Introduction to MPI Programming

Claus Aranha

University of Tsukuba
Center for Computational Sciences

Last Edit: February 14, 2016

Outline

- What is the MPI library?
- How to compile it in your program;
- Basic commands and interfaces;
- Sample programs;

What will you learn:

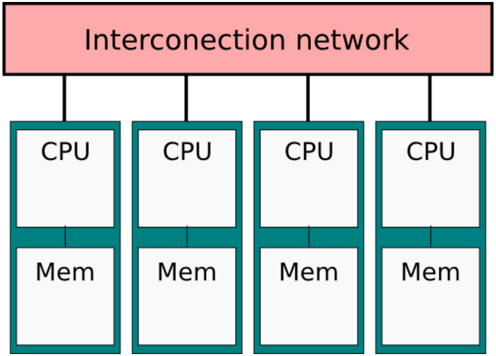
Basic information necessary to using an MPI library, and start self learning from there.

Download the materials from this class (notes and code)

`http://conclave.cs.tsukuba.ac.jp/pub/MPIintro.zip`

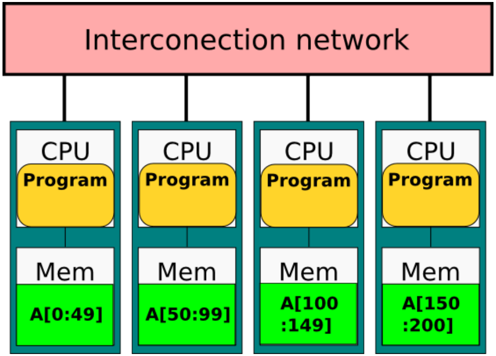
Distributed Memory Machine

- Computers/Nodes, connected by a interconnection network;
- Node: a CPU and memory;
- Each node execute a program, and communicates data through the network.



Single Program, Multiple Data (SPMD)

- Parallel of the same program independently;
- Data is different for each instance;
- Programs interact through message exchange;

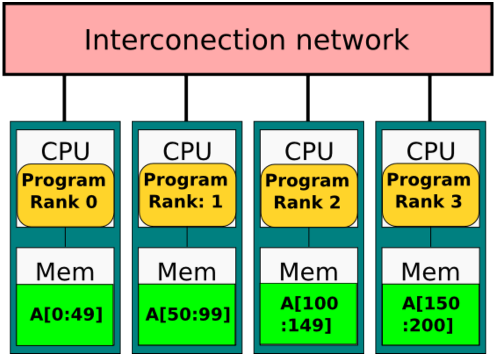


MPI - The Message Passing Interface

- Standard for Message Passing Interface;
- Portable Parallel Library;
- 8 communication modes, collective communication, communication domain, process topology;
- Defines more than 100 iterfaces;
- C, C++, Fortran (and many wrappers);
- Frequent Updates by the MPI Forum: (`http://www.mpi-forum.org`)
  - MPI-1 (1994)
  - MPI-2 (2009)
  - MPI-3 (2014)
  - MPI-4 (in Discussion)

MPI execution model

- Execute the same program on each processor;
- Not synchronous if no communication happens;
- Each process has an ID (rank);



Requirements of an MPI program: code requirements

These are the essential headers for an MPI program in C:

```
#include <mpi.h> /* MPI library headers */
int main(int argc, char *argv[])
{
  MPI_INIT(&argc,&argv)
  /* Must be called before ANY MPI functions */


  MPI_FINALIZE()
  /* Must be called before the end of the program */
}
```

Note that all MPI functions and structure are prefixed by MPI_

Compiling an MPI Program

- specialized compiler: mpicc -l*libs source*
  MPI library itself does not need to be included.
- specialized run: mpiexec -n N *binary*
  N is the number of processes
  For other options, see the man page

MPI libraries

Some packages with MPI implementations on ubuntu 14.04:

- OpenMPI: openmpi-bin, libopenmpi-dev
- MPICH: mpich, libmpich-dev

Program: sample1.c

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

int main(int argc, char *argv[])
{
  int rank,len;
  char name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Get_processor_name(name,&len);
  printf("Processor %03d: %s; reporting!\n", rank, name);

