COMP 322: Fundamentals of Parallel Programming

Lecture 34: Introduction to the Message Passing Interface (MPI)

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/

Worksheet #33 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 12 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

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).
  - The reason declaring buf as an Object rather than an array was that one would then need to overload with about 9 versions of most methods for arrays, e.g.

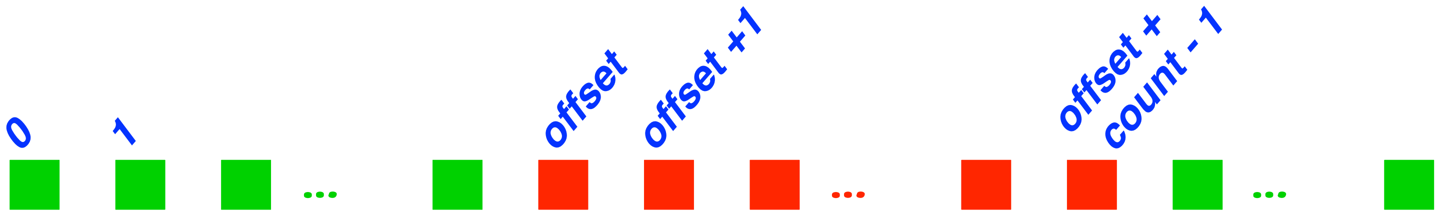
```
void Send(int [] buf, …)
void Send(long [] buf, …)
…
```

  and about 81 versions of operations that involve two buffers, possibly of different type. Declaring Object buf allows any kind of array in one signature.

- offset is the element in the buf array where message starts. count is the number of items to send. type describes the type of these items.

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.

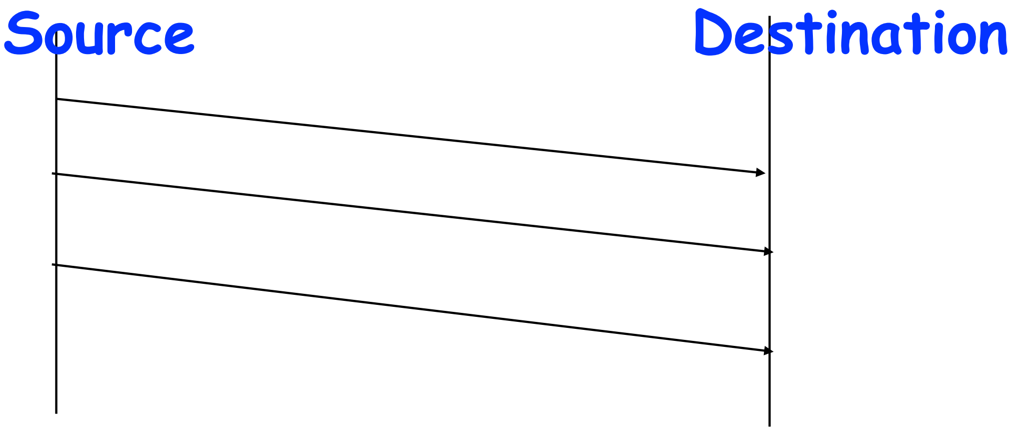
  - In the case of a receive buffer, the red boxes represent elements where the incoming data may be written (other elements will be unaffected). In this case count defines the maximum message size that can be accepted. Shorter incoming messages are also acceptable.

