

# Lecture 31—Message Passing Interface (MPI) & Clusters; The Cloud

ECE 459: Programming for Performance

March 23, 2015

# Part I

## MPI & Clusters

# Continuing to Switch Gears

So far, we've seen how to make things fast on one computer:

- threads;
- compiler optimizations;
- GPUs.

To get a lot of bandwidth, though, you need lots of computers,  
(if you're lucky and the problem allows!)

Today: programming for performance  
with multiple computers via MPI.

## Key Idea: Explicit Communication

Mostly we've seen shared-memory systems;  
complication: must manage contention.

Recently, GPU programming: explicitly copy data.

Message-passing: yet another paradigm.

# What is MPI?

## **Message Passing Interface:**

A language-independent communication protocol for parallel computers.

- Use it to run the same code on a number of **nodes** (different hardware threads; or servers in a cluster).
- Provides explicit message passing between nodes.
- Is the dominant model for high performance computing (de-facto standard).

# High Level View of MPI

MPI is a type of SPMD (single process, multiple data).

Idea: have multiple instances of the same program,  
all working on different data.

The program could be running on the same machine,  
or a cluster of machines.

MPI facilitates communication of data between processes.

# MPI Functions

```
// Initialize MPI
int MPI_Init(int *argc, char **argv)

// Determine number of processes within a communicator
int MPI_Comm_size(MPI_Comm comm, int *size)

// Determine processor rank within a communicator
int MPI_Comm_rank(MPI_Comm comm, int *rank)

// Exit MPI (must be called last by all processors)
int MPI_Finalize()

// Send a message
int MPI_Send (void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)

// Receive a message
int MPI_Recv (void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

# MPI Function Notes

- `MPI_Comm`: a **communicator**,  
often `MPI_COMM_WORLD` for global channel.
- `MPI_Datatype`: just an enum, e.g. `MPI_FLOAT_INT`, etc.
- `dest/source`: “rank” of the process (in a communicator)  
to send a message to/receive a message from;  
you may use `MPI_ANY_SOURCE` in `MPI_Recv`.
- Both `MPI_Send` and `MPI_Recv` are blocking calls—  
see `man MPI_Send` or `man MPI_Recv` for more details.
- The tag allows you to organize your messages,  
so you can filter all but a specific tag.



# Hello, World in MPI

As with OpenCL kernels:

first, figure out what “current” process is supposed to compute.

```
// http://www.dartmouth.edu/~rc/classes/intro\_mpi/
#include <stdio.h>
#include <mpi.h>

int main (int argc, char * argv[])
{
    int rank, size;

    /* start MPI */
    MPI_Init (&argc, &argv);
    /* get current process id */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    /* get number of processes */
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf("Hello_world_from_process_%d_of_%d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

## Longer MPI Example (from Wikipedia)

Here's a common example:

- The “master” (rank 0) process creates some strings and sends them to the worker processes.
- The worker processes modify their string and send it back to the master.

Source:

[http://en.wikipedia.org/wiki/Message\\_Passing\\_Interface](http://en.wikipedia.org/wiki/Message_Passing_Interface).

## Example Code (1)

```
/*  
  "Hello World" MPI Test Program  
*/  
#include <mpi.h>  
#include <stdio.h>  
#include <string.h>  
  
#define BUFSIZE 128  
#define TAG 0  
  
int main(int argc, char *argv[])  
{  
    char idstr[32];  
    char buff[BUFSIZE];  
    int numprocs;  
    int myid;  
    int i;  
    MPI_Status stat;
```

## Example Code (2)

```
/* all MPI programs start with MPI_Init; all 'N'
 * processes exist thereafter
 */
MPI_Init(&argc,&argv);

/* find out how big the SPMD world is */
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

/* and this processes' rank is what? */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

/* At this point, all programs are running equivalently;
 * the rank distinguishes the roles of the programs in
 * the SPMD model, with rank 0 often used specially...
 */
```

## Example Code (3)

```
if(myid == 0)
{
    printf("%d: We have %d processors\n", myid, numprocs);
    for(i=1; i<numprocs; i++)
    {
        sprintf(buff, "Hello %d! ", i);
        MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,
                  MPI_COMM_WORLD);
    }
    for(i=1; i<numprocs; i++)
    {
        MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
                  MPI_COMM_WORLD, &stat);
        printf("%d: %s\n", myid, buff);
    }
}
```

## Example Code (4)

```
else
{
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG,
             MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strncat(buff, idstr, BUFSIZE-1);
    strncat(buff, "reporting for duty", BUFSIZE-1);
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG,
             MPI_COMM_WORLD);
}

/* MPI Programs end with MPI Finalize; this is a weak
 * synchronization point.
 */
MPI_Finalize();
return 0;
}
```

# Compiling with OpenMPI

```
// Wrappers for gcc (C/C++)
mpicc
mpicxx

// Compiler Flags
OMPI_MPICC_CFLAGS
OMPI_MPICXX_CXXFLAGS

// Linker Flags
OMPI_MPICC_LDFLAGS
OMPI_MPICXX_LDFLAGS
```