  MPI_Finalize();
  return(0);
}
```

Program: sample1.c

- Each processor runs the same program, but has different rank numbers, and possibly processor names (local memory);
- Rank numbers are defined in relation to the Communicators (COMM_WORLD);
- Use Init and finalize functions around MPI functions;

Communicator (2)

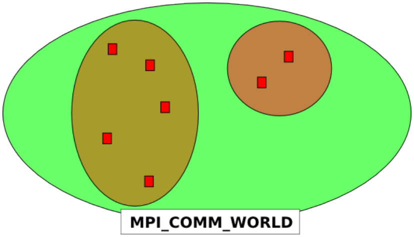
Communication Domain

- Group of Processes;
- May be defined around a topography (1D Ring, 2D Mesh, Torus, etc);
- Stores number of processes, Rank for each process;

MPI_COMM_WORLD: Initial communicator, including all processes;

Communicator (2)

- Communicators define “scopes” for collective communication; e.g.: two thirds of the processes calculate the weather forecast for the current weather forecast; one third compute the initial conditions for the next iteration;
- Intra-communicator and Inter-communicator communication;



Communicator (3)

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Returns the total number of processes in the communicator comm;

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Returns the process rank in the communicator comm;

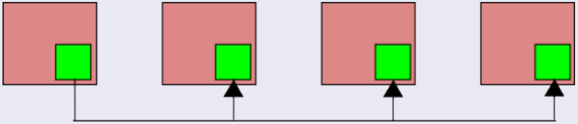
Collective Communication

- Refers to communication that involves all processes (of a communicator);
- Barrier Synchronization (depends on implementation);
- Global Communication: Broadcast, Gather, Scatter, allgather, allscatter, etc;
- Global Reduction: Reduction (sum, maximum, local operator), Prefix computations;

Simple Collective Communication

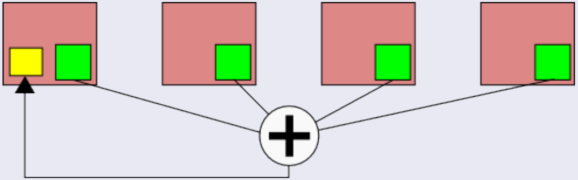
Broadcast

Sends data from one process to all.



Reduce

Gathers and process data from all processes.



- Library optimizes the implementation based on the architecture (no work for the programmer).

MPI_Bcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype type,
              int root, MPI_Comm comm);
```

- returns error value;
- buffer: starting address of data;
- count: number of units in the data;
- type: data type of a data unit;
- root: which rank is sending the data;
- comm: communicator;

MPI_Reduce

```
int MPI_Reduce(void *sendbuf, void* recvbuf, int count,
               MPI_Datatype type, MPI_Op operator,
               int root, MPI_Comm comm);
```

- sendbuf: data to be sent
- recvbuf: resulting data
- operator: reduce operation

Program: sample2.c

Parallel Summation

- Serial Summation:
  $A_1 + A_2 + A_3 + \ldots + A_{999} = \text{sum}$
- Parallel Summation:
  $A_1 + A_2 + A_3 + \ldots + A_{333} = B_1$
  $A_{334} + A_{335} + A_{336} + \ldots + A_{666} = B_2$
  $A_{667} + A_{668} + A_{669} + \ldots + A_{999} = B_3$
  $B_1 + B_2 + B_3 = \text{sum}$

- Each process has a different part of the data
- Calculate its own sum, and put everything together with *Reduce*

Program: sample2.c

```
/**
 * Sample program 2 - sum a large array of numbers
 */

#include <stdio.h>
#include <mpi.h>
#include <string.h>

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

  int local_tsum = 0;
  int local_psum = 0;
  int local_rank = 0;
  int local_data[250];

  int local_i;
```

Program: sample2.c

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&local_rank);

/* Initializing some toy data */
for(local_i = 0; local_i < 250; local_i++)
  local_data[local_i] = local_rank;
/* You would have the real data here */

for(local_i = 0; local_i < 250; local_i++)
  local_psum += local_data[local_i];

MPI_Reduce(&local_psum,&local_tsum,1,MPI_INT,MPI_SUM,
           0,MPI_COMM_WORLD);
```

Program: Sample2.c

```
  printf("Sum for process rank %d: %d\n", local_rank,
         local_psum);
  fflush(0);

  MPI_Barrier(MPI_COMM_WORLD);

  if (local_rank == 0)
    printf("Total Sum: %d\n", local_tsum);

