SE 292: High Performance Computing [3:0][Aug:2014]

Message Passing Interface
MPI

Yogesh Simmhan

Adapted from:
- *“MPI-Message Passing Interface”, Sathish Vadhiyar, SE292 (Aug:2013),*
- *INF3380: Parallel programming for scientific problems, Lecture 6, Univ of Oslo,*
- *12.950: Parallel Programming for Multicore Machines, Evangelinos, MIT,*
- *http://www.mpi-forum.org/docs/docs.html*

Midterm 3 Topics

Thu 13 Nov 8-930AM

Lectures on the following topics, and:

Concurrent Programming

- Bryant, 2011: Ch.12.3—12.5
- Silberschatz, 7th Ed.: Ch.4 & Ch.6

Parallelization

- Grama, 2003: Ch. 3.1, 3.5; 5.1—5.6

Parallel Architectures

100 points (10% weightage)

Assignment 2 Posted

- Due in 1 Week
- ***7AM Tue Nov 18 by email***

Substitute Class

- **Fri 14 Nov, 830AM**

Message Passing Principles

- Used for distributed memory programming
- Explicit communication
- Implicit or explicit synchronization
- Programming complexity is high
- But widely popular
- More control with the programmer

MPI Introduction

- MPI is a library standard for programming distributed memory using explicit message passing in MIMD machines.
- A standard API for message passing communication and process information lookup, registration, grouping and creation of new message datatypes.
- Collaborative computing by a group of individual processes
- Each process has its own local memory

MPI Introduction

- Need for a standard

  >> portability

  >> for hardware vendors

  >> for widespread use of concurrent computers
- MPI implementation(s) available on almost every major parallel platform (also on shared-memory machines)
- Portability, good performance & functionality

MPI Introduction

- 1992-94 the Message Passing Forum defines a standard for message passing (targeting MPPs)
- Evolving standards process:
- 1994: MPI 1.0: Basic comms, Fortran 77 & C bindings
- 1995: MPI 1.1: errata and clarifications
- 1997: MPI 2.0: single-sided comms, I/O, process creation, Fortran 90 and C++ bindings, further clarifications, many other things. Includes MPI-1.2.
- 2008: MPI 1.3, 2.1: combine 1.3 and 2.0, corrections & clarifications
- 2009: MPI 2.2: corrections & clarifications
- 2013: MPI 3.0 released

MPI contains…

- Point-Point (1.1)
- Collectives (1.1)
- Communication contexts (1.1)
- Process topologies (1.1)
- Profiling interface (1.1)
- I/O (2)
- Dynamic process groups (2)
- One-sided communications (2)
- Extended collectives (2)
- About 125 functions; Mostly 6 are used

MPI Implementations

- MPICH (Argonne National Lab)
- LAM-MPI (Ohio, Notre Dame, Bloomington)
- LAM-MPI
- Cray, IBM, SGI
- MPI-FM (Illinois)
- MPI / Pro

(MPI Software Tech.)

- Sca MPI (Scali AS)
- Plenty others...

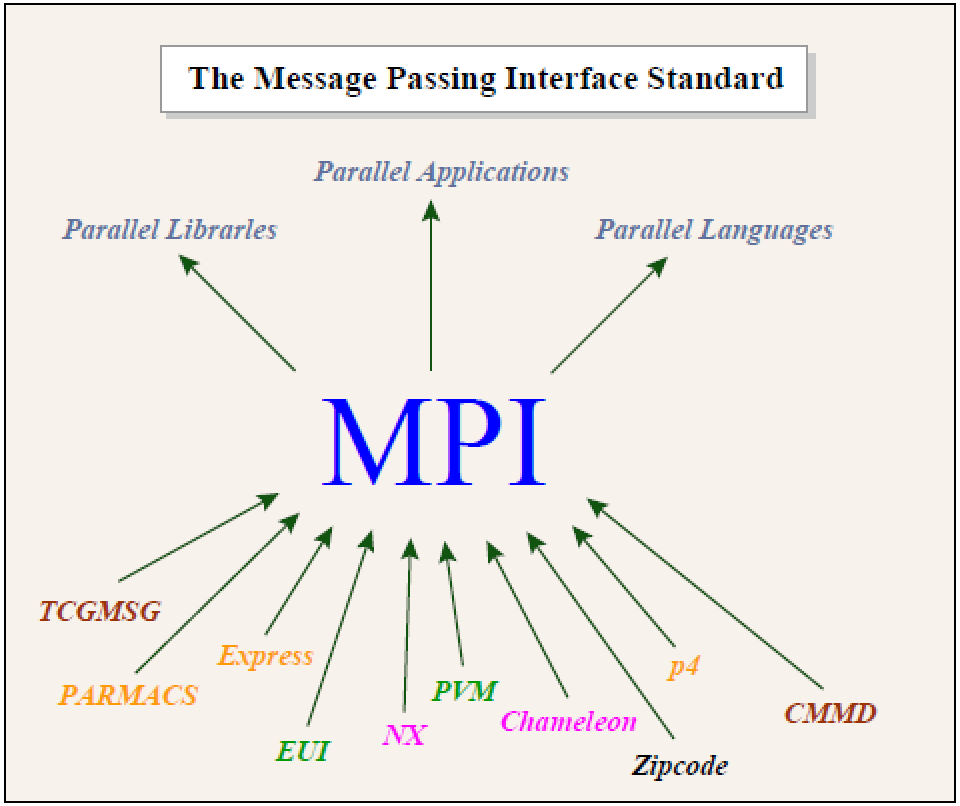
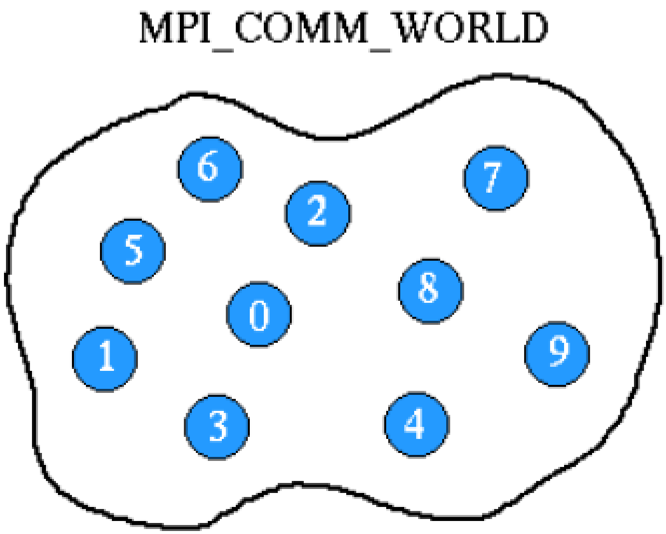


Figure by MIT OpenCourseWare.

