CIS 631
Parallel Processing

Lecture 6: Message Passing
Programming and MPI

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Acknowledgements and Resources

- Portions of the lectures slides were adopted from:
  - Argonne National Laboratory, MPI tutorials.
  - Lawrence Livermore National Laboratory, MPI tutorials
  - See online tutorial links in course webpage

- W. Gropp, E. Lusk, and A. Skjellum,

- W. Gropp, E. Lusk, and R. Thakur,
Outline

- Background
  - The message-passing model
  - Origins of MPI and current status
  - Sources of further MPI information

- Basics of MPI message passing
  - Hello, World!
  - Fundamental concepts
  - Simple examples in Fortran and C

- Extended point-to-point operations
  - Non-blocking communication
  - Modes
Types of Parallel Computing Models

- Data parallel
  - Simultaneous execution on multiple data items
  - Example: Single Instruction, Multiple Data (SIMD)
- Task parallel
  - Different instructions on different data (MIMD)
- SPMD (Single Program, Multiple Data)
  - Combination of data parallel and task parallel
  - Not synchronized at individual operation level
- Message passing is for MIMD/SPMD parallelism
  - Can be used for data parallel programming
The Message-Passing Model

- A process is a program counter and address space
- Processes can have multiple threads (program counters and associated stacks) sharing a single address space
- MPI is for communication among processes (not threads)
- Interprocess communication consists of
  - Synchronization
  - Data movement
**SPMD**

- Data distributed across processes
  - Not shared

“Owner compute” rule: Process that “owns” the data (local data) performs computations on that data.

Shared program

Multiple data
Parallel Architecture Models and MPI

- Shared memory architectures
  - Parallel execution via shared memory
  - Bus-based or hierarchical memory systems
  - Shared memory may be physically distributed!
  - Data coherency issues dealt with in HW and SW

- Distributed memory architectures
  - Communication-base parallel execution
  - Interconnection networks important for performance
  - Scalable architecture (Why?)

- MPI targeted for distributed memory architectures
  - But can also run on shared memory machines (How?)
Message Passing Programming

- Defined by communication requirements
  - Data communication (necessary for algorithm)
  - Control communication (necessary for dependencies)
- Program behavior determined by communication patterns
- Message passing infrastructure attempts to support the forms of communication most often used or desired
  - Basic forms provide functional access
    - Can be used most often
  - Complex forms provide higher-level abstractions
    - Serve as basis for extension
    - Example: graph libraries, meshing libraries, …
  - Extensions for greater programming power
Cooperative Operations for Communication

- Data is cooperatively exchanged in message-passing
- Explicitly sent by one process and received by another
- Advantage of local control of memory
  - Any change in the receiving process’s memory is made with the receiver’s explicit participation
- Communication and synchronization are combined

Process 0

Send(data)

Process 1

Receive(data)

time
One-Sided Operations for Communication

- One-sided operations between processes
  - Include remote memory reads and writes
- Only one process needs to explicitly participate
  - There is still agreement implicit in the SPMD program
- Advantages?
  - Communication and synchronization are decoupled

![Diagram showing one-sided operations between Process 0 and Process 1]
Pairwise vs. Collective Communication

- Communication between process pairs
  - Send/Receive or Put/Get
  - Synchronous or asynchronous (we’ll talk about this later)

- Collective communication between multiple processes
  - Process group (collective)
    - Several processes logically grouped together
  - Communication within group
  - Collective operations
    - Communication patterns
      - broadcast, multicast, subset, scatter/gather, …
    - Reduction operations
What is MPI (Message Passing Interface)?

- Message-passing library (interface) specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- Targeted for parallel computers, clusters, and NOWs
- Specified in C, C++, Fortran 77, F90
- Full-featured and robust
- Designed to access advanced parallel hardware
  - End users
  - Library writers
  - Tool developers
**Why Use MPI?**

- Message passing is a mature parallel programming model
  - Well understood
  - Efficient match to hardware (interconnection networks)
  - Many applications
- 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
- Need standard, rich, and robust implementation
- Two versions: MPI-1 and MPI-2
  - Robust implementations including free MPICH (ANL)
Features of MPI

- **General**
  - Communicators combine context and group for security
  - Thread safety (implementation dependent)

- **Point-to-point communication**
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal, synchronous, ready, buffered

- **Collective**
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology
Features of MPI (continued)

- Application-oriented process topologies
  - Built-in support for grids and graphs (based on groups)
- Profiling
  - Hooks allow users to intercept MPI calls
  - Interposition library interface (PMPI)
  - Many tools (e.g., TAU) use PMPI
- Environmental
  - Inquiry
  - Error control
Is MPI Large or Small?

