Lecture 3 Message-Passing Programming Using MPI (Part 2)

Non-blocking Communication

- Advantages:

  -- allows the separation between the initialization of the communication and the completion.

  -- can avoid deadlock

  -- can reduce latency by posting receive calls early

- Disadvantages:

  -- complex to develop, maintain and debug code

Non-block Send/Recv Syntax

- ```
  int MPI_Isend(void* message /* in */,
      int             count /* in */,
      MPI_Datatype    datatype /* in */,
      int             dest /* in */,
      int             tag /* in */,
      MPI_Comm        comm /* in */,
      MPI_Request*    request /* out */)
  ```

- ```
  int MPI_Irecv(void* message /* out */,
      int             count /* in */,
      MPI_Datatype    datatype /* in */,
      int             source /* in */,
      int             tag /* in */,
      MPI_Comm        comm /* in */,
      MPI_Request*    request /* out */)
  ```

Non-blocking Send/Recv Details

- Non-blocking operation requires a minimum of two function calls: a call to start the operation and a call to complete the operation.
- The “request” is used to query the status of the communicator or to wait for its completion.
- The user **must NOT** overwrite the send buffer until the send (data transfer) is complete.
- The user **can NOT** use the receiving buffer before the receive is complete.

Non-blocking Send/Recv Communication Completion

- Completion of a non-blocking send operation means that the sender is now free to update the send buffer “message”.
- Completion of a non-blocking receive operation means that the receive buffer “message” contains the received data.

- int MPI_Wait(MPI_Request* request /* in-out */,
  MPI_Status* status /* out */)

- int MPI_Test(MPI_Request* request /* out */,
  int* flag /* out*/,
  MPI_Status* status /* out */)

Details of Wait/Test

- "request" is used to identify a previously posted send/receive
- MPI_Wait() returns when the operation is complete, and the status is updated for a receive.
- MPI_Test() returns immediately, with "flag" = true if posted operation corresponding to the "request" handle is complete.

Non-blocking Send/Recv Example

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char** argv)
{      /*** sample_nonblock2.c   ***/

      int my_rank, nprocs, recv_count;
      MPI_Request  request;
      MPI_Status status;
      double s_buf[100], r_buf[100];

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

      if (my_rank==0){
           MPI_Irecv(r_buf, 100, MPI_DOUBLE, 1, 22, MPI_COMM_WORLD, &request);
           MPI_Send(s_buf, 100, MPI_DOUBLE, 1, 10, MPI_COMM_WORLD);
           MPI_Wait(&request, &status);
      }
      else if(my_rank == 1){
           MPI_Irecv(r_buf, 100, MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, &request);
           MPI_Send(s_buf, 100, MPI_DOUBLE, 0, 22, MPI_COMM_WORLD);
           MPI_Wait(&request, &status);
      }
      MPI_Get_count(&status, MPI_DOUBLE, &recv_count);
      printf("proc %d, source %d, tag %d, count %d\n", my_rank,
            status.MPI_SOURCE, status.MPI_TAG, recv_count);
      MPI_Finalize();
}
```

Use MPI_Isend (not Safe to Change the Buffer)

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char** argv)
{     /** sample_unsafe_isend.c **/

      int my_rank, nprocs, recv_count;
      MPI_Request  request;
      MPI_Status status;
      double s_buf[100], r_buf[100];

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

      if (my_rank==0){
           MPI_Isend(s_buf, 100, MPI_DOUBLE, 1, 10, MPI_COMM_WORLD, &request);
           MPI_Recv(r_buf, 100, MPI_DOUBLE, 1, 22, MPI_COMM_WORLD, &status);
           MPI_Wait(&request, &status);
      }
      else if(my_rank == 1){
           MPI_Isend(s_buf, 100, MPI_DOUBLE, 0, 22, MPI_COMM_WORLD, &request);
           MPI_Recv(r_buf, 100, MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, &status);
           MPI_Wait(&request, &status);
      }
      MPI_Get_count(&status, MPI_DOUBLE, &recv_count);
      printf("proc %d, source %d, tag %d, count %d\n", my_rank,
            status.MPI_SOURCE, status.MPI_TAG, recv_count);
      MPI_Finalize();
}
```

