

COMP 322: Fundamentals of Parallel Programming

Lecture 29: Introduction to the Message Passing Interface (MPI) cont.

Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@rice.edu

<http://comp322.rice.edu>



Worksheet #28 solution: MPI send and receive

```
1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0, 1);
10.    System.out.println("a = " + a + " ; b = " + b);
11.}
12. ...
```

Question: In the space below, indicate what values you expect the print statement in line 10 to output (assuming the program is invoked with 2 processes).

Answer: Nothing! The program will deadlock due to mismatched tags, with process 0 blocked at line 4, and process 1 blocked at line 8.



mpiJava vs. OpenMPI Java API

- mpiJava is a standalone and prototype Java library developed 10+ years ago as part of the HPJava project at Indiana University
- OpenMPI is a large consortium of universities/companies building an open-source implementation of the MPI programming model
 - Recently added Java APIs, similar to mpiJava (but more modern)
 - We will use mpiJava in lecture slides, but OpenMPI for Lab 10 and Homework 5

	mpiJava	OpenMPI Java API
Package name	package mpi	package mpi
Main class	mpi.MPI	mpi.MPI
Get MPI Rank	MPI.COMM_WORLD.Rank()	MPI.COMM_WORLD.getRank()
Get # MPI Ranks	MPI.COMM_WORLD.Size()	MPI.COMM_WORLD.getSize()
Send MPI Msg	MPI.COMM_WORLD.Send(...)	MPI.COMM_WORLD.send(...)
Recv MPI Msg	MPI.COMM_WORLD.Recv(...)	MPI.COMM_WORLD.recv(...)



Outline of today's lecture

- Blocking communications (contd)
- Non-blocking communications
- Collective communications



Basic Datatypes

- mpiJava defines 9 basic datatypes: these correspond to the 8 primitive types in the Java language, plus the MPI.OBJECT datatype that stands for an Object (or, more formally, a Java reference type).
 - MPI.OBJECT value can only be dereferenced on process where it was created
- The basic datatypes are available as static fields of the MPI class. They are:

mpiJava datatype	Java type
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.OBJECT	Object



Communication Buffers

- Most of the communication operations take a sequence of parameters like
Object buf, int offset, int count, Datatype type
- In the actual arguments passed to these methods, buf must be an array (or a run-time exception will occur).

Would need to override with 8 versions of methods using 1 buffer

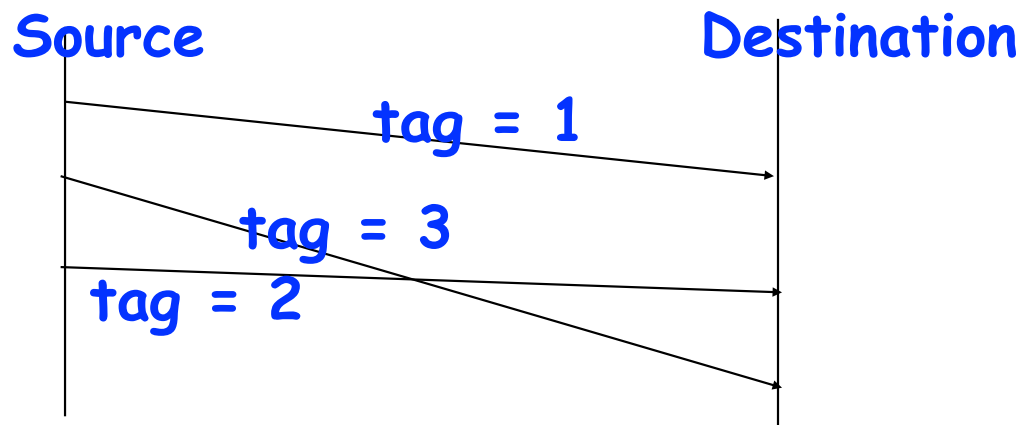
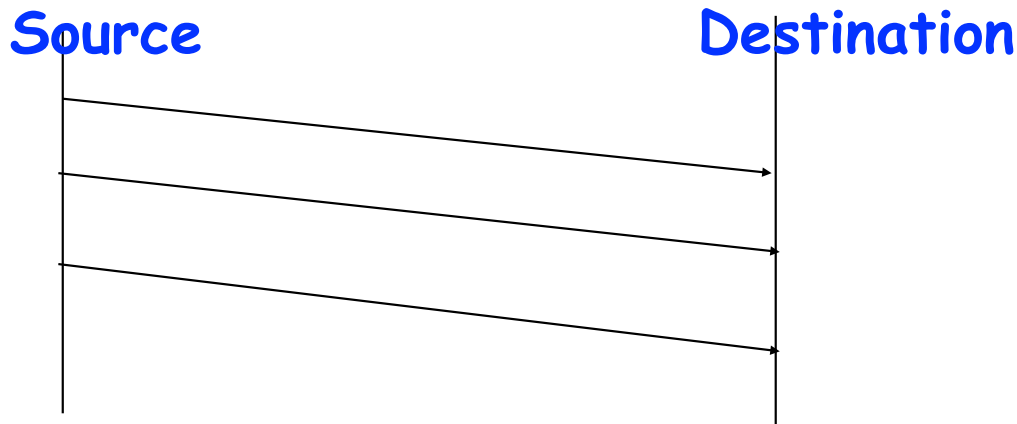
```
void Send(int[] buf, ...)  
void Send(long[] buf, ...)
```

