1

INTRODUCTION TO MPI PROGRAMMING

Module HPC - ED SPI – Mars 2025

Présentée par Jian-Jin Li

1

2

MPI – Message-Passing Interface

- Open standard interface to write parallel programs
- Moves data from the address space of one process to that of another process
- Distributed memory paradigm
- Support distributed memory parallel machine, homogenous or heterogenous cluster, …
- Advantages
  - Practical
  - Portability
  - Flexibility
  - Efficiency

- A library
- To use with a programming language

2

3

MPI – Message-Passing Interface

- History
  - Version 1.0: 1994
  - Version 2.0: 1997
    - Process creation & management
    - Collective communication
    - One-sided communication, parallel I/O
  - Version 3.0: 2012
    - Non-blocking collective communication
    - One-sided communication improvement
  - Version 3.1: 2015
    - Non-blocking collective I/O
  - Version 3.1: 2015
    - Non-blocking collective I/O
  - Version 4.0: 2021
  - Version 4.1: 2023

3

4

References

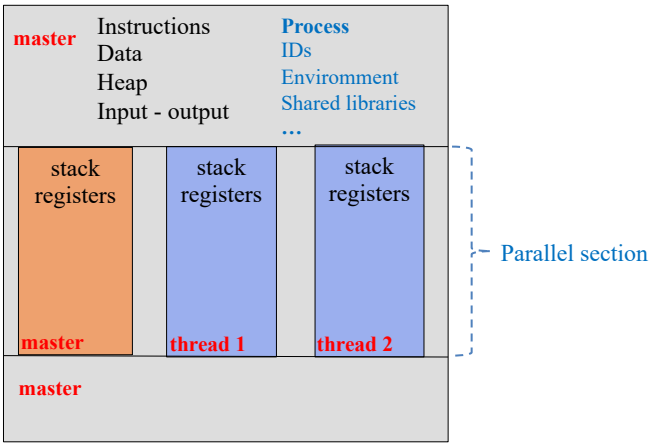
- W. Gropp, E. Lusk and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 3rd edition, The MIT Press, 2014.
- W. Gropp, E. Lusk and R. Thakur, Using Advanced MPI: Modern Features of the Message-Passing Interface, The MIT Press, 2014.
- https://www.open-mpi.org, consulted in January 2025.
- https://www.mpich.org, consulted in January 2025.
- https://www.mpi-forum.org, consulted in January 2025.
- http://www.idris.fr/formations/mpi, consulted in January 2025.
- B. Barney, Introduction to Parallel Computing, https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial, consulted in January 2025.

4

5

OpenMP Programming Model

- Multi-threading

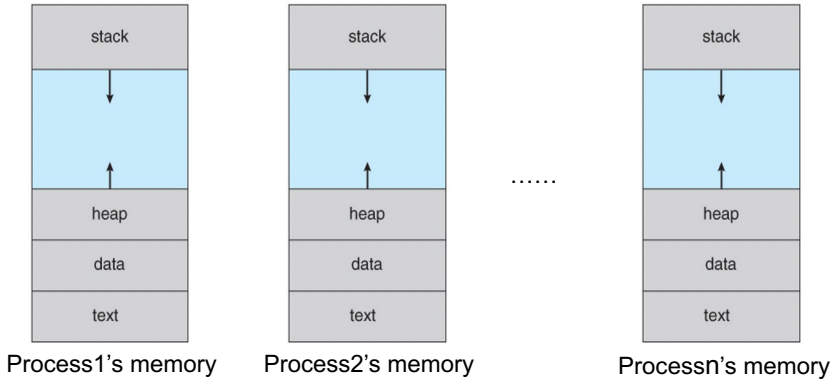


5

6

MPI Programming Model

- Multi-processing



6

Parallel Computers – Classification of Flynn

- Classification depends on
  - Flux of instruction and data

Instruction flux \ Data flux	Single	Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

shared memory

distributed memory

- SISD: 1 instruction works on 1 data; serial computer
- SIMD: 1 instruction works on multiple data; vector processor, processor array, GPU
- MISD: multiple instructions work in pipeline
- MIMD: multiple instructions work asynchronously on multiple data;

7

Top 500 Supercomputer – Jun 2024

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

8

Top 500 Supercomputer – 11/2023

5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
6	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	238.70	304.47	7,404
7	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 40C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR200, EVIDEN EuroHPC/BSC Spain	680,960	138.20	265.57	2,560