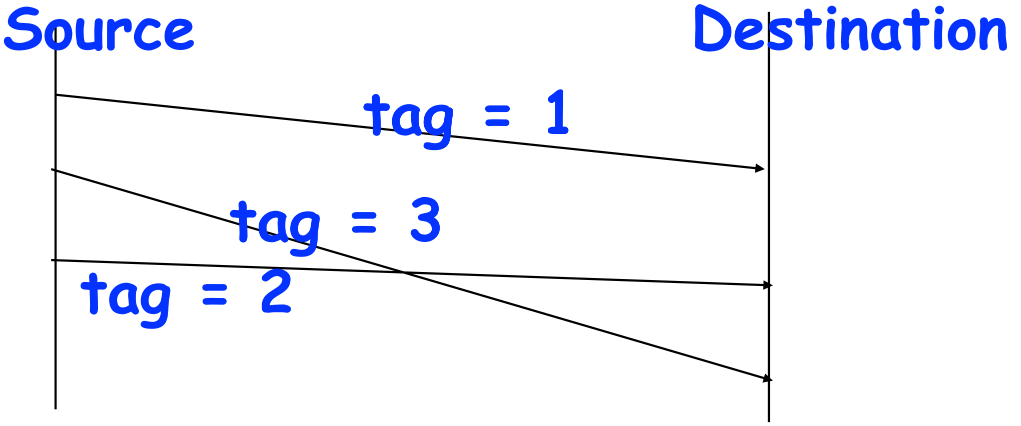
Basic Datatypes

- 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

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)

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)

```
int a[], b[];
. . .
int npes = MPI.COMM_WORLD.siz();
int myrank = MPI.COMM_WORLD.rank()
MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, (myrank+1)%npes, 1);
MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, (myrank-1+npes)%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 (as illustrated in Scenario #2), MPI provides a more specialized operation for this.
- 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) ;
```

  — This can be more efficient than doing separate sends and receives, and it can be used to avoid deadlock conditions in certain situations

    – Analogous to phaser “next” operation, where programmer does not have access to individual signal/wait operations

  — There is also a variant called Sendrecv_replace() which only specifies a single buffer: the original data is sent from this buffer, then overwritten with incoming data.

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-1+npes)%npes, 1);

