COMP/CS 605: Introduction to Parallel Computing
Topic : Distributed Memory Programming: Message Passing Interface

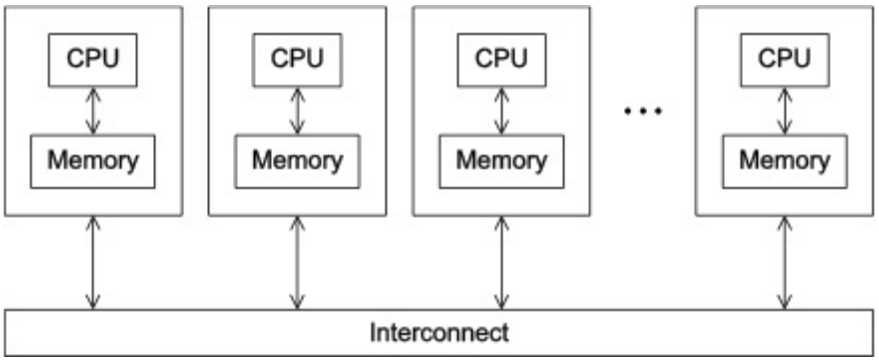
Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

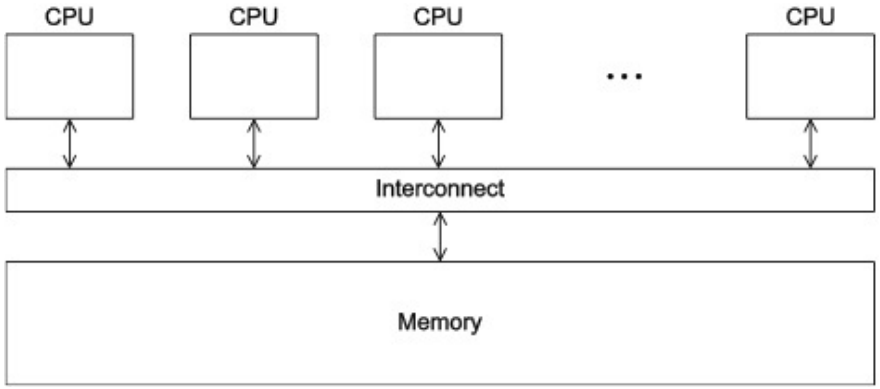
Presented: 02/13/17
Updated: 02/13/17

Table of Contents

Distributed-Memory Programming with MPI



Distributed-memory system: collection of cores, connected with a network, each with its own memory.



Shared-memory system: collection of cores interconnected to a global memory.

