Course Outline

Parallel Programming
Introduction to MPI
Getting started with MPI
Communication modes
Collective communication????
What else is there?

Parallel Programming

Outline
› An example
› Programming models
› Issues to consider
  • Data distribution
  • Flow control

π=+∫
0
1
(1+1/x^4)
dx = π

We can approximate the integral as a sum of rectangles

\[ \sum_{j=0}^{N} F(x_j) \Delta x = \pi \]

Parallel Programming

Programming models
› Shared memory
  • All processes have access to global memory
› Distributed memory
  • Processes have access to only local memory.
  • Data is shared via explicit message passing
› Combination shared/distributed
  • Groups of processes share access to “local” data while data is shared between groups via explicit message passing
Parallel Programming

Issues to consider

- Data Distribution
  - Minimize overhead
    - Latency (message set up time)
    - Memory movement
  - Maximize load balance
    - Less idle time waiting for data or synchronizing
    - Each process should do about the same work
- Flow Control
  - Minimize waiting

Data Distribution

Suppose to calculate new values for each cell, we need the old values from each neighbor. What is the best way to distribute the array across, say, 9 processors?

Distributing the data like this would mean most processors need only send and receive 2 messages.

However, if the data is stored in Fortran order, the elements to be sent and received are not stored in contiguous locations in the memory.

For Fortran order matrices, this distribution means most processors need only send and receive 2 messages, and the data is stored contiguously in memory.

What if more work needs to be done on the left side of the matrix? Then the processors working on those regions would finish first and then have no more work to do.
This distribution may minimize message passing overhead and maximize the overall load balance.

Suppose each processor must receive data from its neighbor to the left (and consequently must send data to its neighbor to the right)

How do we control the flow of data?

This pattern causes the data flow to be serialized. Process n cannot receive data until process n-1 sends it, but this will not happen until after process n-1 has received its data, which will not happen until process n-2 sends its data, etc.

This pattern allows all the data to start flowing concurrently.

We will see later that this pattern is not safe as shown, but we will also see later an easy way to make this pattern safe.

What is MPI?

- MPI stands for “Message Passing Interface”
- In ancient times (late 1980’s early 1990’s) each vendor had its own message passing library
  - Non-portable code
  - Not enough people doing parallel computing due to lack of standards
What is MPI?

April 1992 was the beginning of the MPI forum
- Organized at SC92
- Consisted of hardware vendors, software vendors, academicians, and end users
- One of the main contributors: Dr. Jack Dongarra
- Held 2 day meetings every 6 weeks
- Created drafts of the MPI standard
- This standard was to include all the functionality believed to be needed to make the message passing model a success
- Final version released May, 1994

What is MPI?

A standard library specification!
- Defines syntax and semantics of an extended message passing model
- It is not a language or compiler specification
- It is not a specific implementation
- It does not give implementation specifics
  - Hints are offered, but implementers are free to do things however they want
  - Different implementations may do the same thing in a very different manner
- http://www.mpi-forum.org

What is MPI?

A library specification designed to support parallel computing in a distributed memory environment
- Routines for cooperative message passing
  - There is a sender and a receiver
  - Point-to-point communication
  - Collective communication
- Routines for synchronization
- Derived data types for non-contiguous data access patterns
- Ability to create sub-sets of processors
- Ability to create process topologies

Continuing to grow!
- New routines have been added to replace old routines
- New functionality has been added
  - Dynamic process management
  - One sided communication
  - Parallel I/O

Getting Started with MPI

Outline
- Introduction
- 6 basic functions
- Basic program structure
- Groups and communicators
- A very simple program
- Message passing
  - A simple message passing example
- Types of programs
  - Traditional
  - Master/Slave
  - Examples
- Unsafe communication

MPI contains 125 routines (more with the extensions)!
Many programs can be written with just 6 MPI routines!
Upon startup, all processes can be identified by their rank, which goes from 0 to N-1 where there are N processes
6 Basic Functions

