MPI – Message Passing Interface

Source: http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

Message Passing Principles

- Explicit communication and synchronization
- Programming complexity is high
- But widely popular
- More control with the programmer

MPI Introduction

- A standard for explicit message passing in MIMD machines.
- Need for a standard

  >> portability

  >> for hardware vendors

  >> for widespread use of concurrent computers
- Started in April 1992, MPI Forum in 1993, 1st MPI standard in May 1994, MPI-2 in 1997, MPI-3 in 2012

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)

Communication Primitives

- Communication scope
- Point-point communications
- Collective communications

Point-Point communications – send and recv

MPI_SEND(buf, count, datatype, dest, tag, comm)

Message

Rank of the destination

Message identifier

Communication context

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

MPI_GET_COUNT(status, datatype, count)

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

Utility Functions

- MPI_Init, MPI_Finalize
- MPI_Comm_size(comm, &size);
- MPI_Comm_rank(comm, &rank);
- MPI_Wtime()

Example 1: Finding Maximum using 2 processes

```
#include ''mpi.h''
int main(int argc, char** argv){
int n;
int *A, *local_array;
int max, local_max, rank1_max, i;
MPI_Comm comm;
MPI_Status status;
int rank, size;
int LARGE_NEGATIVE_NUMBER = -999999;

    MPI_Init(&argc, &argv);

    comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm, &size);
    MPI_Comm_rank(comm, &rank);

    if(size != 2){
      printf(''This program works with only two processes.\n'');
      exit(0);
    }

```

Example 1: Finding Maximum using 2 processes

```
if(rank == 0){
  /* Read N from console */
  MPI_Send(&N,1,MPI_INT,1,5,comm);
  /* Do dynamic allocation of A array with N elements */
  /* Initialize an array $A$ of $N$ elements */
  /* Do dynamic allocation of local_array with N/2 elements */
  for(i=0; i<N/2; i++){
    local_array[i] = A[i];
  }
  MPI_Send(A+N/2,N/2,MPI_INT,1,10,comm);
}
else{
  MPI_Recv(&N,1,MPI_INT,0,5,comm,&status);
  /* Do dynamic allocation of local_array with N/2 elements */
  MPI_Recv(local_array,N/2,MPI_INT,0,10,comm,&status);
```

Example 1: Finding Maximum using 2 processes

```
    local_max = LARGE_NEGATIVE_NUMBER;
    for(i=0; i<N/2; i++){
      if(local_array[i] > local_max){
        local_max = local_array[i];
      }
    }

    if(rank == 1){
      MPI_Send(&local_max,1,MPI_INT,0,15,comm);
    }
    else{
      max = local_max;
      MPI_Recv(&rank1_max,1,MPI_INT,1,15,comm);
      if(rank1_max > max){
        max = rank1_max;
      }
    }

     printf(``Maximum number is %d\n'', max);

     MPI_Finalize();

}
```

Buffering and Safety

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

Safe buffering:

	Process 0	Process 1
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 …………..

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)**

Non-blocking

- 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
- Efficiency depends on the implementation

Other Non-blocking communications

- **MPI_WAITANY(count, array_of_requests, index, status)**
- **MPI_TESTANY(count, array_of_requests, index, flag, status)**
- **MPI_WAITALL(count, array_of_requests, array_of_statuses)**
- **MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)**
- **MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)**
- **MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)**

Buffering and Safety

Process 0

Process 1

MPI_Send(1)
MPI_Send(2)
…………..

MPI_Irecv(2)
MPI_Irecv(1)
…………..

Safe

MPI_Isend
MPI_Recv
…………..

MPI_Isend
MPI_Recv
………

Safe

Example: Finding a Particular Element in an Array

```
#include ''mpi.h''
int main(int argc, char** argv){
....
....
int other_i, other_rank, flag;
MPI_Request request;
....
....

   ....
   ....
   other_rankif(rank == 0) other\_rank=1
   other_rank = (rank==0)?1:0;
   MPI_Irecv(&other_i,1,MPI_INT,other_rank,10,comm,&request);
   for(i=0; i<N/2; i++){
     if(A[i] == elem}{
     /* Ah, found it! Inform the other process of the index
         position*/
       MPI\_Send(&i,1,MPI_INT,other_rank,10,comm);
       if(rank == 0){
         global_pos = i;
       }
     /* Cancel the posted Irecv posted by this process */
       MPI_Cancel(&request);
       break;
     }
```

