中国科学院深圳先进技术研究院
SHENZHEN INSTITUTES OF ADVANCED TECHNOLOGY
CHINESE ACADEMY OF SCIENCES

Basic of Parallel Computing

Rongliang Chen

Shenzhen Institutes of Advanced Technology
Chinese Academy of Sciences

Russian-Chinese School on
Mathematical Modeling, Parallel Computing, and Applied Statistics
09/22-25, 2020

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example： Computing Pi
  - Example： Matrix product in parallel
  - Advanced MPI collective operations

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example： Computing Pi
  - Example： Matrix product in parallel
  - Advanced MPI collective operations

What is Supercomputer

A supercomputer is basically an extremely powerful computer.

Difficult to define - it's a moving target

- **In 1980s:**
  - ✓ a "supercomputer" was performing 100 Mega FLOPS
  - ✓ FLOPS: FLoating point Operations Per Second
- **Today:**
  - ✓ a single CPU performs a few Tetra FLOPS
  - ✓ a "supercomputer" performs tens of Peta FLOPS

10^3	Kf
10^6	Mf
10^9	Gf
10^{12}	Tf
10^{15}	Pf

CP-PACS/2048
2.048TFlops (No. 1 at 1996)

ARM v8.2-A CPU
3.072TFlops

Fugaku
513.9PFlops (No.1 2020)

History of the Supercomputer

The CDC 6600, the world's first supercomputer

The CDC 6600 was the flagship of the 6000 series of mainframe computer systems manufactured by Control Data Corporation. Generally considered to be the first successful supercomputer, it outperformed the industry's prior recordholder, the IBM 7030 Stretch, by a factor of three. **With performance of up to 3 mega FLOPS, the CDC 6600 was the world's fastest computer from 1964 to 1969**, when it relinquished that status to its successor, the CDC 7600.

CDC 6600, 1965, 1 MFLOPS

History of the Supercomputer

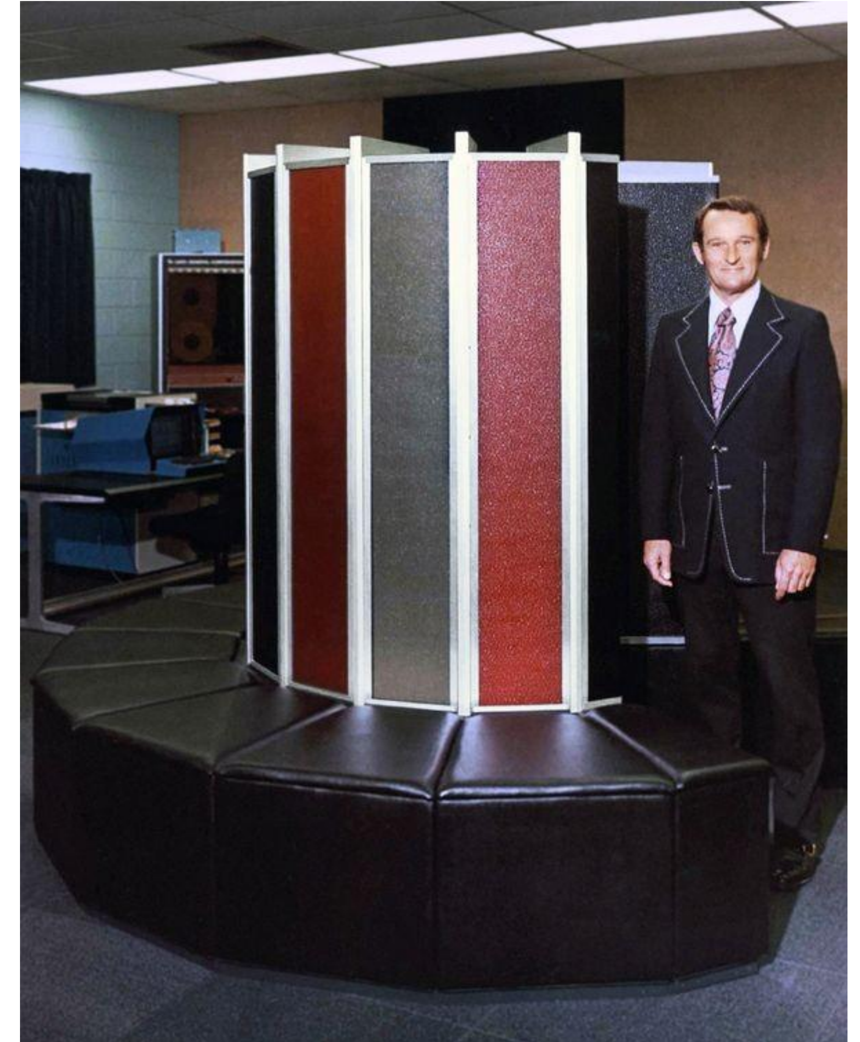
The ILLIAC IV, the most infamous computer

The ILLIAC IV was an important disaster. **Only one model was ever built**. It cost four times as much as initial estimates, and was finished years late. That's the disaster part, but the important? Well, the ILLIAC IV was **the first computer to be built with parallel architecture**. This means it had multiple processors working together, it was seen as a way to get around the technological limitations of processing power at the time.

ILLIAC IV, 1972, 1 GFLOPS

The Cray-1

The Cray-1 was the first supercomputer to successfully implement the **vector processor design**. The Cray-1 was a supercomputer designed, manufactured and marketed by Cray Research. Announced in 1975, the first Cray-1 system was installed at Los Alamos National Laboratory in 1976. Eventually, over 100 Cray-1s were sold, making it one of the **most successful supercomputers in history**. It is perhaps best known for its unique shape, a relatively small C-shaped cabinet with a ring of benches around the outside covering the power supplies and the cooling system.



Cray 1, 1976,133 MFLOPS

History of the Supercomputer

Intel ASCI Red

ASCI Red was the first computer built under the Accelerated Strategic Computing Initiative, the supercomputing initiative of the United States government created to help the maintenance of the United States nuclear arsenal after the 1992 moratorium on nuclear testing.

Intel ASCI Red , 1997
1.0 TFLOPS

IBM Roadrunner

Roadrunner was a supercomputer built by IBM for the Los Alamos National Laboratory in New Mexico, USA. The US$100-million Roadrunner was designed for a peak performance of 1.7 petaflops. It achieved 1.026 petaflops on May 25, 2008, to become the world's first TOP500 LINPACK sustained 1.0 petaflops system.

IBM Roadrunner, 2009
1.7 PFLOPS

History of the Supercomputer

RIKEN K Computer

The K computer was a supercomputer manufactured by Fujitsu, installed at the Riken Advanced Institute for Computational Science campus in Kobe, Japan. The K computer was based on a distributed memory architecture with over 80,000 compute nodes. It was used for a variety of applications, including climate research, disaster prevention and medical research. The K computer's operating system was based on the Linux kernel, with additional drivers designed to make use of the computer's hardware.

RIKEN K Computer, 2011
10 PFLOPS

History of the Supercomputer

Sunway TaihuLight

The Sunway TaihuLight is a Chinese supercomputer which, as of November 2018, is ranked third in the TOP500 list, with a LINPACK benchmark rating of 93 petaflops. The name is translated as divine power, the light of Taihu Lake. This is nearly three times as fast as the previous Tianhe-2, which ran at 34 petaflops. As of June 2017, it is ranked as the 16th most energy-efficient supercomputer in the Green500, with an efficiency of 6.051 GFlops/watt. It was designed by the National Research Center of Parallel Computer Engineering & Technology and is located at the National Supercomputing Center in Wuxi in the city of Wuxi, in Jiangsu province, China.

Sunway TaihuLight, 2016
125 PFLOPS

Fugaku

The supercomputer, a CPU-only project utilizing the ARM architecture, is currently running 152,064 nodes with each compute node featuring a Fujitsu-designed A64FX 48 core processor and 32GB of HBM2 memory bringing the total to 7,299,072 cores and 4,866,048 GB of memory. Each compute node has a Tofu interconnect (28 Gbps x 2 lanes x 10 ports) providing up to 560 Gbps of inter-node bandwidth. There are also 16 PCI-E 3.0 lanes for connecting to GPUs, FPGAs, or other accelerator cards or I/O.

Fugaku, 2020
513.9PFlops

The latest top 10 supercomputer in the world

Top 10 positions of the 55th TOP500 in June 2020[24]

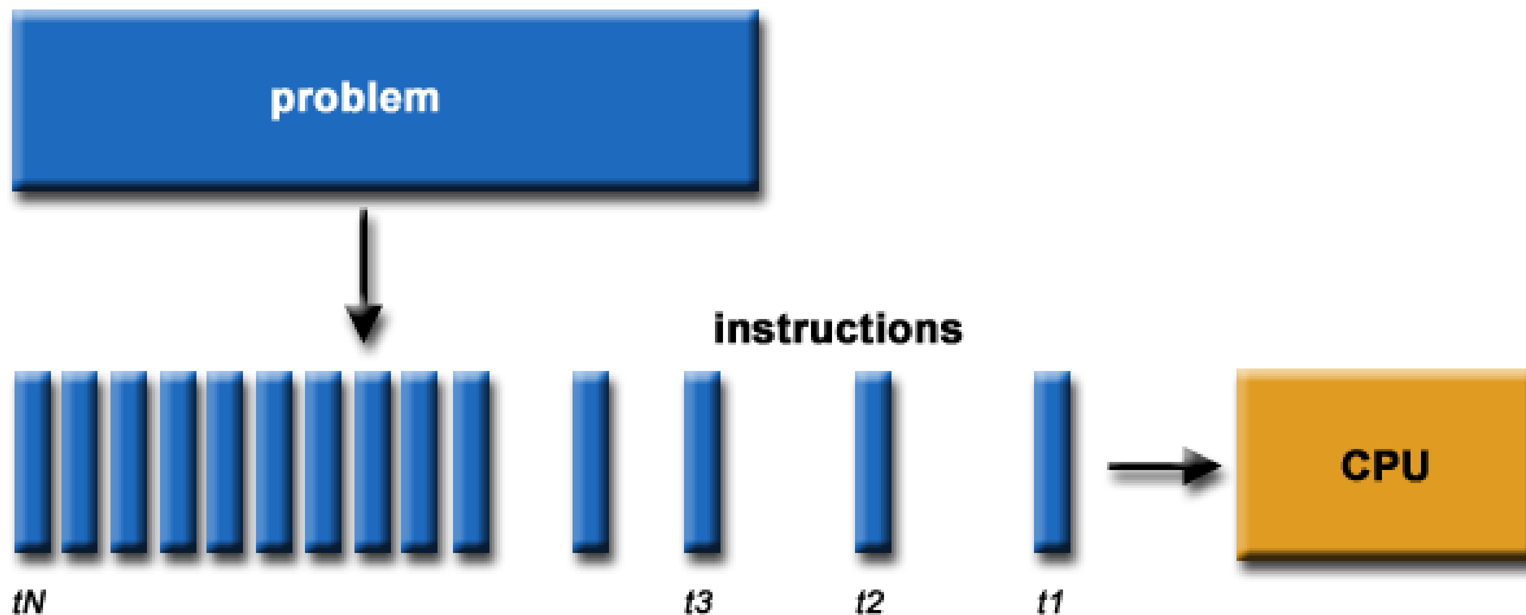
Rank	Rmax Rpeak (PFLOPS)	Name	Model	Processor	Interconnect	Vendor	Site country, year	Operating system
1 ▲	415.530 513.855	Fugaku	Supercomputer Fugaku	A64FX	Tofu interconnect D	Fujitsu	RIKEN Center for Computational Science Japan, 2020	Linux (RHEL)
2 ▼	148.600 200.795	Summit	IBM Power System AC922	POWER9, Tesla V100	InfiniBand EDR	IBM	Oak Ridge National Laboratory United States, 2018	Linux (RHEL)
3 ▼	94.640 125.712	Sierra	IBM Power System S922LC	POWER9, Tesla V100	InfiniBand EDR	IBM	Lawrence Livermore National Laboratory United States, 2018	Linux (RHEL)
4 ▼	93.015 125.436	Sunway TaihuLight	Sunway MPP	SW26010	Sunway[25]	NRCPC	National Supercomputing Center in Wuxi China, 2016[25]	Linux (Raise)
5 ▼	61.445 100.679	Tianhe-2A	TH-IVB-FEP	Xeon E5–2692 v2, Matrix-2000[26]	TH Express-2	NUDT	National Supercomputing Center in Guangzhou China, 2013	Linux (Kylin)
6 ▲	35.450 51.721	HPC5	Dell	Xeon Gold 6252, Tesla V100	Mellanox HDR Infiniband	Dell EMC	Eni Italy, 2020	Linux (CentOS)
7 ▲	27.580 34.569	Selene	Nvidia	Epyc 7742, Ampere A100	Mellanox HDR Infiniband	Nvidia	Nvidia United States, 2020	Linux (Ubuntu)
8 ▼	23.516 38.746	Frontera	Dell C6420	Xeon Platinum 8280 (subsystems with e.g. POWER9 CPUs and Nvidia GPUs were added after official benchmarking[10])	InfiniBand HDR	Dell EMC	Texas Advanced Computing Center United States, 2019	Linux (CentOS)
9 ▲	21.640 29.354	Marconi-100	IBM Power System AC922	POWER9, Volta V100	Dual-rail Mellanox EDR Infiniband	IBM	CINECA Italy, 2020	Linux (RHEL)
10 ▼	21.230 27.154	Piz Daint	Cray XC50	Xeon E5-2690 v3, Tesla P100	Aries	Cray	Swiss National Supercomputing Centre Switzerland, 2016	Linux (CLE)

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example： Computing Pi
  - Example： Matrix product in parallel
  - Advanced MPI collective operations

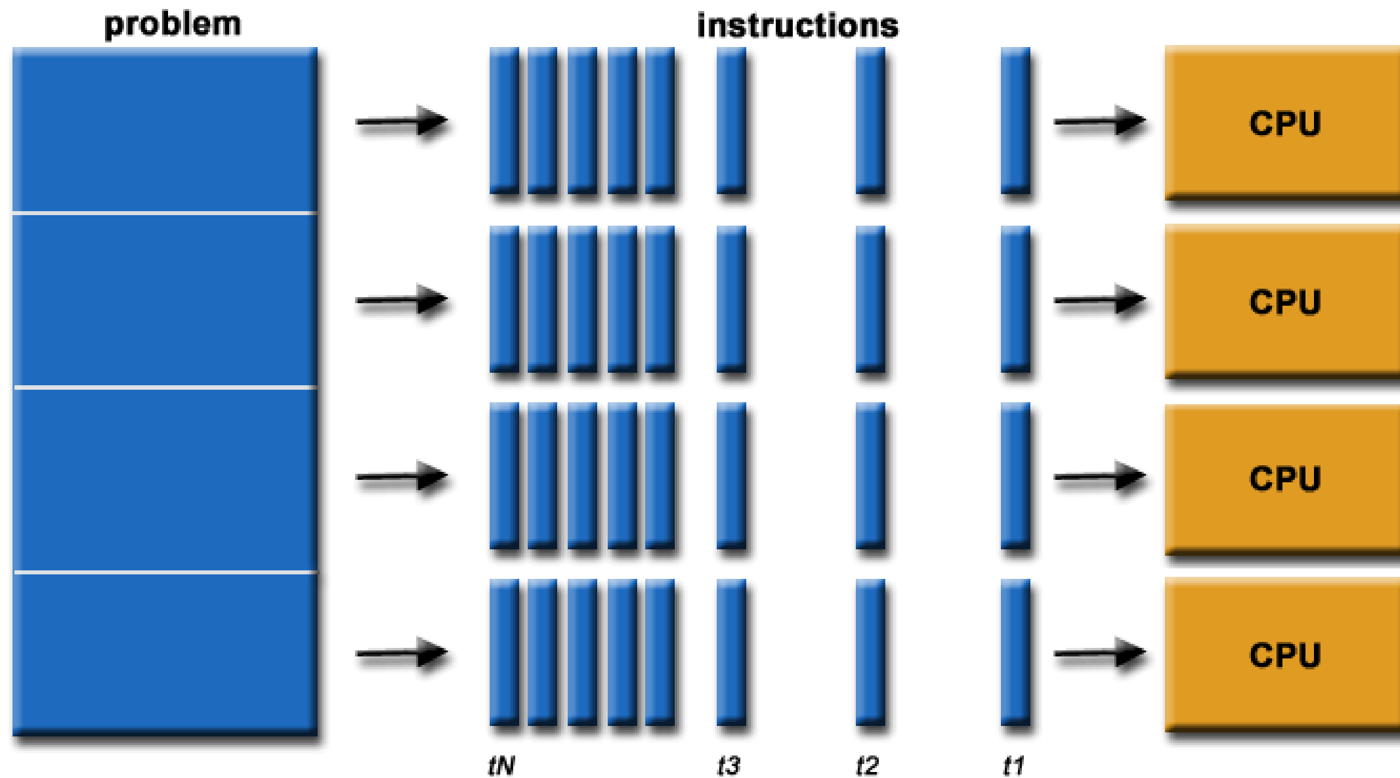
What is Parallel Computing? (1)

- Traditionally, software has been written for ***serial*** computation:
  - ✓ To be run on a single computer having a single Central Processing Unit (CPU);
  - ✓ A problem is broken into a discrete series of instructions.
  - ✓ Instructions are executed one after another.
  - ✓ Only one instruction may execute at any moment in time.



What is Parallel Computing? (2)

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - ✓ To be run using multiple CPUs
  - ✓ A problem is broken into discrete parts that can be solved concurrently
  - ✓ Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

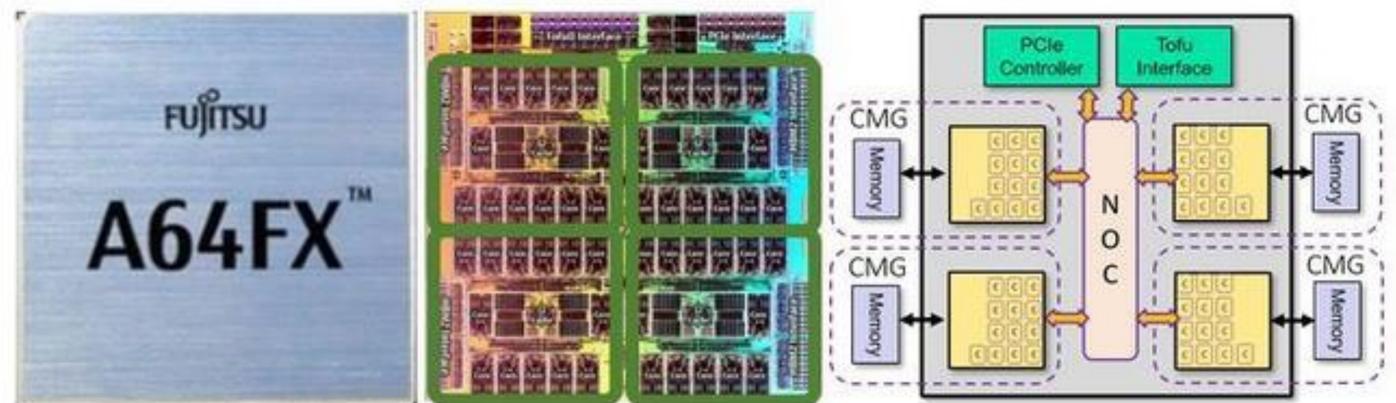


Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example：Computing Pi
  - Example：Matrix product in parallel
  - Advanced MPI collective operations

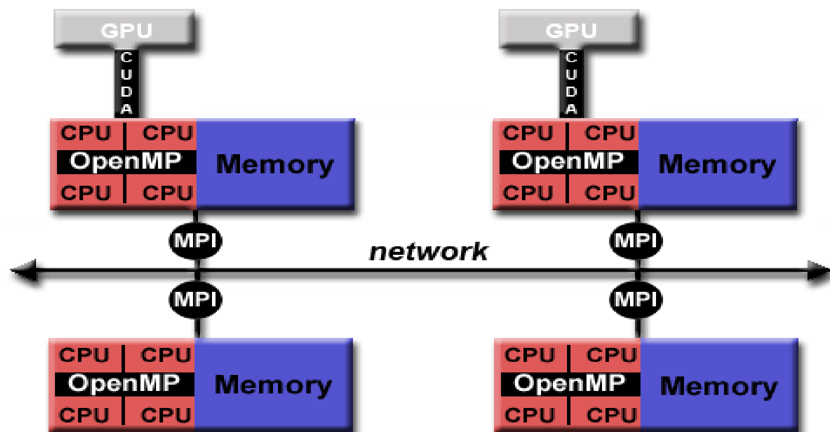
Why Use Parallel Computing?

