RIKEN International Summer School 2021

– Toward Society 5.0 –

Basics of Parallel Programming and Execution

Miwako Tsuji

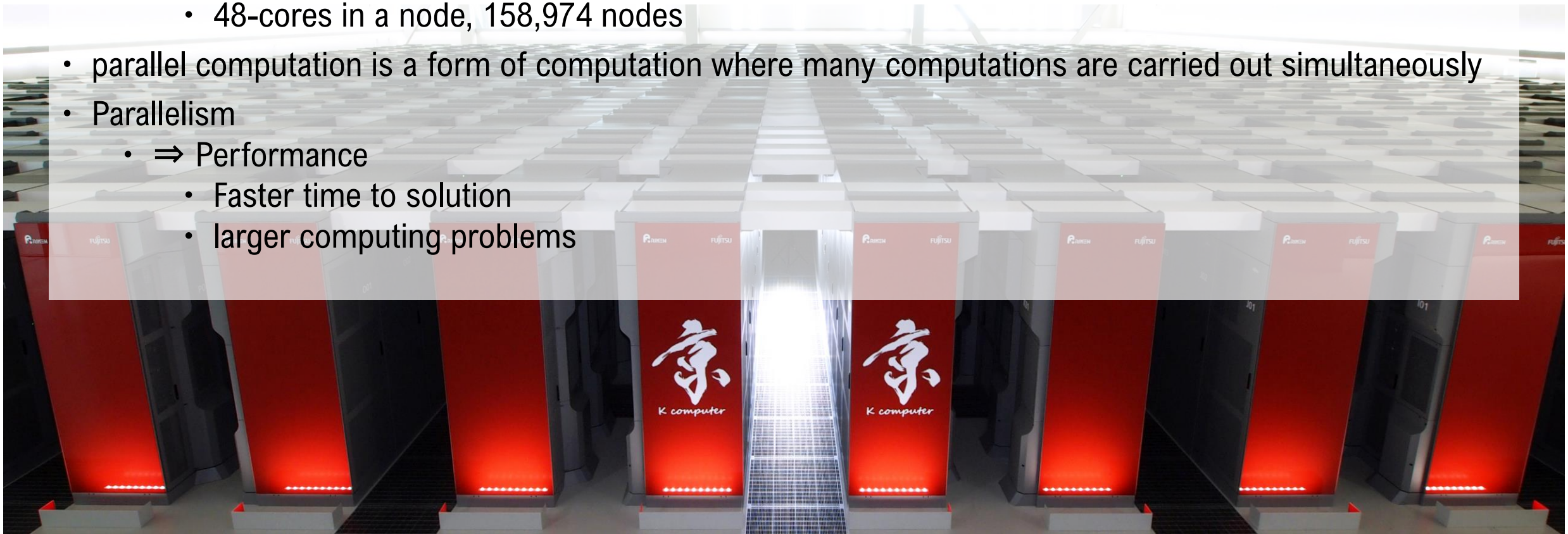
Programming Environment Research Team
RIKEN Center for computational Science

Agenda

1. Parallel computing and background
2. Parallel architectures
3. Some of important concepts to learn parallel programming
4. Setup your environment and login supercomputer Fugaku
5. Hands-on
   - OpenMP
   - MPI

Parallel Computing

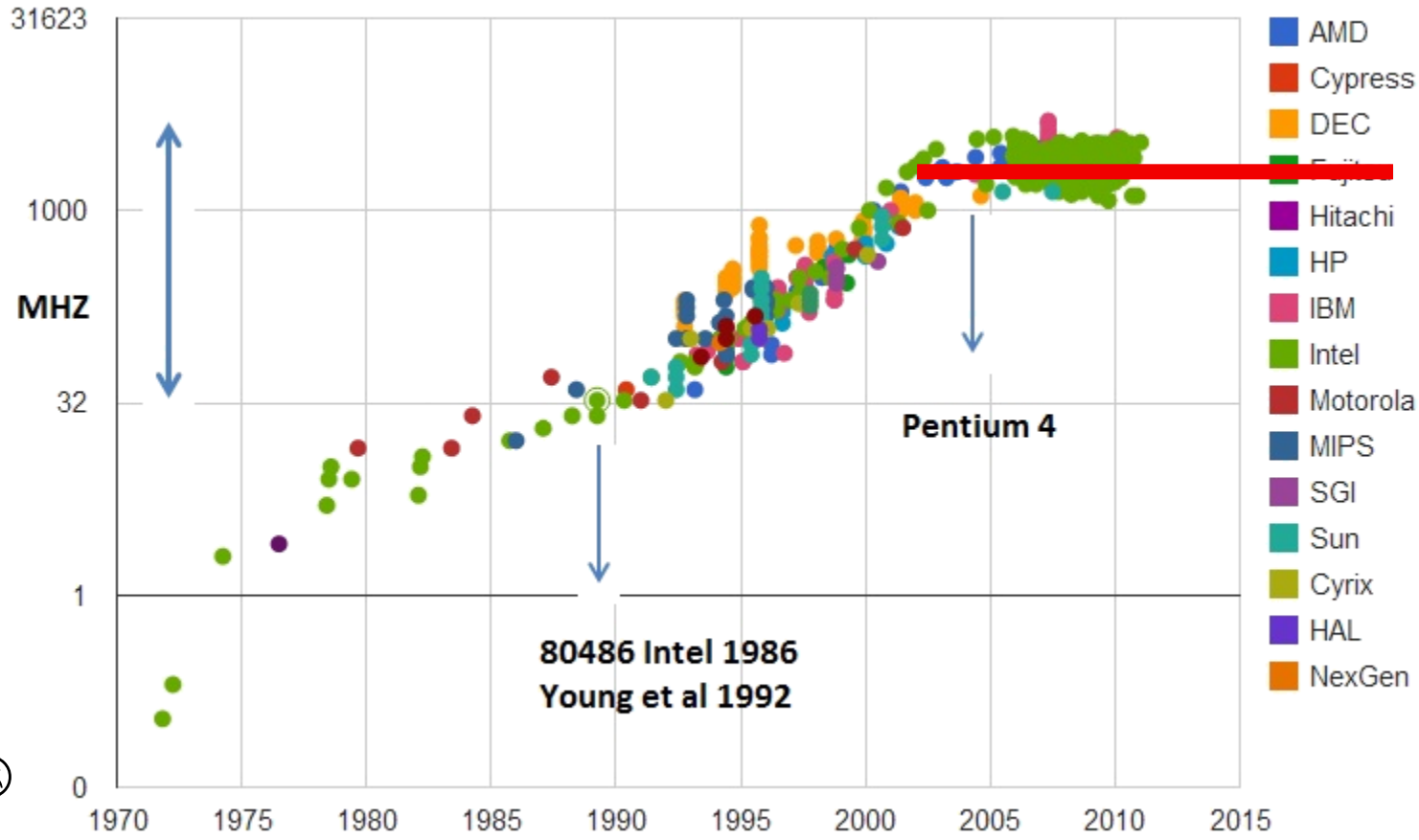
- parallel computer is a computer system that uses multiple processing elements (PEs) simultaneously
  - Apple A13 on IPhone 11 Pro/11 Pro Max
    - 2 Lighting cores, 4 Thunder cores
  - K-computer
    - 8-cores in a node, 88128 nodes
  - Supercomputer Fugaku
    - 48-cores in a node, 158,974 nodes
- parallel computation is a form of computation where many computations are carried out simultaneously
- Parallelism
  - ⇒ Performance
    - Faster time to solution
    - larger computing problems



