 0) neighbour_num = 1;
if(proc_num == 0) ping_pong_flag = 1;
if(proc_num == 1) neighbour_num = 0;
if(proc_num == 1) ping_pong_flag = -1;

for(ping_pong_count = 1; ping_pong_count <= 10; ping_pong_count++) {
ping_pong_flag = ping_pong_flag * -1;

if(ping_pong_flag == 1) {
printf ("%d Sending %d \n", proc_num, ping_pong_count);
MPI_Send(&i,1,MPI_INT,neighbour_num,0,MPI_COMM_WORLD);
}

if(ping_pong_flag == -1) {
MPI_Recv(&i,1,MPI_INT,neighbour_num,0,MPI_COMM_WORLD,&status);
printf ("%d Recieved %d \n", proc_num, ping_pong_count);
}
}
```

Passing a message forward

Consider sending a message from process 0 -> 1, then from 1->2 … n-2 -> n-1 …

In this example it is worth nothing that

Process 0 is only sending.

Process n-1 is only receiving

Every other process is receiving the sending to its neighbour.

```
if(proc_num == 0) printf ("Number of Procs %d \n", proc_num);

if(proc_num == 0) {
printf ("%d Sending to %d \n", proc_num, proc_num+1);
MPI_Send(&i,1,MPI_INT,proc_num+1,0,MPI_COMM_WORLD);
}

if(proc_num > 0 && proc_num < nprocs-1) {
MPI_Recv(&i,1,MPI_INT,proc_num-1,0,MPI_COMM_WORLD,&status);
printf ("%d Received from %d \n", proc_num, proc_num-1);
printf ("%d Sending to %d \n", proc_num, proc_num+1);
MPI_Send(&i,1,MPI_INT,proc_num+1,0,MPI_COMM_WORLD);
}

if(proc_num == nprocs-1) {
MPI_Recv(&i,1,MPI_INT,proc_num-1,0,MPI_COMM_WORLD,&status);
printf ("%d Received from %d \n", proc_num, proc_num-1);
}
```

```
X:\MPI_Tutorial>mpiexec -host localhost -np 4 \\10.0.0.2\WorkDir\MPI_Tutorial\Test.exe
 Number of Procs -            4
          0 Sending to            1
          1 Received from            0
          1 Sending to            2
          2 Received from            1
          2 Sending to            3
          3 Received from            2
```

Forward then Back

Here we have two similar blocks.

The forward block is identical to the previous example.

The reverse block now has n-1 sending and 0 receiving.

The intermediate processes now receive from +1 and send to -1

```
\\Forwards
if(proc_num == 0) {
printf ("%d Sending to %d \n", proc_num, proc_num+1);
MPI_Send(&i,1,MPI_INT,proc_num+1,0,MPI_COMM_WORLD);
}
if(proc_num > 0 && proc_num < nprocs-1) {
MPI_Recv(&i,1,MPI_INT,proc_num-1,0,MPI_COMM_WORLD,&status);
printf ("%d Received from %d \n", proc_num, proc_num-1);
printf ("%d Sending to %d \n", proc_num, proc_num+1);
MPI_Send(&i,1,MPI_INT,proc_num+1,0,MPI_COMM_WORLD);
}
if(proc_num == nprocs-1) {
MPI_Recv(&i,1,MPI_INT,proc_num-1,0,MPI_COMM_WORLD,&status);
printf ("%d Received from %d \n", proc_num, proc_num-1);
}
\\Backwards
if(proc_num == 0) {
MPI_Recv(&i,1,MPI_INT,proc_num+1,0,MPI_COMM_WORLD,&status);
printf ("%d Received from %d \n", proc_num, proc_num+1);
}
if(proc_num > 0 && proc_num < nprocs-1) {
MPI_Recv(&i,1,MPI_INT,proc_num+1,0,MPI_COMM_WORLD,&status);
printf ("%d Received from %d \n", proc_num, proc_num+1);
printf ("%d Sending to %d \n", proc_num, proc_num-1);
MPI_Send(&i,1,MPI_INT,proc_num-1,0,MPI_COMM_WORLD);
}
if(proc_num == nprocs-1) {
printf ("%d Sending to %d \n", proc_num, proc_num-1);
MPI_Send(&i,1,MPI_INT,proc_num-1,0,MPI_COMM_WORLD);
}
```