More about Communication Modes

Send Modes	MPI function	Completion Condition
Synchronous send	MPI_Ssend() MPI_Issend()	A send will not complete until a matching receive has been posted and the matching receive has begun reception of the data. Completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution
Buffered send (It has additional associated functions. The send operation is local.)	MPI_Bsend() MPI_Ibsend()	Bsend() always completes (unless an error occurs) Completion is irrespective of the receiver.
**Standard send	MPI_Send() MPI_Isend()	message sent (no guarantee that the receive has started). It is up to MPI to decide what to do.
Ready send	MPI_Rsend() MPI_Irsend()	may be used only when the a matching receive has already been posted

http://www.mpi-forum.org/docs/mpi-11-html/node44.html#Node44

http://www.mpi-forum.org/docs/mpi-11-html/node40.html#Node40

- MPI_Ssend()

  -- synchronization of source and destination

  -- the behavior is predictable and safe

  -- recommend for debugging purpose
- MPI_Bsend()

  -- only do copy message to buffer

  -- completes immediately

  -- predictable behavior and no synchronization

  -- user must allocate extra buffer space by MPI_Buffer_attach()
- MPI_Rsend()

  -- completes immediately

  -- will succeed only if a matching receive is already posted

  -- if receiving process is not ready, action is undefined.

  -- may improve performance

“**Recommendations**: In general, use MPI_Send. If non-blocking routines are necessary, then try to use MPI_Isend or MPI_Irecv. Use MPI_Bsend only when it is too inconvenient to use MPI_Isend. The remaining routines, MPI_Rsend, MPI_Issend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.” --- http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

- See also ping_pong.c

Buffered Mode

- Standard Mode – If buffer is provided, amount of buffering is not defined by MPI
- Buffered Mode - Send may start and return before a matching receive. Necessary to specify buffer space via routine MPI_Buffer_attach().

```
int MPI_Buffer_attach(void *buffer, int size)
int MPI_Buffer_detach(void *buffer, int *size)
```

- The buffer size given should be the sum of the sizes of all outstanding MPI_Bsends, plus MPI_BSEND_OVERHEAD for each MPI_Bsend that will be done.

- MPI_Buffer_detach() returns the buffer address and size so that nested libraries can replace and restore the buffer.
- See sample_Bsend.c

MPI collective Communications

- Routines that allow groups of processes to communicate.
- Classification by Operation:
  - One-To-All Mode
    - One process contributes to the results. All processes receive the result.
    - MPI_Bcast()
    - MPI_Scatter(), MPI_Scatterv()
  - All-To-One Mode
    - All processes contribute to the result. One process receive the result.
    - MPI_Gather(), MPI_Gatherv()
    - MPI_Reduce()
  - All-To-All Mode
    - All processes contribute to the result. All processes receive the result.
    - MPI_Alltoall(), MPI_Alltoallv()
    - MPI_Allgather(), MPI_Allgatherv()
    - MPI_Allreduce(), MPI_Reduce_scatter()
  - Other
    - Collective operations that do not fit into above categories
    - MPI_Scan()
    - MPI_Barrier()

Barrier Synchronization

MPI_Barrier(MPI_Comm comm)

- This routine provides the ability to block the calling process until all processes in the communicator have reached this routine.