9

MPI Programming

- Suitable architectures – Frontier

OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

ABOUT OLCF | OLCF RESOURCES | R&D ACTIVITIES | SCIENCE AT OLCF | FOR USERS | COMMUNITY

Specifications and Features

Compute Node:
1 64-core AMD "Optimized 3rd Gen EPYC" CPU
4 AMD Instinct MI250X GPUs

GPU Architecture:
AMD Instinct MI250X GPUs, each feature 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node

System Interconnect:
4-port HPE Slingshot 200 Gbps (25 GB/s) NICs providing a node-injection bandwidth of 800 Gbps (100 GB/s)

High-Performance Storage:
700 PB HDD+11 PB Flash Performance Tier, 9.4 TB/s and 10 PB Metadata Flash Lustre

Programming Models:
MPI, OpenMP, OpenACC, CUDA, OpenCL, DPC++, HIP, RAJA, Kokkos, and others

Node Performance:
53 TF double precision

System Size:
9,402 nodes

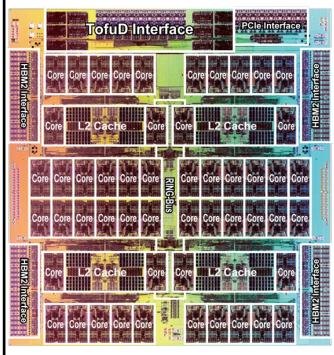
source: internet

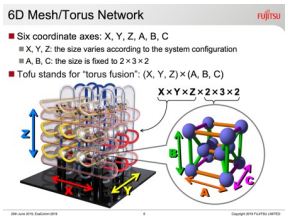
10

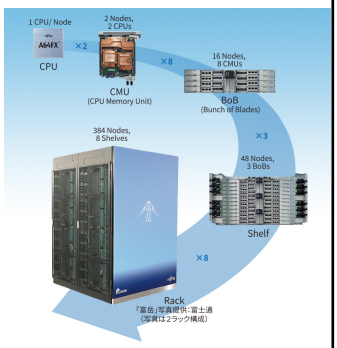
11

MPI Programming

- Suitable architectures – Fugaku
  - **Distributed Memory MIMD**
  - Fujitsu MPI (Based on OpenMPI), MPICH-Tofu (Based on MPICH)





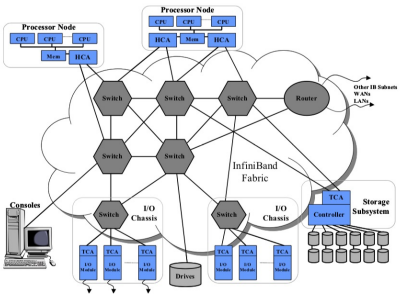


source: internet

12

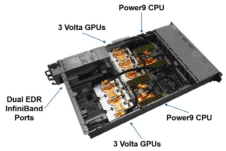
MPI Programming

- Suitable architectures – Summit (TOP 1, 06/2018- 11/2019)
  - **Distributed Memory MIMD**



InfiniBand Fabric

256 cabinets, 18 nodes per cabinet
=> 4608 nodes



1 node:
2 IBM Power9 CPU (22 cores)
6 NVIDIA V100 GPU (640 cores)

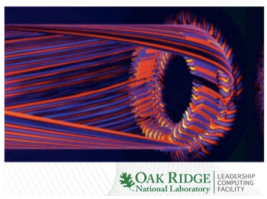
Source: ORNL – Mellanox Technologies

12

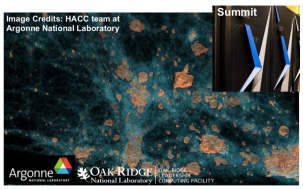
13

MPI Programming

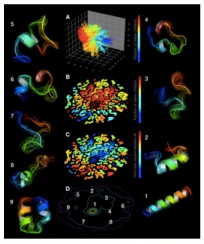
- Suitable architectures – Summit
  - Software
    - OS: RHEL 7.4; Compiler: XLC 13.1, nvcc 9.2
    - Math Lib.: ESSL, CUBLAS 9.2; MPI: Spectrum MPI
    - AI software: PowerAI DDL
  - Applications



