Message Passing Computing

A closer look at MPI

Demo programs

Demo programs in: `/opt/examples`

Message Passing Interface

A standard form of communication across different processes.

- Point-to-point communication
- Collective communication
- Derived data types

Message tags

Demo: `tagtest.cpp`

```
if (rank == 0) {
    int N=12;
    int M=29;
    // Message A
    MPI_Send(&N, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    // Message B
    MPI_Send(&M, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    int A=0;
    int B=0;
    MPI_Status Stat;
    // Want to receive message B
    MPI_Recv(&B, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &Stat);
    printf("B: Got %d from master with tag %d\n", B, TAG2);

    // Want to receive message A
    MPI_Recv(&A, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &Stat);
    printf("A: Got %d from master with tag %d\n", A, TAG1);
  }
```

Blocking routines

MPI_Send() and MPI_Recv() are blocking routines.

Return when they are locally complete - when the location used to hold the message can be used again or altered without affecting the message being sent.

A blocking send will send the message and return This doesn’t mean that the message has been received, just that the process is free to move on without adversely affecting the message.

Non-blocking routines

Non-blocking send - MPI_Isend() will return immediately even before the source location is safe to be altered.

Non-blocking receive - MPI_Irecv() will return even if there is no message to accept.

Non-blocking routine formats

```
MPI_Request request
MPI_Isend(&buf, count, datatype, dest, tag, comm, &request)
MPI_Irecv(&buf, count, datatype, dest, tag, comm, &request)
```

- Completion detected by MPI_Wait() and MPI_Test()
- MPI_Wait(&request, &status) waits until operation is completed and then returns
- MPI_Test(&request, &flag, &status) returns with flag set indicating whether or not operation completed at that time.
- Operation determined by accessing the request parameter.

Example

See demo program `nonblock.cpp`

To send an integer x from process 0 to process 1 and allow process 0 to continue

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
        int x;
        MPI_Request req1;
        MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
        // Do something here, but DON'T modify x
        MPI_Wait(req1, status);
}