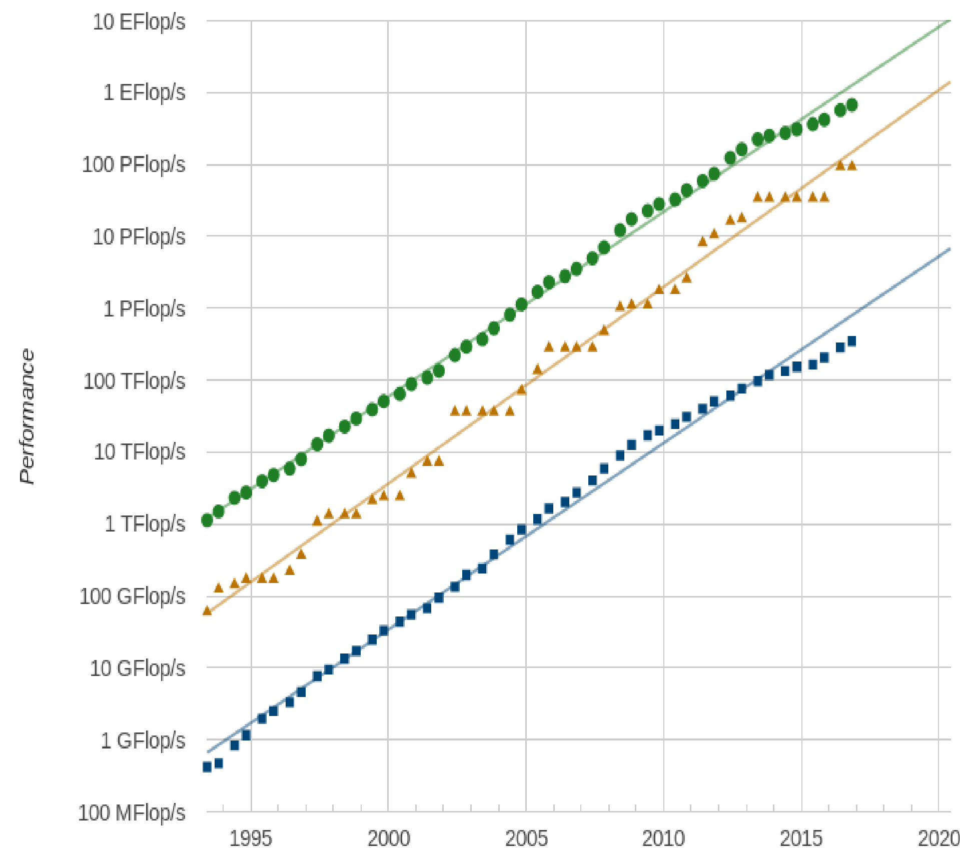
- **Save Time and/or Money**
  - ✓ In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings
  - ✓ Parallel computers can be built from cheap, commodity components
- **Solve Larger/More Complex Problems**
  - ✓ Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer
- **Make Better Use of Underlying Parallel Hardware**
  - ✓ Modern computers, even laptops, are parallel in architecture with multiple processors/cores



Opportunities and Challenges

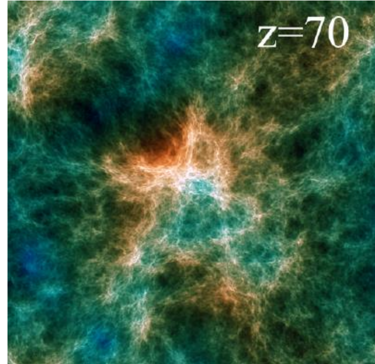
- **Opportunities**
- ✓ The computer is very powerful now (500 PFlop/s)
- ✓ The performance increases very fast (1000 PFlop/s in 2020)
- **Challenges**
- ✓ Massively Parallel (Sunway TaihuLight has 10 million cores)
- ✓ Hybrid architecture (CPU+GPU, CPU+MIC, CPU+XXX)



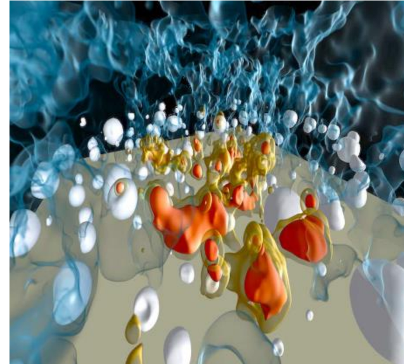


State of the Art in HPC

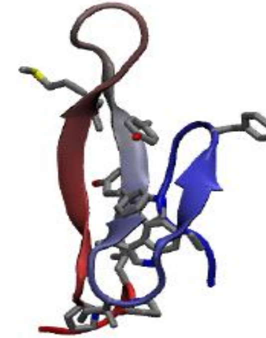
Gordon-Bell performance prize:



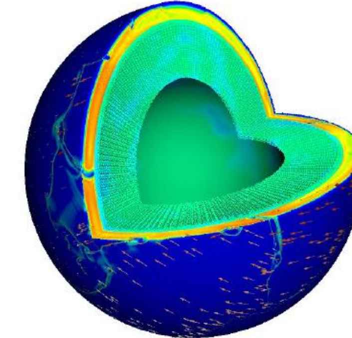
- Dark matter simulation
- Japan, 2012
- K computer
- 82944 nodes
- 4.45 Pflops



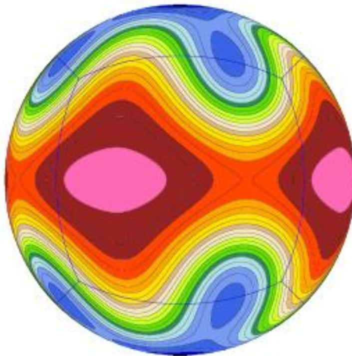
- Simulations of Cloud Cavitation Collapse
- Switzerland, 2013
- IBM Sequoia
- 1.6M cores
- 11 Pflops



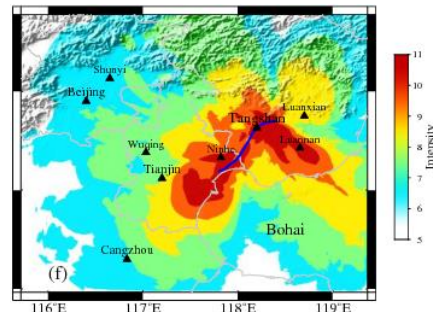
- folding kinetics of protein
- USA, 2014
- **Anton 2**
- 33792 cores



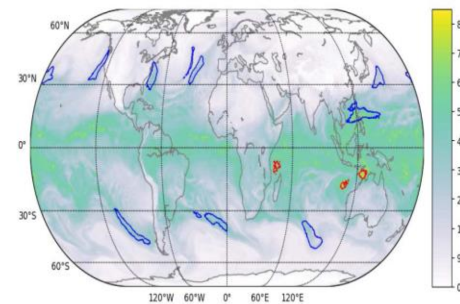
- Flow in Earth's Mantle
- USA, 2015
- Sequoia
- 1.5M
- 687 TFlops



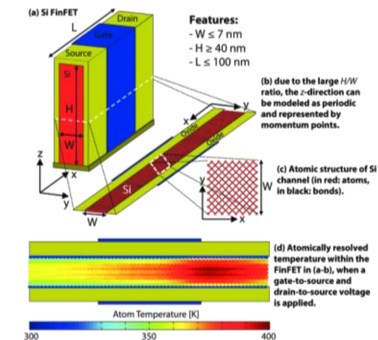
- Atmospheric Dynamics
- China, 2016
- Sunway TaihuLight
- 10M
- 7.95 PFLops



- Earthquake Simulation
- China, 2017
- Sunway TaihuLight
- 10M
- 18.9 PFLops



- Genetic Architectures and Climate Analytics
- USA, 2018
- Summit
- 2.4M
- 189 PFLops

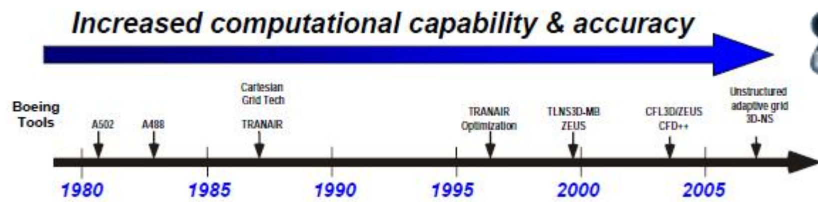


- Simulation Maps Heat in Transistors
- Switzerland, 2019
- Summit
- 2.4M
- 90.89 PFLops

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example：Computing Pi
  - Example：Matrix product in parallel
  - Advanced MPI collective operations

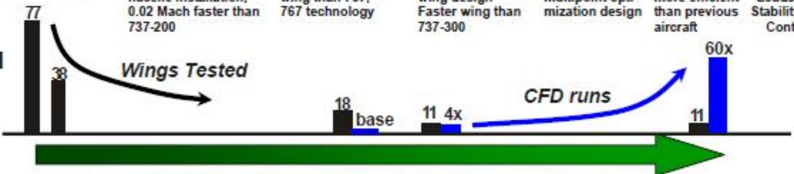
What Parallel Computing Can do?



CFD Tools

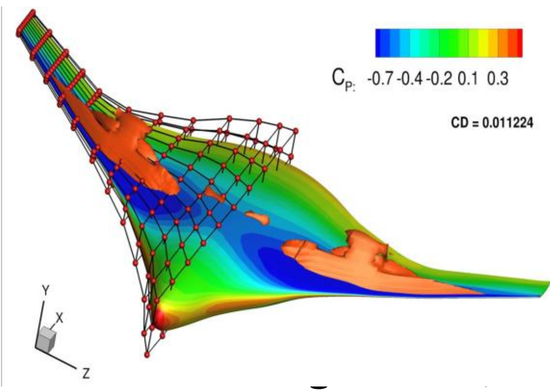
Boeing Products

Wind Tunnel vs. CFD



Less testing, lower cost, better products

COPYRIGHT © 2005 THE BOEING COMPANY



Coordinates: (x,y,z)
Velocity Components: (u,v,w)
Time : t
Density: ρ
Total Energy: Et
Pressure: p
Stress: τ
Heat Flux: q
Reynolds Number: Re
Prandtl Number: Pr

Continuity: $\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho w)}{\partial z} = 0$

X – Momentum: $\frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} + \frac{\partial(\rho uw)}{\partial z} = -\frac{\partial p}{\partial x} + \frac{1}{Re_r}\left[\frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{xz}}{\partial z}\right]$

Y – Momentum: $\frac{\partial(\rho v)}{\partial t} + \frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho v^2)}{\partial y} + \frac{\partial(\rho vw)}{\partial z} = -\frac{\partial p}{\partial y} + \frac{1}{Re_r}\left[\frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{yz}}{\partial z}\right]$

Z – Momentum $\frac{\partial(\rho w)}{\partial t} + \frac{\partial(\rho uw)}{\partial x} + \frac{\partial(\rho vw)}{\partial y} + \frac{\partial(\rho w^2)}{\partial z} = -\frac{\partial p}{\partial z} + \frac{1}{Re_r}\left[\frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z}\right]$

Energy:

$$\frac{\partial(E_T)}{\partial t} + \frac{\partial(uE_T)}{\partial x} + \frac{\partial(vE_T)}{\partial y} + \frac{\partial(wE_T)}{\partial z} = -\frac{\partial(up)}{\partial x} - \frac{\partial(vp)}{\partial y} - \frac{\partial(wp)}{\partial z} - \frac{1}{Re_r Pr_r}\left[\frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + \frac{\partial q_z}{\partial z}\right]$$
$$+ \frac{1}{Re_r}\left[\frac{\partial}{\partial x}(u\tau_{xx} + v\tau_{xy} + w\tau_{xz}) + \frac{\partial}{\partial y}(u\tau_{xy} + v\tau_{yy} + w\tau_{yz}) + \frac{\partial}{\partial z}(u\tau_{xz} + v\tau_{yz} + w\tau_{zz})\right]$$

What Parallel Computing Can do?

Much CFD penetration. Opportunities exist for higher accuracy and expanded complexity

Some CFD penetration. Opportunities exist for large increases in design process speed and application

CFD penetration opportunity

Wind Tunnel Corrections

Interior Air Quality

Cabin Noise

Connexion Antenna

High-Speed Wing Design

Wing Tip Design

Control Failure Analysis

Vertical Tail and Aft Body Design

High-Lift Wing Design

Buffet Boundary

Design For Stability & Control

Cab Design

ECS Inlet Design

Icing

Wing Controls

Air Data System Location

Design for FOD Prevention

Inlet Design

Inlet Certification

Nacelle Design

Engine Bay Thermal Analysis

Exhaust System Design

Wing-Body Fairing Design

APU Inlet And Ducting

Design For Loads

Avionics Cooling

Engine/Airframe Integration

Vortex Generator Placement

Flutter

Wake Vortex Alleviation

Reynolds Number Corrections for Loads and S&C

Thrust Reverser Design

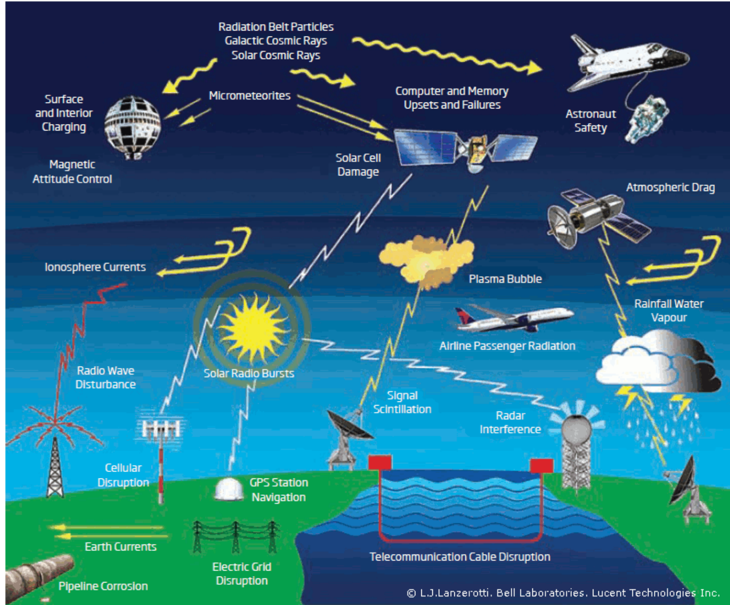
Planform Design

Aeroelastics

Community Noise

APU and Propulsion Fire Suppression

What Parallel Computing Can do?



ather and climate models

are essential for the production of high-quality weather information. They are an indispensable tool in the ecasts and climate scenarios. KNMI is continually ese models and to keep up with the latest y. But how does such a model work?

In a column of grid cells we encounter modules for condensation, precipitation, radiation, turbulence, evaporation and surface processes.

❸ Weather forecast

The initial conditions of the quantities in each grid cell in the forecast model are determined from observations by weather satellites, ground stations, weather balloons and other measurements.

The observations and the model are not perfect. A slight deviation from the initial state leads to different weather situations. By slightly changing the initial conditions and the physical modules a 'weather plume' is created.

Narrow plume: fairly certain weather forecast
Wide plume: uncertain forecast

❹ Climate scenarios

For climate simulations the model calculates far ahead. Factors that affect the climate, such as greenhouse gases, are taken into account.

Models used at KNMI

ECMWF
Global model from the European Weather Centre in Reading (UK). Used for forecasts up to 2 weeks on a grid of 9 x 9 km (around 600 cells for The Netherlands).

HARMONIE
Model for The Netherlands and surroundings. In use since 2012 for forecasts up to 2 days with cells of 2.5 x 2.5 km (around 10000 cells for The Netherlands).

Supercomputer
HARMONIE requires around 3 quadrillion calculations. KNMI has a computer with a capacity of 50 trillion calculations/sec (50 teraflops) that is used to make 8 weather forecasts per day.

❷ Calculations

In the model the atmosphere is divided up in grid cells

In each 3-dimensional grid cell, quantities are maintained, such as: • temperature • pressure • humidity • wind • radiation etc.

The actual values of these quantities are constantly changing as radiation is reflected, water evaporates, turbulence causes mixing, etc.

These changes are calculated by the model in modules that describe the physical processes.

Each calculation moves the model forward in 60 second steps: now, 60 sec., 60 sec., 60 sec., etc.

This is a publication by ©KNMI 2017 Also see (in Dutch): www.knmi.nl

$$\frac{du}{dt} - fv - \frac{uv}{a}\tan\varphi = -\frac{1}{a\cos\varphi}\frac{\partial\Phi}{\partial\lambda} - \frac{RT}{a\cos\varphi}\frac{\partial\pi}{\partial\lambda} + \frac{g}{p_s}\frac{\partial F_u^{vdf}}{\partial\sigma} + F_\lambda^{diff},$$

$$\frac{dv}{dt} + fu + \frac{u^2}{a}\tan\varphi = -\frac{1}{a}\frac{\partial\Phi}{\partial\varphi} - \frac{RT}{a}\frac{\partial\pi}{\partial\varphi} + \frac{g}{p_s}\frac{\partial F_v^{vdf}}{\partial\sigma} + F_\varphi^{diff},$$

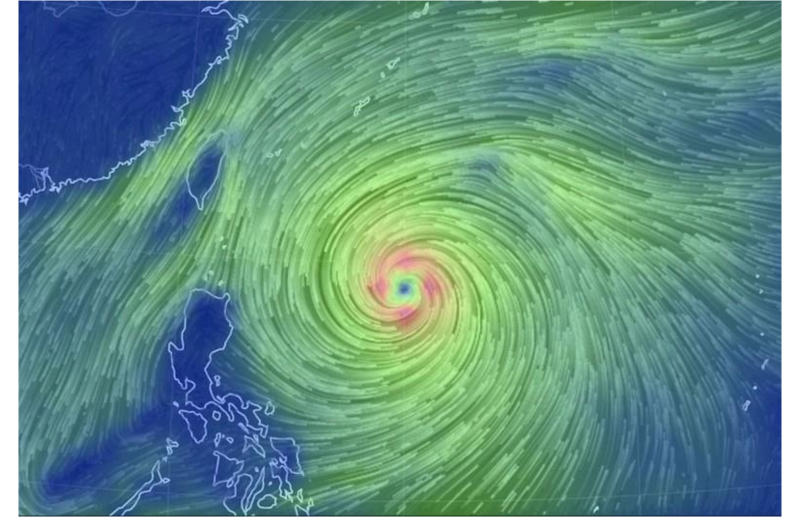
$$\frac{d\pi}{dt} = -\frac{1}{a\cos\varphi}\frac{\partial u}{\partial\lambda} - \frac{1}{a\cos\varphi}\frac{\partial}{\partial\varphi}(v\cos\varphi) - \frac{\partial\dot\sigma}{\partial\sigma},$$

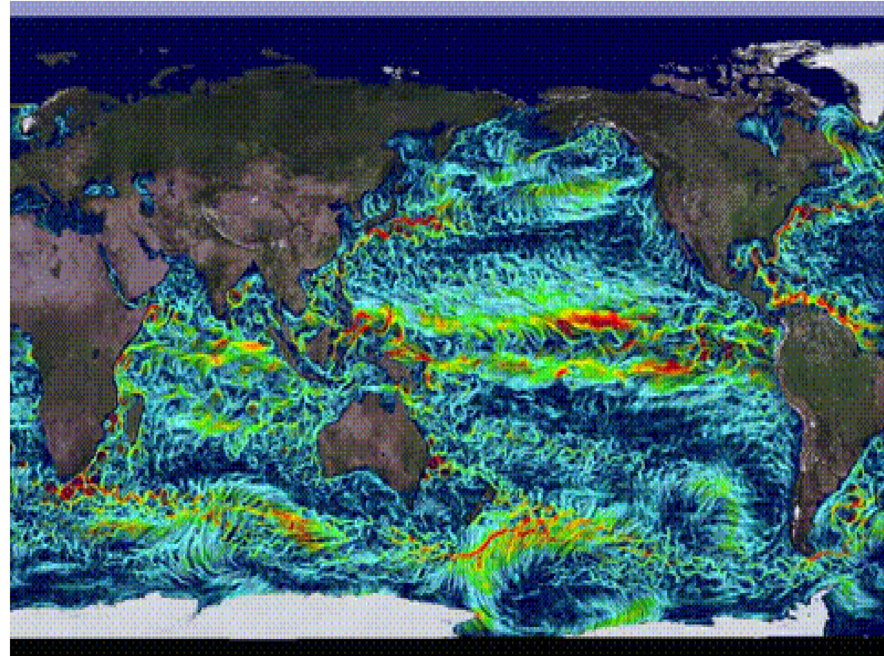
$$\frac{\partial\Phi}{\partial\sigma} = -\frac{RT}{\sigma},$$

$$\frac{dq}{dt} = \frac{g}{p_s}\frac{\partial F_q^{vdf}}{\partial\sigma} + F_q^{diff} + S_q^{cond},$$

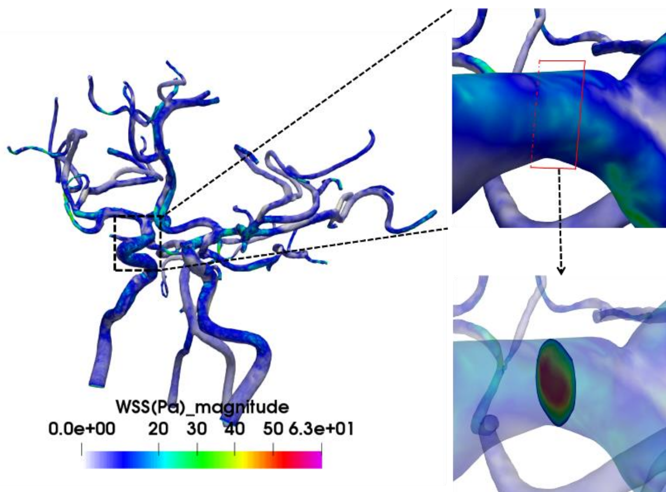
$$\frac{dT}{dt} = \frac{RT}{c_p}\left\{\frac{\partial\pi}{\partial t} + \frac{u}{a\cos\varphi}\frac{\partial\pi}{\partial\lambda} + \frac{v}{a}\frac{\partial\pi}{\partial\varphi} + \frac{\dot\sigma}{\sigma}\right\} + \frac{1}{c_p}\left(\frac{g}{p_s}\frac{\partial F_T^{vdf}}{\partial\sigma} + \frac{g}{p_s}\frac{\partial F_{rad}^{vdf}}{\partial\sigma}\right) + F_T^{diff} + LS_q^{cond},$$

$$C_g\frac{\partial T_g}{\partial t} = q_{rad} + q_{vdfT} + q_{vdfq}.$$

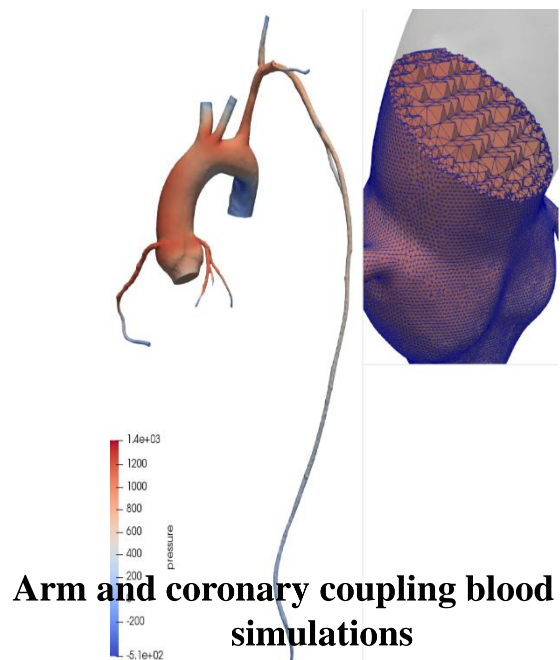




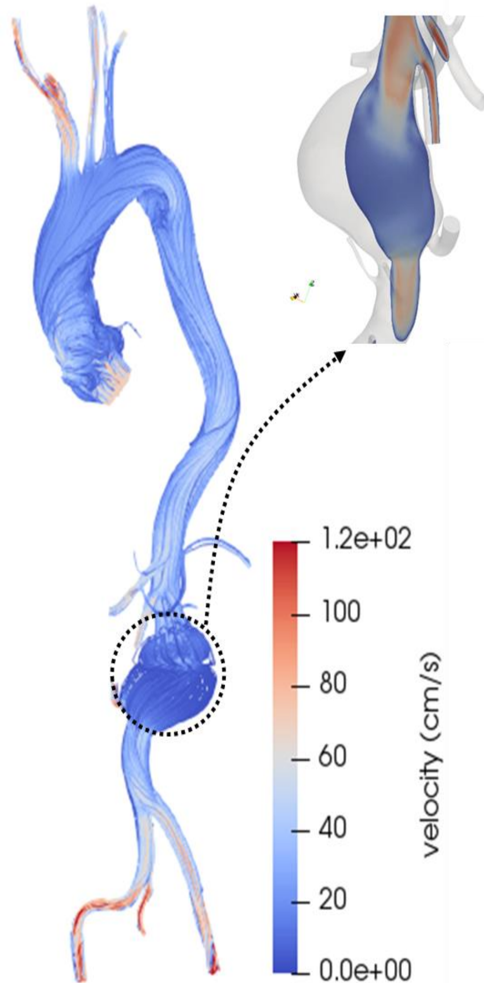
What Parallel Computing Can do?