MPI_INIT: Initialize MPI
MPI_Finalize: Finalize MPI
MPI_COMM_SIZE: How many processes are running?
MPI_COMM_RANK: What is my process number?
MPI_SEND: Send a message
MPI_RECV: Receive a message

MPI_INIT (ierr)

ierr: Integer error return value. 0 on success, non-zero on failure.
This MUST be the first MPI routine called in any program.
Can only be called once
Sets up the environment to enable message passing

MPI_FINALIZE (ierr)

ierr: Integer error return value. 0 on success, non-zero on failure.
This routine must be called by each process before it exits
This call cleans up all MPI state
No other MPI routines may be called after MPI_FINALIZE
All pending communication must be completed (locally) before a call to MPI_FINALIZE

Basic Program Structure

```
program main
  include 'mpi.h'
  integer ierr
  call MPI_INIT (ierr)
  ........
  Do some work
  ........
  call MPI_FINALIZE (ierr)
  Maybe do some additional
  Local computation

end
```

```
#include "mpi.h"

int main ()
{
  MPI_Init ()
  ........
  Do some work
  ........
  MPI_Finalize ()
  Maybe do some additional
  Local computation

}
```

Groups and communicators

We will not be using this, but it is important to understand so you understand other routines
Groups can be thought of as sets of processes
These groups are associated with what are called “communicators”
Upon startup, there is a single set of processes associated with the communicator
MPI_COMM_WORLD
Groups can be created which are sub-sets of this original group, also associated with communicators

MPI_COMM_RANK (comm, rank, ierr)

comm: Integer communicator. We will always use MPI_COMM_WORLD rank: Returned rank of calling process ierr: Integer error return code

This routine returns the relative rank of the calling process, within the group associated with comm.
**MPI_COMM_SIZE (comm, size, ierr)**

Comm: Integer communicator identifier
Size: Upon return, the number of processes in the group associated with comm. For our purposes, always the total number of processes

This routine returns the number of processes in the group associated with comm.

---

**A Very Simple Program**

**Hello World**

```fortran
program main
    include 'mpi.h'
    integer ierr, size, rank
    call MPI_INIT (ierr)
    call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)
    call MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierr)
    print *, 'Hello World from process', rank, 'of', size
    call MPI_FINALIZE (ierr)
end
```

- `mpirun –np 4 a.out`
  ```
  Hello World from 2 of 4
  Hello World from 0 of 4
  Hello World from 3 of 4
  Hello World from 1 of 4
  ```

---

**Message Passing**

Message passing is the transfer of data from one process to another
- This transfer requires cooperation of the sender and the receiver, but is initiated by the sender
- There must be a way to “describe” the data
- There must be a way to identify specific processes
- There must be a way to identify messages

---

**Message Passing**

Data is described by a triple
1. Address: Where is the data stored
2. Count: How many elements make up the message
3. Datatype: What is the type of the data
   - Basic types (integers, reals, etc)
   - Derived types (good for non-contiguous data access)

Processes are specified by a double
1. Communicator: We will always use MPI_COMM_WORLD
2. Rank: The relative rank of the specified process within the group associated with the communicator

Messages are identified by a single tag
This can be used to differentiate between different types of messages
**MPI_SEND** (buf, cnt, dtype, dest, tag, comm, ierr)

buf: The address of the beginning of the data to be sent
cnt: The number of elements to be sent
dtype: Datatype of each element
dest: The rank of the destination
tag: The message tag
comm: The communicator

**MPI_SEND**

Once this routine returns, the message has been copied out of the user buffer and the buffer can be reused
This may require the use of system buffers. If there are insufficient system buffers, this routine will block until a corresponding receive call has been posted
Completion of this routine indicates nothing about the designated receiver

---

**MPI_RECV** (buf, cnt, dtype, source, tag, comm, status, ierr)

buf: Starting address of receive buffer
cnt: Max number of elements to receive
dtype: Datatype of each element
source: Rank of sender (may use MPI_ANY_SOURCE)
tag: The message tag (may use MPI_ANY_TAG)
comm: Communicator
status: Status information on the received message

