# COMP 322: Fundamentals of Parallel Programming

# Lecture 34: Introduction to MPI (contd)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322



# Acknowledgments for Today's Lecture

- "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
  - Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html
- "Parallel Architectures", Calvin Lin
  - Lectures 5 & 6, CS380P, Spring 2009, UT Austin
  - http://www.cs.utexas.edu/users/lin/cs380p/schedule.html
- Slides accompanying Chapter 6 of "Introduction to Parallel Computing", 2nd Edition, Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, Addison-Wesley, 2003
  - http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap6_slides.pdf
- MPI slides from "High Performance Computing: Models, Methods and Means", Thomas Sterling, CSC 7600, Spring 2009, LSU
  - http://www.cct.lsu.edu/csc7600/coursemat/index.html
- mpiJava home page: http://www.hpjava.org/mpiJava.html
- MPI lectures given at Rice HPC Summer Institute 2009, Tim Warburton, May 2009



# Example of Blocking Send() and Recv() calls in MPI (Recap)

```
1.import mpi.*;

3.class myProg {
4.  public static void main( String[] args ) {
5.    int tag0 = 0;
6.    MPI.Init( args );                    // Start MPI computation
7.    if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
8.      int loop[] = new int[1]; loop[0] = 3;
9.      MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
10.       MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag0 );
11.     } else {                                 // rank 1 = receiver
12.       int loop[] = new int[1]; char msg[] = new char[12];
13.       MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
14.       MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag0 );
15.       for ( int i = 0; i < loop[0]; i++ ) System.out.println( msg );
16.     }
17.     MPI.Finalize( );                     // Finish MPI computation
18.   }
19.}
```