else if (myrank == 1) {
        int x;
        MPI_Request req2;
        MPI_Irecv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, req2);
        // Do something here but don't mess with x
        MPI_Wait(req2, status);
}
```

Send communication modes

- Standard mode
  - Not assumed that corresponding receive routine has started
  - Example: MPI_Send
- Buffered mode
  - Send may start and return before a matching receive
- Synchronous mode
  - Send and receive can start before each other but can only complete together
- Ready mode
  - Send can only start if matching receive already reached

Blocking and non-blocking

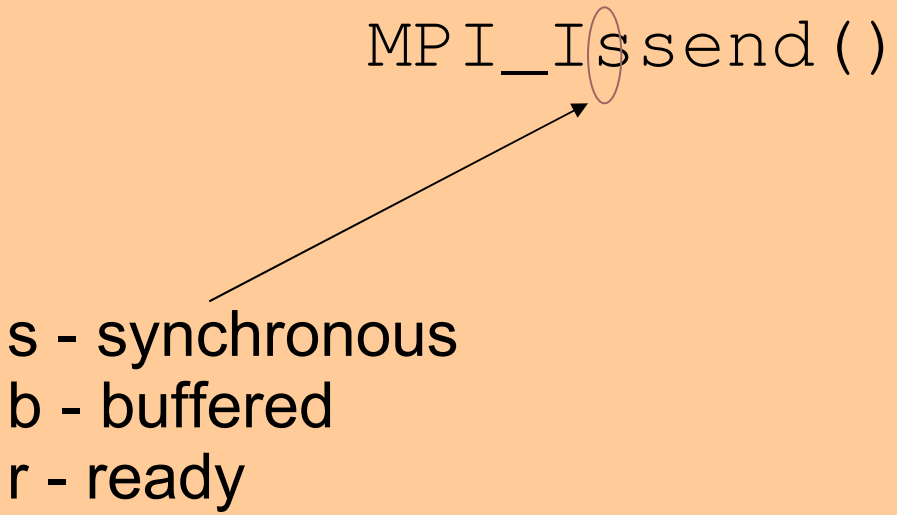
Each of the four modes can be applied to both blocking and non-blocking send routines.

Only the standard mode is available for the blocking and non-blocking receive routines.

Any type of send routine can be used with any type of receive routine.

Communication modes

The 3 non standard modes are identified in the routine mnemonics, eg non-blocking synchronous send



Collective communication

Involves a set of processes, defined by an intra-communicator

Communication coordinated amongst processes

3 types:
- Data movement
- Collective computation
- Synchronization

No tags or blocking.

Broadcast and scatter routines

`MPI_Bcast()`	- Broadcast from root to all other processes
`MPI_Reduce()`	- Combine values on all processes to a single value
`MPI_Scatter()`	- Scatters buffer in parts of group to other processes
`MPI_Gather()`	- Gathers values from group of processes
`MPI_Alltoall()`	- Sends data from all processes to all processes
`MPI_Scan()`	- Compute prefix reductions of data on processes

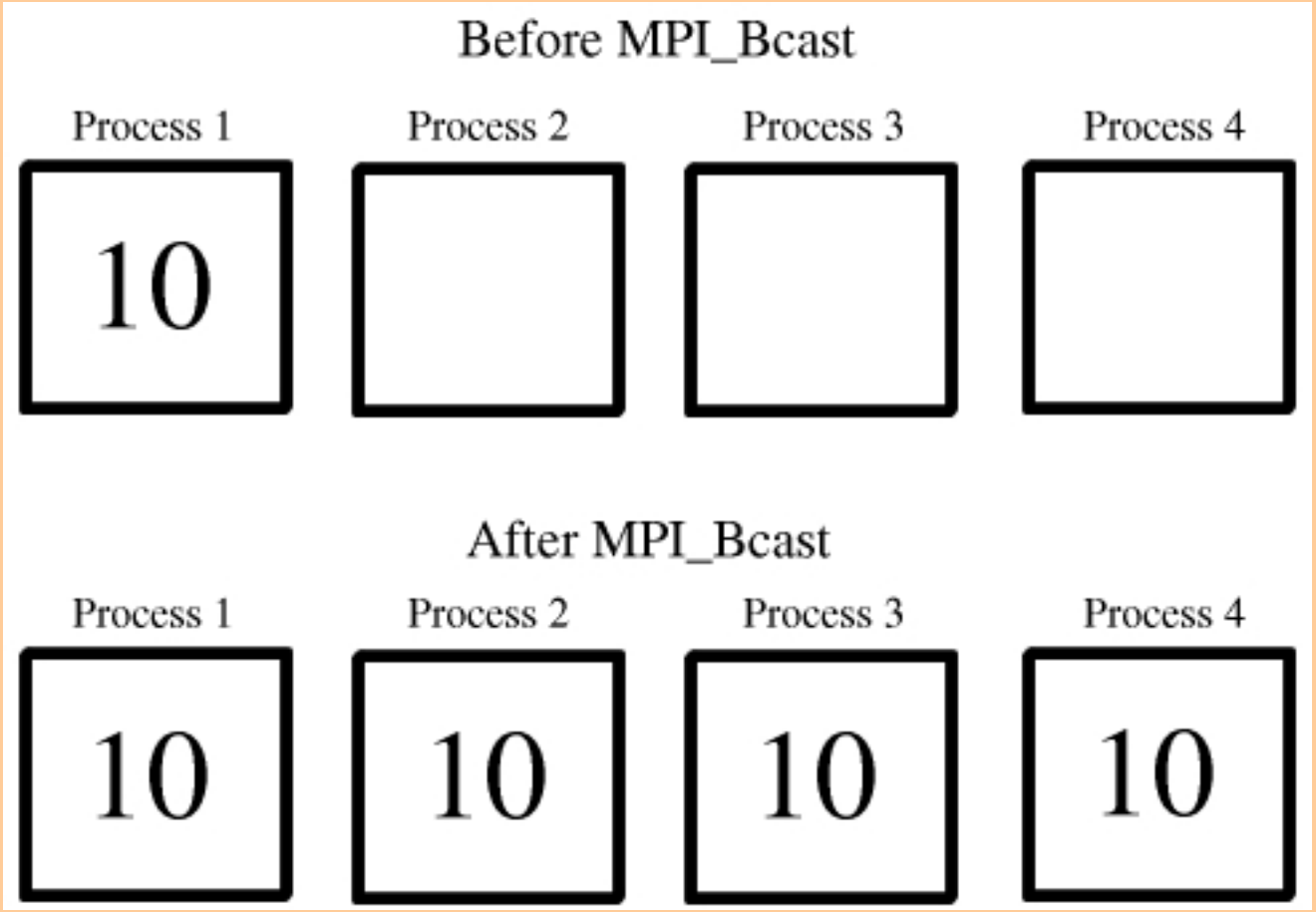
+ many others that are variations on these.

Data movement

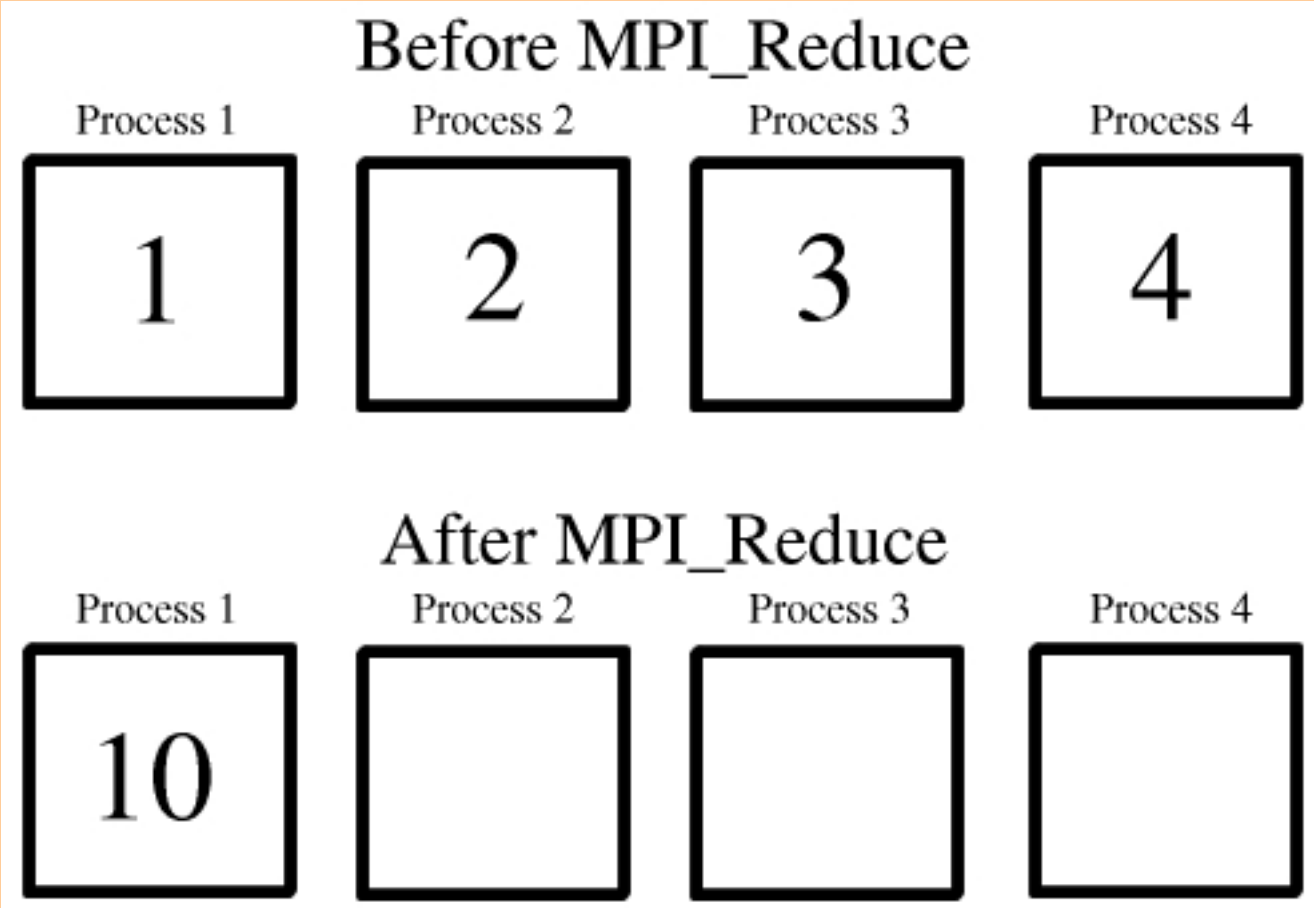
```
if (myid == 0) {//this is the master
   N=atoi(argv[1]);//get the number the user wants
   for  (i=1;i<numproc;i++) {//send to all nodes
     MPI_Send(&N, 1, MPI_INT, i,0, MPI_COMM_WORLD);
   }
   sum0=0;//partial result for node 0
   for(i=1;i<=N/numproc;i++) sum0=sum0+i;
   result=sum0;
   for (i=1;i<numproc;i++) {//receive from all nodes
     MPI_Recv(&sum1, 1, MPI_INT, i,0, MPI_COMM_WORLD, &Stat);
     result=result+sum1;//adds the various sums
   }
   fprintf(stdout,"The sum from 1 to %d is %d \n",N,result);
 }
 else {//this is not the master
   MPI_Recv(&N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &Stat);
   sum1=0;
   for(i=(N/numproc*myid)+1;i<=(N/numproc*(myid+1));i++)
     sum1=sum1+i;
   MPI_Send(&sum1, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
 }
```

This can be simplified by collective broadcasting.

MPI_Bcast


Before MPI_Bcast
Process 1
Process 2
Process 3
Process 4
10
After MPI_Bcast
Process 1
Process 2
Process 3
Process 4
10
10
10
10

MPI_Reduce


Before MPI_Reduce
Process 1
Process 2
Process 3
Process 4
1
2
3
4
After MPI_Reduce
Process 1
Process 2
Process 3
Process 4
10

Broadcast/reduce

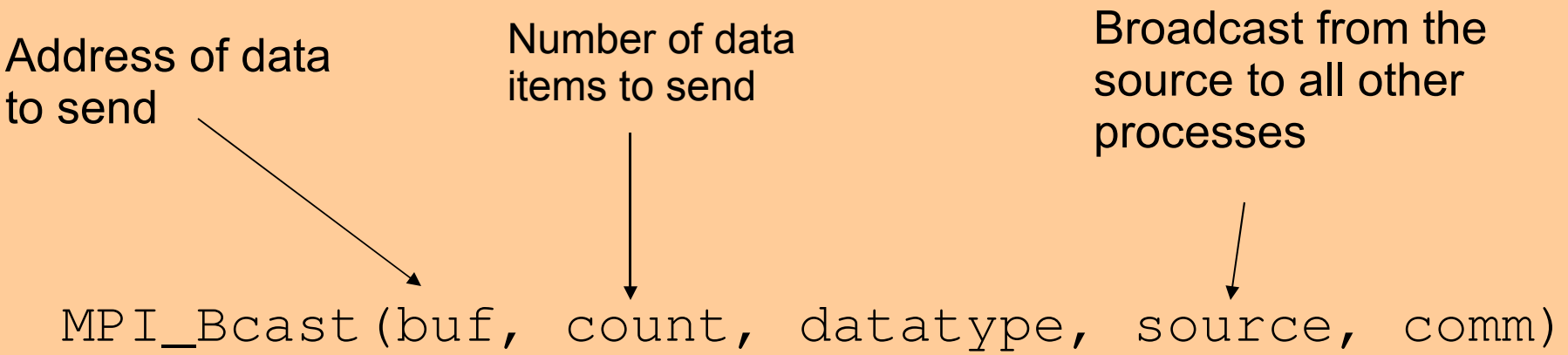
Demo: `broadcast.c`

Send N from master to all slaves

