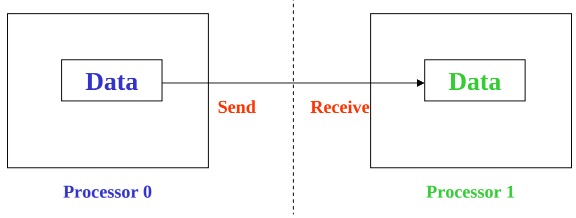
Lecture 8 – MPI (Continued)
Send and Receive



- Cooperative data transfer
- To (from) whom is data sent (received)?
- What is sent?
- How does the receiver identify it?

Message Passing: Send

MPI_Send(address, count, datatype, dest, tag, comm)

- (address, count) = a contiguous area in memory containing the message to be sent
- datatype = Type of data, e.g. integer, double precision (note that MPI had standard datatypes)
- dest = integer identifier representing the processor to send the message to
- tag = non-negative integer that the destination can use to selectively screen messages
- comm = communicator = group of processors

Message Passing: Receive

MPI_Recv(address, count, datatype, source, tag, comm, status)

- (address, count) = a contiguous area in message reserved for the message to be received
- datatype = Type of data, e.g. integer, double precision (note that MPI had standard datatypes)
- source = integer identifier representing the processor that sent the message
- tag = non-negative integer that the destination can use to selectively screen messages
- comm = communicator = group of processors
- status = information about the message that is received

Single Program Multiple Data (SPMD)

- Proc 0 and Proc 1 are actually performing different operations
- However, not necessary to write separate programs for each processor
- Typically, use conditional statement and proc id to define the job of each processor:

```
integer :: a(10)

if(my_id == 0) then
   MPI_Send(a,10,MPI_INT,1,0,MPI_COMM_WORLD)
else if(my_id == 1) then
   MPI_Recv(a,10,MPI_INT,0,0,MPI_COMM_WORLD)
end if
```

Different Types of Sends and Receives

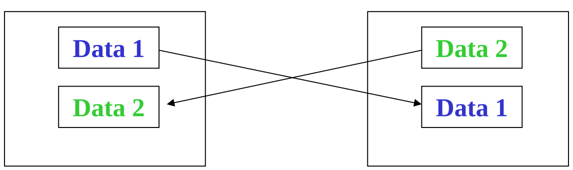
- **There are 4 types of Sends and Receives:**
  - Standard (blocking)
  - Synchronous
  - Buffered
  - Ready
- **Each of these types can be performed in:**
  - Blocking mode
  - Non-blocking mode
- **The code programmer must make the choice of which type and mode to use depending on the circumstance**

Send/Receive Types

- **Standard: similar to Blocking except receive will not allow processor to continue only until its buffer can be reused.**
- **Blocking: receive will not allow processor to continue until it has received its message. Receive acts as a Barrier to that processor.**
- **Synchronous: send (or receive) does not start until a matching receive (or send) is posted indicating it is ready. Send acts as "blocking" until matching receive occurs. In this case, send acts as a Barrier for those processors.**
- **Buffered: either a system or user-defined buffer is made available for send/receive so that communication can proceed.**

Deadlock

- **Example: exchange data between 2 procs:**



Processor 0	Processor 1
MPI_Send	MPI_Send
MPI_Recv	MPI_Recv

- **MPI_Send is a synchronous operation. If no system buffering is used, it keeps waiting until a matching receive is posted.**

Deadlock

- **Both processors are waiting for each other →deadlock**
- **However, OK if system buffering exists →unsafe programming, however**
- **Note: MPI_Recv is blocking and non-buffered**
- **Another real deadlock:**

Proc 0	Proc 1
MPI_Recv	MPI_Recv
MPI_Send	MPI_Send

- **Fix by reordering communication**

Proc 0	Proc 1
MPI_Send	MPI_Recv
MPI_Recv	MPI_Send

Buffered/Nonbuffered Communications

- **No-buffering (phone calls)**
  - Proc 0 initiates the send request and rings Proc 1. It waits until Proc 1 is ready to receive. The transmission starts.
  - Synchronous communication – completed only when the message was received by the receiving proc
- **Buffering (beeper)**
  - The message to be sent (by Proc 0) is copied to a system-controlled block of memory (buffer)
  - Proc 0 can continue executing the rest of its program
  - When Proc 1 is ready to receive the message, the system copies the buffered message to Proc 1
  - Asynchronous communication – may be completed even though the receiving proc has not received the message

Buffered Communication

- **Buffering requires system resources, e.g. memory, and can be slower if the receiving proc is ready at the time of requesting the send**
- **Application buffer: address space that holds the data in the user's computer program**
- **System buffer: system space for storing messages. In buffered communication, data in application buffer is copied to/from system buffer**
- **MPI allows communication in buffered mode:**
  MPI_Bsend, MPI_Ibsend
- **User allocates the buffer by:**
  MPI_Buffer_attach(buffer, buffer_size)
- **Free the buffer by MPI_Buffer_detach**
- **An alternate to MPI_Buffer commands is to allocate memory for buffer with standard allocate statement**