Send() and Recv() calls are blocking operations by default



# Example of using Sendrecv() for Deadlock Avoidance (Recap)

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

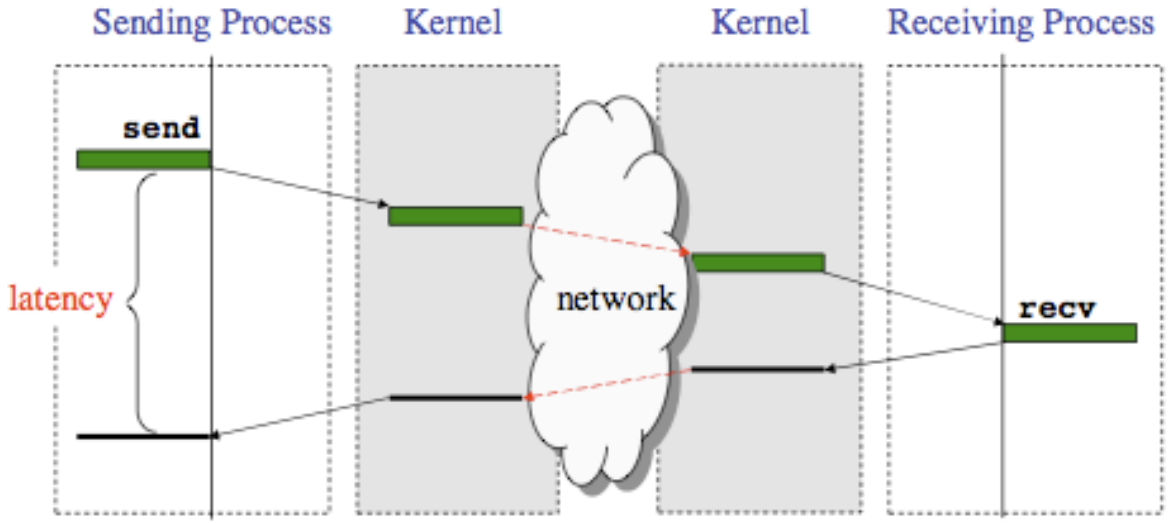
- Non-blocking communications

- Collective communciations

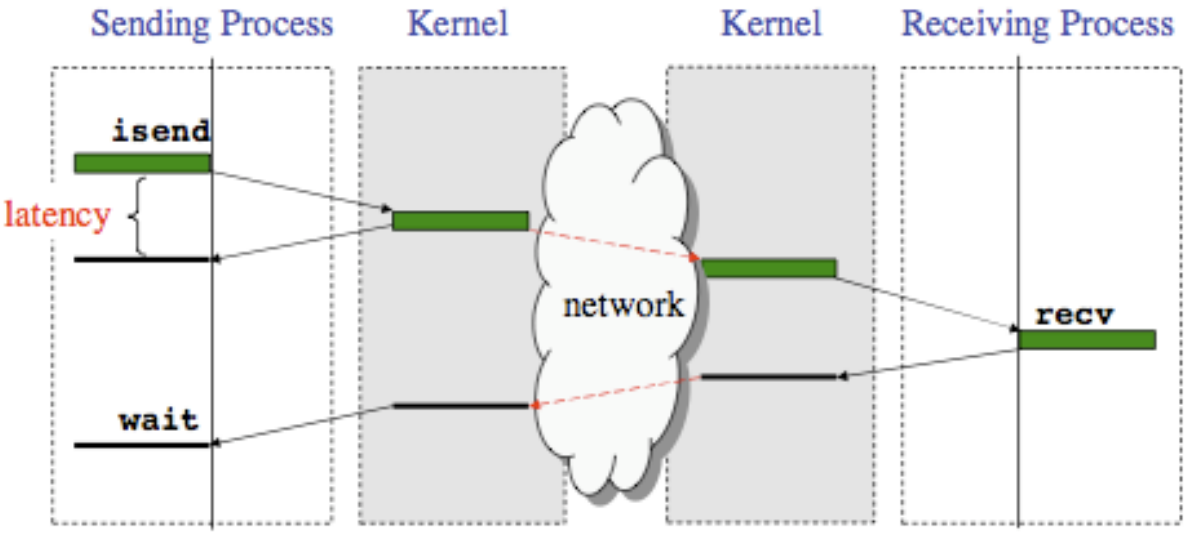


# Latency in Blocking vs. Nonblocking Communication

Blocking communication




Nonblocking communication (like an async or future task)






# Non-blocking Example

Example pseudo-code on process 0:

```
if(procid==0){

  Isend outgoing  to 1
  Irecv  incoming from 1

  .. compute ..

  Wait until Irecv  has received incoming

  .. compute ..

  Wait until Isend does not need outgoing

}
```

Example pseudo-code on process 1:

```
if(procid==1){

  Isend outgoing  to 1
  Irecv  incoming from 1

  .. compute ..

  Wait until Irecv  has received incoming

  .. compute ..

  Wait until Isend does not need outgoing

}
```

**Using the "*non-blocked*" send and receives allows us to overlap the latency and buffering overheads with useful computation.**



# 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) ;
- Function Test() tests whether or not the non-blocking send or receive operation identified by its request has finished.

  Status Test(Request request)
- Wait waits() for the operation to complete.

  Status Wait(Request request)



# 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 operation
Request request =
        Irecv(intBuf, 0, n, MPI.INT, MPI.ANY_SOURCE,  0) ;
// Do some work while the receive is in progress
...
// Finished that work, now make sure the message has arrived
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.



# Non-blocking Code Snippets (C version)

Post Irecv first

Set up outgoing data

Post Isend

Do local work

Wait for incoming data

Do local work

Make sure data left

```
/* process 0 does its thing */
if(procid==0){ dest = 1; source = 1; }
if(procid==1){ dest = 0; source = 0; }

/* this process requests unblocked receive from other process */
MPI_Irecv(indata, N, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &inrequest);

/* fill up outgoing data */
if(procid==0) for(n=0;n<N;++n) outdata[n] = 1./(   n+1.);
if(procid==1) for(n=0;n<N;++n) outdata[n] = 1./(.1*n+2.);

/* process 0 requests send to process 1 */
MPI_Isend(outdata, N, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD, &outrequest);

/* compute (each process does its own thing) */
if(procid==0) for(d=1,n=0;n<Nits;++n) d += 1./(n+d);
if(procid==1) for(d=1,n=0;n<Nits;++n) d += 1./(n+2*d);

/* now MPI_Wait to make sure incoming data arrived */
MPI_Wait(&inrequest, &status);

/* now can use inbound data */
for(n=0;n<N;++n) d += indata[n];

/* print out result */
printf("proc: %d result = %f\n", procid, d);

/* now MPI_Wait to make sure outgoing data has gone */
MPI_Wait(&outrequest, &status);
```



# 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. It returns an array of statuses for each of the requests.

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.