Plasma Simulation



Cosmological Survey Simulation

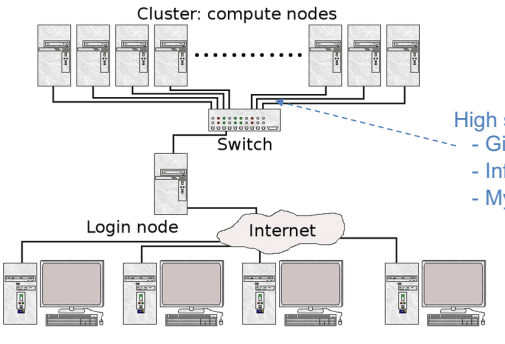


Cancer Surveillance
Source : National Cancer Institute

14

MPI Programming

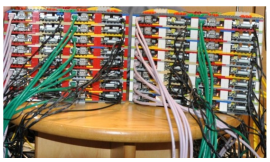
- Suitable architectures
  - **Cluster**



NOW-Cluster (Source : Google)

Beowulf project – NASA 1994

High speed interconnect technologies:
- Gigabit Ethernet
- InfiniBand
- Myrinet



Raspberry Pi Supercomputer
University of Southampton 2012

15

15

16

16

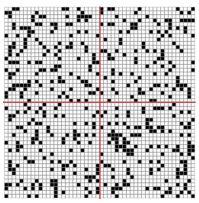
MPI - MPI Programming Model

- Group of processes to solve together a problem

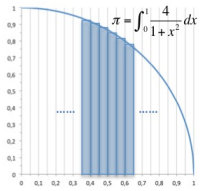
4 sub-domains

4 tasks

4 processes



Conway's Game of Life
(Source: internet)



Calculation of PI by integration

m processes

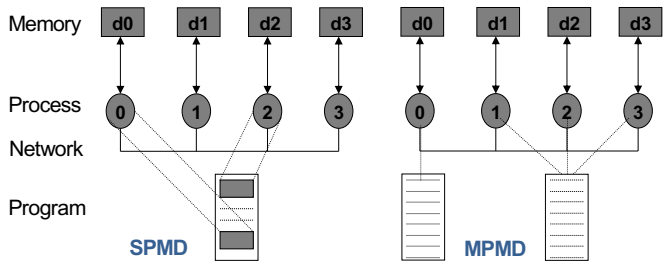
n sub-intervals on *x*

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

17

MPI - MPI Programming Model

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

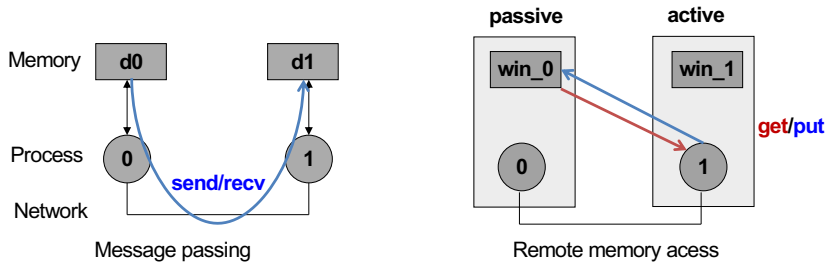


17

18

MPI - MPI Programming Model

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



18

19

MPI – Message passing library

- Include:
  - Environment management routines
  - Point-to-point communication
  - Datatypes management
  - Collective communications
  - Groups of processes and communicators
  - Process topologies
  - Parallel I/O, RMA, dynamic process (MPI-2)
  - Non-blocking collective communication, RMA improvement (MPI-3)

to use with C (C++), Fortran

Version of MPI with OpenMPI: `$ ompi_info`

19

20

MPI – Environment Management Routines --- Examples

- Process enrolment into MPI environment

```
double MPI_Init(int *ptArgc, char ***ptArgv);
```

- Get out of MPI environment

```
double MPI_Finalize(void);
```

- Get the processor name and its length

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

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

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

20

21

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=-1, nbProcs=0;   ! 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);          myRank: 0,1, … nbProcs-1

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

21

22

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 procName[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( procName, &nameLength) ;
   cpuId = sched_getcpu();
   printf( " Hello from proc. %d/%d on CPU %d of %s\n ",
           myRank, nbProcs, cpuId, procName);

   MPI_Finalize(); return 0;
}
```

22

23

MPI – Compiling and running MPI applications

- Connecting to `frontalhpc2020`