OpenMPI does not recommend that you set the flags yourself.  
To see them, try:

```
# Show the flags necessary to compile MPI C applications
shell$ mpicc --showme:compile

# Show the flags necessary to link MPI C applications
shell$ mpicc --showme:link
```

# Compiling and Running

```
mpirun -np <num_processors> <program>  
mpiexec -np <num_processors> <program> # a synonym
```

Starts `num_processors` instances of the program using MPI.

```
jon@riker examples master % mpicc hello_mpi.c  
jon@riker examples master % mpirun -np 8 a.out  
0: We have 8 processors  
0: Hello 1! Processor 1 reporting for duty  
0: Hello 2! Processor 2 reporting for duty  
0: Hello 3! Processor 3 reporting for duty  
0: Hello 4! Processor 4 reporting for duty  
0: Hello 5! Processor 5 reporting for duty  
0: Hello 6! Processor 6 reporting for duty  
0: Hello 7! Processor 7 reporting for duty
```

- By default, MPI uses the lowest-latency communication resource available; shared memory, in this case.



# MPI Matrix Multiplication Example

Highlights of: <http://www.nccs.gov/wp-content/training/mpi-examples/C/matmul.c>.

To compute the matrix product  $AB$ :

- ① Initialize MPI.
- ② If the current process is the master task (task id 0):
  - ① Initialize matrices.
  - ② Send work to each worker task:  
row number (offset); number of rows; row contents from  $A$ ; complete contents of matrix  $B$ .  

```
MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest,  
        mtype, MPI_COMM_WORLD);
```
  - ③ Wait for results from all worker tasks (MPI\_Recv).
  - ④ Print results.
- ③ For all other tasks:
  - ① Receive offset, number of rows, partial matrix  $A$ , and complete matrix  $B$ , using MPI\_Recv:  

```
MPI_Recv(&offset, 1, MPI_INT, MASTER,  
        mtype, MPI_COMM_WORLD, &status);
```
  - ② Do the computation.
  - ③ Send the results back to the sender.

## Other Things MPI Can Do

We can use nodes on a network (by using a `hostfile`).

We can even use MPMD:

- *multiple processes, multiple data*

```
% mpirun -np 2 a.out : -np 2 b.out
```

This launches a single parallel application.

- All in the same `MPI_COMM_WORLD`; but
- Ranks 0 and 1 are instances of `a.out`, and
- Ranks 2 and 3 are instances of `b.out`.

You could also use the `-app` flag with an appfile instead of typing out everything.

# Performance Considerations

Your bottleneck for performance here is message-passing.

Keep the communication to a minimum!

In general, the more machines/farther apart they are, the slower the communication.

Each step from multicore machines to GPU programming to MPI triggers an order-of-magnitude decrease in communication bandwidth and similar increase in latency.

## Part II

# Cloud Computing

# Using a Cluster

Historically:

- find \$\$\$;
- buy and maintain pile of expensive machines.

Not anymore!

We'll talk about Amazon's Elastic Compute Cloud (EC2)  
and principles behind it.

# Evolution of servers

You want a server on the Internet.

- Once upon a time: had to get a physical machine hosted (e.g. in a rack).  
Or, live with inferior shared hosting.
- Virtualization: pay for part of a machine on that rack.  
A win: you're usually not maxing out a computer, and you'd be perfectly happy to share it with others, as long as there are good security guarantees. All users can get root access.
- Clouds enable you to add more machines on-demand.  
Instead of having just one virtual server, spin up dozens (or thousands) of server images when you need more compute capacity.  
Servers typically share persistent storage, also in the cloud.

# Paying for Computes

Cloud computing:

- pay according to the number of machines, or instances, that you've started up.

Providers offer different instance sizes;  
sizes vary according to the  
number of cores, local storage, and memory.

Some instances even have GPUs!

# Launching Instances

Need more computes? Launch an instance!

Input: Virtual Machine image.

Mechanics: use a command-line or web-based tool.

New instance gets an IP address and is network-accessible.  
You have full root access to that instance.



# What to Launch?

Amazon provides public images:

- different Linux distributions;
- Windows Server; and
- OpenSolaris (maybe not anymore?).

You can build an image which contains software you want, including Hadoop and OpenMPI.

# Cleanup

Presumably you don't want to pay forever for your instances.

When you're done with an instance:

- shut it down, stop paying for it.

All data on instance goes away.

# Data Storage

To keep persistent results:

- mount a storage device, also on the cloud (e.g. Amazon Elastic Block Storage); or,
- connect to a database on a persistent server (e.g. Amazon SimpleDB or Relational Database Service); or,
- you can store files on the Web (e.g. Amazon S3).

# Summary

- MPI is a powerful tool for highly parallel computing across multiple machines.
- MPI Programming is similar to a more powerful version of `fork/join`; but you have to manage communication more explicitly.
- Saw cloud computing basics.