Motivation of parallel computing

- Motivation: accelerate computation
  - Speedup based on frequency scaling ⇒ Limited due to the physical limits to transistor scaling
  - Energy consumption
    - approximately proportional to the CPU frequency, and to the square of the CPU voltage
  - ⇒ **Parallel computing**
    - consumption is proposal to the concurrency
- increase frequency
  - surely increase performance ☺
- increase concurrency
  - not always increase performance ☹
  - energy efficient

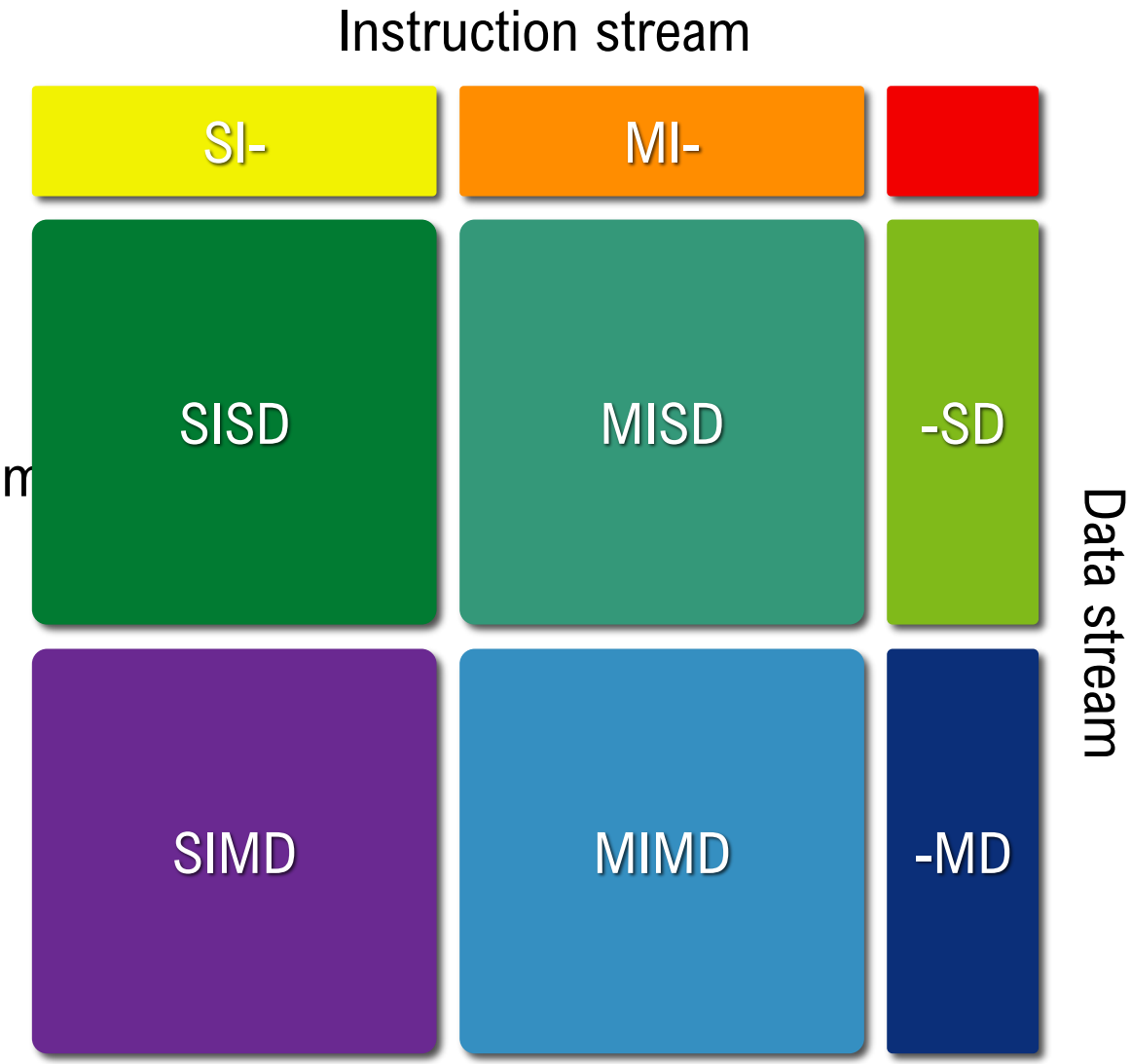
  ⇒ need to understand parallel computing



https://en.wikipedia.org/wiki/File:Clock_CPU_Scaling.jpg

Flynn's Taxonomy

- SISD Single Instruction stream, Single Data stream
  - No parallelism, entirely serial program
- SIMD Single Instruction stream, Multiple Data stream
  - the same operation over different data
- MISD Multiple Instruction stream, Single Data stream
  - (rarely used)
  - Multiple instructions operate on one data stream
- MIMD Multiple Instruction stream, Multiple Data stream
  - Multiple independent processors simultaneously executing different instructions on different data
- Modern HPC systems : hybrids of these categories

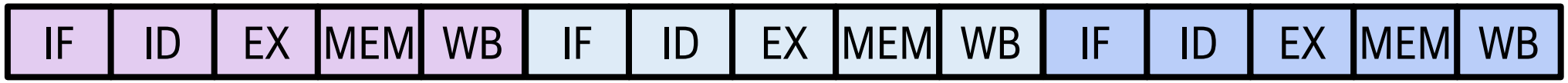


Parallel architectures supporting parallelism

- instruction-level parallelism
- SIMD
- distributed parallel system
- shared memory parallel system

Types of parallelism: instruction-level parallelism

- performing a number of instructions during a single clock cycle
- a program is a stream of instructions