**MPI_RECV**

When this call returns, the data has been copied into the user buffer
Receiving fewer than cnt elements is ok, but receiving more is an error
Status is a structure in C (MPI_Status) and an array in Fortran (integer status(MPI_STATUS_SIZE))

---

**MPI_STATUS**

The status parameter is used to retrieve information about a completed receive
In C, status is a structure consisting of at least 3 fields:
MPI_SOURCE, MPI_TAG, MPI_ERROR
 status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code, respectively
In Fortran, status must be an integer array of size MPI_STATUS_SIZE
status(MPI_SOURCE), status(MPI_TAG), and status(MPI_ERROR) contain the source, tag, and error code

---

**Send/Recv Example**

```c
program main
    include 'mpi.h'
    CHARACTER*20 msg
    integer ierr, rank, tag, status(MPI_STATUS_SIZE)
    tag = 99
    call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    if (rank == 0) then
        msg = "Hello there"
call MPI_SEND(msg, 11, MPI_CHARACTER, 1, tag, MPI_COMM_WORLD, ierr)
    else if (rank == 1) then
        call MPI_RECV(msg, 20, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, status, ierr)
    endif
    call MPI_FINALIZE(ierr)
end
```
Types of MPI Programs

Traditional
- Break the problem up into about even sized parts and distribute across all processors
- What if problem is such that you cannot tell how much work must be done on each part?

Master/Slave
- Break the problem up into many more parts than there are processors
- Master sends work to slaves
- Parts may be all the same size or the size may vary

Traditional Example

Compute the sum of a large array of N integers

```plaintext
Comm = MPI_COMM_WORLD
Call MPI_COMM_RANK (comm, rank)
Call MPI_COMM_SIZE (comm, npes)
Stride = N/npes
Start = (stride * rank) + 1
Sum = 0
DO (I = start, start+stride)
  sum = sum + array(I)
ENDDO
If (rank .eq. 0) then
  DO (I = 1, npes-1)
    call MPI_RECV(tmp, 1, MPI_INTEGER, &
      I, 2, comm, status)
    sum = sum + tmp
  ENDDO
ELSE
  MPI_SEND (sum, 1, MPI_INTEGER, &
    0, 2, comm)
ENDIF
```

Master/Slave Example

Do some computation for each of N elements in a large array. The amount of computation depends on the value of each element.

Main
- Call MPI_INIT ()
- Comm = MPI_COMM_WORLD
- Call MPI_COMM_RANK (comm, rank)
- Call MPI_COMM_SIZE (comm, size)
- If (rank .eq. 0)
  call master ()
- Else
  call slave()
- Endif
- Call MPI_FINALIZE ()

Slave code

```plaintext
100  Call MPI_RECV (info, 2, MPI_INTEGER, 0, MPI_ANY_TAG, comm, status)
  if (status(MPI_TAG) .eq. 4)
    goto 200
  do (I = info(1), info(2))
    do_compute (array(I))
  enddo
  call MPI_SEND (tmp, 0, MPI_BYTE, 0, 3, comm)
  goto 100
200  continue
```

Not quite safe, since slaves will send requests that are never received by the master. This makes calling MPI_Finalize erroneous.
Unsafe Communication Patterns

- Process 0 and process 1 must exchange data
- Process 0 sends data to process 1 and then receives data from process 1
- Process 1 sends data to process 0 and then receives data from process 0
- If there is not enough system buffer space for either message, this will deadlock
- Any communication pattern that relies on system buffers is unsafe
- Any pattern that includes a cycle of blocking sends is unsafe

Communication Modes

Outline

- Standard mode
  - Blocking
  - Non-blocking
- Non-standard mode
  - Buffered
  - Synchronous
  - Ready
- Performance issues

Point-to-Point Communication Modes

Standard Mode:

- Blocking:
  - MPI_SEND (buf, count, datatype, dest, tag, comm)
  - MPI_RECV (buf, count, datatype, source, tag, comm, status)

  - Generally ONLY use if you cannot call earlier AND there is no other work that can be done!
  - Standard ONLY states that buffers can be used once calls return. It is implementation dependent on when blocking calls return.
  - Blocking sends MAY block until a matching receive is posted. This is not required behavior, but the standard does not prohibit this behavior either. Further, a blocking send may have to wait for system resources such as system managed message buffers.