Would need to override with 64 versions of methods using 2 buffers

```
void Reduce(int[] sbuf, ...int[] rbuf)  
void Reduce(int[] sbuf, ...long[] rbuf)
```



Message Ordering in MPI



- FIFO ordering only guaranteed for same source, destination, data type, and tag
- In HJ actors, FIFO ordering was guaranteed for same source and destination
 - Actor send is also “one-sided” and “non-blocking” (unlike send/recv in MPI)



Layout of Buffer

- If type is a basic datatype (corresponding to a Java type), the message corresponds to a subset of the array buf, defined as follows:



- In the case of a send buffer, the red boxes represent elements of the buf array that are actually sent.
- In the case of a receive buffer, the red boxes represent elements where the incoming data may be written.



Scenario #1

Consider:

```
int a[], b[];
...
if (MPI.COMM_WORLD.rank() == 0) {
    MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
    MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
}
else {
    Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
    Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0, 1);
}
...
```

Blocking semantics for `Send()` and `Recv()` can lead to a deadlock.



Approach #1 to Deadlock Avoidance --- Reorder Send and Recv calls

We can break the circular wait in the worksheet by reordering Recv() calls to avoid deadlocks as follows:

```
int a[], b[];
...
if (MPI.COMM_WORLD.rank() == 0) {
    MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
    MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
}
else {
    Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0, 1);
    Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
}
...
```



Scenario #2

Consider the following piece of code, in which process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (modulo the number of processes)

```
1. int a[], b[];
2. . . .
3. int npes = MPI.COMM_WORLD.size();
4. int myrank = MPI.COMM_WORLD.rank();
5. MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, (myrank+1)%npes, 1);
6. MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, (myrank+npes-1)%npes, 1);
```

Question: does this MPI code deadlock?



Approach #2 to Deadlock Avoidance --- a combined Sendrecv() call

- Since it is fairly common to want to simultaneously send one message while receiving another.
- In mpiJava, the Sendrecv() method has the following signature:

```
Status Sendrecv(Object sendBuf, int sendOffset, int sendCount,  
                Datatype sendType, int dst, int sendTag,  
                Object recvBuf, int recvOffset, int recvCount,  
                Datatype recvType, int src, int recvTag) ;
```

More efficient than separate sends
and receives

Can avoid deadlock

— There is also a variant called Sendrecv_replace() which only specifies a single buffer