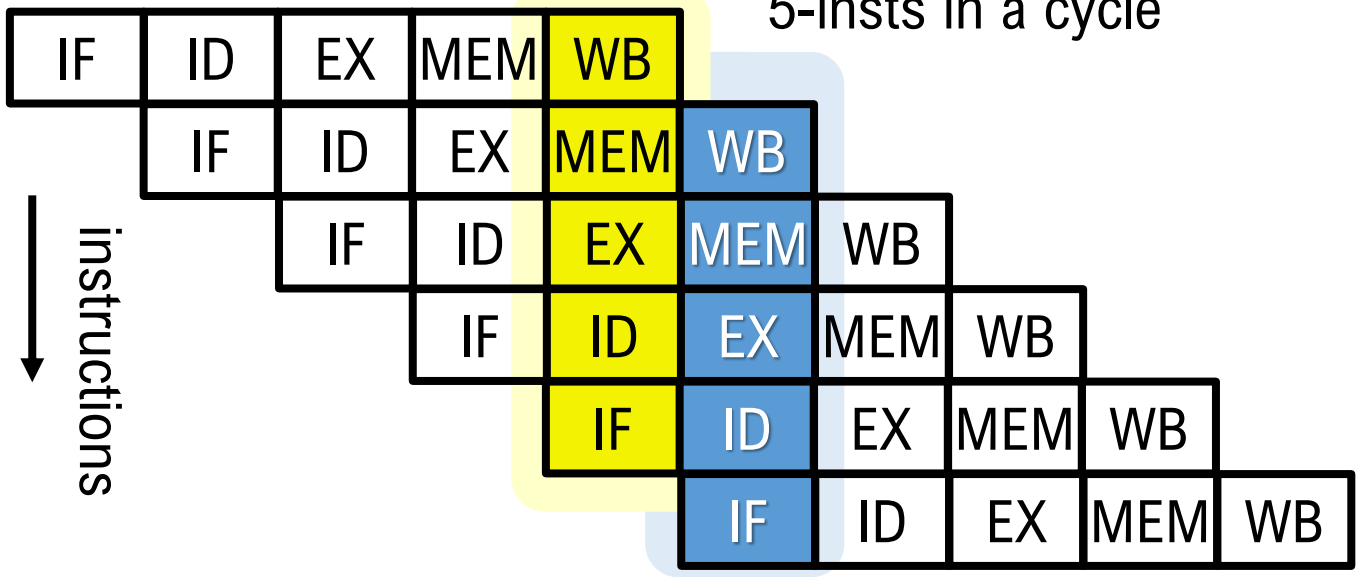
time

parallelization based on "**pipelining**"

time

5-insts in a cycle

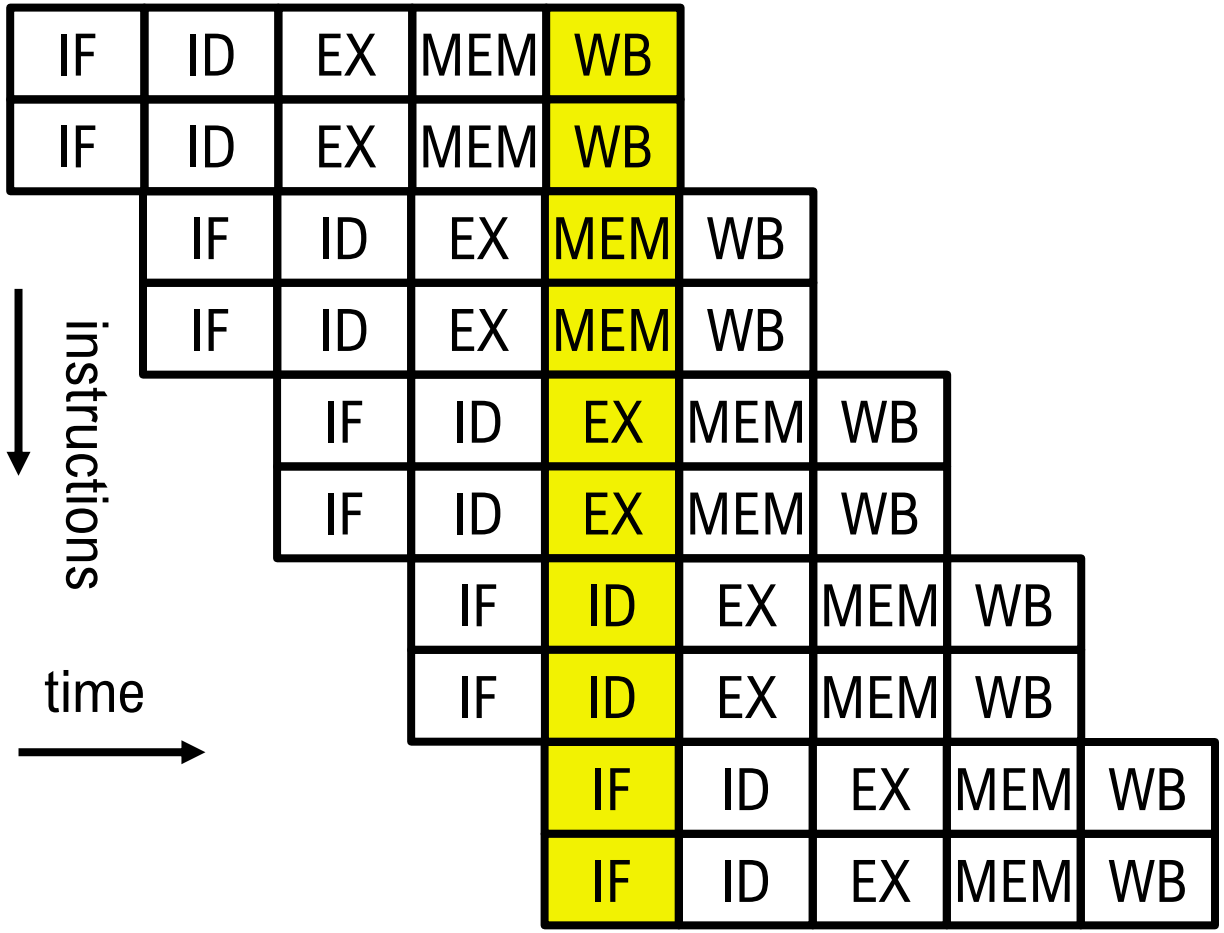
5-insts in a cycle



IF: Instruction Fetch
ID: Instruction Decode
EX: Execute
MEM: Memory access
WB: Register Write Back

Types of parallelism: instruction-level parallelism

- performing a number of instructions during a single clock cycle
- a program is a stream of instructions
- modern processors can issue more than one instruction at a cycle
  - ex: K computer
    - 4 instructions per cycle / core
- out-of-order execution
- instruction level parallelism