```
$ ssh my_account@frontalhpc2020.local.isima.fr
```

- Compiling

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

- Running Running without SLURM Do NOT do it!
  - 8 processes on local node (`frontalhpc2020`)

```
$ mpiexec -np 8 ./hello_mpi
```

23

24

MPI – Compiling and running MPI applications

- Running Running with SLURM

```
$ sbatch submit_hello_mpi.sh
$ more slurm-xxxxxx.out # result is in this file
```

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

#SBATCH --partition=peda     # execution partition 'court'
#SBATCH --ntasks=8            # 8 tasks => 8 processes in parallel
#SBATCH --cpus-per-task=1     # of 1 thread
#SBATCH --ntasks-per-core=1   # 1 task per core
#SBATCH --job-name=hello_mpi

#execution
mpiexec ./hello_mpi
```

24

25

MPI – Point-to-point communication

- Communication between process 0 and process 1: hello_1to0.c

lab work 2

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

int main(int argc, char **argv)
{
   int        myRank, nbProcs, cpuId=-1, tag=50, nameLength;
   char       message[512], 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();
```

25

MPI – Point-to-point communication

- Communication between process 0 and others: hello_1to0.c

```
   if ( myRank == 1 ) { // processes of rank 1
      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 if ( myRank == 0 ) // process of rank 0
      MPI_Recv( message, 512, MPI_CHAR, 1,
                tag, MPI_COMM_WORLD, &status );
      printf( "Proc %d: %s\n", myRank, message );
   }

   MPI_Finalize();
   return 0;
}
```

26

27

MPI – Datatypes for data communication

- Similar to C, examples:

MPI datatype	C datatype
MPI_CHAR	char
MPI_INT	int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	

- Complex data types
  - `MPI_PACKED` : to send data of different types in a packet
  - Derived types: structure, column of matrix …

27

28

MPI – Point-to-point communication

- Communication between process 0 and others: hello_master.c

lab work 3

```
#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[512], 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();
```

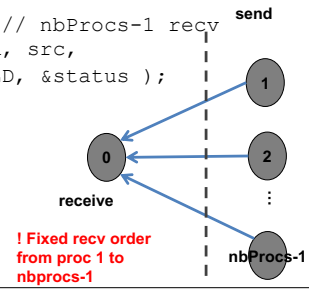
28

29

MPI – Point-to-point communication

- Communication between process 0 and others: hello_master.c

```
   if ( myRank != 0 ) { // processes of rank ≠ 0
      sprintf( message, "Hello from %d on node %s-cpu%d!",
               myRank, procName, cpuId );
      MPI_Send( message, strlen(message)+1, MPI_CHAR,
                0, tag+myRank, MPI_COMM_WORLD ); }
   else // process of rank 0
      for ( src=1; src<nbProcs; src++ ) { // nbProcs-1 recv
          MPI_Recv( message, 512, MPI_CHAR, src,
                    tag+src, MPI_COMM_WORLD, &status );
          printf( "%s\n", message );
      }

   MPI_Finalize();
   return 0;
}
```

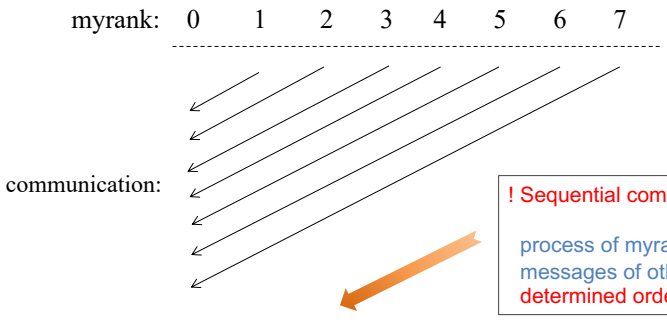


29

30

MPI – Point-to-point communication

- Running of hello_master.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

30

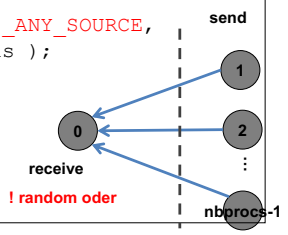
31

MPI – Point-to-point communication

- Modify the reception order of messages

lab work 3

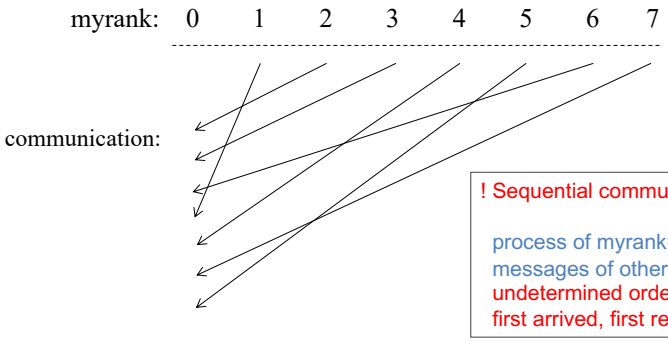
```
   if ( myrank != 0 ) {
      sprintf( message, "Hello from %d on %s-cpu%d!",
               myRank, procName, cpuid );
      MPI_Send( message, strlen(message)+1, MPI_CHAR,
                0, tag, MPI_COMM_WORLD ); }
   else
      for ( src=1; src<nbprocs; src++ ) {
          MPI_Recv( message, 512, MPI_CHAR, MPI_ANY_SOURCE,
                    tag, MPI_COMM_WORLD, &status );
          printf( "%s\n", message );
      }