```
if (myid == 0) N = atoi(argv[1]);

MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

int first = N/numproc * myid + 1;
int last = N/numproc * (myid+1);
for (i=first; i <= last; ++i) sum +=i;

MPI_Reduce(&sum, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (myid==0) printf("Sum= %d\n", result);
```

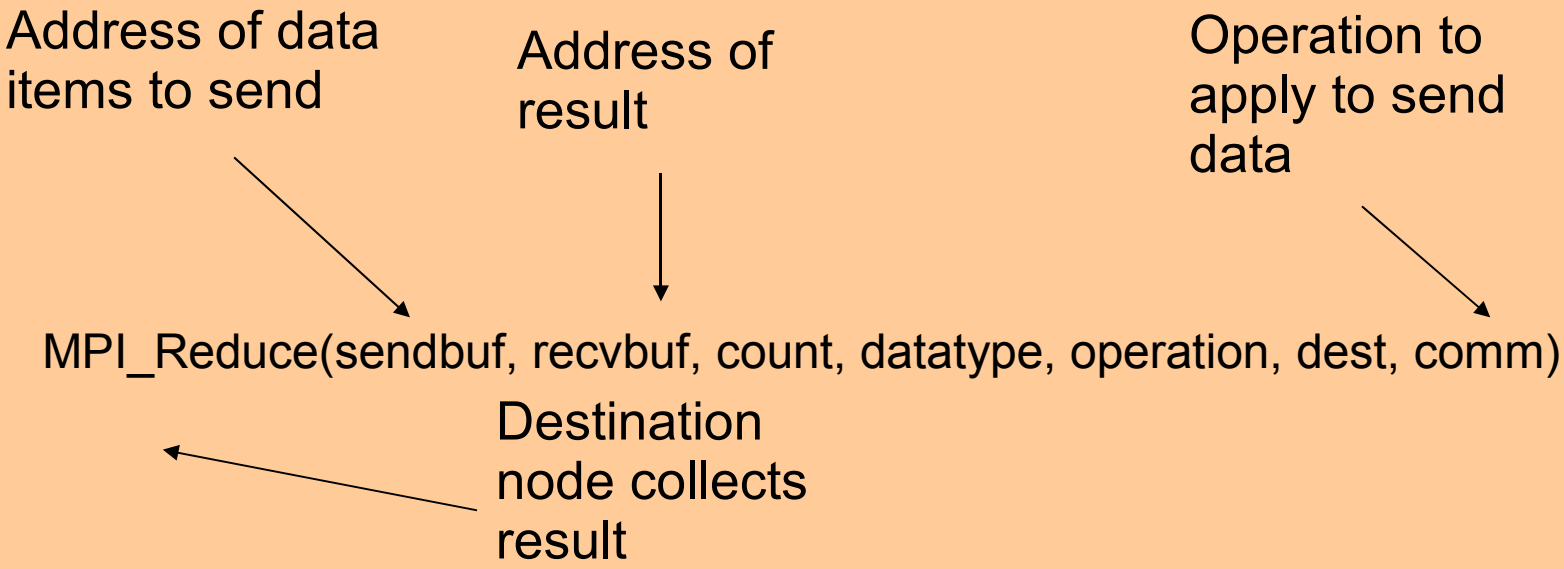
Combine sum from all slaves into result

MPI Broadcast



- Multinode send/receive are combined in one routine.
- If the process is the source, this routine sends the data to all other processes.
- If the process is not the source, this routine receives the data from the source.

MPI Reduce



If process is other than destination, send the data.

If process is destination, receive data, apply operation and put the result in recvbuf

Collective computation

MPI name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Summation
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical XOR