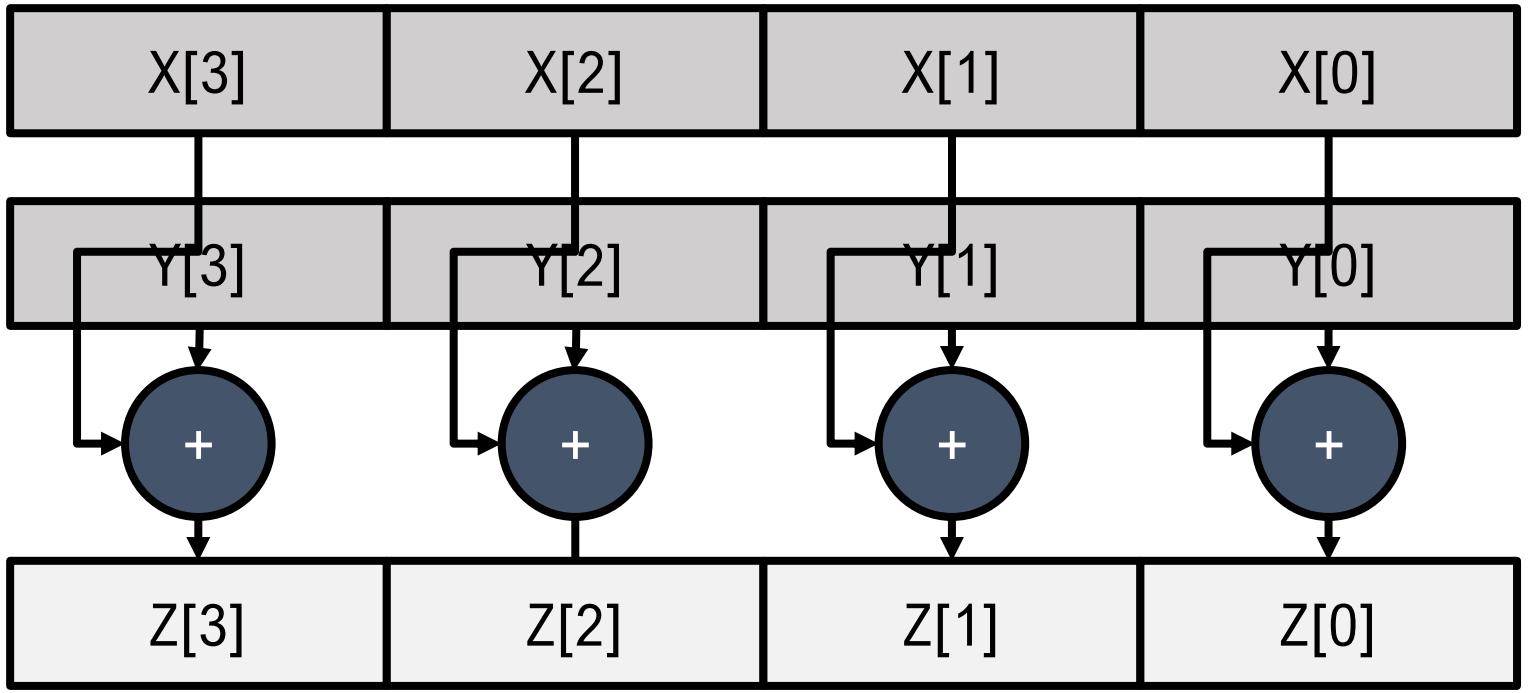
Types of parallelism: SIMD instruction

- an instruction for multiple data (data array) at a single cycle

  for(i=0; i<N; i++)

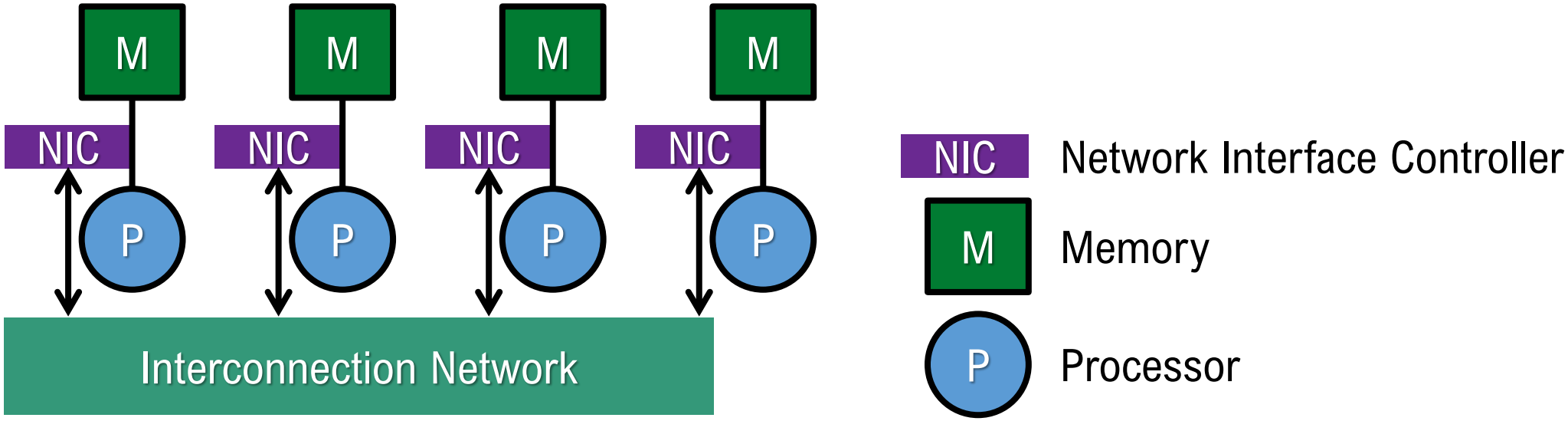
  z[i] = x[i]+y[i]

  - SPARC64 VIIIfx processor (K computer) : 128-bit SIMD (2 double precision can be processed in parallel)
  - SPARC64 XIfx processor (FX100 series) : 256-bit SIMD
  - Intel-AVX512 series: 512-bit SIMD



Distributed memory system

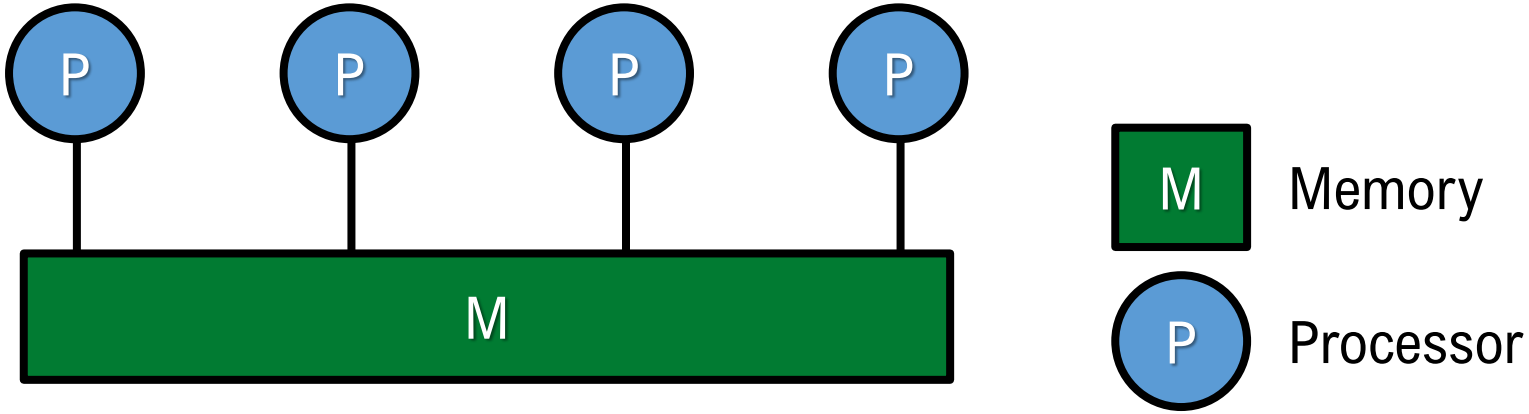
- each processor has its own local address space
- memory is logically or **physically** distributed
- systems, where compute nodes (w/ CPU and memory) are connected via network
- each program on each compute node exchanges data (messages) through network
- expandable
  - Massively Parallel Processor
  - Cluster



Shared memory system

- all processors can access a single address space
- each program (thread) on each compute node reads/writes a memory to exchange data