# Outline of today's lecture

- Non-blocking communications

- Collective communciations



# Collective Communications

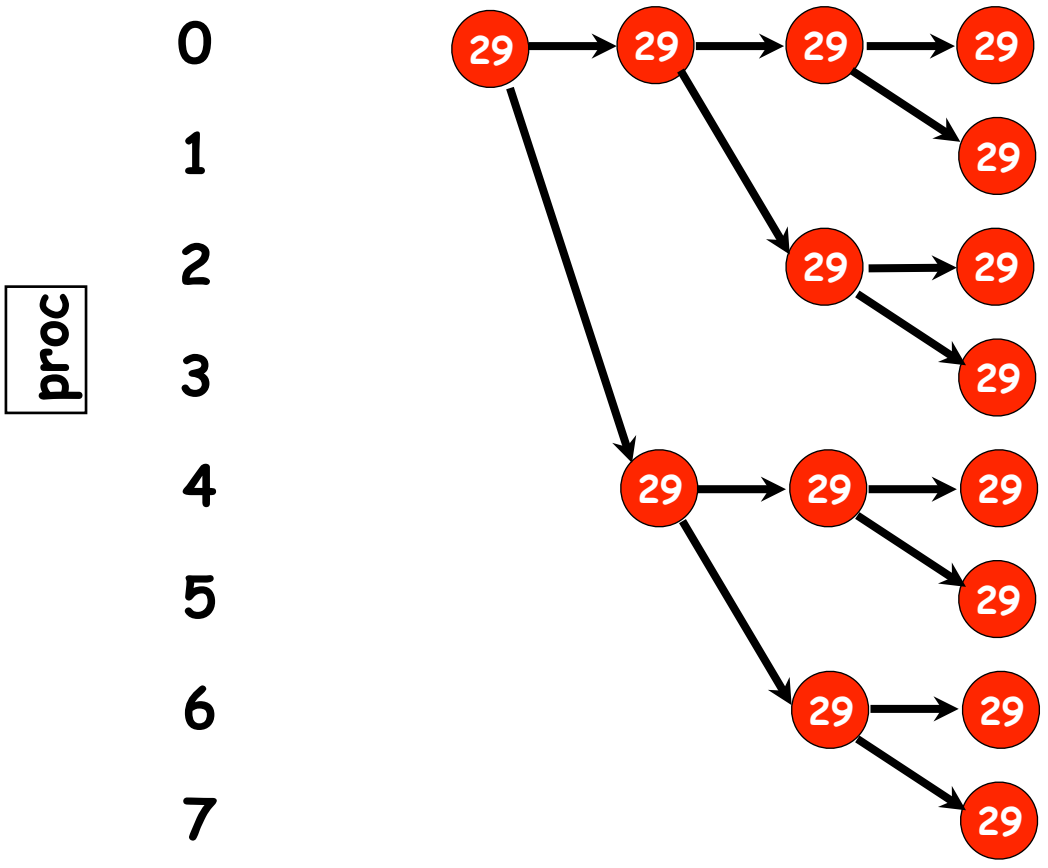
- A popular feature of MPI is its family of collective communication operations.
- Each of these operations is defined over a communicator.
  - All processes in a communicator must perform the same operation
  - Implicit barrier (next)
- The simplest example is the broadcast operation: all processes 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




Simple tree broadcast

A root process sends same message to all

29 represents an array of values



# More Examples of Collective Operations

- All the following are instance methods of Intracom:

  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.