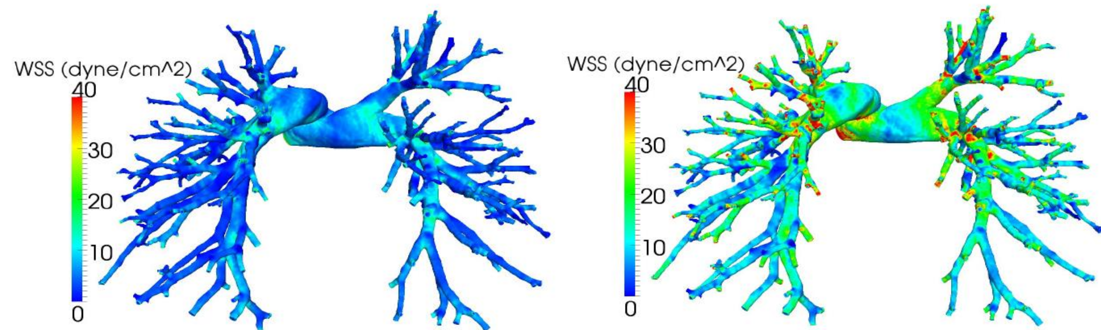
Cerebral blood flow simulations



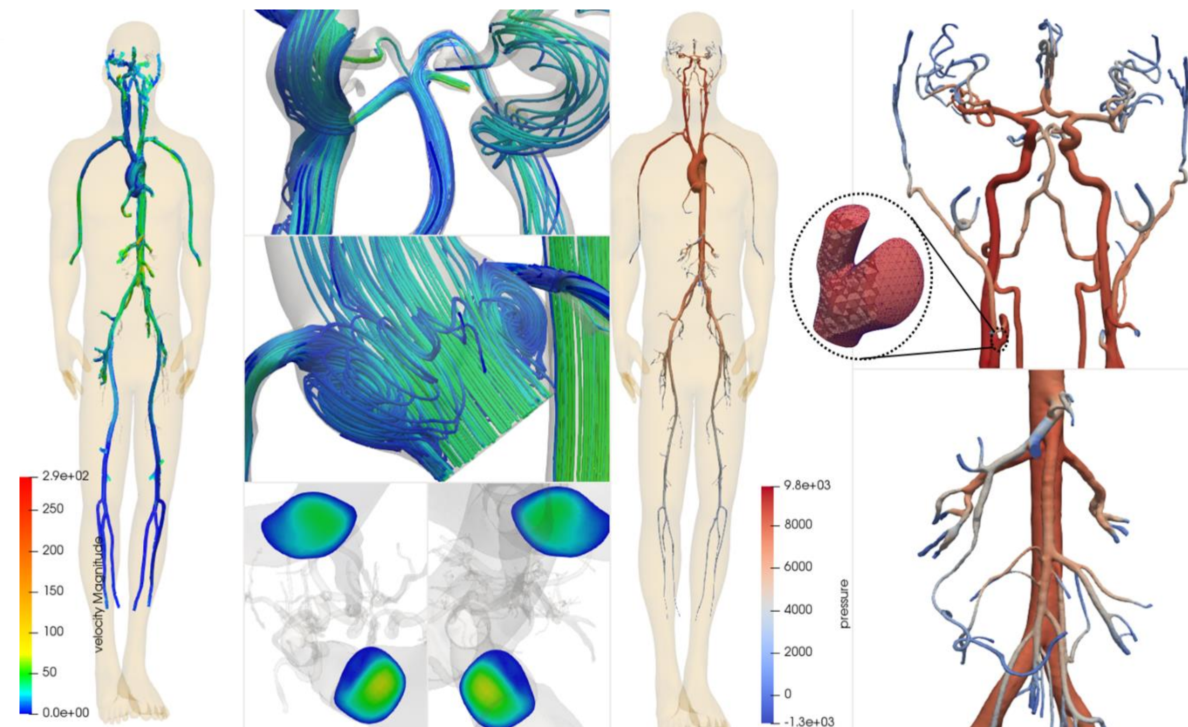
Arm and coronary coupling blood flow simulations



Aorta abdominal blood flow simulation

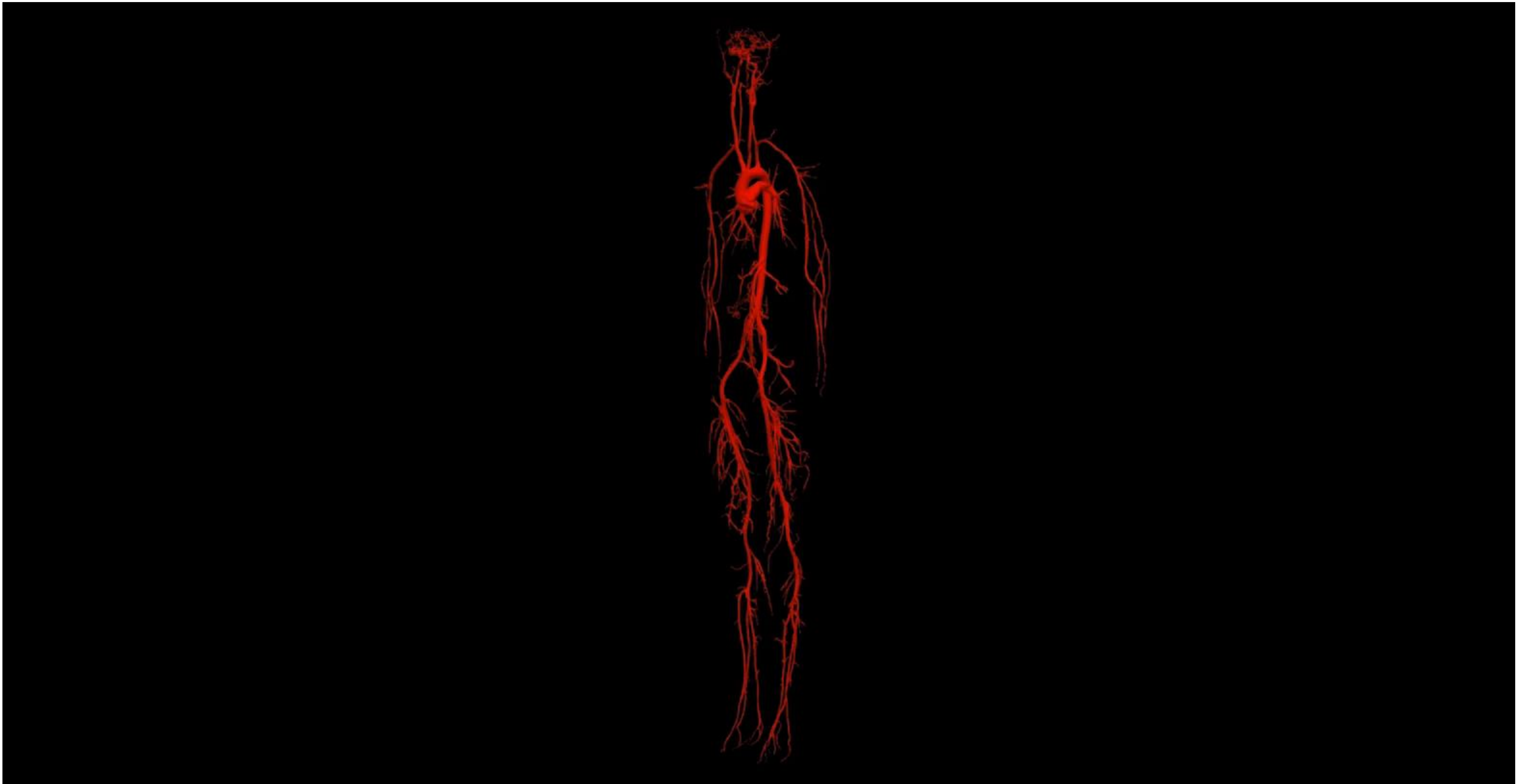


Lung blood flow simulations

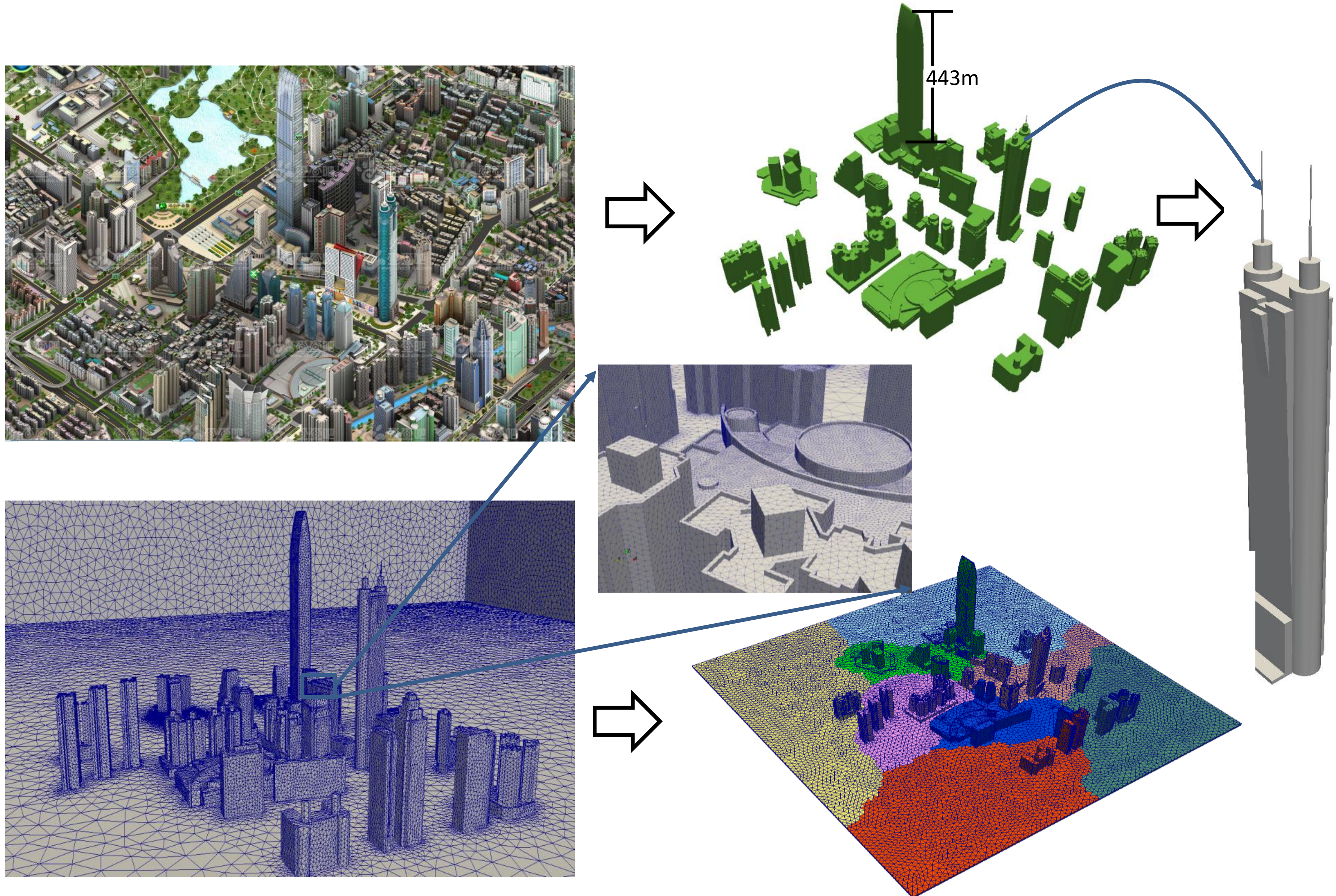


Blood flow simulation of the whole body artery

What Parallel Computing Can do?



What Parallel Computing Can do?



What Parallel Computing Can do?

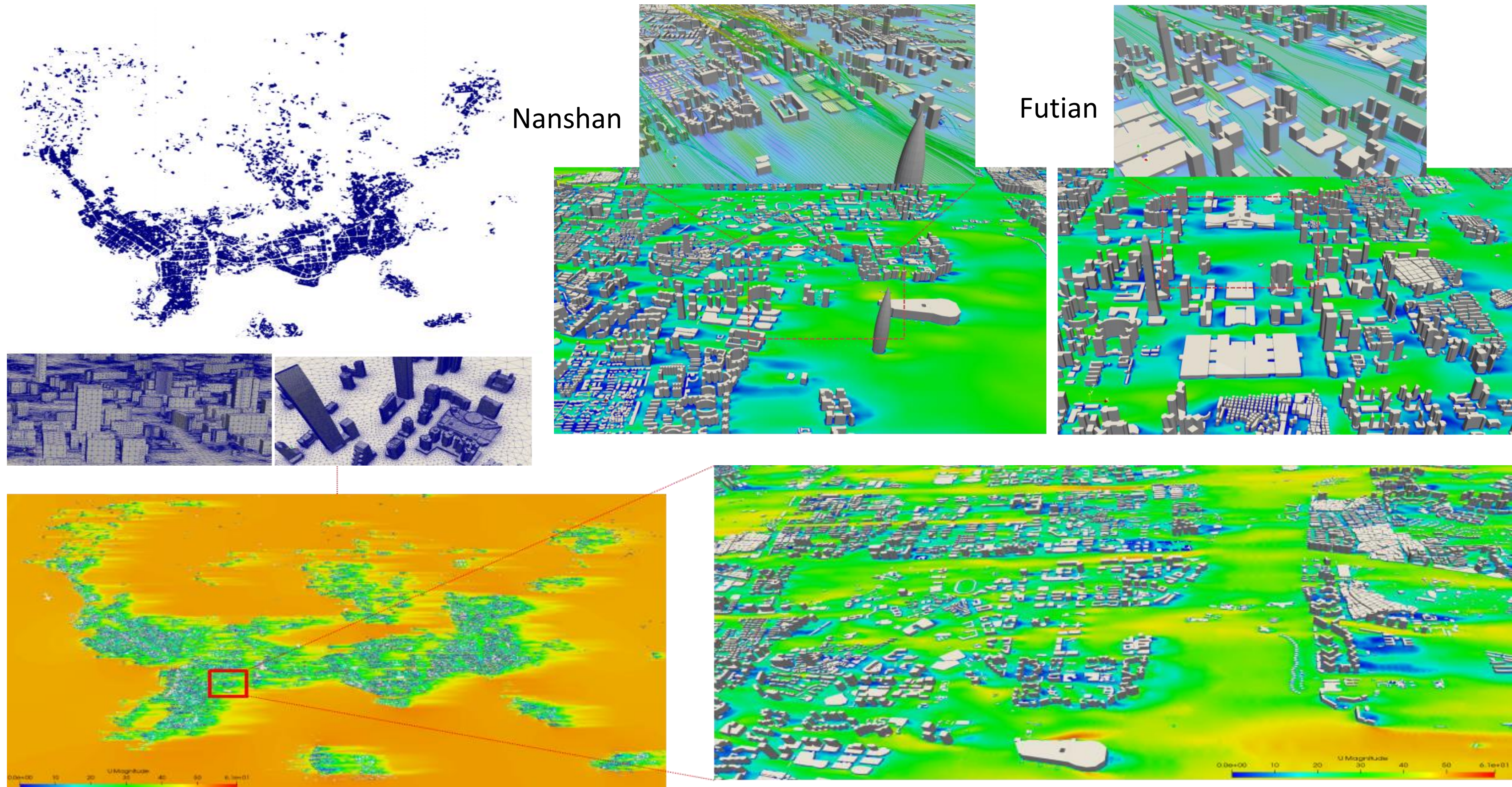


Figure: The wind simulation results of Shenzhen with over 150,000 buildings

Contents

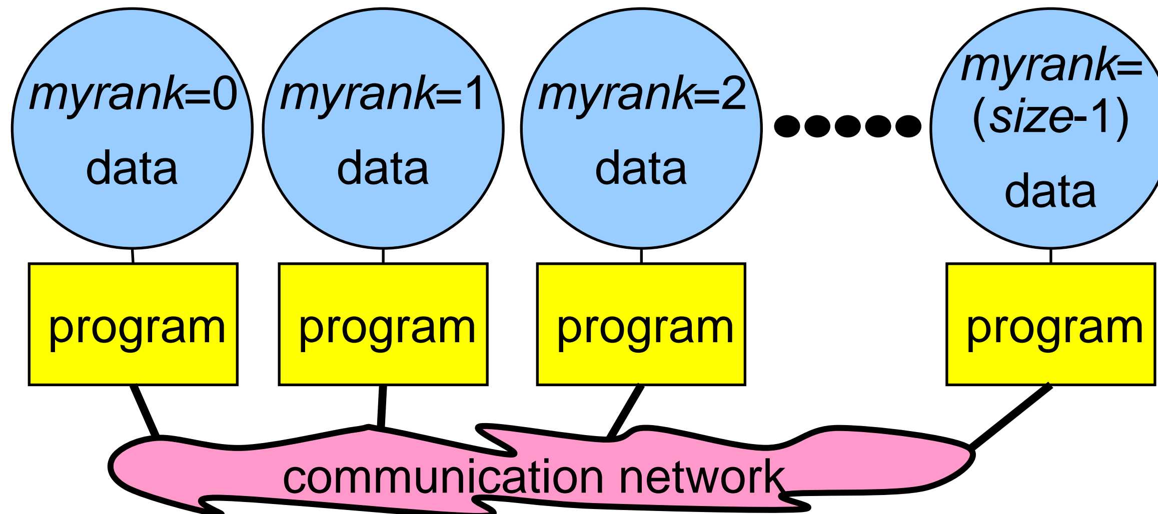
- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example：Computing Pi
  - Example：Matrix product in parallel
  - Advanced MPI collective operations

References

- 1) W. Gropp, E. Lusk, A. Skjellum, Using MPI Portable Parallel Programming with the Message-Passing Interface, The MIT Press, 2014
- 2) W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using Advanced MPI: Modern Features of the Message-Passing Interface, The MIT Press, 2014

What is MPI?

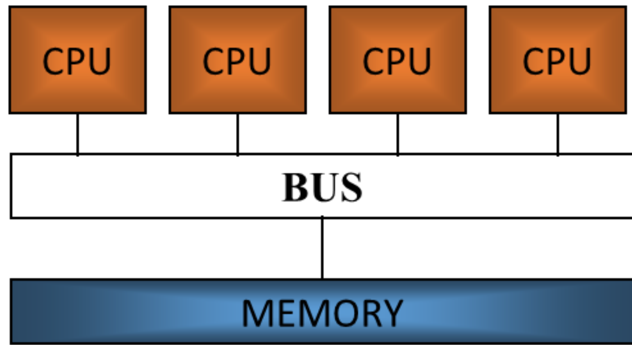
- MPI = “*Message Passing Interface*”
- Message Passing means
  - Each process is a standalone *sequential* program.
  - All *data is private* to each process.
  - Communication is performed via *library function calls*.
  - The *underlying language is standard*: Fortran, C, (F90, C++)...
  - MPI is *SPMD* (Single Program Multiple Data).



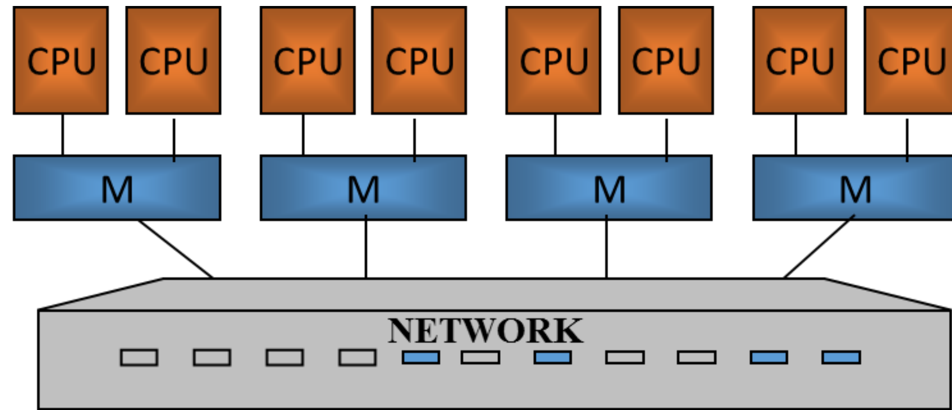
Platform (1)

Platform (2)

Platform (3)



Shared memory



Cluster

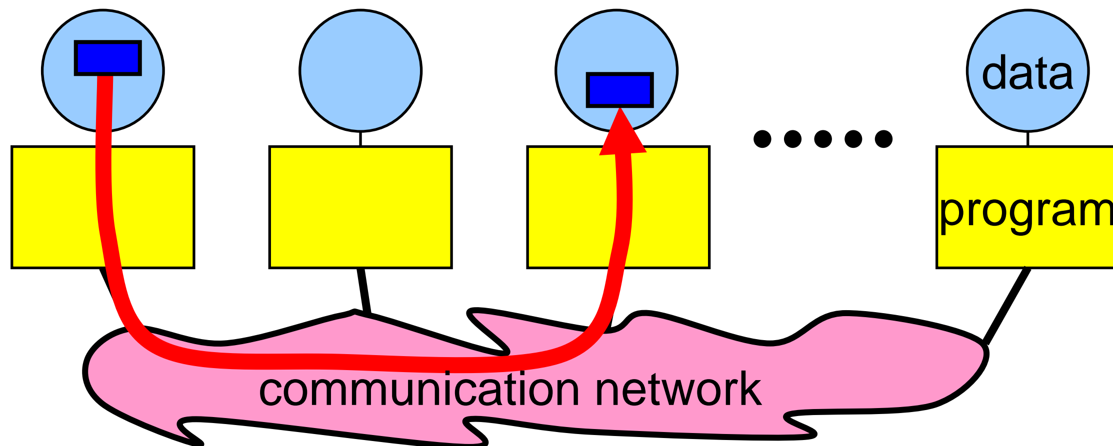
MPICH

- **A portable implementation of MPI developed at *Argonne National Labs* *(ANL) (USA)***
  - ***mpicc*, *mpicxx*, *mpif90*: compiler.**
  - ***mpirun*, *mpiexec*: portable scripts for launching parallel processes.**
  - ***installation***
    - download: http://www.mpich.org/
    - tar -xzvf mpich-x.x.x.tgz
    - cd mpich-x.x.x
    - ./configure --prefix=/public/home/leixu/software/mpich3.1.3
    - make
    - make install
    - vim ~/.bashrc
    - export PATH=/public/home/leixu/software/mpich3.1.3/bin/:$PATH
    - source ~/.bashrc
- **Another open-source popular implementation of MPI is *OpenMPI* (previously known as *LAM-MPI*).**

Message passing

- Messages are packets of data moving between processes
- Necessary information for the message passing system:
  - sending process
  - source location
  - source data type
  - source data size
  - receiving process
  - destination location
  - destination data type
  - destination buffer size

} i.e., the ranks



Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example： Computing Pi
  - Example： Matrix product in parallel
  - Advanced MPI collective operations

Hello world in C (1)

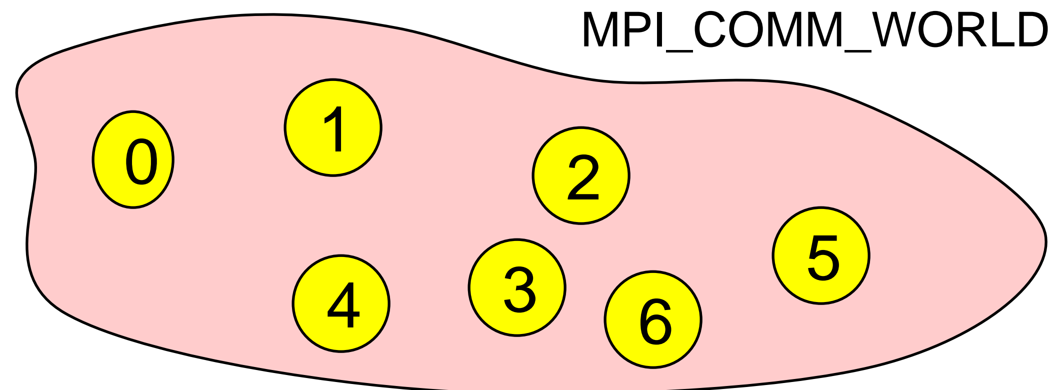
```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char **argv) {
5      int ierror, rank, size;
6      MPI_Init(&argc, &argv);
7      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9      printf("Hello world. I am %d out of %d.\n",rank, size);
10     MPI_Finalize();
11     return 0;
12 }
```

- All programs start with *MPI_Init()* and end with *MPI_Finalize()*.
- *MPI_Comm_size()* returns the total number *size* of processes involved in this parallel run. *MPI_Comm_rank()* returns through *rank*, the *id* of the process in this parallel run (*0<=myrank<size*).