- Modern CPUs include multi-processor cores and a shared memory



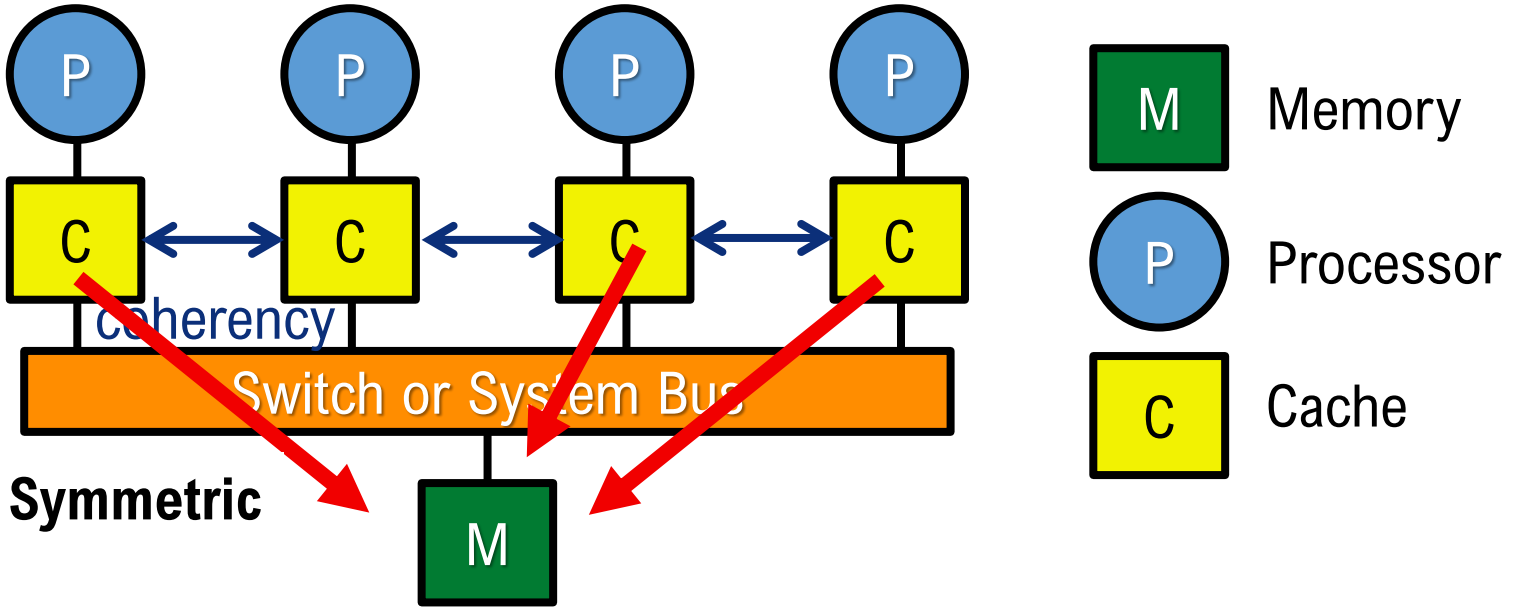
Shared memory system: SMP Symmetric Multi-Processor

- multi-processors are connected to a single, shared main memory
  - multi processors access a (set of) shared memory module(s) via network switch or bus
- all processors are treated equally
- traditional: without cache
- modern: with coherent caches, which keep the data in the caches consistent
- limits on the scalability of SMP, cache coherence and shared objects
- Performance degradation when traffic is concentrated

Fujitsu HPC2500 (2002)

Hitachi SR16000 (2011)

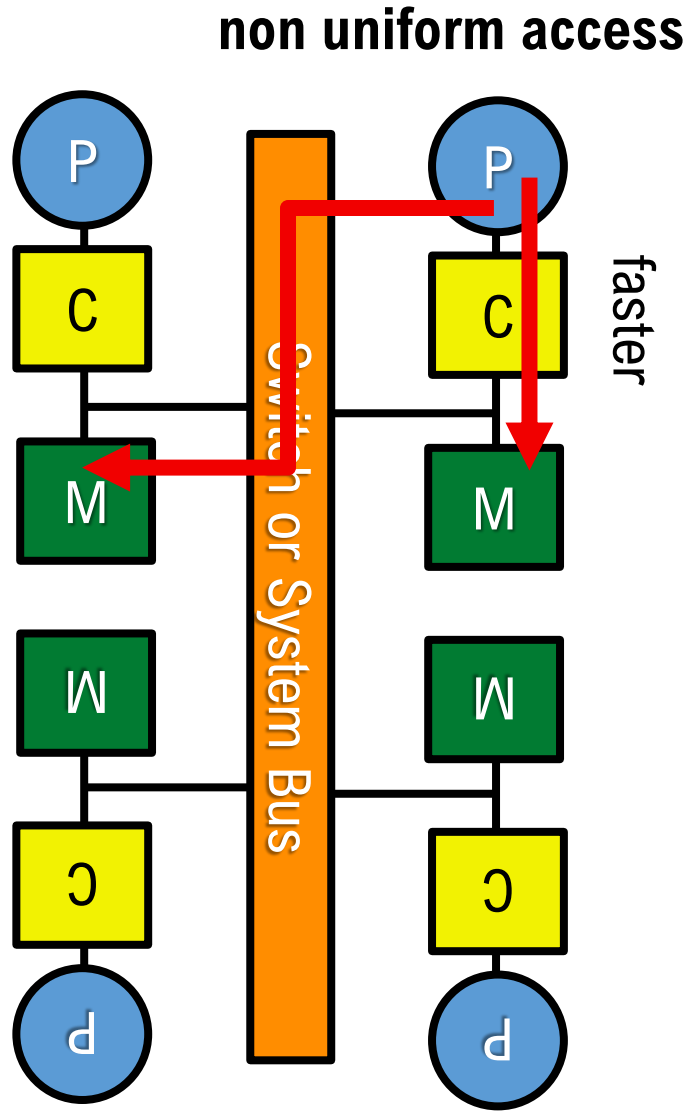
etc..



Shared memory system: NUMA Non-Uniform Memory Access

- a memory module (local memory) dedicated to each CPU
- a CPU can access a memory dedicated to a different CPU via shared bus or switch (remote memory)
- non-symmetric, where access to remote memories takes a longer time than access to local memory

