1

# Parallel Computing with MPI

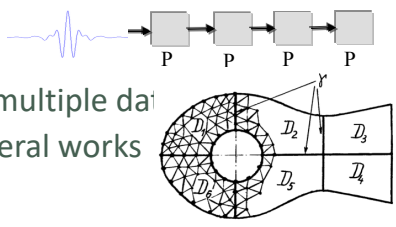
Module HPC - ED SPI 2018-2019

Présentée par Jian-Jin Li

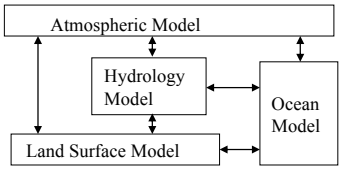
2

## Introduction – Parallel computing & Parallelism

- 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 dat
  - Do simultaneously several works

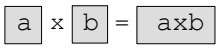




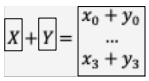



# Introduction - Parallel architectures

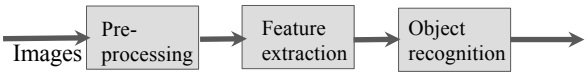
- Classification of Flynn
  - SISD: 1 instruction works on 1 data; serial computer

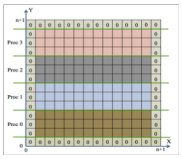



  - SIMD: 1 instruction works on multiple data; vector processor, processor array, GPU




  - MISD: multiple instructions work in pipeline



  - MIMD: multiple instructions work asynchronously on multiple data

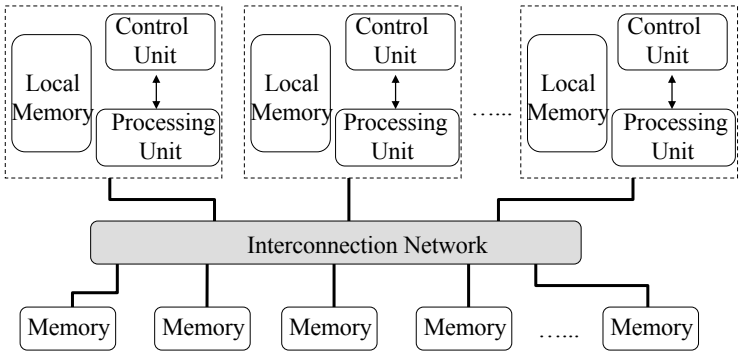


# Introduction - Parallel architectures

- MIMD - Characteristics
  - Asynchronous operation on multiple processors
  - Communication way: Shared or Distributed memory

# Introduction - Parallel architectures

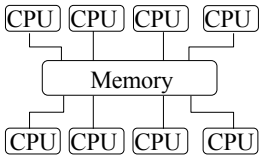
- Shared Memory MIMD
  - Communication way: shared memory





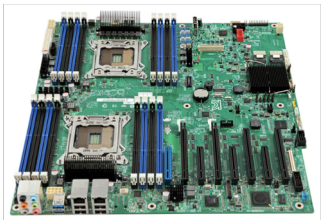

# Introduction - Parallel architectures

- Shared Memory MIMD
  - SMP (Symmetric Multi-Processor)
    - Identical processors (2-128)
    - UMA / NUMA