   MPI_Finalize();
   return 0;
}
```



32

MPI – Point-to-point communication

- Run the program with 8 processes



! Sequential communication !

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

33

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)

33

34

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

34

35

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

35

36

MPI – Non-blocking communication

Modify the programme hello_1to0.c

- Example P0 ←→ P1: hello_to_you.c **nbProcs!**

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

#define MSG_LEN 128

int main( int argc, char **argv ) {
    int myRank=-1, nbProcs=0;
    char sendMsg[MSG_LEN], recvMsg[MSG_LEN];

    MPI_Request requestS, requestR;
    MPI_Status status;

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

    sprintf(sendMsg, "Hello from proc %d/%d!", myRank, nbProcs);
```

36

37

MPI – Non-blocking communication

- Example P0 ←→ P1: hello_to_you.c

```
   if (myRank == 0) {
      MPI_Isend(sendMsg, strlen(sendMsg)+1, MPI_CHAR, 1,
                100, MPI_COMM_WORLD, &requestS);
      MPI_Irecv(recvMsg, MSG_LEN, MPI_CHAR, 1,
                200, MPI_COMM_WORLD, &requestR);
   }
   else if (myRank == 1) {
      MPI_Isend(sendMsg, strlen(sendMsg)+1, MPI_CHAR, 0,
                200, MPI_COMM_WORLD, &requestS);
      MPI_Irecv(recvMsg, MSG_LEN, MPI_CHAR, 0,
                100, MPI_COMM_WORLD, &requestR);
   }
   if (myRank==0 || myRank==1) {
      MPI_Wait(&requestR, &status);
      printf("Proc %d: received message: %s\n", myRank, recvMsg);}
   MPI_Finalize();
   return 0;
}
```

nbProcs!

37

38

MPI – Non-blocking communication

- Example

```
MPI_Comm_rank (MPI_COMM_WORLD, myrank);
if (myrank == 0) {
   MPI_Isend( sendMsg, strlen(sendMsg)+1, MPI_CHAR, 1,
              100, MPI_COMM_WORLD, &requestS );
   MPI_Recv( recvMsg, 128, MPI_CHAR, 1,
             200, MPI_COMM_WORLD, &status );
    /* replace MPI_Irecv + MPI_Wait */
}
else if (myrank == 1) {
    MPI_Isend( sendMsg, strlen(sendMsg)+1, MPI_CHAR, 0,
               200, MPI_COMM_WORLD, &requestS );
    MPI_Recv( recvMsg, 128, MPI_CHAR, 0,
              100, MPI_COMM_WORLD, &status );
    /* replace MPI_Irecv + MPI_Wait */
}
 if (myRank==0 || myRank==1) {
    printf("Proc %d: received message: %s\n", myRank, recvMsg);
}
```