Blocking / Non-blocking Communication

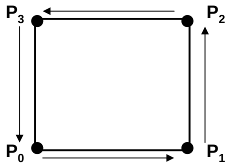
- **Blocking Communication (McDonald's)**
  - The receiving proc has to wait if the message is not ready and has not received initial signal from sending proc
  - Different from synchronous communication (where sending proc will not begin sending until it has received explicit permission from receiving proc)
  - Proc 0 may have already buffered the message to system and Proc 1 is ready, but the interconnection network is busy
- **Non-blocking Communication (In & Out)**
  - Proc 1 checks with the system if the message has arrived yet. If not, it continues doing other stuff. Otherwise, get the message from the system.
- **Useful when computation and communication can be performed at the same time**
- **MPI allows both non-blocking send and receive**

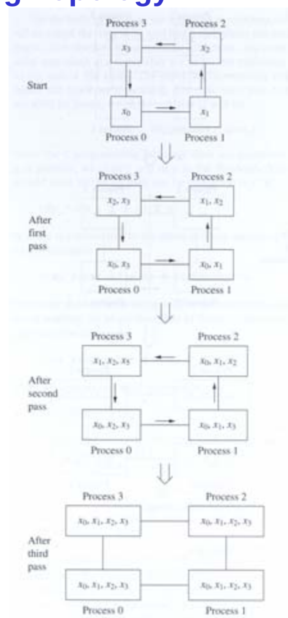
MPI_Isend and MPI_Irecv

- **In non-blocking send, program identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for message to be copied out from the application buffer**
- **The user's program *should not* modify the application buffer until the non-blocking send has completed**
- **Non-blocking communication can be combined with non-buffering: MPI_Issend, or buffering: MPI_Ibsend**
- **Use MPI_Wait or MPI_Test to determine if the non-blocking send or receive has completed**

Example: Data Exchange in a Ring Topology

Analogous to
MPI_Allgather



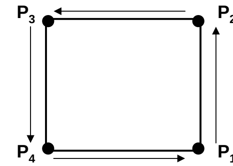


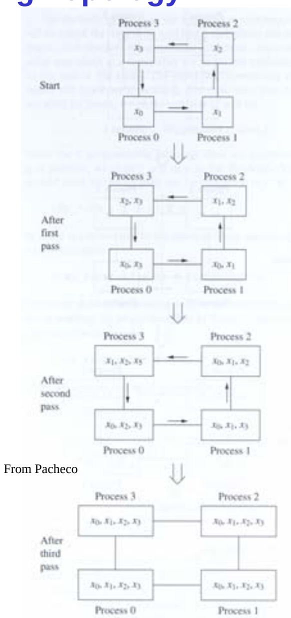
- **Blocking version:**

```
for (i=0; i<p; i++) {
    send_offset = ((my_id-i+p)%p)*blksize;
    recv_offset  = ((my_id-i-1+p)%p)*blksize;
    MPI_Send(y+send_offset,blksize,MPI_FLOAT,
                my_id+1,0,ring_com);
    MPI_Recv(y+recv_offset,blksize,MPI_FLOAT,
                my_id-1,0,ring_com,&status);
}
```

From Pacheco

Example: Data Exchange in a Ring Topology





From Pacheco

- **Non-Blocking version:**

```
send_offset = my_id*blksize;
recv_offset  = (my_id-1+p)*blksize;
for (i=0; i<p; i++) {
    MPI_Isend(y+send_offset,blksize,MPI_FLOAT,
                 my_id+1,0,ring_com,&send_request);
    MPI_Irecv(y+recv_offset,blksize,MPI_FLOAT,
                 my_id-1,0,ring_com,&recv_request);
    send_offset = ((my_id-i-1+p)%p)*blksize;
    recv_offset  = ((my_id-i-2+p)%p)*blksize;
    MPI_Wait(&send_request,&status);
    MPI_Wait(&recv_request,&status);
}
```

- **The communication and computations of next offsets are overlapped.**

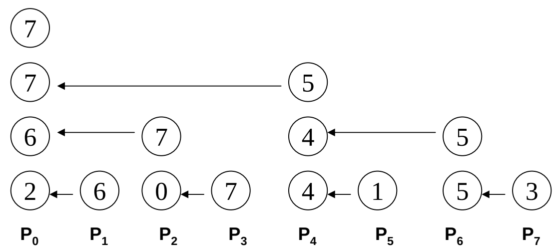
Summary of Communication Modes

- **4 communication modes in MPI: standard (blocking), buffered, synchronous, ready. They can be either blocking or non-blocking**
- **In standard (blocking) modes (MPI_Send, MPI_Recv,…), it is up to the system to decide whether messages should be buffered. Note there is a limited, finite amount of memory for system buffers.**
- **In synchronous mode, a send will not complete until a matching receive has been posted which has begun reception of the data**
  - MPI_Ssend (blocking), MPI_ISsend (non-blocking)
  - No system buffering
- **In buffered mode, the completion of a send does not depend on the existence of a matching receive**
  - MPI_Bsend (blocking), MPI_IBsend (non-blocking)
  - System buffering by MPI_Buffer_attach and MPI_Buffer_detach
- **Ready mode not discussed**