  MPI_Finalize();
  return(0);
}
```

Program: sample3.c

Pi Calculation

- Use the Riemann Sum;
  $\sum_0^1 \frac{4}{1+t^2} \Delta t$
- Choose n (number of divide parts) and broadcast;
- Each processor calculates a number of parts;
  p1: 1,4,7,10...n;
  p2: 2,5,8,11...n+1;
  p3: 3,6,9,12...n+2;
- Partial sums are reduced;

Hint: Try to execute with N $>$ 64000 to see the speed difference!

Program: sample3.c

```
/**
 * Sample program 3 - calculate PI using the Riemann Sum
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

double f(double a)
{
  return (4.0/(1.0 + a*a));
}
```

Program: sample3.c

```
int main(int argc, char *argv[])
{
  int n = 0, myid, nprocs, i;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x;
  double startwtime = 0.0, endwtime;

  int namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Get_processor_name(processor_name, &namelen);

  if (argc > 1)
    n = atoi(argv[1]);
  startwtime = MPI_Wtime();
```

Program: sample3.c

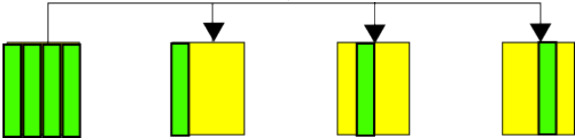
```
/* broadcast 'n' */
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n <= 0)
  {
    fprintf(stderr, "usage: %s <#partition>\n", argv[0]);
    MPI_Abort(MPI_COMM_WORLD,1);
  }

/* calculate each part of pi */
h = 1.0/n;
sum = 0.0;
for (i = myid+1; i <= n; i+= nprocs)
  {
    x = h * (i-0.5);
    sum += f(x);
  }
mypi = h * sum;
```

Program: sample3.c

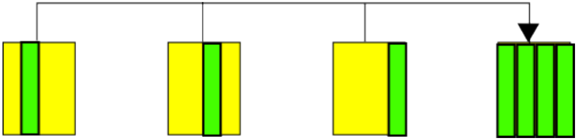
```
  /* sum up each part of pi */
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);

  if (myid == 0)
    {
      printf("pi is approximately %.16f, Error is %.16f\n",
     pi, fabs(pi - PI25DT));
      endwtime = MPI_Wtime();
      printf("wall clock time = %f\n",
             endwtime - startwtime);
    }
  MPI_Finalize();
  return(0);
}
```

A few more collective communication functions

- scatter: split the data into subarrays and send each part to a process.



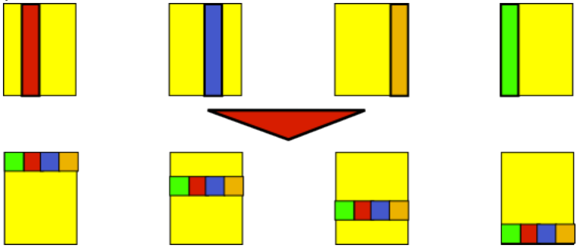
- gather: get split data from each process and put it together in an array.



A few more collective communication functions

"All-*" functions.

- allreduce: result of reduce operation is sent to all processes
- allgather: split data is put together and sent to all processes
- allscatter, alltoall: data at each process is split and sent to other processes



(can be seen as a matrix transformation of 2D data);

Point to Point communication

Data Transfer Between Two Processes

- Process A sends some data to process B (send);
- Process B receives the data from process A (recv);

Point to Point communication (2)

- Data Type specified in MPI Call;
  Basic type, array, structure, vector, user-defined;
  (MPI_INT, MPI_DOUBLE,..);

- The send/receive pair is specified by:
  Communicator, message tag, source rank, destination rank;
  (can use some wildcards: MPI_ANY_TAG, MPI_ANY_SOURCE, etc);

Blocking and Non-Blocking Communication

Blocking

- Send(A...) returns when send buffer can be re-used;
  But the message isn't transmitted yet.
- Receive returns when the receive buffer is available for use;

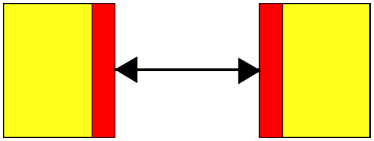
Non-Blocking

- Send and Receive are separated into "post" and "complete";
- Enables overlapping of computation and communication;

Note that it is perfectly legal to mix both!

Blocking and Non-Blocking Communication (2)

Message Exchange



Blocking

- MPI_Send(dest,data)
- MPI_Recv(src,data)

If both sides do this at the same time, in a non-buffered mode, may result in a deadlock

Non-Blocking

- MPI_Isend(dst,data,request)
- MPI_Irecv(src,data,request)

  ...
- MPI_Waitall(request)

Message exchange always complete, regardless of comm mode. Portable.

Communication Modes

4 Communication modes dictate the behavior of send and receive:

- Standard mode: MPI decides if the message should be buffered or not. Can't assume it is buffered!
- Buffered mode: Outgoing message is always buffered. Send operation guaranteed to be local;
  User must guarantee the buffer!
- Synchronous mode: Send completes only if a matching receive is posted, and data begins transfering.
  Send operation guaranteed to be non-local;
- Ready mode: Checks if a receive has been posted before starting send. Sending without a matching receive return error.
  Can remove hand-shake operations;

Large combination of communication modes, blocking and non blocking sends and receives. Check the manual!.

Blocking Communication

MPI_Send

