CS267
MPI

Bill Saphir

Berkeley Lab/NERSC
Phone: 510-486-4373
wcsaphir@lbl.gov
What is Message Passing?

- Message passing is a model for programming distributed memory parallel computers
  - Every processor executes an independent process
  - Disjoint address spaces, no shared data
  - All communication between processes is done cooperatively, through subroutine calls

- SPMD: single program, multiple data
  - Every processes is the “same” (e.g. a.out); may act on different data
- MPMD: multiple program, multiple data
  - Not all processes are the “same” (e.g. a.out, b.out, c.out)
What is the Message Passing Interface?

MPI is the de facto standard for scientific programming on distributed memory parallel computers.

- MPI is a library of routines that enable message passing applications
- MPI is an interface specification, not a specific implementation
- Almost all high performance scientific applications run at NERSC and other supercomputer centers use MPI

The message passing model is

- A painful experience for many application programmers
- Old technology – “assembly language for parallel programming”

Message passing has succeeded because

- It maps well to a wide range of hardware
- Parallelism is explicit and communication is explicit
  - Forces the programmer to tackle parallelization from the beginning.
- Parallelizing compilers are very hard
- MPI makes programs portable
MPI History

- Before MPI: different library for each type of computer:
  - CMMD (Thinking Machines CM5)
  - NX (Intel iPSC/860, Paragon)
  - MPL (SP2)
  - and many more
- PVM: tried to be a standard, but not high performance, not carefully specified
- MPI was developed by the MPI Forum: voluntary organization representing industry, government labs, academia
  - 1994 MPI-1 – codified existing practice
  - 1997 MPI-2 – research project
- Both MPI-1 and MPI-2 were designed by committee. There is a core of good stuff but just because it’s in the standard doesn’t mean you should use it.
What’s in MPI

• MPI-1
  • Utilities: “who am I?”, “how many processes are there”
  • Send/recv communication
  • Collective communication e.g. broadcast, reduction, all-to-all
  • Lots of other stuff (for the longer version of this talk)

• MPI-2
  • Parallel I/O
  • C++/Fortran 90
  • One-sided communication – get/put
  • More stuff that is rarely used

Most of this talk will be about how to exchange data using MPI
Cooperative Data Transfer

Send operation in process 1 is matched by receive operation in process 2:
Models related to message passing

Active messages
- Message contains address of handler that processes incoming data
- No receive operations
- Separate bulk transfer mechanism

Remote memory operations (get/put, 1-sided communication)
- Process may directly access memory of another process with get and put operations
- Other synchronization mechanisms to coordinate access

Common features
- Separate processes
- Separate address spaces (distributed memory model)
- Processes execute independently and concurrently
What’s in MPI

• MPI-1
  • Utilities: “who am I?”, “how many processes are there”
  • Send/receive communication
  • Collective communication e.g. broadcast, reduction, all-to-all
  • Many other things

• MPI-2
  • Parallel I/O
  • C++/Fortran 90
  • One-sided communication – get/put
  • Many other things

• Not in MPI
  • Process startup, environment, standard input/output
  • Fault tolerance
An MPI Application

The elements of the application are:
- 4 processes, numbered zero through three
- Communication paths between them

The set of processes plus the communication channels is called “MPI_COMM_WORLD”. More on the name later.
“Hello World” — C

```c
#include <mpi.h>
main(int argc, char *argv[]) {
    int me, nprocs
    MPI_Init(&argc, &argv)
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
    MPI_Comm_rank(MPI_COMM_WORLD, &me)
    printf("Hi from node %d of %d\n", me, nprocs)
    MPI_Finalize()
}
```
Compiling and Running

Different on every machine.

Compile:

```
mpicc -o hello hello.c
mpif77 -o hello hello.c
```

Start four processes (somewhere):

```
mpirun -np 4 ./hello
```
“Hello world” output

Run with 4 processes:

Hi from node 2 of 4
Hi from node 1 of 4
Hi from node 3 of 4
Hi from node 0 of 4

Note:

- Order of output is not specified by MPI
- Ability to use `stdout` is not even guaranteed by MPI!
Point-to-point communication in MPI

MPI_Send(data, ...)  MPI_Recv(data, ...)
Point-to-point Example

Process 0 sends array “A” to process 1 which receives it as “B”

1:

```c
#define TAG 123
double A[10];
MPI_Send(A, 10, MPI_DOUBLE, 1, TAG, MPI_COMM_WORLD)
```

2:

```c
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0, TAG,
          MPI_COMM_WORLD, &status)
```

or

```c
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &status)
```
Some Predefined datatypes

C:

MPI_INT
MPI_FLOAT
MPI_DOUBLE
MPI_CHAR
MPI_LONG
MPI_UNSIGNED

Fortran:

MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_CHARACTER
MPI_COMPLEX
MPI_LOGICAL

Language-independent

MPI_BYTE
Source/Destination/Tag

src/dest

dest
  • Rank of process message is being sent to (destination)
  • Must be a valid rank (0...N-1) in communicator

src
  • Rank of process message is being received from (source)
  • “Wildcard” MPI_ANY_SOURCE matches any source

tag
  • On the sending side, specifies a label for a message
  • On the receiving side, must match incoming message
  • On receiving side, MPI_ANY_TAG matches any tag
Status argument

In C: MPI_Status is a structure

• `status.MPI_TAG` is tag of incoming message
  (useful if `MPI_ANY_TAG` was specified)
• `status.MPI_SOURCE` is source of incoming message
  (useful if `MPI_ANY_SOURCE` was specified)
• How many elements of given datatype were received
  `MPI_Get_count(IN status, IN datatype, OUT count)`

In Fortran: status is an array of integer

```fortran
integer status(MPI_STATUS_SIZE)
status(MPI_SOURCE)
status(MPI_TAG)
```

In MPI-2: Will be able to specify `MPI_STATUS_IGNORE`
Guidelines for using wildcards

Unless there is a good reason to do so, do not use wildcards

Good reasons to use wildcards:

• Receiving messages from several sources into the same buffer but don’t care about the order (use MPI_ANY_SOURCE)

• Receiving several messages from the same source into the same buffer, and don’t care about the order (use MPI_ANY_TAG)
Exchanging Data

- Example with two processes: 0 and 1
- General data exchange is very similar

Requires Buffering to succeed!
Deadlock

The MPI specification is wishy-washy about deadlock.

- A **safe** program does not rely on system buffering.

- An **unsafe** program may rely on buffering but is not as portable.

**Ignore this.** MPI is all about writing portable programs.

Better:

- A **correct** program does not rely on buffering

- A program that relies on buffering to avoid deadlock is **incorrect**.

In other words, it is your fault if your program deadlocks.
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.

```c
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 operations

```c
#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)
```

• No deadlock
• Data may be transferred concurrently
Using non-blocking operations (II)

Also possible to use nonblocking send:

```c
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
p=1-me; /* calculates partner in 2 process exchange */
```

Process 0 and 1:
```c
MPI_Isend(A, 100, MPI_DOUBLE, p, MYTAG, WORLD, &request)
MPI_Recv(B, 100, MPI_DOUBLE, p, MYTAG, WORLD, &status)
MPI_Wait(&request, &status)
```

- No deadlock
- “status” argument to `MPI_Wait` doesn’t return useful info here.
- Better to use `Irecv` instead of `Isend` if only using one.
**Overlapping communication and computation**

On some computers it may be possible to do useful work while data is being transferred.

```c
MPI_Request requests[2];
MPI_Status statuses[2];

MPI_Irecv(B, 100, MPI_DOUBLE, p, 0, WORLD, &request[1])
MPI_Isend(A, 100, MPI_DOUBLE, p, 0, WORLD, &request[0])

.... do some useful work here ....