MPI Communicator

- *communication universe* for a group of processes
- **MPI COMM WORLD** – name of the default MPI communicator, i.e., the collection of all processes
- Each process in a communicator is identified by its rank
- Almost every MPI command needs to provide a communicator as input argument



MPI process rank

- Each process has a unique rank, i.e. an integer identifier, within a communicator
- The rank value is between 0 and #procs-1
- The rank value distinguishes one process from another

```
#include <mpi.h>
...
int size, my_rank;
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
if (my_rank==0) {
  ...
}
```

6 Key MPI Commands

- `MPI_Init` - initiate an MPI computation
- `MPI_Finalize` - terminate the MPI computation and clean up
- `MPI_Comm_size` - how many processes participate in a given MPI communicator?
- `MPI_Comm_rank` - which one am I? (A number between 0 and size-1.)
- `MPI_Send` - send a message to a particular process within an MPI communicator
- `MPI_Recv` - receive a message from a particular process within an MPI communicator

Example

```
#include <stdio.h>
#include <mpi.h>
int main (int nargs, char** args) {
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d
      procs.\n", my_rank, size);
  MPI_Finalize ();
  return 0;
}
```

Compile: `mpicc hello.c`

Run: `mpirun -np 4 a.out`

Output:

```
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
Hello world, I've rank 0 out of 4 procs.
```

Communication Primitives

- Communication scope
- Point-point communications
- Collective communications

Point-Point Communications

$$\texttt{MPI\_SEND(}\underbrace{\texttt{buf, count, datatype,}}_{\text{Message}}\ \texttt{dest, tag, comm)}$$

dest: Rank of the destination

tag: Message identifier

comm: Communication context

This blocking send function returns when the data has been delivered to the system and the buffer can be reused. The message may not have been received by the destination process.

An MPI message is an array of data elements "inside an envelope"
Data: start address of the message buffer, counter of elements in the buffer, data type
Envelope: source/destination process, message tag, communicator

Point-Point Communications

```
MPI_RECV(buf, count, datatype, source, tag,
   comm, status)
```

- This blocking receive function waits until a matching message is received from the system so that the buffer contains the incoming message.
- Match of data type, source process (or MPI ANY SOURCE), message tag (or MPI ANY TAG).
- Receiving fewer datatype elements than count is ok, but receiving more is an error

Point-Point Communications

```
MPI_GET_COUNT(status, datatype, count)
status.MPI_SOURCE
status.MPI_TAG
```

- The source or tag of a received message may not be known if wildcard values were used in the receive function. In C, MPI_Status is a structure that contains further information.

A Simple Example

```
comm = MPI_COMM_WORLD;
rank = MPI_Comm_rank(comm, &rank);
for(i=0; i<n; i++) a[i] = 0;
if(rank == 0){
  MPI_Send(a+n/2, n/2, MPI_INT, 1, tag, comm);
}
else{
  MPI_Recv(b, n/2, MPI_INT, 0, tag, comm, &status);
}
/* process array a */

/* do reverse communication */
```

Communication Scope

- Explicit communications
- Each communication associated with communication scope
- Process defined by
  - *Group*
  - Rank within a group

Message label by
  - Message *context*
  - Message tag

A communication handle called *Communicator* defines the scope

Communicator

- Communicator represents the communication domain
- Helps in the creation of process groups
- Can be intra or inter (more later).
- Default communicator – MPI_COMM_WORLD includes all processes
- Wild cards:
  - The receiver source and tag fields can be wild carded – MPI_ANY_SOURCE, MPI_ANY_TAG

Buffering and Safety

The previous send and receive are **blocking**. Buffering mechanisms can come into play.

	Process 0	Process 1
Safe buffering: OK	MPI_Send MPI_Recv	MPI_Recv MPI_Send ....
Leads to deadlock	MPI_Recv MPI_Send	MPI_Recv MPI_Send ....
May or may not succeed. Unsafe	MPI_Send MPI_Recv	MPI_Send MPI_Recv ....

Type of P2P Communication

- Blocking comms: Block until completed (*send stuff on your own*)
- Non-blocking comms: Return without waiting for completion (*give them to someone else*)
- Forms of Blocking Sends:
  - Synchronous MPI_Ssend: message gets sent only when it is known that someone is already waiting at the other end (*think fax*)
  - Buffered MPI_Bsend: message gets sent and if someone is waiting for it so be it; otherwise it gets saved in a temporary buffer until someone retrieves it. (*think mail*)
  - Ready MPI_Rsend: Like synchronous, only there is no ack that there is a matching receive at the other end, just a programmer's assumption! **(Use it with extreme care)**

Blocking Send Performance

- Synchronous sends offer the highest data rate (AKA bandwidth) but the startup cost (latency) is very high, and they run the risk of deadlock.
- Buffered sends offer the lowest latency but:
  - suffer from buffer management complications
  - have bandwidth problems because of the extra copies and system calls
- Ready sends *should* offer the best of both worlds but are prone to cause trouble. Avoid!
- Standard sends usually carefully optimized by the implementors. For large message sizes they can always deadlock.

Message passing restrictions

- Order is preserved. For a given channel (sender, receiver, communicator) message order is enforced:
- If P sends to Q, messages *sa* and *sb* in that order, that is the order they will be received at B, even if *sa* is a medium message sent with MPI_Bsend and *sb* is a small message sent with MPI_Send. Messages do not overtake each other.
- If the corresponding receives *ra* and *rb* match both messages (same tag) again the receives are done in order of arrival.
- This is actually a performance drawback for MPI *but* helps avoid major programming errors.

Non-blocking communications

- A *post* of a send or recv operation followed by *complete* of the operation

  MPI_ISEND (buf, count, datatype, dest, tag, comm, request)

  MPI_IRECV (buf, count, datatype, dest, tag, comm, request)

  MPI_WAIT (request, status)

  MPI_TEST (request, flag, status)

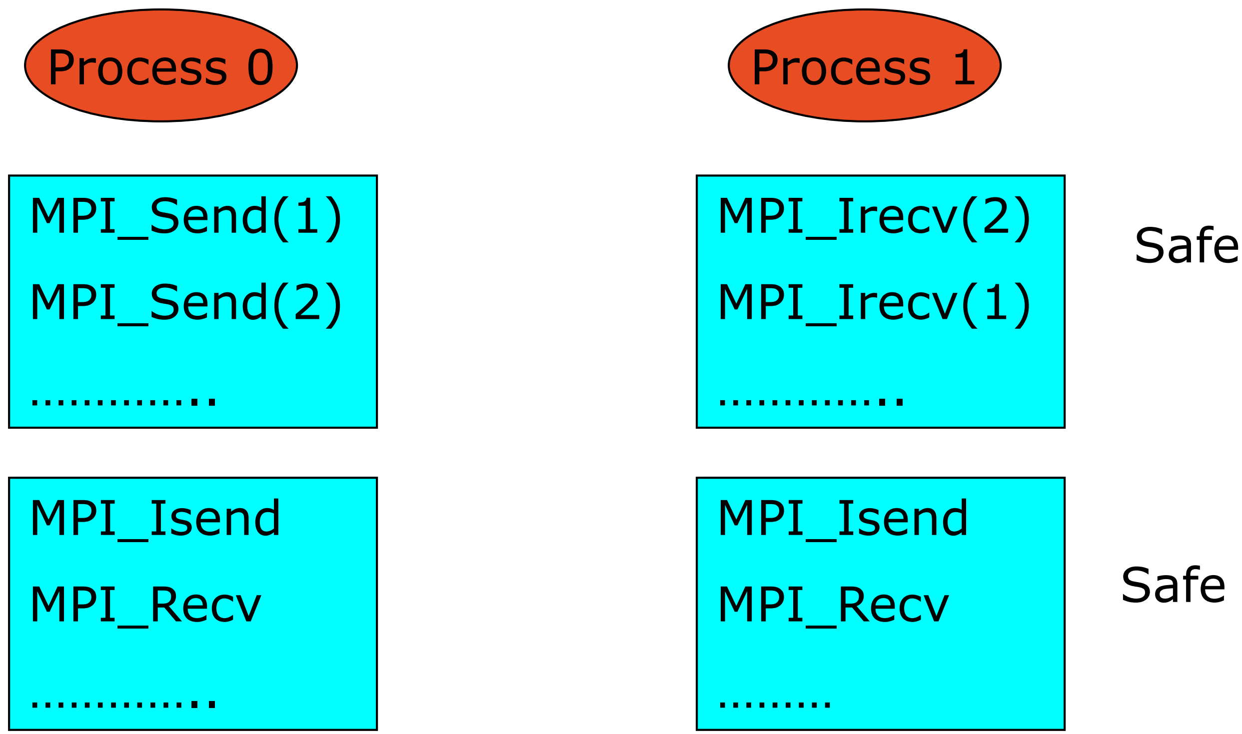
  MPI_REQUEST_FREE (request)

