Parallel Programming and MPI- Lecture 2

Abhik Roychoudhury
CS 3211
National University of Singapore

Sample material: Parallel Programming by Lin and Snyder, Chapter 7.

Summary of previous lecture

- MPI as a programming interface
- Message passing communication
  - Communicating sequential processes
- Entering and Exiting MPI
  - MPI_Init, MPI_Finalize
- Point-to-point communication
  - Blocking & Non-blocking
  - MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv
  - Wait and test operations to complete communication.

In today's lecture

- Collective communication
  - Communicate between multiple processes simultaneously.
  - Substantially differs from send-receive based point-to-point communication studied earlier.
  - What are the communication primitives?

Collective communication in MPI

- Barrier communication across a set of processes.
- Global communication functions
  - Broadcast to a set of processes.
  - Gather data from all members for a member.
  - Scatter data to all members
- Global reduction operations
  - Possible reduction functions include sum, max, min etc
  - Accumulating return values from a set of processes, and employ a reduction function to obtain a result.
  - Result may be returned to all members, or only to a selected process.

Collective communication features

- In MPI, they have the following features
  - Amount of data sent must exactly match the amount of data specified by receiver.
  - No message tags are used.
  - Only blocking communication is allowed.

Communicators

- A scoping mechanism to define a set of processes, communicating with each other.
  - e.g. define a separate communicator for libraries, to keep messages from library routines distinct from appl. level routines.
  - A group of processes, assigned with a globally unique id.
- A group is an ordered set of processes.
  - Each process in the group has a unique rank.
    - Previous lecture!
  - A process can, of course, belong to multiple groups.
  - We can assume that the communicators we deal with, have its own group as well.

Barrier synchronization

- int MPI_Barrier(MPI_Comm comm)

- Blocks the caller, until all group members have called it.
- Returns at any process, only after all group members have entered the call.

Global communication

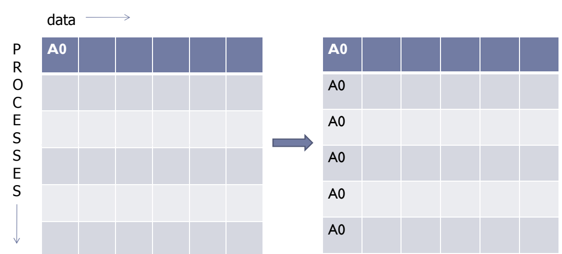
- Broadcast
- Scatter
- Gather
- Allgather
- …

Broadcast

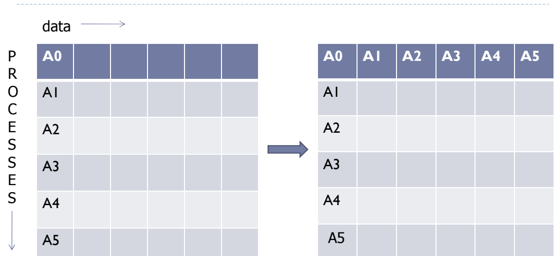
- Int MPI_Bcast(buffer, count, datatype, root, comm)
  - Starting address of buffer
  - # of entries in buffer
  - Data type of buffer
  - Rank of the broadcasting process
  - The communicator capturing the group of processes.
- Example:

```
MPI_Comm comm;
int array[100], root = 0;
…
MPI_Bcast(array, 100, MPI_INT, root, comm);
```


Broadcast



Gather



MPI_Gather

- int MPI_Gather(sendbuf, sendcount, sendtype,
- recvbuf, recvcount, recvtype,
- root, comm)
  - Starting addr, # of elem., datatype of send buffer
  - Starting addr, # of elem., in any single receive, datatype of receive buffer
    - Receive buffer is ignored for non-root processes.
  - Rank of receiving process
  - Communicator
- Each process (root process also) sends contents of its send buffer to root process.
- Root process receives messages, and stores them in rank order, in the receive buffer.

Effect of Gather

- As if
  - All N processes in the group (including root) execute
    - MPI_send(sendbuf, sendcount, sendtype, root, …)
  - Root executes N receives
    - MPI_recv(recvbuf,+i * …, recvcount, recvtype, i, …)
- Example: Gather 100 ints from every process to the root.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf;
…
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray,100,MPI_INT, rbuf,100,MPI_INT, root, comm)
```

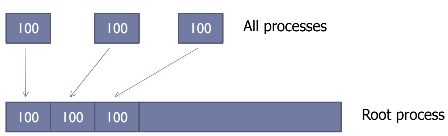

More on Gather

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, myrank, *rbuf;
…
MPI_Comm_rank(comm, &myrank);
if (myrank == root){
    MPI_Comm_size(comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray, 100, MPI_INT, rbuf,100,MPI_INT, root, comm);
```

- This code is better, since only the root process allocates memory for the receive buffer.