```
0 Sending to      1
1 Received from   0
1 Sending to      2
2 Received from   1
2 Sending to      3
3 Received from   2
3 Sending to      2
2 Received from   3
2 Sending to      1
1 Received from   2
1 Sending to      0
0 Received from   1
```

Odd-Even Comms

If information is known on a process and only needs communicating with it's neighbours then multiple send and receives can occur concurrently

In one dimension, a way to do this is by letting the even processes send and the odd receive.

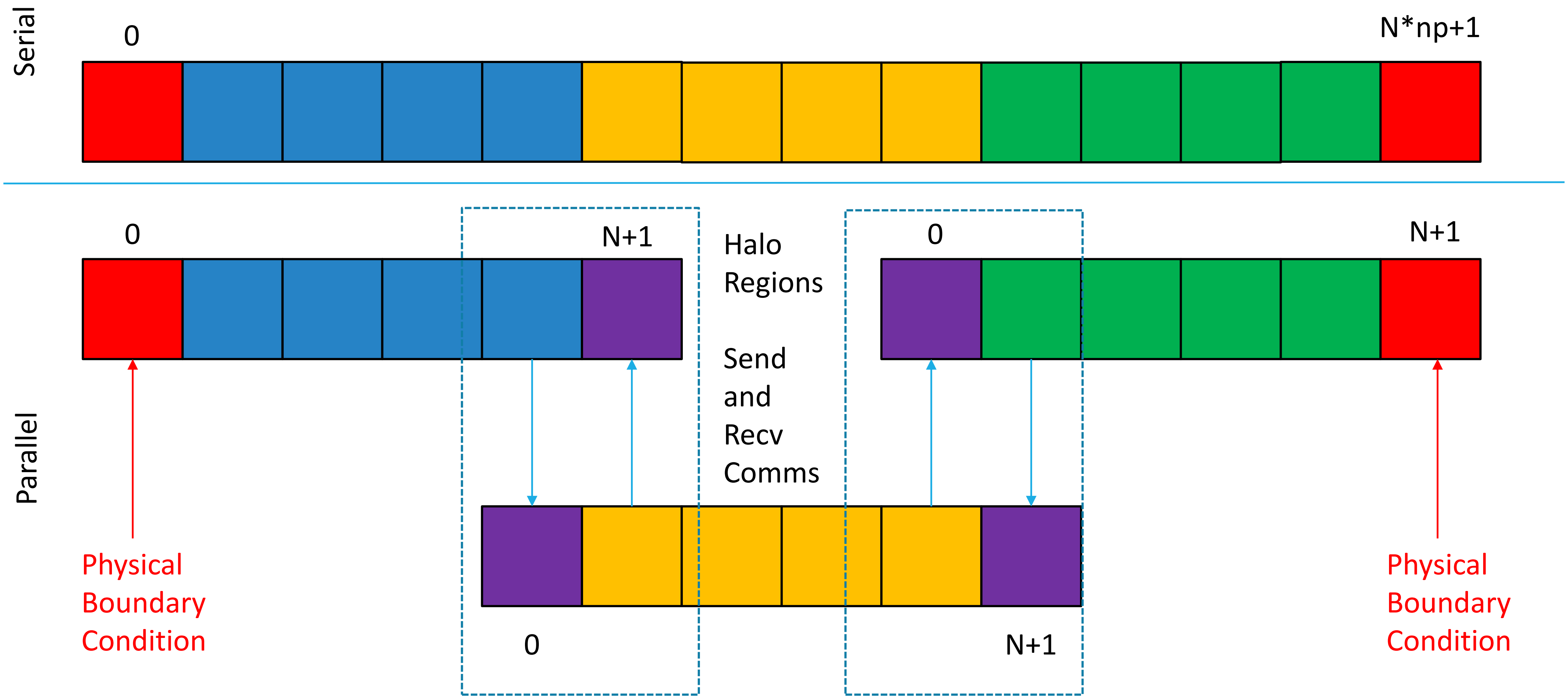
Then followed by the odd processes sending and the even receive.

```
2 Sent to      1
3 Sent to      2
1 Recv from      2
1 Sent to      0
1 Sent to      2
3 Recv from      2
2 Recv from      3
2 Recv from      1
2 Sent to      3
0 Recv from      1
0 Sent to      1
1 Recv from      0
```

```
WNei = proc_num - 1;
ENei = proc_num + 1;
if (proc_num == nprocs-1) ENei=-1;  /*For last processor set ENei to -1*/