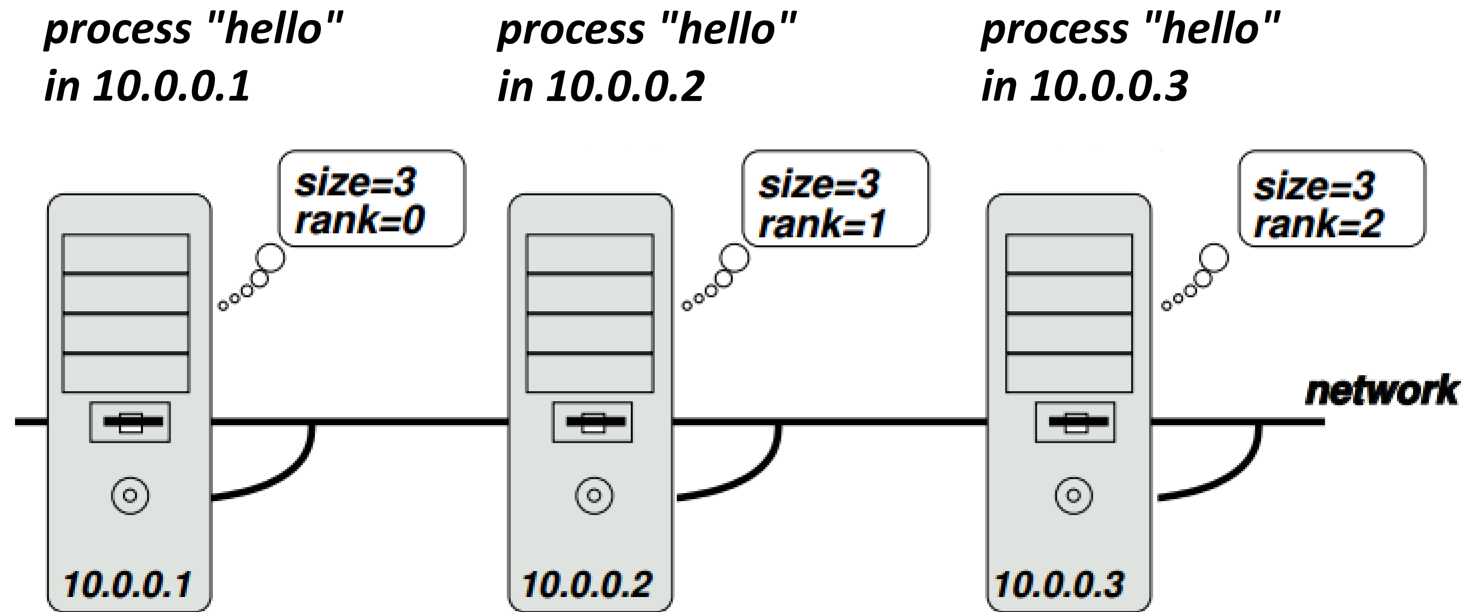
Hello world in C (2)

Communicator MPI_COMM_WORLD

- All processes of an MPI program are members of the default **communicator** **MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined **handle** in mpi.h and mpif.h.
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



Hello world in C (3)



- At the moment of launching the program in parallel (we will see how it is done below) a copy of the program starts execution in each of the selected nodes. In the figure it runs on 3 nodes.
- Each process obtains a unique *id* (usually called *rank*, *myrank*.)
- *Oversubscription:* In general we can have more than one *process* per *processor* (but it may not be useful, though).

Hello world in C (4)

- **If we compile and execute *hello*, then when running we obtain the normal output.**
  1. **[rlchen@chen-x280]$** mpicc -o hello hello.c
  2. **[rlchen@chen-x280]$** ./hello
  3. Hello world. I am 0 out of 1.
- **In order to run it on several nodes we generate a *machine.dat* file, with the processors names one per line.**
  1. **[rlchen@chen-x280]$** cat ./machine.dat
  2. node1
  3. node2
  4. **[rlchen@chen-x280]$** mpirun -np 3 -machinefile ./machine.dat ./hello

- **The *mpirun* script, which is part of the MPICH distribution, launches a copy of *hello* in the processor where *mpirun* has been called and two processes in the nodes corresponding to the first two lines of *machine.dat*.**

Master/Slave strategy with SPMD (in C)

```
1  // ......
2  int main(int argc, char **argv) {
3     int ierror, rank, size;
4     MPI_Init(&argc, &argv);
5     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6     MPI_Comm_size(MPI_COMM_WORLD, &size);
7     // ......
8     If (0 == rank) {
9        /* master code */
10    } else {
11       /* slave code */
12    }
13    // ......
14    MPI_Finalize();
15    return 0;
16 }
```

MPI function call format

- **C:**
  1. ***int ierr = MPI_Xxxxx(parameter, ...);***
  2. ***MPI_Xxxxx(parameter, ...);***

- **Fortan:**
  1. ***CALL MPI_XXXX(parameter, ..., ierr);***

Error codes

- Error codes are rarely used.
- Proper usage is like this:

```
1 ierror = MPI_Xxxx(parameter, ...);
2 if (ierror != MPI_SUCCESS) {
3     /* deal with failure */
4     abort();
5 }
```

MPI is small - MPI is large

Moderately complex programs can be written with *just 6 functions*:

- *MPI_Init*: It's used once at *initialization*.
- *MPI_Comm_size:* Identify *how many* processes participate in this parallel run.
- *MPI_Comm_rank* : Identify the *id* of my process in the parallel run.
- *MPI_Finalize* : *Last* function to be called. Ends MPI.
- *MPI_Send* : *Sends* a message to another process (point to point).
- *MPI_Recv* : *Receives* a message sent by other process.

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example：Computing Pi
  - Example：Matrix product in parallel
  - Advanced MPI collective operations

Operating system

MPI Data Type	C/C++ Data Type
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Send a message (1)

- **Template:**

  MPI_Send(address, length, type, destination, tag, communicator)

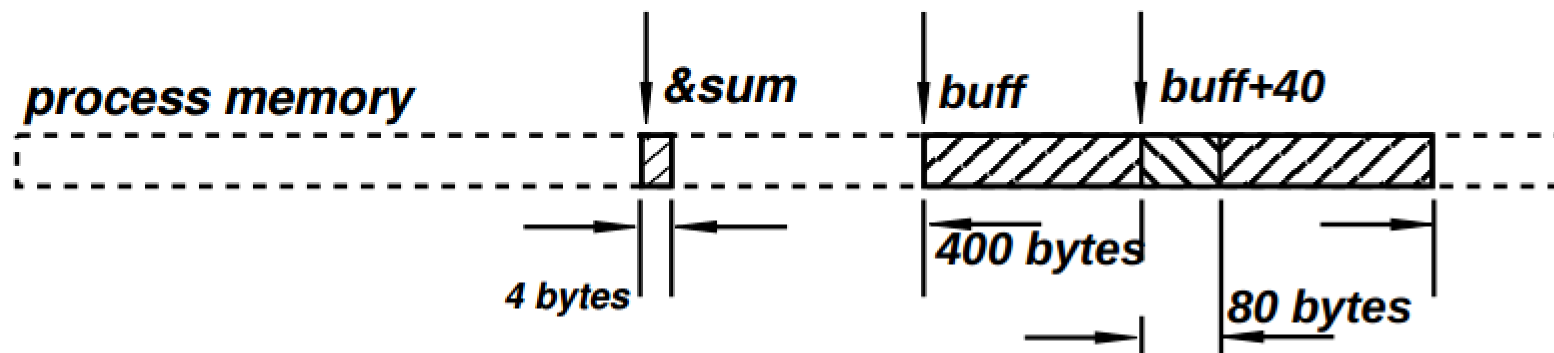
- **C:**

  ierr = MPI_Send(&sum, 1, MPI_FLOAT, 0, mtag1, MPI_COMM_WORLD);

- **Fortran (note extra parameter):**

  call MPI_SEND(sum, 1, MPI_REAL, 0, mtag1, MPI_COMM_WORLD, ierr);

Send a message (2)




```
1 int buff[100];
2 // Fill buff ...
3 for (int j = 0; j < 100; j++) buff[j] = j;
4 ierr = MPI_Send(buff, 100, MPI_INT, 0, mtag1, MPI_COMM_WORLD);
5 int sum;
6 ierr = MPI_Send(&sum, 1, MPI_INT, 0, mtag2, MPI_COMM_WORLD);
7 ierr = MPI_Send(buff + 40, 20, MPI_INT, 0, mtag3,  MPI_COMM_WORLD);

8 /* Error! Region sent extends, beyond the end of buff */
9 ierr = MPI_Send(buff + 80, 40, MPI_INT, 0, mtag4, MPI_COMM_WORLD);
```

Receive a message (1)

- **Template:**

  MPI_Recv(address, length, type, source, tag, communicator, status)

- **C:**

  ierr = MPI_Recv(&result, 1, MPI_FLOAT, MPI_ANY_SOURCE, mtag1,

  MPI_COMM_WORLD, &status);

- **Fortran (note extra parameter):**

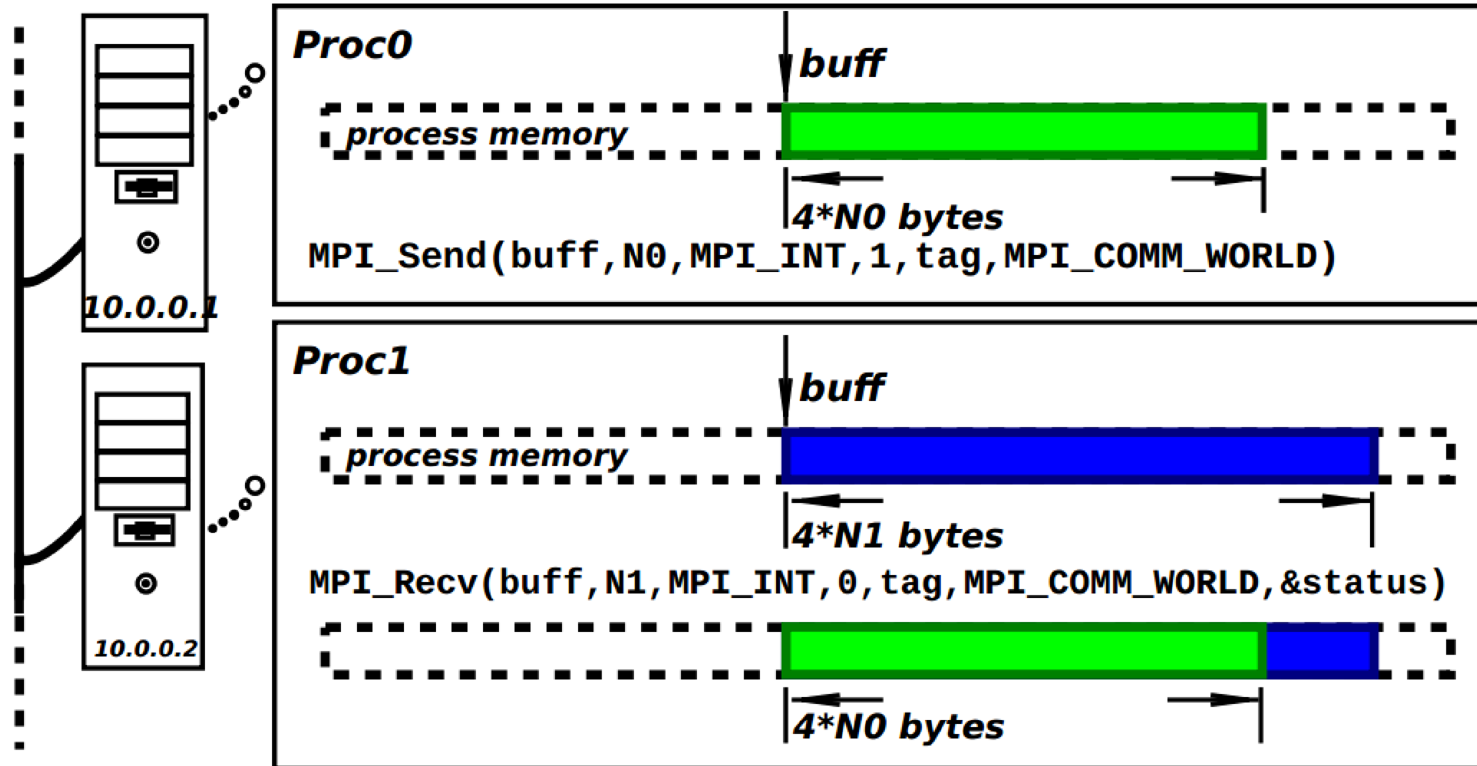
  call MPI_RECV(result, 1, MPI_REAL, MPI_ANY_SOURCE, mtag1,

  MPI_COMM_WORLD, status, ierr);

Receive a message (2)

- (*address*, *length*) reception buffer
- *type* standard MPI type:
  1. C: *MPI_FLOAT*, *MPI_DOUBLE*, *MPI_INT*, *MPI_CHAR*
  2. Fortran: *MPI_REAL*, *MPI_DOUBLE_PRECISION*, *MPI_INTEGER*, *MPI_CHARACTER*
- (*source*, *tag*, *communicator*): selects message
- *status*: Allows inspection of the data *effectively received* (e.g. length)

Receive a message (3)



OK if ***N1 >= N0***

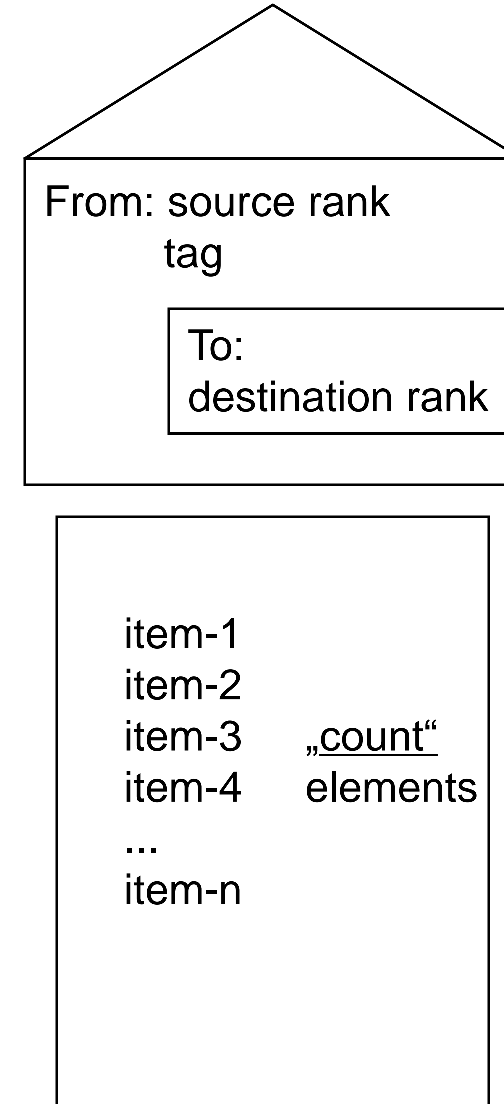
Receive a message (4)

- *tag*: message indentifier
- *communicator* : Process group, for instance *MPI_COMM_WORLD*
- *Status*: source, tag, and length of the received message
- Wildcards: *MPI_ANY_SOURCE*, *MPI_ANY_TAG*

Receive a message (5)

Communication Envelope

- Envelope information is returned from MPI_RECV in *status*.
- C: status.MPI_SOURCE
  status.MPI_TAG
  count via MPI_Get_count()
- Fortran: status(MPI_SOURCE)
  status(MPI_TAG)
  count via MPI_GET_COUNT()



Point-to-point communication

Each *send* must be balanced by a receive in the corresponding node *recv*

```
1  if (myid==0) {
2     for(i = 1; i < numprocs; i++)
3        MPI_Recv(&result, 1, MPI_FLOAT, MPI_ANY_SOURCE, mtag1, MPI_COMM_WORLD, &status);
4  } else {
5     MPI_Send(&sum, 1, MPI_FLOAT, 0, mtag1, MPI_COMM_WORLD);
6  }
```

When a message is received?

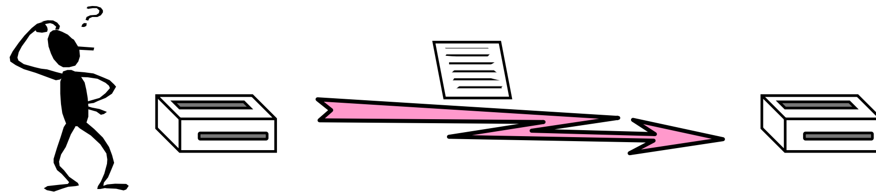
When a posted *receive* matches the *"envelope"* of the message:
envelope = source/destination, tag, communicator

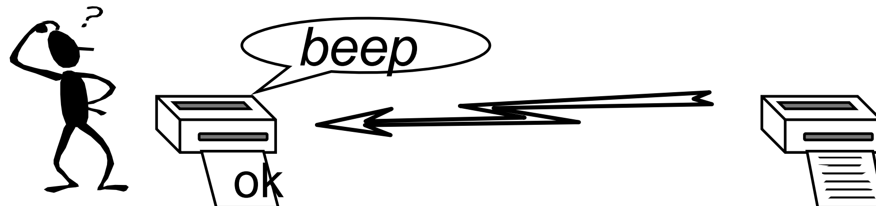
- size(receive buffer) < size(data sent) → *error*
- size(receive buffer) ≥ size(data sent) → *OK*
- types don't match → *error*

Dead-lock (1)

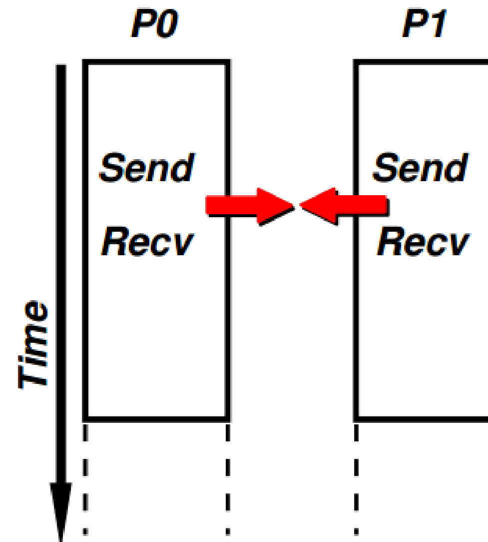
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.