Example: Finding a Particular Element in an Array

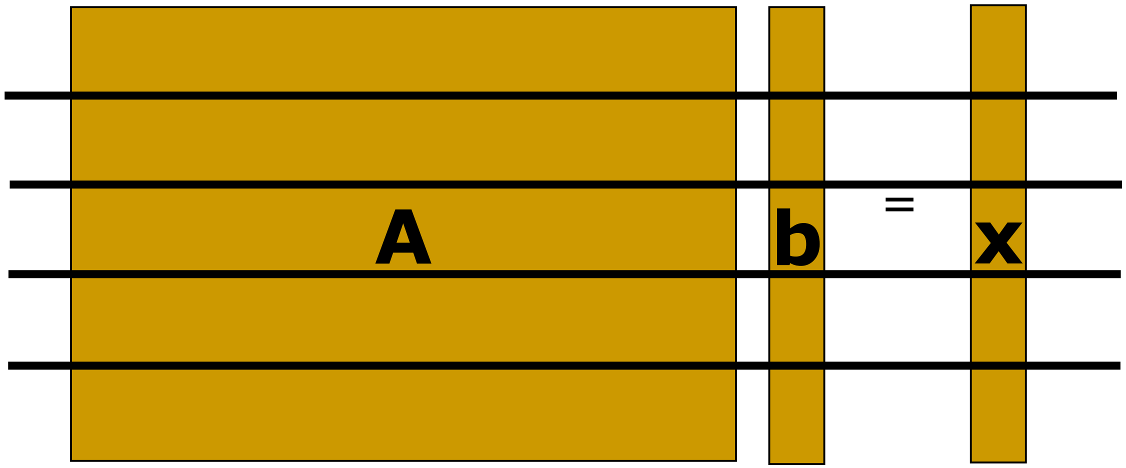
```
      else{
        MPI_Test(&request,&flag,&status);
        if(flag == 1){
        /* Ah, the other proccess has found it. */
          MPI_Wait(&request,&status);
          if(rank == 0){
            global_pos = other_i + N/2;
          }
          break;
        }
      }
    }

    if(rank == 0){
      printf("Found the element %d in position %d\n", elem,
          global_pos);
    }

     MPI_Finalize();
```