MPI_Waitall(2, requests, statuses)
```

- **Irecv/Isend** initiate communication
- Communication proceeds “behind the scenes” while processor is doing useful work
- Need both **Isend** and **Irecv** for real overlap (not just one)
- Hardware support necessary for true overlap
- This is why “o” in “LogP” is interesting.
Operations on MPI_Request

**MPI_Wait**(INOUT request, OUT status)
- Waits for operation to complete
- Returns information (if applicable) 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
- Returns information in status if complete
- Frees request object if complete

**MPI_Request_free**(INOUT request)
- Frees request object but does not wait for operation to complete

**MPI_Waitall**(..., INOUT array_of_requests, ...
**MPI_Testall**(..., INOUT array_of_requests, ...
**MPI_Waitany/MPI_Testany/MPI_Waitsome/MPI_Testsome**

**MPI_Cancel** cancels or completes a request. Problematic.
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 gotchas:

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.
MPI_Send semantics

Most important:
- Buffer may be reused after MPI_Send() returns
- May or may not block until a matching receive is called (non-local)

Others:
- Messages are non-overtaking
- Progress happens
- Fairness not guaranteed

**MPI_Send does not require a particular implementation, as long as it obeys these semantics.**
Point-to-point Performance (review)

How do you model and measure point-to-point communication performance?

\[
data \text{ transfer time} = f(\text{message size})
\]

Often a linear model is a good approximation

\[
data \text{ transfer time} = \text{latency} + \frac{\text{message size}}{\text{bandwidth}}
\]

- \text{latency} is startup time, independent of message size
- \text{bandwidth} is number of bytes per second

- linear is often a good approximation
- piecewise linear is sometimes better
- the latency/bandwidth model helps understand performance issues
Latency and bandwidth

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

What are short and long?

\[
\text{latency term} = \frac{\text{message size}}{\text{bandwidth}}
\]

when

\[
\text{latency} = \frac{\text{message size}}{\text{bandwidth}}
\]

Critical message size = \textit{latency} * \textit{bandwidth}

Example: \textbf{50 \, us} * \textbf{50 MB/s} = \textbf{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

\[
\text{copy time} = \text{copy latency} + \frac{\text{message size}}{\text{copy bandwidth}}
\]

For a single-buffered message:

\[
\text{total time} = \text{buffer copy time} + \text{network transfer time}
\]
\[
= \text{copy latency} + \text{network latency} + \frac{\text{message size} \times (1/\text{copy bandwidth} + 1/\text{network bandwidth})}{\text{1/effective bandwidth}}
\]

Copy latency is sometimes trivial compared to effective network latency

\[
\frac{1}{\text{effective bandwidth}} = \frac{1}{\text{copy bandwidth}} + \frac{1}{\text{network bandwidth}}
\]

Lesson: **Buffering hurts bandwidth**
Mixing protocols for high performance of MPI_Send

Description
- **Eager** for short messages
- **Rendezvous** for long messages
- Switch protocols near latency-bandwidth product

Features
- Low latency for latency-dominated (short) messages
- High bandwidth for bandwidth-dominated (long) messages
- Reasonable memory management (upper limit on size of message that may be buffered)
- Non-ideal performance for some messages near critical size
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`

Don't use these. They exist because MPI was designed by committee and they offer little benefit.
Communicators

- MPI_COMM_WORLD is a communicator

- A communicator is an object that represents
  - A set of processes
  - Private communication channels between those processes

- Uses of communicators
  - Scope for collective operations
  - Writing safe libraries
    isend(); irecv();
    library_call_with_internal_communication();
    MPI_Wait();
**MPI_COMM_WORLD**

**MPI_COMM_WORLD** is
- A group of all initial MPI processes
- Communication channels between them

```plaintext
MPI_Send(buf, len, type, dest, tag, MPI_COMM_WORLD)
```

dest is a rank in **MPI_COMM_WORLD**
Collective Operations

Collective communication is communication among a group of processes:

- Broadcast
- Synchronization (barrier)
- Global operations (reductions)
- Scatter/gather
- Parallel prefix (scan)
Barrier

MPI_Barrier(communicator)

No process leaves the barrier until all processes have entered it.

Model for collective communication:
- All processes in communicator must participate
- Process might not finish until have all have started.
Broadcast

\texttt{MPI\_Bcast(buf, len, type, root, comm)}

- Process with rank $=$ root is source of data (in buf)
- Other processes receive data

\texttt{MPI\_Comm\_rank(MPI\_COMM\_WORLD, \&myid);} 
\texttt{if (myid == 0) \{ 
  /* read data from file */ 
\}} 
\texttt{MPI\_Bcast(data, len, type, 0, MPI\_COMM\_WORLD);} 

Note:
- All processes must participate
- MPI has no “multicast” that is matched by a receive
Reduction

Combine elements in input buffer from each process, placing result in output buffer.

\[
\text{MPI\_Reduce}(\text{indata}, \text{outdata}, \text{count}, \text{type}, \text{op}, \text{root}, \text{comm})
\]
\[
\text{MPI\_Allreduce}(\text{indata}, \text{outdata}, \text{count}, \text{type}, \text{op}, \text{comm})
\]

- Reduce: output appears only in buffer on root
- Allreduce: output appears on all processes

operation types:

- \text{MPI\_SUM}
- \text{MPI\_PROD}
- \text{MPI\_MAX}
- \text{MPI\_MIN}
- \text{MPI\_BAND}
- arbitrary user-defined operations on arbitrary user-defined datatypes
Reduction example: dot product

/* distribute two vectors over all processes such that
   processor 0 has elements 0...99
   processor 1 has elements 100...199
   processor 2 has elements 200...299
   etc.
 */

double dotprod(double a[100], double b[100])
{
    double gresult = lresult = 0.0;
    integer i;
    /* compute local dot product */
    for (i = 0; i < 100; i++) lresult += a[i]*b[i];
    MPI_Allreduce(lresult, gresult, 1, MPI_DOUBLE,
                  MPI_SUM, MPI_COMM_WORLD);
    return(gresult);
}
Data movement: all-to-all

All processes send and receive data from all other processes.