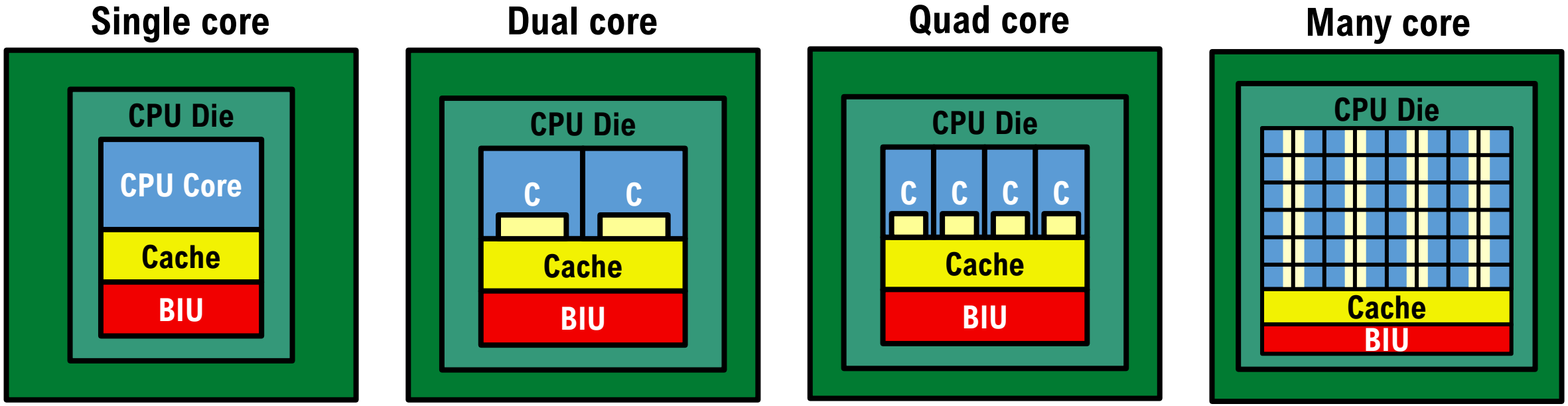
- AMD Opteron Barcelona (2007)



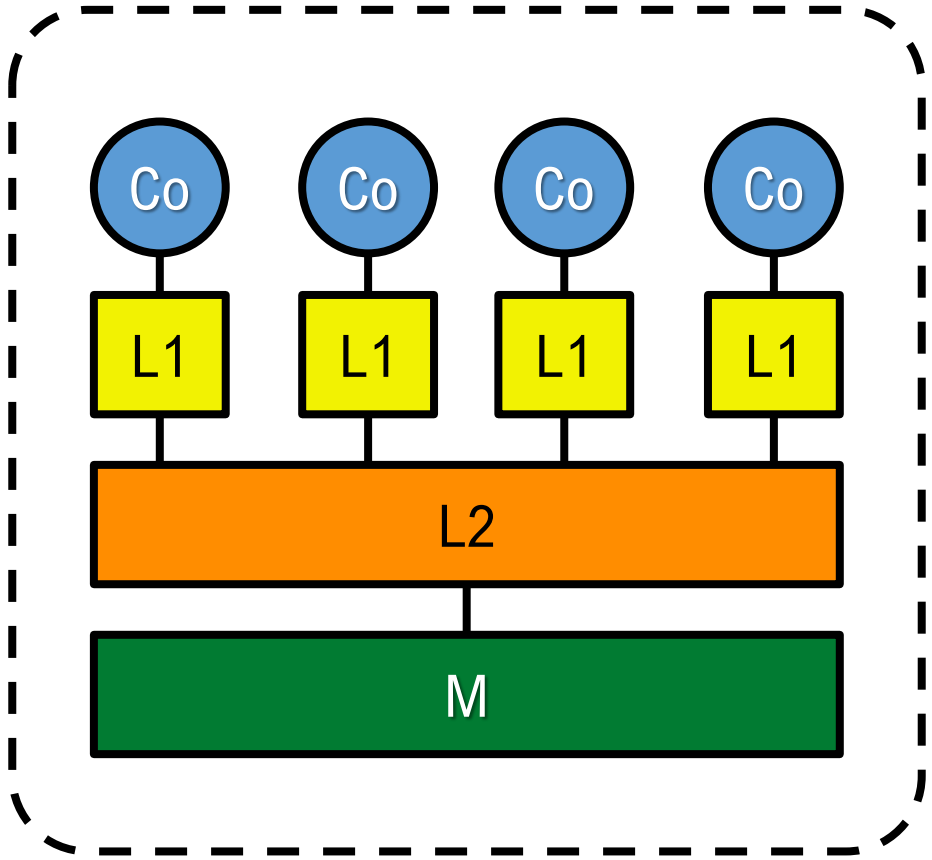
Multi-core processors

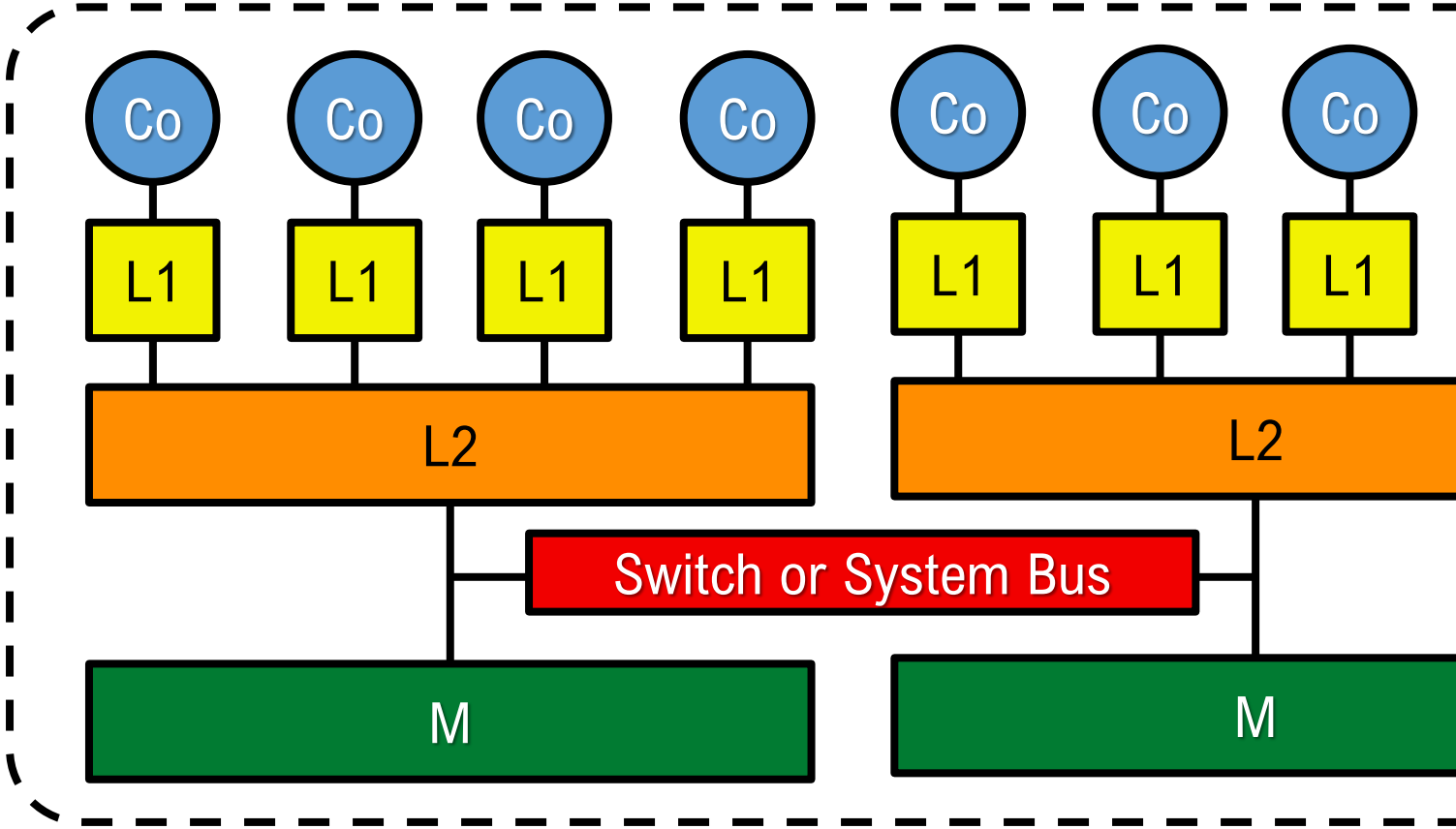
- core is a processing unit
- two or more cores in a computer processor : multi-core processor
  - ex. 8-cores in the SPARC64 VIIIfx processor (K-computer)
  - cores are independent
    - a processor can issue multiple (different/same) instructions from multiple cores
  - inter-core communication:
    - via message passing
    - **via shared-memory**