All of type MPI_Op

Gathering data

Demo: gather.c

Instead of collecting the results with MPI_Reduce, we could collect the individual partial sums using MPI_Gather().

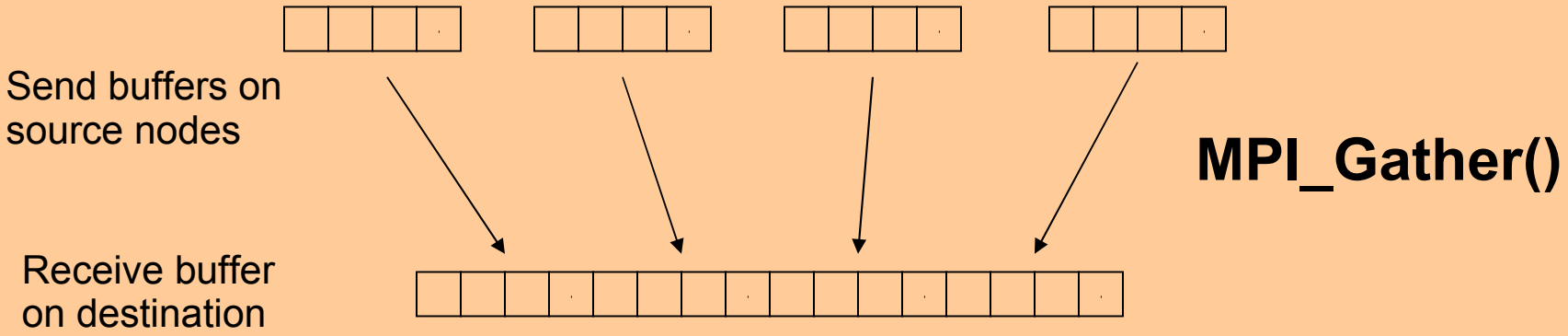
```
MPI_Gather(&sum, 1, MPI_INT, recvbuf, 1, MPI_INT, 0,
    MPI_COMM_WORLD);
```

The sum values from all slave processes are collected and stored in the data array recvbuf on the master process.

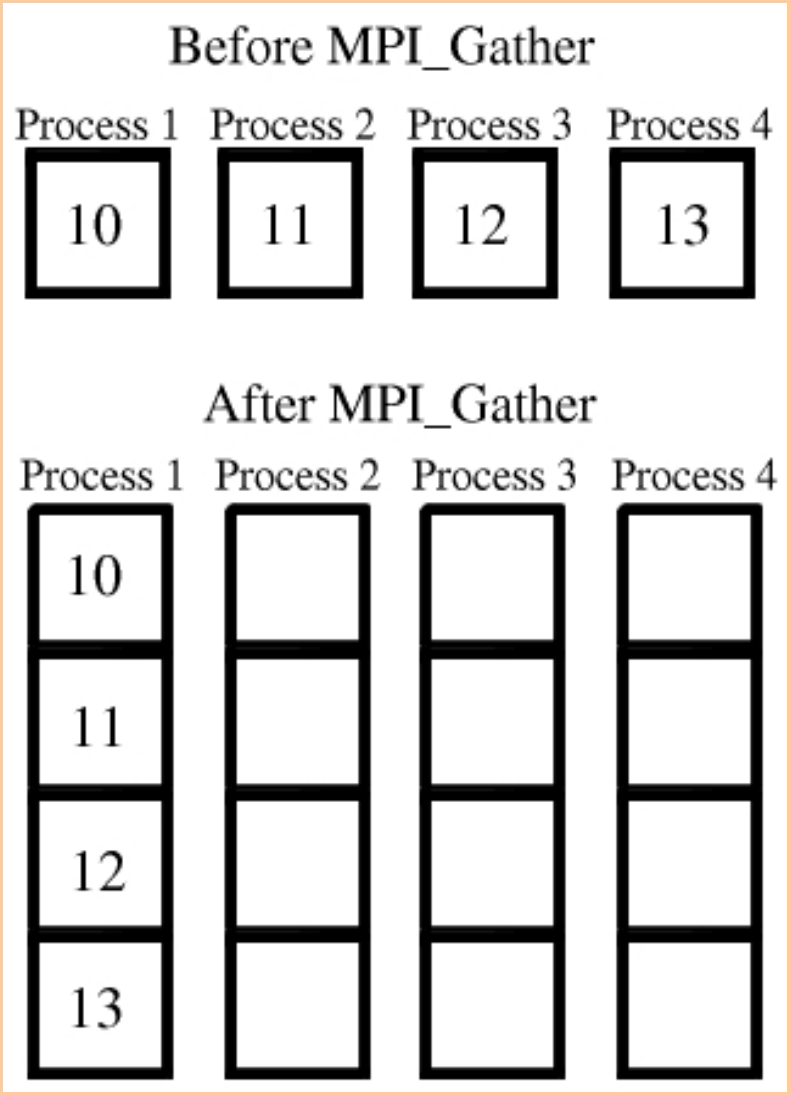
MPI_Gather

Demo: gather.c

MPI_Gather(sendbuf, sendcount, datatype, recvbuf, recvcount, datatype, root, comm)



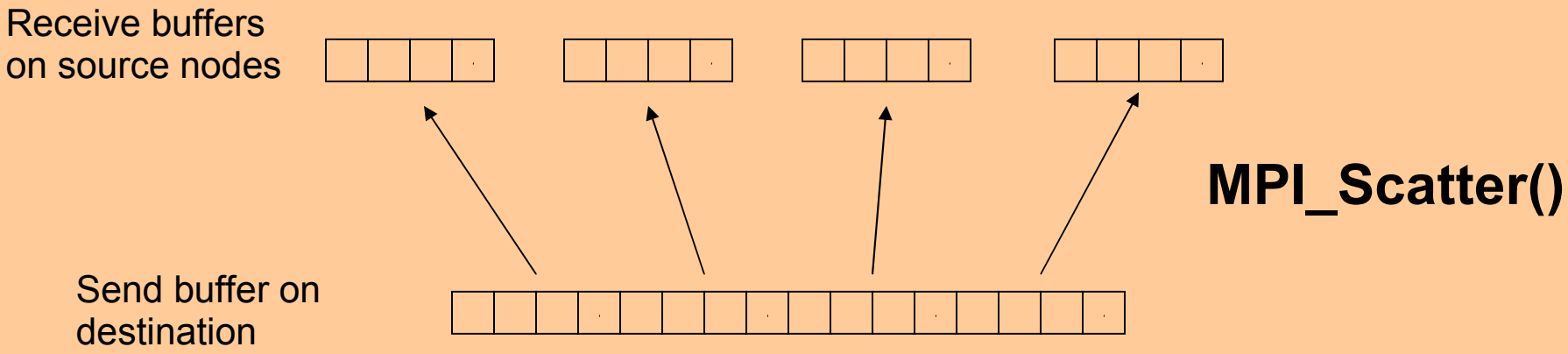
MPI_Gather


Before MPI_Gather
Process 1
Process 2
Process 3
Process 4
10
11
12
13
After MPI_Gather
Process 1
Process 2
Process 3
Process 4
10
11
12
13

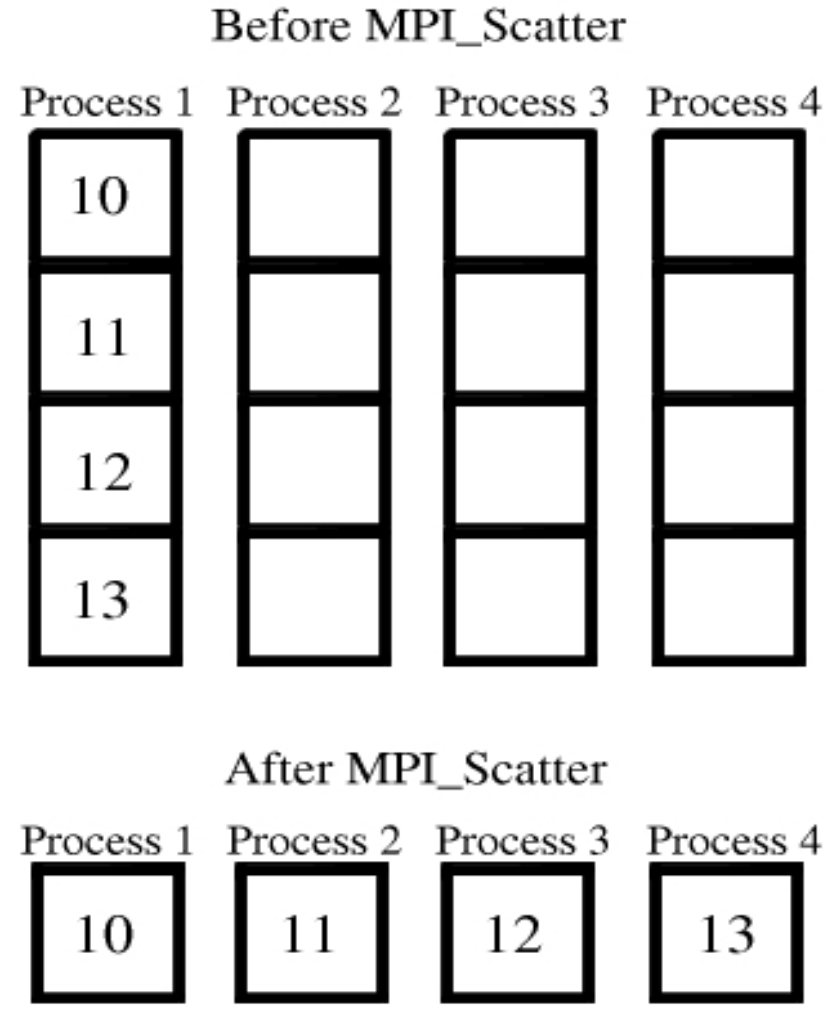
MPI_Scatter

Demo: scatter.c

MPI_Scatter(sendbuf, sendcount, datatype, recvbuf, recvcount, datatype, root, comm)