Collective Communications

- **Communication pattern involving all the procs; usually more than 2**
- **MPI_Barrier: synchronize all processors**
- **Broadcast (MPI_Bcast)**
  - A single proc sends the same data to every other proc
- **Reduction (gather/add) (MPI_Reduce)**
  - All the procs contribute data that is combined using a binary operation
  - Example: max, min, sum, etc.
  - One proc obtains the final answer
- **Allreduce (MPI_Allreduce)**
  - Same as MPI_Reduce but every proc contains the final answer
  - Effectively as MPI_Reduce + MPI_Bcast, but more efficient

An Implementation of the “Max” Function

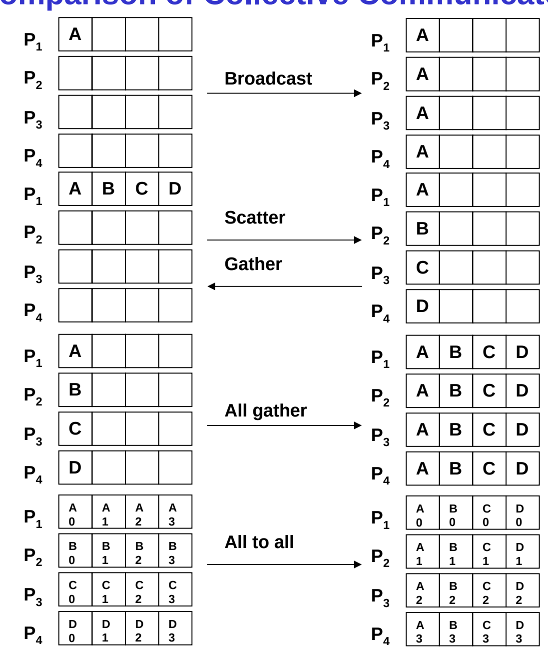


- **Tree-structured communication: (find the maximum among procs)**
- **Only needs $\log_2 p$ stages of communication**
- **Not necessarily optimum on a particular architecture**

Other Collective Communicators

- **Scatter (MPI_Scatter)**
  - Split the data on the root processor into p segments
  - The 1st segment is sent to proc 0, the 2nd to proc 1, etc.
  - Similar to but more general than MPI_Bcast
- **Gather (MPI_Gather)**
  - Collect the data from each processor and store the data on root processor
  - Similar to but more general than MPI_Reduce
- **Can collect and store the data on all procs using MPI_Allgather**

Comparison of Collective Communicators



Homework 3

- **Finish reading Chap. 1-3 of Using MPI by Gropp et al.**
- **Look at the parallel routine to compute π (calcpip) in the Codes directory (and the Examples directory on Wopr)**
- **Due Friday Oct. 23: Modify the calcpip.f routine to do a more accurate integration (Simpson’s Rule) described in the next slide**
  - Provide a listing of all subroutines and test this algorithm for different numbers of processors using parallel run.qsub batch submit procedure on wopr
  - Provide the CPU time as a function of the number of processors (up to 8) and the answer

Homework 3

. A more accurate alternative to the trapezoidal rule is Simpson's rule. The basic idea is to approximate the graph of $f(x)$ by arcs of parabolas rather than line segments. Suppose that $p < q$ are real numbers, and let r be the midpoint of the segment $[p, q]$. If we let $h = (q - p)/2$, then an equation for the parabola passing through the points $(p, f(p))$, $(r, f(r))$, and $(q, f(q))$ is

$$y = \frac{f(p)}{2h^2}(x-r)(x-q) - \frac{f(r)}{h^2}(x-p)(x-q) + \frac{f(q)}{2h^2}(x-p)(x-r).$$

If we integrate this from p to q, we get

$$\frac{h}{3}[f(p) + 4f(r) + f(q)].$$

Thus, if we use the same notation that we used in our discussion of the trapezoidal rule and we assume that n, the number of subintervals of $[a, b]$, is even, we can approximate

$$\int_a^b f(x)dx \doteq \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)].$$

Assuming that n/p is even, write

a. a serial program and
b. a parallel program that uses Simpson's rule to estimate $\int_a^b f(x)dx$.

From Parallel Programming with MPI by Pacheco