1

INTRODUCTION TO MPI PROGRAMMING

Module HPC - ED SPI 2020-2021

Présentée par Jian-Jin Li

2

MPI – Message Passing Interface

- Open standard interface to write parallel programs
- Distributed memory paradigm
- Support distributed memory parallel machine, homogenous or heterogenous cluster, …
- Primarily addresses the message-passing programming model
  - portability
  - flexibility
  - efficiency

3

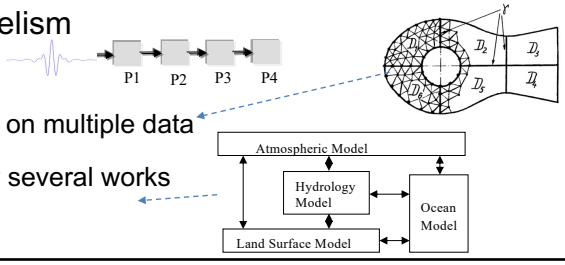
References

- W. Gropp, E. Lusk and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 3rd edition, The MIT Press, 2014.
- W. Gropp, E. Lusk and R. Thakur, Using Advanced MPI: Modern Features of the Message-Passing Interface, The MIT Press, 2014.
- https://www.open-mpi.org/
- https://www.mpi-forum.org/docs/
- http://www.idris.fr/formations/mpi/, consulted on March 2021.
- B. Barney, Introduction to Parallel Computing, https://computing.llnl.gov/tutorials/parallel_comp, consulted on March 2021.
- V. Eijkhout, Introduction to High-Performance Computing, http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html, consulted on March 2021.

4

Introduction - Parallel computing

- Parallel computing
  - Solve larger problem in less time in using concurrent processes/threads
- Parallelism
  - Parallelism refers to the simultaneous occurrence of events on a computer
  - Levels of parallelism: bit-level, instruction-level, **task-level**, job-level
- Sources of parallelism
  - Work in pipeline
  - Repeat one action on multiple data
  - Do simultaneously several works



5

Introduction - Parallel architectures

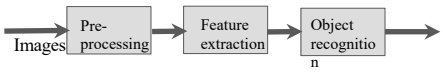
- Classification of Flynn
  - SISD: 1 instruction works on 1 data; serial computer

$$a \times b = axb$$

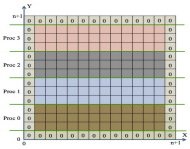
  - SIMD: 1 instruction works on multiple data; vector processor, processor array, GPU

$$X + Y = \begin{matrix} x_0 + y_0 \\ \dots \\ x_3 + y_3 \end{matrix}$$

  - MISD: multiple instructions work in pipeline



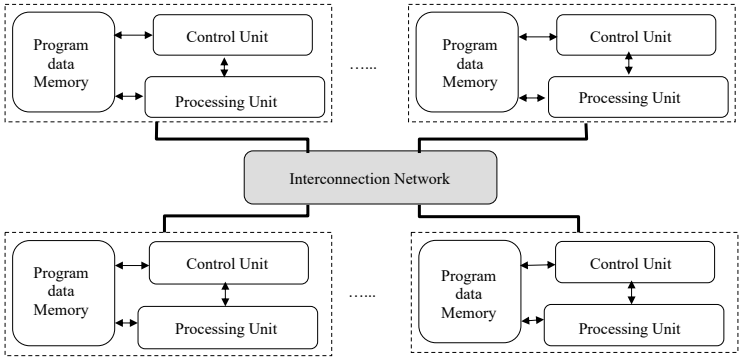
  - **MIMD**: multiple instructions work asynchronously on multiple data



6

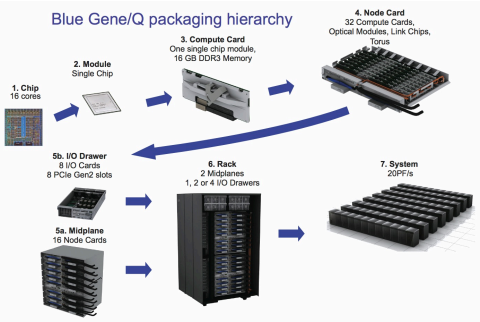
Introduction - Parallel architectures

- MIMD - Multiple Instructions Multiple Data
  - Shared Memory MIMD
  - **Distributed Memory MIMD**



