Logistics

- **Exams not graded yet.**
  - Need to get them copied too!
    - Certification process
    - CS425 may not be taught in spring 2006

- **Project approval this week.**
  - No project leads to an automatic failing grade.

- Any Questions???
MPI Basic Concepts

• Processes have separate address spaces
• Communication is copying a part of one process’s address space to one or more other process’s address space.

Send

Receive

data

data

• The first process that “has” the data being copied issues the “send” library call the other process issues the “receive” library call.
Minimal Arguments

• **Send**
  - The data
    - Pointer to data (an application buffer)
    - Length of data in bytes
  - The ID of the destination process

• **Receive**
  - The “application buffer” to receive data in
    - Pointer to application buffer
    - Length of the application buffer
  - Variable to store the ID of the sender process
Minimal is not enough

• **Message Matching**
  - A message tag (another kind of ID)
    - Used by the queuing mechanisms to “filter” the right message for a given receive call.
    - The message that is actually received is the one that has the right tag.
  - The Source process ID
    - A single process that sent the message
    - A wild card that says receive the message with a given tag from any sending process.
  - On the receive side the actual length
    - Must be less than or equal to the designated buffer’s length.
Minimal Arguments

• **Send** *(address, length, destination, tag)*
  - The data
    - Pointer to data, Length of data in bytes
  - The ID of the destination process
  - The message tag ID

• **Receive** *(address, length, source, tag, actual_len)*
  - The “buffer” to receive data in
    - Pointer to buffer, Length of the buffer
  - ID of the sender process
    - Input, a screening parameter
    - Output, an information resource
  - The message tag ID
  - The actual length
Describing Message Buffers

• The (address, length) tuple for describing data is not really enough
  ➢ If the data is not contiguous
    ▪ Packing and unpacking has to occur
  ➢ If your system is heterogeneous
    ▪ Data formats may be different
    ▪ Byte order may be different

• The MPI solution
  ➢ (Address, count, datatype) tuple
    ▪ The starting address of “datatype” elements and count of them.
    ▪ “datatype” can be more complex than just “integer” etc.
      ♦ Can be noncontiguous structures as well.
Families of Messages

- The tag argument allows for scheduling of messages via the queuing mechanism of the message passing library
  - Necessary but insufficient

- What happens if you use a tag that a library developer has used.
  - Since message passing is inherently MIMD/SPMD there may be a point where the application is “sending” a message that a “library” will use in a different “context”

- MPI adds the context to the tag
  - Context is a runtime augmentation of “tag”
Process Identification

- Processes belong to groups.
- A group has “N” processes
- The rank of a process is from zero to “N-1”
- All processes in the parallel “job” belong to an initial group of processes.
MPI Communicators

- Combine the notion of “context” and “groups”
- The “communicator” is an argument to most communication calls.
  - Source and destination IDs refer to process IDs within the group of processes in the communicator.
  - Tags refer to tag IDs with the communicator
    - Adds the context to tags at run-time 😊
MPI_Send Specification

- **MPI_Send**(address, count, datatype, destination, tag, comm)
  - (address, count, datatype) tuple describes “count” occurrences of items of the form “datatype” starting at “address”
  - Destination is the rank of the destination in the group of processes associated with the communicator “comm”
  - “tag” is an integer used for message matching
  - “comm” identifies the group of processes and the communication “context”
MPI_Recv Specification

- **MPI_Recv(address, maxcount, datatype, source, tag, comm, status)**
  - The (address, maxcount, datatype) tuple describe the buffer that is used to receive the message
    - The actual “count” of “datatype” entities can be less than or equal to “maxcount”
  - Source is the rank of the process sending the message or a wild-card identifier that allows any source process.
  - Tag is the message filter ID or can also be a wild card.
  - “comm” identifies the group of processes and the communication “context”
  - Status is a structure that identifies the actual source, tag, and count of the received message.
MPI Collective Operations

• **Data movement operations**
  - Rearrange data among processes
  - **Examples:**
    - Broadcast
    - Gather
    - Scatter

• **Collective computation operations**
  - Reduction kinds of operations
  - **Examples:**
    - Min/max
    - Sum
    - Logical and, or, etc.
    - User-defined operations!
MPI Collective Operations [2]

• In both of the data movement and computational collective operations
  ➢ The system can optimize the library by
    ▪ Taking advantage of the architecture
    ▪ Increasing parallelism of operations

• Group Management
  ➢ Create subgroups
    ▪ For algorithms that minimally scale
  ➢ Task or functional parallelism
Virtual Topologies

- Allow conceptual view of processes in an application-oriented topology
  - Essentially ignoring the physical topology of the system.
  - Remember mapping the ring to a mesh
- General graphs and grids of processes are supported
  - We will use this in our MPI homework!
- Provide a high-level method for managing process groups without directly dealing with them.
MPI Communication Modes

• **Standard**
  - The blocking, and non-blocking mechanisms we have discussed operate in the way we described.