Dead-lock (2)



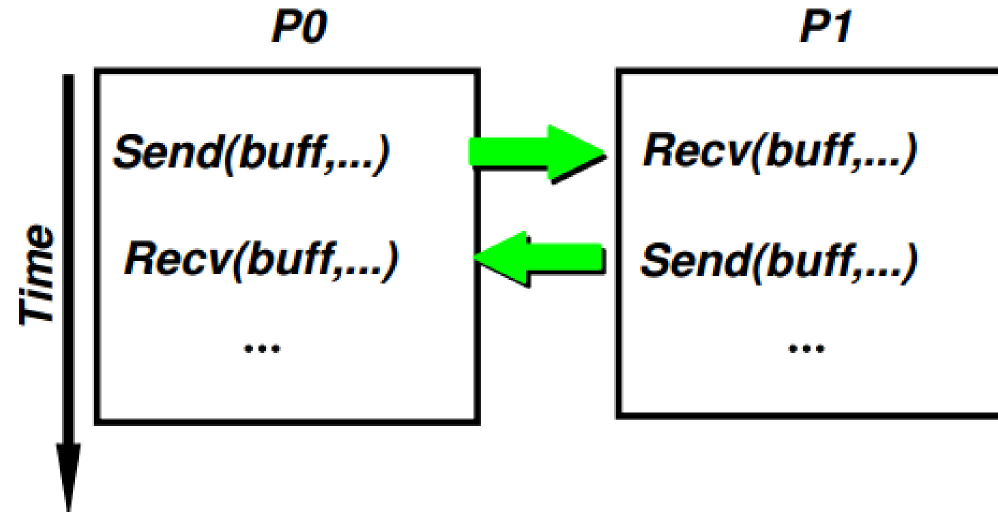
1. MPI_Send(buff, length, MPI_FLOAT, !myrank, tag, MPI_COMM_WORLD);
2. MPI_Recv(buff, length, MPI_FLOAT, !myrank, tag, MPI_COMM_WORLD, &status);

***!myrank*: Common C language to represent the *other* process. (1 → 0, 0 → 1).**

Also *1-myrank* or *(myrank? 0 : 1)*

***MPI_Send* and *MPI_Recv* are *blocking*, This means that code execution doesn't advance until the sending/reception is *completed*.**

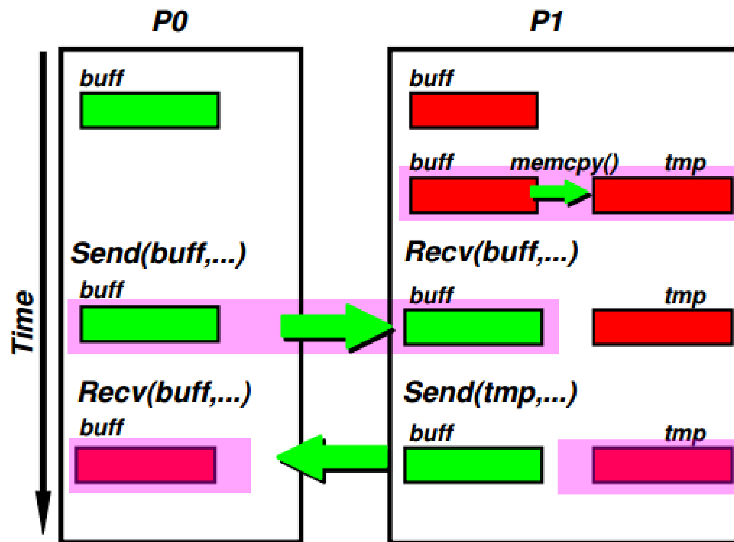
Correct calling order (1)




```
1  if (!myrank) {
2     MPI_Send(buff, length, MPI_FLOAT, !myrank, tag, MPI_COMM_WORLD);
3     MPI_Recv(buff, length, MPI_FLOAT, !myrank, tag, MPI_COMM_WORLD, &status);
4  } else {
5     MPI_Recv(buff, length, MPI_FLOAT, !myrank, tag, MPI_COMM_WORLD, &status);
6     MPI_Send(buff, length, MPI_FLOAT, !myrank, tag, MPI_COMM_WORLD);
7  }
```

Correct calling order (2)

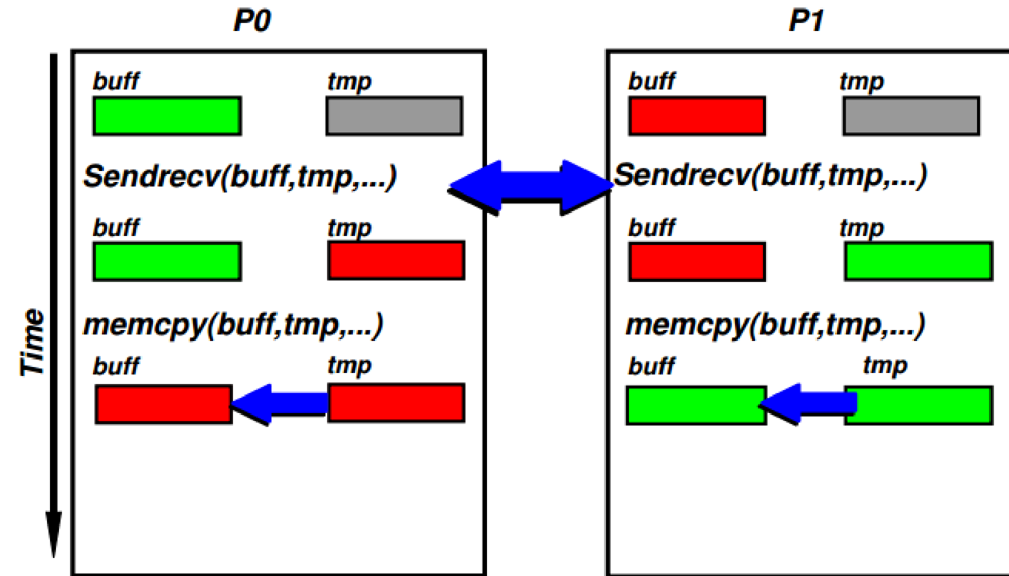
The previous code erroneously *overwrites* the reception buffer. We need a *temporal buffer* .




```
1  if (!myrank) {
2      MPI_Send(buff, length, MPI_FLOAT, !myrank, tag,
           MPI_COMM_WORLD);
3      MPI_Recv(buff, length, MPI_FLOAT, !myrank, tag,
           MPI_COMM_WORLD, &status);
4  } else {
5      float *tmp =new float[length];
6      memcpy(tmp, buff, length * sizeof(float));
7      MPI_Recv(buff, length, MPI_FLOAT, !myrank,
           tag, MPI_COMM_WORLD, &status);
8      MPI_Send(tmp, length, MPI_FLOAT, !myrank, tag,
           MPI_COMM_WORLD);
```

Correct calling order (3)

```
MPI_Sendrecv(void *sendbuf, int sendcount,
             MPI_Datatype sendtype,
             int  dest, int sendtag,
             void *recvbuf, int recvcount,
             MPI_Datatype recvtype,
             int  source,  int recvtag,
            MPI_Comm comm,
            MPI_Status status);
```



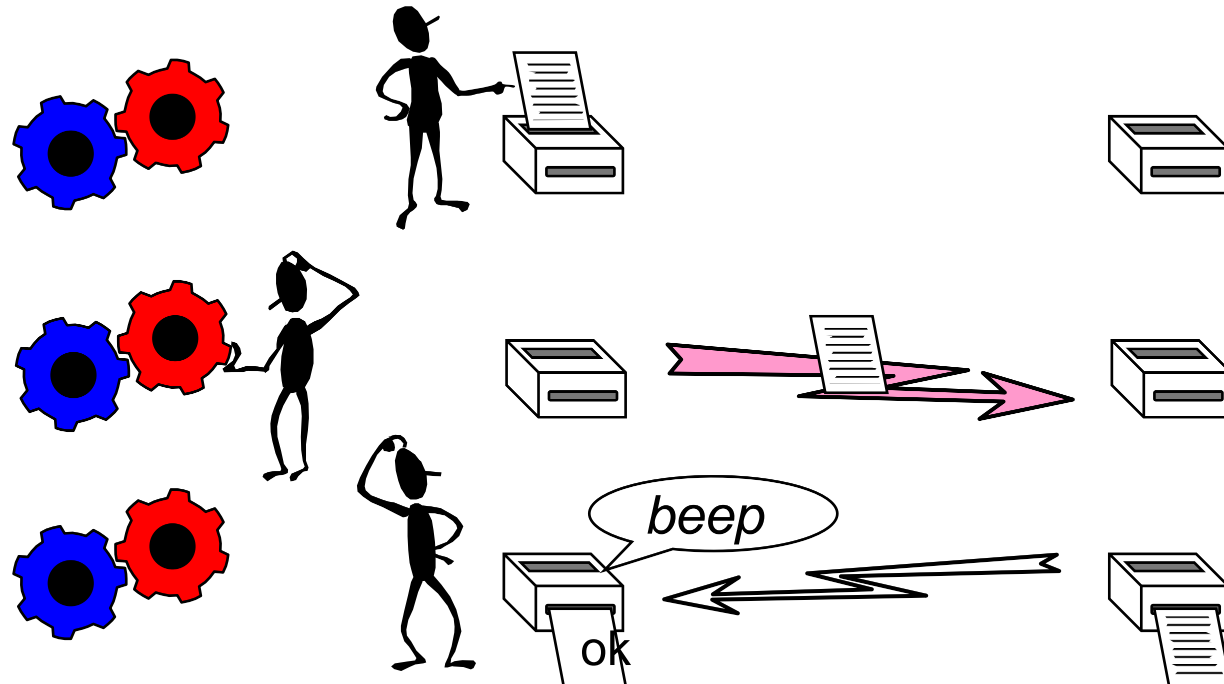
MPI_Sendrecv: sends and receives *at the same time*.

```
1  float *tmp = new float[length];
2  int MPI_Sendrecv(buff, length, MPI_FLOAT, !myrank, stag, tmp, length, MPI_FLOAT, !myrank, rtag, MPI_COMM_WORLD,
&status)
3  memcpy(buff, tmp, length * sizeof(float));
4  delete[] tmp;
```

Correct calling order (4)

Non-Blocking Operations

- Non-blocking operations return immediately and allow the process to perform other work.



Correct calling order (5)

- **Template:**

  MPI_Isend(sbuf, count, datatype, dest, tag, comm, request);

- **C:**

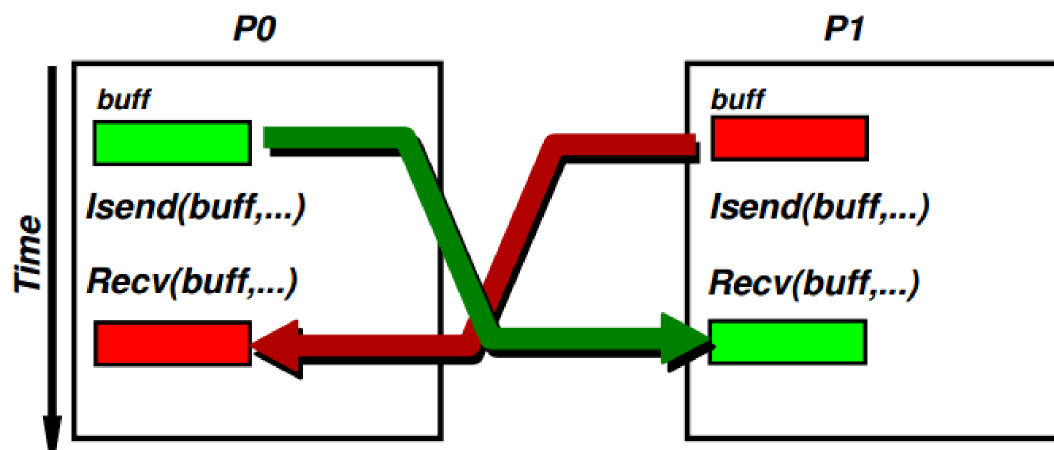
  ierr = MPI_Isend(&sum, 1, MPI FLOAT, 0, mtag1, MPI_COMM_WORLD , request);

  MPI_Irecv(rbuf, count, datatype, source, tag, comm, request);

- **C:**

  ierr = MPI_Irecv(&result, 1, MPI_FLOAT, 1, mtag1, MPI_COMM_WORLD, request);

Correct calling order (6)



- The code is *the same* for the two processes.
- Needs *auxiliary buffer* (not shown here)

Use *non-blocking* send/receive

```
1  MPI_Request request;
2  MPI_Isend(. . . ., request);
3  MPI_Recv(. . . .);
4  while(1) {
5     MPI_Test(request, flag, status);
6     if(flag)   break;
```

Use *non-blocking* send/receive

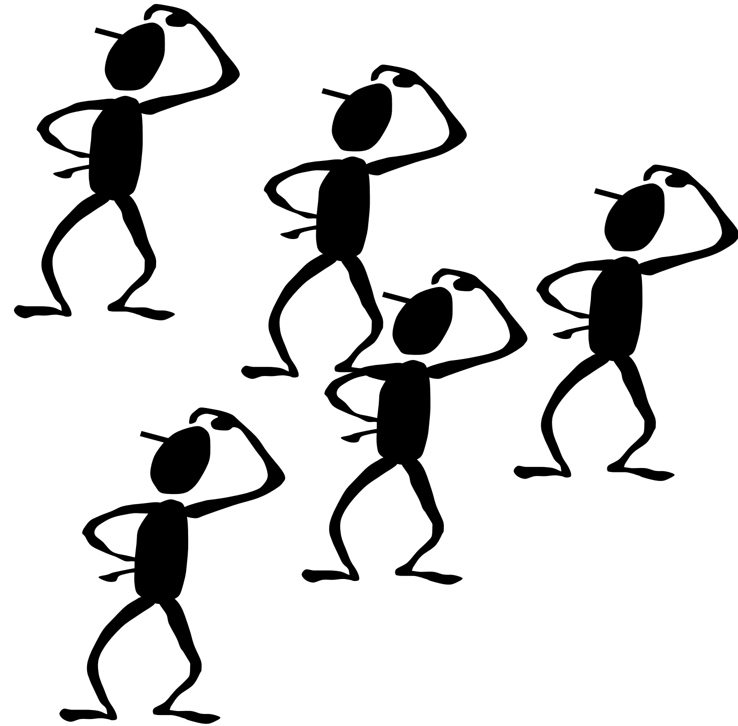
```
1  MPI_Request request;
2  MPI_Isend(. . . ., request);
3  MPI_Recv(. . . .);
4  MPI_Wait(request, status);
```

Contents

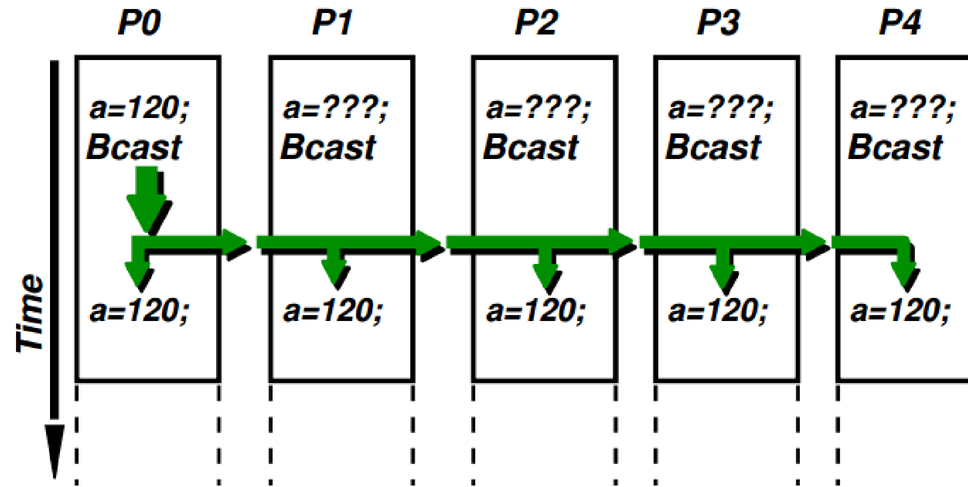
- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example： Computing Pi
  - Example： Matrix product in parallel
  - Advanced MPI collective operations

Message broadcast (1)

- A one-to-many communication.



Message broadcast (2)



Template:

MPI_Bcast(address, length, type, source, comm)

C:

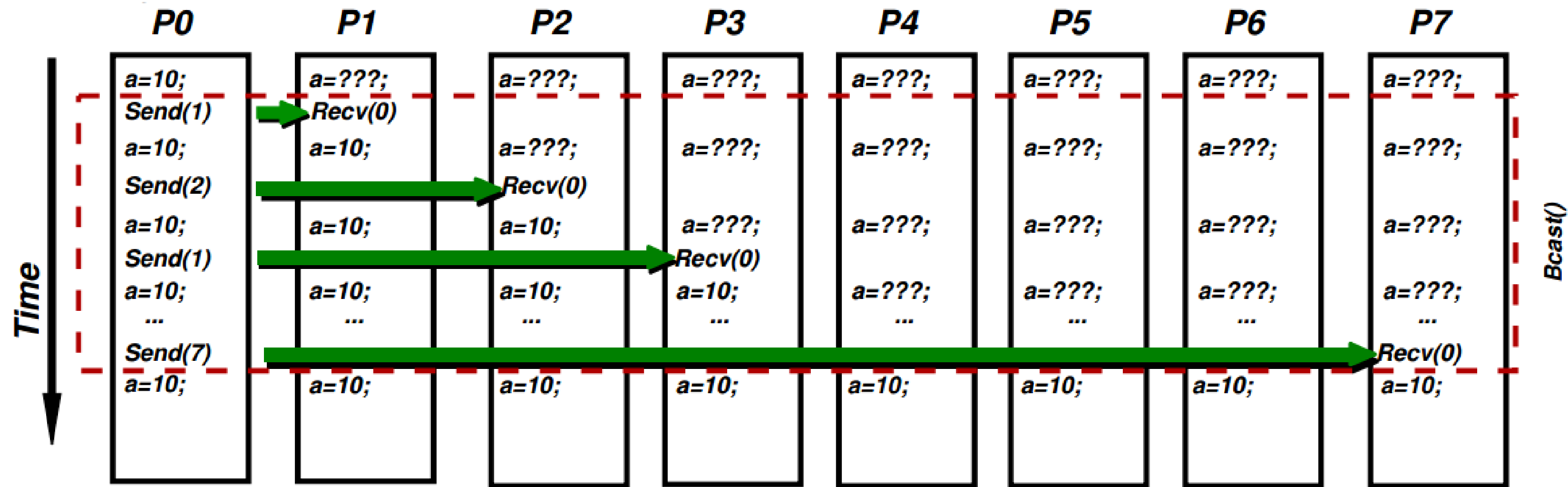
ierr = MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);

Fortran:

call MPI_Bcast(a, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

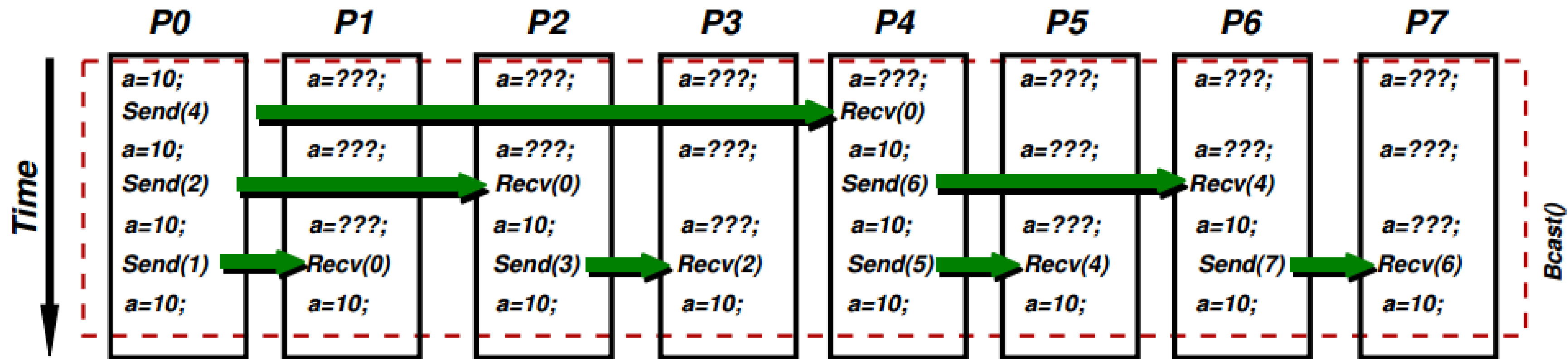
Message broadcast (3)

MPI_Bcast() is conceptually equivalent to a series of Sends/Receives, but it may be much more efficient.

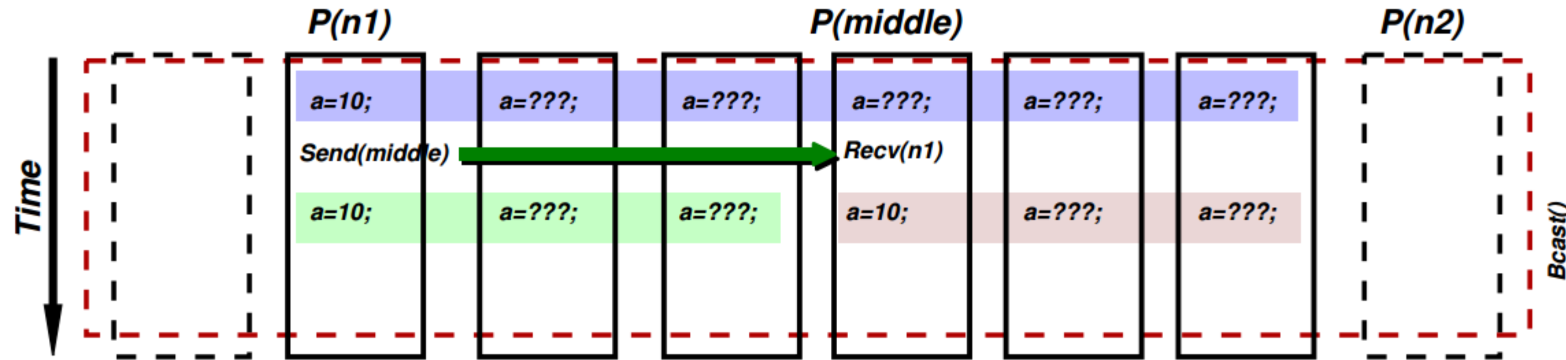



```
1 if (!myrank) {
2     for (int j = 1; j < numprocs; j++)   MPI_Send(buff, . . ., j);
3 } else {
4     MPI_Recv(buff, . . ., 0, . . .);
```

Message broadcast (4)



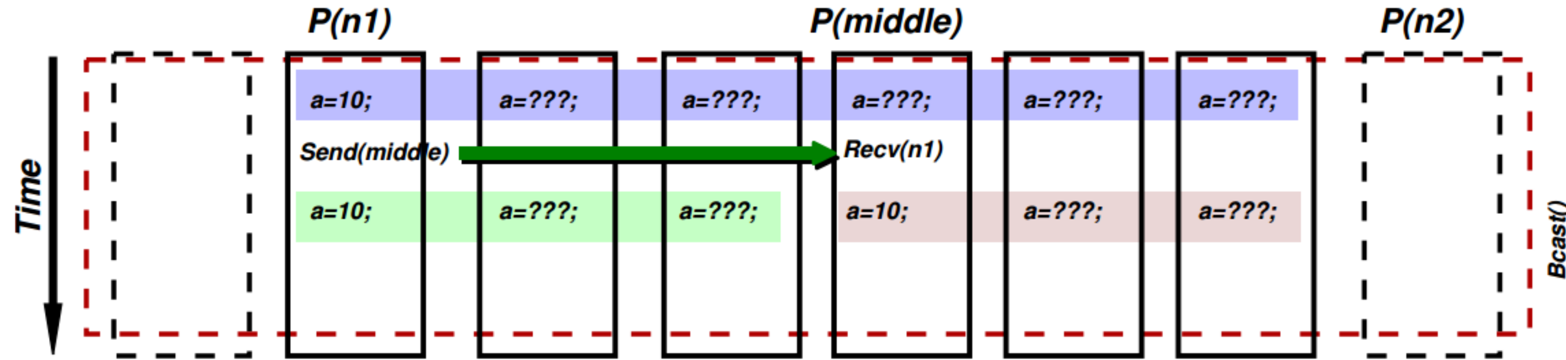
Message broadcast (5)



Efficient implementation of *MPI_Bcast()* with Send/Receives.