- A post-send *returns before* the message is copied out of the send buffer
- A post-recv *returns before* data is copied into the recv buffer

Non-blocking

- Call MPI_Isend | MPI_Irecv, store the request handle, do some work to keep busy and then call MPI_Wait with the handle to complete the send.
- MPI_Isend | MPI_Irecv produces the request handle, MPI_Wait consumes it.
- Efficiency depends on the implementation

Buffering and Safety



To avoid deadlock we need to interlace nonblocking sends with blocking receives, or nonblocking receives with blocking sends; nonblocking calls always precede the blocking ones.

Non-blocking communications

- One or none completed

MPI_WAITANY (count, request[], index, status)

MPI_TESTANY (count, request[], index, flag, status)

- All are completed

MPI_WAITALL (count, request[], status[])

MPI_TESTALL (count, request[], flag, status[])

- Return after some time, when "some" are completed

MPI_WAITSOME (incount, request[], outcount, index[], status[])

MPI_TESTSOME (incount, request[], outcount, index[], status[])

Communication Modes

Mode	**Start**	**Completion**
Standard (MPI_Send)	Before or after recv	Before recv (buffer) or after (no buffer)
Buffered (MPI_Bsend) (Uses MPI_Buffer_Attach)	Before or after recv	Before recv
Synchronous (MPI_Ssend)	Before or after recv	Particular point in recv
Ready (MPI_Rsend)	After recv	After recv

Collective Communications

Broadcast

MPI_Bcast (void *buf, int cnt, MPI_Datatype type, int root, MPI_Comm comm)

- root has to be the same on all procs, can be nonzero

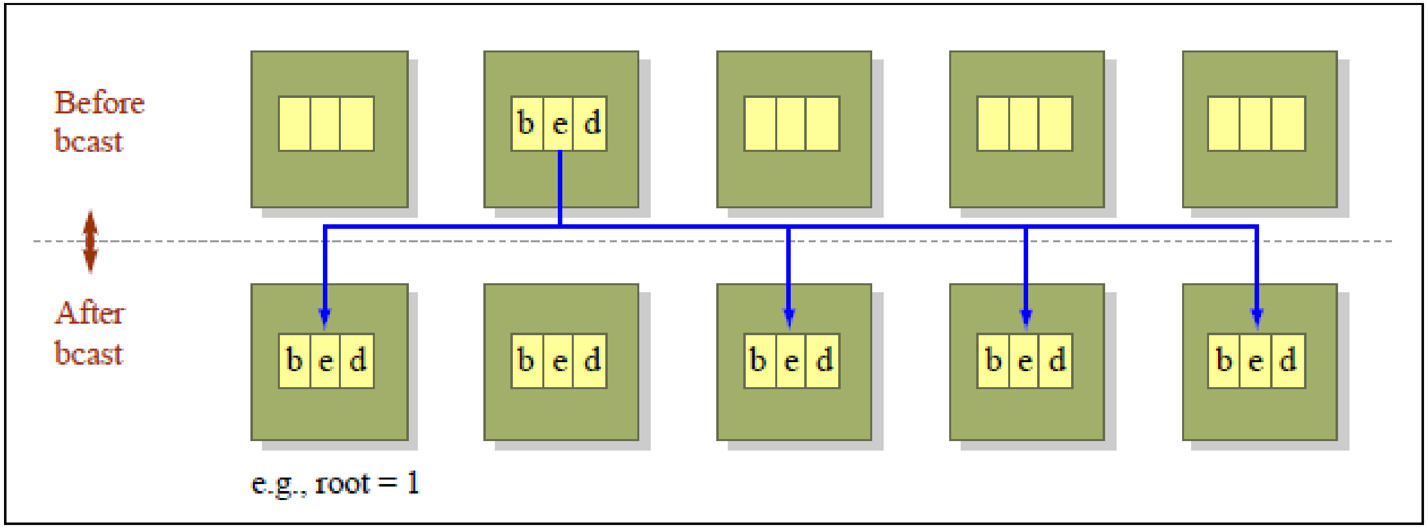


Figure by MIT OpenCourseWare.