An HPC Cluster

A Cluster has multiple, separate nodes, each has multiple cores

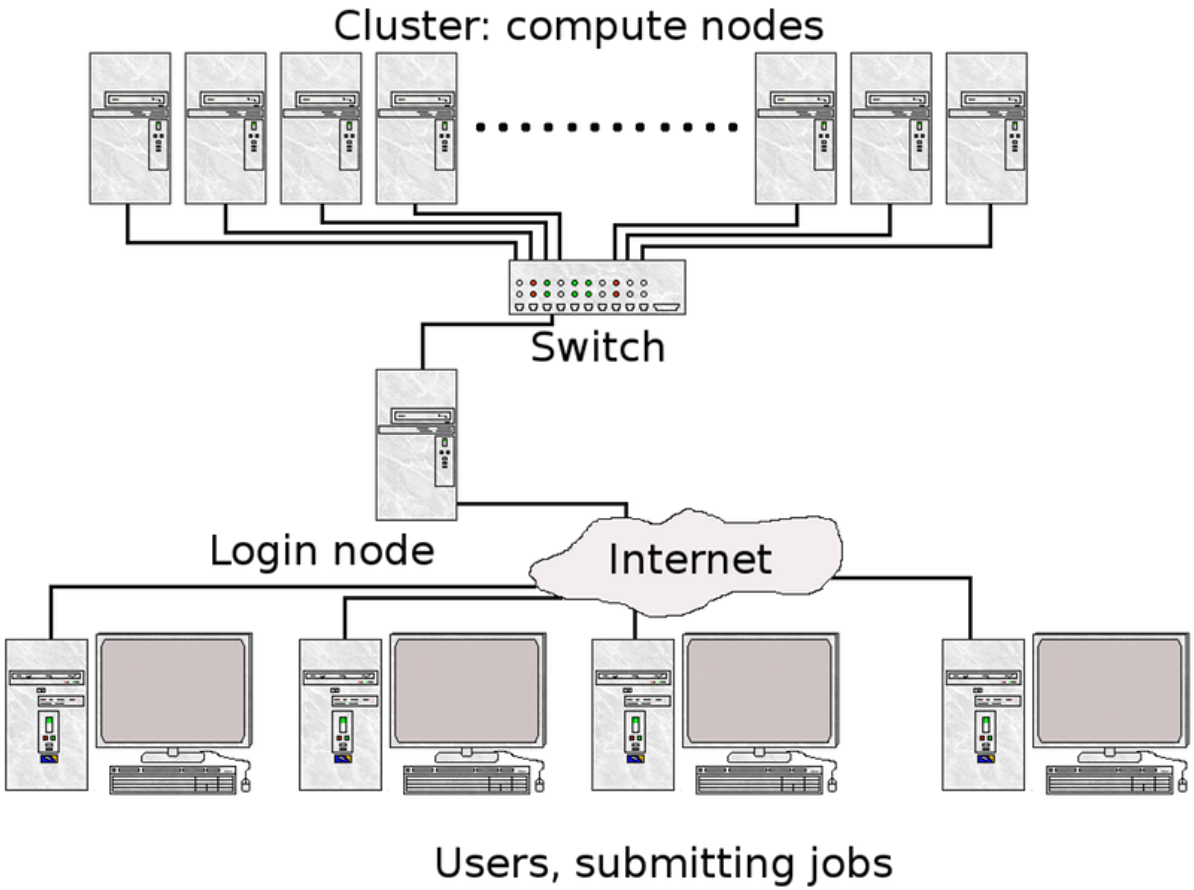


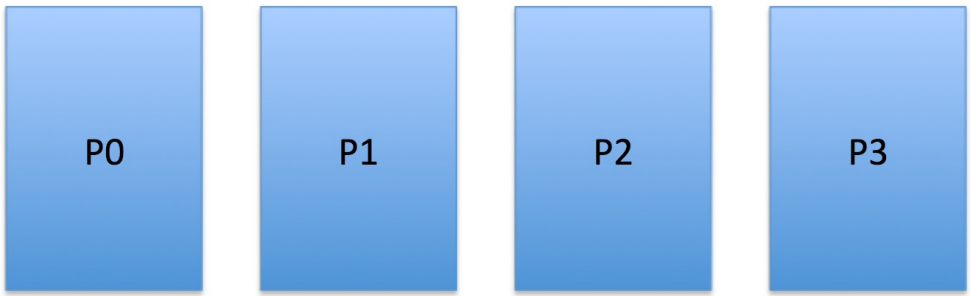
Figure: Diagram of a cluster

Source: `https://www.hpc2n.umu.se/support/beginners_guide`

Student Cluster: tuckoo.sdsu.edu

```
[mthomas@tuckoo:~] date
Fri Feb 12 09:57:36 PST 2016
[mthomas@tuckoo]$ cat /etc/motd

   the cluster system has 11 compute nodes with various CPUs:
    Node name               #Avail Cores   Node Properties**    Got GPUs?
-------------------------------------------------------------------------------
    node1,node2,node3,node4  4ea.          core4, mpi            no
-------------------------------------------------------------------------------
    node6                    6             core6, mpi            no
    node9                    6             core6, mpi            yes
-------------------------------------------------------------------------------
    node5                    8             core8, mpi            no
    node8                    8             core8, mpi            yes
-------------------------------------------------------------------------------
    node7                    12            core12,mpi            yes
-------------------------------------------------------------------------------
    node11                   16            core16,mpi            yes
-------------------------------------------------------------------------------
     **see the output from "pbsnodes -a".

  CPUs & RAM
   ------------------------------------------
   node1 thru node4, Xeon X3360   @ 2.83GHz, 8GB ea.
   node5             Xeon E5420   @ 2.50GHz, 20GB
   node6             Xeon E5-1650 @ 3.20GHz, 64GB
   node7             Xeon X5650   @ 2.67GHz, 48GB
   node8             Xeon E5620   @ 2.40GHz, 48GB
   node9             Xeon E5-1660 @ 3.30GHz, 32GB
   node11            Xeon E5-2650 @ 2.60GHz, 64GB

   GPUs
   ------------------------------------------
   node9  has 2     GTX 480 gpu cards (1.6GB dev ram ea.)
   node8  has 2     C2075   gpu cards (  6GB dev ram ea.)
   node7  has 2     C1060   gpu cards (  4GB dev ram ea.)
   node11 has 1     K40     gpu card  (                 )
```

How does MPI Work?