• **Synchronous**
  - Fully synchronous mechanism.
    - Helpful for debugging

• **Ready (for sends)**
  - System notified of the receive post
    - Allows the system to use a faster protocol if available.

• **Buffered**
  - Allows a user-controllable buffering for send operations.
Message Passing support for Libraries

• The communicator context and groups allow the independent development of “other” libraries.

• Libraries can create groups and base operations on these groups

• Tag matching depends upon the communicator context
  ➢ A big problem before MPI

• Independent error handlers etc.
Heterogeneous by definition

• MPI standard uses “datatypes” to identify data being sent from process to process.
• Allows machines that have different numerical representation formats to work in the same MPI program.
• Derived datatypes are formed from the basic pre-defined datatypes.
• MPI does not specify the mechanism used to make data representations uniform
  ➢ Conversion mechanism is up to the implementation
MPI does not specify all aspects

• Implementation details left to the Implementer
  ➢ Process startup
    ▪ MPICH uses “mpirun” a script
    ▪ IBM uses it’s POE (Parallel Operating Environment)
  ➢ Error codes
    ▪ Implementers can augment the standard specified ones
    ▪ Read the implementation details.
  ➢ System buffers available for messages
  ➢ Algorithms for collective operations
    ▪ K-ary (usually binary) tree operations
    ▪ Implementer can match the hardware
Minimal MPI

• The basic commands that can be used to write a parallel application (e.g., those used by this course – **nine routines**!)

  ➢ **MPI_Init**
  ➢ **MPI_Comm_size**
  ➢ **MPI_Comm_rank**
  ➢ **MPI_Send**
  ➢ **MPI_Recv**
  ➢ **MPI_Bcast**
  ➢ **MPI_Reduce**
  ➢ **MPI_Finalize**
  ➢ **MPI_Abort**

  - Initialize MPI
  - Number of processes
  - Which process I am
  - Send a message
  - Receive a message
  - Broadcast data with all
  - Global reductions
  - Terminate MPI
  - Handle errors “cleanly”
C version of Pi [1]

#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i; double mypi, pi, h, sum, x;
    double PI25DT = 3.141592653589793238462643;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) { if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d", &n);
    } /* end of if */
C version of Pi [2]

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0) break;
else {
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 
               pi, fabs(pi - PI25DT));
} /* close of while */    MPI_Finalize();     return 0; }
C++ version of PI [1]

```c
#include <math.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
    int n, rank, size, i; double mypi, pi, h, sum, x;
    double PI25DT = 3.141592653589793238462643;
    MPI::Init(argc, argv);
    size = MPI::COMM_WORLD.Get_size();
    rank = MPI::COMM_WORLD.Get_rank();
    while ((n = 1) {
        if (rank == 0) {
            cout << "Enter the number of intervals: (0 quits)"
            << endl;
            << endl;
            cin >> n;
        }
    }
```
C++ version of PI [2]

MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0);
if (n==0) break;
else {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI::COMM_WORLD.Reduce(&mypi, &pi, 1,
                              MPI::DOUBLE, MPI::SUM, 0);
if (rank == 0)
    cout << "pi is approximately " << pi
    << ", Error is " << fabs(pi - PI25DT)
    << endl;
}
} }  MPI::Finalize( );  return 0; }
Notes on the C++ interface

• It’s the newest component
  ➢ It may or may not work on our available resources.
    ▪ I don’t know either way!

• My experience is “ZERO” with it.
  ➢ Almost 😊

• C and Fortran work fine.

• C++ is pretty much at your own risk.
  ➢ Using the C interface in C++ is fine!
Basic MPI Timing Routines

• **MPI_Wtime**
  ➢ Returns an elapsed time on the calling processor
  ➢ **Synopsis**
    ▪ `#include "mpi.h"`
    ▪ `double MPI_Wtime()`
  ➢ **Return value**
    ▪ Time in seconds since an arbitrary time in the past.

• **MPI_Wtick**
  ➢ returns the resolution of MPI_WTIME in seconds. For example, if the hardware clock is incremented every millisecond, the value returned should be $10^{-3}$
  ➢ **Synopsis**
    ▪ `#include “mpi.h”`
    ▪ `double MPI_Wtick(void)`
  ➢ **Return value**
    ▪ Time between clock ticks
## MPI Datatypes in C

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>long</td>
</tr>
<tr>
<td>MPI_LONG_LONG_LONG_INT</td>
<td>long long</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>short</td>
</tr>
</tbody>
</table>
# MPI Datatypes in FORTRAN

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>FORTRAN Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHARACTER</td>
<td>character*1</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>double precision</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>real</td>
</tr>
<tr>
<td>MPI_INTEGER</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_INTEGER8</td>
<td>integer*8</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>complex</td>
</tr>
<tr>
<td>MPI_DOUBLE_COMPLEX</td>
<td>double precision complex</td>
</tr>
</tbody>
</table>
Minimal MPI