- MPI is large
  - MPI-1 is 128 functions, MPI-2 is 152 functions
  - Extensive functionality requires many functions
  - Not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs use just 6 basic functions
- “MPI is just right,” said Baby Bear
  - One can access flexibility when it is required
  - One need not master all parts of MPI to use it
Features not in MPI-1

- Non-message-passing concepts not included:
  - Process management
  - Remote memory transfers
  - Active messages
  - Threads
  - Virtual shared memory

- MPI does not address these issues, but has tried to remain compatible with these ideas
  - Example: thread safety as a goal

- Some of these features are in MPI-2
  - Example: put and get support and I/O
**To Use or Not Use MPI? That is the Question?**

**USE**
- You need a portable parallel program
- You are writing a parallel library
- You have irregular or dynamic data relationships that do not fit a data parallel model
- You care about performance and have to do Exercise 1

**NOT USE**
- You don’t need parallelism at all (Ha!)
- You can use libraries (which may be written in MPI)
- You can use multi-threading in a concurrent environment
Getting Started

- Writing MPI programs
- Compiling and linking
- Running MPI programs
A Simple 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;
}
```

What does this program do?
A Simple MPI Program (C++)

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

int main( int argc, char *argv[] )
{
    MPI::Init(argc, argv);
    cout << "Hello, world!" << endl;
    MPI::Finalize();
    return 0;
}
```
A Minimal MPI Program (Fortran)

```fortran
program main
  use MPI
  integer ierr

  call MPI_INIT( ierr )
  print *, 'Hello, world!'
  call MPI_FINALIZE( ierr )
end
```
MPI_Init

- What happens during MPI initialization?
- Think about it
- How do hardware resources get allocated?
  - Hmm, is this part of MPI?
- How do processes on different nodes get started?
  - Where does their executable program come from?
- What do the processes need to know?
- What about OS resources?
- What about tools that are running with MPI?
- …
MPI_Finalize

- Why do we need to finalize MPI?
- What happens during MPI finalization?
- Think about it
- What is necessary for a “graceful” MPI exit?
  - Can bad things happen otherwise?
  - Suppose the one process exits?
- How do resources get de-allocated?
- What about communications?
- What type of exit protocol might be used?
- What about tools?
Notes on C and Fortran

- C and Fortran library bindings correspond closely

- In C:
  - `mpi.h` must be included
  - MPI functions return error codes or `MPI_SUCCESS`

- In Fortran:
  - `mpif.h` must be included, or use MPI module (MPI-2)
  - All MPI calls are to subroutines
    - place for the return code in the last argument

- C++ bindings, and Fortran-90 issues, are part of MPI-2
Error Handling

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

- MPI-1 does not specify how to run an MPI program 🎉
- Starting an MPI program is dependent on implementation
  - Scripts, program arguments, and/or environment variables
- `% mpirun -np <procs> a.out`
  - For MPICH under Linux
- `mpiexec <args>`
  - Recommended part of MPI-2, as a recommendation
  - `mpiexec` for MPICH (distribution from ANL)
  - `mpirun` for SGI’s MPI
Finding Out About the Environment

- Two important questions that arise in message passing
  - How many processes are being used in 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
    - number between 0 and size-1
    - identifies the calling process


**Better “Hello World” (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( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- What does this program do and why is it better?
Better “Hello World” (Fortran)

program main
use MPI
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
MPI Basic Send/Receive

☐ We need to fill in the details in:

Process 0  Process 1

Send(data)  Receive(data)

time

☐ 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*
  - Must be received in the same context!
- A group and context together form a *communicator*
- A process is identified by its *rank*
  - With respect to the group associated with a communicator
- There is a default communicator **MPI_COMM_WORLD**
  - Contains all initial processes
**MPI Datatypes**

- Message data (sent or received) is described by a triple
  - address, count, datatype

- An MPI *datatype* is recursively defined as:
  - Predefined data type from the language
  - 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
  - Array of (int, float) pairs
  - Row of a matrix stored columnwise
MPI Tags

- Messages are sent with an accompanying user-defined integer \textit{tag}
  - Assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag
  - \texttt{MPI\_ANY\_TAG} matches any tag in a receive
- Tags are sometimes called “message types”
  - MPI calls them “tags” to avoid confusion with datatypes
**MPI Basic (Blocking) Send**

MPI\_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by:
  - \texttt{start}, \texttt{count}, \texttt{datatype}

- The target process is specified by \texttt{dest}
  - Rank of the target process in the communicator specified by \texttt{comm}

- Process blocks until:
  - Data has been delivered to the system
  - Buffer can be reused

- Message may not have been received by target process!
**MPI Basic (Blocking) Receive**