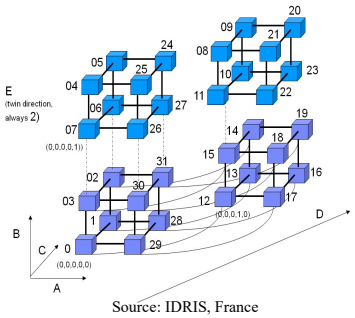
Introduction - Parallel architectures

- Examples - Sequoia (IBM) (N° 1 – June 2012)



Source: LLNL, USA

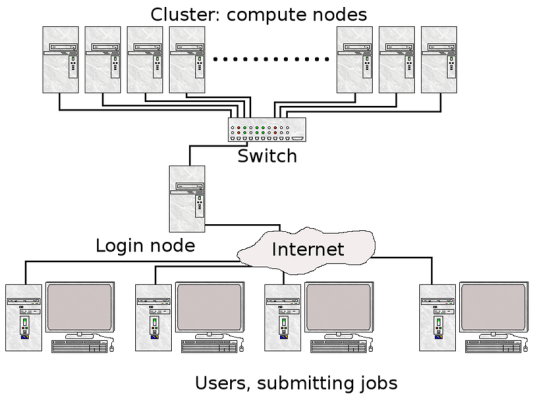
IBM BlueGene/Q



Source: IDRIS, France

Introduction - Parallel architectures

- Cluster



NOW-Cluster (Source : Google)

9

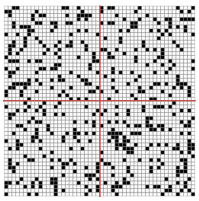
10

MPI - MPI Programming Model

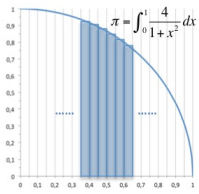
- Group of processes to solve together a problem

4 tasks

4 processes



Conway's Game of Life
(Source: internet)



Calculation of PI by integration

n sub-intervals

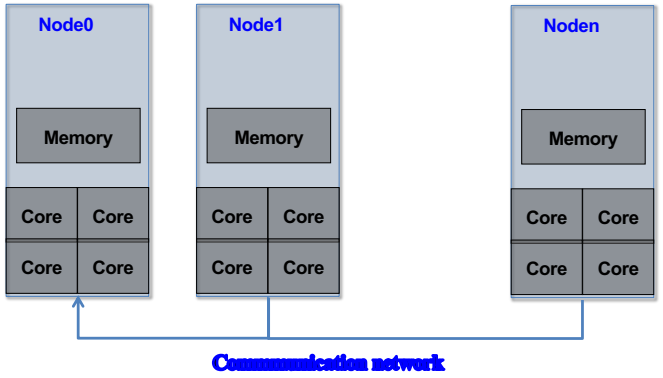
m processes

- New issues
  - Parallel tasks identification; problem decomposition
  - Tasks synchronization, communication, optimization
  - New bug types (deadlock, interference), starvation…

11

MPI - MPI Programming Model

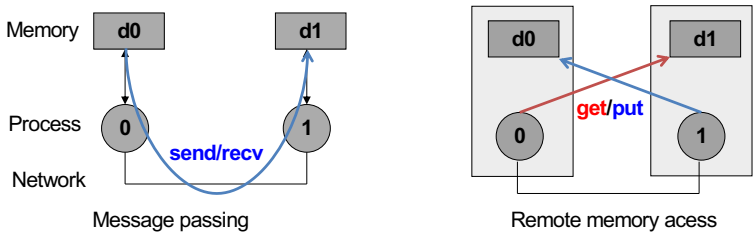
- Distributed memory API



12

MPI - MPI Programming Model

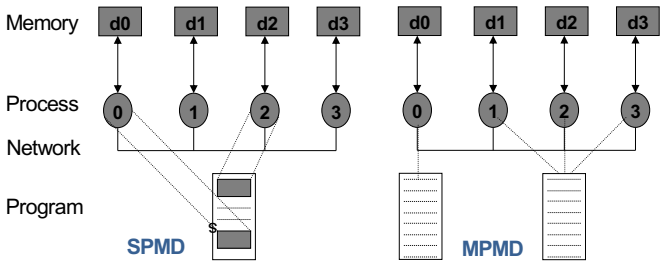
- Communication model
  - Message passing (two sides operation)
  - Remote memory access (one side operation)