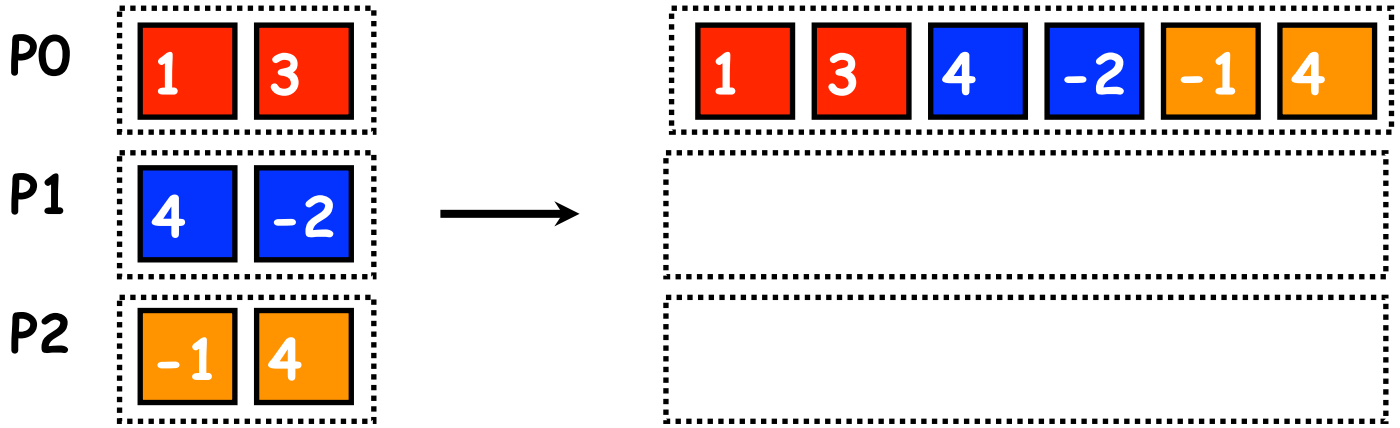
  void Scatter(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)
  - Inverse of the operation Gather.

  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

- On occasion it is necessary 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



# 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);
```



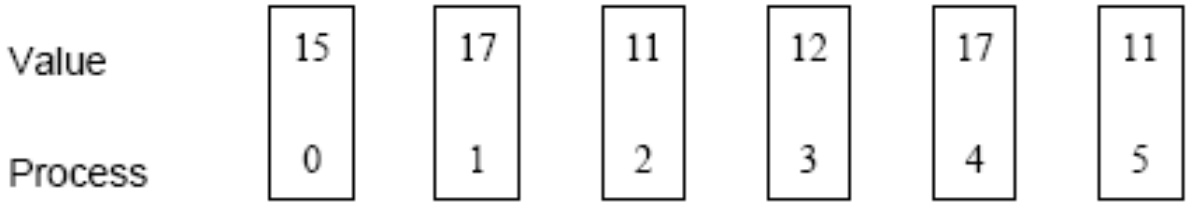
# Predefined Reduction Operations

| Operation | Meaning | Datatypes |
|---|---|---|
| MPI_MAX | Maximum | 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 |



# 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 $v_i$.




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

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



# Datatypes for MPI_MAXLOC and MPI_MINLOC

MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations.

| MPI Datatype | C Datatype |
|---|---|
| MPI_2INT | **pair of** int**s** |
| MPI_SHORT_INT | short **and** int |
| MPI_LONG_INT | long **and** int |
| MPI_LONG_DOUBLE_INT | long double **and** int |
| MPI_FLOAT_INT | float **and** int |
| MPI_DOUBLE_INT | double **and** int |



# More 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)
```

