Today's Lecture
- Message Passing, largely for distributed memory
- Message Passing Interface (MPI): a Local View language
- Chapter 3 in textbook
- Sources for this lecture
  - Textbook slides
  - Online MPI tutorial

Message Passing and MPI
- Message passing is the principle alternative to shared memory parallel programming, predominant programming model for supercomputers and clusters
  - Portable
  - Low-level, but universal and matches earlier hardware execution model
- What it is
  - A library used within conventional sequential languages (Fortran, C, C++)
  - Based on Single Program, Multiple Data (SPMD)
  - Isolation of separate address spaces
    - no data races, but communication errors possible
    - exposes execution model and forces programmer to think about locality, both good for performance
  - Complexity and code growth!

Like OpenMP, MPI arose as a standard to replace a large number of proprietary message passing libraries.

Administrative
- Next programming assignment due on Monday, Nov. 7 at midnight
- Need to define teams and have initial conversation with me about projects by Nov. 10
- Project needs to be signed off on by me Nov. 22
Message Passing Library Features

- All communication, synchronization require subroutine calls
  - No shared variables
  - Program runs on a single processor just like any uniprocessor program, except for calls to message passing library
- Subroutines for
  - Communication
    - Pairwise or point-to-point: A message is sent from a specific sending process (point a) to a specific receiving process (point b).
    - Collectives involving multiple processors
      - Move data: Broadcast, Scatter/gather
      - Compute and move: Reduce, AllReduce
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Queries
    - How many processes? Which one am I? Any messages waiting?

MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - MPI_Comm_size reports the number of processes.
  - MPI_Comm_rank reports the rank, a number between 0 and size-1, identifying the calling process

Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Greetings from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```
Hello (C++)

```cpp
#include "mpi.h"
#include <iostream>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "Greetings from process " << rank << " of " << size << "\n";
    MPI::Finalize();
    return 0;
}
```

Compilation

- mpicc -g -Wall -o mpi_hello mpi_hello.c
- mpiexec -n <number of processes> ./mpi_hello

Execution

- mpiexec -n 1 ./mpi_hello
- mpiexec -n 4 ./mpi_hello
**MPI Components**

- **MPI_Init**
  - Tells MPI to do all the necessary setup.
  ```c
  int MPI_Init (
    int argc, char** argv)
  ```

- **MPI_Finalize**
  - Tells MPI we're done, so clean up anything allocated for this program.
  ```c
  int MPI_Finalize(void);
  ```

**Basic Outline**

```c
#include <mpi.h>

int main(int argc, char* argv[]) {
  ...
  MPI_Finalize();
  ...
  return 0;
}
```

**MPI Basic (Blocking) Send**

- **MPI_SEND(start, count, datatype, dest, tag, comm)**
- The message buffer is described by (start, count, datatype).
- The target process is specified by dest, which is the rank of the target process in the communicator specified by comm.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

**Things that need specifying:**
- How will "data" be described?
- How will processes be identified?
- How will the receiver recognize/screen messages?
- What will it mean for these operations to complete?

**Process 0**
- Send (data)

**Process 1**
- Receive (data)
MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (both source and tag) message is received from the system, and the buffer can be used
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE
- tag is a tag to be matched on or MPI_ANY_TAG
- receiving fewer than count occurrences of datatype is OK, but receiving more is an error
- status contains further information (e.g., size of message)

Some Basic Clarifying Concepts

- How to organize processes
  - Processes can be collected into groups
  - Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
  - A group and context together form a communicator
  - A process is identified by its rank in the group associated with a communicator

- There is a default communicator whose group contains all initial processes, called MPI_COMM_WORLD

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where

- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes

- There are MPI functions to construct custom datatypes, in particular ones for subarrays

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message

- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive

- Some non-MPI message-passing systems have called tags "message types," MPI calls them tags to avoid confusion with datatypes
A Simple MPI Program

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, buf;
    MPI_Status status;
    MPI_Init(&argv, &argc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* Process 0 sends and Process 1 receives */
    if (rank == 0) {
        buf = 123456;
        MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Received %d\n", buf);
    }
    MPI_Finalize();
    return 0;
}
```

Recall Trapezoidal Rule from L6: Serial algorithm

```c
/* Input: a, b, n */
```

```c
h = (b-a)/n;
```

```c
approx = (f(a) + f(b))/2.0;
```

```c
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
```

```c
approx = h*approx;
```

Parallel pseudo-code (naïve)

```c
1. Get a, b, n;
2. h = (b-a)/n;
3. local_n = n/comm_sz;
4. loccs_x = a + my_rank*local_n;
5. local_lb = local_x - local_n/2;
6. local_integ = Trap(loccs_x, local_lb, local_x, b);
7. if (my_rank != 0) {
8. Send local_integ to process 0;
9. } else if (my_rank == 0) {
10. total_integ = local_integ;
11. for (proc = 1; proc < comm_sz; proc++) {
12. Receive local_integ from proc;
13. total_integ += local_integ;
14. }
15. if (my_rank == 0) {
16. Print total result;
17. }
```

First version (1)

```c
1. int main(void) {
2. int my_rank, comm_sz, n = 1624, local_x;
3. double x = 0.0, b = 3.0, a = local_x, local_lb;
4. double local_int, total_int;
5. int source;
6. MPI_Init(NULL, NULL);
7. MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
8. MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
9. if (my_rank == 0) {
10. b = (b-a)/n;  // & is the same for all processes of
11. local_x = n/comm_sz;  // So is the number of trapezoids /
12. local_lb = a + my_rank*local_x;
13. local_int = Trap(a, local_lb, local_x, b);
14. total_int = local_int;
15. for (proc = 1; proc < comm_sz; proc++) {
16. Receive local_int from proc;  // local_x = a + proc*local_x;
17. total_int += local_int;
18. if (my_rank == 0) {
19. MPI_F40D&local_int, 1, MPI_DOUBLE, 0, 0,
20. MPI_COMM_WORLD);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

CS4961
First version (2)

```c
{ 
  total_int = local_int;
  for (source = 1; source < comm_sz; source++) {
  MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
  MPI_COMM_WORLD);
  if (key_rank == 0) {
    printf("With n = %d trapezoids, our estimate is: \n", n);
    printf("the integral from %f to %f = %f\n", a, b, total_int);
    MPI_Finalize();
    return 0;
  } /* make */
}
```
Predefined reduction operators in MPI

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>

Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function.
- For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.

- The arguments passed by each process to an MPI collective communication must be "compatible."
- For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

- The output_data_p argument is only used on dest_process.
- However, all of the processes still need to pass in an actual argument corresponding to output_data_p, even if it’s just NULL.
<table>
<thead>
<tr>
<th><strong>Collective vs. Point-to-Point Communications</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Point-to-point communications are matched on the basis of tags and communicators.</td>
</tr>
<tr>
<td>• Collective communications don't use tags.</td>
</tr>
<tr>
<td>• They're matched solely on the basis of the communicator and the order in which they're called.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Next Time</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• More detail on communication constructs</td>
</tr>
<tr>
<td>- Blocking vs. non-blocking</td>
</tr>
<tr>
<td>- One-sided communication</td>
</tr>
<tr>
<td>• Support for data and task parallelism</td>
</tr>
</tbody>
</table>