
Message Passing Programming (MPI)

Slides adopted from class notes by
Kathy Yelick

www.cs.berkeley.edu/~yellick/cs276f01/lectures/Lect07.html

(Which she adopted from Bill Saphir, Bill Gropp, Rusty Lusk,
Jim Demmel, David Culler, David Bailey, and Bob Lucas.)

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
- Not designed for fault tolerance

History of MPI

MPI Forum: government, industry and academia.

- Formal process began November 1992
- Draft presented at Supercomputing 1993
- Final standard (1.0) published May 1994
- Clarifications (1.1) published June 1995
- MPI-2 process began April, 1995
- MPI-1.2 finalized July 1997
- MPI-2 finalized July 1997

Current status of MPI-1

- Public domain versions from ANL/MSU (MPICH), OSC (LAM)
- Proprietary versions available from all vendors
 - Portability is the key reason why MPI is important.

MPI Programming Overview

1. Creating parallelism
 - SPMD Model
2. Communication between processors
 - Basic
 - Collective
 - Non-blocking
3. Synchronization
 - Point-to-point synchronization is done by message passing
 - Global synchronization done by collective communication

SPMD Model

- Single Program Multiple Data model of programming:
 - Each processor has a copy of the same program
 - All run them at their own rate
 - May take different paths through the code
- Process-specific control through variables like:
 - My process number
 - Total number of processors
- Processors may synchronize, but none is implicit

Hello World (Trivial)

- A simple, but not very interesting, SPMD Program.

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

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

Hello World (Independent Processes)

- MPI calls to allow processes to differentiate themselves

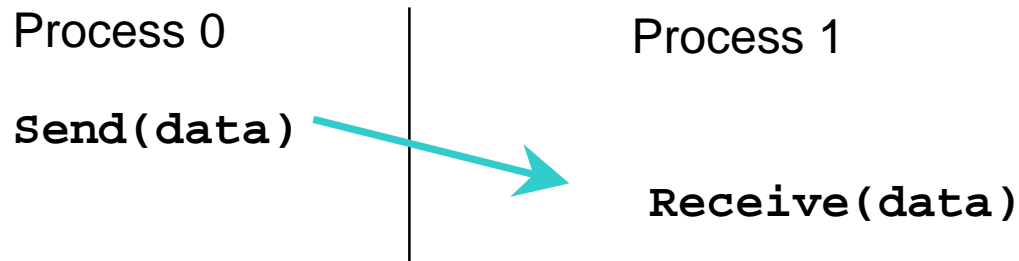
```
#include "mpi.h"
#include <stdio.h>

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 process %d of %d.\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

- This program may print in any order
(possibly even intermixing outputs from different processors!)

MPI Basic Send/Receive

- “Two sided” – both sender and receiver must take action.



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

Identifying Processes: MPI Communicators

- Processes can be subdivided into groups:
 - A process can be in many groups
 - Groups can overlap
- Supported using a “communicator:” a message *context* and a *group* of processes
 - More on this later...
- In a simple MPI program all processes do the same thing:
 - The set of all processes make up the “world”:
 - MPI_COMM_WORLD
 - Name processes by number (called “rank”)

Point-to-Point Communication Example

Process 0 sends 10-element array "A" to process 1

Process 1 receives it as "B"

1:

```
#define TAG 123
```

```
double A[10];
```

```
MPI_Send(A, 10, MPI_DOUBLE, 1,  
         TAG, MPI_COMM_WORLD)
```

2:

```
#define TAG 123
```

```
double B[10];
```

```
MPI_Recv(B, 10, MPI_DOUBLE, 0,  
        TAG, MPI_COMM_WORLD, &status)
```

or