• The basic commands that can be used to write a parallel application (e.g., those used by this course – *nine routines!*)

  - `MPI_Init` Initialize MPI
  - `MPI_Comm_size` Number of processes
  - `MPI_Comm_rank` Which process I am
  - `MPI_Send` Send a message
  - `MPI_Recv` Receive a message
  - `MPI_Bcast` Broadcast data with all
  - `MPI_Reduce` Global reductions
  - `MPI_Finalize` Terminate MPI
  - `MPI_Abort` Handle errors “cleanly”
MPI_init

• Initialize the MPI execution environment

• Synopsis
  ➢ #include "mpi.h"
  ➢ int MPI_Init(int *argc, char **argv)

• Input Parameters
  ➢ argc
    ➢ Pointer to the number of arguments
  ➢ argv
    ➢ Pointer to the argument vector
MPI_Comm_size

- Determines the size of the group associated with a communicator

**Synopsis**

- `#include "mpi.h"
- `int MPI_Comm_size (MPI_Comm comm, int *size)`

**Input Parameter**

- `comm`
  - communicator (handle)

**Output Parameter**

- `size`
  - number of processes in the group of `comm` (integer)
MPI_Comm_rank

- **Determines the rank of the calling process in the communicator**

- **Synopsis**
  - `#include "mpi.h"
  - `int MPI_Comm_rank ( MPI_Comm comm, int *rank )`

- **Input Parameters**
  - `comm`
    - communicator (handle)

- **Output Parameter**
  - `rank`
    - rank of the calling process in group of comm (integer)
MPI_Send

- Performs a basic send
- Synopsis
  - #include "mpi.h"
  - int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )
- Input Parameters
  - buf initial address of send buffer (choice)
  - count number of elements in send buffer
  - datatype datatype of each send buffer element
  - dest rank of destination (integer)
  - tag message tag (integer)
  - comm communicator (handle)
MPI_Recv

- **Basic receive**
- **Synopsis**
  - `#include "mpi.h"
  - `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

- **Output Parameters**
  - `buf` initial address of receive buffer (choice)
  - `status` status object (Status)

- **Input Parameters**
  - `count` maximum number of elements in receive buffer
  - `datatype` datatype of each receive buffer element (handle)
  - `source` rank of source (integer)
  - `tag` message tag (integer)
  - `comm` communicator (handle)
MPI_Bcast

• **Broadcasts a message from the process with rank "root" to all other processes of the group.**

• **Synopsis**
  - #include "mpi.h"
  - int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )

• **Input/output Parameters**
  - buffer starting address of buffer (choice)
  - count number of entries in buffer (integer)
  - datatype data type of buffer (handle)
  - root rank of broadcast root (integer)
  - comm communicator (handle)
MPI_Reduce

• Reduces values on all processes to a single value

• Synopsis
  ```
  #include "mpi.h"
  int MPI_Reduce ( void *sendbuf, void *recvbuf, int count, 
  MPI_Datatype datatype, MPI_Op op, int root, 
  MPI_Comm comm )
  ```

• Input Parameters
  - sendbuf address of send buffer (choice)
  - count number of elements in send buffer (integer)
  - datatype data type of elements of send buffer (handle)
  - op reduce operation (handle)
  - root rank of root process (integer)
  - comm communicator (handle)

• Output Parameter
  - recvbuf address of receive buffer (significant only at root)
Reduction Operators

- **MPI_MAX** return the maximum
- **MPI_MIN** return the minimum
- **MPI_SUM** return the sum
- **MPI_PROD** return the product
- **MPI_LAND** return the logical and
- **MPI_BAND** return the bitwise and
- **MPI_LOR** return the logical or
- **MPI_BOR** return the bitwise or
- **MPI_LXOR** return the logical exclusive or
- **MPI_BXOR** return the bitwise exclusive or
- **MPI_MINLOC** return minimum and location
- **MPI_MAXLOC** return maximum and location
Reduction Operators

- Notes on collective operations
  - The reduction functions (MPI_Op) do not return an error value. As a result, if the functions detect an error, all they can do is either call MPI_Abort or silently skip the problem.
    - Thus, if you change the error handler from MPI_ERRORS_ARE_FATAL to something else, for example, MPI_ERRORS_RETURN, then no error may be indicated.
  - The reason for this is the performance problems in ensuring that all collective routines return the same error value.
  - Note that not all datatypes are valid for these functions. For example, MPI_COMPLEX is not valid for MPI_MAX and MPI_MIN.
  - In addition, the MPI 1.1 standard did not include the C types MPI_CHAR and MPI_UNSIGNED_CHAR among the lists of arithmetic types for operations like MPI_SUM.
    - However, since the C type char is an integer type (like short), it should have been included.
How does a broadcast work?