```
MPI_Send(void *data, int count, MPI_Datatype type,
         int dest, int tag, MPI_Comm comm);
```

MPI_Recv

```
MPI_Recv(void *data, int count, MPI_Datatype type,
         int dest, int tag, MPI_Comm comm,
         MPI_Status *status);
```

- Tag - message identifier
- Status - structure with tag, source and other data.

Non Blocking Communication

MPI_Isend

```
MPI_Isend(void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm,
          MPI_Request *request);
```

MPI_Irecv

```
MPI_Irecv(void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm,
          MPI_Request *request);
```

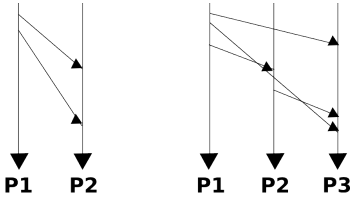
MPI_Wait

```
MPI_Wait(MPI_Request *request, MPI_Status *status);
```

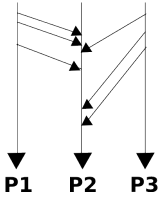
- Request: Contains information about the message received or sent.

Communication Caveats

- Caveat 1: With more than 2 processes, message arrival order is not guaranteed;



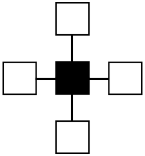
- Caveat 2: Message fairness is not guaranteed either;



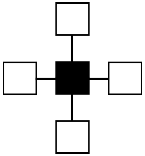
Program: sample4.c

Explicit Solution of Laplace equation

- update by averaging data of four neighbor cells;
- two arrays with old and new values;
- segmentation by region;
- compute residual to check for convergence;

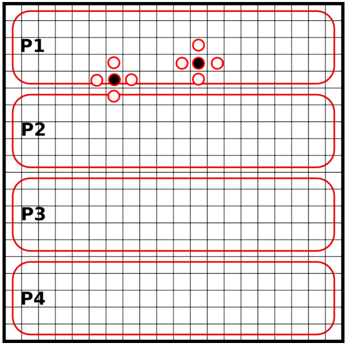


Program: sample4.c



Explicit Solution of Laplace equation

- Block distribution of the 2D data;
- Boundary elements require neighborhood communication;
- Data exchange between boundary elements (Arrays);



Process Topology

MPI can create a virtual spatial relationship between communicators (above, below, etc), called a topology.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int *dims, int *periods, int reorder,
                    MPI_Comm *comm_cart);
```

- ndims: number of dimensions in the grid;
- *dims: size for each dimension;
- *periods: whether each dimension wraps around;
- reorder: if the ranks can be reordered for optimization;

Process Topology

To calculate the *dest*, we use a Cart_shift function.

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int disp, int *rank_source,
                   int *rank_dest);
```

- comm: Holds the grid topology;
- direction: dimension of the shift (horizontal, vertical, etc);
- disp: size of the shift;
- Returns MPI_PROC_NULL rank if boundary is non-periodical, and tries to go to an invalid rank;

Program: sample4.c

```
/**
 * Sample program 4 - laplace equation with explicit method
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

/* Square Region */

#define XSIZE 256
#define YSIZE 256
#define PI 3.1415927
#define NITER 10000

double u[XSIZE+2][YSIZE+2], uu[XSIZE+2][YSIZE+2];
/* 2d target domain, uu is used for new values */
```

Program: sample4.c

```
double time1, time2;
void lap_solve(MPI_Comm);
int myid, numprocs;
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int xsize;

void initialize()
{
  int x,y;