Gather

```
MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Make sure recvbuf is large enough on root where it matters, elsewhere it is ignored

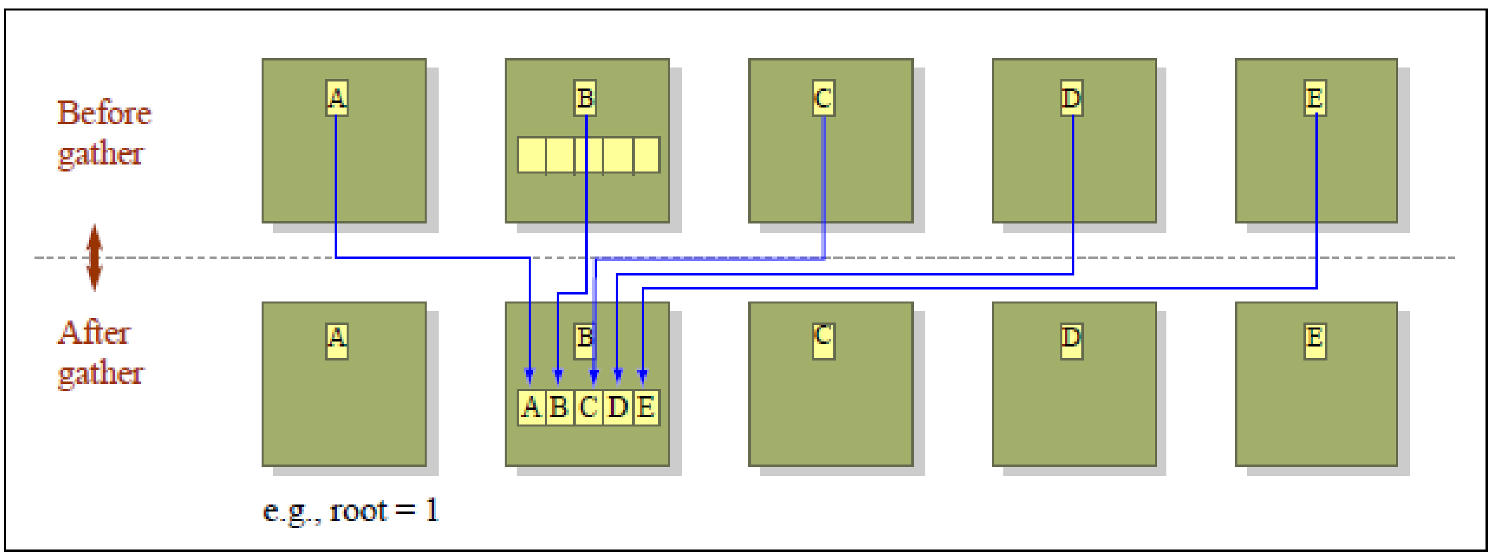


Figure by MIT OpenCourseWare.

All Gather

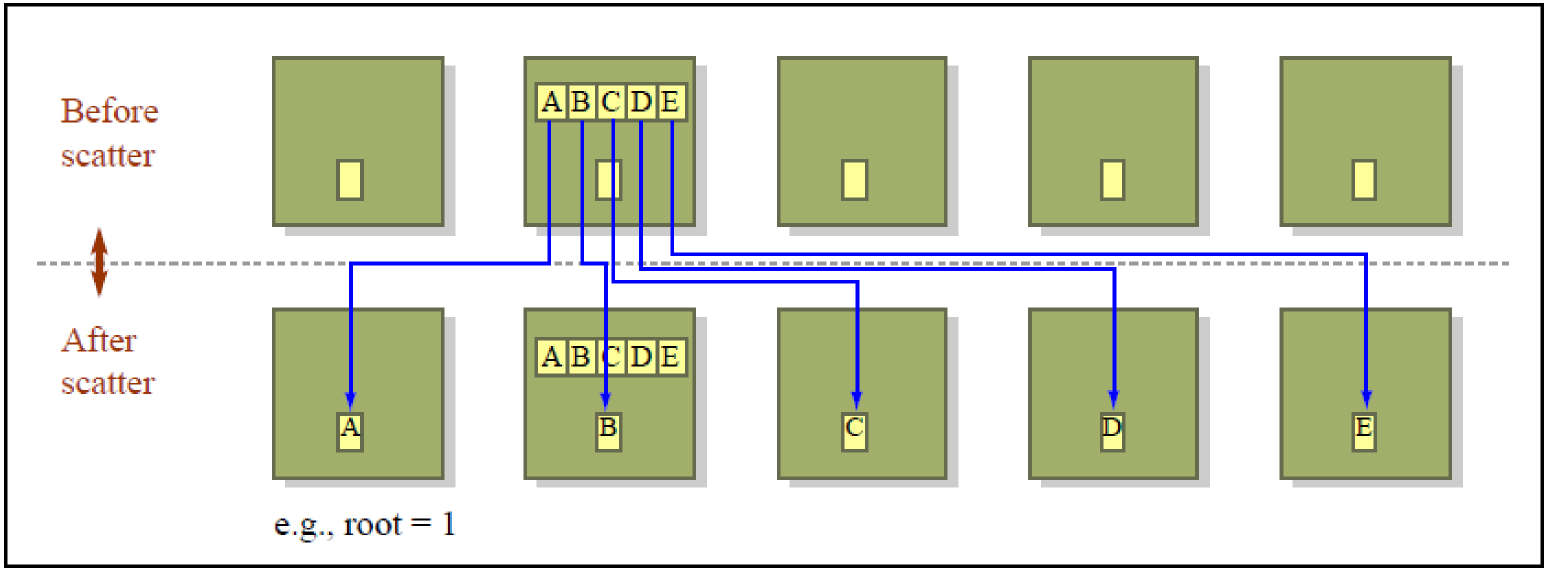
```
MPI_Allgather (void *sendbuf, int
sendcnt, MPI_Datatype sendtype, void
*recvbuf, int recvcnt, MPI_Datatype
recvtype, MPI_Comm comm)
```

- Can be thought of as an MPI_Gather followed by an MPI_Bcast, with an unspecified root process
- *Make sure recvbuf is large enough on all procs*

Scatter

```
MPI_Scatter (void *sendbuf, int
sendcnt, MPI_Datatype sendtype, void
*recvbuf, int recvcnt, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

- Make sure recvbuf is large enough on all procs, sendbuf matter only on root



Reduce

```
MPI_Reduce (void *sendbuf, void
*recvbuff, int cnt, MPI_Datatype type,
MPI_Op op, int root, MPI_Comm comm)
```

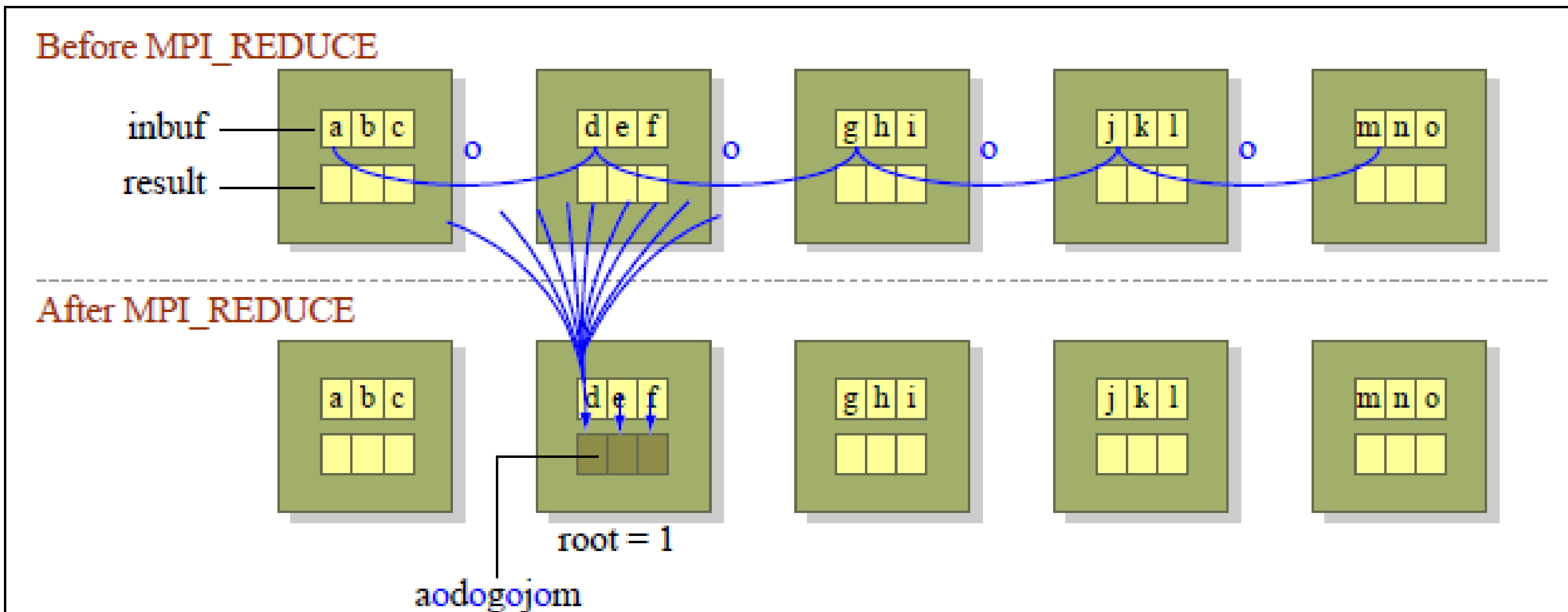
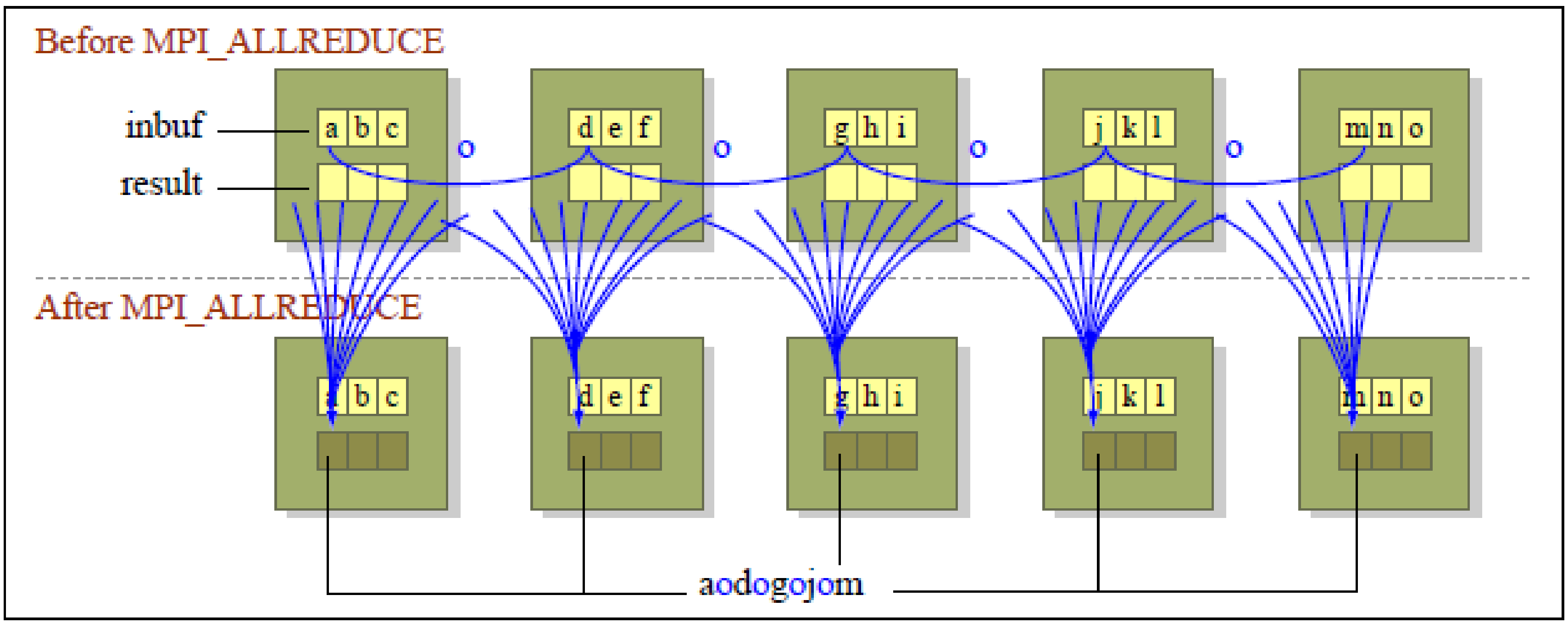