...
```

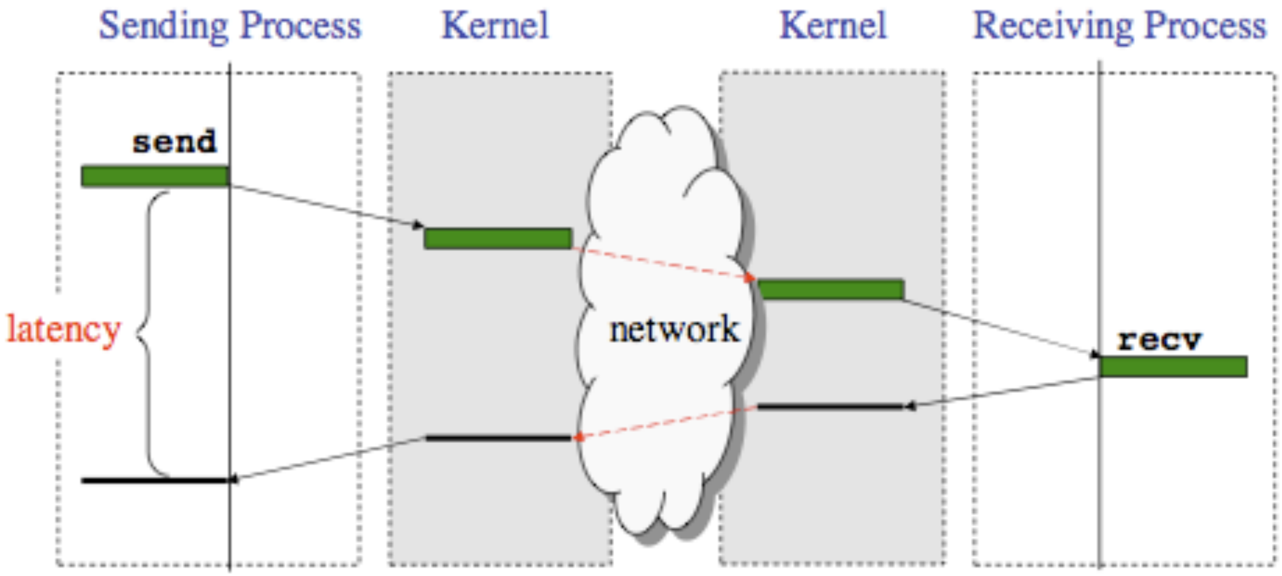
A combined Sendrecv() call avoids deadlock in this case

Outline of today’s lecture

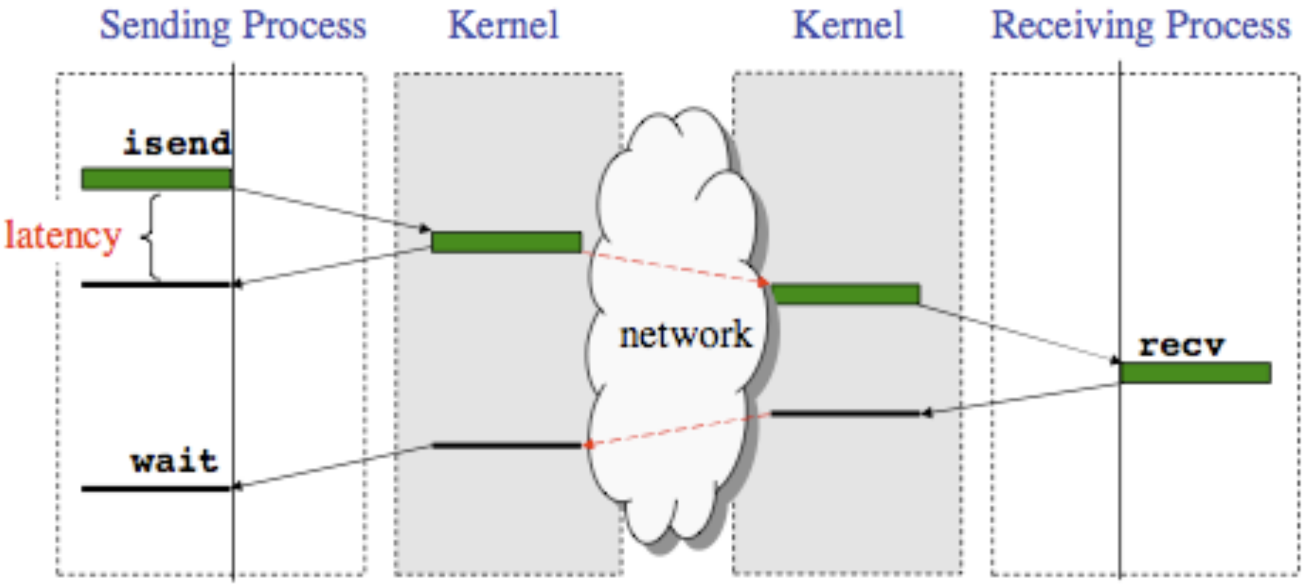
- Blocking communications (contd)
- <u>Non-blocking communications</u>
- Collective communications

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")
- The method signatures for Isend() and Irecv() are similar to those for Send() and Recv(), except that Isend() and Irecv() return objects of type Request:

```
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) ;
```

- Wait waits() for the operation to complete.

  Status Wait(Request request)

  - Wait is like a future get(), if you think of Isend() and Irecv() as future tasks
- Function Test() tests whether or not the non-blocking send or receive operation identified by its request has finished.

  Status Test(Request request)

  - Potential source of nondeterminism

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
…
```

- 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.

Waitall() vs. Waitany()

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

- **Waitall() blocks until all of the operations associated with the active requests in the array 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 in the array 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.