```
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE,  
        MPI_ANY_TAG, MPI_COMM_WORLD, &status)
```

Process ID's



Describing Data: MPI Datatypes

- The data in a message to be sent or received 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_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 Predefined Datatypes

C:

- **MPI_INT**
- **MPI_FLOAT**
- **MPI_DOUBLE**
- **MPI_CHAR**
- **MPI_LONG**
- **MPI_UNSIGNED**

Language-independent

- **MPI_BYTE**

Fortran:

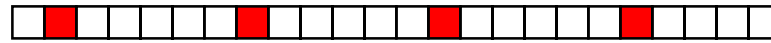
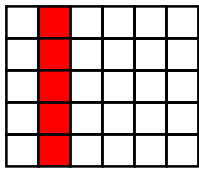
- **MPI_INTEGER**
- **MPI_REAL**
- **MPI_DOUBLE_PRECISION**
- **MPI_CHARACTER**
- **MPI_COMPLEX**
- **MPI_LOGICAL**

Why Make Datatypes Explicit?

- Can't the implementation just “send the bits?”
- To support heterogeneous machines:
 - All data is labeled with a type
 - MPI implementation can support communication on heterogeneous machines without compiler support
 - I.e., between machines with very different memory representations (big/little endian, IEEE fp or others, etc.)
- Simplifies programming for application-oriented layout:
 - Matrices in row/column
- May improve performance:
 - reduces memory-to-memory copies in the implementation
 - allows the use of special hardware (scatter/gather) when available

Using General Datatypes

- Can specify a strided or indexed datatype



layout in memory

- Aggregate types
 - **Vector**
 - Strided arrays, stride specified in elements
 - **Struct**
 - Arbitrary data at arbitrary displacements
 - **Indexed**
 - Like vector but displacements, blocks may be different lengths
 - Like struct, but single type and displacements in elements
- Performance may vary!

Recognizing & Screening Messages: MPI Tags

- Messages are sent with a user-defined integer *tag*:
 - Allows receiving process in identifying the message.
 - Receiver may also screen messages by specifying a tag.
 - Use `MPI_ANY_TAG` to avoid screening.
- Tags are called “message types” in some non-MPI message passing systems.

Message Status

- `status` is a data structure allocated in the user's program.
- Especially useful with wild-cards to find out what matched:

```
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 );
```

MPI Basic (Blocking) Send

MPI_SEND (**start**, **count**, **datatype**, **dest**, **tag**, **comm**)

- **start**: a pointer to the start of the data
 - **count**: the number of elements to be sent
 - **datatype**: the type of the data
 - **dest**: the rank of the destination process
 - **tag**: the tag on the message for matching
 - **comm**: the communicator to be used.
-
- Completion: When this function returns, the data has been delivered to the “system” and the data structure (start...start+count) can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

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

- **start**: a pointer to the start of the place to put data
 - **count**: the number of elements to be received
 - **datatype**: the type of the data
 - **source**: the rank of the sending process
 - **tag**: the tag on the message for matching
 - **comm**: the communicator to be used
 - **status**: place to put status information
-
- Waits until a matching (on **source** and **tag**) message is received from the system, and the buffer can be used.
 - Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.

Summary of Basic Point-to-Point MPI

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
 - `MPI_INIT`
 - `MPI_FINALIZE`
 - `MPI_COMM_SIZE`
 - `MPI_COMM_RANK`
 - `MPI_SEND`
 - `MPI_RECV`
- Point-to-point (send/recv) isn't the only way...

Collective Communication 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_Bcast(start, count, datatype,  
source, comm);
```
 - `MPI_REDUCE` combines data from all processes in communicator and returns it to one process.

```
MPI_Reduce(in, out, count, datatype,  
operation, dest, comm);
```
- In many algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

Example: Calculating PI

```
#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: Calculating PI (continued)

```
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;
}
```

Aside: this is a lousy
way to compute pi!

Non-Blocking Communication

- So far we have seen:
 - Point-to-point (blocking send/receive)
 - Collective communication
- Why do we call it blocking?
- The following is called an “unsafe” MPI program

Process 0

Process 1

Send(1)

Send(0)

Recv(1)

Recv(0)

- It may run or not, depending on the availability of system buffers to store the messages

Non-blocking Operations

Split communication operations into two parts.

- First part initiates the operation. It does not block.
- Second part waits for the operation to complete.

MPI_Request request;

MPI_Recv(buf, count, type, dest, tag, comm, status)

=

MPI_Irecv(buf, count, type, dest, tag, comm, &request)

+

MPI_Wait(&request, &status)

MPI_Send(buf, count, type, dest, tag, comm)

=

MPI_Isend(buf, count, type, dest, tag, comm, &request)

+

MPI_Wait(&request, &status)

Using Non-blocking Receive

- Two advantages:
 - No deadlock (correctness)
 - Data may be transferred concurrently (performance)

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
```