MPI_Scatter

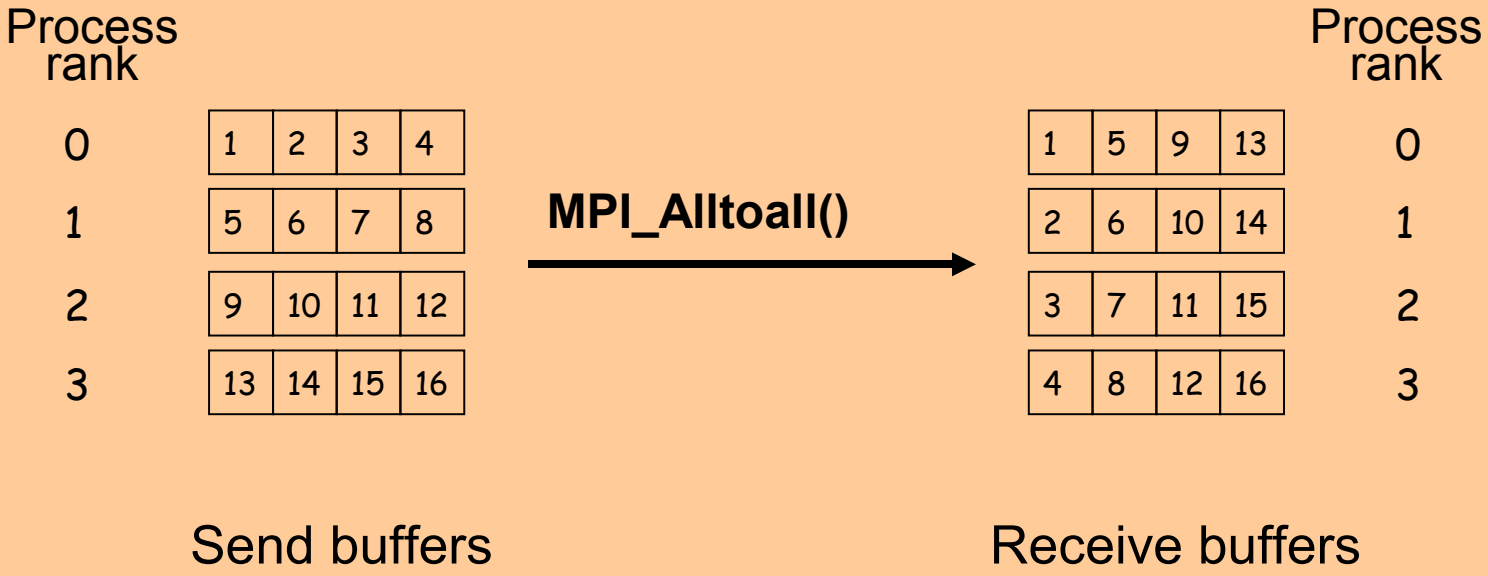

Before MPI_Scatter
Process 1
Process 2
Process 3
Process 4
10
11
12
13
After MPI_Scatter
Process 1
Process 2
Process 3
Process 4
10
11
12
13

MPI_Alltoall

Demo: alltoall.c

Each process sends data to each other process

MPI_Alltoall(sendbuf, sendcount, datatype, recvbuf, recvcount, datatype, comm)



Demo: `alltoall.c`

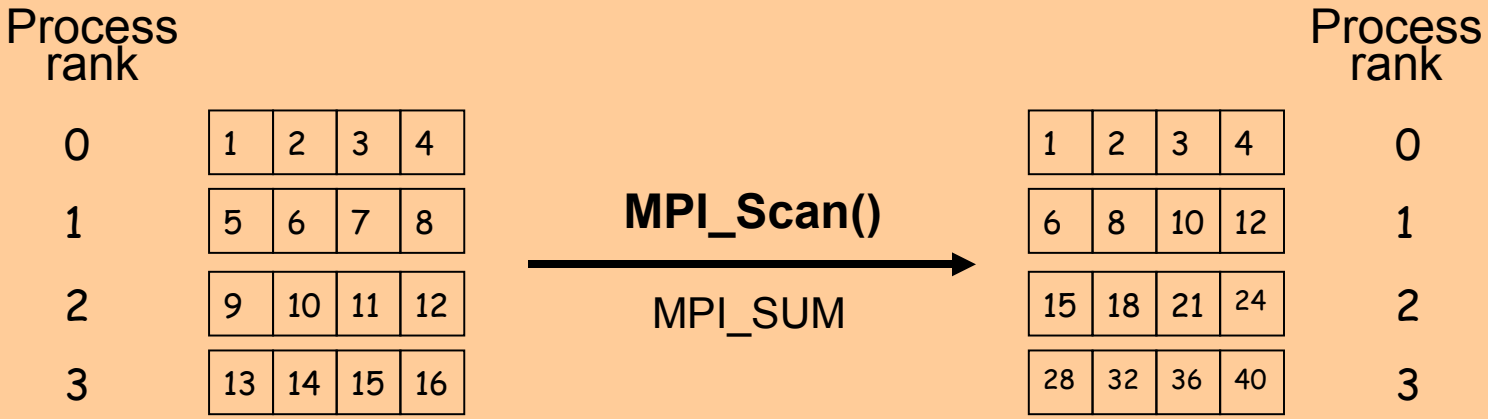
```
int *recvdata;
int *senddata;
int numproc, myid, i, N, ndata;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numproc);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
senddata = (int*)malloc(numproc * sizeof(int));
recvdata = (int*)malloc(numproc * sizeof(int));
for (i = 0; i < numproc; ++i)
  senddata[i] = 1 + myid * numproc + i;
printf("Before ID=%d : ", myid);
for (i = 0; i < numproc; ++i)
  printf(" %d", senddata[i]);
ndata = 1;
MPI_Alltoall(senddata, ndata, MPI_INT, recvdata, ndata, MPI_INT,
       MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
printf("After  ID=%d : ", myid);
for (i = 0; i < numproc; ++i)
  printf(" %d", recvdata[i]);
MPI_Finalize();
```

MPI_Scan

Demo: scan.c

Another collective computation routine

MPI_Scan(sendbuf, recvbuf, count, datatype, operation, comm)



Send buffers

Receive buffers

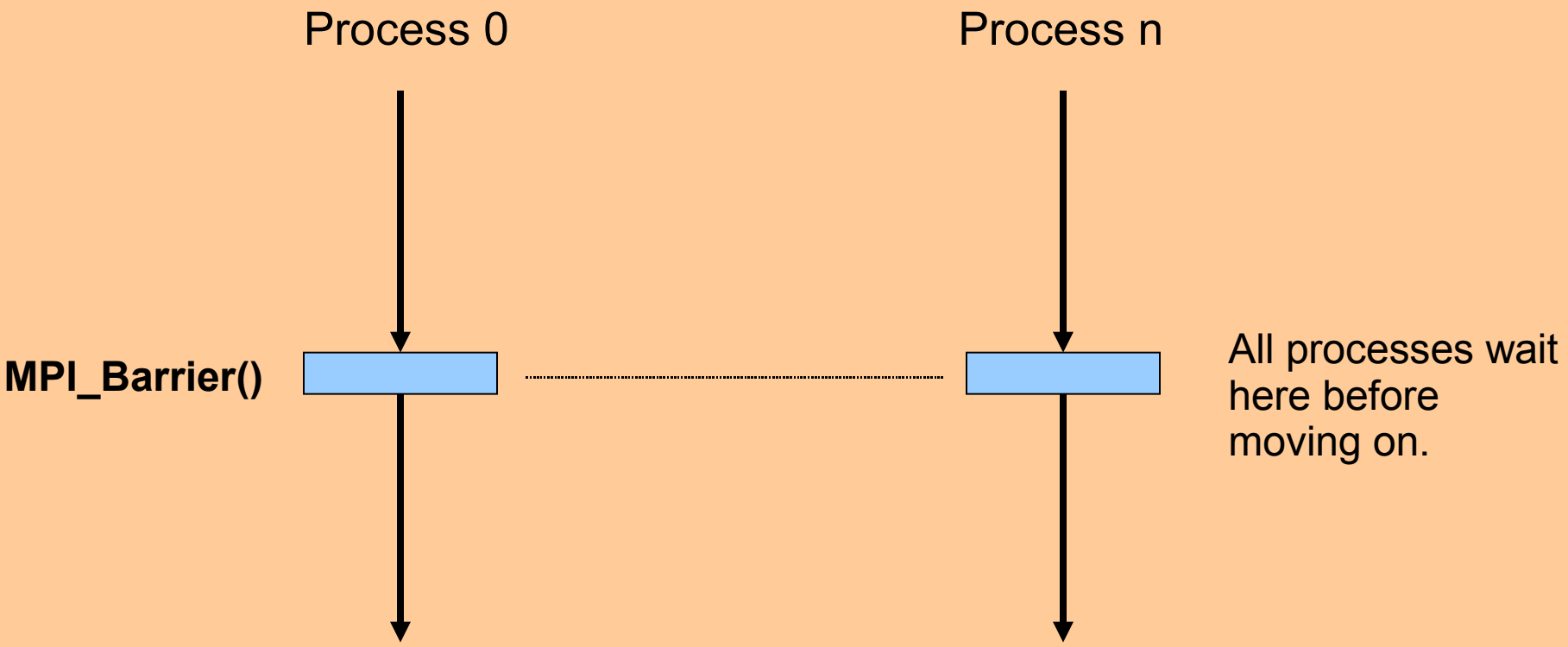
Synchronization

As in all message passing systems, MPI provides a means of synchronizing processes by stopping each one until they all have reached a specific “barrier call”.

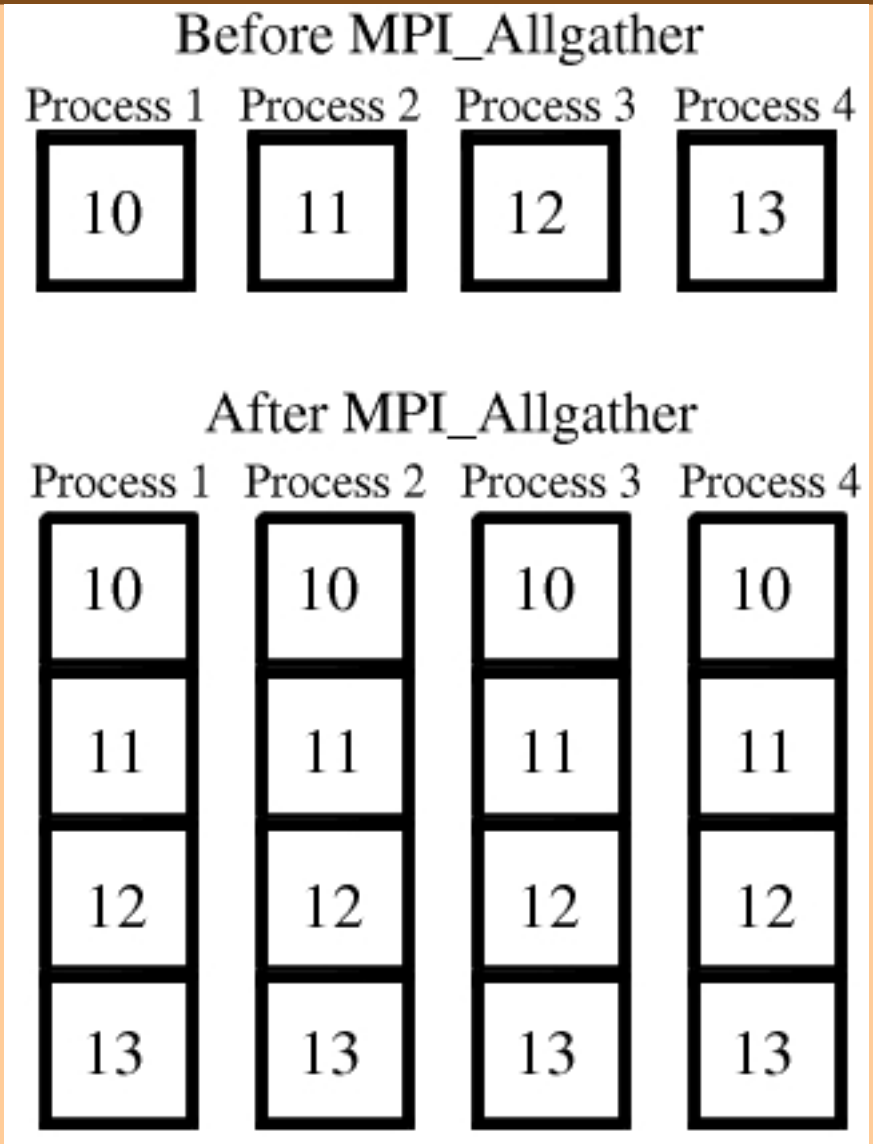
MPI_Barrier(comm)

Wait until all processes have called this routine before moving on.

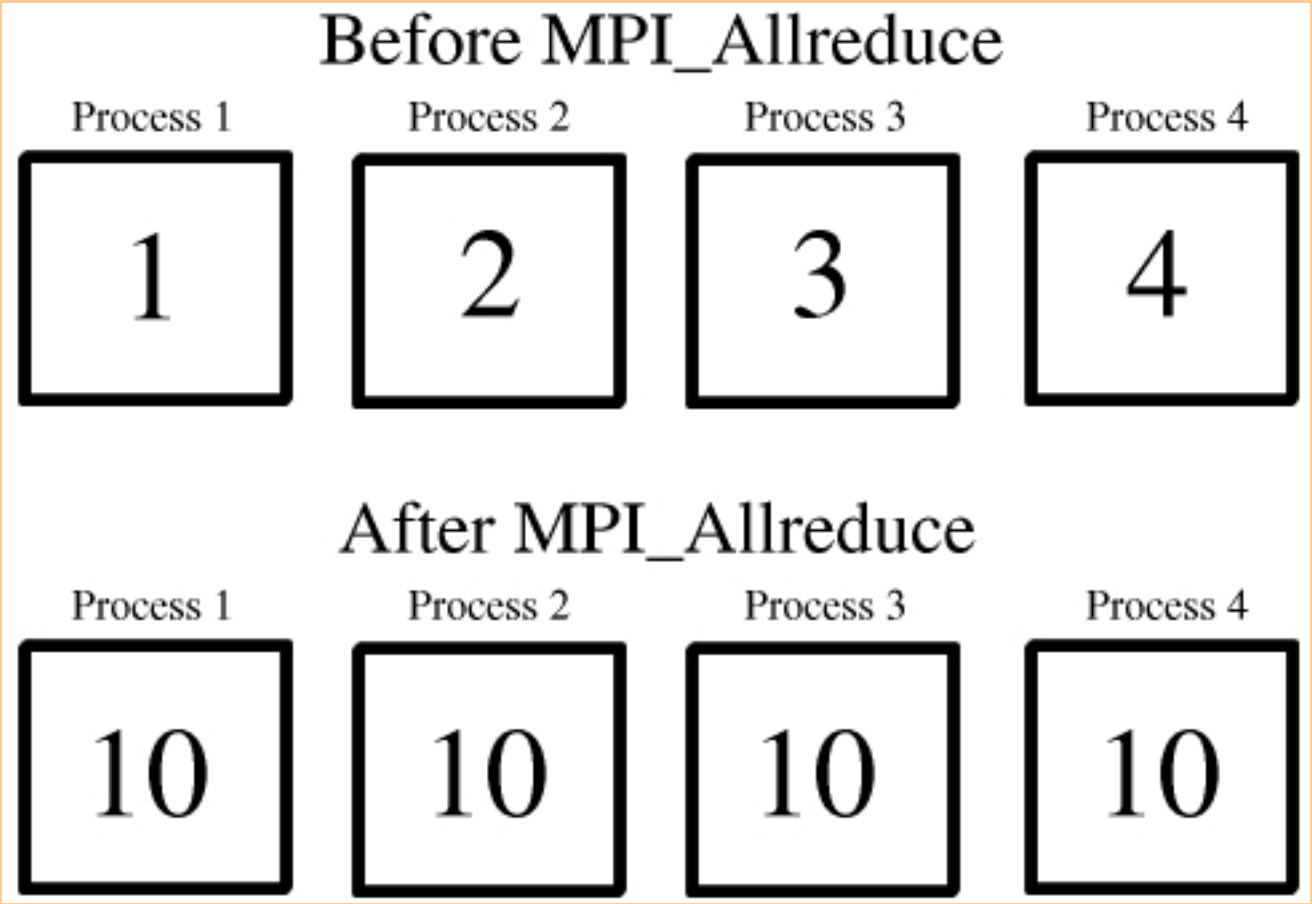
MPI_Barrier


Process 0
Process n
MPI_Barrier()
All processes wait here before moving on.

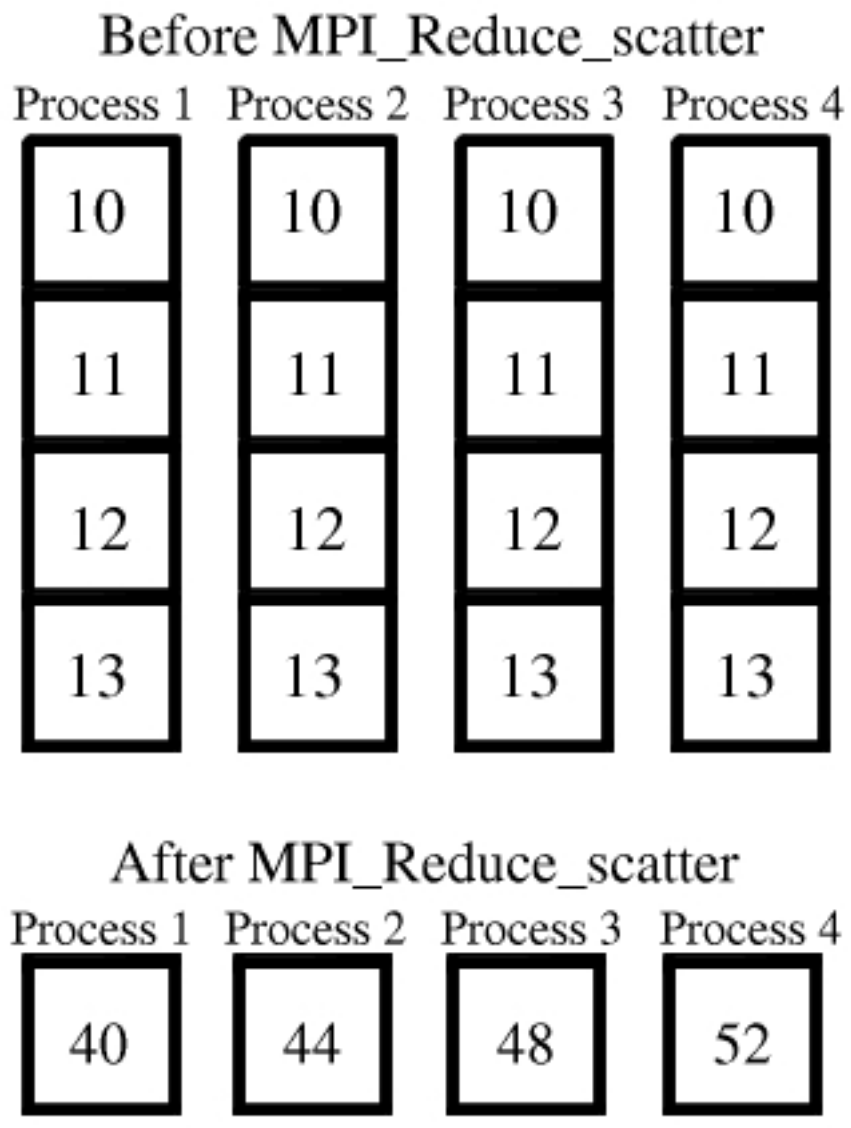
MPI_Allgather



MPI_Allreduce



MPI_Reduce_scatter