Gather

- We ensure that only root process allocates memory for receive buffer.



In place gathering

```
MPI_Comm comm; int gsize;
int root, myrank, *rbuf, *sbuf;
MPI_Comm_rank(comm, &myrank);
if (myrank == root){
   MPI_Comm_size(comm, &gsize);
    rbuf = (int *) malloc(gsize*100*sizeof(int));
    sbuf = rbuf + 100*myrank;
} else{
     sbuf = (int *)malloc(100*sizeof(int));
}
if (myrank == root)
    MPI_Gather(MPI_IN_PLACE,100,MPI_INT,rbuf,100,MPI_INT,root,comm);
else
    MPI_Gather(sbuf,100, MPI_INT,0, 0, 0, root, comm);
```

1. Contribution of the root towards the gathered data assumed to be in place.
2. Separate Gather call patterns in diff processes.

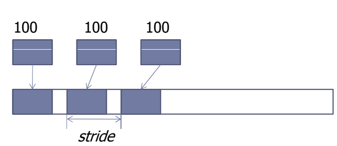
Gather, Vector variant

MPI_Gatherv(sendbuf, sendcount, sendtype,
recvbuf, recvcounts, displs, recvtype,
root, comm)

- recvcounts --- is an array of integers
  - Different counts from different sending processes
- displs --- is an array of integers
  - Provides flexibility of where the data is placed in the root.
  - Root process places the data of process i at the location
    - recvbuf + displs[i]

Example

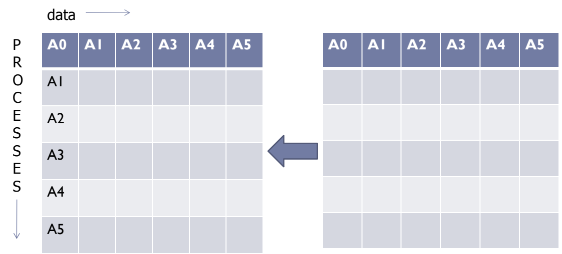
- Each process sends 100 integers to the root process.
- Each set of 100 integers is placed *stride* integers apart.
- Assume *stride* ≥ 100



The solution

```
MPI_Comm comm;
int gsize, sendarray[100], root, *rbuf, stride, *displs, i, *rcounts;
…
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i < gsize; i++){
    displs[i] = i* stride;  rcounts[i] = 100;
}
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                root, comm);
```


MPI_Scatter



The inverse operation of MPI_Gather.

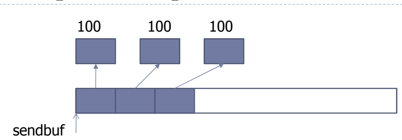
MPI_Scatter

- int MPI_Scatter(sendbuf, sendcount, sendtype,
- recvbuf, recvcount, recvtype,
- root, comm)
  - Starting addr, # of elem., datatype of send buffer
  - Starting addr, # of elem., datatype of receive buffer
  - Rank of receiving process
  - Communicator

MPI_Scatter

- As if, the root sends n messages
  - MPI_Send(sendbuf+i*…, sendcount, sendtype, i, …)
- And each process executes a receive
  - MPI_Recv(recvbuf, recvcount, recvtype, root, …)

A simple example



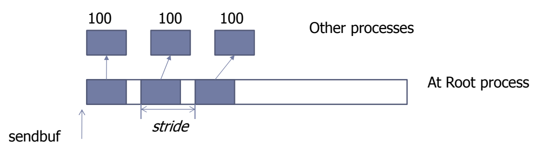
- MPI_Scatter(sendbuf, 100, MPI_INT,
- rbuf, 100, MPI_INT,
- root, comm)

MPI_Scatterv, vector variant.