Process 0:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
MPI_Wait(&request, &status)
```

Process 1:

```
MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
MPI_Wait(&request, &status)
```

Using Non-Blocking Send

Also possible to use non-blocking send:

- “status” argument to **MPI_Wait** doesn’t return useful info here.
- But better to use **Irecv** instead of **Isend** if only using one.

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
p=1-me; /* calculates partner in exchange */
Process 0 and 1:
MPI_Isend(A, 100, MPI_DOUBLE, p, MYTAG, WORLD,
          &request)
MPI_Recv(B, 100, MPI_DOUBLE, p, MYTAG, WORLD,
         &status)
MPI_Wait(&request, &status)
```

Operations on MPI Request

- `MPI_Wait`(INOUT request, OUT status)
 - Waits for operation to complete and returns info in status
 - Frees request object (and sets to `MPI_REQUEST_NULL`)
- `MPI_Test`(INOUT request, OUT flag, OUT status)
 - Tests to see if operation is complete and returns info in status
 - Frees request object if complete
- `MPI_Request_free`(INOUT request)
 - Frees request object but does not wait for operation to complete
- Wildcards:
 - `MPI_Waitall`(..., INOUT array_of_requests, ...)
 - `MPI_Testall`(..., INOUT array_of_requests, ...)
 - `MPI_Waitany`/`MPI_Testany`/`MPI_Waitsome`/`MPI_Testsome`

Non-Blocking Communication Gotchas

- Obvious caveats:

- 1. You may not modify the buffer between `Isend()` and the corresponding `Wait()`. Results are undefined.
- 2. You may not look at or modify the buffer between `Irecv()` and the corresponding `Wait()`. Results are undefined.
- 3. You may not have two pending `Irecv()`s for the same buffer.

- Less obvious:

- 4. You may not *look* at the buffer between `Isend()` and the corresponding `Wait()`.
- 5. You may not have two pending `Isend()`s for the same buffer.

- **Why the `isend()` restrictions?**

- Restrictions give implementations more freedom, e.g.,
 - Heterogeneous computer with differing byte orders
 - Implementation swap bytes in the original buffer

More Send Modes

- **Standard**

- Send may not complete until matching receive is posted
- **MPI_Send, MPI_Isend**

- **Synchronous**

- Send does not complete until matching receive is posted
- **MPI_Ssend, MPI_Issend**

- **Ready**

- Matching receive must already have been posted
- **MPI_Rsend, MPI_Irsend**

- **Buffered**

- Buffers data in user-supplied buffer
- **MPI_Bsend, MPI_Ibsend**

Two Message Passing Implementations

- **Eager:** send data immediately; use pre-allocated or dynamically allocated remote buffer space.
 - One-way communication (fast)
 - Requires buffer management
 - Requires buffer copy
 - Does not synchronize processes (good)
- **Rendezvous:** send request to send; wait for ready message to send
 - Three-way communication (slow)
 - No buffer management
 - No buffer copy
 - Synchronizes processes (bad)

Point-to-Point Performance (Review)

- How do you model and measure point-to-point communication performance?
 - linear is often a good approximation
 - piecewise linear is sometimes better
 - the latency/bandwidth model helps understand performance
- A simple linear model:
data transfer time = latency + message size / bandwidth
 α β
- **latency** is startup time, independent of message size
- **bandwidth** is number of bytes per second (β is inverse)

- **Model:**

11/2/2001

MPI

Latency and Bandwidth

- for **short messages**, **latency dominates** transfer time
- for **long messages**, the **bandwidth** term **dominates** transfer time