38

39

39

40

40

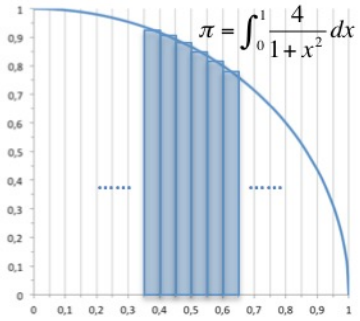
MPI – Point-to-Point Communication

- 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 $\boldsymbol{\pi}$:

$$\sum_{i=0}^{n-1} h\frac{4}{1+(ih+0.5h)^2}$$



41

MPI – Point-to-Point Communication

- Exercise: Sequential computing of PI pi_rectangle_sequential.c

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

int main(int argc, char **argv)
{
        double PI25DGT = 3.14159265358979323846264 3;
        int i, nb_rectangles = 10000;
        double x, sh, sum=0.0, pi=0.0;

        printf("Please input the number of rectangles for [0-1]: "); fflush(stdout);
        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 = sum * h;

        printf("pi is approximately %.16f\n", pi);
        printf("error to PI25DGT is %.16f\n", fabs(pi-PI25DGT));
        return 0;
}
```

42

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= 100000, nbRects_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);
```

43

MPI – Point-to-Point Communication

- Exercise: Parallel computing of PI – Send/Recv

```
if (myRank==0) { // process of rank 0 does inputs
    scanf("%d", &nb_rectangles);

    nbRects_per_proc = nb_rectangles / nbProcs;

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

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

43

44

MPI – Point-to-Point Communication

- Exercise: Parallel computing of PI – Send/Recv

```
   my_deb = myRank*nbRects_per_proc;
   my_end = my_deb+nbRects_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 of rank 1,2,…
      }
      pi = my_sum * h; printf("Pi is approximatly %0.16f\n", pi);
   }
   MPI_Finalize();   return 0;
}
```

44

45

Exercise: Calculation of PI – Send/Recv

- Resume
  - The computation of the `sum` is well distributed.
  - Sequential communication at the beginning and 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

45

46

46

47

MPI – Collective communication

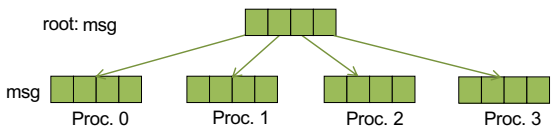
- What is it?
  - Communication involving all processes of a group
- Objective
  - Increase the performance of parallel program -> in reducing the communication time
- How?
  - By reduce of idle processes, decrease the global communication time
- Use cases
  - When I/O
  - Parallel algorithms need collective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

47

48

MPI – Collective communication

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



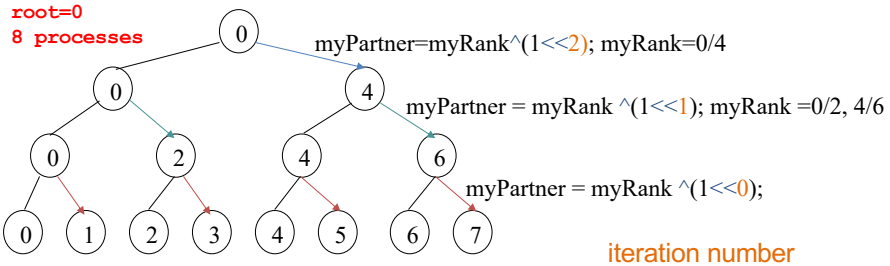

```
int MPI_Bcast(void *msg, int count,
     MPI_Datatype datatype, int root, MPI_Comm comm);
```

48

49

MPI – Collective communication

- Broadcast: Possible implementation



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

49

50

MPI – Collective communication

- Bitwise operation

Operation	Meaning	Example
&	AND	110 & 101 = 100
|	OR	110 | 101 = 111
^	XOR	110 ^ 101 = 011
~	NOT	~110 = 001 (on 3 bits)
<<	left shift	1<<2 = 4
>>	right shift	100>>2 = 1

Logical operators:
&& AND
|| OR
! NOT

50

51

MPI – Collective communication

- Broadcast - Broadcast of nbRects_per_proc

…

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

```
/* suppose nbRects_per_proc is filled by process 0 */
if (myRank == 0){
   for ( i=1; i<nbProcs; i++)
       MPI_Send( &nbRects_per_proc, 1, MPI_INT, i, tag, MPI_COMM_WORLD);
}
else
   MPI_Recv(&nbRects_per_proc, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
```

All processes call

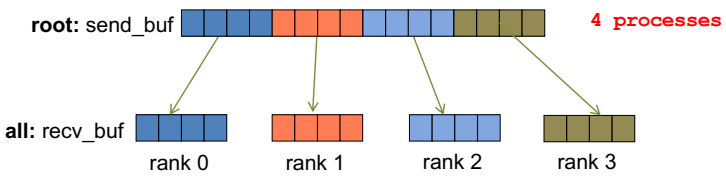
```
MPI_Bcast(&nbRects_per_proc, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