- Inverse operation of MPI_Gatherv
- Extends MPI_Scatter by
  - Allowing variable amount of data to be sent to each process.
  - Also, allows flexibility about where the data is taken from the root – by allowing a displs argument (similar to MPI_Gatherv)
- MPI_Scatterv(sendbuf, sendcounts, displs, sendtype,
- recvbuf, recvcount, recvtype,
- root, comm)
  - sendcounts is an array of integers
  - displs is an array of integers

Example

- Each process receives 100 integers from root process.
- Each set of 100 integers are *stride* integers apart, in the send buffer.
- Assume *stride* ≥ 100



The solution

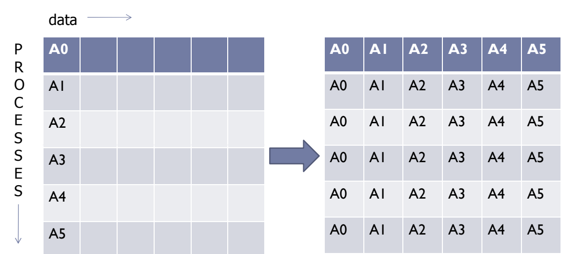
```
MPI_Comm comm;
int gsize, *sendbuf,root,stride, rbuf[100], i, *displs, *scounts;
…
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
…
displs  = (int *)malloc(gsize*sizeof(int));
scounts = (int *) malloc(gsize*sizeof(int));
for(i = 0; i < gsize; i++){
   displs[i] = i*stride; scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT,  rbuf, 100, MPI_INT,
                root, comm);
```


Gather to All

- MPI_Allgather(sendbuf, sendcount, sendtype,
- recvbuf, recvcount, recvtype,
- comm)

- There is no root process.
- All-to-all communication.
- All processes receive the gathered result, rather than only the root process.
- As if all the N processes executed N calls to MPI_Gather with root = 0,1,…, N-1.

Allgather



Gather to All – Vector variant

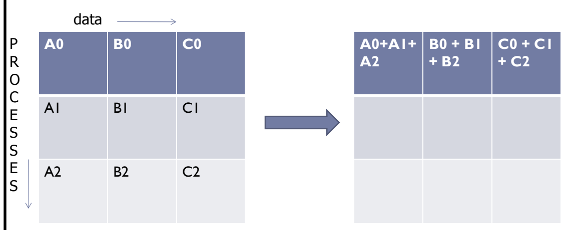
- MPI_Allgatherv(sendbuf, sendcount, sendtype,
- recvbuf, recvcounts, displs, recvtype,
- comm)

- There is no root process.
- All processes receive the gathered result, rather than only the root process.
- As if all the N processes executed N calls to MPI_Gatherv with root = 0,1,…, N-1.

Recall: Collective comm. in MPI

- Barrier communication across a set of processes.
- Global communication functions
  - Broadcast to a set of processes.
  - Gather data from all members for a member.
  - Scatter data to all members.
- Global reduction operations
  - Possible reduction functions include sum, max, min etc
  - Accumulating return values from a set of processes, and employ a reduction function to obtain a result.
  - Result may be returned to all members, or only to a selected process.

MPI_reduce



MPI_Reduce using MPI_SUM as the reduction operation.

MPI_Reduce

- MPI_Reduce(sendbuf, recvbuf, count, datatype,
- op, root, comm)
  - Addr of send, recv buffer
  - count is Number of elements in send buffer
  - Datatype of elements in send buffer
  - Reduction operation to be performed.
  - The root process who receives the reduced result
  - The communicator.

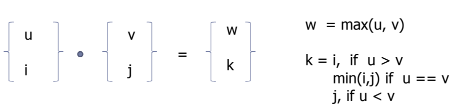
So, what does MPI_Reduce do?

- Combine the elements in the sendbuf of each process
  - Use operation op to combine them.
- Place the combined value in recvbuf
  - Recvbuf accessed by root process.
- Predefined reduction operations
  - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD
  - MPI_LAND, MPI_LOR, MPI_LXOR
    - Logical operations
  - MPI_BAND, MPI_BOR, MPI_BXOR
    - Bitwise operations
  - MPI_MAXLOC, MPI_MINLOC
    - Max value and location, Min value and location.

More on MPI_Reduce

- Predefined reduction operations
  - MPI_MAXLOC, MPI_MINLOC
    - Max value and location, Min value and location.
    - One application is to obtain the min/max value and the rank of the process containing the min/max value.
  - Requires new types at the receiver's end
    - Say the values are integers
      - Receiver's type will be MPI_2INT
    - Or, say the values are floating point numbers
      - Receiver's type will be MPI_FLOAT_INT

MPI_MAXLOC



MPI_MAXLOC computes a global maximum and an index associated with it.

In the above illustration schematic,

-u and v are the values over which a maximum is taken.

-i and j are the indices associated with u and v.