1. The parallel job is controlled by the *resource manager* on the cluster.
2. On Initialization, MPI assigns P processors (cores) to a global "communicator" group called *MPI_COMM_WORLD*.
3. MPI sets up the MPI environment on each of the P_i cores.
4. MPI launches an identical copy of the *executable* on the P_i cores.
5. Program queries *MPI_COMM_WORLD* to get group information:
   - Number of processes
   - Process ID/Rank

MPI Programming Model

Message Passing Interface

- Written in C (or Fortran, Python, etc.)
- Has main.
- Uses stdio.h, string.h, etc.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with MPI_.
- First letter following underscore is uppercase.
- For function names and MPI-defined types.
- Helps to avoid confusion

Basic MPI Routines

Message Passing Interface

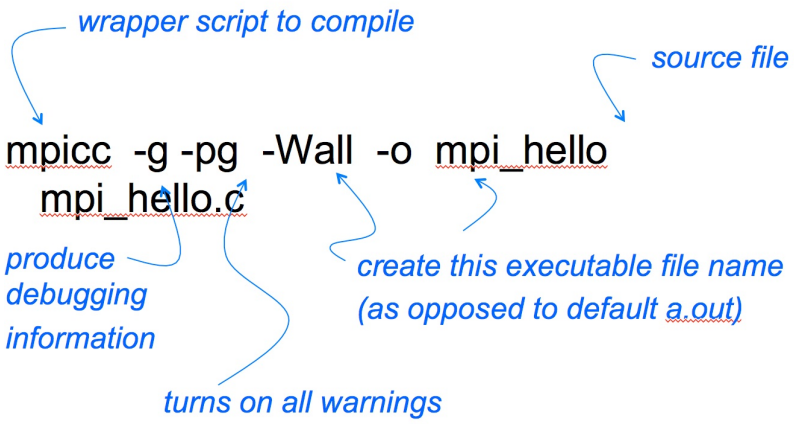
- For running codes on distributed memory systems.
- Data resides on other processes – accessed through MPI calls.
- The minimal set of routines that most parallel codes use:
  - MPI_INIT
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_SEND
  - MPI_RECV
  - MPI_FINALIZE

Serial Hello World

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
     char cptr[100];

   gethostname(cptr,100);
   printf("hello, world from %s\n", cptr);

   return 0;
}
```

MPI Hello World

```
/* File:
 *    mpi_hello.c
 *
 * Purpose:
 *    A "hello,world" program that uses MPI
 *
 * Compile:
 *    mpicc -g -Wall -std=C99 -o mpi_hello mpi_hello.c
 * Usage:
 *    mpiexec -np <number of processes> ./mpi_hello
 *
 * Input:
 *    None
 * Output:
 *    A greeting from each process
 *
 * Algorithm:
 *    Each process sends a message to process 0, which prints
 *    the messages it has received, as well as its own message.
 *
 * IPP:  Section 3.1 (pp. 84 and ff.)
 */
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;
int main(void) {
   char  greeting[MAX_STRING];  /* String storing message */
   int     comm_sz;               /* Number of processes    */
   int     my_rank;               /* My process rank        */
   int     q;

/* Start up MPI */
MPI_Init(NULL, NULL);

/* Get the number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

/* Get my rank among all the processes */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank != 0) {
   /* Create message */
   sprintf(greeting, "Greetings from process %d of %d!",
         my_rank, comm_sz);
   /* Send message to process 0 */
   MPI_Send(greeting, strlen(greeting)+1,
         MPI_CHAR, 0, 0, MPI_COMM_WORLD);
} else {
   /* Print my message */
   printf("Greetings from Master process %d of %d!\n",
               my_rank, comm_sz);
   for (q = 1; q < comm_sz; q++) {
      /* Receive message from process q */
      MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
         0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      /* Print message from process q */
      printf("%s\n", greeting);
   }
}

/* Shut down MPI */
MPI_Finalize();

return 0;
}  /* main */
```




```
batch here
=====================
COMPILE CODE
=====================
[tuckoo]$ mpicc  -g -pg  -Wall  -o  mpi_hello  mpi_hello.c
```

```
=====================
RUN CODE FROM COMMAND LINE
=====================
[mthomas@tuckoo ch3]$ mpirun -np 16 ./mpi_hello
Greetings from process 0 of 16!
Greetings from process 1 of 16!
Greetings from process 2 of 16!
Greetings from process 3 of 16!
Greetings from process 4 of 16!
Greetings from process 5 of 16!
Greetings from process 6 of 16!
Greetings from process 7 of 16!
Greetings from process 8 of 16!
Greetings from process 9 of 16!
Greetings from process 10 of 16!
Greetings from process 11 of 16!
Greetings from process 12 of 16!
Greetings from process 13 of 16!
Greetings from process 14 of 16!
Greetings from process 15 of 16!