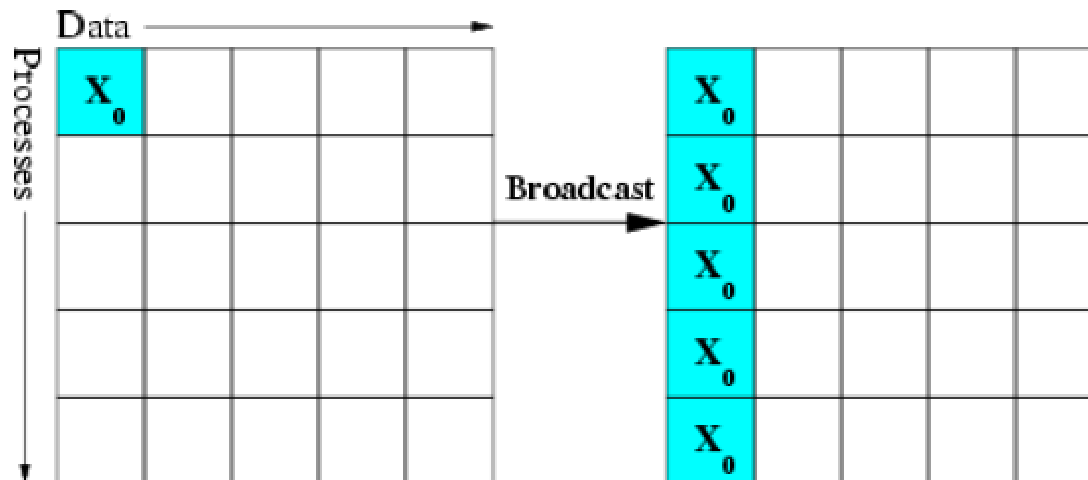
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
   int rank, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world.  I am %d of %d\n", rank, procs);
   fflush(stdout);