Using MPI_Reduce

- **The dot product** is an algebraic operation that takes two equal-length sequences of numbers and returns a single number obtained by multiplying corresponding entries and adding up those products. The name is derived from the dot that is often used to designate this operation; the alternative name is **scalar product.**
- **Compute the dot product of two vectors that are distributed across a group of processes, and return the answer at process zero.**

Code template

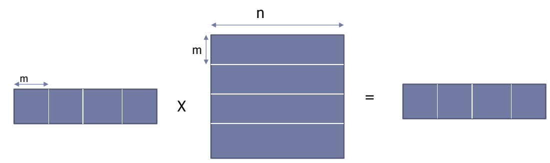
```
/* perform local sum first */
sum = 0;
for (i=0; i < m; i ++){  sum = sum + a[i] * b[i]; }

/* Use MPI_Reduce to perform global sum */
MPI_Reduce(sum, c, 1, MPI_INT, MPI_SUM, 0, comm);
```

A note about the above code template:
The final result appears in variable c of process 0.

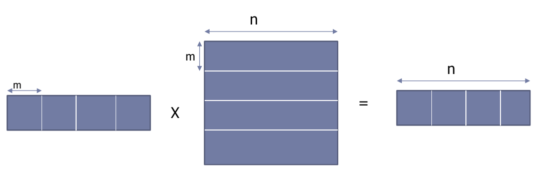
Exercise

- A routine that computes the product of a vector and an array that are distributed across a group of processes.



Sample code

```
/* perform local sum first */
for (j = 0;  j < n; j++){
    sum[j] = 0;
    for (i=0; i < m; i ++){  sum = sum + a[i] * b[i,j]; }
}
MPI_Reduce(sum, c, n, MPI_INT, MPI_SUM, 0, comm);
```



MPI_Allreduce

- MPI_Allreduce(sendbuf, recvbuf, count, datatype,
- op, comm)
- Same as MPI_Reduce, except
  - The result appears in receive buffer of all processes.

data →

PROCESSES ↓

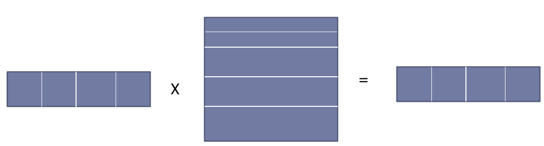
A0	B0	C0
A1	B1	C1
A2	B2	C2

⇒

A0+A1+ A2	B0 + B1 + B2	C0 + C1 + C2
A0+A1+ A2	B0+B1+ B2	C0 + C1 + C2
A0 + A1 + A2	B0 + B1 + B2	C0 + C1 + C2

Exercise

- A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer in all nodes.

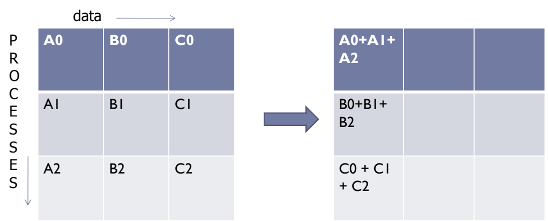


Code template

```
for (j=0; j < N; j++){
    tmp = 0;
    for (i=0; I < M; i++){ tmp = tmp + a[i] * b[j][i]; }
    sum[j] = tmp;
}

MPI_Allreduce(sum, c, N, MPI_INT, MPI_SUM).
```

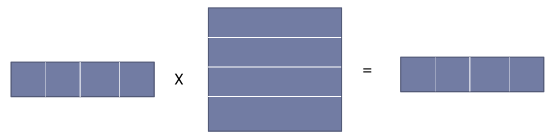
Reduce-Scatter



Reduce-Scatter

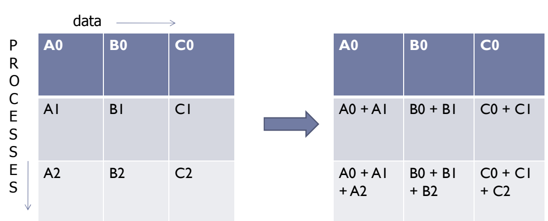
MPI_Reduce_Scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm)

recvcounts is an array of integers.



Use MPI_Reduce_Scatter to compute the product of a vector with an array. All of the vectors and arrays are distributed across processes, as shown (the local slices are shown).