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

```
int MPI_Allgather(void *sendbuf, int sendcount,
                           MPI_Datatype senddatatype, void
*recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```

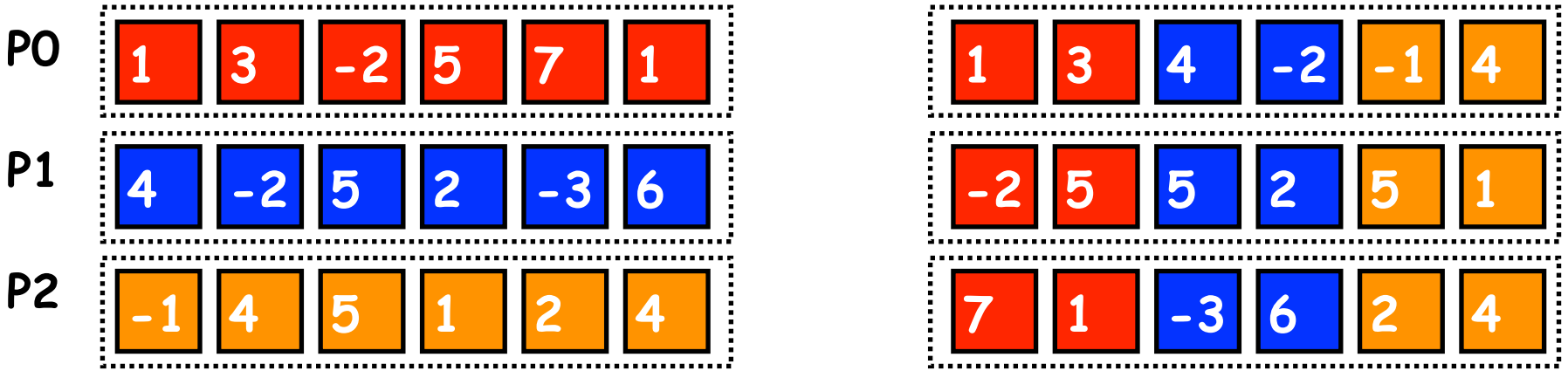
- To compute prefix-sums, MPI provides:

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



# MPI_Alltoall