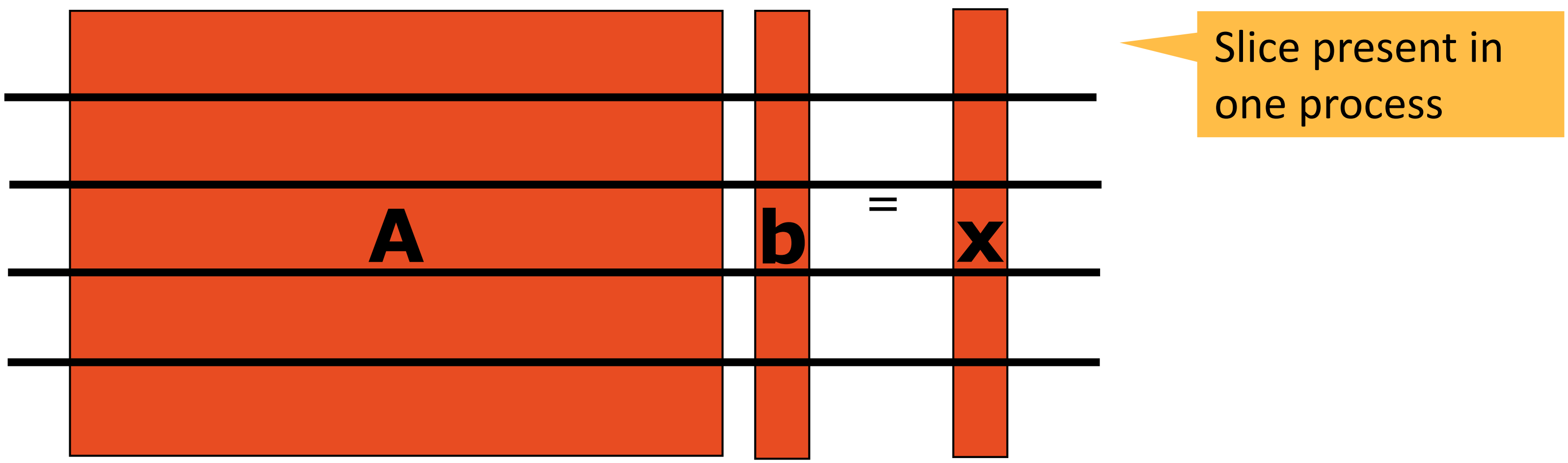
Figure by MIT OpenCourseWare.

All Reduce

```
MPI_Allreduce (void *sendbuf, void
*recvbuff, int cnt, MPI_Datatype type,
MPI_Op op, MPI_Comm comm)
```



Example: Matrix-vector Multiply



Communication:

All processes should gather all elements of b.

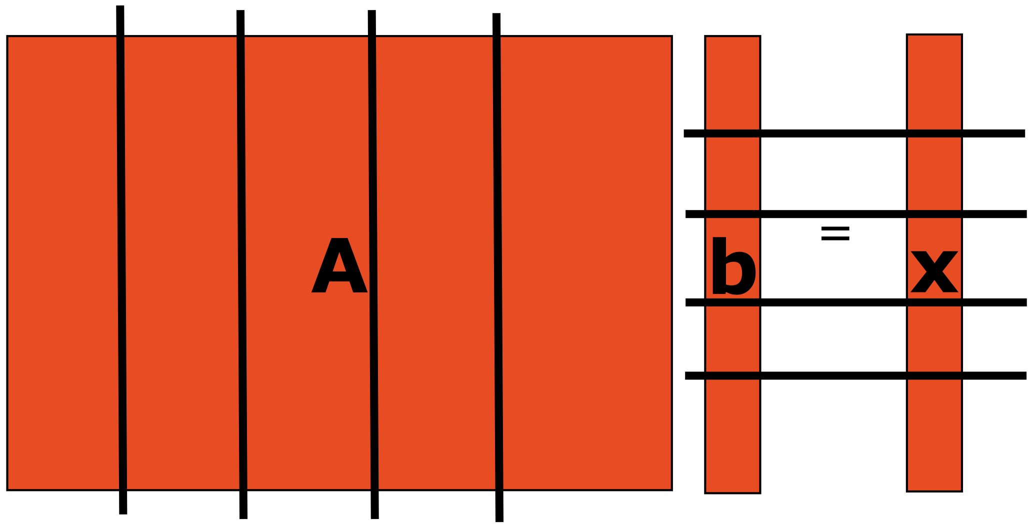
Example: Row-wise Matrix-Vector Multiply

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
nlocal = n/size ;