Using Sendrecv for Deadlock Avoidance in Scenario #2

Consider the following piece of code, in which process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (modulo the number of processes)

```
int a[], b[];
. . .
int npes = MPI.COMM_WORLD.size();
int myrank = MPI.COMM_WORLD.rank()
MPI.COMM_WORLD.Sendrecv(a, 0, 10, MPI.INT, (myrank+1)%npes, 1,
                        b, 0, 10, MPI.INT, (myrank+npes-1)%npes, 1);
. . .
```

A combined Sendrecv() call avoids deadlock in this case



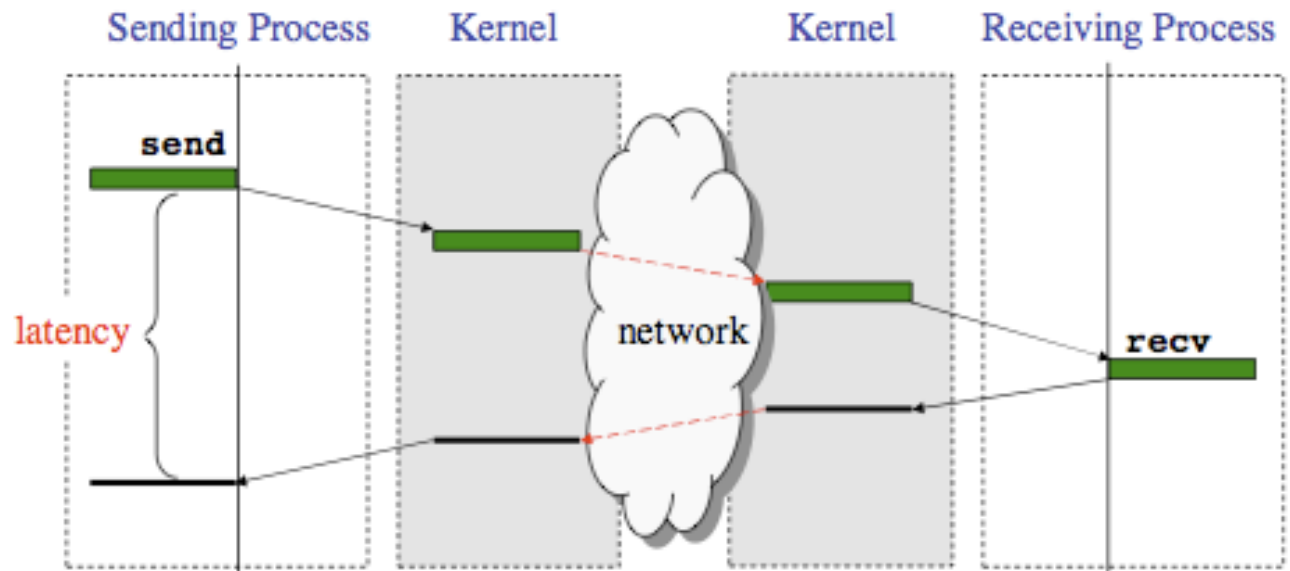
Outline of today's lecture

- **Blocking communications (contd)**
- **Non-blocking communications**
- **Collective communications**

