Message Passing Paradigm
MPI : Distributed memory systems

➢ MPI_Send

- *Basic blocking send operation.*

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

➢ MPI_Recv

*MPI_Recv(void *recv_buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)*

- Data to be sent by the sending processes is copied from the user memory space to the system buffer
- The data is sent from the system buffer over the network to the system buffer of receiving process
- The receiving process copies the data from system buffer to local user memory space

Communication modes has both **blocking and non-blocking primitives**

- In blocking point to point communication the send call blocks until the send block can be reclaimed. Similarly the receive function blocks until the buffer has successfully obtained the contents of the message.
- In the non-blocking point to point communication the send and receive calls allow the possible overlap of **communication with computation**.
- Communication is usually done in 2 phases: the posting phase and the test for completion phase.

MPI communication

- **Synchronization Overhead**: the time spent in waiting for an event to occur on another task.

- **System Overhead**: the time spent when copying the message data from sender's message buffer to network and from network to the receiver's message buffer.

Sending and Receiving Messages

- MPI allows specification of wildcard arguments for both source and tag.
- If source is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If tag is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- On the receive side, the message must be of length equal to or less than the length field specified.

Sending and Receiving Messages

- In the receiving process- the status variable can be used to get information about the `MPI_Recv` operation.
- The corresponding data structure contains:

```
typedef struct MPI_Status {
        int MPI_SOURCE;
        int MPI_TAG;
        int MPI_ERROR; };
```

- The MPI_Get_count function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
   datatype, int *count)
```

Avoiding Deadlocks

Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If MPI_Send is blocking, there is a deadlock.

Sending and Receiving Messages Simultaneously

To exchange messages, MPI provides the following function:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
 MPI_Datatype senddatatype, int dest, int
sendtag, void *recvbuf, int recvcount,
MPI_Datatype recvdatatype, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

Overlapping Communication with Computation

- Overlap communication with computation - **non-blocking send and receive operations**.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm,
              MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

- These operations return before the operations have been completed. Function `MPI_Test` tests whether or not the non-blocking send or receive operation identified by its request has finished.

```
int MPI_Test(MPI_Request *request, int *flag,
        MPI_Status *status)
```

Collective Communication & Computation Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.

Collective Calls

- A communication pattern that involves all processes within a communicator is known as **collective communication**

- MPI has several collective communication calls:
  - *Synchronization - processes wait until all members of the group have reached the synchronization point.*
    - Barrier
  - *Communication/ Data Movement*
    - Broadcast, scatter/gather, all to all.
  - *Reduction / Collective Computation*
    - Reduce

- All collective communication is blocking

MPI Collective Calls : Barrier

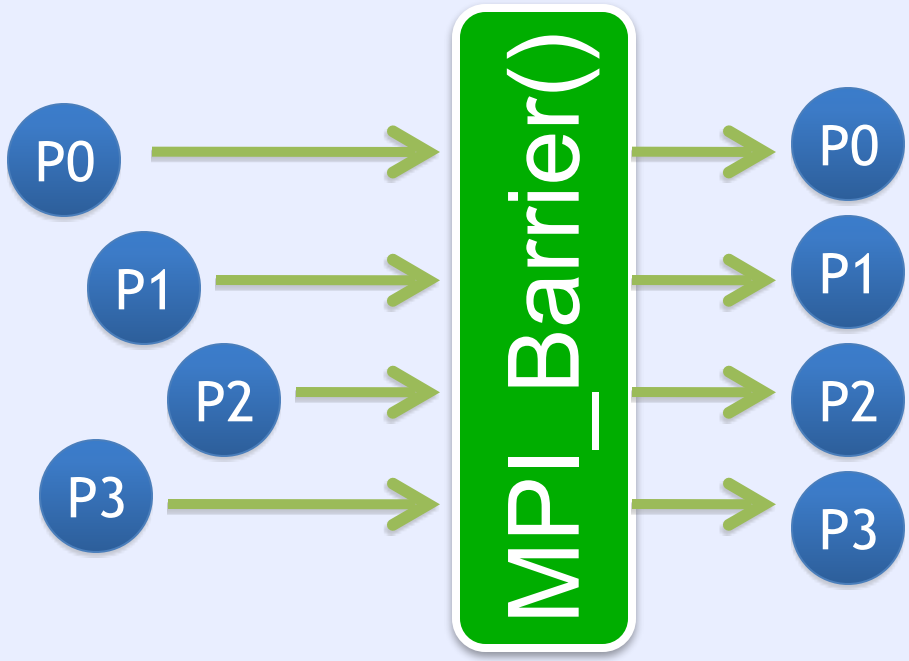
Function: MPI_Barrier()