MPI_Allgather(local_b, nlocal, MPI_DOUBLE, b, nlocal,
  MPI_DOUBLE, comm);


for(i=0; i<nlocal; i++){
  x[i] = 0.0;
  for(j=0; j<n; j+=)
    x[i] += a[i*n+j]*b[j];
}
```

Example: Column-wise Matrix-vector Multiply



Dot-products corresponding to each element of x will be parallelized

Steps:

1. Each process computes its contribution to x

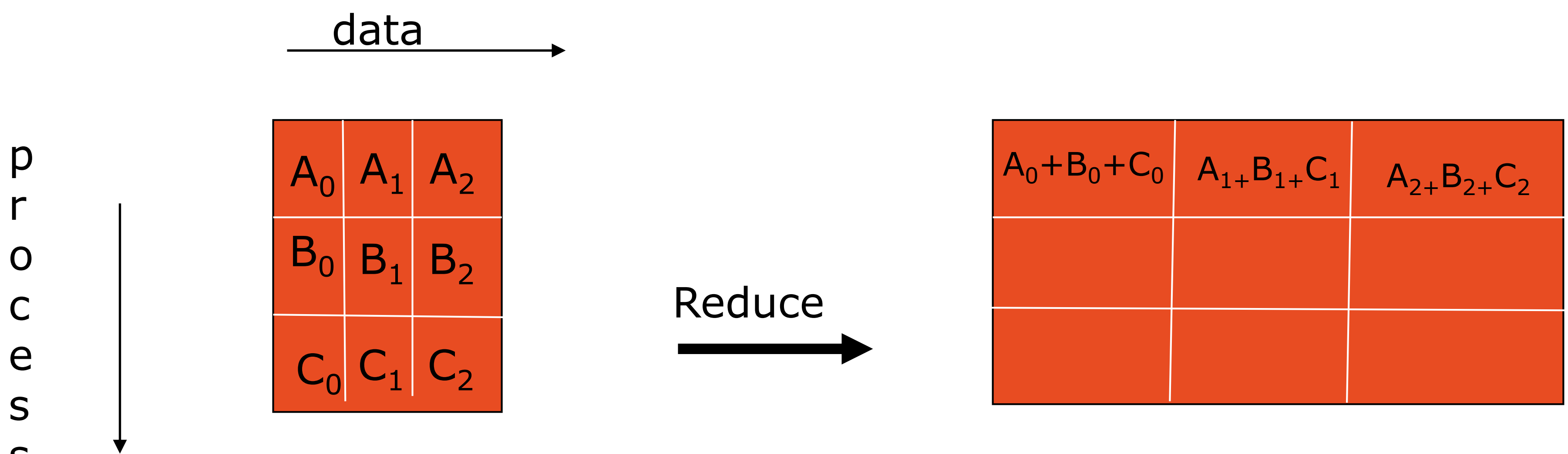
2. Contributions from all processes are added and stored in appropriate process.

Example: Column-wise Matrix-Vector Multiply

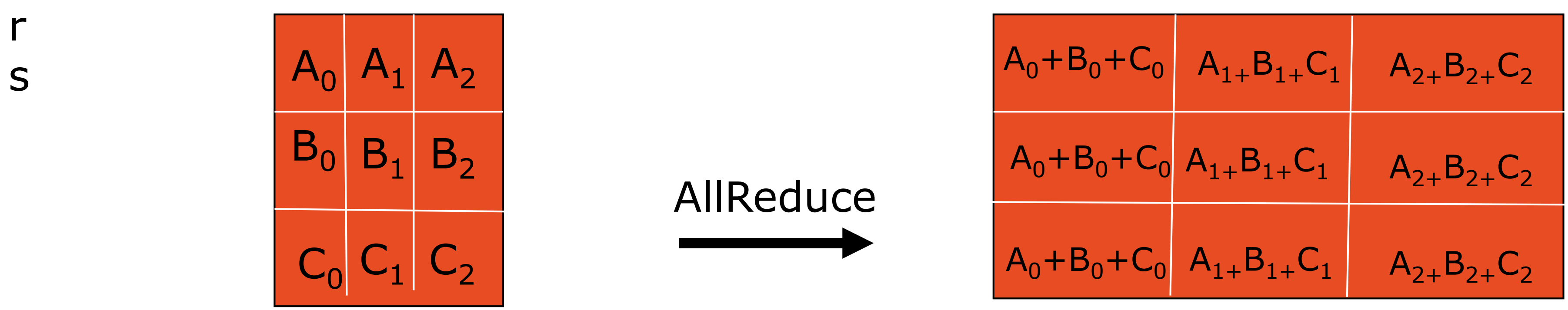
```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
nlocal = n/size;

/* Compute partial dot-products */
for(i=0; i<n; i++){
  px[i] = 0.0;
  for(j=0; j<nlocal; j+=)
    px[i] += a[i*nlocal+j]*b[j];
}
```

Collective Communications – Reduce, Allreduce

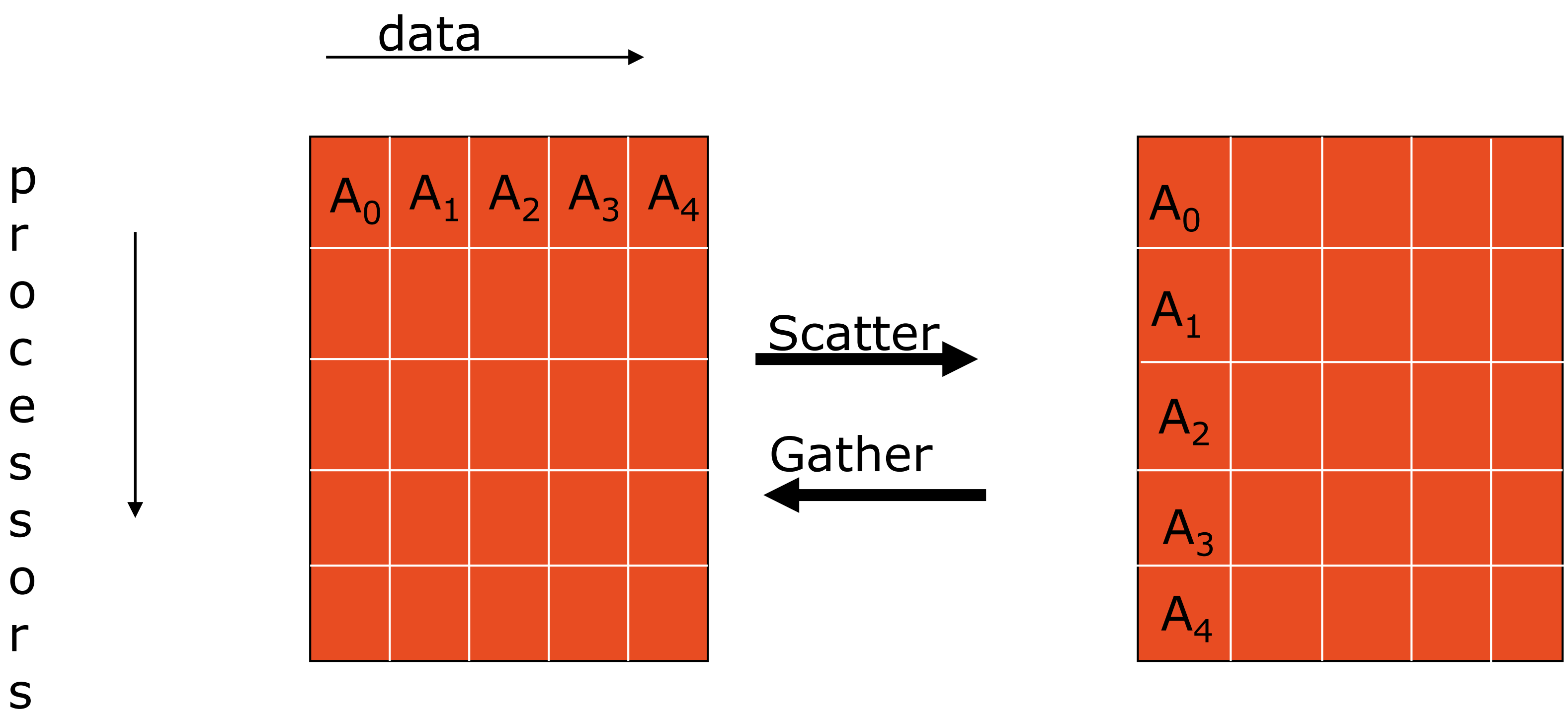


MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)



MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

Collective Communications – Scatter & Gather



MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
MPI_SCATTERV(sendbuf, array_of_sendcounts, array_of_displ, sendtype, recvbuf, recvcount, recvtype, root, comm)

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, array_of_recvcounts, array_of_displ, recvtype, root, comm)

Example: Column-wise Matrix-Vector Multiply