  - Be VERY careful of deadlock when using blocking calls!

Non-blocking (immediate) sends/receives:

- MPI_ISEND (buf, count, datatype, dest, tag, comm, request)
- MPI_IRECV (buf, count, datatype, source, tag, comm, request)
- MPI_WAIT (request, status)
- MPI_TEST (request, flag, status)

  - Allows communication calls to be posted early, which may improve performance.
  - Overlap computation and communication
  - Latency tolerance
  - Less (or no) buffering

  - * MUST either complete these calls (with wait or test) or call MPI_REQUEST_FREE

MPI ISEND (buf, cnt, dtype, dest, tag, comm, request)

- Same syntax as MPI_SEND with the addition of a request handle
- Request is a handle (int in Fortran) used to check for completeness of the send
- This call returns immediately
- Data in buf may not be accessed until the user has completed the send operation
- The send is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI IRECV(buf, cnt, dtype, source, tag, comm, request)

- Same syntax as MPI_RECV except status is replaced with a request handle
- Request is a handle (int in Fortran) used to check for completeness of the recv
- This call returns immediately
- Data in buf may not be accessed until the user has completed the receive operation
- The receive is completed by a successful call to MPI_TEST or a call to MPI_WAIT
**MPI_WAIT (request, status)**

Request is the handle returned by the non-blocking send or receive call. Upon return, status holds source, tag, and error code information. This call does not return until the non-blocking call referenced by request has completed. Upon return, the request handle is freed. If request was returned by a call to MPI_Isend, return of this call indicates nothing about the destination process.

**MPI_TEST (request, flag, status)**

*Request* is a handle returned by a non-blocking send or receive call. Upon return, *flag* will have been set to true if the associated non-blocking call has completed. Otherwise, it is set to false. If *flag* returns true, the request handle is freed and *status* contains source, tag, and error code information. If *request* was returned by a call to MPI_Isend, return with *flag* set to true indicates nothing about the destination process.

**Non-blocking Communication**

1. Continue
   - If (err .lt. Delta) goto 200
   - Do some computation
   - Do (i = 0, npes)
     - If (i .ne. myrank)
       - Set up data to send
       - Call MPI_Send (data, cnt, dtype, &I, tag, comm, ierr)
     - Endif
   - Enddo
   - Do (i = 0, npes)
     - If (i .ne. myrank)
       - Set up data to recv
       - Call MPI_Recv (data, cnt, dtype, &I, tag, comm, status, ierr)
     - Endif
   - Enddo
   - Goto 100

Clearly unsafe

May run out of handles

**Point-to-Point Communication Modes (cont)**

Non-standard mode communication
- Only used by the sender! (MPI uses the push communication model)
- Buffered mode - A buffer must be provided by the application
- Synchronous mode - Completes only after a matching receive has been posted
- Ready mode - May only be called when a matching receive has already been posted

Safe, and pretty good
Point-to-Point Communication Modes: Buffered

MPI_BSEND (buf, count, datatype, dest, tag, comm)
MPI_IBSEND (buf, count, dtype, dest, tag, comm, req)
MPI_BUFFER_ATTACH (buf, size)
MPI_BUFFER_DETACH (buf, size)

Buffered sends do not rely on system buffers
The user supplies a buffer that **MUST** be large enough for all messages
User need not worry about calls blocking, waiting for system buffer space
The buffer is managed by MPI
The user **MUST** ensure there is no buffer overflow

Buffered Sends

```
define BUFFSIZE 1000
char *buff;
char b1[500], b2[500];
MPI_Buffer_attach (buff, BUFFSIZE);
MPI_Bsend(b1, 500, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
MPI_Bsend(b2, 500, MPI_CHAR, 2, 1, MPI_COMM_WORLD);
```