13

MPI - MPI Programming Model

- SPMD or MPMD
  - Single / Multiple Program Multiple Data
  - Data distributed over processes
  - Communication between processes via network



14

MPI – Message passing library

- Include:
  - Environment management routines
  - Point to point communication
  - Datatypes management
  - Collective communications
  - Groups of process and communicators
  - Process topologies
  - Parallel I/O, RMA, dynamic process (MPI-2)
  - Non-blocking collective communication, RMA improvement, parallel programming environment (MPI-3)
  - Version of MPI with OpenMPI: `$ ompi_info`

to use with C, C++, Fortran

MPI API: 3.0.0 sur frontalhpc

15

MPI - Environment Management Routines

- hello_mpi.c

```
#include <stdio.h>
#include "mpi.h"          MPI's header file

int main(int argc, char **argv)
{
    int myrank, nbprocs;    ! one copy per process

    MPI_Init( &argc, &argv );          Execution environment initialization
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Comm_size( MPI_COMM_WORLD, &nbprocs);

    printf( " Hello from proc. %d/%d\n ",
            myrank, nbprocs);

    MPI_Finalize(); End of MPI execution
    return 0;
}
```

16

MPI - Environment Management Routines

- Get the processor name and its length

```
int MPI_Get_processor_name(char *name, int resultlength);
```

- Get the ellapsed wall clock time in seconds

```
double MPI_Wtime(void);
```

- Terminates all process of communicator if exception: ex. after `malloc()`

```
int MPI_Abort(MPI_Comm comm, int errorcode);
```

17

MPI – Environment Management Routines

- hello_mpi.c with processor's information — **lab work 1**

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

int main(int argc, char **argv)
{
   int  myrank=-1, nbprocs=0, nameLength=0;
   int  cpuId=-1; /* Number of core used */
   char proc_name[MPI_MAX_PROCESSOR_NAME]; /* name of node used */

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   MPI_Comm_size( MPI_COMM_WORLD, &nbprocs );

   MPI_Get_processor_name( proc_name, &nameLength) ;
   cpuID = sched_getcpu();
   printf( " Hello from proc. %d/%d on CPU %d of %s\n ",
           myrang, nbprocs, cpuID, proc_name);

   MPI_Finalize(); return 0;
}
```

18

MPI – Compiling and running MPI applications

- Compiling — **lab work 1**
  - Compiling command

```
$ mpicc hello_mpi.c -o hello_mpi   # ( +compiling options )
```

  - Before, add following lines in your .bashrc

```
if ! (which mpicc>/dev/null 2>&1) && [ -d /usr/lib64/openmpi ]
then
    export PATH=/usr/lib64/openmpi/bin:$PATH
    export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64/openmpi/lib
fi
```

MPI – Compiling and running MPI applications

- Running

lab work 1

```
$ sbatch submit_hello_mpi.sh
```

```
#!/bin/bash
# submit_hello_mpi.sh execution script

#SBATCH --partition=court     # execution partition 'court'
#SBATCH --ntasks=8            # 8 tasks
#SBATCH -cpus-per-task=1      # de 1 thread
#SBATCH --ntasks-per-core=1   # 1 task par core
#SBATCH --job-name=hello_mpi

#execution
mpiexec ./hello_mpi
```


MPI – Point-to-point communication

lab work 2

- Communication between process 0 and others: p02all.c

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

int main(int argc, char **argv)
{
   int        myRank, nbProcs, cpuId=-1, src, tag=50, nameLength;
   char       message[100], procName[MPI_MAX_PROCESSOR_NAME];
   MPI_Status status;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
   MPI_Comm_size( MPI_COMM_WORLD, &nbProcs);

   MPI_Get_processor_name(procName, &nameLength);
   cpuId = sched_getcpu();
```