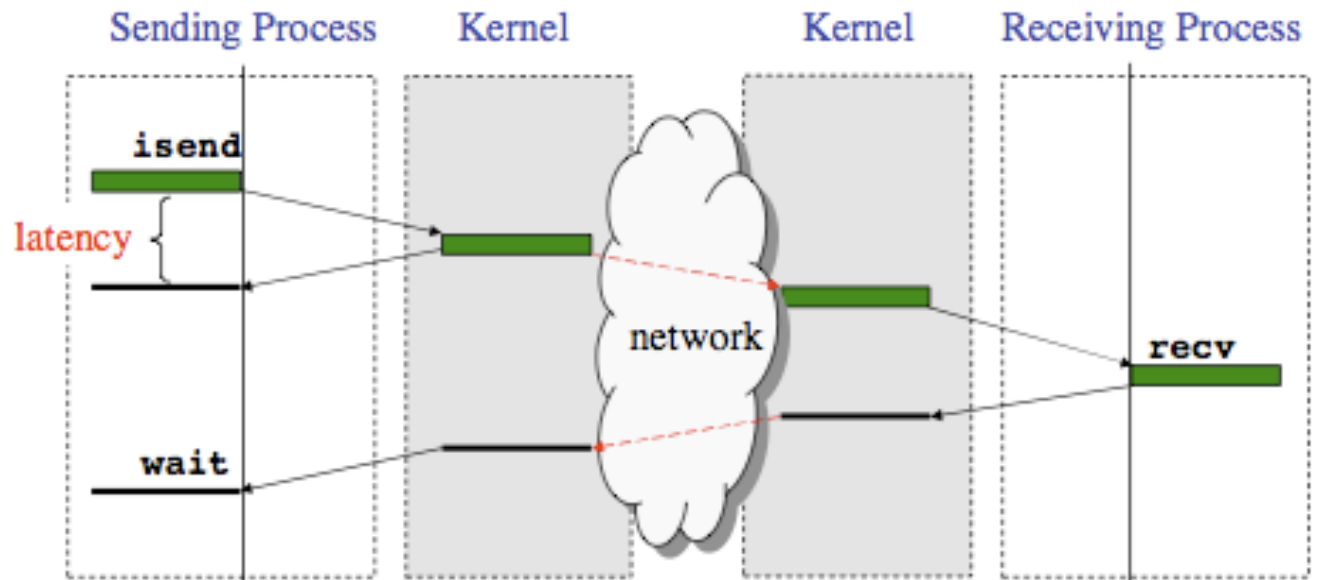


Latency in Blocking vs. Nonblocking Communication

Blocking communication



Nonblocking communication (like an async or future task)



Non-Blocking Send and Receive operations

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations (“I” stands for “Immediate”)

`Request Isend(Object buf, int offset, int count, Datatype type, int dst, int tag) ;`
`Request Irecv(Object buf, int offset, int count, Datatype type, int src, int tag) ;`

- Use `Wait()` to wait for operation to complete (like future get).

`Status Wait(Request request)`

- The `Wait()` operation is declared to return a `Status` object. In the case of a non-blocking receive operation, this object has the same interpretation as the `Status` object returned by a blocking `Recv()` operation.



Simple Irecv() example

- The simplest way of waiting for completion of a single non-blocking operation is to use the instance method `Wait()` in the `Request` class, e.g:

```
// Post a receive (like a “communication async”)
Request request = Irecv(intBuf, 0, n, MPI.INT,
                       MPI.ANY_SOURCE, 0) ;
// Do some work while the receive is in progress
...
// Wait for message to arrive (like a future get)
Status status = request.Wait() ;
// Do something with data received in intBuf
...
```



Waitall() vs. Waitany()

```
public static Status[] Waitall (Request [] array_of_request)
```

- **Waitall()** blocks until all operations associated with the active requests have completed.
- Returns an array of statuses for each of the requests.
 - **Waitall()** is a like a finish scope for all requests in the array

```
public static Status Waitany(Request [] array_of_request)
```

- **Waitany()** blocks until one of the operations associated with the active requests has completed.
 - **Source of nondeterminism**



Outline of today's lecture

- **Blocking communications (contd)**
- **Non-blocking communications**
- **Collective communications**



Collective Communications

- A popular feature of MPI is its family of collective communication operations.
- Each collective operation is defined over a communicator (most often, MPI.COMM_WORLD)
- Each collective operation contains an *implicit barrier*. The operation completes and execution continues when all processes in the communicator perform the *same* collective operation.
- A mismatch in operations results in *deadlock* e.g.,
Process 0: MPI.Bcast(...)
Process 1: MPI.Bcast(...)
Process 2: MPI.Gather(...)
- A simple example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