Safe

Seg violation

```
define BUFFSIZE 1000
char *buff;
char b1[500], b2[500];
buff = (char*) malloc(BUFFSIZE);
MPI_Buffer_attach(buff, BUFFSIZE);
MPI_Bsend(b1, 500, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
MPI_Bsend(b2, 500, MPI_CHAR, 2, 1, MPI_COMM_WORLD);
```

Buffer overflow

```
int size;
char *buff;
char b1[500], b2[500];
MPI_Pack_size (500, MPI_CHAR, MPI_COMM_WORLD, &size);
size += MPI_BSEND_OVERHEAD;
size = size * 2;
buff = (char*) malloc(size);
MPI_Buffer_attach(buff, size);
MPI_Bsend(b1, 500, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
MPI_Bsend(b2, 500, MPI_CHAR, 2, 1, MPI_COMM_WORLD);
MPI_Buffer_detach(&buff, &size);
```

Point-to-Point Communication Modes: Synchronous

MPI_SSEND (buf, count, datatype, dest, tag, comm)
MPI_ISSEND (buf, count, dtype, dest, tag, comm, req)

Can be started (called) at any time.
Does not complete until a matching receive has been posted and the receive operation has been started
* Does NOT mean the matching receive has completed
Can be used in place of sending and receiving acknowledgements
Can be more efficient when used appropriately buffering may be avoided

Point-to-Point Communication Modes: Ready Mode

MPI_RSEND (buf, count, datatype, dest, tag, comm)
MPI_IRSEND (buf, count, dtype, dest, tag, comm, req)

May ONLY be started (called) if a matching receive has already been posted.
If a matching receive has not been posted, the results are undefined
May be most efficient when appropriate
Removal of handshake operation
Should only be used with **extreme** caution

```
while (! done) {
    MPI_Recv (NULL, 0, MPI_INT, MPI_ANYSOURCE, 1, MPI_COMM_WORLD, &status);
    source = status.MPI_SOURCE;
    get_work (…..);
    MPI_Rsend (buff, count, datatype, source, 2, MPI_COMM_WORLD);
    if (no more work) done = TRUE;
}
```

SAFE

```
MASTER
while (! done) {
    MPI_Ssend (NULL, 0, MPI_INT, MASTER, 1, MPI_COMM_WORLD);
    MPI_Wait (&request, &status);
    …
}
```

UNSAFE

```
MASTER
while (! done) {
    MPI_Irecv (buff, count, datatype, MASTER, 2, MPI_COMM_WORLD, &request);
    MPI_Ssend (NULL, 0, MPI_INT, MASTER, 1, MPI_COMM_WORLD);
    MPI_Wait (&request, &status);
    …
}
```

Ready Mode

```
MASTER
while (! done) {
    MPI_Send (NULL, 0, MPI_INT, MASTER, 1, MPI_COMM_WORLD);
    MPI_Recv (buff, count, datatype, MASTER, 2, MPI_COMM_WORLD, &status);
    …
}
```

SAFE

```
MASTER
while (! done) {
    MPI_Irecv (buff, count, datatype, MASTER, 2, MPI_COMM_WORLD, &request);
    MPI_Ssend (NULL, 0, MPI_INT, MASTER, 1, MPI_COMM_WORLD);
    MPI_Wait (&request, &status);
    …
}
```

Point-to-Point Communication Modes: Performance Issues

> Non-blocking calls are almost always the way to go
  > Communication can be carried out during blocking system calls
  > Computation and communication can be overlapped if there is special purpose communication hardware
  > Less likely to have errors that lead to deadlock
  > Standard mode is usually sufficient - but buffered mode can offer advantages
  > Particularly if there are frequent, large messages being sent
  > If the user is unsure the system provides sufficient buffer space
  > Synchronous mode can be more efficient if acks are needed
  > Also tells the system that buffering is not required
Collective Communication

Outline
- Introduction
- Barriers
- Broadcasts
- Gather
- Scatter
- Allgather
- Alltoall
- Reduction
- Performance issues

Collective Communication

Amount of data sent must exactly match the amount of data received
Collective routines are collective across an entire communicator and must be called in the same order from all processors within the communicator
Collective routines are all blocking
This simply means buffers can be re-used upon return
Collective routines return as soon as the calling process’ participation is complete
Does not say anything about the other processors
Collective routines may or may not be synchronizing
No mixing of collective and point-to-point communication