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

52

MPI – Collective communication

- Scatter – Data distribution

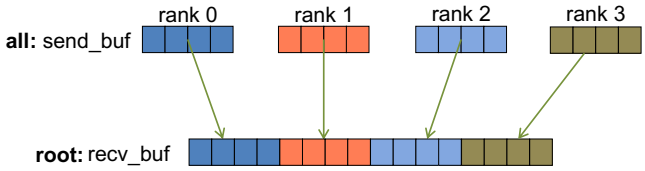



```
int MPI_Scatter( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                 int root, MPI_Comm comm );
```

53

MPI – Collective communication

- Gather – Data fusion



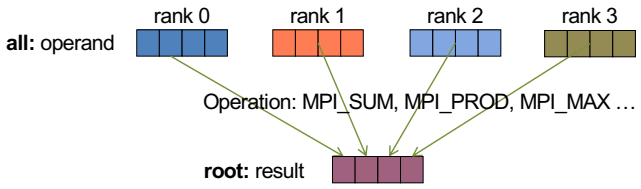

```
int MPI_Gather( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm );
```

53

54

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 );
```

54

55

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, nbRects_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);
```

55

56

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]: ");
   fflush(stdout);
   scanf("%d%", &nb_rectangles);

   nbRects_per_proc = nb_rectangles / nbProcs;
}

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

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

56

57

MPI – Collective communication

- Exercise: Parallel computing of PI – Collective comm.

```
    my_deb = myRank * nbRects_per_proc;
    my_end = my_deb + nbRects_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;
}
```

57

58

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

58

59

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

59

60

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, nbRects_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);
```

60

61

MPI – Time measurement

- Exercise: Parallel computing of PI

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

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

   nbRects_per_proc = nb_rectangles / nbProcs;
}
MPI_Bcast(&nbRects_per_proc, 1, MPI_INT, 0, MPI_COMM_WORLD);
…
MPI_Reduce(&my_sum_y,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

execTime= MPI_Wtime()- start;
printf("Proc %d: execTime=%.12f\n", myRank, execTime);
```

and take the largest one

61

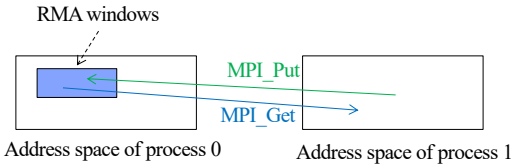
62

62

63

MPI – RMA (Remote Memory Access)

- Decouple data movement with process synchronization
- Direct access by one process to parts of the memory of another process
- Other processes can directly read from / write to the memory
- Example
  - Process 1 put data into process 0's data window
  - Process 1 get data from process 0's data window



63

64

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

64

65

MPI – RMA

- Exercise – Calculation of PI

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

int  main(int argc, char **argv)
{
   long    i, nb_rectangles=0, nbRects_par_proc=0;
   long    my_deb=0, my_end=0;
   double  x, h, my_sum=0.0, pi=0.0;
    int     myrank, nbprocs;
   MPI_Win win_pi, win_nbRects;

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

   /* Lecture of n from keyboard by process of rank 0 */
```

65

66

MPI – RMA

- Exercise – Calculation of PI

```
if (myrank==0) {
    MPI_Win_create(&nb_rectangles, sizeof(long), sizeof(long),
        MPI_INFO_NULL, MPI_COMM_WORLD, &win_nbRects);
    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_nbRects);
    MPI_Win_create(MPI_BOTTOM, 0, sizeof(double),
        MPI_INFO_NULL, MPI_COMM_WORLD, &win_pi);
}

MPI_Win_fence(0, win_nbRects);
if (myrank != 0)
   MPI_Get(&nb_rectangles,1,MPI_LONG,0,0,1,MPI_LONG,win_nbRects);
MPI_Win_fence(0, win_nbRects);
```

66

MPI – RMA

- Exercise – Calculation of PI

```
   h = 1.0/nb_rectangles;  nbRects_par_proc = nb_rectangles/nbprocs;

   my_deb = myrank*nbRects_par_proc;
   my_end = my_deb+nbRects_par_proc;

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

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

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

   MPI_Finalize();  return 0;
}
```