[tuckoo]$ mpirun -np 16 --nooversubscribe ./mpi_hello
--------------------------------------------------------------------------
There are not enough slots available in the system to
satisfy the 16 slots that were requested by the application:
  ./mpi_hello

Either request fewer slots for your application, or
make more slots available for use.
--------------------------------------------------------------------------
```

MPI Components

- MPI_Init
  - Tells MPI to do all the necessary setup.

```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

- MPI_Finalize
  - Tells MPI we're done, so clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

Basic Outline

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.
- Called MPI_COMM_WORLD.

Communicators

```
int MPI_Comm_size(
      MPI_Comm   comm          /* in  */,
      int*       comm_sz_p     /* out */);
```

number of processes in the communicator

```
int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

my rank
(the process making this call)

SPMD

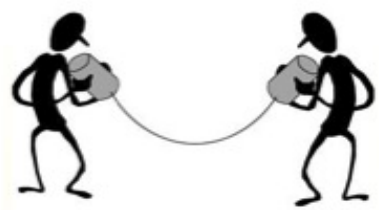
- Single-Program Multiple-Data
- We compile one program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.
- The if-else construct makes our program SPMD.

Data types

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

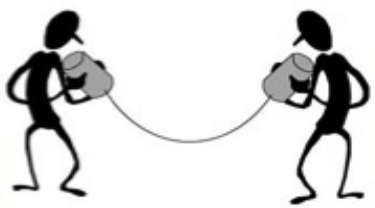
Communication

```
int MPI_Send(
   void*          msg_buf_p     /* in */,
   int            msg_size      /* in */,
   MPI_Datatype   msg_type      /* in */,
   int            dest          /* in */,
   int            tag           /* in */,
   MPI_Comm       communicator  /* in */);
```



Communication

```
int MPI_Recv(
      void*          msg_buf_p      /* out */,
      int            buf_size       /* in  */,
      MPI_Datatype   buf_type       /* in  */,
      int            source         /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* out */);
```



Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

r

MPI_Send
src = q

MPI_Recv
dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

Receiving messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.

status_p argument

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

MPI_Status*

MPI_SOURCE
MPI_TAG
MPI_ERROR

```
MPI_Status* status;

status.MPI_SOURCE
status.MPI_TAG
```

How much data am I receiving?

```
int MPI_Get_count(
      MPI_Status*   status_p   /* in  */,
      MPI_Datatype  type       /* in  */,
      int*          count_p    /* out */);
```

Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Know your implementation; don't make assumptions!

MPI Template (C)

```
#include <stdio.h>
#include "mpi.h"
#include <math.h>
main(int argc, char *argv[]) {
     int        p;
     int        my_rank;
     int        ierr;

     /* start up initial MPI environment */
     MPI_Init(&argc, &argv);

     /* get the number of PE's in the group:  MPI_COMM_WORLD */
     MPI_Comm_size(MPI_COMM_WORLD, &p);

     /* get my rank in the group:  MPI_COMM_WORLD */
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

     /* say hello */
     printf("My rank: PW[%d] out of  %d Total Processors \n",my_rank,p);


      MPI_Finalize();    /* shut down MPI env */
}  /* main */
```

MPI Template (FORTRAN 90)

```
program template
!-- Template for any mpi program
    implicit none    ! highly recommended. It will make
                     ! debugging infinitely easier.
!--Include the mpi header file
    include 'mpif.h'              ! --> Required statement

!--Declare all variables and arrays.
    integer ierr, myid, numprocs, itag, irc

!--Initialize MPI
    call MPI_INIT( ierr )         ! --> Required statement
!--Who am I? --- get my rank=myid
    call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
!--How many processes in the global group?
    call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

!--Finalize MPI
    call MPI_FINALIZE(irc)        ! ---> Required statement
    stop end
```