Collective Communication

Barrier: MPI_BARRIER (comm)
Only collective routine which provides explicit synchronization
Returns at any processor only after all processes have entered the call

Collective Communication

Collective Communication Routines:
- Except broadcast, each routine has 2 variants:
  - Standard variant: All messages are the same size
  - Vector Variant: Each item is a vector of possibly varying length
- If there is a single origin or destination, it is referred to as the root
- Each routine (except broadcast) has distinct send and receive arguments
- Send and receive buffers must be disjoint
- Each can use MPI_IN_PLACE, which allows the user to specify that data contributed by the caller is already in its final location.

Collective Communication

Collective Communication Routines:
- Except broadcast, each routine has 2 variants:
  - Standard variant: All messages are the same size
  - Vector Variant: Each item is a vector of possibly varying length
- If there is a single origin or destination, it is referred to as the root
- Each routine (except broadcast) has distinct send and receive arguments
- Send and receive buffers must be disjoint
- Each can use MPI_IN_PLACE, which allows the user to specify that data contributed by the caller is already in its final location.

Collective Communication: Bcast

MPI_BCAST (buffer, count, datatype, root, comm)
Strictly in place
MPI-1 insists on using an intra-communicator
MPI-2 allows use of an inter-communicator
REMEMBER: A broadcast need not be synchronizing.
Returning from a broadcast tells you nothing about the status of the other processes involved in a broadcast.
Furthermore, though MPI does not require MPI_BCAST to be synchronizing, it neither prohibits synchronous behavior.

BCAST

If (myrank == root) {
    fp = fopen (filename, 'r');
    fscanf (fp, '%d', &iters);
    fclose (fp);
}

MPI_Bcast (&iters, 1, MPI_INT, root, MPI_COMM_WORLD);

else {
    MPI_Recv (&iters, 1, MPI_INT, root, tag, MPI_COMM_WORLD, &status);
}

THAT’S BETTER

OOPS!

If (myrank == root) {
    fp = fopen (filename, 'r');
    fscanf (fp, '%d', &iters);
    fclose (fp);
}

MPI_Bcast (&iters, 1, MPI_INT, root, MPI_COMM_WORLD);

cont
Collective Communication: Gather

MPI_GATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

- Receive arguments are only meaningful at the root
- Each processor must send the same amount of data
- Root can use MPI_IN_PLACE for sendbuf:
  - data is assumed to be in the correct place in the recvbuf

```
P1 = root
P2
P3
MPI_GATHER
```

Collective Communication: Gatherv

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

- Vector variant of MPI_GATHER
- Allows a varying amount of data from each proc
- Allows root to specify where data from each proc goes
- No portion of the receive buffer may be written more than once
- MPI_IN_PLACE may be used by root.

```
P1 = root
P2
P3
P4
MPI_GATHERV
```

Collective Communication: Scatter

MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

- Opposite of MPI_GATHER
- Send arguments only meaningful at root
- Root can use MPI_IN_PLACE for recvbuf

```
P1
P2
P3
P4
MPI_SCATTER
```

MPI_Gather

```
int tmp[20];
int res[320];
for (i = 0; i < 20; i++) {
    do some computation
    tmp[i] = some value;
}
MPI_Gather (tmp, 20, MPI_INT, res, 20, MPI_INT, 0, MPI_COMM_WORLD);
if (myrank == 0)
    write out results
for (i = 0; i < 20; i++) {
    do some computation
    if (myrank == 0)
        res[i] = some value
    else tmp[i] = some value
}
if (myrank == 0)
    MPI_Gather (MPI_IN_PLACE, 20, MPI_INT, res, 20, MPI_INT, 0, MPI_COMM_WORLD);
write out results
else
    MPI_Gather (tmp, 20, MPI_INT, res, 320, MPI_REAL,
        MPI_COMM_WORLD);
```

A OK
Collective Communication: Alltoall (scatter/gather)