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

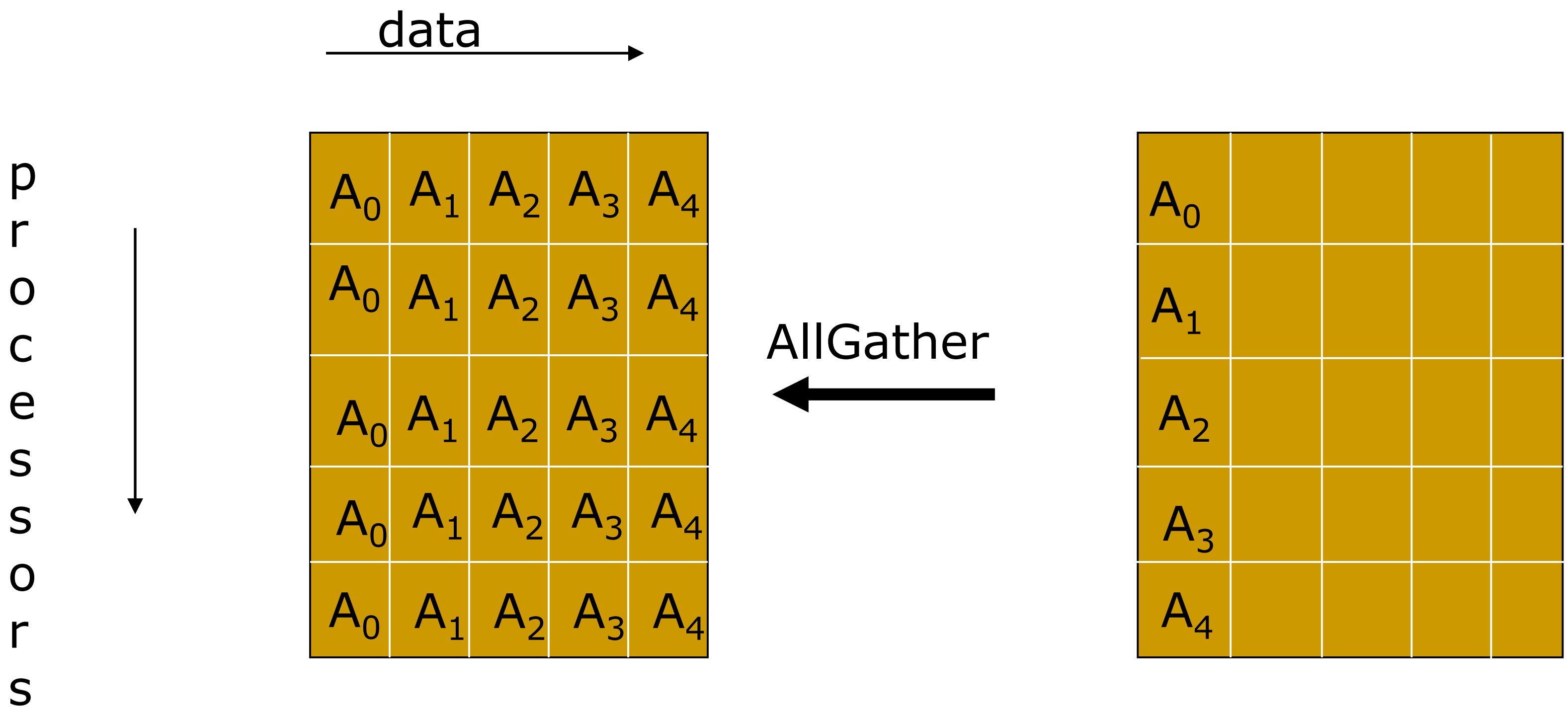
Example: Matrix-vector Multiply



Communication:

All processes should gather all elements of b.

Collective Communications – AllGather



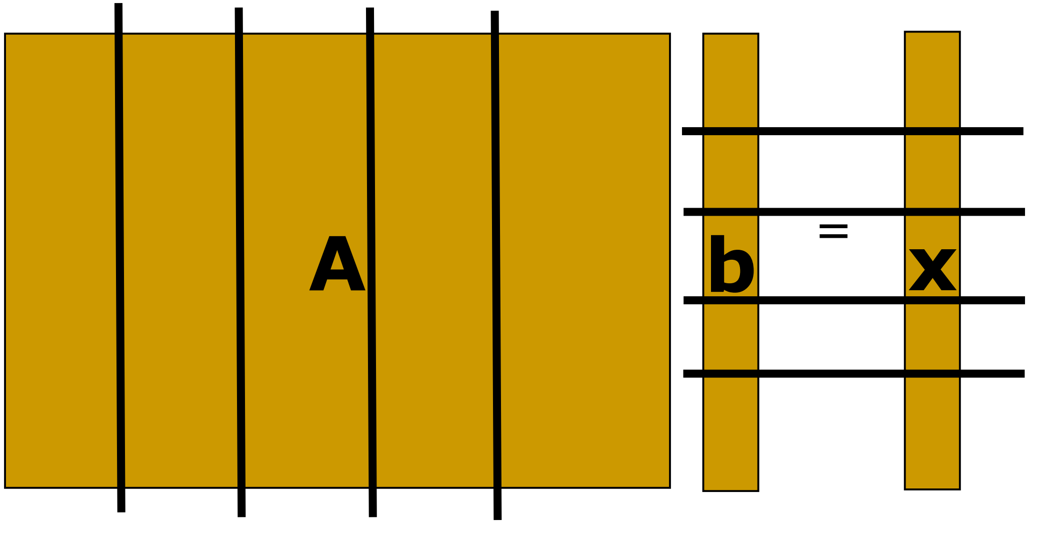
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, array_of_recvbuf, array_of_displ, recvcount, recvtype, comm)

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:

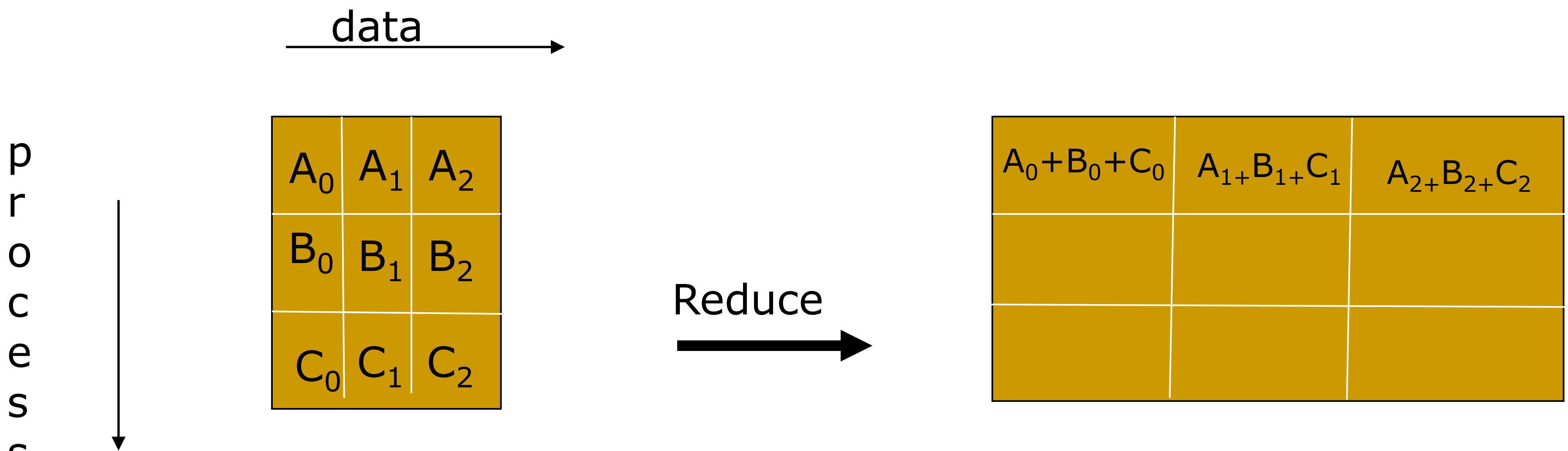
1. Each process computes its contribution to x