   MPI_Finalize();
   return 0;
}
```

Broadcast (One-To-All)

MPI_Bcast(void *buffer /* in/out */, int count /* in */,

MPI_Datatype datatype /* in */, int root /* in */, MPI_Comm comm)

- Broadcasts a message from the process with rank "root" to all other processes of the communicator.
- All members of the communicator use the same argument for “comm”, “root”.
- On return, the content of root’s buffer has been copied to all processes.



Tags and Synchronization

Time	Root (x=5, y = 10)	Process B	Process C
1	MPI_Bcast &x	Local work	Local work
2	MPI_Bcast &y	Local work	Local work
3	Local work	MPI_Bcast &y	MPI_Bcast &x
4	Local work	MPI_Bcast &x	MPI_Bcast &y

On Process B: x = 10, y = 5
On Process C: x = 5,   y = 10

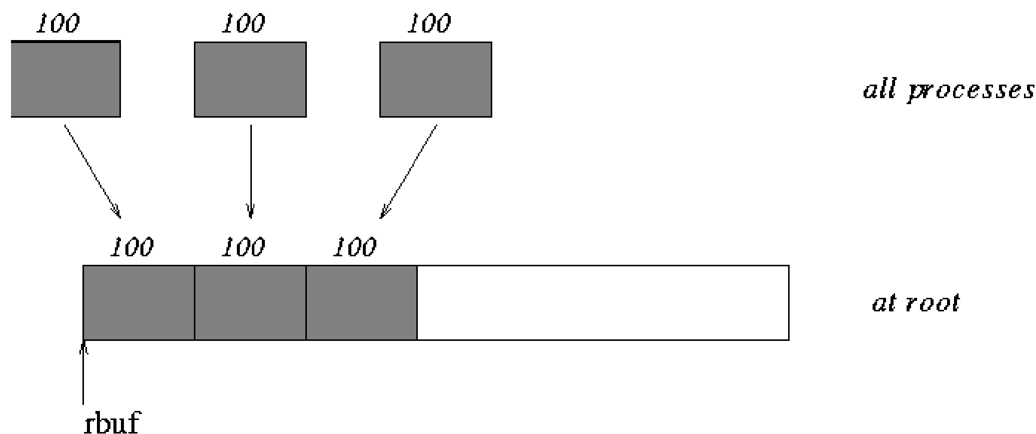
1. There is **no tag** in collective communication.
2. Normally, broadcast (and all other collective communication calls) are points of **synchronization**: on a given process the broadcast would not return until every process had received the broadcast data.
3. On current system, restriction on synchronization has been relaxed. It’s OK for root to complete two broadcast before other processes begin their calls. However, in terms of data communicated, the **effect** must be the same as if the processes synchronized.
4. Corresponding with 3, the system is assumed to providing buffering. In MPI parlance, it is unsafe.

Gather (All-To-One)

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

MPI_Gather collects the data from each process in the same communicator and store the data in process rank order on the process with rank *root*.