21

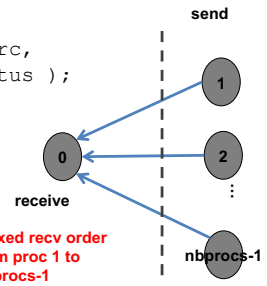
MPI – Point-to-point communication

lab work 2

- Communication between process 0 and others: p02all.c

```
   if ( myRank != 0 ) {
      sprintf( message, "Hello from %d on node %s-cpu%d!",
               myRank, procName, cpuId );
      MPI_Send( message, strlen(message)+1, MPI_CHAR,
                0, tag, MPI_COMM_WORLD ); }
   else
      for ( src=1; src<nbProcs; src++ ) {
          MPI_Recv( message, 100, MPI_CHAR, src,
                    tag, MPI_COMM_WORLD, &status );
          printf( "%s\n", message );
      }

   MPI_Finalize();
   return 0;
}
```

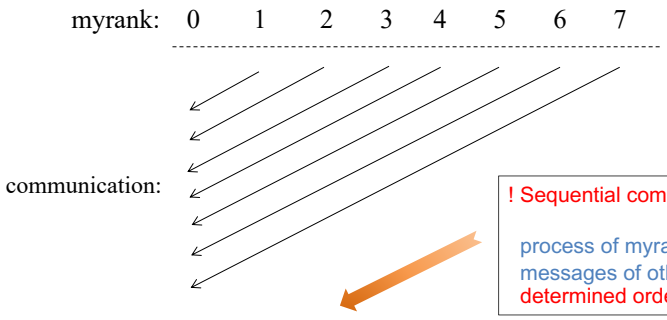


! Fixed recv order from proc 1 to nbprocs-1

22

MPI – Point-to-point communication

- Running of p02all.c with 8 processes



! Sequential communication !

process of myrank=0 receives the messages of others processes in a determined order.

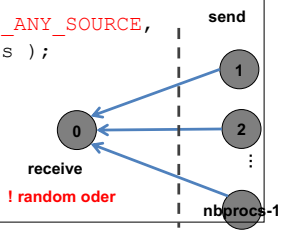
Loss of time if the messages were not sent/arrived in that order

MPI – Point-to-point communication

- Modify the reception order of messages **lab work 2**

```
   if ( myrank != 0 ) {
      sprintf( message, "Hello from %d on %s-cpu%d!",
               myrank, procName, cpuid );
      MPI_Send( message, strlen(message)+1, MPI_CHAR,
                0, tag, MPI_COMM_WORLD ); }
   else
      for ( src=1; src<nbprocs; src++ ) {
          MPI_Recv( message, 100, MPI_CHAR, MPI_ANY_SOURCE,
                    tag, MPI_COMM_WORLD, &status );
          printf( "%s\n", message );
      }

   MPI_Finalize();
   return 0;
}
```

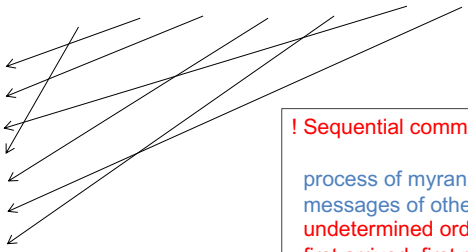


MPI – Point-to-point communication

- Run the program with 8 processes

myrank: 0 1 2 3 4 5 6 7

communication:



! Sequential communication !

process of myrank=0 receives the messages of others processes in a undetermined order.
first arrived, first received

25

MPI – Point-to-point communication

- Two side operation
  - A Send must be matched by a Recv
- Safe program in blocking communication

```
MPI_Comm_rank (MPI_COMM_WORLD, myrank);
if (myrank == 0) {
   MPI_Send( sendbuf, count, MPI_INT, 1, tag1, MPI_COMM_WORLD );
   MPI_Recv( recvbuf, count, MPI_INT, 1,
             tag2, MPI_COMM_WORLD,  &status );
}
else if (myrank == 1) {
   MPI_Recv( recvbuf, count, MPI_INT, 0,
             tag1, MPI_COMM_WORLD, &status );
   MPI_Send( sendbuf, count, MPI_INT, 0, tag2, MPI_COMM_WORLD );
}
```

possible dead lock if we exchange the order of Recv and Send ! (depending on the implementation of MPI)