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

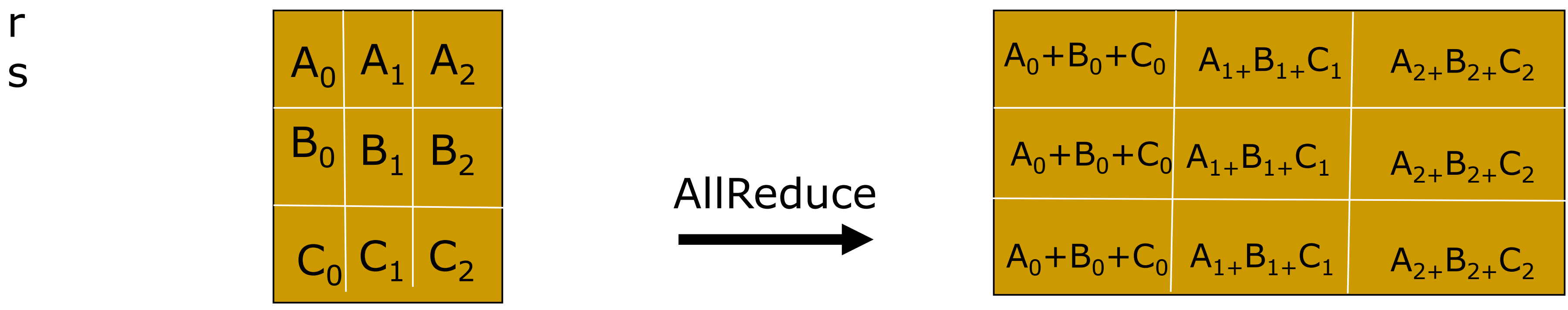
Example: Column-wise Matrix-Vector Multiply

```
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

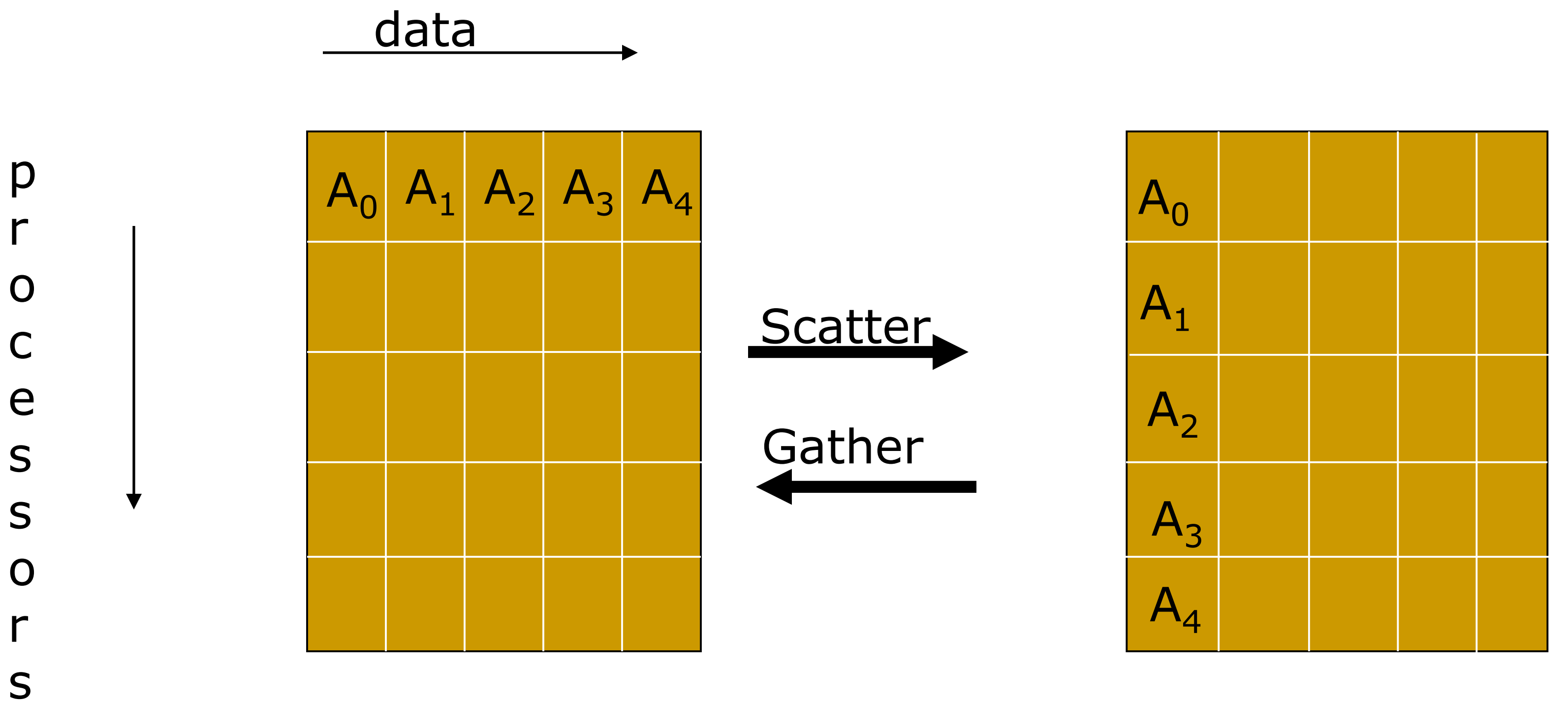


MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)



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);

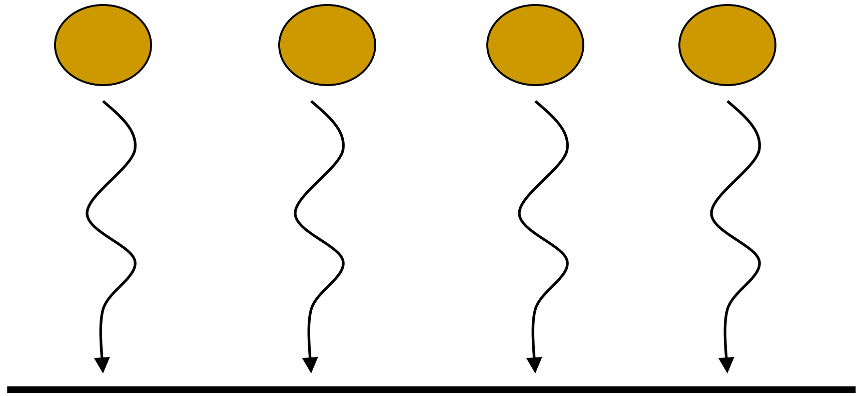
Or…