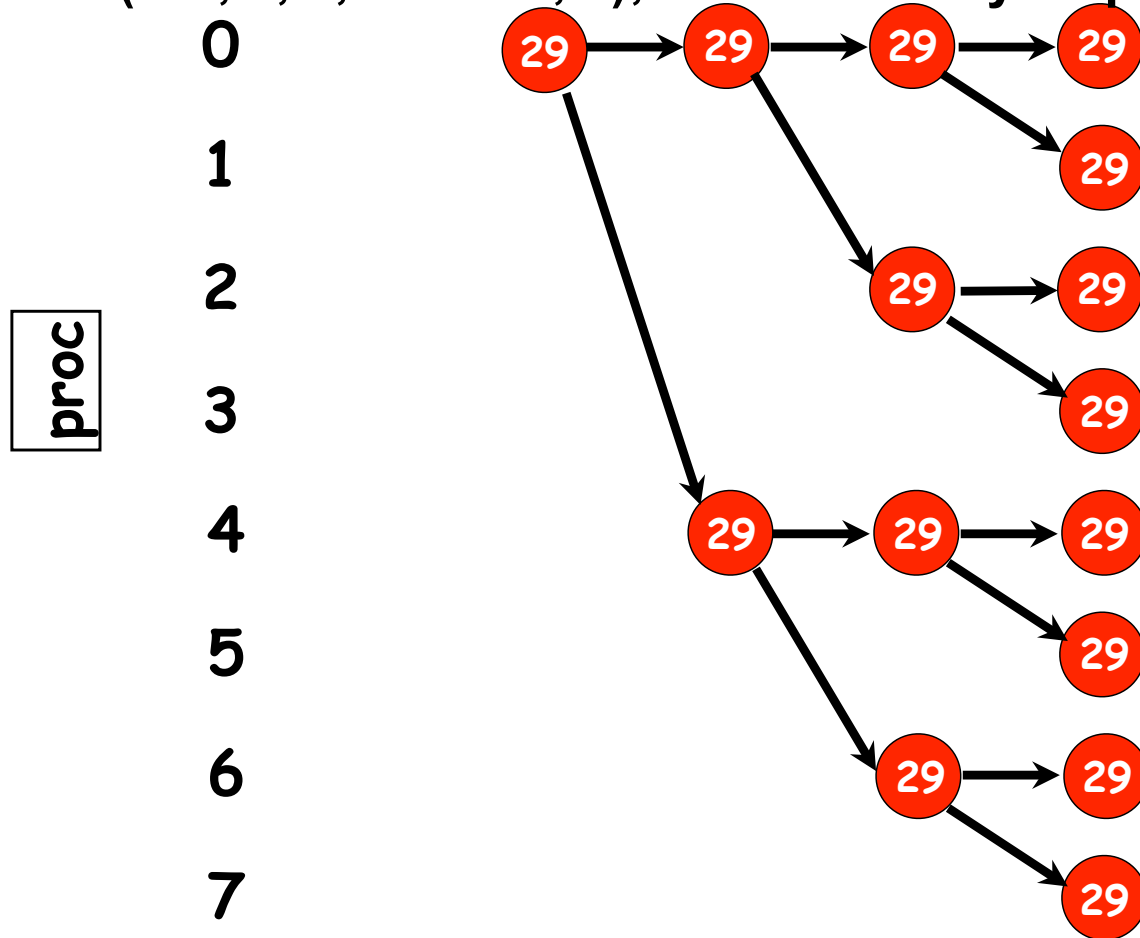
```
void Bcast(Object buf, int offset, int count, Datatype type, int root)
```



MPI_Bcast

```
buf = new int[1]; if (rank==0) buf[0] = 29;
```

```
void Bcast(buf, 0, 1, MPI.INT, 0); // Executed by all processes
```



A root process sends same message to all

29 represents an array of values

Broadcast can be implemented as a tree by MPI runtime



More Examples of Collective Operations

void Gather(Object sendbuf, **int** sendoffset, **int** sendcount, Datatype sendtype, Object recvbuf, **int** recvoffset, **int** recvcount, Datatype recvtype, **int** root)

- Each process sends the contents of its send buffer to the root process.

void Scatter(Object sendbuf, **int** sendoffset, **int** sendcount, Datatype sendtype, Object recvbuf, **int** recvoffset, **int** recvcount, Datatype recvtype, **int** root)