26

MPI – Blocking communication

- Features
  - Completion of `MPI_Send` means send variable can be reused
  - Completion of `MPI_Recv` mean receive variable can be read
  - Cause synchronization -> Increase communication time
  - Affect the performance of parallel program
- Solution
  - Non-blocking communication

27

MPI – Non-blocking communication

- Operation in 2 steps
  - Request: `MPI_Isend`, `MPI_Irecv`

```
MPI_Isend(buf, count, datatype, dest,
          tag, comm, &request);
MPI_Irecv(buf, count, datatype, src,
          tag, comm, &request);
```

  - Completion: `MPI_Wait(&request, &status);`
  - Test of completion:

```
MPI_Test(&request, &flag, &status);
```

  - Avoid dead lock
  - Allow communication / computation overlapping
  - Persistent request can be used if many communication

28

MPI – Non-blocking communication

- Example

```
MPI_Comm_rank (MPI_COMM_WORLD, myrank);
if (myrank == 0) {
   MPI_Isend( sendbuf, count, MPI_INT, 1,
              tag, MPI_COMM_WORLD, &request0 );
   MPI_Recv( recvbuf, count, MPI_INT, 1,
             tag, MPI_COMM_WORLD,  &status );
    /* or MPI_Irecv + MPI_Wait */
}
else if (myrank == 1) {
    MPI_Isend( sendbuf, count, MPI_INT, 0,
               tag, MPI_COMM_WORLD, &request1 );
    MPI_Recv( recvbuf, count, MPI_INT, 0,
              tag, MPI_COMM_WORLD, &status );
    /* or MPI_Irecv + MPI_Wait */
}
```

29

30

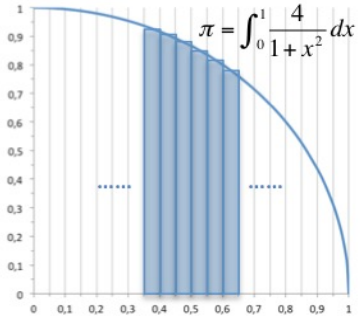
MPI – Point-to-Point Communication

- Exercise: Computing of PI – Numerical integration principle
  - Mathematical formula

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

  - Rectangle rule
    - Let n be the number of rectangles
    - Let h be the width of rectangles
    - We have: $h = \frac{1}{n}$
  - An approximation of $\boldsymbol{\pi}$:

$$\sum_{i=0}^{n-1} h \frac{4}{1+(ih+0{,}5h)^2}$$



31

MPI – Point-to-Point Communication

- Exercise: Sequential computing of PI

```
#include <stdio.h>

int  main(int argc, char **argv)
{
   int    i, nb_rectangles=1000000; // read from keyboard in program
   double x, h, sum=0.0, pi=0.0;

   printf("Please input the number of rectangles of [0-1]: ");
   scanf("%d", &nb_rectangles);

   h = 1.0 / nb_rectangles ;

   for (i=0; i<nb_rectangles; i++) {
      x = (i+0.5)*h;    sum += 4.0 / (1.0 + x*x);
   }

   pi = h * sum ;
   printf("Pi is approximatly: %.16f\n", pi) ;
   return 0;
}
```

32

MPI – Point-to-Point Communication

- Exercise: Parallel computing of PI – Send/Recv

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

int  main(int argc, char **argv)
{
   int    i, nb_rectangles= 1000000, rects_per_proc=0;
   int    my_deb=0, my_end=0;
   double x, h, my_sum=0.0, pi=0.0;

   int        myRank, nbProcs, tag=50;
   MPI_Status status;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
   MPI_Comm_size( MPI_COMM_WORLD, &nbProcs);
```

33

MPI – Point-to-Point Communication

- Exercise: Parallel computing of PI – Send/Recv

