15-418 Spring’19
Recitation: Introduction to MPI

Lecturer: Nathan Beckmann

Based on slides by Greg Kesden
Based on earlier slides by William Gropp, Ewing Lusk of Argonne National Laboratory
Today we’ll learn…

• Message Passing Interface (MPI)
  – Basics
    • Communicators
    • Datatypes
  – How to build & run MPI programs
  – Send / Receive messages
    • Blocking
    • Non-blocking
  – Broadcast / Reduce
  – Debug / Profile
The Message-Passing Model

• A *process* is (traditionally) a program counter and address space.

• Processes may have multiple *threads* (program counters and associated stacks), which share a single address space.

• **MPI** is for communication among processes
  – Synchronization + data movement between address spaces
Flynn Parallelism Taxonomy

- SIMD (data-parallel): Vector
- SPMD (loosely sync’d data-parallel): GPU / MPI?
- MIMD (task-parallel): Pthreads / MPI
- MISD: streaming ???
MPI is Simple

• Many parallel programs can be written using just these six functions:
  
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECV`

  - **Setup / teardown**
    - **Who am I?**
    - **Message passing**
…But often painful

• In OpenMP, only needed a few #pragmas to make sequential code parallel
  – Easy because hardware takes care of data movement implicitly + guarantees coherence
  – ➔ Threads get the data they need when they need it automatically

• MPI requires explicit data movement

• ➔ Programmer (that’s you!) must say exactly what data goes where and when
Cooperative Operations for Communication

• The message-passing approach makes the exchange of data cooperative.
• Data is explicitly sent by one process and received by another.
• An advantage is that any change in the receiving process’s memory is made with the receiver’s explicit participation.
• ➔ Communication and synchronization combined!
One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled

```
Process 0

Put(data)  (memory)

(memory)  Get(data)

Process 1
```
What is MPI?

• **A message-passing library specification**
  – extended message-passing model
  – not a language or compiler specification
  – not a specific implementation or product

• For parallel computers, clusters, and heterogeneous networks

• Designed to provide access to advanced parallel hardware for
  – end users
  – library writers
  – tool developers
Why Use MPI?

- MPI provides a powerful, efficient, and portable way to express parallel programs
- MPI was explicitly designed to enable libraries...
- … which may eliminate the need for many users to learn (much of) MPI
A Minimal MPI Program (C)

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```
Error Handling

• By default, an error causes all processes to abort.
• The user can cause routines to return (with an error code) instead.
  – In C++, exceptions are thrown (MPI-2)
• A user can also write and install custom error handlers.
• Libraries might want to handle errors differently from applications.
Running MPI Programs

• MPI does not specify how to run an MPI program
  – Just as the C/C++ standard does not specify how to run a C/C++ program

• `mpirun <args>` is a recommendation, but not a requirement
Building MPI programs on GHC machines

• Setup your environment:
  – export PATH=$PATH:/usr/lib64/openmpi/bin

• Compile with MPIC++ / MPICC:
  – $ mpic++ -o hello hello.cpp

• Run via mpirun:
  – $ mpirun -c <NPROCS> hello
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
Better Hello (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( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}

Better Hello

• Note that in MPI each process is identical

• There is no “main thread” where execution begins
MPI Basic Send/Receive

- We need to fill in the details in

  Process 0
  
  Send(data)  
  
  Process 1
  
  Receive(data)

- 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?
What is message passing?

- Data transfer plus synchronization
- Requires cooperation of sender and receiver
- Cooperation not always apparent in code
Some Basic Concepts

• Processes can be collected into *groups*.
• Each message is sent in a *context*, and must be received in the same context.
• Group + context $\rightarrow$ *communicator*.

• There is a default communicator whose group contains all initial processes, called *MPI_COMM_WORLD*.
MPI Datatypes

- Messages are 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_PRECISION)
  - 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, such an array of (int, float) pairs, or a row of a matrix stored columnwise.
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.
Tags and Contexts

• Separation of messages used to be accomplished by use of tags, but
  – this requires libraries to be aware of tags used by other libraries.
  – this can be defeated by use of “wild card” tags.