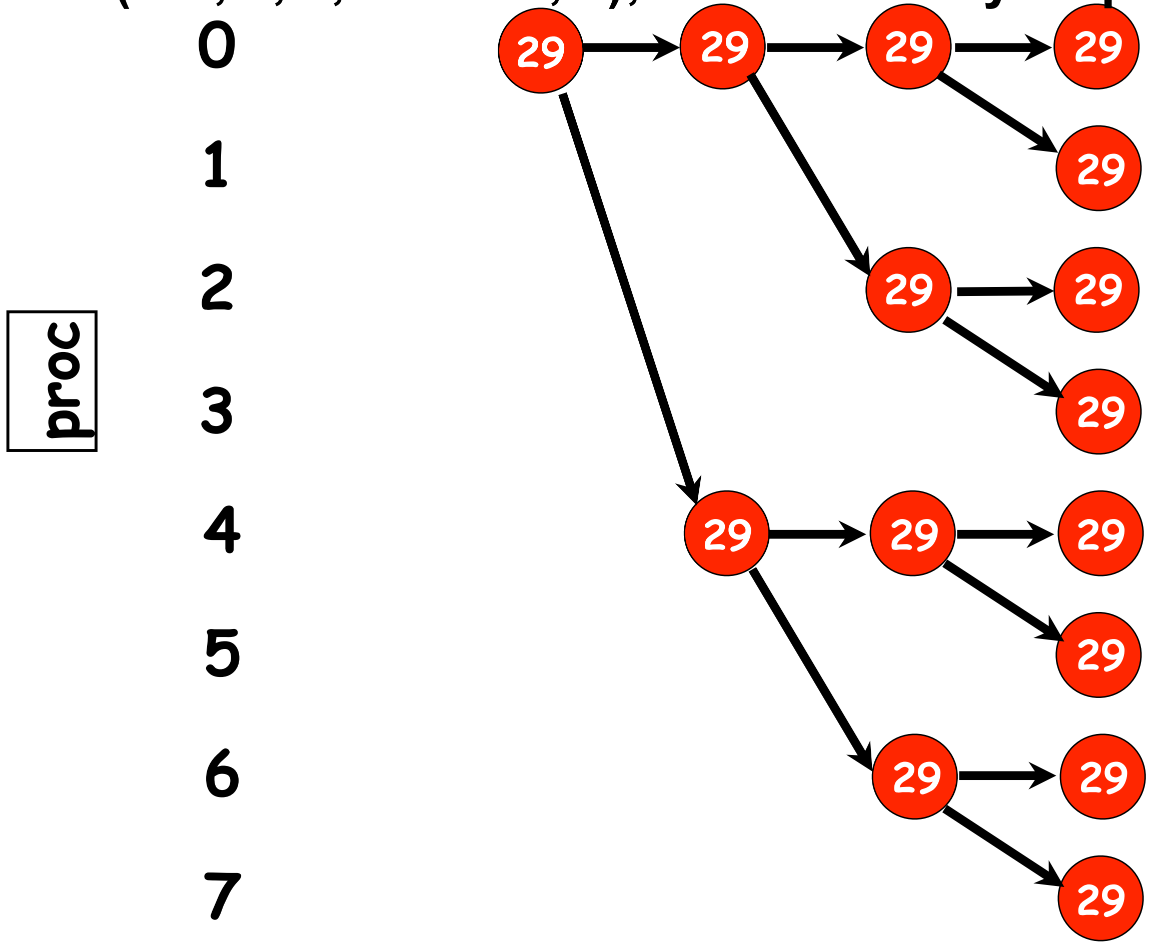
```
void Bcast(Object  buf, int  offset, int  count, Datatype type, int  root)
```

  – Broadcast a message from the process with rank root to all processes of the group

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 Barrier()
```

- Blocks the caller until all processes in the group have called it.