```
if (myRank==0) { // process of rank 0 does inputs
    printf("Please input the number of rectangles for [0, 1]: ");
    scanf("%d%*c", &nb_rectangles);

    rects_per_proc = nb_rectangles / nbProcs;

    // rank 0 sends rects_per_proc to the other
    for (i=1; i<nbProcs; i++)
        MPI_Send(&rects_per_proc,1,MPI_INT,i,tag,MPI_COMM_WORLD);
}
else // process of rank≠0 receives rects_per_proc from rank 0
    MPI_Recv(&rects_per_proc, 1, MPI_INT, 0, tag,
             MPI_COMM_WORLD, &status);

h = 1.0 / (rects_per_proc*nbProcs);
```

34

MPI – Point-to-Point Communication

- Exercise: Parallel computing of PI – Send/Recv

```
my_deb=myRank*rects_per_proc;  my_end=my_deb+rects_per_proc;
for (i=my_deb; i<my_end; i++) {
   x = (i+0.5)*h;    my_sum += 4.0 / (1.0 + x*x);
}
if (myRank != 0)
   MPI_Send(&my_sum, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
else {  // rank 0
   for (i=1; i<nbprocs; i++) {
      MPI_Recv(&pi, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               tag, MPI_COMM_WORLD, &status);
      my_sum = my_sum + pi;
   }
   pi = my_sum * h; printf("Pi is approximatly %0.16f\n", pi);
}
MPI_Finalize();   return 0;
}
```

35

Exercise: Calculation of PI – Parallel – V1

- Resume
  - The computation of the `sum` is well distributed.
  - Sequential communication involves all processes at the end of program.
    - → May be improved by using collective communication
- To show more collective communication functions, we assume that only process of `rank 0` has the value of `nb_rectangles` at the beginning of the program

36

37

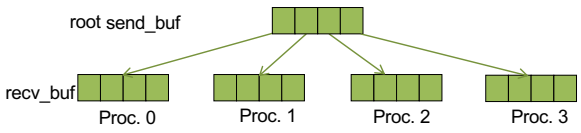
MPI – Collective communication

- What is it?
  - Communication involving all processes of a group
- Objective
  - Increase the performance of parallel program
- How?
  - By reduce of idle processes, decrease the global communication time
- Use cases
  - When I/O
  - Parallel algorithms need collective communication

38

MPI – Collective communication

- Broadcast
  - A process (`root`) has a message to send to others

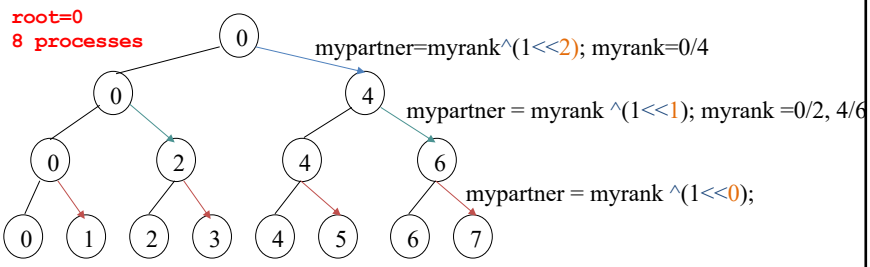



```
int MPI_Bcast(void *msg, int count,
     MPI_Datatype datatype, int root, MIP_Comm comm);
```

39

MPI – Collective communication

- Broadcast: Possible implementation



Bitwise operation:
- bit shift: <<, >>
- XOR: ^
- …

40

MPI – Collective communication

- Broadcast - Broadcast of the dimension of a image