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

- Process blocks (waits) until:
  - A matching message is received from system
    - Matches on `source` and `tag`
  - Buffer must be available
  - `source` is rank in communicator specified by `comm`
    - Or `MPI_ANY_SOURCE`
- Status contains further information
- Receiving fewer than `count` is OK, more is not
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 );
```
program main
use MPI

integer rank, size, to, from, tag, count, i, ierr
integer src, dest
integer st_source, st_tag, st_count
integer status(MPI_STATUS_SIZE)
double precision data(10)

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'Process ', rank, ' of ', size, ' is alive'
dest = size - 1
src = 0
Simple Fortran Example - 2

if (rank .eq. 0) then
    do 10, i=1, 10
        data(i) = i
    10  continue
    call MPI_SEND( data, 10, MPI_DOUBLE_PRECISION,
                  dest, 2001, MPI_COMM_WORLD, ierr)
else if (rank .eq. dest) then
    tag = MPI_ANY_TAG
    source = MPI_ANY_SOURCE
    call MPI_RECV( data, 10, MPI_DOUBLE_PRECISION,
                   source, tag, MPI_COMM_WORLD,
                   status, ierr)
Simple Fortran Example - 3

call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION, + st_count, ierr )

st_source = status( MPI_SOURCE )
st_tag = status( MPI_TAG )
print *, 'status info: source = ', st_source,
+ ' tag = ', st_tag, 'count = ', st_count
endif

call MPI_FINALIZE( ierr )
end
Why Datatypes?

- All data is labeled by type in MPI
- Enables heterogeneous communication
  - Support communication between processes on machines with different memory representations and lengths of elementary datatypes
  - MPI provides the representation translation if necessary
- Allows application-oriented layout of data in memory
  - Reduces memory-to-memory copies in implementation
  - Allows use of special hardware (scatter/gather)
Tags and Contexts

- Separation of messages by use of tags
  - Requires libraries to be aware of tags of 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
Programming MPI with Only Six Functions

Many parallel programs can be written using:

- MPI_INIT()
- MPI_FINALIZE()
- MPI_COMM_SIZE()
- MPI_COMM_RANK()
- MPI_SEND()
- MPI_RECV()

What might be not so great with this?

Point-to-point (send/recv) isn’t the only way...

- Add more support for communication
Introduction to Collective Operations in MPI

- Called by all processes in a communicator

- **MPI_BCAST**
  - Distributes data from one process (the root) to all others

- **MPI_REDUCE**
  - Combines data from all processes in communicator
  - Returns it to one process

- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency
Example:  **PI in Fortran - 1**

Computes π using Monte-carlo integration

```fortran
program main
use MPI
double precision  PI25DT
parameter (PI25DT = 3.141592653589793238462643d0)
double precision  mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr

! function to integrate
f(a) = 4.d0 / (1.d0 + a*a)
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

10 if ( myid .eq. 0 ) then
   write(6,98)
98 format('Enter the number of intervals: (0 quits)')
read(5,99) n
99 format(i10)
endif
```
Example: PI in Fortran - 2

call MPI_BCAST( n, 1, MPI_INTEGER, 0,
+ MPI_COMM_WORLD, ierr)

c check for quit signal
if ( n .le. 0 ) goto 30

c calculate the interval size
h = 1.0d0/n
sum = 0.0d0
do 20 i = myid+1, n, numprocs
  x = h * (dble(i) - 0.5d0)
  sum = sum + f(x)
20 continue
mypi = h * sum

c collect all the partial sums
  call MPI_REDUCE( mypi, pi, 1, MPI_DOUBLE_PRECISION,
+ MPI_SUM, 0, MPI_COMM_WORLD,ierr)
c node 0 prints the answer
    if (myid .eq. 0) then
        write(6, 97) pi, abs(pi - PI25DT)
    97 format(' pi is approximately: ', F18.16,
            ' Error is: ', F18.16)
    endif
    goto 10
30 call MPI_FINALIZE(ierr)
end
#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;
    }
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;
```
Alternative set of 6 Functions for Simplified MPI

- Replace send and receive functions
  - MPI_INIT
  - MPI_FINALIZE
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_BCAST
  - MPI_REDUCE

- What else is needed (and why)?
Need to be Careful with Communication

- 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)

<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>

- What is wrong with this?

- This is unsafe because it depends on availability of system buffers
Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

  Process 0    Process 1

  Send(1)      Recv(0)
  Recv(1)      Send(0)

- Use non-blocking operations

  Process 0    Process 1

  Isend(1)     Isend(0)
  Irecv(1)     Irecv(0)
  Waitall      Waitall
Toward a Portable MPI Environment (MPICH)

- MPICH is a high-performance portable implementation
- It runs on MPP's, clusters, and heterogeneous NOWs
- In a wide variety of environments, one can do:
  - configure
  - make
  - mpicc -mpitrace myprog.c
  - mpirun -np 10 myprog
  - upshot myprog.log
- ... to build, compile, run, and analyze performance.
Extending the Message-Passing Interface

- Dynamic Process Management
  - Dynamic process startup
  - Dynamic establishment of connections
- One-sided communication
  - Put/get
  - Other operations
- Parallel I/O
- Other MPI-2 features
  - Generalized requests
  - Bindings for C++/Fortran-90; interlanguage issues
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
Next Class

☐ More MPI
PSIAM Talk

Re-emergent Informatics:
Bridging the Gap Between Science and Discipline
Neutral Cyberinfrastructure

Peter Fox, Rensselaer Polytechnic Institute

Thursday, April 15, 4:00pm
Herrington Room, Jaqua Academic Learning Center
Exercise 2 – Communication Patterns

☐ Create code skeletons for four communication patterns:
  ☐ Master - slave
  ☐ Ring
  ☐ Binary tree
  ☐ 2D nearest-neighbor

☐ Implement for general number of processes

☐ Develop testcases to demonstrate on Mist cluster