node27&29:
E5-2670 2.60GHz



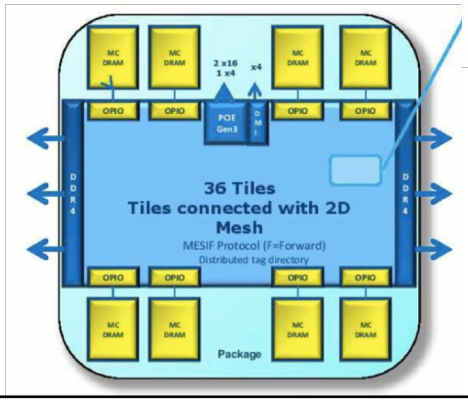
Multi Xeon motherboard

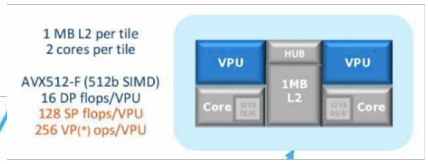
  - Multi-sockets server

7

# Introduction - Parallel architectures

- SMP - example
  - Intel Xeon Phi 72 cores

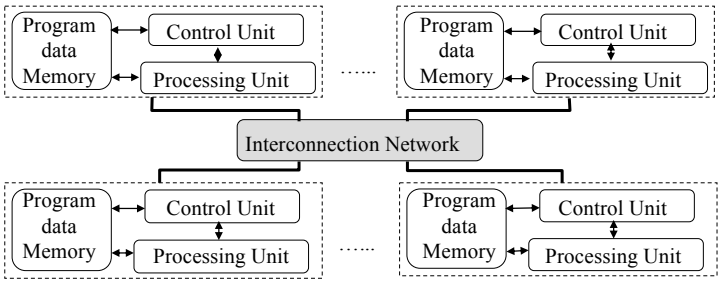



8

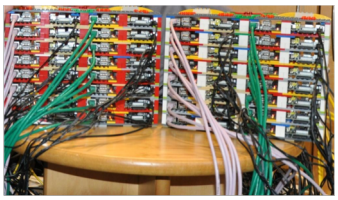
# Introduction - Parallel architectures

- Distributed Memory MIMD
  - Communication way: message passing



# Introduction - Parallel architectures

- Distributed Memory MIMD - Cluster
  - Collection of computers interconnected by :
    - a local network (Gigabit Ethernet)
    - or a dedicated interconnection network (infiniband)
  - Slower inter-processors communication
  - Distributed computing



Raspberry Pi – University of Southampton

Beowulf project – NASA 1994

Cluster LIMOS



# Introduction - Parallel architectures

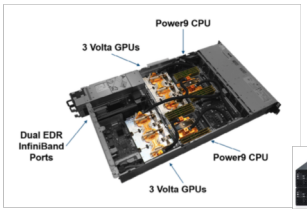
- TOP 500 Supercomputer

| Rank | System | Cores | Rmax (TFlop/s) | Rpeak (TFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 1 | **Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , **IBM**<br>DOE/SC/Oak Ridge National Laboratory<br>**United States** | 2,397,824 | 143,500.0 | 200,794.9 | 9,783 |
| 2 | **Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , **IBM / NVIDIA / Mellanox**<br>DOE/NNSA/LLNL<br>**United States** | 1,572,480 | 94,640.0 | 125,712.0 | 7,438 |
| 3 | **Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , **NRCPC**<br>National Supercomputing Center in Wuxi<br>**China** | 10,649,600 | 93,014.6 | 125,435.9 | 15,371 |
| 4 | **Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , **NUDT**<br>National Super Computer Center in Guangzhou<br>**China** | 4,981,760 | 61,444.5 | 100,678.7 | 18,482 |
| 5 | **Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , **Cray Inc.**<br>Swiss National Supercomputing Centre (CSCS)<br>**Switzerland** | 387,872 | 21,230.0 | 27,154.3 | 2,384 |

# Introduction - Parallel architectures

- Summit (IBM) – DOE/SC/ORNL (USA) (N° 1 – Nov. 2018)

256 cabinets, 18 nodes per cabinet
=> 4608 nodes



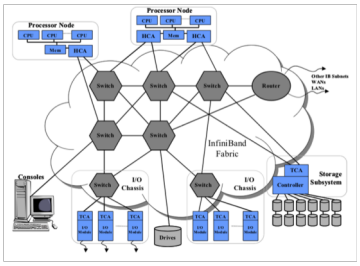

1 node:
2 IBM Power9 CPU (22 cores)
6 NVIDIA V100 GPU (640 cores)

Source: ORNL – Mellanox Technologies

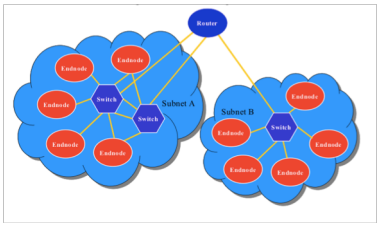


# Introduction - Parallel architectures

- Summit
  - Connection between nodes: Mellanox dual-rail EDR 100Gb/s IB
  - Connection between CPUs and GPUs: NVLink



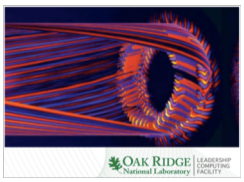

InfiniBand Fabric



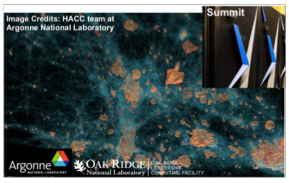

(source: Mellanox Technologies)