```
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, void
*recvbuf,
                 int recvcount, MPI_Datatype
recvdatatype,    MPI_Comm comm)
```

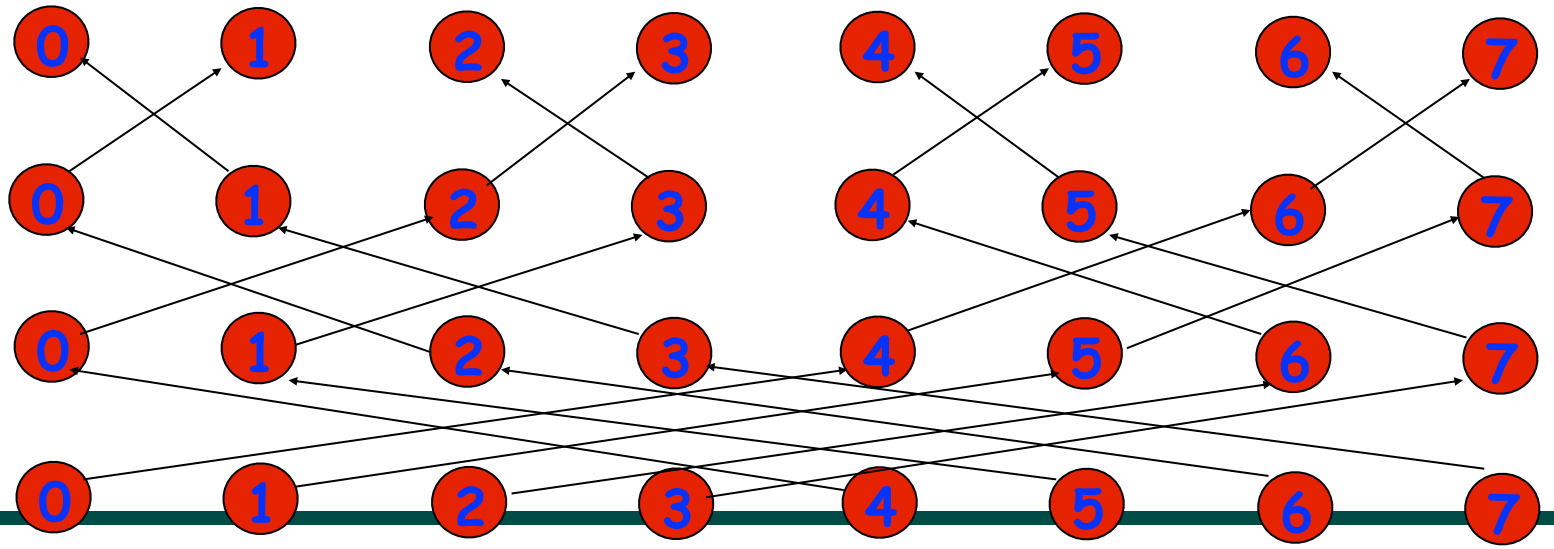



- Each process submits an array to MPI_Alltoall.
- The array on each process is split into *nprocs* sub-arrays
- Sub-array *n* from process *m* is sent to process *n* placed in the *m*'th block in the result array.



# MPI_Allreduce

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



# Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key.

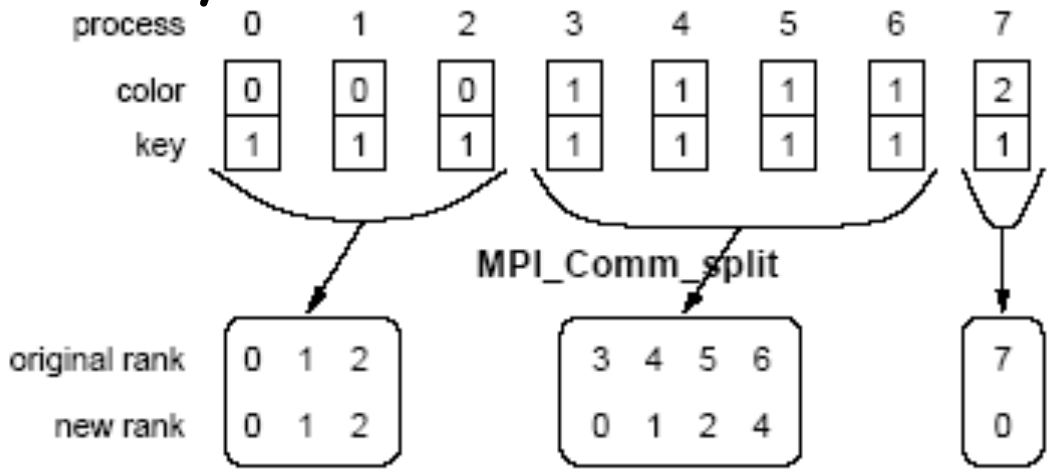


# Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

- This operation groups processors by color and sorts resulting groups on the key.






# Summary of MPI Collective Communications

- A large number of collective operations are available with MPI
- Too many to mention...
- This table summarizes some of the most useful collective operations

| Collective Function | Action |
|---|---|
| MPI_Gather | gather together arrays from all processes in comm |
| MPI_Reduce | reduce (elementwise) arrays from all processes in communicator |
| MPI_Scatter | a "root" process sends consecutive chunks of an array to all processes |
| MPI_Alltoall | Block transpose |
| MPI_Bcast | a "root" process sends the same array of data to all processes. |



# MPI Resources

- MPI: http://www.mcs.anl.gov/research/projects/mpi/
- MPICH2: http://www.mcs.anl.gov/research/projects/mpich2/
- Wiki: http://en.wikipedia.org/wiki/Message_Passing_Interface

- mpiJava home page: http://www.hpjava.org/mpiJava.html
  - Download: http://sourceforge.net/projects/mpijava/
- Web tutorials:
  - https://computing.llnl.gov/tutorials/mpi/
  - http://www.ecmwf.int/services/computing/training/material/hpcf/Intro_MPI_Programming.pdf (F77)
- Books:
  - http://www.cs.usfca.edu/mpi/
  - http://www.netlib.org/utk/papers/mpi-book/mpi-book.html



# Reminders

- Graded midterms can be picked up from Amanda Nokleby in Duncan Hall 3137
- Homework 6 now available
  - Deadline: April 20th
  - Automatic penalty-free extension to April 27th
- Take-home final exam (2 hours, like midterm)
  - Available for pick-up starting April 20th
  - Must be returned by April 27th