1. At every moment we are in process *myrank* and we have an interval *[n1,n2)* such that *myrank* is in *[n1,n2)*. Remember that *[n1,n2)={j such that n1 <= j < n2}* .
2. Initially *n1=0*, *n2=NP* (number of processors).
3. In each step *n1* sends to *middle=(n1+n2)/2* and this will receive.
4. In the next step we update the range to *[n1,middle)* if *myrank < middle* or else *[middle,n2)*.
5. The process ends when *n2-n1==1*

Message broadcast (6)



Pseudocode:

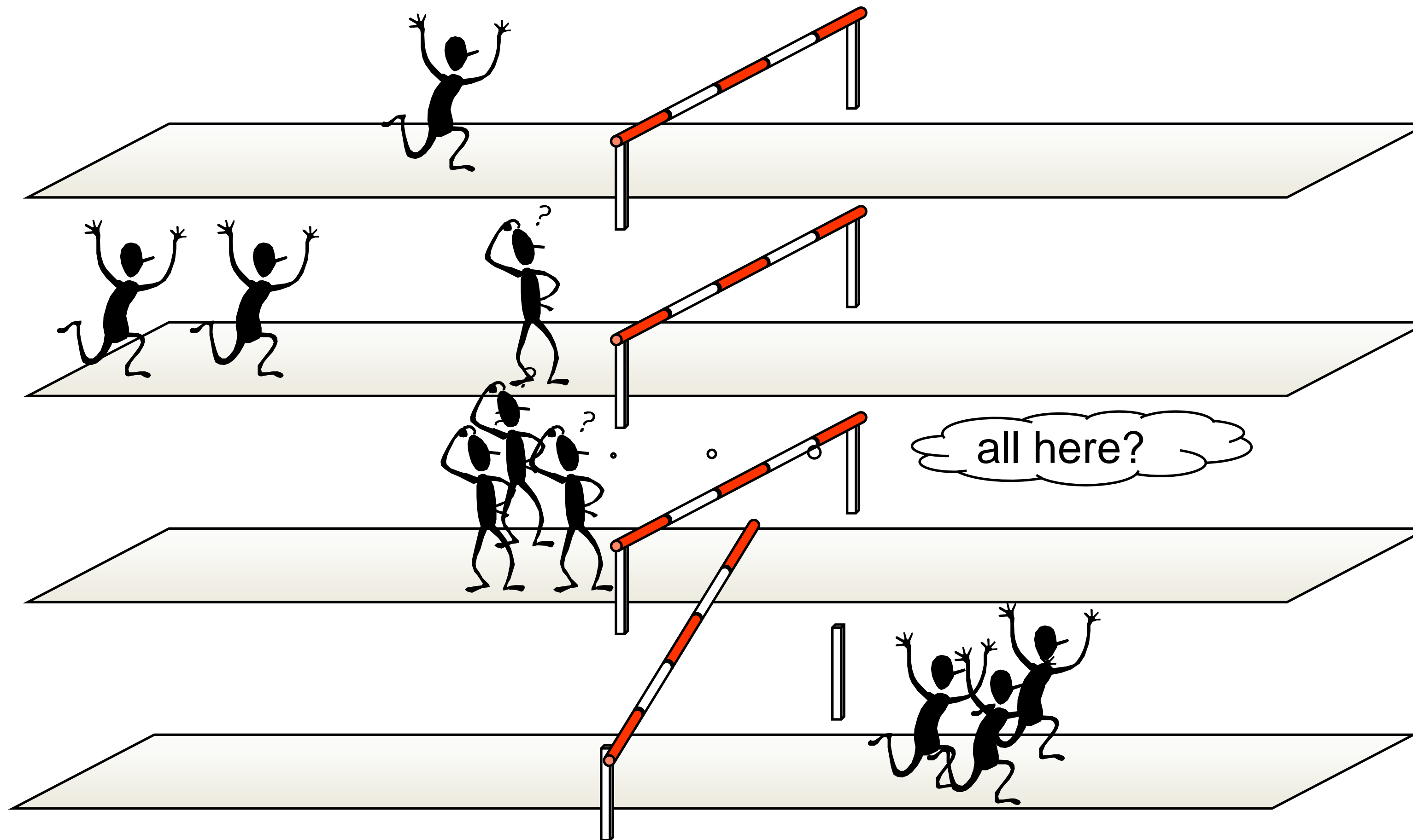
```
1  int n1 = 0, n2 = numprocs;
2  while (1) {
3      int middle = (n1 + n2) / 2;
4      if (myrank == n1) MPI_Send(buff, . . ., middle, . . .);
5      else if (myrank == middle)   MPI_Recv(buff, . . ., n1, . . .);
6      if (myrank < middle)   n2 = middle;
7      else n1 = middle;
```

Collective calls

These routines are *collective* (in contrast to the *point-to-point* *MPI_Send()* and *MPI_Recv()*). All processors in the communicator must call the function, and normally the collective call imposes an *implicit barrier* in the code execution.

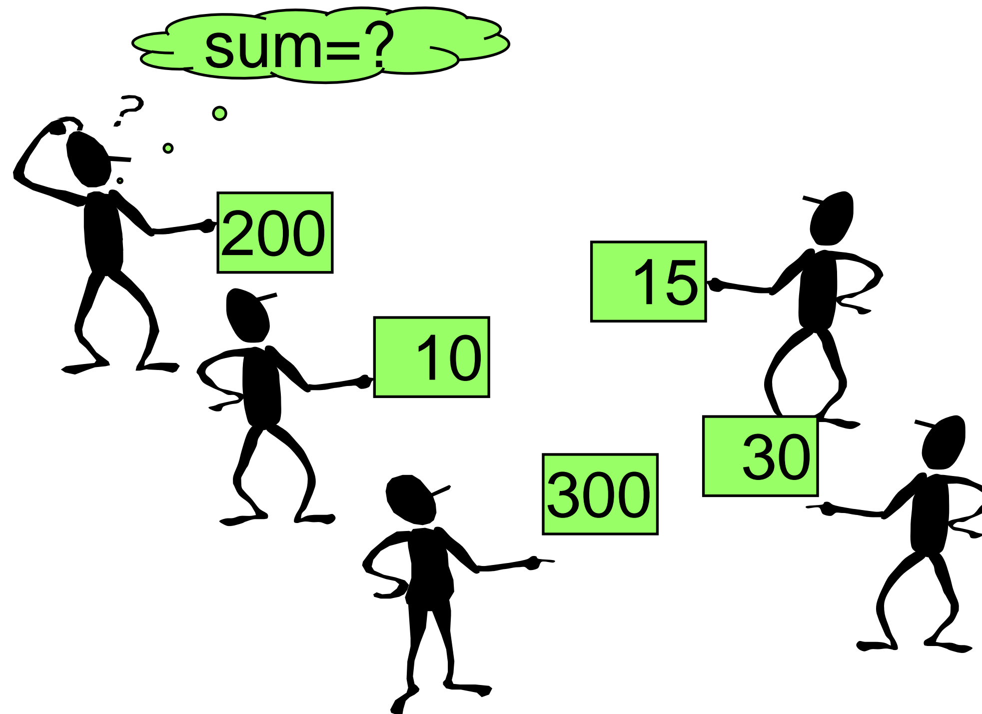
```
int MPI_Barrier(MPI_Comm comm)
```

MPI_Barrier()

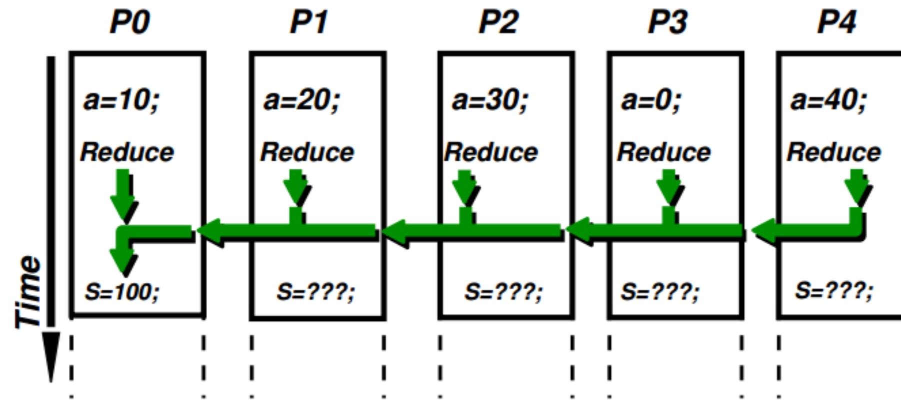


Global reduction (1)

- Combine data from several processes to produce a single result.



Global reduction (2)



- **Template:**

  MPI_Reduce(s_address, r_address, length, type, operation, destination, comm)

- **C:**

  ierr = MPI_Reduce(&a, &s, 1, MPI_FLOAT, MPI SUM, 0, MPI_COMM_WORLD);

- **Fortran:**

  call MPI_REDUCE(a, s, 1, MPI_REAL, MPI_SUM, 0, MPI COMM WORLD, ierr)

MPI associative global operations (1)

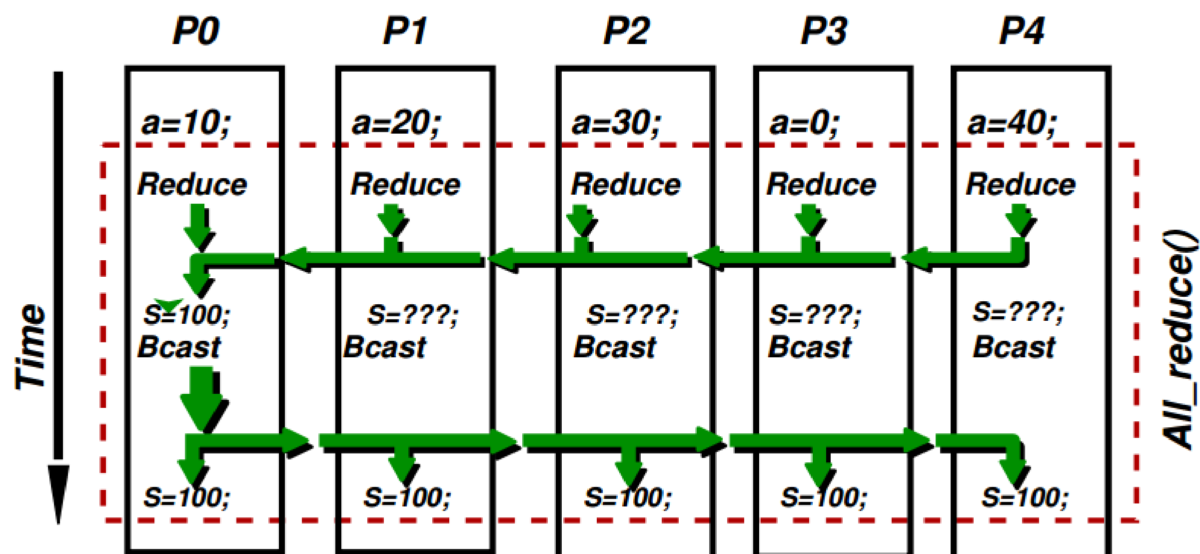
Reduction functions apply a *binary associative operation* to a set of values.

Typically,

- *MPI_SUM* sum
- *MPI_MAX* maximum
- *MPI_MIN* minimum
- *MPI_PROD* product
- *MPI_AND* boolean
- *MPI_OR* boolean

It is not specified the order in which the binary operations are done, so that it is very important that the function must be *associative*.

MPI associative global operations (2)



If the result of the reduction is needed in *all* processors, then we must use

MPI_Allreduce(s_address, r_address, length, type, operation, comm)

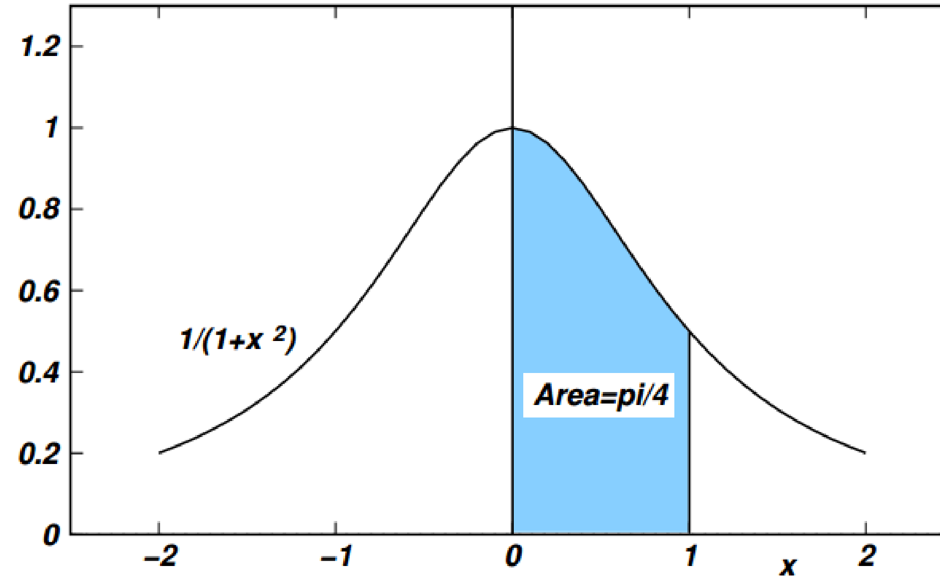
This is conceptually equivalent to a *MPI_Reduce()* followed by a *MPI_Bcast()*.

Warning: MPI_Bcast() and MPI_Reduce() are *collective* functions. *All processors must call them!!*

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example: Computing Pi
  - Example: Matrix product in parallel
  - Advanced MPI collective operations

Computing Pi by numerical integration

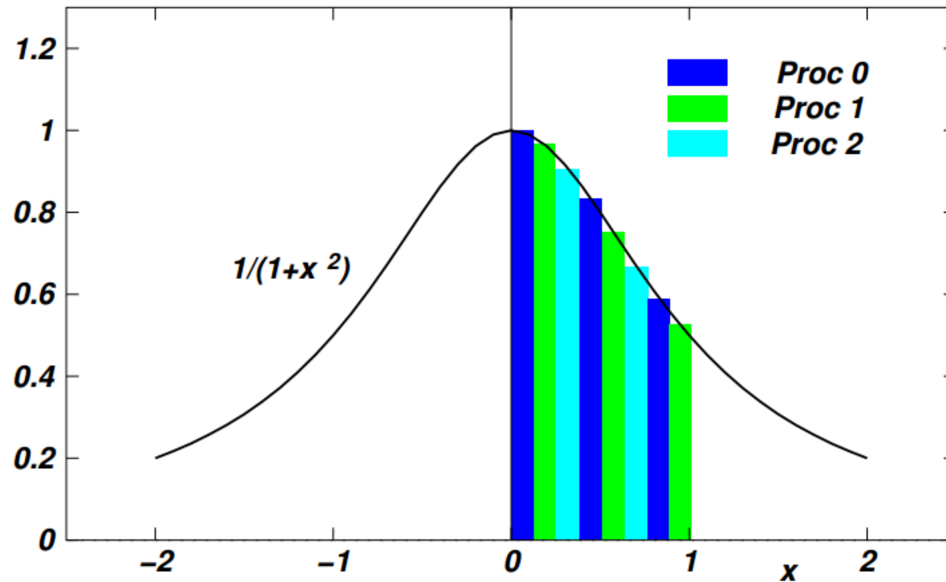


$$\tan\left(\frac{\pi}{4}\right) = 1 \quad \rightarrow \quad atan(1) = \frac{\pi}{4}$$

$$\frac{d\ atan(x)}{dx} = \frac{1}{1+x^2}$$

$$\frac{\pi}{4} = atan(1) - atan(0) = \int_0^1 \frac{1}{1+x^2}dx$$

Numerical integration (1)



Using the *midpoint rule*

numprocs **= number of processors**

n **= Number of intervals (may be a multiple of *numprocs* or not)**

h = ***1/n*****= interval width**

Numerical integration (2)

```
1  // Inicialization (rank,size) . . .
2  while (1) {
3     // Master (rank==0) read number of intervals 'n' . . .
4     // Broadcast 'n' to computing nodes . . .
5     if (n==0) break;
6     // Compute 'mypi' (local contribution to 'pi') . . .
7     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,  MPI_SUM, 0, MPI_COMM_WORLD);
8     // Master reports error between computed pi and exact
9  }
10 MPI_Finalize();
```

Code (1)

```
1  //**********************************************************
2  //  compute pi by integrating f(x) = 4/(1 + x**2)
3  //  Each node:
4  //   1) receives the number of rectangles used in the approximation.
5  //   2) calculates the areas of its rectangles.
6  //  3) Synchronizes for a global summation.
7  //  Node 0 prints the result.
8  //  Variables:
9  //  pi         the calculated result
10 //  n          number of points of integration.
11 //  x          midpoint of each rectangle’s interval
12 //  f          function to integrate
13 //  sum, pi   area of rectangles
14 //  tmp        temporary scratch space for global summation
15 //  i          do loop index
16 //**********************************************************
```

Code (2)

```
17 #include <mpi.h>
18 #include <cstdio>
19 #include <cmath>
20
21 // The function to integrate
22 double f(double x) { return 4. / (1. + x * x); }
23
24 int main(int argc, char **argv) {
25
26    MPI_Init(&argc, &argv);     // Initialize MPI environment
27
28    int myrank;                          // Get the process number and assign it to the variable myrank
29    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
30
31    // Determine how many processes the program will run on and
32   // assign that number to size
33   int size;
34   MPI_Comm_size(MPI_COMM_WORLD, &size);
35
36    // The exact value
37    double PI = 4 * atan(1.0);
```

Code (3)

```
38 // Enter an infinite loop. Will exit when user enters n = 0
39 while (1) {
40      int n;
41      // Test to see if this is the program running on process 0,
42     // and run this section of the code for input.
43     if (!myrank) {
44         printf("Enter the number of intervals: (0 quits) > ");
45         scanf("%d",&n);
46     }
47
48     // The argument 0 in the 4th place indicates that
49     // process 0 will send the single integer n to every
50     // other process in processor group MPI-COMM-WORLD.
51     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
52
53     // If the user puts in a negative number for n we leave the program by branching to MPI_FINA
54     if (n < 0) break;
55
56  // Now this part of the code is running on every node and each one shares the same value
57 // of n. But all  other variables are local to each individual  process. So each process then calculates
58 // the each interval size.
```

Code (4)

```
60      //*****************************************************
61      // Main Body : Runs on all processors
62      //*****************************************************
63      // even step size h as a function of partitions
64      double h = 1.0 / double(n);
65      double sum = 0.0;
66      for (int i = myrank + 1; i <= n; i += size) {
67      double x = h * (double(i) - 0.5);
68      sum = sum + f(x);
69    }
70    double pi, mypi = h * sum; // this is the total area in this process, (a partial sum.)
71
72    // Each individual sum should converge also to PI, compute the max error
73    double error, my_error = fabs(size * mypi - PI);
74    MPI_Reduce(&my_error, &error, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
75
76    // After each partition of the integral is calculated, we collect all the partial sums.
77    // The MPI_SUM argument is the operation that adds all the values of mypi into pi of
78    // process 0 indicated by the 6th argument.
79    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

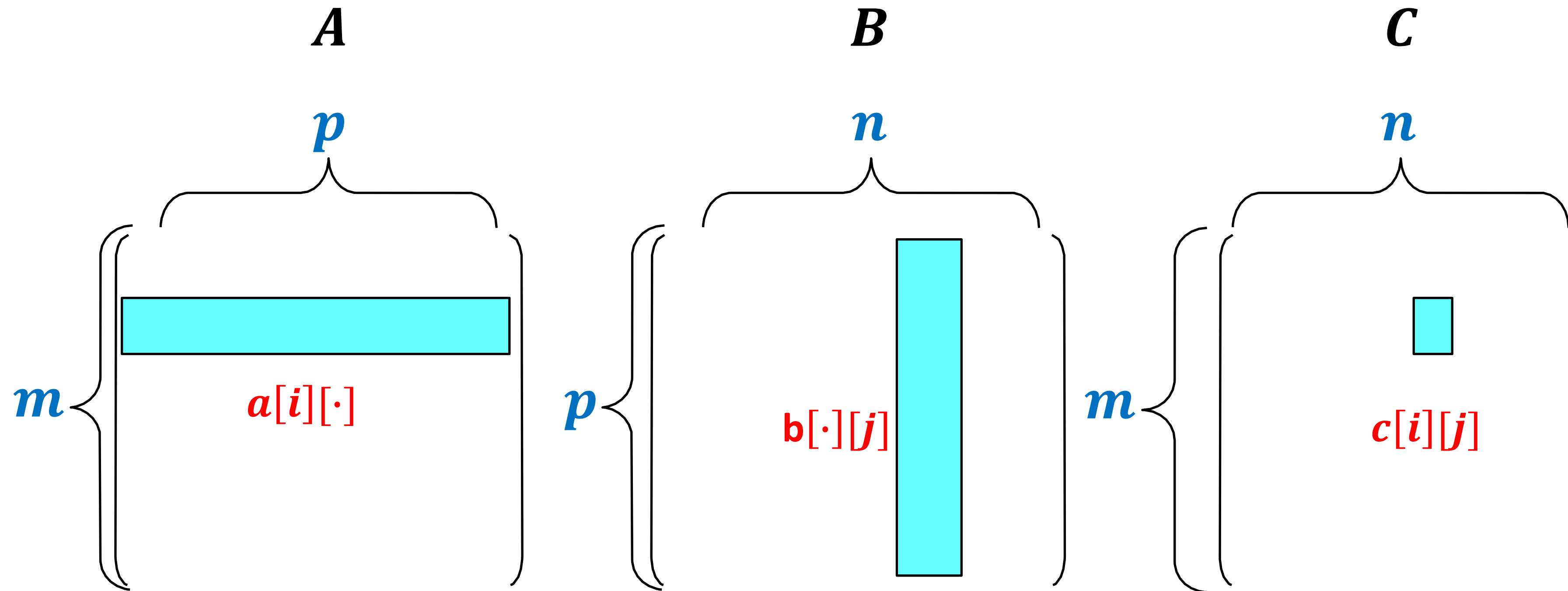
Code (5)

```
80        //*****************************************************
81        // Print results from Process 0
82        //*****************************************************
83
84        // Finally the program tests if myrank is node 0
85        // so process 0 can print the answer.
86        if (!myrank)
87            printf("pi is aprox: %f, (error %f, max err over procs %f)\n", pi, fabs(pi - PI), my_error);
88        // Run the program again.
89    }
90    // Branch for the end of program. MPI-FINALIZE will close
91    // all the processes in the active group.
92
93    MPI_Finalize();
94 }
```

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example：Computing Pi
  - Example：Matrix product in parallel
  - Advanced MPI collective operations

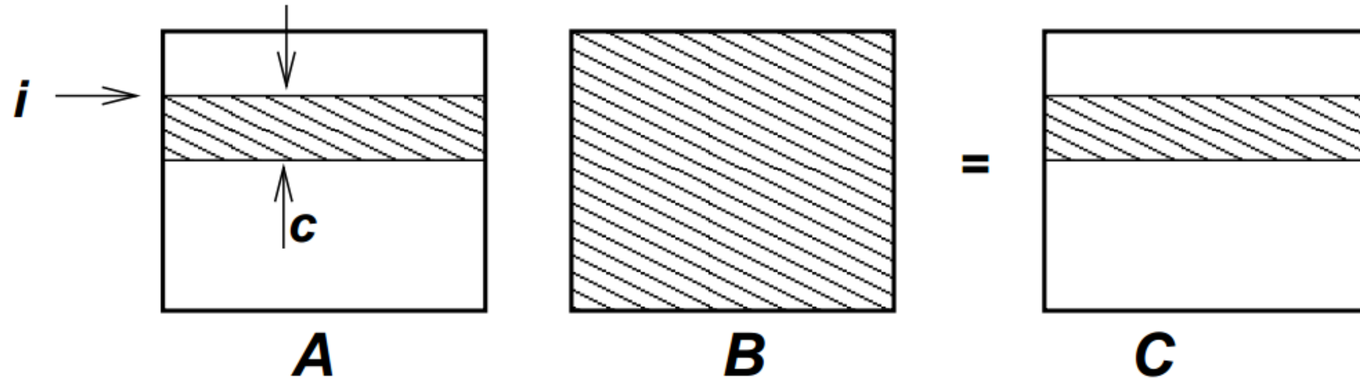
Matrix Product in Parallel (1)



$$C_{ij} = \sum_{0}^{p-1} A_{ik} B_{kj}$$