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

Collective Communication: AlltoallV

MPI_ALLTOALLV (sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)
- Same as MPI_ALLTOALL, but the vector variant
  - Can specify how many blocks to send to each processor,
    location of blocks to send, how many blocks to receive from
    each processor, and where to place the received blocks

Collective Communication: AlltoallW

MPI_ALLTOALLW (sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)
- Same as MPI_ALLTOALLV, except different datatypes can be specified for data scattered as well as data gathered
  - Can specify how many blocks to send to each processor,
    location of blocks to send, how many blocks to receive from
    each processor, and where to place the received blocks
  - Displacements are now in terms of bytes rather than types

Collective Communication: Reduction

MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)
- recvbuf only meaningful on root
- Combines elements (on an element by element basis) in sendbuf according to op
- Results of the reduction are returned to root in recvbuf
- MPI_IN_PLACE can be used for sendbuf on root

REAL a(n), b(n,m), c(m)
REAL sum(m)
DO j=1,m
  sum(j) = 0.0
  DO i = 1,n
    sum(j) = sum(j) + a(i)*b(i,j)
  ENDDO
ENDDO
CALL MPI_REDUCE(sum, c, m, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
Collective Communication: Reduction (cont)

- MPI_ALLREDUCE (sendbuf, recvbuf, count, datatype, op, comm)
  - Same as MPI_REDUCE, except all processors get the result
- MPI_REDUCE_SCATTER (sendbuf, recv_buff, recvcounts, datatype, op, comm)
  - Acts like it does a reduce followed by a scatterv

MPI_REDUCE_SCATTER

Collective Communication: Prefix Reduction

MPI_SCAN (sendbuf, recvbuf, count, datatype, op, comm)
  - Performs an inclusive element-wise prefix reduction
MPI_EXSCAN (sendbuf, recvbuf, count, datatype, op, comm)
  - Performs an exclusive prefix reduction
  - Results are undefined at process 0

Collective Communication: Reduction - user defined ops

MPI_OP_CREATE (function, commute, op)
  - If commute is true, operation is assumed to be commutative
  - Function is a user defined function with 4 arguments
    - invvec: input vector
    - inoutvec: input and output value
    - len: number of elements
    - datatype: MPI_DATATYPE
  - Returns invvec[i] op inoutvec[i], i = 0..len-1
- MPI_OP_FREE (op)

Collective Communication: Performance Issues

Collective operations should have much better performance than simply sending messages directly
  - Broadcast may make use of a broadcast tree (or other mechanism)
  - All collective operations can potentially make use of a tree (or other) mechanism to improve performance
Important to use the simplest collective operations which still achieve the needed results
Use MPI_IN_PLACE whenever appropriate
  - Reduces unnecessary memory usage and redundant data movement
What Else is There

Lots of lesser used routines not covered by this class
Derived datatypes
Process groups and communicators
Process topologies
Profiling
MPI-2
  Parallel I/O
  Dynamic process management
  One sided communication

Homework: How to compile and run MPI jobs

Use MPICH
Make sure MPI bin directory is in your path
To compile an MPI program, simply type:
> mpicc program.c
> mpiCC program.C
> mpif77 program.f
> mpif90 program.f
To run an MPI program, simply type:
> mpirun -np # a.out

Homework
Timing Routines

double MPI_Wtime()
DOUBLE PRECISION MPI_WTIME ()
  Returns a floating point number representing the elapsed number of seconds since some time in the past
  This time in the past is guaranteed not to change during the life of the process
  Clocks on each processor may not be synchronized

Homework assignment #1
ping-pong

Write an MPI program that has 2 processes continually send a message back and forth
  Have message sizes vary from 0 to 1 million bytes (0, 10, 100, etc)
  Time the exchange and calculate bandwidth (use MPI_Wtime())

Homework assignment #2
PI

Write an MPI program which will estimate the value of PI and which can run on an arbitrary number of processors
You should get the same value no matter the number of processors

Homework assignment #3
Safe communication

Rewrite the following unsafe program using 3 separate strategies for ensuring safeness.