## Introduction - Parallel architectures

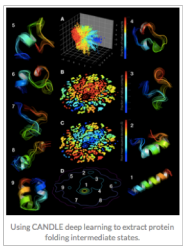
- Summit
  - Software
    - OS: RHEL 7.4; Compiler: XLC 13.1, nvcc 9.2
    - Math Lib.: ESSL, CUBLAS 9.2; MPI: Spectrum MPI
    - AI software: PowerAI DDL
  - Applications



Plasma Simulation



Cosmological Survey Simulation

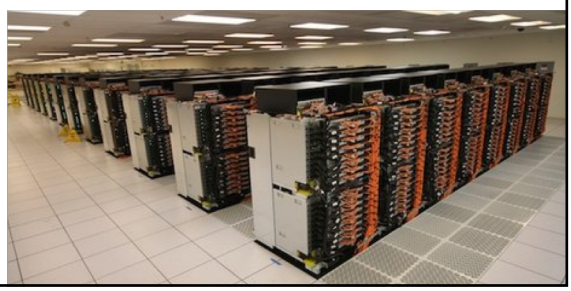


Cancer Surveillance
Source : National Cancer Institute



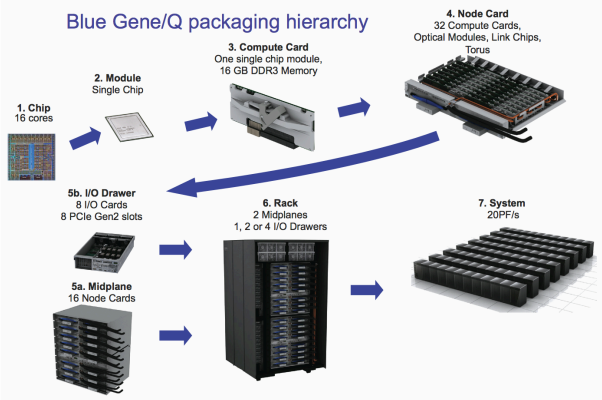
## Introduction - Parallel architectures

- Sequoia (IBM) – DOE/NNSA/ ORNL (USA) (N° 1 – June 2012)
  - BlueGene/Q system
  - Soc custom
    - chip→module→node→ rack→system
  - IBM Power BQC, 1 572 864 cores
  - Network
    - 5D torus + Control network



# Introduction - Parallel architectures

- Sequoia - BlueGene/Q
  - SoC based

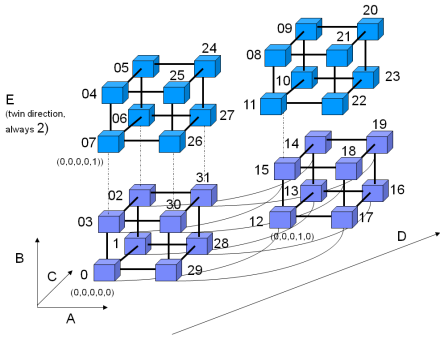



Source: LLNL, USA



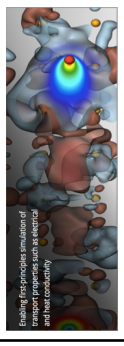
# Introduction - Parallel architectures

- Sequoia - BlueGene/Q
  - 5D torus




Source: IDRIS, France

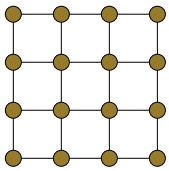
- Softwares
  - Linux environment: Fortran, C, C++, MPI, OpenMP, ESSL
- Applications

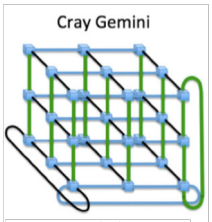

Source: LLNL

# Interconnection Networks - Direct network

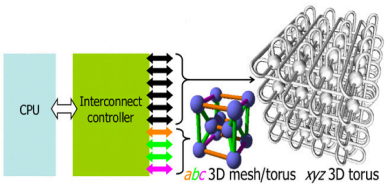
- Direct networks



**2D Mesh**

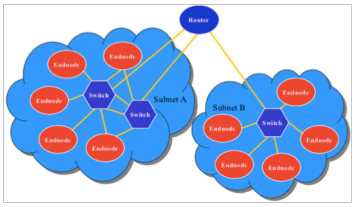



Cray Gemini 3D Torus, Titan
Source: ORNL, USA




Source: AICS, Kobe, Japan

- Indirect networks
  - Fat tree (Summit – ORNL/USA)