```
int myRank, nbProcs, dims[2], i, tag=30;
MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );
if (myRank == 0){
   /* Fill dims */
   for ( i=1; i<nbProcs; i++)
       MPI_Send( dims, 2, MPI_INT, i, tag, MPI_COMM_WORLD);
}
else
   MPI_Recv(dims, 2, MPI_INT, 0, tag, MPI_COMM_WORLD);
```

Point to point communication:
Number of steps - $O(nbProcs)$

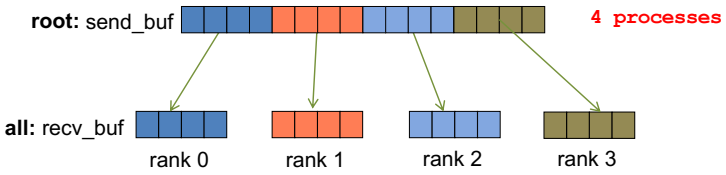
All processes call `MPI_Bcast(dims, 2, MPI_INT, 0, MPI_COMM_WORLD);`

Collective communication:
Number of steps – $O(\log_2(nbProcs))$ with binary tree

41

MPI – Collective communication

- Scatter – Data distribution

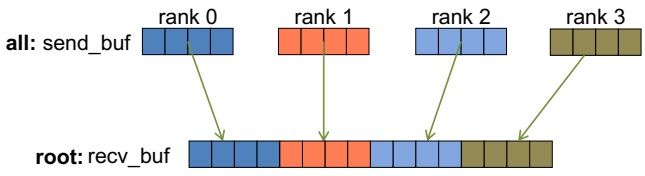



```
int MPI_Scatter( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                 int root, MPI_Comm comm );
```

42

MPI – Collective communication

- Gather – Data fusion

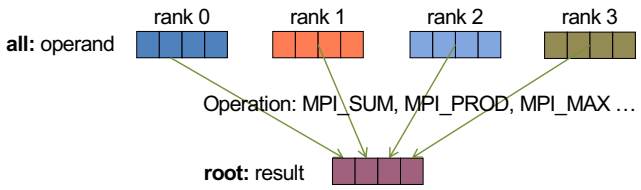



```
int MPI_Gather( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm );
```

43

MPI – Collective communication

- Reduce – Gather + operation on data




```
int MPI_Reduce( void *operand, void *result,
      int count, MPI_Datatype datatype, MIP_Op op,
      int root, MPI_Comm comm );
```

44

MPI – Collective communication

- Exercise: Parallel computing of PI – Collective comm.

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

int  main(int argc, char **argv)
{
   int    i, nb_rectangles=1000000, rects_per_proc=0;
   int    my_deb=0, my_end=0;
   double x, h, my_sum=0.0, pi=0.0;

   int    myRank, nbProcs;


   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
   MPI_Comm_size( MPI_COMM_WORLD, &nbProcs);
```

45

MPI – Collective communication

- Exercise: Parallel computing of PI – Collective comm.

```
if (myRank==0) { // process of rank 0 does inputs
   printf("Please input the number of rectangles for [0, 1]: ");
   scanf("%d%*c", &nb_rectangles);

   rects_per_proc = nb_rectangles / nbProcs;
}

MPI_Bcast(&rects_per_proc, 1, MPI_INT, 0, MPI_COMM_WORLD);

h = 1.0 / (rects_per_proc*nbProcs);
```

46

MPI – Collective communication

- Exercise: Parallel computing of PI – Collective comm.

```
my_deb = myRank * rects_per_proc;
my_end = my_deb + rects_per_proc;

for (i=my_deb; i<my_end; i++) {
   x = (i+0.5) * h;
   my_sum += 4.0 / (1.0 + x*x);
}
my_sum = h * my_sum ;

MPI_Reduce(&my_sum,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

if (myRank==0)
   printf("Pi is approximatly %0.16f\n", pi);

MPI_Finalize();
return 0;
}
```

47

MPI – Collective communication

- Barrier synchronization
  - Make a appointment for all processes
- Use case: time measurement
  - Time measurement for each process

```
double start, exectime;
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
……
exectime= MPI_Wtime()- start;
```

In main function

  - Execution time of program: the one of the slowest process

48

MPI – Time measurement

- MPI program
  - Execution time
    - ☆Time for computation
    - ☆Time for communication
    - ☆Time for processes synchronization & management
  - Execution time measurement: **Wall clock time**
    - ☆Time elapsed between the end and the beginning of a program