• Contexts are different from tags
  – no wild cards allowed
  – allocated dynamically by the system when a library sets up a communicator for its own use.

• User-defined tags still provided in MPI for user convenience in organizing application

• Use MPI_Comm_split to create new communicators
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.
  – Beware: The message may not have been received by the target process!
MPI Basic (Blocking) Receive

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

• Waits until a matching (on 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.
• status contains further information
• Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.
MPI_Status

typedef struct __MPI_Status {
    int count;
    int cancelled;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status, *PMPI_Status;

Retrieving Further Information

- **Status** is a data structure allocated in the user’s program.
- In C:
  ```c
  int recvd_tag, recvd_from, recvd_count;
  MPI_Status status;
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
  recvd_tag = status.MPI_TAG;
  recvd_from = status.MPI_SOURCE;
  MPI_Get_count( &status, datatype, &recvd_count );
  ```
Send & Receive Example
(non-MPI version)

#include "assert.h"
#include <stdio.h>

int main(int argc, char* argv[]) {
    int N = 32;
    double fibs[N+2];
    fibs[0] = 1; fibs[1] = 1;
    for (int i = 2; i < N; i++) {
        fibs[i] = fibs[i-1] + fibs[i-2];
        printf("The %dth Fibonacci number is %g.\n", i, fibs[i]);
    }
    return 0;
}
#include "mpi.h"
#include "assert.h"
#include <stdio.h>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Status status;
    double msg[2] = {1,1};

    if (rank > 0) {
        double fibs[2];

        do {
            MPI_Recv(fibs, 2, MPI_DOUBLE,
                MPI_ANY_SOURCE, /*tag*/ 0,
                MPI_COMM_WORLD, &status);
        } while (status.MPI_ERROR);

        double next = fibs[0] + fibs[1];
        msg[0] = fibs[1]; msg[1] = next;

        printf("The %dth Fibonacci number is %g.\n", rank+2, next);
    }

    if (rank+1 < size) {
        int ret;
        ret = MPI_Send(msg, 2, MPI_DOUBLE,
            /*dest*/ rank + 1,
            /*tag*/ 0, MPI_COMM_WORLD);
        assert(ret == MPI_SUCCESS);
    }

    MPI_Finalize();
    return 0;
}
Sources of Deadlocks

• Send a large message from process 0 to process 1
  – If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

• What happens with

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Send (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Recv (0)</td>
</tr>
</tbody>
</table>

• This is called “unsafe” because it depends on the availability of system buffers
Deadlock example

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

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Status status;

    int msg = 1;

    MPI_Recv(&msg, 1, MPI_INTEGER, (rank-1) % size, 0, MPI_COMM_WORLD, NULL);
    MPI_Send(&msg, 1, MPI_INTEGER, (rank+1) % size, 0, MPI_COMM_WORLD);

    printf("Process %d done.\n", rank);