# Interconnection Networks - Direct network

- Basic properties: Topology
  - *Node degree*: number of channels connecting this node to its neighbors
  - *Diameter*: the maximum distance between two nodes in the network
  - *Regularity: all the nodes of a network have the same degree*
  - *Symmetry: a network looks alike from every node*

- "Good" properties of interconnection network
  - Performance: low diameter, low latency, high bandwidth
  - Programming facility: regularity, scalability

19

## Programming MIMD architecture

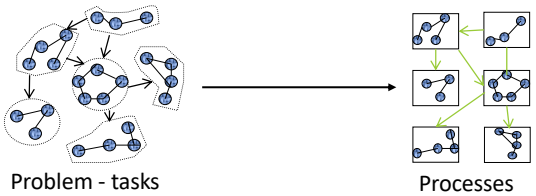
- Programmers must use threads or processes
- Spread the workload across multiple cores
- Write parallel algorithms
  - Easy and quick parallel programming
    - Parallelization with compiler
    - Limitation of processors number
    - Programming library: OpenMP, pthread, TBB
  - Very-large number of processors
    - Parallel programming difficult
    - Programming library: MPI
    - Hybrid programming: MPI for DM nodes & OpenMP/TBB/GP for SM cores

DM – Distribute Memory
SM – Shared Memory

20

## MPI - MPI Programming Model

- Group of processes to solve together a problem



Problem - tasks

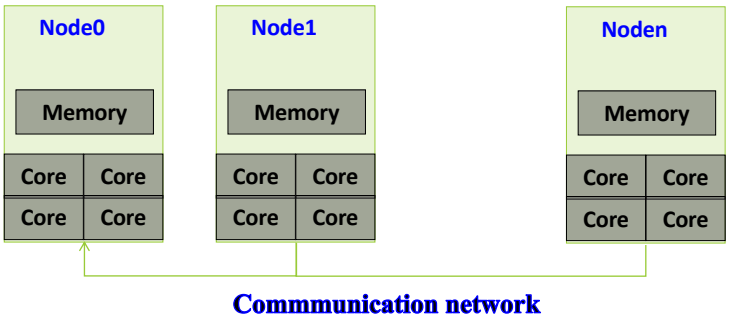
Processes

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

21

# MPI - MPI Programming Model

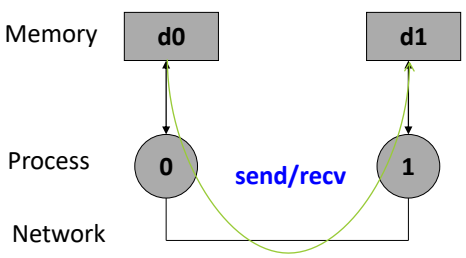
- Distributed memory API



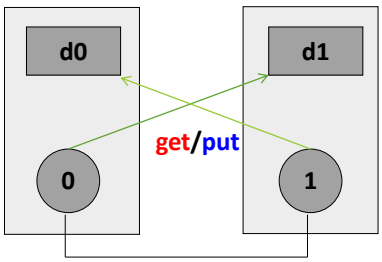

22

# MPI - MPI Programming Model

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




Message passing

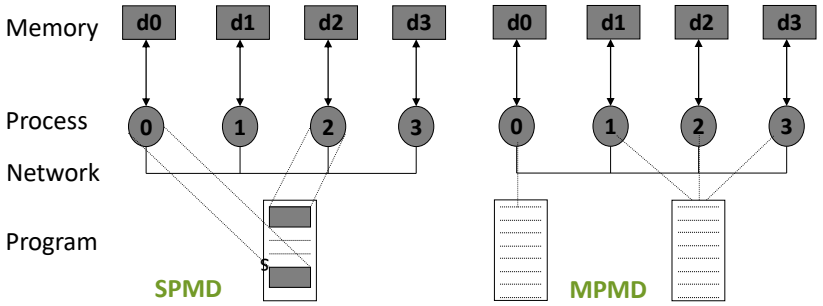



Remote memory acess

23

# MPI - MPI Programming Model

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




24

# 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)

to use with C, C++, Fortran

25

## 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/
- http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html
- P. Pacheco, M. Malensek, An Introduction to Parallel Programming, Elsevier Science, 2nd ed., 2019.
- B. Barney, Introduction to Parallel Computing, https://computing.llnl.gov/tutorials/parallel_comp, consulted on Nov. 2018.
- V. Eijkhout, Introduction to High-Performance Computing, http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html, consulted on Nov. 2018.