```
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

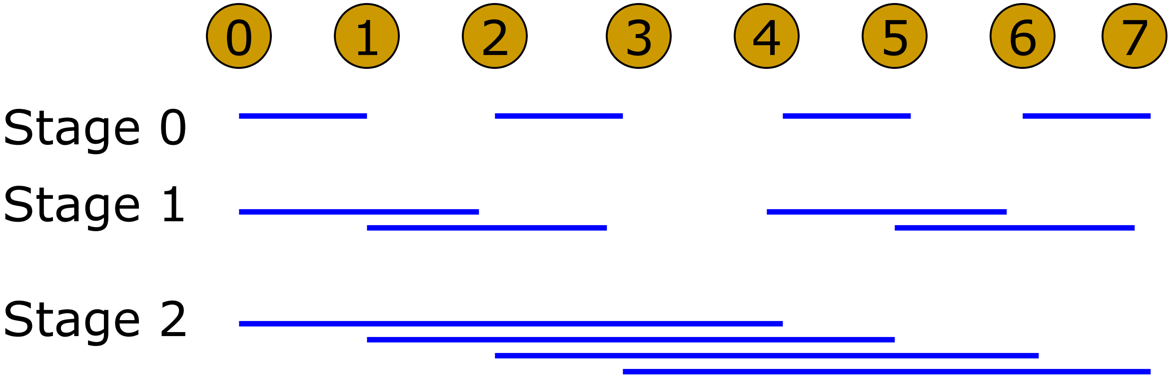


MPI_BARRIER(comm)

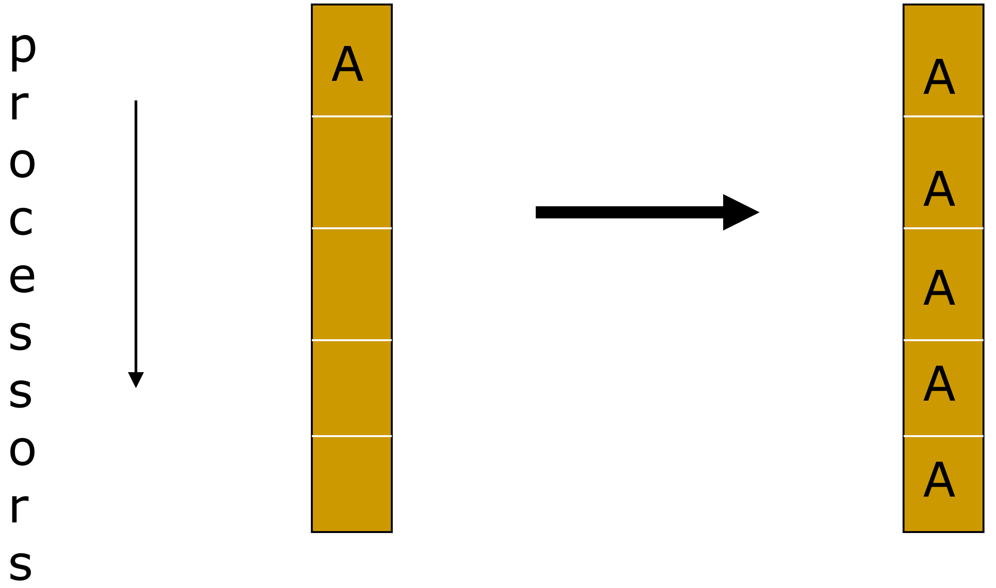
A return from barrier in one process tells the process that the other processes have ***entered*** the barrier.