Derived data types

How do we send many variables in one message?

2 cases

- All of the **same** type:
  - use “count” field
  - Use contiguous derived data types
- Variables are **mixed** types
  - Used structured derived data types

Derived data types

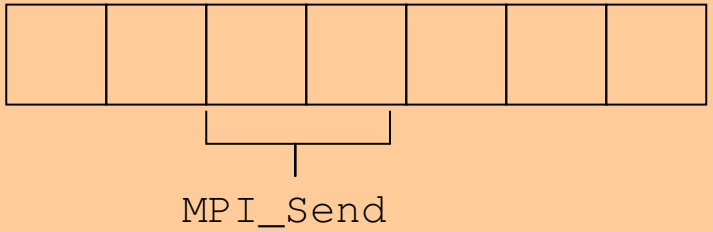
- 4 routines
  - MPI_Type_contiguous()
  - MPI_Type_vector()
  - MPI_Type_indexed()
  - MPI_Type_struct()
- Auxiliary routines
  - MPI_Type_extent()
  - MPI_Type_commit()
- Two examples: contiguous and struct

Derived data types - contiguous

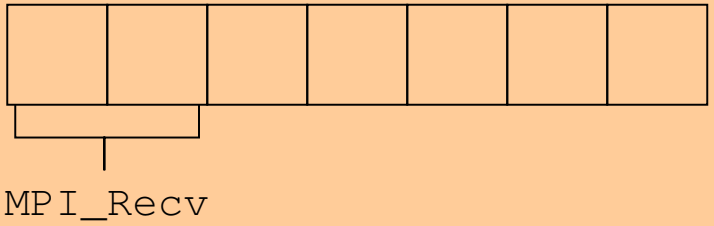
```
int numbers[SIZE] = {2,5,6,8,9,7,1,3,10};
Int numbers_slave[SIZE];
MPI_Datatype mytype;
MPI_Init(&argc, &argv);
MPI_Type(contiguous(SIZE/numproc, MPI_INT, &mytype);
MPI_Type_commit(&mytype):
if (myid == 0) {
        for (i=1; i<numproc; ++i)
                MPI_Send(&numbers[(SIZE/numproc)*i],1,mytype,
                        MPI_COMM_WORLD);
}