26

## MPI - Environment Management Routines

- hello_mpi.c

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

int main(int argc, char **argv)
{
        int myrank, nbprocs;

        MPI_Init( &argc, &argv );
        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();
        return 0;
}
```

MPI's header file

Execution environment initialization

End of MPI execution

27

# MPI – Compiling and running MPI applications

- Compiling

```
$ mpicc hello_mpi.c -o hello_mpi
```

- Running

```
$ sbatch submit_hello_mpi.sh
```

```
#!/bin/bash
# submit_hello_mpi.sh script d'execution
#SBATCH --partition=court     # partition 'court' pour execution
#SBATCH --ntasks=8            # 8 tasks
#SBATCH --ntasks-per-core=1   # 1 task par core
#SBATCH --job-name=hello_mpi

#execution
mpiexec ./hello_mpi
```

28

# MPI – Get processor information

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

int main(int argc, char **argv)
{
    int  rang=-1, nbProcs=0, nameLength=0;
    int  cpuId=-1; /* Numero du core du nœud utilise */
    char proc_name[MPI_MAX_PROCESSOR_NAME]; /* Nom du nœud utilise */

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rang );
    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 ",
            rang, nbProcs, cpuID, proc_name);

    MPI_Finalize(); return 0;
}
```

29

## MPI - Environment Management Routines

- `MPI_Get_processor_name`: return the processor name and its length

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

- `MPI_Wtime`: return an ellapsed wall clock time in seconds

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

- `MPI_Abort`: terminates all process of communicator if exception: ex. malloc

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

30

## MPI – Point-to-point communication

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

int main(int argc, char **argv)
{
    int        myrank, nbprocs, tag=50, tag2=60, nameLength;
    MPI_Status status;
    float      ts[2]={1.0, 2.0}, tr[2]={-1.0, -1.0};

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

    ts[0] = ts[0]+ts[1]*myrank;
```

31

## MPI – Point-to-point communication

```
    if (myrank == 0) {
       MPI_Send( ts, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD );
       MPI_Recv( tr, 1, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD,  &status );

       printf(" Proc %d: sent data %f, received data %f\n ", myrank, ts[0], tr[0]);
    }
    else if (myrank == 1) {
       MPI_Recv( tr, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status );
       MPI_Send( ts, 1, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD );

       printf(" Proc %d: sent data %f, received data %f\n ", myrank, ts[0], tr[0]);
    }

    MPI_Finalize();
    return 0;
}
```

Dead lock!

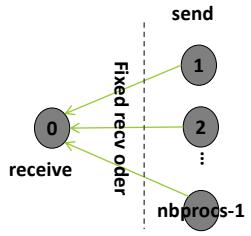
32

## MPI – All-to-one

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

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

   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();
```

# MPI – All-to-one

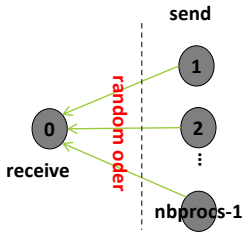
```
   if ( myrank != 0 ) {
      sprintf( message, "Hello from %d on %s-cpu%d!" , myrank, procName, cpuid );
      MPI_Send( message, strlen(message)+1, MPI_CHAR,
                    0, tag, MPI_COMM_WORLD ); }
   else
      for ( src=1; src<nbprocs; src++ ) {
          MPI_Recv( message, 100, MPI_CHAR,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

   MPI_Finalize();
   return 0;
}
```






# MPI – All-to-one

```
   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;
}
```



35

## 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 );
}
```

36

## 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

37

## 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

38

## 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 */
}
```

39

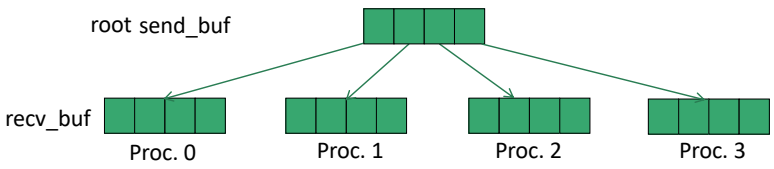
## MPI – Collective communication

- 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 communication time
- Use cases
  - When I/O
  - Parallel algorithms need collective communication