```
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)
```

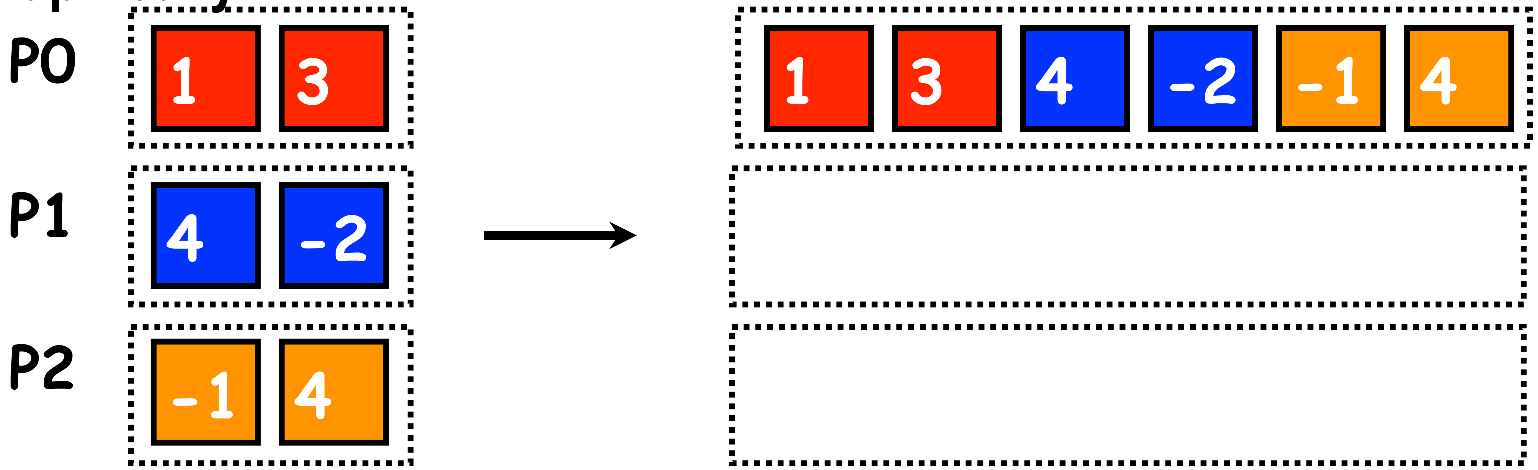
- Inverse of the operation Gather.

```
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:



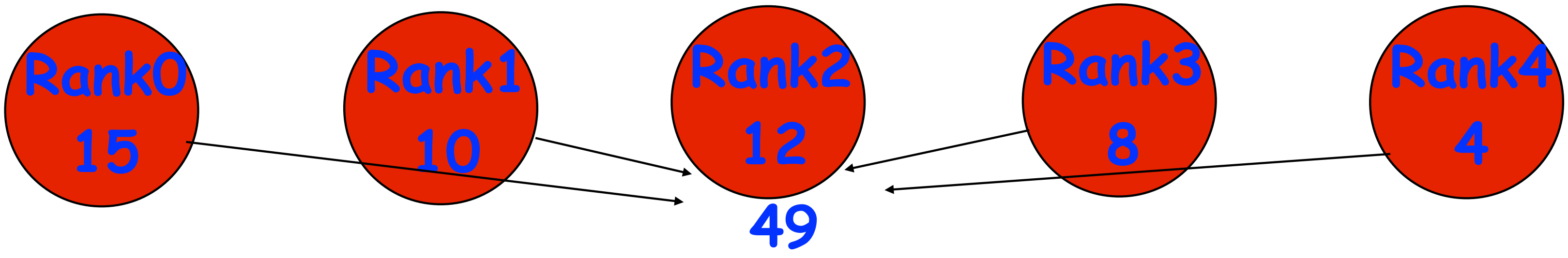
- Note: only process 0 (P0) needs to supply storage for the output

```
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.**

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 */ )
```



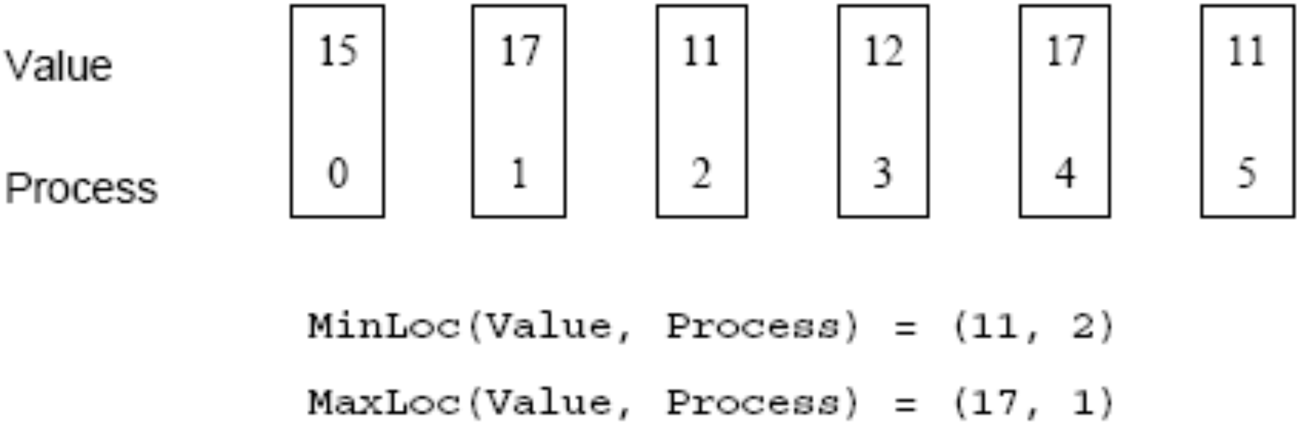
```
MPI.COMM_WORLD.Reduce(msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
```