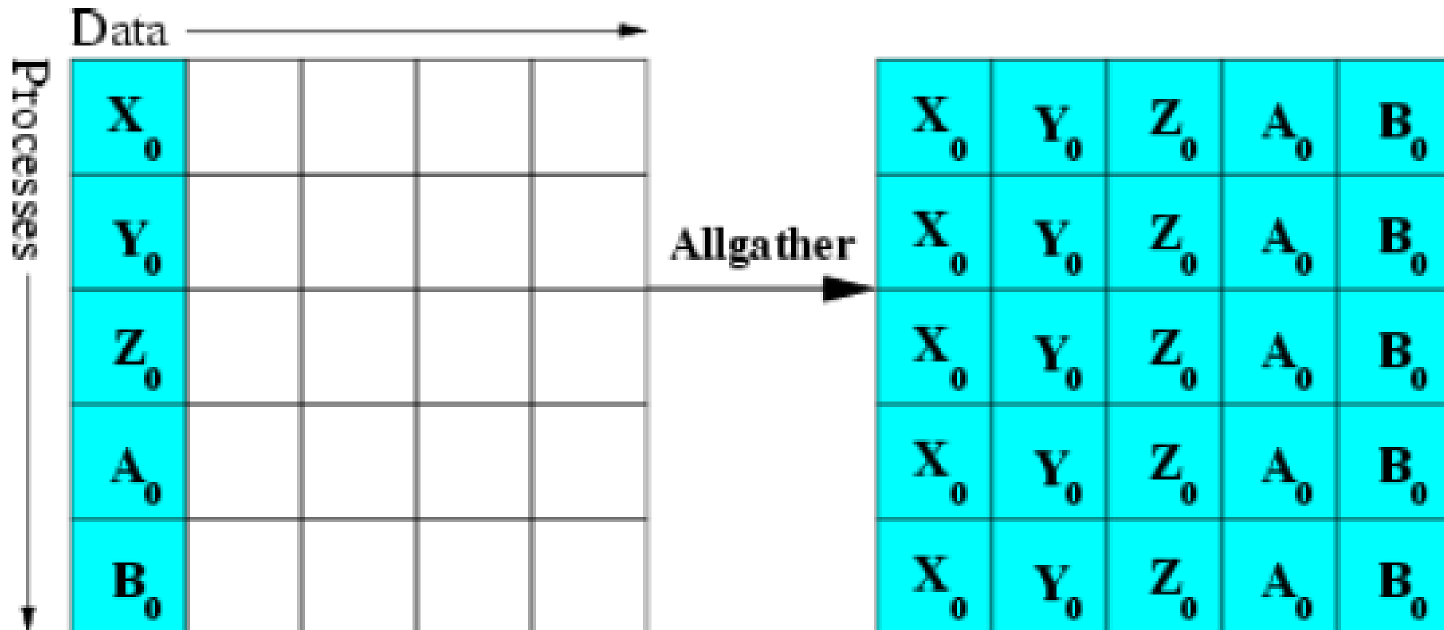
- Each process sends contents in “sendbuf” to “root”.
- Root stores received contents in rank order
- “recvbuf” is the address of receive buffer, which is significant only at “root”.
- “recvcnt” is the number of elements for any single receive, which is significant only at “root”.



AllGather (All-To-All)

int MPI_Allgather(void *sendbuf /* in */, int sendcount /* in */, MPI_Datatype sendtype /* in */, void *recvbuf /* out */, int recvcount /* in */, MPI_Datatype recvtype /* in */, MPI_Comm comm /* in */)

- Gather data from all tasks and distribute the combined data to all tasks
- recvcount: number of elements received from any process (integer)
- Similar to Gather + Bcast

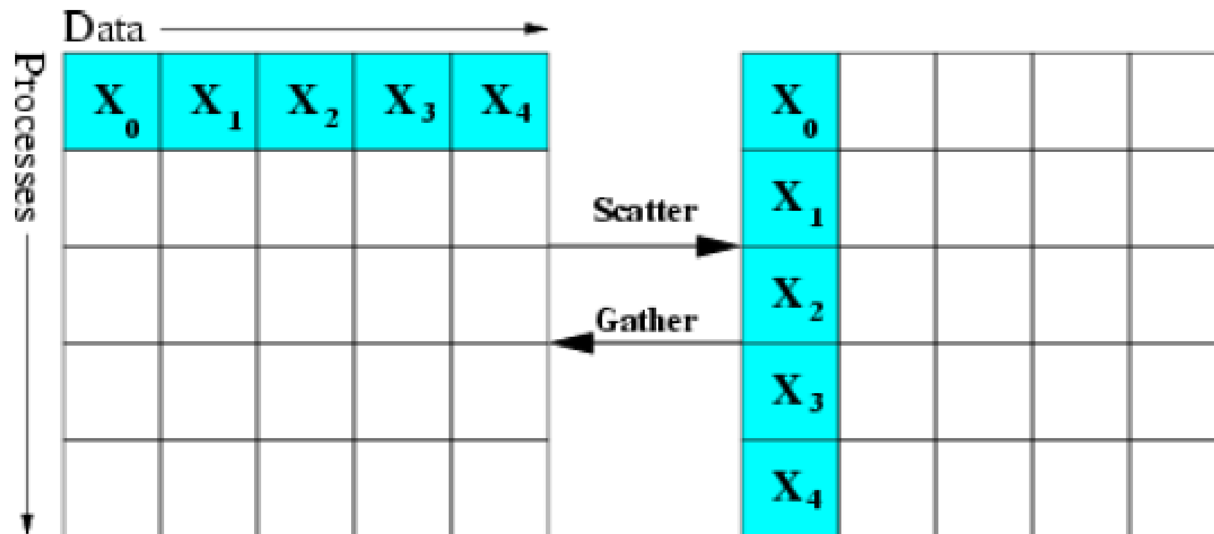


Scatter (One-To-All)

int MPI_Scatter(void *sendbuf /* in */, int sendcnt /* in */, MPI_Datatype sendtype /* in */, void *recvbuf /* out */, int recvcnt /* in */, MPI_Datatype recvtype /* in */, int root /* in */, MPI_Comm comm /* in */);

- Send data from one process “root” to all other processes in “comm”.
- It is the reverse operation of MPI_Gather
- It is a One-To-All operation which each recipient get a different chunk.
- “sendbuf”, “sendcnt” and “sendtype” are significant only at “root”.

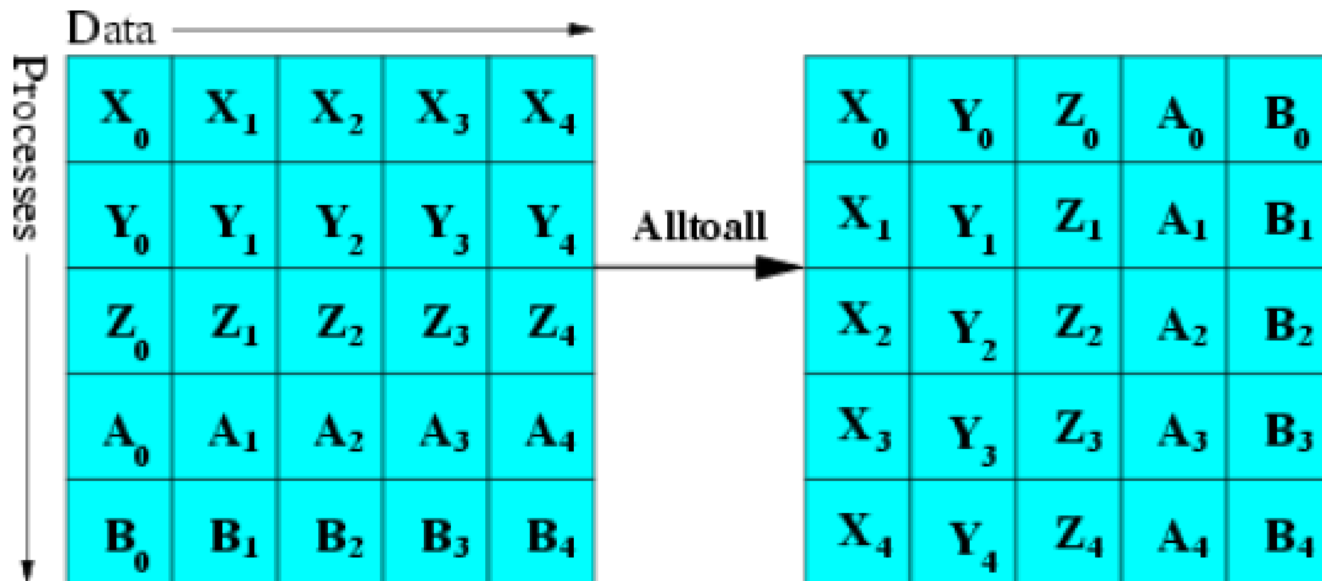
MPI_Scatter splits the data referenced by *sendbuf* on the process with rank *root* into *p* segments, each of which consists of *sendcnt* elements of type *sendtype.* The first segment is sent to process 0, the second to process 1, etc.



Alltoall (All-To-All)