40

## MPI – Collective communication

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

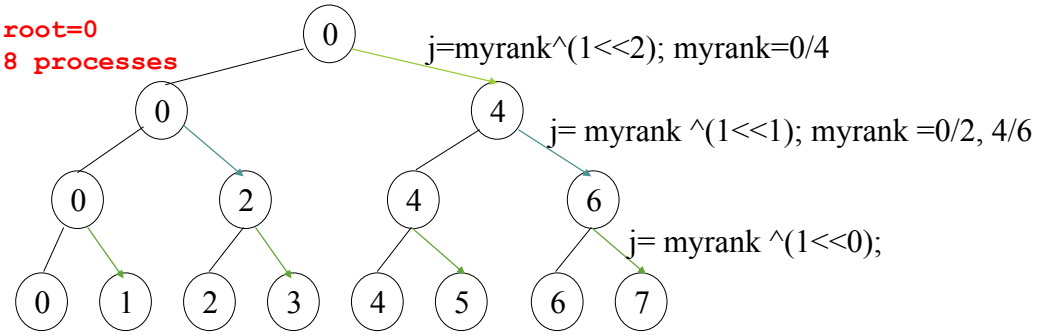



```
int MPI_Bcast(void *msg, int count,
     MPI_Datatype datatype, int root, MIP_Comm comm);
```

41

## MPI – Collective communication

- Broadcast
  - Possible implementation:




42

## MPI – Collective communication

- Broadcast - Broadcast of the dimension of a image

```
int myrank, size, 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<size; 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:**
Steps - $O(nbprocs)$

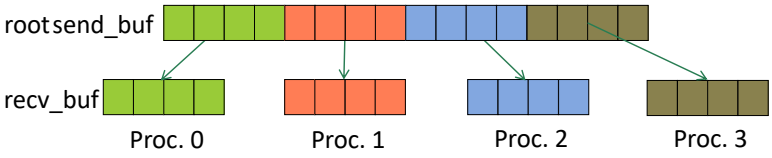
```
MPI_Bcast(dims, 2, MPI_INT, 0, MPI_COMM_WORLD);
```

**Collective communication:**
Steps – $O(\log_2(nbproces))$
with binary tree

43

## MPI – Collective communication

- Scatter – Data distribution

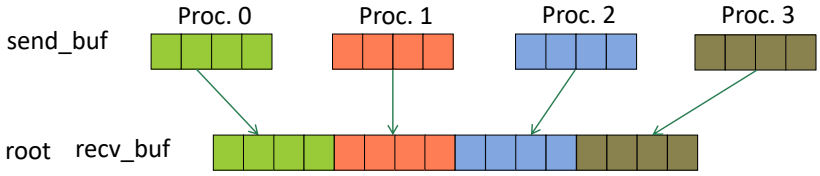



```
int MPI_Scatter( void *sendbuf, int sendcnt,
      MPI_Datatype sendtype, void *recvbuf,
      int recvcnt, MPI_Datatype recvtype,
      int root, MPI_Comm comm );
```

44

## MPI – Collective communication

- Gather - The processes of a group have data to merge together

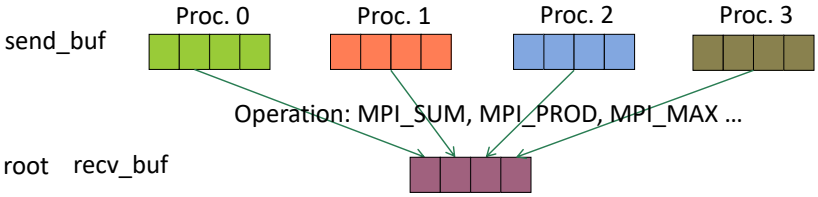