- What are short and long?

latency term = bandwidth term

when

latency = message_size/bandwidth

- Critical message size = **latency * bandwidth**
- Example: 50 us * 50 MB/s = 2500 bytes
 - messages longer than 2500 bytes are bandwidth dominated
 - messages shorter than 2500 bytes are latency dominated

Effect of Buffering on Performance

- Copying to/from a buffer is like sending a message
 $\text{copy time} = \text{copy latency} + \text{message_size} / \text{copy bandwidth}$
- For a single-buffered message:
 $\text{total time} = \text{buffer copy time} + \text{network transfer time}$
 $= \text{copy latency} + \text{network latency}$
 $+ \text{message_size} * (1/\text{copy bandwidth} + 1/\text{network bandwidth})$
- Copy latency is sometimes trivial compared to effective network latency
 $1/\text{effective bandwidth} = 1/\text{copy_bandwidth} + 1/\text{network_bandwidth}$
- Lesson: **Buffering hurts bandwidth**

Communicators

- What is MPI_COMM_WORLD?
- A **communicator** consists of:
 - A group of processes
 - Numbered 0 ... N-1
 - Never changes membership
 - A set of private communication channels between them
 - Message sent with one communicator cannot be received by another.
 - Implemented using hidden message tags
- Why?
 - Enables development of safe libraries
 - Restricting communication to subgroups is useful

Safe Libraries

- User code may interact unintentionally with library code.
 - User code may send message received by library
 - Library may send message received by user code

```
start_communication();  
library_call(); /* library communicates internally */  
wait();
```

- Solution: library uses private communication domain
- A communicator is private virtual communication domain:
 - All communication performed w.r.t a communicator
 - Source/destination ranks with respect to communicator
 - Message sent on one cannot be received on another.

Notes on C and Fortran

- MPI is language independent, and has “language bindings” for C and Fortran, and many other languages
 - C and Fortran 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, with a place for the return code in the last argument.
- C++ bindings, and Fortran-90 issues, are part of MPI-2.

Free MPI Implementations (I)

- **MPICH** from Argonne National Lab and Mississippi State Univ.
 - <http://www.mcs.anl.gov/mpi/mpich>
- Runs on
 - Networks of workstations (IBM, DEC, HP, IRIX, Solaris, SunOS, Linux, Win 95/NT)
 - MPPs (Paragon, CM-5, Meiko, T3D) using native M.P.
 - SMPs using shared memory
- Strengths
 - Free, with source
 - Easy to port to new machines and get good performance (ADI)
 - Easy to configure, build
- Weaknesses
 - Large
 - No virtual machine model for networks of workstations

Free MPI Implementations (II)

- **LAM** (Local Area Multicomputer)
- Developed at the Ohio Supercomputer Center
 - <http://www.mpi.nd.edu/lam>
 - Runs on
 - SGI, IBM, DEC, HP, SUN, LINUX
 - Strengths
 - Free, with source
 - Virtual machine model for networks of workstations
 - Lots of debugging tools and features
 - Has early implementation of MPI-2 dynamic process management
 - Weaknesses
 - Does not run on MPPs

MPI Sources

- The Standard itself is at: <http://www.mpi-forum.org>
 - All MPI official releases, in both postscript and HTML
- Books:
 - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
 - *MPI: The Complete Reference*, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
 - *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
 - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
 - *MPI: The Complete Reference Vol 1 and 2*, MIT Press, 1998(Fall).
- Other information on Web:
 - <http://www.mcs.anl.gov/mpi>

MPI-2 Features

- Dynamic process management
 - Spawn new processes
 - Client/server
 - Peer-to-peer
- One-sided communication
 - Remote Get/Put/Accumulate
 - Locking and synchronization mechanisms
- I/O
 - Allows MPI processes to write cooperatively to a single file
 - Makes extensive use of MPI datatypes to express distribution of file data among processes
 - Allow optimizations such as collective buffering
- I/O has been implemented; 1-sided becoming available.