```
int MPI_Barrier (
        MPI_Comm comm )
```

Description:

Creates barrier synchronization in a communicator group *comm*.

Each process, when reaching the MPI_Barrier call, blocks until all the processes in the group reach the same MPI_Barrier call.

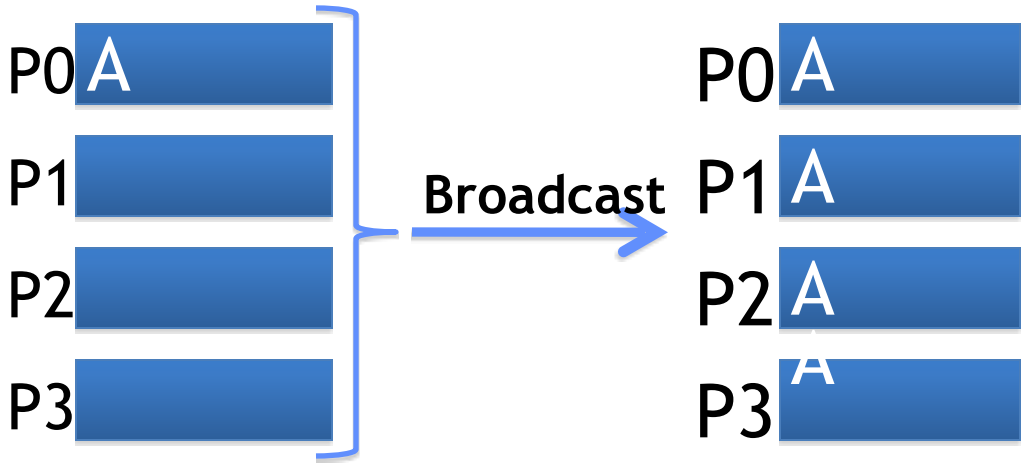


Ref: http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Barrier.html

MPI Collective Calls : Broadcast

Function: MPI_Bcast()

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



A collective communication call where a single process sends the same data contained in the *message* to every process in the communicator.

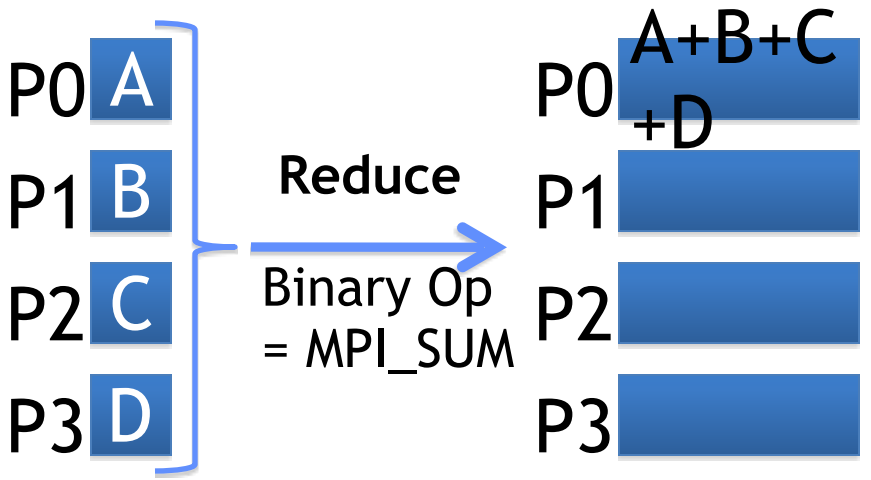
All the processes invoke the *MPI_Bcast* call with the same arguments for root and *comm*,

```
 float       endpoint[2];
...
 MPI_Bcast(endpoint, 2, MPI_FLOAT, 0, MPI_COMM_WORLD);
...
```

MPI Collective Calls : Reduce

Function: MPI_Reduce()

```
int MPI_Reduce (
      void          *operand,
      void          *result,
      int           count,
      MPI_Datatype  datatype,
      MPI_Op        operator,
      int           root,
      MPI_Comm      comm )
```



A collective communication call where all the processes in a communicator contribute data that is combined using binary operations (MPI_Op) such as addition, max, min, logical, and, etc.