```
int MPI_Gather( void *sendbuf, int sendcnt,
      MPI_Datatype sendtype, void *recvbuf,
      int recvcnt, MPI_Datatype recvtype,
      int root, MPI_Comm comm );
```

45

# MPI – Collective communication

- Reduce – Gather + operation on data

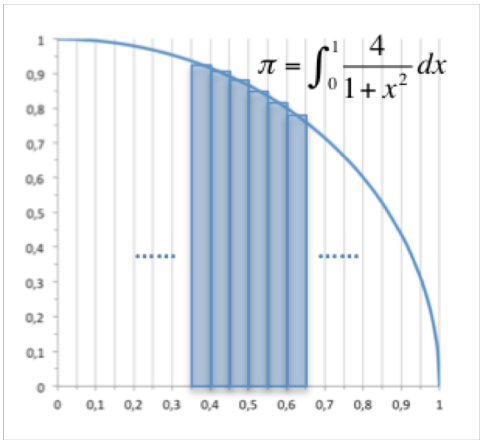



```
int MPI_Reduce( void *operand, void *result,
        int count, MPI_Datatype datatype, MIP_Op op,
        int root, MPI_Comm comm );
```

46

# MPI – Exercice – Computing Pi

```
#include <stdio.h>

int  main(int argc, char **argv)
{
   long nb_rectangles=1000000, i;
   double x, pas;
   double sum=0.0, pi=0.0;

   pas = 1.0 / nb_rectangles ;

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

   pi = pas * sum ;
   printf("Pi=%f\n", pi) ;
   return 0;
}
```



47

## MPI – Exercice – Computing Pi

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

int  main(int argc, char **argv)
{
   long   i, nb_rectangles=0, deb=0, fin=0, rects_par_proc=0;
   double x, pas, 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);

   if (myrank==0) nb_rectangles=1000000;

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

48

## MPI – Exercice – Computing Pi

```
   pas = 1.0 / nb_rectangles ;

   rects_par_proc=nb_rectangles / nbprocs;
   deb=myrank*rects_par_proc;
   fin=deb+rects_par_proc;

   for (i=deb; i<fin; i++) {
      x = (i+0.5)*pas;
      sum += 4.0 / (1.0 + x*x);
   }
   sum = pas * sum ;

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

   if (myrank==0) printf("Pi=%0.121f\n", pi);
   MPI_Finalize();
   return 0;
}
```

49

## MPI – Collective communication

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

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

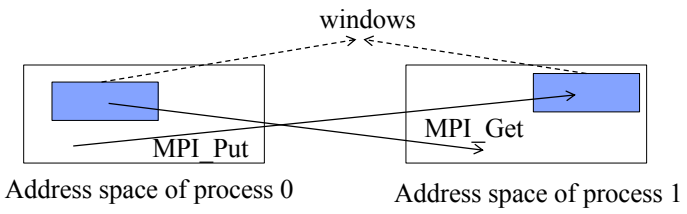
In main function

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

50

## 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



51

## 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

52

## MPI – Exercice – Computing Pi

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

int  main(int argc, char **argv)
{
   long   i, n=0, rects_par_proc=0, deb=0, fin=0;
   double x, pas, 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);

   if (myrank==0) n=1000000;
```

53

## MPI – Exercice – Computing 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);
```

54

## MPI – Exercice – Computing Pi

```
   pas = 1.0 / n ;

   rects_par_proc = n / nbprocs;
   deb = myrank*rects_par_proc;
   fin = deb+rects_par_proc;

   for (i=deb; i<fin; i++) {
      x = (i+0.5)*pas;    sum += 4.0 / (1.0 + x*x);
   }
   pi = pas * 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("Pi=%0.121f\n", pi);

   MPI_Finalize();  return 0;
}
```

55

## Parallel computing – Performance evaluation

- Sequential program: user CPU usage time

```
#include <sys/resource.h>
#include <time.h>
#include <stdio.h>

int main() {
    struct rusage ru;
    struct timeval tim;
    double total; int    i;

    for (i=0; i<10000000; i++) {};
    getrusage(RUSAGE_SELF, &ru);
    tim = ru.ru_utime;
    total = (double)tim.tv_sec + (double)tim.tv_usec/1000000;