- Inverse of the operation Gather.

void Reduce(Object sendbuf, **int** sendoffset, Object recvbuf, **int** recvoffset, **int** count, Datatype datatype, Op op, **int** root)

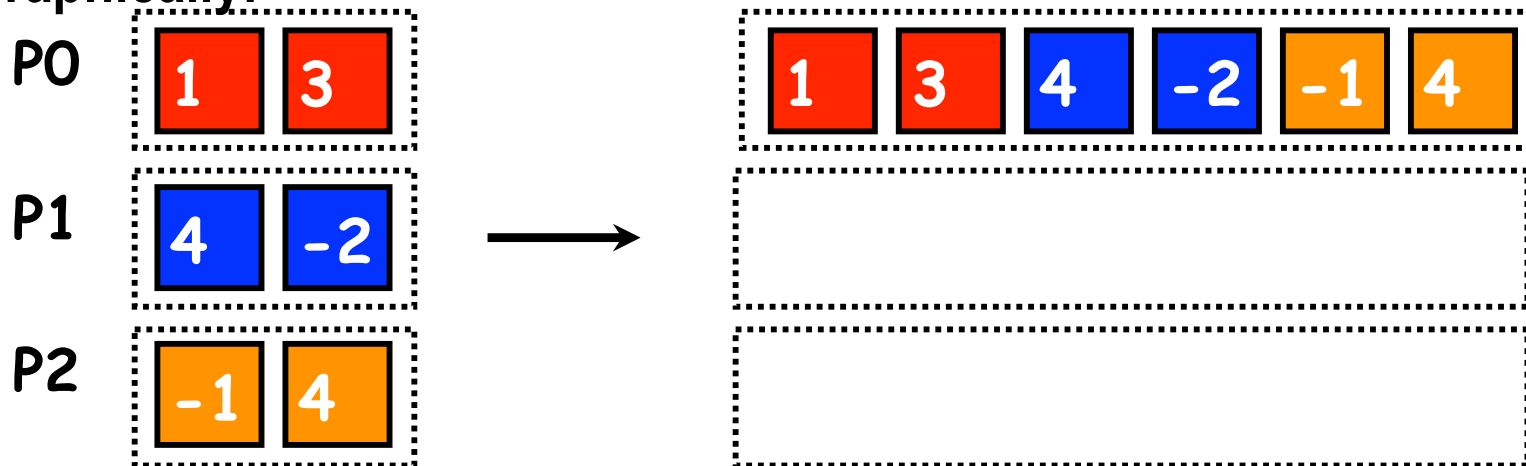
- Combine elements in send buffer of each process using the reduce operation, and return the combined value in the receive buffer of the root process.



MPI_Gather

- Use to copy an array of data from each process into a single array on a single process.

- Graphically:



- Note: only process 0 (P0) needs to supply storage for the output
`void Gather(Object sendbuf, int sendoffset, int sendcount,
Datatype sendtype, Object recvbuf, int recvoffset,
int recvcount, Datatype recvtype, int root)`
 - Each process sends the contents of its send buffer to the root process.



Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	int, long, float, double
MPI_MIN	Minimum	int, long, float, double
MPI_SUM	Sum	int, long, float, double
MPI_PROD	Product	int, long, float, double
MPI_LAND	Logical AND	int, long
MPI_BAND	Bit-wise AND	byte, int, long
MPI_LOR	Logical OR	int, long
MPI_BOR	Bit-wise OR	byte, int, long
MPI_LXOR	Logical XOR	int, long
MPI_BXOR	Bit-wise XOR	byte, int, long
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs



MPI Reduce

```
void MPI.COMM_WORLD.Reduce(  
    Object sendbuf      /* in */,  
    int sendoffset      /* in */,  
    Object recvbuf      /* out */,  
    int recvoffset      /* in */,  
    int count           /* in */,  
    MPI.Datatype datatype /* in */,  
    MPI.Op operator     /* in */,  
    int root            /* in */)
```



```
MPI.COMM_WORLD.Reduce(msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
```