```c
MPI_Alltoall(sendbuf, sendcount, sendtype,
             recvbuf, recvcount, recvtype,
             comm)
```

For a communicator with N processes:

- `sendbuf` contains N blocks of `sendcount` elements each
- `recvbuf` receives N blocks of `recvcount` elements each
- Each process sends block `i` of `sendbuf` to process `i`
- Each process receives block `i` of `recvbuf` from process `i`

Example: multidimensional FFT (matrix transpose)
Other collective operations

There are many more collective operations provided by MPI:

**MPI_Gather/Gatherv/Allgather/Allgatherv**

- each process contributes local data that is gathered into a larger array

**MPI_Scatter/Scatterv**

- subparts of a single large array are distributed to processes

**MPI_Reduce_scatter**

- same as Reduce + Scatter

**Scan**

- prefix reduction

The “v” versions allow processes to contribute different amounts of data
Semantics of collective operations

For all collective operations:
• Must be called by all processes in a communicator

Some collective operations also have the “barrier” property:
• Will not return until all processes have started the operation
  • MPI_Barrier, MPI_Allreduce, MPI_Alltoall, etc.

Others have the weaker property:
• May not return until all processes have started the operation
  • MPI_Bcast, MPI_Reduce, MPI_Comm_dup, etc.
Performance of collective operations

Consider the following implementation if \texttt{MPI\_Bcast}:

```
if (me == root) {
    for (i = 0; i < N; i++) {
        if (i != me) MPI\_Send(buf, ..., dest=i, ...);
    }
} else {
    MPI\_Recv(buf, ..., src=i, ...);
}
```

**Non-scalable**: time to execute grows linearly with number of processes.

High-quality implementations of collective operations use algorithms with better scaling properties \textit{if} the network supports multiple simultaneous data transfers.

- Algorithm may depend on size of data
- Algorithm may depend on topology of network
Timing

Double precision wallclock time, in seconds.

double t1, t2;
t1 = MPI_Wtime();

.... do some work ...

t2 = MPI_Wtime();
printf("Elapsed time is %f seconds\n", t2-t1);

Notes:
• Time starts at some arbitrary point in the past
• Note times not synchronized unless MPI_WTIME_IS_GLOBAL
Accurate timing is not simple

Three standard problems

- Processes are unsynchronized to start
- Load imbalance shows up in collective and point-to-point operations
- Extra synchronization to avoid problems 1+2 causes network contention
Motivation for MPI-2 I/O

• Consider 2D array, row-major order, to be stored in single file, partitioned among 4 processors

• Each processor writes many small non-contiguous blocks
C++

• MPI needs a C++ interface
  • Key question: closely related to C interface or full-blown OO?
  • Decision: C++ interface is close to C/Fortran interface.

• General principles:
  • MPI handles (MPI_Comm, etc.) become C++ objects.
  • MPI functions become methods on C++ classes.
  • Do what C++ programmers expect where possible but
  • Don't stray too far from MPI principles.
More C++ principles

- Shallow copies

- Constructors create `MPI Xxx NULL`. Destructors do not free.
  - User must generally use `create` and `free`
  - Reasons
    - Variables going in and out of scope could be collective operations
    - Automatic destruction violates shallow copy semantics
Fortran 90 support

Fortran 90 has many "modern" features.

- User-defined types
- Function overloading
- Parameterized types
- Mechanisms for strict type checking (interface blocks)
- First class arrays

Can MPI take advantage of these? Mostly no.
Fortran 90 vs. MPI

Fortran 90 and MPI are not completely compatible

- **MPI has choice arguments**
  - F90 argument checking is strict
  - Derived types require argument checking

- **MPI assumes flat address space**
  - F90 does not require sequence association
Advanced topics for followup reading

- **Creation and manipulation of communicators**
  - Useful if you need to do collective operations over subsets of processes
- **Topologies**
  - Allow applications with simple communication topology to be well-mapped to network topology. Can be important on machines with mesh networks (Red Storm, X1, Blue Gene)
- **Profiling interface**
  - Makes it easy and transparent to application to wrap MPI routines with profiling routines
- **User Defined Datatypes**
  - Difficult to use, and can result in poor performance
- **Persistent communication**
- **One-sided communication – get/put**
  - MPI is not the standard interface. Cray shmem library is still the standard
- **Dynamic process management** (a solution in search of a problem)
  - And a large number of other features.
Where to get more information

Home pages
- http://www.mpi-forum.org
- http://www.mcs.anl.gov/mpi

Newsgroups
- comp.parallel.mpi

Books
- Using MPI, by Gropp, Lusk, Skjellum. The MIT Press
- Parallel Programming with MPI, by Pacheco. Morgan Kauffman