```
/* Summing the dot-products */
MPI_Reduce(px, fx, n, MPI_DOUBLE,
  MPI_SUM, 0, comm);


/* Now all values of x is stored in
  process 0. Need to scatter them */
MPI_Scatter(fx, nlocal, MPI_DOUBLE, x,
  nlocal, MPI_DOUBLE, 0, comm);
```

Or…

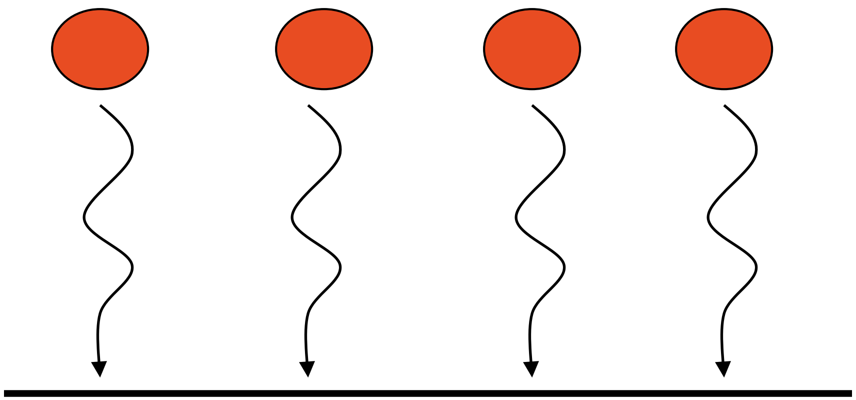
```
for(i=0; i<size; i++){
    MPI_Reduce(px+i*nlocal, x, nlocal,
 MPI_DOUBLE, MPI_SUM, i, comm);
}
```

Collective Communications

- Only blocking; standard mode; no tags
- Simple variant or “vector” variant
- Some collectives have “root”s
- Different types
  - One-to-all
  - All-to-one
  - All-to-all

Collective Communications - Barrier

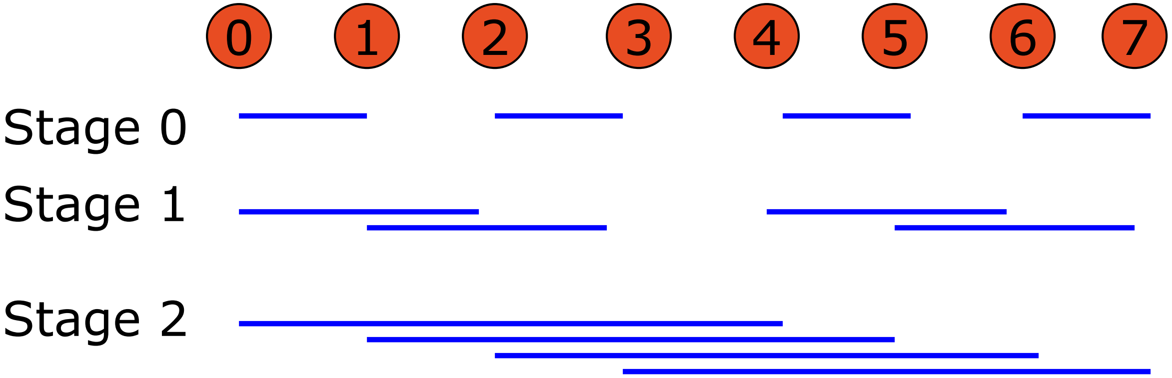
MPI_BARRIER(comm)



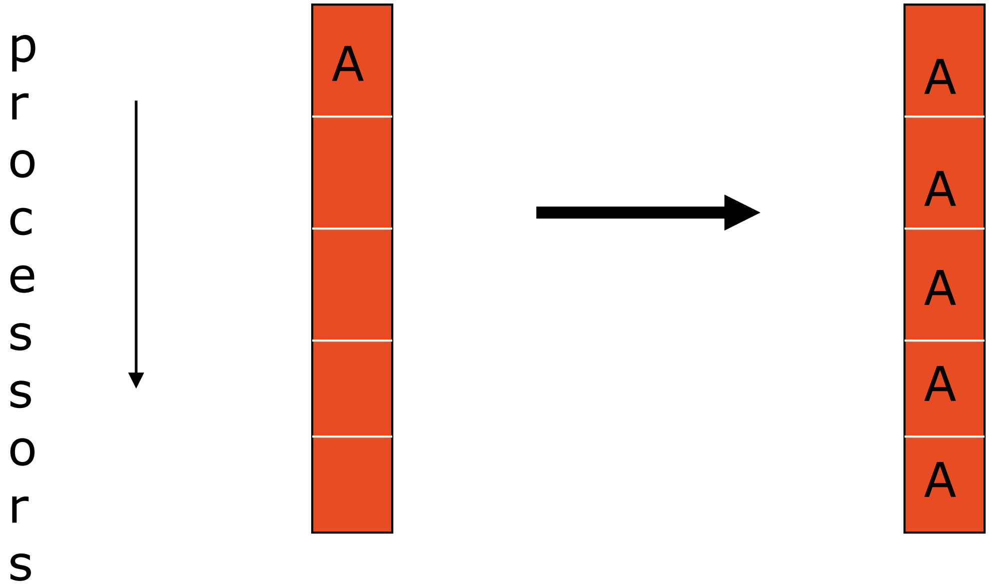
A return from barrier in one process tells the process that the other processes have ***entered*** the barrier.