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



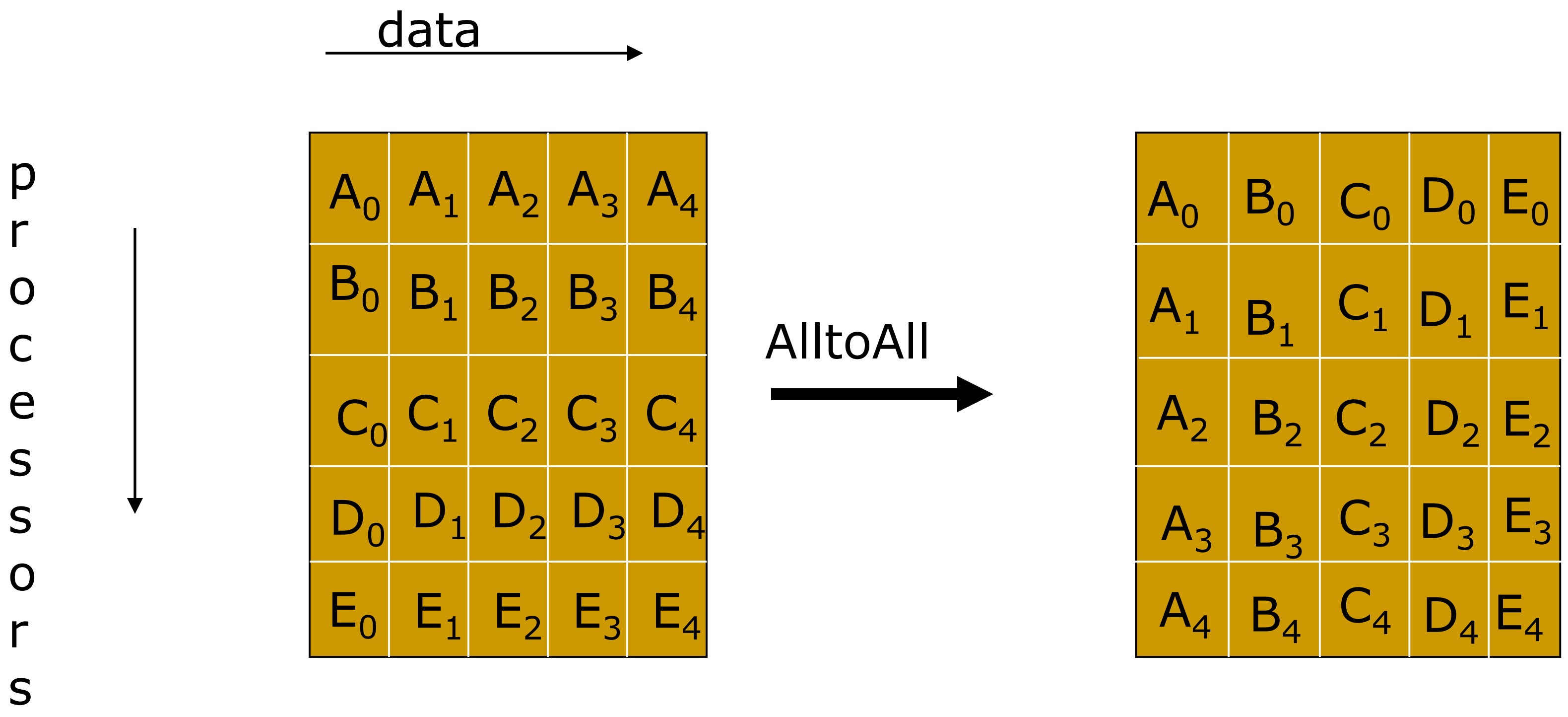
Collective Communications - Broadcast



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)

AlltoAll

- The naive implementation

```
 for all procs. i in order{
     if i # my proc., then send to i and recv from i
}
```

- MPICH implementation – similar to naïve, but doesn't do it in order

```
for all procs. i in order{
  dest = (my_proc+i)modP
  src = (myproc-i+P)modP
     send to dest and recv from src
}
```

Collective Communications – ReduceScatter, Scan

data →

processors ↓

A_0	A_1	A_2
B_0	B_1	B_2
C_0	C_1	C_2

ReduceScatter →

$A_0+B_0+C_0$		
$A_{1+}B_{1+}C_1$		
$A_{2+}B_{2+}C_2$		

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

A_0	A_1	A_2
B_0	B_1	B_2
C_0	C_1	C_2

scan →

A_0	A_1	A_2
A_0+B_0	$A_{1+}B_1$	$A_{2+}B_2$
$A_0+B_0+C_0$	$A_{1+}B_{1+}C_1$	$A_{2+}B_{2+}C_2$

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

Allgather implementation

- In general, optimized allxxx operations depend on hardware topology, network contentions etc.
- Circular/ring allgather
- Each process receives from left and sends to right
- P steps