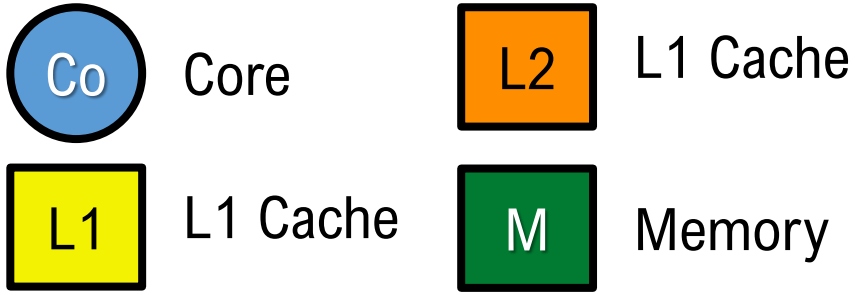
} can be both



Multi-core processors: SMP (SMC?) / NUMA

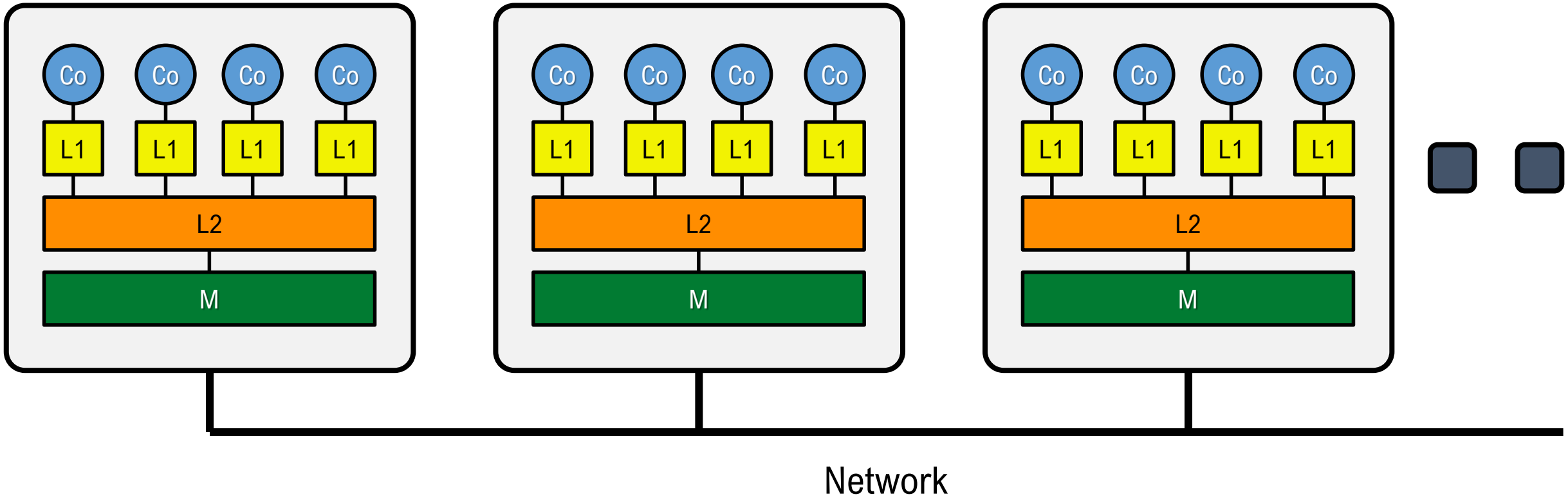






Hybrid of distributed parallel and shared memory systems

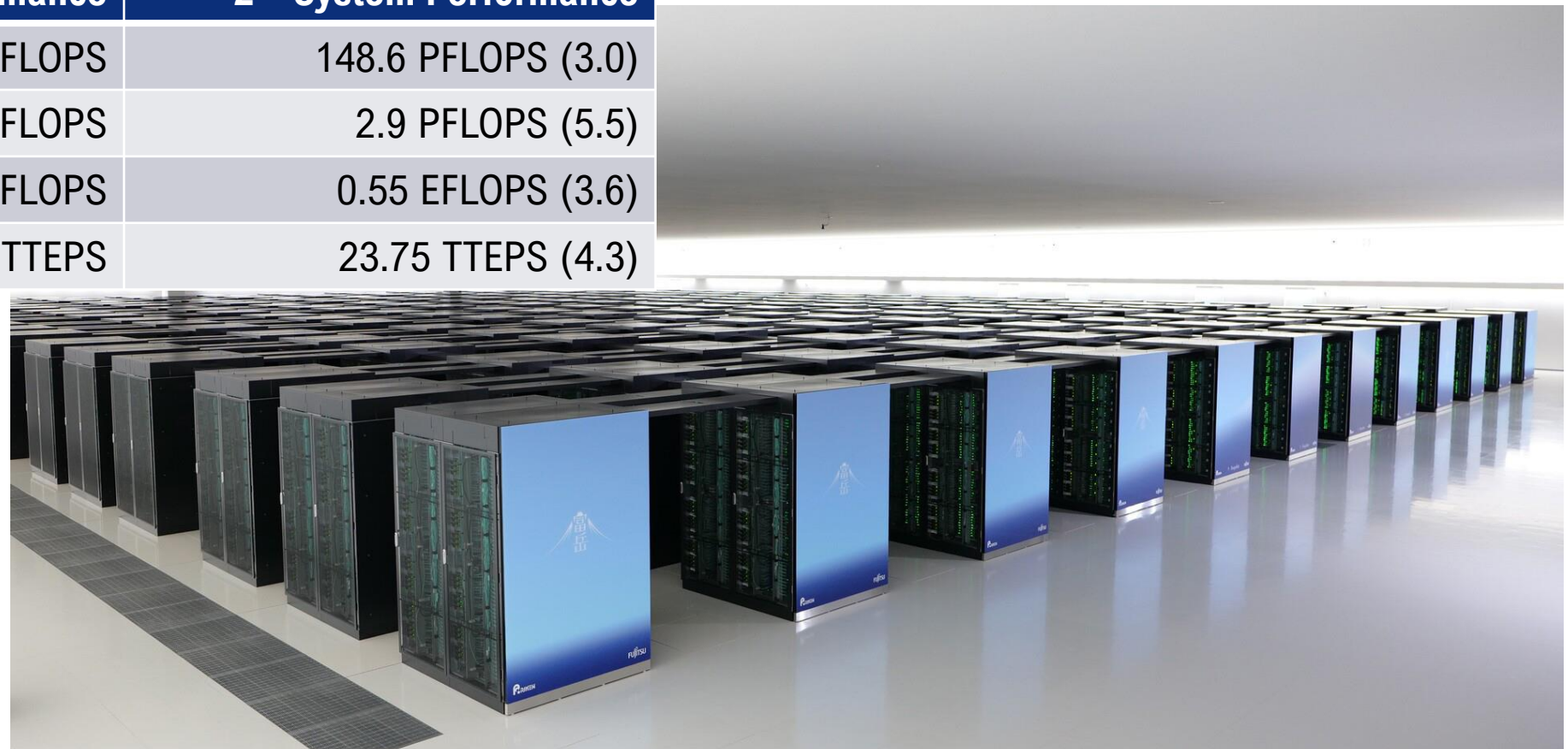
- several (80,000+ in case of K-computer) nodes of shared memory systems are connected as a distributed parallel system
- because of the popularity of the shared memory architecture in a single processor, i.e. a multi-core processor