if(proc_num % 2 == 0) {  /*For even Processor Numbers*/
if(WNei >= 0) MPI_Send(&a,1,MPI_DOUBLE,WNei,WNei, MPI_COMM_WORLD);  /*Sd to W*/
if(WNei >= 0) printf("%d Sent to %d \n", proc_num, WNei);
if(ENei >= 0) MPI_Recv(&a,1,MPI_DOUBLE,ENei,proc_num, MPI_COMM_WORLD, &status); /*Rv fm E*/
if(ENei >= 0) printf("%d Recv from %d \n", proc_num, ENei);
if(WNei >= 0) MPI_Recv(&a,1,MPI_DOUBLE,WNei,proc_num, MPI_COMM_WORLD, &status); /*Rv fm W*/
if(WNei >= 0) printf("%d Recv from %d \n", proc_num, WNei);
if(ENei >= 0) MPI_Send(&a,1,MPI_DOUBLE,ENei,ENei, MPI_COMM_WORLD); /*Sd to E*/
if(ENei >= 0) printf("%d Sent to %d \n", proc_num, ENei);
}
else {
if(ENei >= 0) MPI_Recv(&a,1,MPI_DOUBLE,ENei,proc_num, MPI_COMM_WORLD, &status); /*Rv fm E*/
if(ENei >= 0) printf("%d Recv from %d \n", proc_num, ENei);
if(WNei >= 0) MPI_Send(&a,1,MPI_DOUBLE,WNei,WNei, MPI_COMM_WORLD); /*Sd to W*/
if(WNei >= 0) printf("%d Sent to %d \n", proc_num, WNei);
if(ENei >= 0) MPI_Send(&a,1,MPI_DOUBLE,ENei,ENei, MPI_COMM_WORLD); /*Sd to E*/
if(ENei >= 0) printf("%d Sent to %d \n", proc_num, ENei);
if(WNei >= 0) MPI_Recv(&a,1,MPI_DOUBLE,WNei,proc_num, MPI_COMM_WORLD, &status); /*Rv fm W*/
if(WNei >= 0) printf("%d Recv from %d \n", proc_num, WNei);
}
```

Domain Decomposition for a 1D problem



MPI_Broadcast

MPI_BCAST broadcasts a message to all other processes.

I.e. All processes end up with the same value as the sender (root).

In the example below each process has a assigned to their process number.

Process 2 is the root and broad casts it's value to all other processes.

a then becomes 2 on all processes.

This could be done with send/recv, i.e. from passing in a ring forward. But is it optimal?

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

Input/Output Parameters

Buffer - starting address of buffer (choice)

Input Parameters

Count- number of entries in buffer (integer)
Datatype - data type of buffer (handle)
Root - rank of broadcast root (integer)
Comm - communicator (handle)

```
a= proc_num;

printf("rank %d a %d \n", proc_num, a);
MPI_Bcast(&a,1,MPI_INT,2,MPI_COMM_WORLD);
printf("rank %d a2 %d \n", proc_num, a);
```

```
X:\MPI_Tutorial>mpiexec -host localhost -np 4
 rank           1 a   1.000000
 rank           0 a  0.0000000E+00
 rank           2 a   2.000000
 rank           3 a   3.000000
 rank           0 a2   2.000000
 rank           1 a2   2.000000
 rank           2 a2   2.000000
 rank           3 a2   2.000000
```

MPI_AllReduce

MPI_AllReduce - Combines values from all processes and distributes the result back to all processes

So for example if you want to find the sum value across all processes

Or the min/max?

Consider that in the Jacobi/gauss seidel codes the while loop only exists when a tolerance has been met.

However, this may occur on one process, but NOT on others. This would lead to a deadlock as a process exits loop but it neighbour doesn't.

int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Input Parameters

Sendbuf - starting address of send buffer (choice)
Count - number of elements in send buffer (integer)
Datatype - data type of elements of send buffer (handle)
Op - operation (handle)
Comm - communicator (handle)

Output Parameters

Recvbuf - starting address of receive buffer (choice)

```
a = proc_num;
b = 0;

printf("rank %d a %d %d \n", proc_num, a,b);
MPI_Allreduce(&a,&b,1,MPI_INT,MPI_MAX,MPI_COMM_WORLD);
printf("rank %d a2 %d %d \n", proc_num, a,b);
```

MPI_MAX

```
rank           0 a  0.0000000E+00  0.0000000E+00
rank           1 a   1.000000      0.0000000E+00
rank           3 a   3.000000      0.0000000E+00
rank           2 a   2.000000      0.0000000E+00
rank           1 a2   1.000000       3.000000
rank           0 a2  0.0000000E+00   3.000000
rank           2 a2   2.000000       3.000000
rank           3 a2   3.000000       3.000000
```

MPI_SUM

```
rank           1 a   1.000000      0.0000000E+00
rank           2 a   2.000000      0.0000000E+00
rank           3 a   3.000000      0.0000000E+00
rank           0 a  0.0000000E+00  0.0000000E+00
rank           3 a2   3.000000       6.000000
rank           0 a2  0.0000000E+00   6.000000
rank           1 a2   1.000000       6.000000
rank           2 a2   2.000000       6.000000
```