```
for( i = 0; i < m; i++ )
   for( j = 0; j < n; j++ )
      for( k = 0; k < p; k++ )
         c[i][j] = c[i][j] + a[i][k] * b[k][j]
```

Matrix Product in Parallel (2)



All nodes have all ***B***, and receive part of ***A*** (a range (chunk) of files ***A(i:i+n-1,:)***. The node makes the product ***A(i:i+n-1,:)*B*** and returns the result.

- Static load balance: needs to know the computing speed.

Complete code (1)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <mpi.h>
4   #include <time.h>
5   int main(int argc, char *argv[])
6   {
7     double start, stop;
8     int i, j, k, l;
9     int *a, *b, *c, *buffer, *ans;
10    int n = 10;
11    int rank, size, line;
12
13    MPI_Init(&argc, &argv);    //MPI Initialize
14    MPI_Comm_rank(MPI_COMM_WORLD, &rank);      //obtain the rank id
15    MPI_Comm_size(MPI_COMM_WORLD, &size);      //obtain the number of processes
16    line = n / size;      // divide the data into data chunk
17    a = (int*)malloc(sizeof(int) * n * n);
18    b = (int*)malloc(sizeof(int) * n * n);
19    c = (int*)malloc(sizeof(int) * n * n);
20    buffer = (int*)malloc(sizeof(int) * n * line);
21    ans = (int*)malloc(sizeof(int) * n * line);      // used to store results
22    if (rank == 0){      // master process
23        start = MPI_Wtime();
24        for(i = 0; i < n; i++){ // assign value
25          for(j = 0; j < n; j++){
26            a[i * n + j] = i;
27          }
28        }
29
30        for(i = 0; i < n; i++){
31          for(j = 0; j < n; j++){
32            b[i * n + j] = 2;
33          }
34        }
35
36        for (i = 1; i < size; i++){      // send b to each process
37          MPI_Send(b, n * n, MPI_INT, i, 0, MPI_COMM_WORLD);
38        }
39
40        for (i = 1; i < size; i++){    // send a chunk of a to each process
41          MPI_Send(a + (i - 1) * line * n, n * line, MPI_INT, i, 1, MPI_COMM_WORLD);
42        }
43
```

Complete code (2)

```
44          for (k = 1; k < size; k++){  // receive the answers
45            MPI_Recv(ans, line * n, MPI_INT, k, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
46            for (i = 0; i < line; i++){ // copy the results from ans to c
47              for (j = 0; j < n; j++){
48                c[((k - 1) * line + i) * n + j] = ans[i * n + j];
49              }
50            }
51          }
52          for (i = (size - 1) * line; i < n; i++){      // compute the left in a
53            for (j = 0; j < n; j++){
54              int temp = 0;
55              for (k = 0; k < n; k++){
56                temp += a[i * n + k] * b[k * n + j];
57              }
58              c[i * n + j] = temp;
59            }
60          }
61          printf("The matrix a:\n");
62          for(i = 0; i < n; i++){
63            for(j = 0; j < n; j++){
64              printf("%5d ", a[i * n + j]);
65            }
66            printf("\n");
67          }
68          printf("The matrix b:\n");
69          for(i = 0; i < n; i++){
70            for(j = 0; j < n; j++){
71              printf("%5d ", b[i * n + j]);
72            }
73            printf("\n");
74          }
75          printf("The matrix c=a*b:\n");
76          for(i = 0; i < n; i++){
77            for(j = 0; j < n; j++){
78              printf("%5d ", c[i * n + j]);
79            }
80            printf("\n");
81          }
82          stop = MPI_Wtime();
83          printf("rank:%d time:%lf s\n", rank, stop - start);
84        }else{
85          MPI_Recv(b, n * n, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
86          MPI_Recv(buffer, n * line, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Complete code (3)

```
87          // compute the results
88          for (i = 0; i < line; i++){
89            for (j = 0; j < n; j++){
90              int temp = 0;
91              for(k = 0; k < n; k++){
92                temp += buffer[i * n + k] * b[k * n + j];
93              }
94              ans[i * n + j] = temp;
95            }
96          }
97          MPI_Send(ans, line * n, MPI_INT, 0, 3, MPI_COMM_WORLD);      // send back the computed result
98      }
99
100     free(a);
101     free(b);
102     free(c);
103     free(buffer);
104     free(ans);
105
106     MPI_Finalize();
107     return 0;
108 }
109
```

Contents

- What is Supercomputer?
- What is Parallel Computing?
- Why Use Parallel Computing?
- What Parallel Computing Can do?
- How to do Parallel Computing?
  - Basic introduction of MPI
  - Basic use of MPI
  - Point to point communication
  - Global communication
  - Example：Computing Pi
  - Example：Matrix product in parallel
  - Advanced MPI collective operations

Advanced MPI collective operations

We have already seen the basic collective operations (*MPI_Bcast()* and *MPI_Reduce()*). Collective functions have the advantage that allow to perform complex common operations in simply and efficiently.

There are other collective operations, namely

- *MPI_Scatter(), MPI_Gather()*
- *MPI_Allgather()*
- *MPI_Alltoall()*

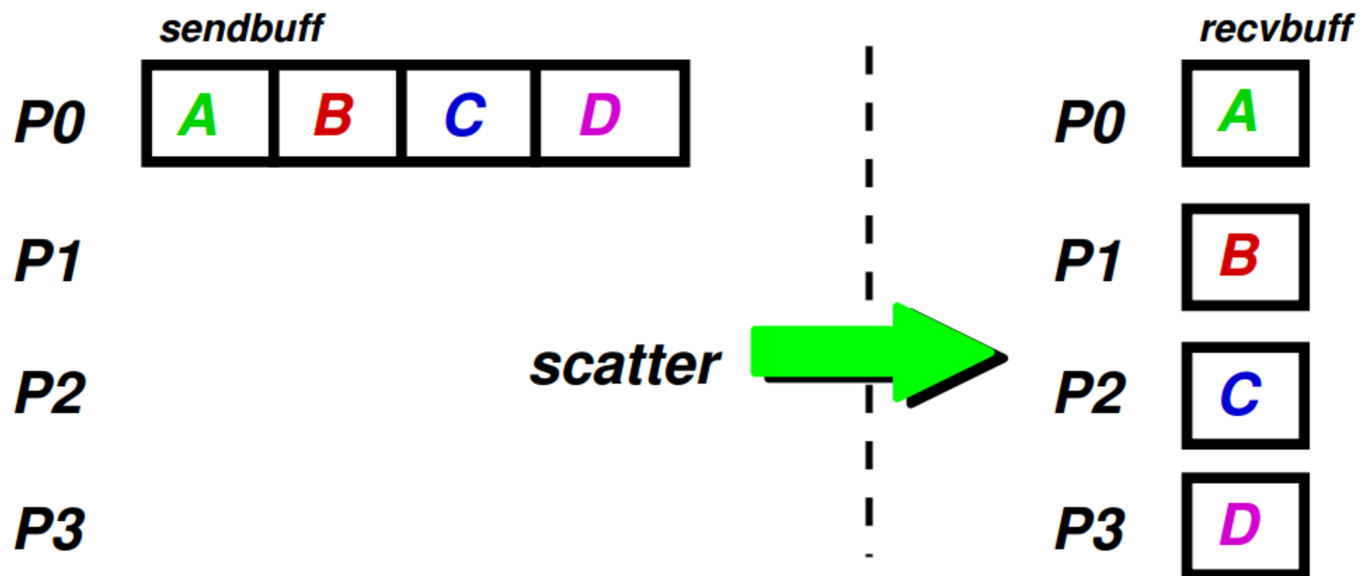
And their vectorized (varible length per processor) versions

- *MPI_Scatterv(), MPI_Gatherv()*
- *MPI_Allgatherv()*
- *MPI_Alltoallv()*

Scatter operations (1)

Sends a certain amount of data of the same size and type to the other processes (as in *MPI_Bcast()*, but the data to be sent is not the same to all processes).

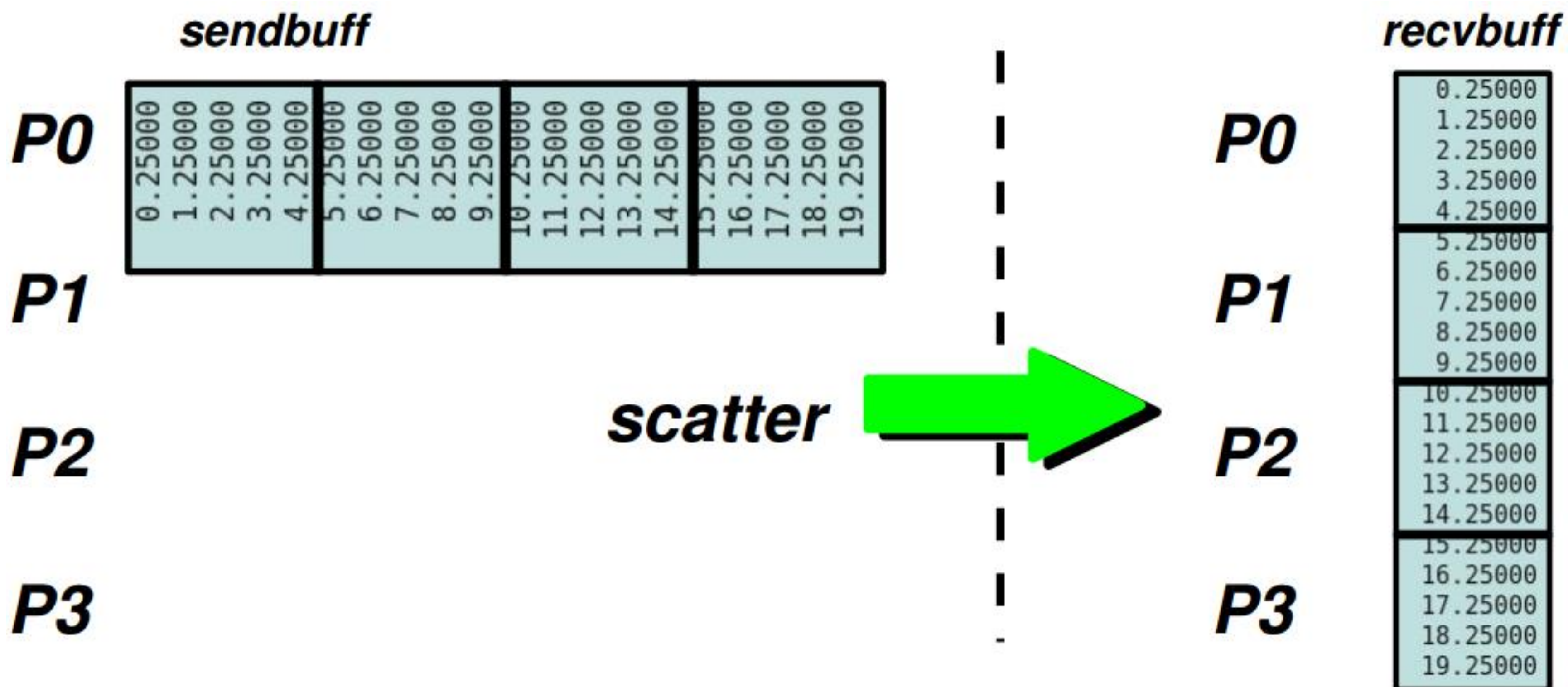
```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
                int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



Scatter operations (2)

```
1  #include <mpi.h>
2  #include <cstdio>
3  int main(int argc, char **argv) {
4     MPI_Init(&argc, &argv);
5     int myrank, size;
6     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7     MPI_Comm_size(MPI_COMM_WORLD, &size);
8     int N = 5; // Nbr of elements to send to each processor
9     double *sbuff = NULL;
10    if (!myrank) {
11       sbuff = new double[N * size]; // send buffer only in master
12        for (int j = 0; j < N * size; j++) sbuff[j] = j + 0.25;   // fills 'sbuff'
13     }
14     double *rbuff = new double[N]; // receive buffer in all procs
15     MPI_Scatter(sbuff, N, MPI_DOUBLE, rbuff, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
16     for (int j = 0; j < N; j++)   printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
17     MPI_Finalize();
18     if (!myrank) delete[ ] sbuff;
19     delete[ ] rbuff;
20  }
```

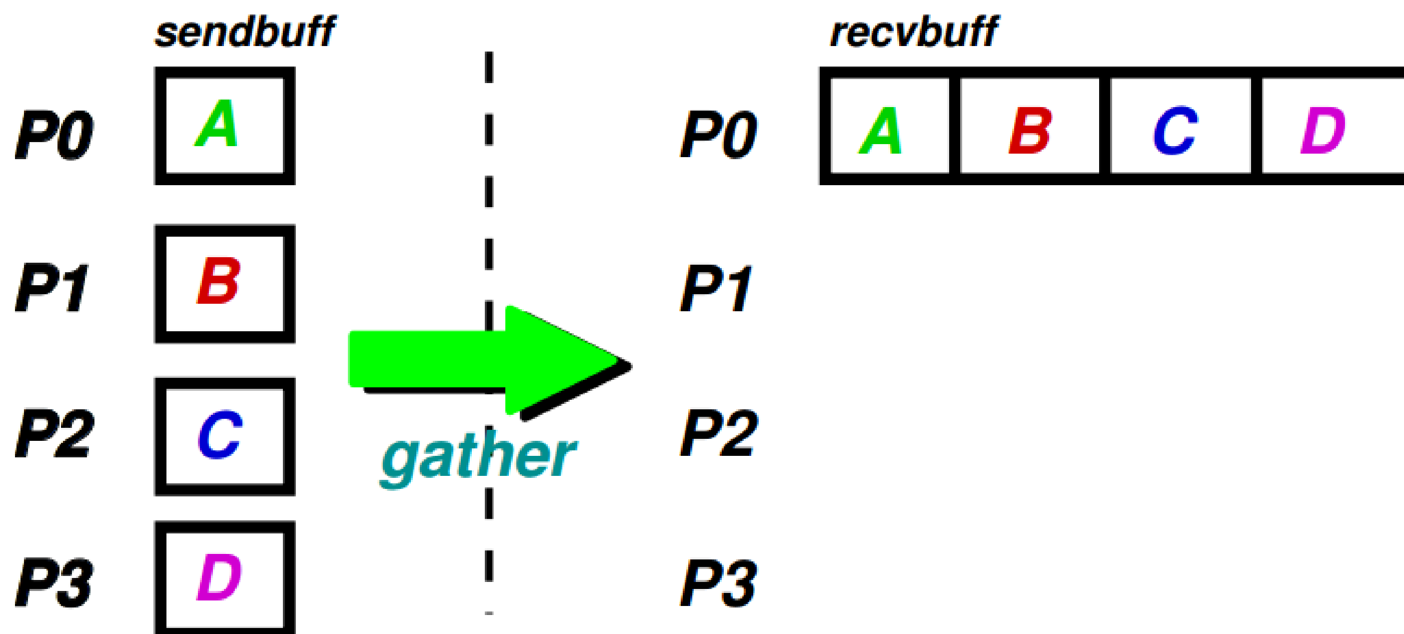
Scatter operations (3)



Gather operations (1)

Is the inverse to scatter, (*gathers*) a certain length of data from each processor in a destination processor.

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



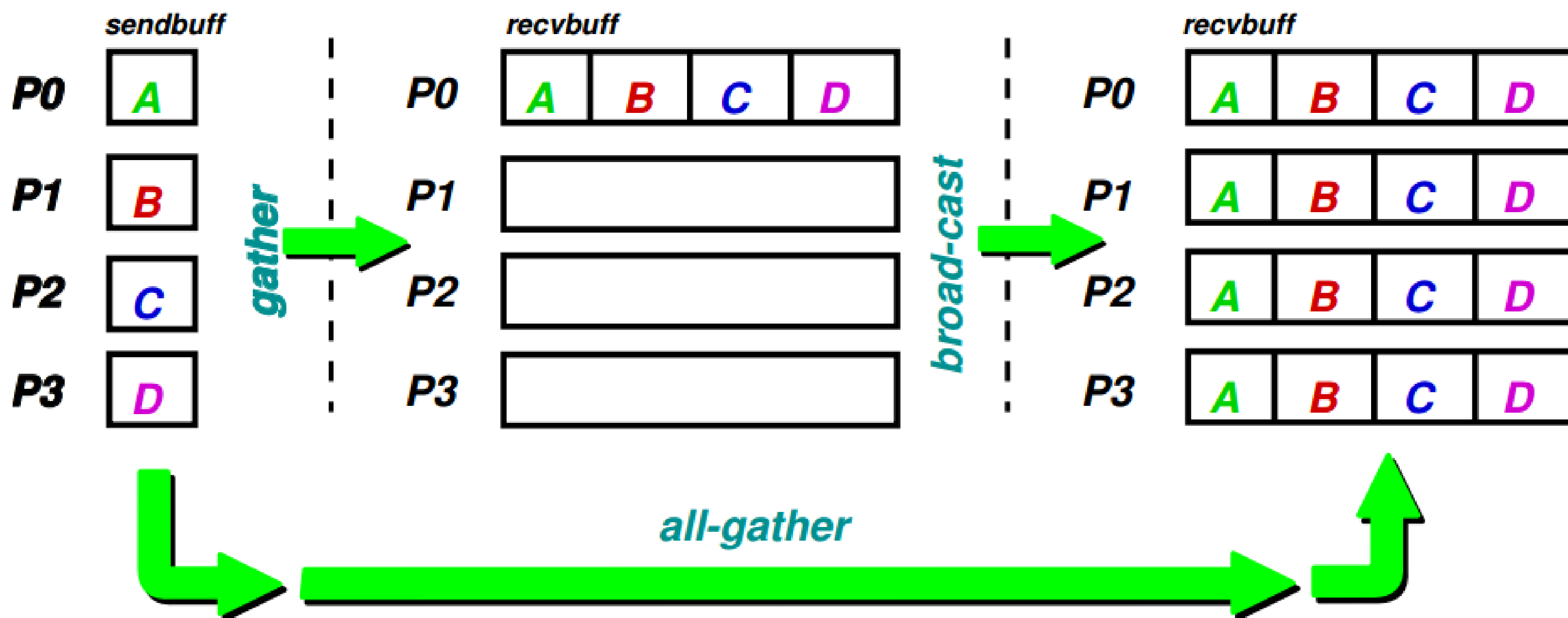
Gather operations (2)

```
1   #include <mpi.h>
2   #include <cstdio>
3   int main(int argc, char **argv) {
4       MPI_Init(&argc, &argv);
5       int myrank, size;
6       MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7       MPI_Comm_size(MPI_COMM_WORLD, &size);
8       int N = 5; // Nbr of elements to send to each processor
9       double *sbuff = new double[N]; // send buffer in all procs
10      for (int j = 0; j < N; j++) sbuff[j] = myrank * 1000.0 + j;
11      double *rbuff = NULL;
12       if (!myrank) {
13          rbuff = new double[N * size]; // recv buffer only in master
14       }
15       MPI_Gather(sbuff, N, MPI_DOUBLE, rbuff, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
16       if (!myrank)
17          for (int j = 0; j < N * size; j++)    printf("%d -> %f\n", j, rbuff[j]);
18       MPI_Finalize();
19       delete[ ] sbuff;
20       if (!myrank) delete[ ] rbuff;
21  }
```

All-gather operations (1)

It's conceptually equivalent to perform a gather followed by a broadcast.

int MPI_Allgather(*void* *sbuf, *int* scount, MPI_Datatype stype, *void* *rbuf, *int* rcount,
MPI_Datatype rtype, MPI_Comm comm)



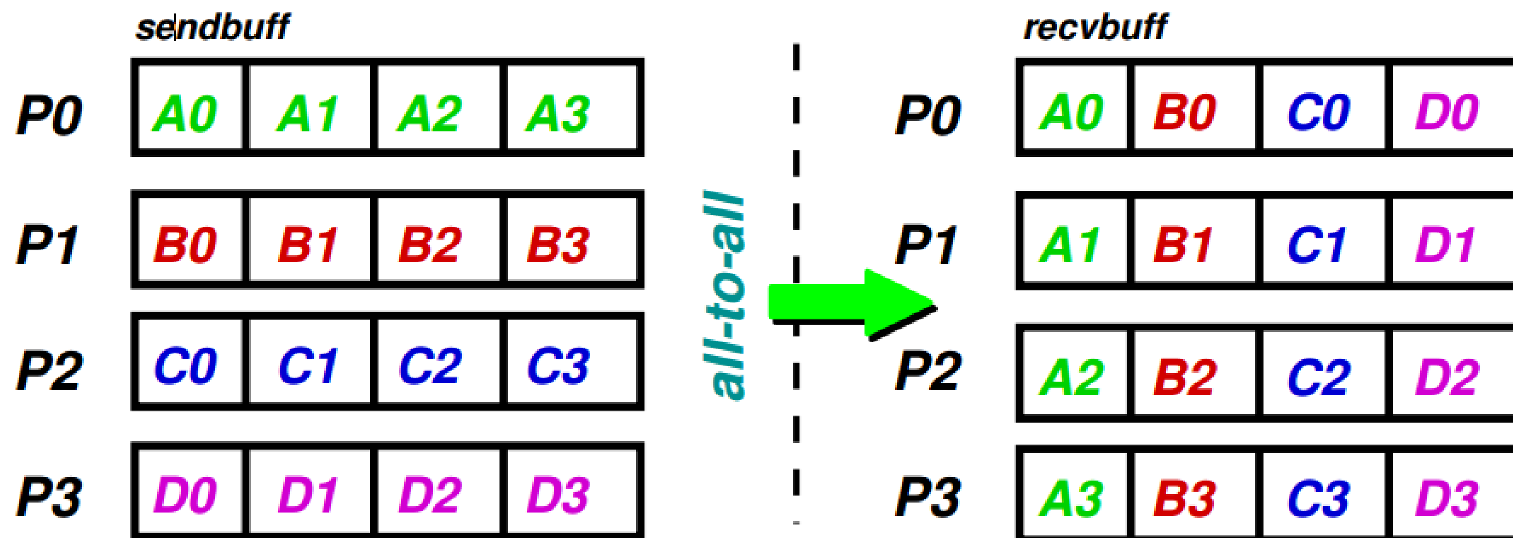
All-gather operations (2)

```
1  #include <mpi.h>
2  #include <cstdio>
3  int main(int argc, char **argv) {
4     MPI_Init(&argc, &argv);
5     int myrank, size;
6     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7     MPI_Comm_size(MPI_COMM_WORLD, &size);
8     int N = 5; // Nbr of elements to send to each processor
9     double *sbuff = new double[N]; // send buffer in all procs
10    for (int j = 0; j < N; j++) sbuff[j] = myrank * 1000.0 + j;
11    double *rbuff = new double[N * size]; // receive buffer in all procs
12    MPI_Allgather(sbuff, N, MPI_DOUBLE, rbuff, N, MPI_DOUBLE, MPI_COMM_WORLD);
13    for (int j = 0; j < N * size; j++)  printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
14     MPI_Finalize();
15     delete[ ] sbuff;
16     delete[ ] rbuff;
17 }
```

All-to-all operations (1)

Its conceptually equivalent to a scatter from P_0 followed by a scatter from P_1, etc..., or either a gather to P_0, followed by a gather to P_1, and so on...