int MPI_Alltoall(void *sendbuf /* in */, int sendcount /* in */, MPI_Datatype sendtype /* in */, void *recvbuf /* out */, int recvcount /* in */, MPI_Datatype recvtype /* in */, MPI_Comm comm /* in */)

- an extension of MPI_ALLGATHER to case where each process sends distinct data to each of the receivers.
- the *j*th block from process *i* is received by process *j* and is placed in the *i*th block of recvbuf.
- The type signature associated with *sendcount*, *sendtype* at a process must be equal to the type structure associated with *recvcount*, *recvtype* at any other process.



Reduction (All-To-One)

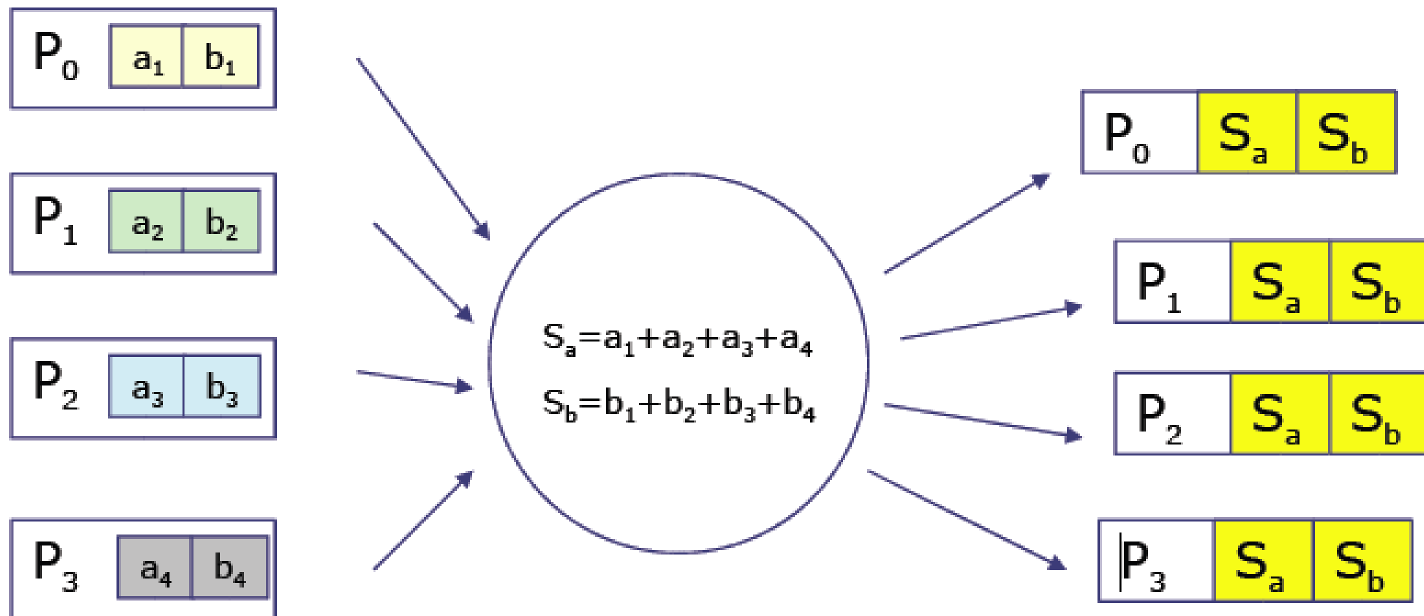
int MPI_Reduce(void *sendbuf /* in */, void *recvbuf /* out */, int count /* in */, MPI_Datatype datatype /* in */, MPI_Op op /* in */, int root /* in */, MPI_Comm comm /* in */)

- This routine combines values in “sendbuf” on all processes to a single value using the specified operation “op”.
- The combined value is put in “recvbuf” of the process with rank “root”.
- The routine is called by all group members using the same arguments for *count, datatype, op, root* and *comm*.

Predefined Reduction Operations

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MINLOC	min value and location
MPI_MAXLOC	max value and location

- Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-by-element on each entry of the sequence.



Benchmarking Parallel Performance

double MPI_Wtime(void)

- Return an elapsed time in seconds on the calling processor
- There is no requirement that different nodes return “the same time”.