    MPI_Finalize();
    return 0;
}
```
Some Solutions to the “unsafe” Problem

• Order the operations more carefully:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

• Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
#include "mpi.h"
#include "assert.h"
#include <stdio.h>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Status status;

    int msg = 1;

    MPI_Send(&msg, 1, MPI_INTEGER, (rank+1) % size, 0, MPI_COMM_WORLD);
    MPI_Recv(&msg, 1, MPI_INTEGER, (rank-1) % size, 0, MPI_COMM_WORLD, NULL);

    printf("Process %d done.\n", rank);

    MPI_Finalize();
    return 0;
}
Non-Blocking Receive and Send

- int MPI_Isend(const void *buf,
  int count,
  MPI_Datatype datatype,
  int dest, int tag,
  MPI_Comm comm,
  MPI_Request *request)

- int MPI_Irecv(void *buf,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Request *request)
Waiting for a Non-Blocking Send and Receive to Complete

• Isend/Irecv return a MPI_Request* handle

• int MPI_Wait( MPI_Request *request,
                  MPI_Status *status)
  – Blocks for a previously non-blocking receive

• int MPI_Test( MPI_Request *request,
                int *flag,
                MPI_Status *status)
  – Test determines if done
    • C/C++ Convention: True/0, False/Non-Zero otherwise
Fixed deadlock example #1

... int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int sendmsg = 1, recvmsg;
    MPI_Request request;

    MPI_Irecv(&recvmsg, 1, MPI_INTEGER, (rank-1)%size, 0, MPI_COMM_WORLD, &request);
    MPI_Send(&sendmsg, 1, MPI_INTEGER, (rank+1)%size, 0, MPI_COMM_WORLD);
    MPI_Wait(&request, MPI_STATUS_IGNORE);

    printf("Process %d done.\n", rank, recvmsg, 0);
    assert(recvmsg == 1);

    MPI_Finalize();
    return 0;
}
MPI_Probe

- `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`

- Like a MPI_Recv, but just gets status
Probe example

... if (rank == 0) {
    int msglen = rand() % 1024; /* send a message of dynamic size */
    int *msg = new int[msglen];
    for (int i = 0; i < msglen; i++) {
        msg[i] = rand();
    }
    MPI_Send(msg, msglen, MPI_INTEGER, 1, 0, MPI_COMM_WORLD);
    delete [] msg;
}

} else if (rank == 1) {
    MPI_Status status; /* figure out how big the message is before recving */
    MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    int msglen;
    MPI_Get_count(&status, MPI_INTEGER, &msglen);
    int* msg = new int[msglen];
    MPI_Recv(msg, msglen, MPI_INTEGER, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    delete [] msg;
}

...
Introduction to Collective Operations in MPI

- Collective operations are **called by all processes** in a communicator.
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI_REDUCE** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.
Bcast/reduce example:

```c
int main(int argc, char *argv[]) {
    int done = 0, n;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    while (!done) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d", &n);
        if (n == 0) break;
        h = 1.0 / (double) n;
        sum = 0.0;
        for (int i = 1; i <= n; i++) {
            x = h * ((double) i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }
        mypi = h * sum;
        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    }
    return 0;
}
```
Bcast/reduce example (OpenMP):

```c
int main(int argc, char *argv[])
{
    int done = 0, n;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    while (!done) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d", &n);
        if (n == 0) break;
        h = 1.0 / (double) n;
        sum = 0.0;
        # pragma omp parallel for schedule(static)
        for (int i = 1; i <= n; i++) {
            x = h * ((double) i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }
        mypi = h * sum;
        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    }
    return 0;
}
```
Bcast/reduce example (MPI):

```c
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
    }
    MPI_Finalize();
}
```
Example: PI in C - 2

```c
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));

MPI_Finalize();
return 0;
```
Some Simple Exercises

• Compile and run the `hello` and `pi` programs.

• Modify the `pi` program to use send/receive instead of bcast/reduce.

• Write a program that sends a message around a ring. That is, process 0 reads a line from the terminal and sends it to process 1, who sends it to process 2, etc. The last process sends it back to process 0, who prints it.

• Time programs with `MPI_WTIME`. (Find it.)
Debugging MPI programs

• Don’t neglect your old friend printf
  - #define checkpoint() do { fprintf(stderr, “%s:%d\n”, __FILE__, __LINE__) } while(0)

• Attaching gdb to MPI processes
  - [link](https://www.open-mpi.org/faq/?category=debugging)
  - #define wait_for_gdb() do { int __wait_for_gdb = 0; while(__wait_for_gdb == 0) { sleep(1); }; } while(0)
    - Run gdb and attach at runtime, then manually set __wait_for_gdb

• There are more powerful tools…
  - [link](https://portal.tacc.utexas.edu/software/ddt)
MPI Sources

• The Standard itself:
  – at http://www.mpi-forum.org
  – All MPI official releases, in both postscript and HTML
• Books:
  – *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
• 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
Companion Material

- Online examples available at http://www.mcs.anl.gov/mpi/tutorials/perf
- ftp://ftp.mcs.anl.gov/mpi/mpiexmpl.tar.gz contains source code and run scripts that allows you to evaluate your own MPI implementation
Summary

• The parallel computing community has cooperated on the development of a standard for message-passing libraries.
• There are many implementations, on nearly all platforms.
• MPI subsets are easy to learn and use.
• Lots of MPI material is available.