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 (see next slide)
MPI_MINLOC	min-min value-location	Data-pairs

MPI_MAXLOC and MPI_MINLOC

- 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.



Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

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

More Collective Communication Operations

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

```
void AllReduce(Object sendbuf, int sendoffset,
Object recvbuf, int recvoffset, int count,
Datatype datatype, Op op)
```

- MPI also provides the MPI_Allgather function in which the data are gathered at all the processes.

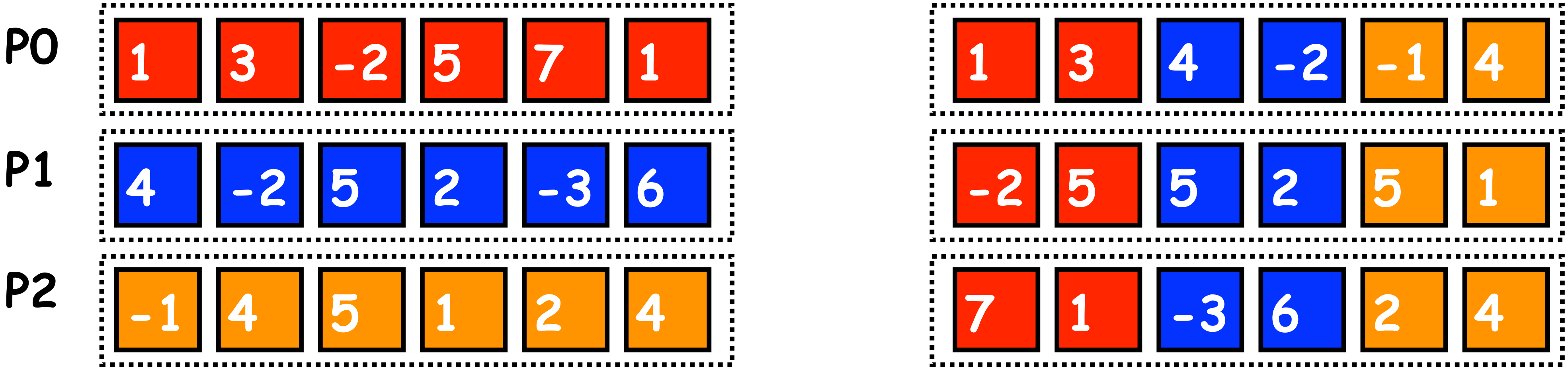
```
void AllGather(Object sendbuf, int sendoffset,
int sendcount, Datatype sendtype, Object recvbuf,
int recvoffset, int recvcount, Datatype recvtype)
```

- To compute prefix-sums in parallel, MPI provides:

```
void Scan(Object sendbuf, int sendoffset,
Object recvbuf, int recvoffset, int count,
Datatype datatype, Op op)
```

MPI_Alltoall

```
void Alltoall(Object sendbuf, int sendoffset, int sendcount,
              Object recvbuf, int recvoffset, int count,
              Datatype datatype)
```



P0 1 3 -2 5 7 1 → 1 3 4 -2 -1 4

P1 4 -2 5 2 -3 6 → -2 5 5 2 5 1

P2 -1 4 5 1 2 4 → 7 1 -3 6 2 4

- Each process submits an array to MPI_Alltoall.
- The array on each process is split into *nprocs* sub-arrays
- Sub-array *n* from process *m* is sent to process *n* placed in the *m*'th block in the result array.