```
...
MPI_Reduce(&local_integral, &integral, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
...
```

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

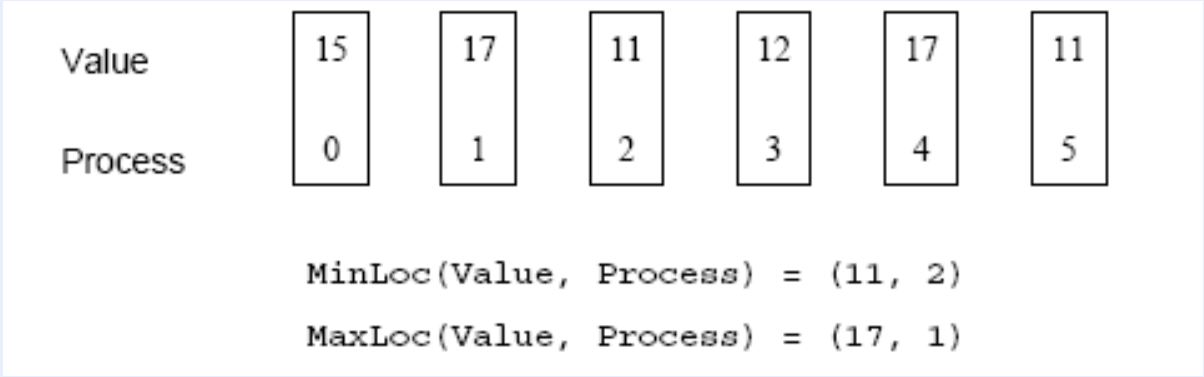
Collective Functions

- *Let us consider there are 4 processes*
- *Rank 1 is designated - data gather/scatter process*
- *a, b, c, d are scalars/ arrays of any data type*
- *Data are gathered/scattered according to rank order*

Process 0	Process 1*	Process 2	Process 3	Operation	Process 0	Process 1*	Process 2	Process 3
	b			*MPI_Bcast*	b	b	b	b
a	b	c	d	*MPI_Gather*		a,b,c,d		
a	b	c	d	*MPI_Allgather*	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
	a,b,c,d			*MPI_Scatter*	a	b	c	d

Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.



An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

Ref: Introduction to Parallel Programming; Ananth Grama, A. Gupta, G. Karypis and V Kumar

Collective Communication Operations

- If the result of the reduction operation is needed by all processes, MPI provides:

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

- To compute prefix-sums, MPI provides:

```
int MPI_Scan(void *sendbuf, void *recvbuf,
int count, MPI_Datatype datatype, MPI_Op
op, MPI_Comm comm)
```

Trapezoidal Rule : with MPI_Bcast, MPI_Reduce

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

/* We'll be using MPI routines, definitions, etc.
*/
#include "mpi.h"


main(int argc, char** argv) {
   int        my_rank;   /* My process rank
*/
   int        p;        /* The number of processes
*/
   float      endpoint[2];  /* Left and right
*/
   int        n = 1024;  /* Number of trapezoids
*/
   float      h;        /* Trapezoid base length
*/
   float      local_a;   /* Left endpoint my
process  */
   float      local_b;   /* Right endpoint my
process */
   int        local_n;   /* Number of trapezoids
for  */
                         /* my calculation        */
```

Trapezoidal Rule : with MPI_Bcast, MPI_Reduce

```
 float Trap(float local_a, float local_b, int local_n,float h);    /* Calculate local
integral  */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);


    if (my_rank==0) {
      endpoint[0] = atof(argv[1]);   /*  left endpoint */
      endpoint[1] = atof(argv[2]);   /* right endpoint */
    }
```

```
MPI_Bcast(endpoint, 2, MPI_FLOAT, 0,MPI_COMM_WORLD);
```

Trapezoidal Rule : MPI_Bcast, MPI_Reduce

```
    h = (endpoint[1]-endpoint[0])/n;              /* h is the same for all processes */
    local_n = n/p;                                /*  so is the number of
trapezoids */
    if (my_rank == 0)
printf("a=%f, b=%f, Local number of trapezoids=%d\n", endpoint[0], endpoint[1],
local_n );

    local_a = endpoint[0] + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

MPI_Reduce(&integral, &total, 1, MPI_FLOAT, MPI_SUM, 0,
MPI_COMM_WORLD);

if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",n);
        printf("of the integral from %f to %f = %f\n",endpoint[0], endpoint[1], total);
    }

MPI_Finalize();
} /*  main  */
```

Common Errors/ Debugging

- Program hangs
  - *Send has no corresponding receive.*
  - *Receive has no corresponding send.*
  - *Send/receive pair do not match in source/recipient or tag*
  - *Condition does not occur the way programmer expect it to occur.*

- Segmentation fault
  - *Trying to access memory which is not allowed to access (e.g. array index out-of-bounds, using non-pointer as pointer)*

- Debugging parallel codes is particularly difficult
- Problem: figuring out what happens on each node
- Solutions: Print statements in each node, Debuggers compatible with MPI

MPI compilation and execution

- $ mpicc <src_file> -o <name_of_executable>

- $ mpiexec -n 24 -machinefile machines ./a.out

- Put the contents in the "**machines**" as follows:

ipaddress slots=8
ipaddress slots=8
ipaddress slots=8