    printf("Total time taken by CPU : %f\n", total);
    return 0;
}
```

56

## Parallel computing – Performance evaluation

- 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

## Parallel computing – Performance evaluation

- Comparison with sequential program
  - Make a appointment for all processes
- Metrics
  - Speedup
  - Efficiency
  - Scalability
  - Redundancy
- Laws
  - Amdahl's laws
  - Gustafson-Barsis law
  - …



## Parallel computing – Performance evaluation

- Second law of Amdhal
  - Speedup

$$S_N(M) = \frac{T_1(M)}{T_N(M)} \leq N$$

Sequential execution of the parallel program
The best sequential algorithme?

N: processes number; M: problem size

  - Efficiency $E_N(M) = \frac{S_N(M)}{N}$
- Gustafson-Barsis law

$S(N) = N - \alpha(N-1)$; $\alpha = a/(a+b)$

a+b: execution time of the parallel program with P processes
a: sequential part; b: parallel part

# Parallel computing – Performance evaluation

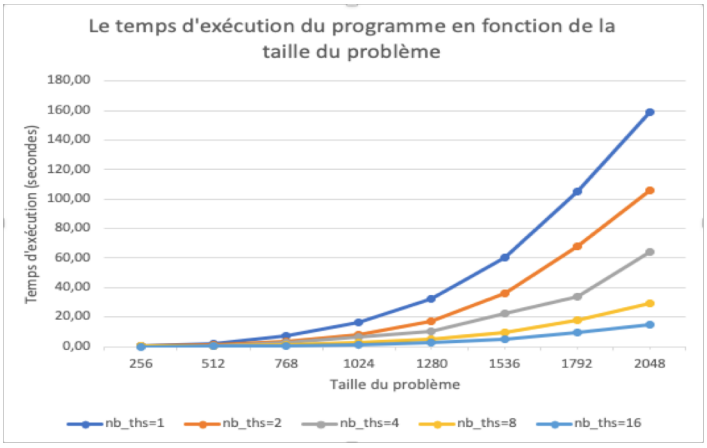
- Example of a openMP program – execution time

| Nb_threads<br>Taille | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| **256** | 0,22 | 0,16 | 0,13 | 0,06 | 0,03 |
| **512** | 1,88 | 1,19 | 0,77 | 0,36 | 0,18 |
| **768** | 7,09 | 3,71 | 2,64 | 1,02 | 0,57 |
| **1024** | 16,29 | 8,24 | 6,35 | 2,36 | 1,27 |
| **1280** | 31,92 | 17,33 | 10,60 | 4,72 | 2,58 |
| **1536** | 60,50 | 36,28 | 22,07 | 9,52 | 5,01 |
| **1792** | 105,13 | 67,82 | 33,59 | 17,83 | 9,33 |
| **2048** | 158,88 | 105,99 | 64,40 | 28,90 | 15,17 |



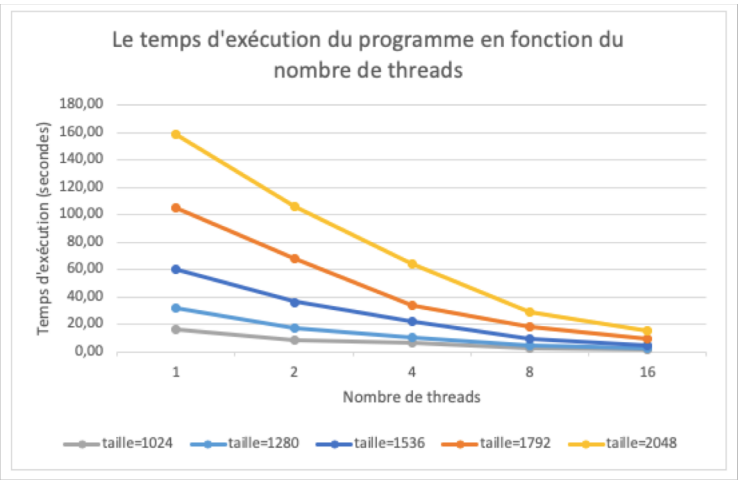
# Parallel computing – Performance evaluation

- Example of a openMP program – execution time



# Parallel computing – Performance evaluation

- Example of a openMP program – execution time






# Parallel computing – Performance evaluation

- Example of a openMP program - Speedup

| Nb_threads<br>Taille | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| **512** | 1 | 1,58 | 2,43 | 5,15 | 10,21 |
| **1024** | 1 | 1,98 | 2,57 | 6,89 | 12,83 |
| **1536** | 1 | 1,67 | 2,74 | 6,36 | 12,08 |
| **2048** | 1 | 1,50 | 2,47 | 5,50 | 10,47 |

# Parallel computing – Performance evaluation

- Example of a openMP program - Speedup