Scan



MPI_Scan

- MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)
  - Count is the number of elements in input buffer.
- Returns in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,1, …,i
- Type of operations (op) as in MPI_Reduce
  - MPI_SUM, …

So far

- MPI_Bcast
- MPI_Gather
  - MPI_Gatherv
- MPI_Scatter
  - MPI_Scatterv
- MPI_Reduce
  - A very general operation with variants
    - MPI_Allreduce
    - MPI_Reduce_Scatter
- MPI_Scan

Possible errors in programming

- switch(rank){
  - case 0:
    - MPI_Bcast(buf1, count, type, 0, comm);
    - MPI_Bcast(buf2, count, type, 1, comm);
    - break;
  - case 1:
    - MPI_Bcast(buf2 count, type, 1, comm);
    - MPI_Bcast(buf1, count, type, 0, comm);
    - Break;
- }

Explanation

- Group of comm. here is {0,1}
- Two processes execute broadcasts in reverse order.
- MPI matches the first calls
  - Error, since root processes do not match.
- *Collective operations must be executed in the same order at all members of the communication group.*
- What if broadcast is a synchronizing operation?
  - Even if the broadcast calls are properly matched by MPI, a deadlock can occur.

Possible errors in programming

- switch(rank) {
  - case 0:
    - MPI_Bcast(buf1, count, type, 0, comm0);
    - MPI_Bcast(buf2, count, type, 2, comm2); break;
  - case 1:
    - MPI_Bcast(buf1, count, type, 1, comm1);
    - MPI_Bcast(buf2, count, type, 0, comm0); break;
  - case 2:
    - MPI_Bcast(buf1, count, type, 2, comm2);
    - MPI_Bcast(buf2, count, type, 1, comm1); break;
- }
- Assume comm0={0,1}, comm1={1,2}, comm2 = {2,0}
- *Collective operations must be executed in an order so that no cyclic dependencies exist – avoid deadlocks!*

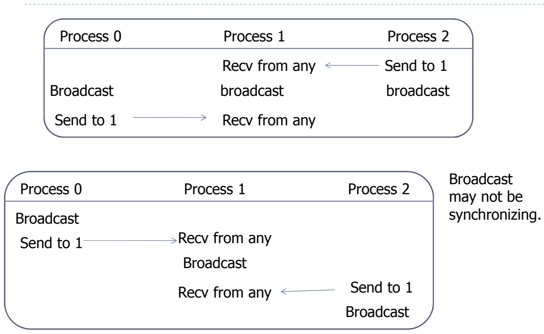
Possible errors in programming

- switch(rank){
  - case 0:
    - MPI_Bcast(buf1, count, type, 0, comm);
    - MPI_Send(buf2, count, type, 1, tag, comm); break;
  - case 1:
    - MPI_Recv(buf2, count, type, 0, tag, comm, &status);
    - MPI_Bcast(buf1, count, type, 0, comm); break;
- }
- What is the error in this one?
  - *The relative order of execution of collective operations and point-to-point operations should be such that, even if the collective operations and point-point operations are synchronizing – no deadlock will occur.*

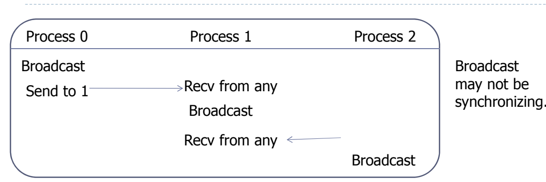
Possible ambiguity in programming

- switch(rank){
  - case 0:
    - MPI_Bcast(buf1, count, type, 0, comm);
    - MPI_Send(buf2,count,type,1,tag,comm);
    - break;
  - case 1:
    - MPI_Recv(buf2,count,type,MPI_ANY_SOURCE,tag,comm,&status);
    - MPI_Bcast(buf1,count, type, 0, comm);
    - MPI_Recv(buf2, count,type, MPI_ANY_SOURCE,tag,comm,&status);
    - break;
  - case 2:
    - MPI_Send(buf2, count, type, 1, tag, comm);
    - MPI_Bcast(buf1, count, type, 0, comm); comm = {0,1,2}
    - break;
- }

Possible Executions



Problematic execution



What is wrong with this execution?

A send by process 0 executed after the broadcast, is received at process 1 before the broadcast!

To disallow this execution, sources of receives should be stated clearly.

Removal of ambiguity

- switch(rank){
  - case 0:
    - MPI_Bcast(buf1, count, type, 0, comm);
    - MPI_Send(buf2,count,type,1,tag,comm);
    - break;
  - case 1:
    - MPI_Recv(buf2,count,type, **2,**tag,comm,&status);
    - MPI_Bcast(buf1,count, type, 0, comm);
    - MPI_Recv(buf2, count,type, **0** ,tag,comm,&status);
    - break;
  - case 2:
    - MPI_Send(buf2, count, type, 1, tag, comm);
    - MPI_Bcast(buf1, count, type, 0, comm);
    - break;
- }