49

MPI – Time measurement

- Exercise: Parallel computing of PI

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

int  main(int argc, char **argv)
{
   int    i, nb_rectangles=1000000, rects_per_proc=0;
   int    my_deb=0, my_end=0;
   double x, h, my_sum=0.0, pi=0.0;
   int    myRank, nbProcs;

   double start, execTime;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
   MPI_Comm_size( MPI_COMM_WORLD, &nbProcs);
```

50

MPI – Time measurement

- Exercise: Parallel computing of PI

```
if (myRank==0) { // process of rank 0 does inputs
    printf("Please input the number of rectangles for [0, 1]: ");
    scanf("%d%*c", &nb_rectangles);

    rects_per_proc = nb_rectangles / nbProcs;
 }

 MPI_Barrier(MPI_COMM_WORLD);
 start = MPI_Wtime();

 MPI_Bcast(&rects_per_proc, 1, MPI_INT, 0, MPI_COMM_WORLD);
 …
 MPI_Reduce(&my_sum,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

 execTime= MPI_Wtime()- start;
```

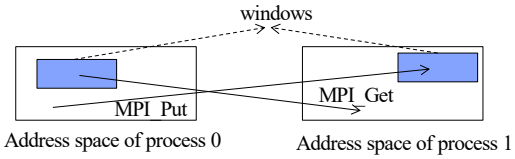
and take the largest one

51

52

MPI – RMA (Remote Memory Access)

- Direct access by one process to parts of the memory of another process
- Software implementation
  - Process 0 put data into process 1's data window
  - Process 1 get data from process 0's data window



53

MPI – RMA (Remote Memory Access)

- Procedure
  - RMA window creation: example with `MPI_Win_Create`
  - Process synchronization: example `MPI_Fence`
  - RMA operations: `MPI_Get`, `MPI_Put`, `MPI_Accumulate`
  - RMA window free: `MPI_Win_free`
- Characteristics
  - Public memory creation
  - Collective operation
  - One process create the window, accessible by all processes

54

MPI – RMA

- Exercise – Calculation of PI

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

int  main(int argc, char **argv)
{
   long    i, n=0, rects_par_proc=0, my_deb=0, my_end=0;
   double  x, h, sum=0.0, pi=0.0;
    int     myrank, nbprocs;
   MPI_Win win_pi, win_n;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   MPI_Comm_size( MPI_COMM_WORLD, &nbprocs);

   /* Lecture of n from keyboard */
```

55

MPI – RMA

- Exercise – Calculation of PI

```
if (myrank==0) {
    MPI_Win_create(&n, sizeof(long), sizeof(long),
        MPI_INFO_NULL, MPI_COMM_WORLD, &win_n);
    MPI_Win_create(&pi, sizeof(double), sizeof(double),
        MPI_INFO_NULL, MPI_COMM_WORLD, &win_pi);
}
else {
    MPI_Win_create(MPI_BOTTOM, 0, sizeof(long),
        MPI_INFO_NULL, MPI_COMM_WORLD, &win_n);
    MPI_Win_create(MPI_BOTTOM, 0, sizeof(double),
        MPI_INFO_NULL, MPI_COMM_WORLD, &win_pi);
}

MPI_Win_fence(0, win_n);
if (myrank != 0)
   MPI_Get(&n, 1, MPI_LONG, 0, 0, 1, MPI_LONG, win_n);
MPI_Win_fence(0, win_n);
```

56

MPI – RMA

- Exercise – Calculation of PI

```
h = 1.0 / n;    rects_par_proc = n / nbprocs;

my_deb = myrank*rects_par_proc;     my_end = deb+rects_par_proc;

for (i=my_deb; i<my_end; i++) {
   x = (i+0.5)*h;     sum += 4.0 / (1.0 + x*x);
}
pi = h * sum ;

MPI_Win_fence(0, win_pi);
if (myrank)
   MPI_Accumulate(&pi, 1, MPI_DOUBLE, 0, 0, 1,
                  MPI_DOUBLE, MPI_SUM, win_pi);
MPI_Win_fence(0, win_pi);

if (myrank==0) printf("Piis approximatly %0.16f\n", pi);

MPI_Finalize();  return 0;
}
```