else {
        MPI_Recv(&numbers_slave, 1, mytype, 0, 0, MPI_COMM_WORLD,
                        &Stat);
}
```

MPI_Type_contiguous

numbers[SIZE]



numbers_slave[SIZE]

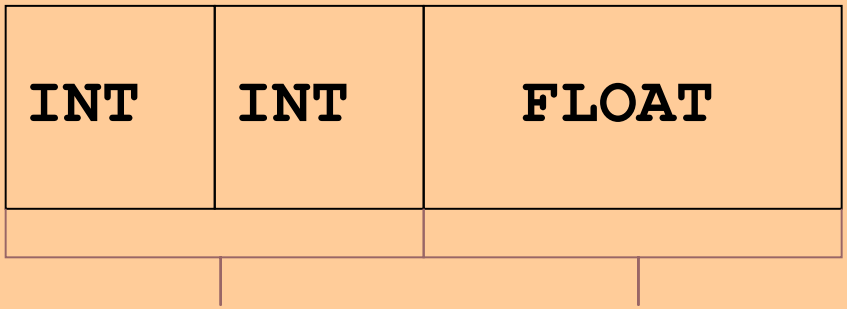


Derived data types

```
typedef struct {int x, y; float z;} message;
message slave_buf[SIZE], master_buf[SIZE];
MPI_Datatype mytype;
MPI_Datatype oldtypes[2] = {MPI_INT, MPI_FLOAT};
int blocklens[2] = {2,1};
MPI_Aint indicies[2], length;
…
MPI_Type_extent(MPI_INT, &length);
indicies[0]=0;
indicies[1]=2*length;
MPI_Type_struct(2, blocklens, indicies, oldtypes, &mytype);
MPI_Type_commit(&mytype);
```

MPI_Type_struct

count=2
(two sets)



blocklens[0]=2
indicies[0]=0
oldtype=MPI_INT

blocklens[1]=1
indicies[1]=2*length(MPI_INT)
oldtype=MPI_FLOAT

Can now send/recv data of new type “mytype”

Timing MPI programs - Wall clock time

```
double time1, time2, time3;
…
time1 = MPI_Wtime()
…
time2 = MPI_Wtime()
…
time3 = MPI_Wtime()
```

Wall clock time uncertainties

Non-deterministic communication pattern

Bottlenecks that do not depend on the master

Others?

Visualizing MPI time flows

Compile program with the mpe library and run the program as per normal

```
mpicc mpi_pi.c -o mpi_pi -llmpe -lmpe
or
mpecc -mpilog mpi_pi.c -o mpi_pi
```

Run program
produces mpi_pi.clog2
Visualise log file using jumpshot

You can add your own instrumentation using mpe calls.
See broadcast-log.c for details.