Barrier Implementation

- **Butterfly barrier** by Eugene Brooks II
- In round k, i synchronizes with $i \oplus 2^k$ pairwise.
- Worstcase – 2logP pairwise synchronizations by a processor



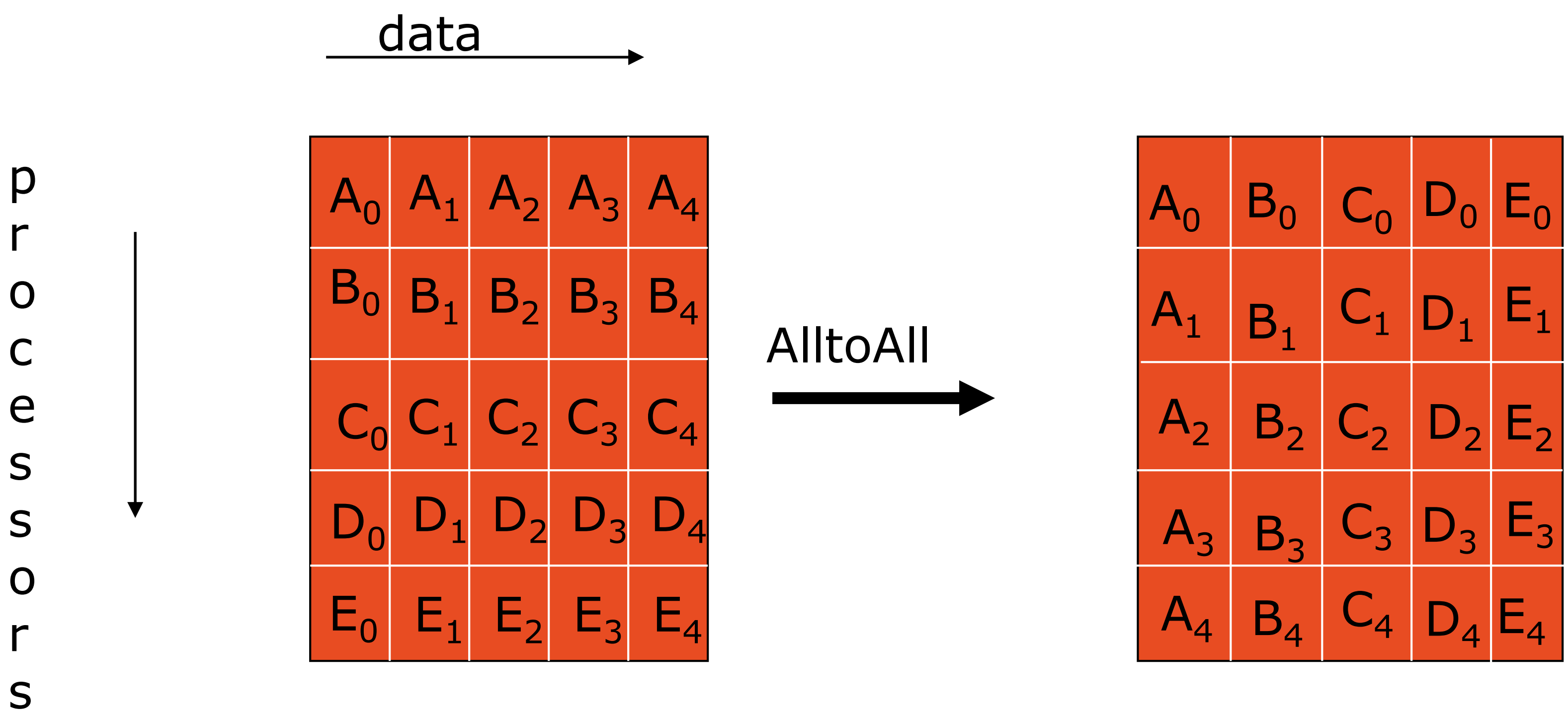
Collective Communications - Broadcast



Can be implemented as trees

MPI_BCAST(buffer, count, datatype, root, comm)

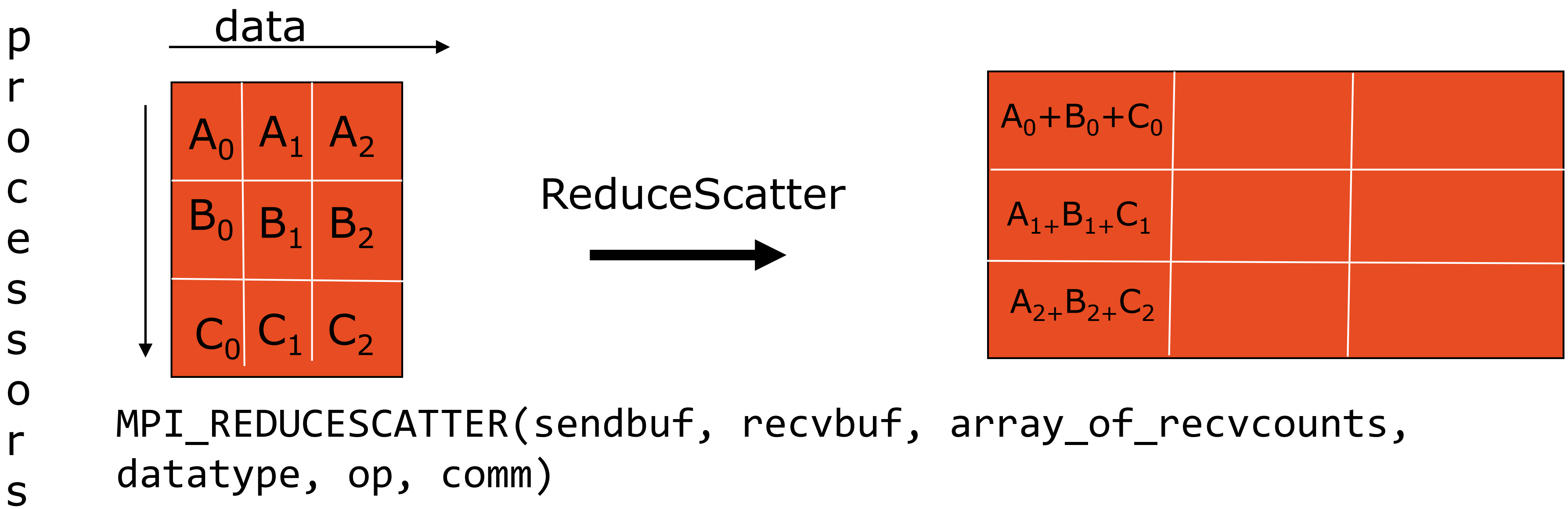
Collective Communications – AlltoAll



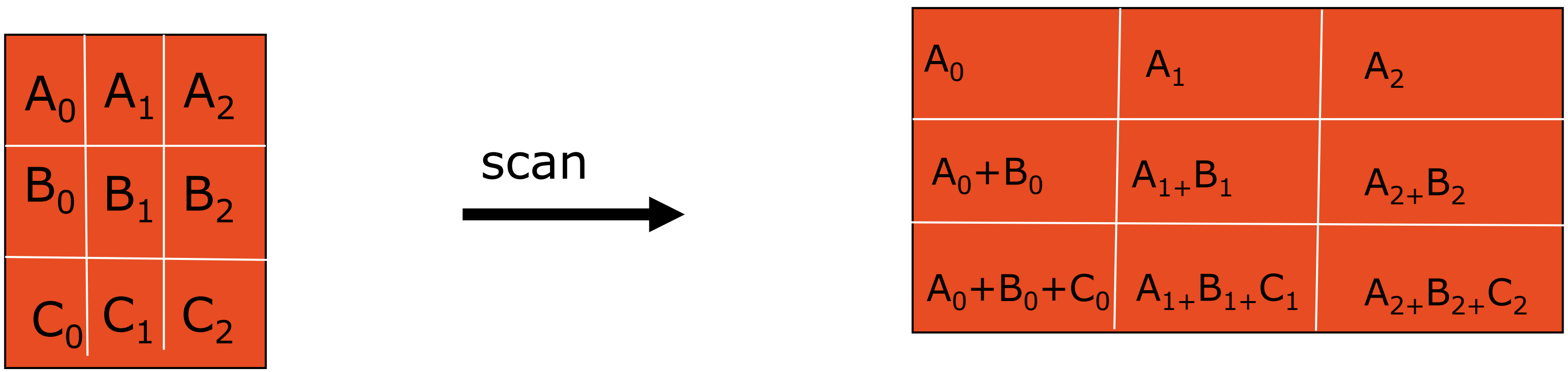
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

MPI_ALLTOALLV(sendbuf, array_of_sendcounts, array_of_displ, sendtype, array_of_recvbuf, array_of_displ, recvcount, recvtype, comm)

Collective Communications – ReduceScatter, Scan




```
MPI_REDUCESCATTER(sendbuf, recvbuf, array_of_recvcounts,
datatype, op, comm)
```




```
MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)
```