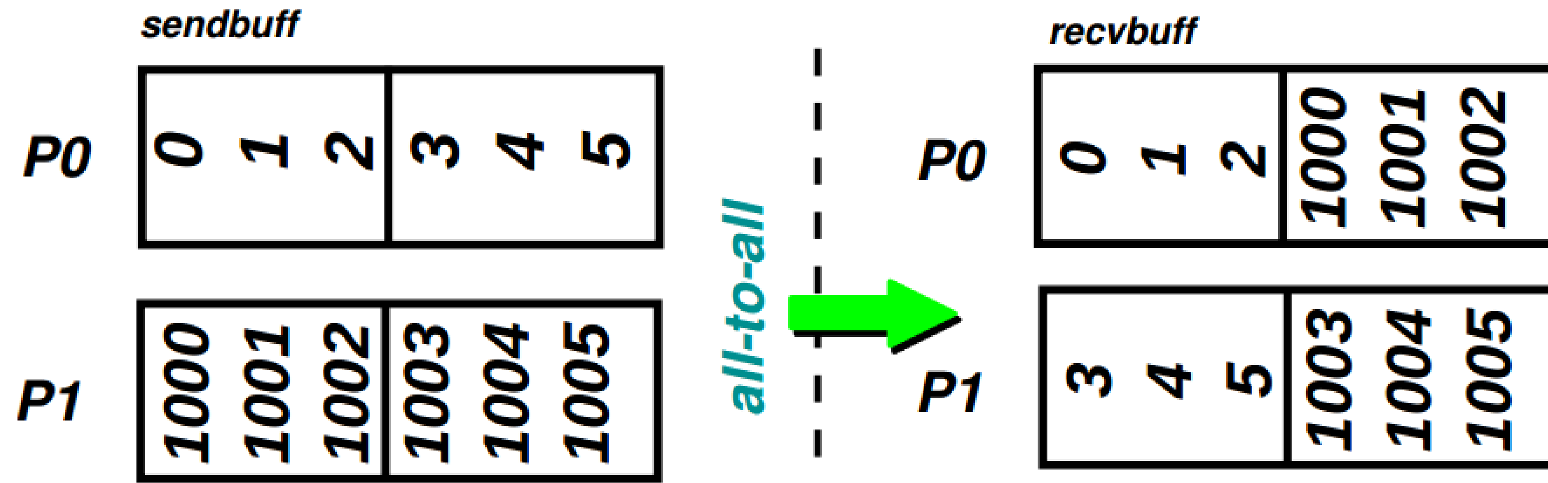
int MPI_Alltoall(*void* *sendbuf, *int* sendcount, MPI_Datatype sendtype, *void* *recvbuf, *int* recvcount, MPI_Datatype recvtype, MPI_Comm comm)



All-to-all operations (2)

```
1  #include <mpi.h>
2  #include <cstdio>
3  int main(int argc, char **argv) {
4      MPI_Init(&argc, &argv);
5      int myrank, size;
6      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7      MPI_Comm_size(MPI_COMM_WORLD, &size);
8      int N = 3; // Nbr of elements to send to each processor
9      double *sbuff = new double[N]; // send buffer in all procs
10     for (int j = 0; j < N; j++) sbuff[j] = myrank * 1000.0 + j;
11     double *rbuff = new double[N * size]; // receive buffer in all procs
12     MPI_Alltoall(sbuff, N, MPI_DOUBLE, rbuff, N, MPI_DOUBLE, MPI_COMM_WORLD);
13     for (int j = 0; j < N * size; j++)  printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
14      MPI_Finalize();
15      delete[ ] sbuff;
16      delete[ ] rbuff;
17 }
```

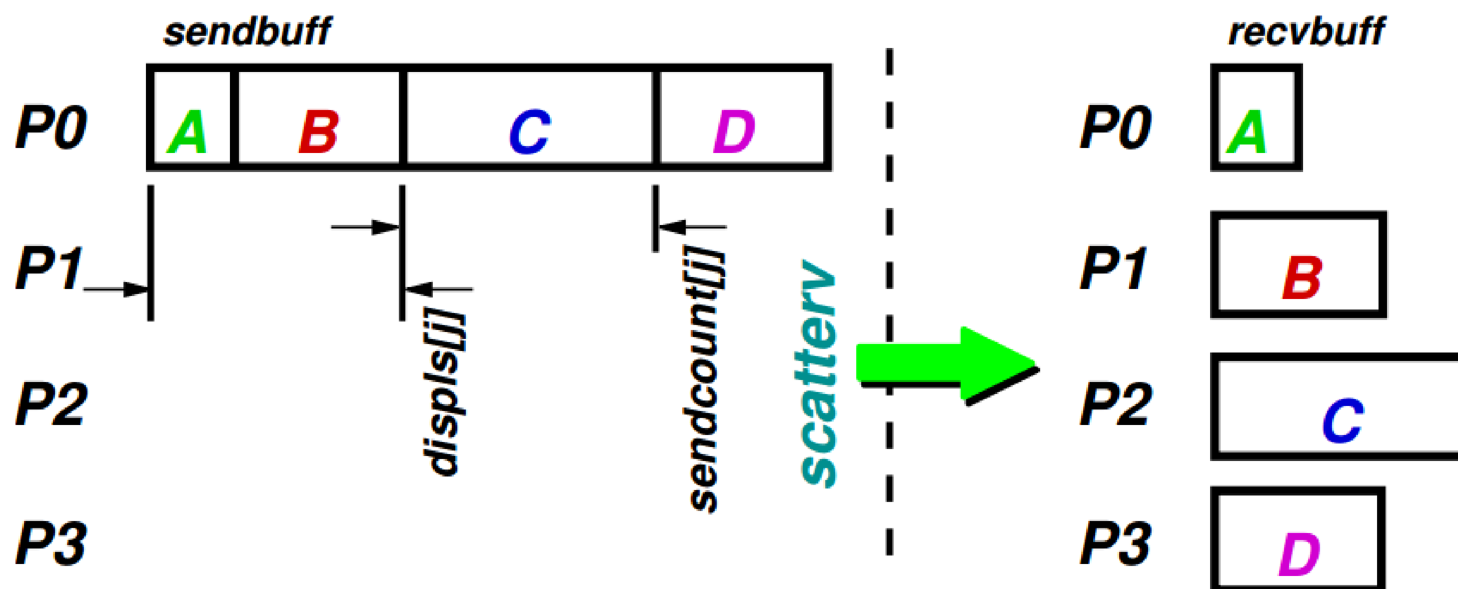
All-to-all operations (3)



Vector scatter (variable length) (1)

It is conceptually equivalente to a *MPI_Scatter()* but allows that the length of data send to each processor may be different.

int MPI_Scatterv(*void* *sendbuf, *int* *sendcnts, *int* *displs, MPI_Datatype sendtype,
void *recvbuf, *int* recvcnt, MPI_Datatype recvtype, *int* root,
MPI_Comm comm);



Vector scatter (variable length) (2)

```
1   int N = size * (size + 1) / 2;
2   double *sbuff = NULL;
3   int *sendcnts = NULL;
4   int *displs = NULL;
5   if (!myrank) {
6       sbuff = new double[N]; // send buffer only in master
7       for (int j = 0; j < N; j++) sbuff[j] = j; // fills 'sbuff'
8       sendcnts = new int[size];
9       displs = new int[size];
10      for (int j = 0; j < size; j++) sendcnts[j] = (j + 1);
11      displs[0] = 0;
12      for (int j = 1; j < size; j++)  displs[j] = displs[j - 1] + sendcnts[j - 1];
13  }
14  double *rbuff = new double[myrank + 1]; // receive buffer in all procs
15  MPI_Scatterv(sbuff, sendcnts, displs, MPI_DOUBLE, rbuff, myrank + 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
16  for (int j = 0; j < myrank + 1;  j++)
17      printf("[%d] %d -> %f\n", myrank, j, rbuff[j]);
```

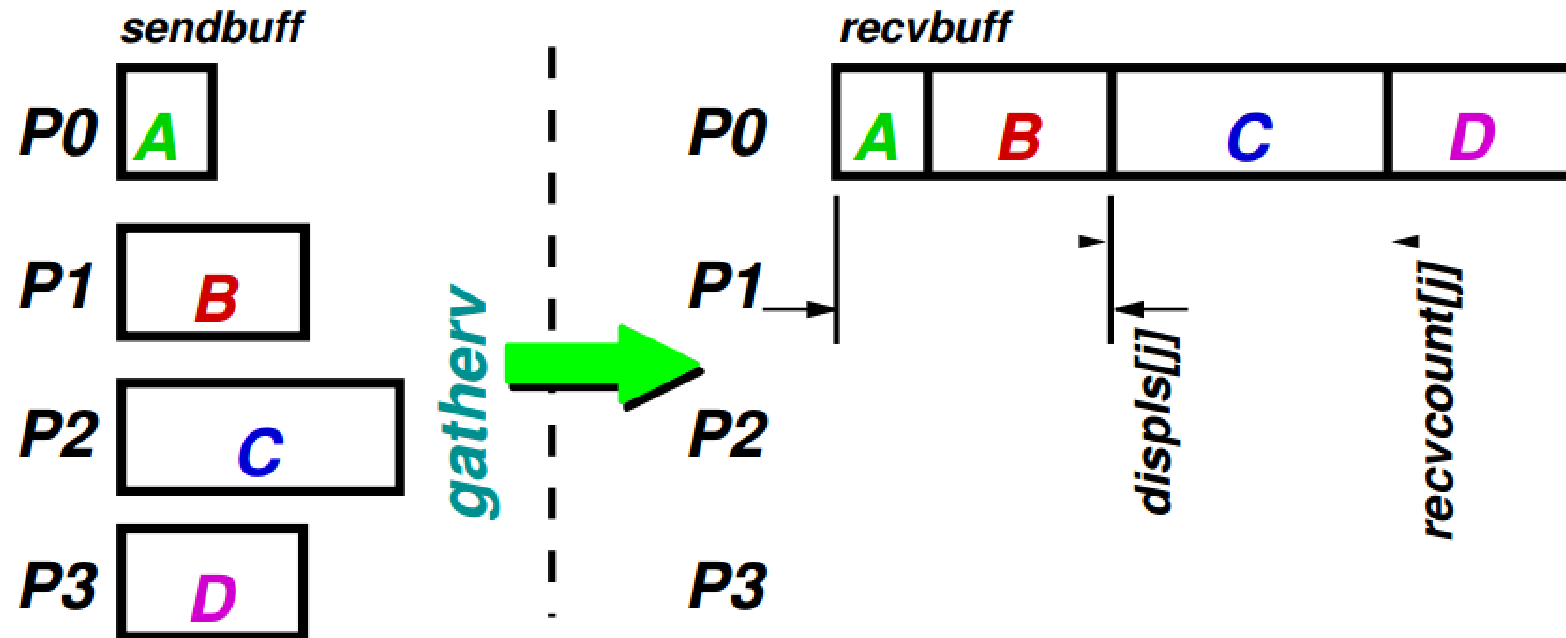
Gatherv operation (1)

Is the same as *gather*, but each processor receives data of different length.

***int* MPI_Gatherv(*void* *sendbuf, *int* sendcnt, MPI_Datatype sendtype, *void* *recvbuf,**

***int* *recvcnts, *int* *displs, MPI_Datatype recvtype, *int* root,**

MPI_Comm comm);



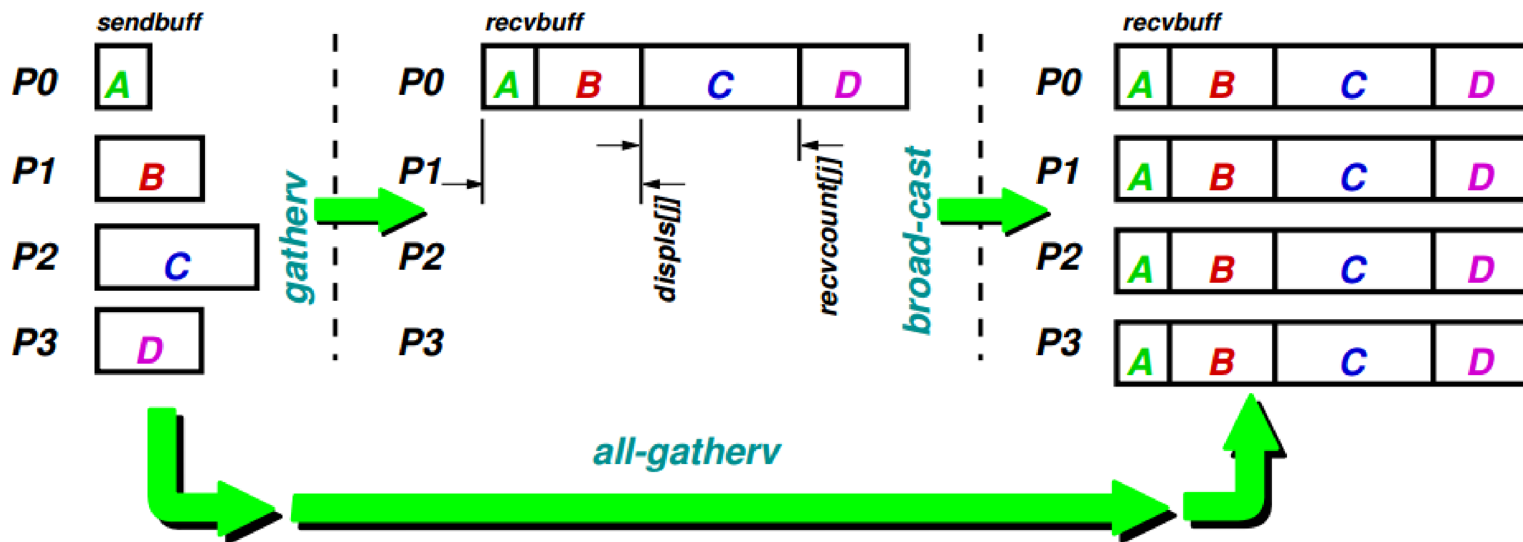
Gatherv operation (2)

```
1   int sendcnt = myrank + 1;   // send buffer in all
2   double *sbuff = new double[myrank + 1] ;
3    for (int j = 0; j < sendcnt; j++)
4        sbuff[j] = myrank * 1000 + j;
5    int rsize = size * (size + 1) / 2;
6    int *recvcnts = NULL;
7    int *displs = NULL;
8    double *rbuff = NULL;
9    if (!myrank) {
10     // receive buffer and ptrs only in master
11     rbuff = new double[rsize];  // recv buffer only in master
12     recvcnts = new int[size];
13     displs = new int[size];
14      for (int j = 0; j < size; j++) recvcnts[j] = (j + 1);
15      displs[0] = 0;
16      for (int j = 1; j < size; j++)  displs[j] = displs[j - 1] + recvcnts[j - 1];
17   }
18   MPI_Gatherv(sbuff, sendcnt, MPI_DOUBLE, rbuff, recvcnts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Allgatherv operation (1)

Is the same as gatherv, followed by a broadcast.

***int* MPI_Allgatherv(*void* *sbuf, *int* scount, MPI_Datatype stype, *void* *rbuf, *int* *rcounts, *int* *displs, MPI_Datatype rtype, MPI_Comm comm)**



Allgatherv operation (2)

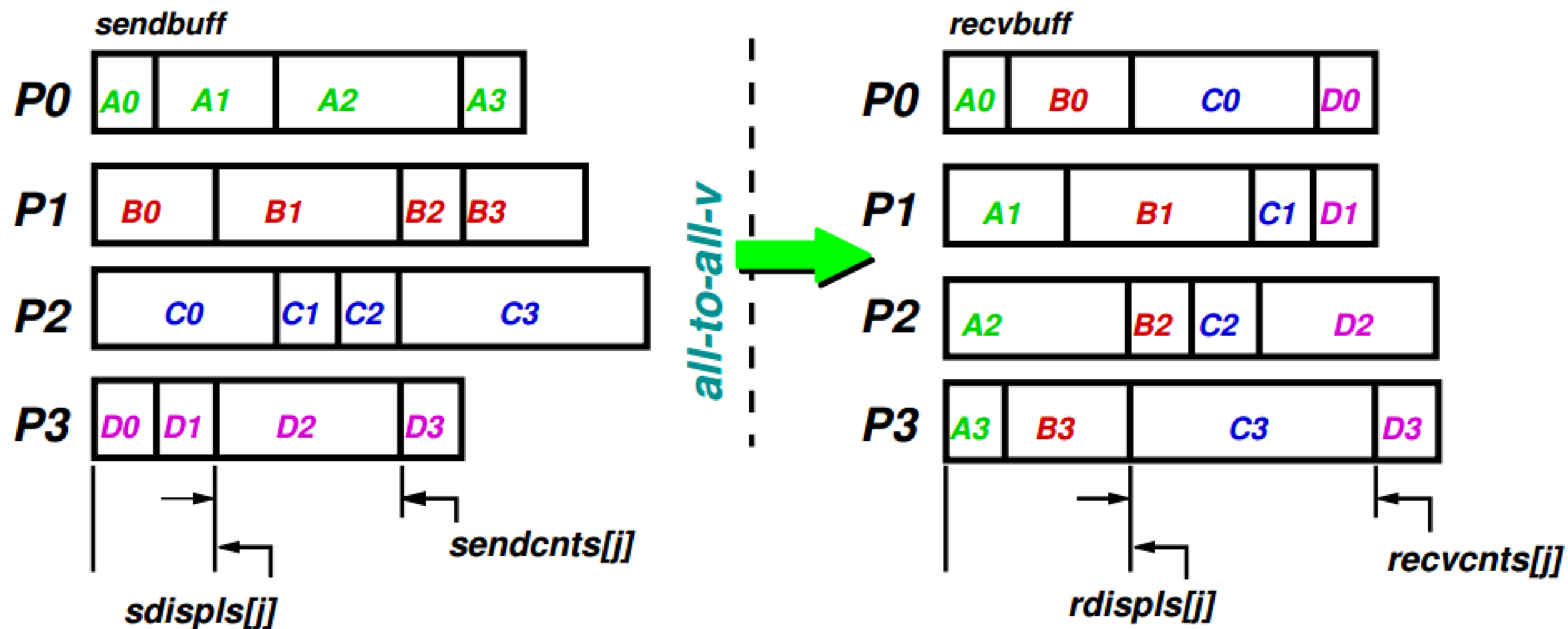
```
1   int sendcnt = myrank + 1;   // send buffer in all
2   double *sbuff = new double[myrank + 1];
3   for (int j = 0; j < sendcnt; j++)
4       sbuff[j] = myrank * 1000 + j;
5   // receive buffer and ptrs in all
6   int rsize = size * (size + 1) / 2;
7   double *rbuff = new double[rsize];
8   int *recvcnts = new int[size];
9   int *displs = new int[size];
10  for (int j = 0; j < size;  j++) recvcnts[j] = (j + 1);
11  displs[0] = 0;
12  for (int j = 1; j < size; j++)
13      displs[j] = displs[j - 1] + recvcnts[j - 1];
14  MPI_Allgatherv(sbuff, sendcnt, MPI_DOUBLE, rbuff, recvcnts, displs, MPI_DOUBLE, MPI _COMM_WORLD);
```

All-to-all-v operation (1)

Vectorized version (variable length data) of *MPI_Alltoall()*.

int MPI_Alltoallv(*void* *sbuf, *int* *scnts, *int* *sdispls, MPI_Datatype stype, *void* *rbuf,

int *rcnts, *int* *rdispls, MPI_Datatype rtype, MPI_Comm comm);



All-to-all-v operation (2)

```
1   int ssize = (myrank + 1) * size; // vectorized send buffer in all
2    double *sbuff = new double[ssize];
3    int *sendcnts = new int[size];
4    int *sdispls = new int[size];
5    for (int j = 0; j < ssize; j++)  sbuff[j] = myrank * 1000 + j;
6    for (int j = 0; j < size; j++) sendcnts[j] = (myrank + 1);
7    sdispls[0] = 0;
8        for (int j = 1; j < size; j++)    sdispls[j] = sdispls[j - 1] + sendcnts[j - 1];
9    int rsize = size * (size + 1) / 2; // vectorized receive buffer and ptrs in all
10    double *rbuff = new double[rsize];
11     int *recvcnts = new int[size];
12     int *rdispls = new int[size];
13     for (int j = 0; j < size; j++) recvcnts[j] = (j + 1);
14         rdispls[0] = 0;
15     for (int j = 1; j < size; j++)    rdispls[j] = rdispls[j - 1] + recvcnts[j - 1];
16     MPI_Alltoallv(sbuff, sendcnts, sdispls, MPI_DOUBLE, rbuff, recvcnts, rdispls, MPI_DOUBLE,
                          MPI_COMM_WORLD);
```

Exercise

1. Try to run the cpi code with different number processors and mesh points to see the change of the accuracy and total compute time.
2. Based on the cpi code, try to do the integration $\int_0^{\pi/2} \sin(x)\,\mathrm{d}x$.
3. Based on the matrix production code, write the parallel code of the Jacobi iterative method for solving a linear system.

Linear System

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \qquad \vdots \qquad\qquad \vdots \quad &\ \ \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

Jacobi Iteration

$$
\begin{aligned}
x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n) \\
x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n) \\
&\vdots \\
x_n &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1})
\end{aligned}
$$

Iteration matrices

Jacobi, Gauss-Seidel, SOR, & SSOR iterations are of the form

$$x^{(k+1)} = Mx^{(k)} + f$$

- $M_{Jac} = D^{-1}(E+F) = I - D^{-1}A$
- $M_{GS}(A) = (D-E)^{-1}F == I - (D-E)^{-1}A$
- $M_{SOR}(A) = (D-\omega E)^{-1}(\omega F + (1-\omega)D) = I - (\omega^{-1}D - E)^{-1}A$
- $M_{SSOR}(A) = I - (2\omega^{-1} - 1)(\omega^{-1}D - F)^{-1}D(\omega^{-1}D - E)^{-1}A$
  $= I - \omega(2\omega - 1)(D - \omega F)^{-1}D(D - \omega E)^{-1}A$

$$
\begin{bmatrix}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
& -1 & 4 & & & -1 & & & \\
-1 & & & 4 & -1 & & -1 & & \\
& -1 & & -1 & 4 & -1 & & -1 & \\
& & -1 & & -1 & 4 & & & -1 \\
& & & -1 & & & 4 & -1 & \\
& & & & -1 & & -1 & 4 & -1 \\
& & & & & -1 & & -1 & 4
\end{bmatrix}
$$

Thanks for your attention !

Back to Page 1