```
#include "mpi.h"
#include <time.h>
#include <stdio.h>
/*measure_time.c*/
int main( int argc, char *argv[] )
{
   double t1, t2;

   MPI_Init( argc, argv);
   t1 = MPI_Wtime();
   sleep(1);
   t2 = MPI_Wtime();
   printf("MPI_Wtime measured a 1 second sleep to be: %1.2f\n", t2-t1);
   fflush(stdout);
   MPI_Finalize( );
   return 0;
}
```

Numerical Integration

- Composite Trapezoidal Rule

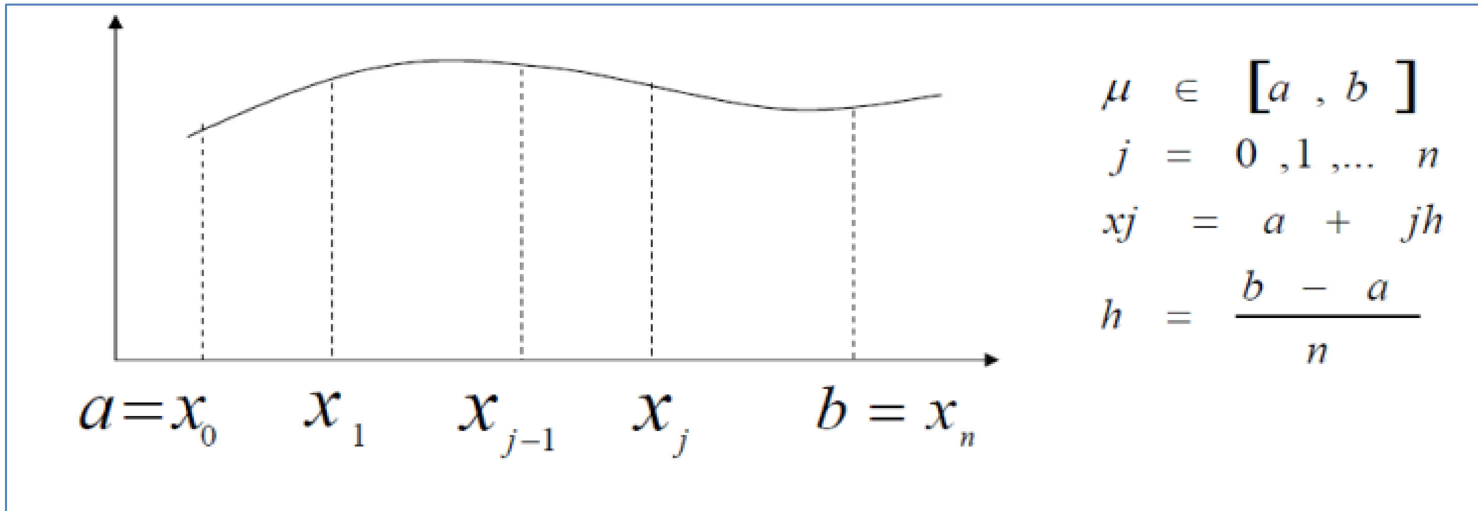


Figure 1 Composite Trapezoidal Rule

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right]$$

- Parallel Trapezoidal Rule

Input: number of processes *p*, entire interval of integration [*a*, *b*], number of subintervals *n*, *f(x)*

Assume *n/p is* integer