  /* initialization */
  for (x=1; x < XSIZE + 1; x++)
    for (y=1; y < YSIZE + 1; y++)
      u[x][y] = sin((x - 1.0)/XSIZE*PI) +
cos((y-1.0)/YSIZE*PI);
```

Program: sample4.c

```
   /* zero clear in the boundary */
   for (x = 0; x < XSIZE + 2; x++)
     {
       u[x][0] = u[x][YSIZE+1] = 0.0;
       uu[x][0] = uu[x][YSIZE+1] = 0.0;
     }
   for (y = 0; y < YSIZE + 2; y++)
     {
       u[0][y] = u[XSIZE+1][y] = 0.0;
       uu[0][y] = uu[XSIZE+1][y] = 0.0;
     }
}
#define TAG_1 100
#define TAG_2 101
#ifndef FALSE
#define FALSE 0
#endif
```

Program: sample4.c

```
void lap_solve(MPI_Comm comm)
{
  int x,y,k;
  double sum, t_sum;
  int x_start, x_end;
  MPI_Request req1, req2;
  MPI_Status status1, status2;
  MPI_Comm comm1d;
  int down, up;
  int periods[1] = { FALSE };

  /*  Create one dimensional cartesian topology with  non
   *  periodical boundary */
  MPI_Cart_create(comm, 1, &numprocs, periods,
                  FALSE, &comm1d);
  /* calculate process ranks for 'down' and 'up' */
  MPI_Cart_shift(comm1d,0,1,&down,&up);
```

Program: sample4.c

```
x_start = 1 + xsize*myid;
x_end = 1 + xsize*(myid+1);

for (k = 0; k < NITER; k++)
{ /* old <- new */
  for (x = x_start; x < x_end; x++)
    for (y = 1; y < YSIZE+1; y++)
      uu[x][y] = u[x][y];

  /* recv from down */
  MPI_Irecv(&uu[x_start-1][1], YSIZE, MPI_DOUBLE,down,
                TAG_1, comm1d, &req1);
  /* recv from up */
  MPI_Irecv(&uu[x_end][1], YSIZE, MPI_DOUBLE,up,TAG_2,
    comm1d,&req2);
```

Program: sample4.c

```
    /* send to down */
    MPI_Send(&u[x_start][1],YSIZE,MPI_DOUBLE,down,TAG_2,
     comm1d);
    /* send to up */
    MPI_Send(&u[x_end -1][1],YSIZE,MPI_DOUBLE,up,TAG_1,
     comm1d);

    MPI_Wait(&req1,&status1);
    MPI_Wait(&req2,&status2);

    /* update */
    for (x = x_start; x < x_end; x++)
      for (y = 1; y < YSIZE + 1; y++)
        u[x][y] = .25*(uu[x-1][y] + uu[x+1][y] +
                       uu[x][y-1] + uu[x][y+1]);
  }
```

Program: sample4.c

```
  /* check sum */
  sum = 0.0;
  for (x = x_start; x < x_end; x++)
    for (y = 1; y < YSIZE + 1; y++)
      sum += uu[x][y] - u[x][y];
  MPI_Reduce(&sum, &t_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
      comm1d);
  if (myid == 0) printf("summ = %g\n", t_sum);
  MPI_Comm_free(&comm1d);
}
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Get_processor_name(processor_name,&namelen);
  fprintf(stderr,"Process %d on %s reporting for duty.\n",
  myid, processor_name);
```

Program: sample4.c

```
  xsize = XSIZE/numprocs;
  if ((XSIZE % numprocs)!=0)
    MPI_Abort(MPI_COMM_WORLD,1);

  initialize();
  MPI_Barrier(MPI_COMM_WORLD);

  time1 = MPI_Wtime();
  lap_solve(MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);
  time2 = MPI_Wtime();

  if (myid == 0)
    printf("time = %g\n", time2 - time1);
  MPI_Finalize();

  return(0);
}
```

Things to Improve

- This program allocates the whole array, althought it is not necessary
  - When a partial array is allocated, take care about computing the local and the global indexes.
  - This is essential for large scale problems, using a distributed memory machine.
- Two dimensional distribution of 2D array is more efficient than one dimensional distribution
  - Reduces the communication size
  - Can be parallelized with a larger number of processors

Open Source MPI

- OpenMPI:
  http://www.open-mpi.org/
- MPICH2:
  http://www-unix.mcs.anl.gov/mpi/mpich2/
- YAMPII:
  http://ww.il.is.s.u-tokyo.ac.jp/yampii/