An example of large-scale parallel systems: Overview of supercomputer Fugaku

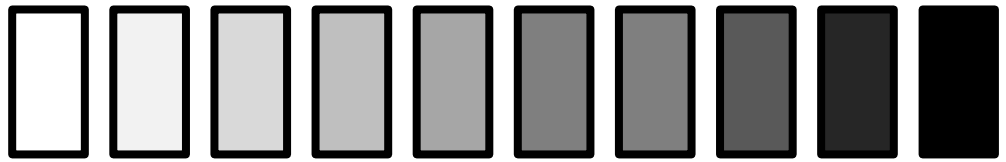
- A supercomputer installed at RIKEN Center for computational science
- Massively parallel system with 158,978 **general purpose CPUs**
- Ranked #1 in the 4 rankings announced in Nov. 2020

Ranking	Performance	2nd System Performance
TOP500	442.01 PFLOPS	148.6 PFLOPS (3.0)
HPGC	16.00 PFLOPS	2.9 PFLOPS (5.5)
HPL-AI	2.00 EFLOPS	0.55 EFLOPS (3.6)
Graph500	102.05 TTEPS	23.75 TTEPS (4.3)



Specification of supercomputer Fugaku (& K)

Specification of supercomputer Fugaku	
CPU Technology	
# of Cores in CPU	
CPU Clock	
Theoretical Peak	
Node Cache	
Memory Technology	
Memory Capacity	
Memory Bandwidth	1024 GB/s
Network	TofuD, 28 Gbps x 2 lane x 10 port
# of nodes	158,978 nodes



10 racks ≒ K computer

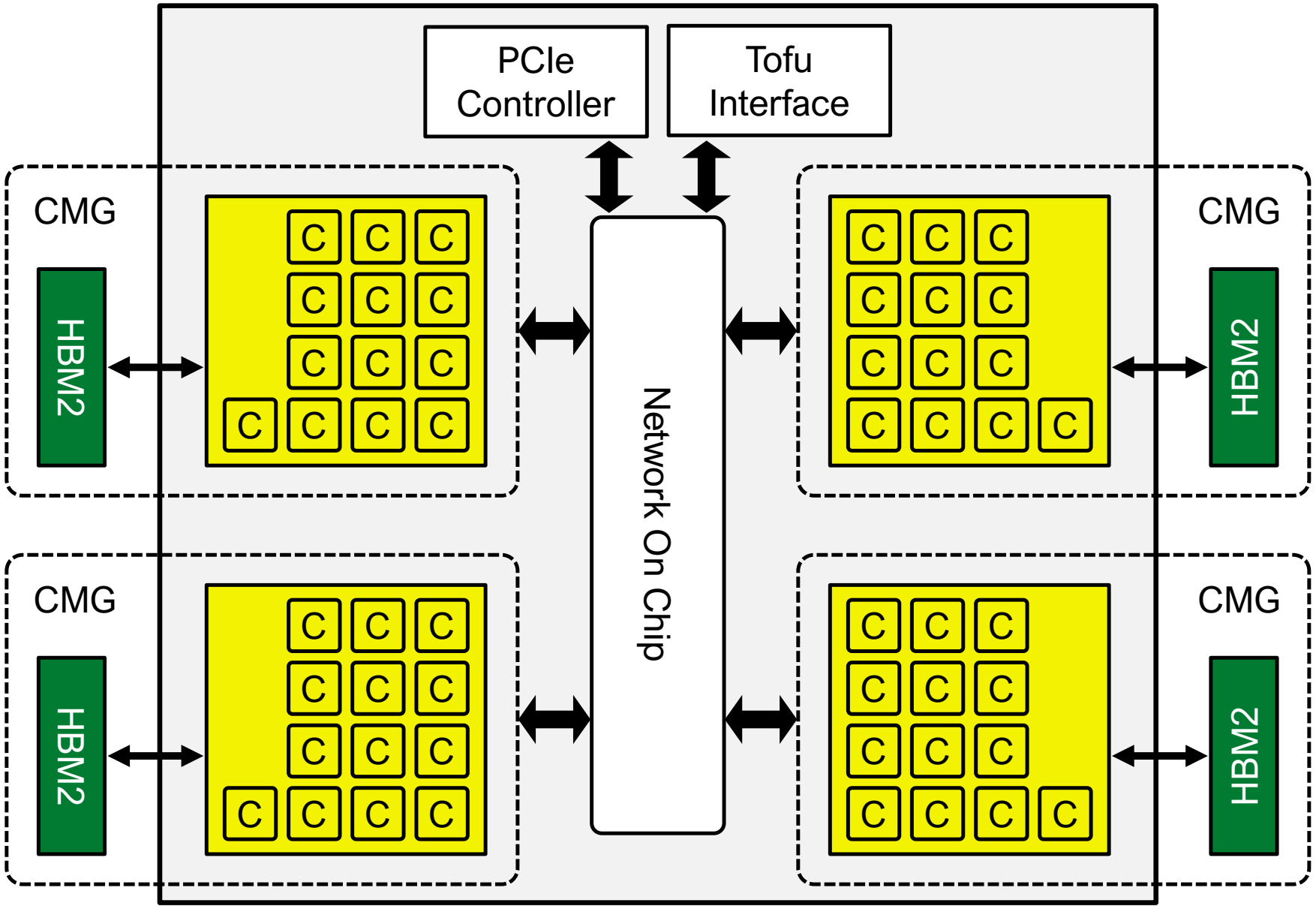
Specification of K computer	
ogy	SPARC64 VIIIfx
PU	8 cores
ock	2 GHz
eak	128Gflops
che	L1 32 KB / core
	L2 6 MB / node
ogy	DDR3
city	16 GB
Memory Bandwidth	64 GB/s
Network	Tofu 6-dimentional torus, 5GB/s (bi-direction)
# of nodes	82944 (88128, including IO-nodes)

K computer

A64FX: Node Overview

C: Core
CMG: Core Memory Group
HBM: High Bandwidth Memory

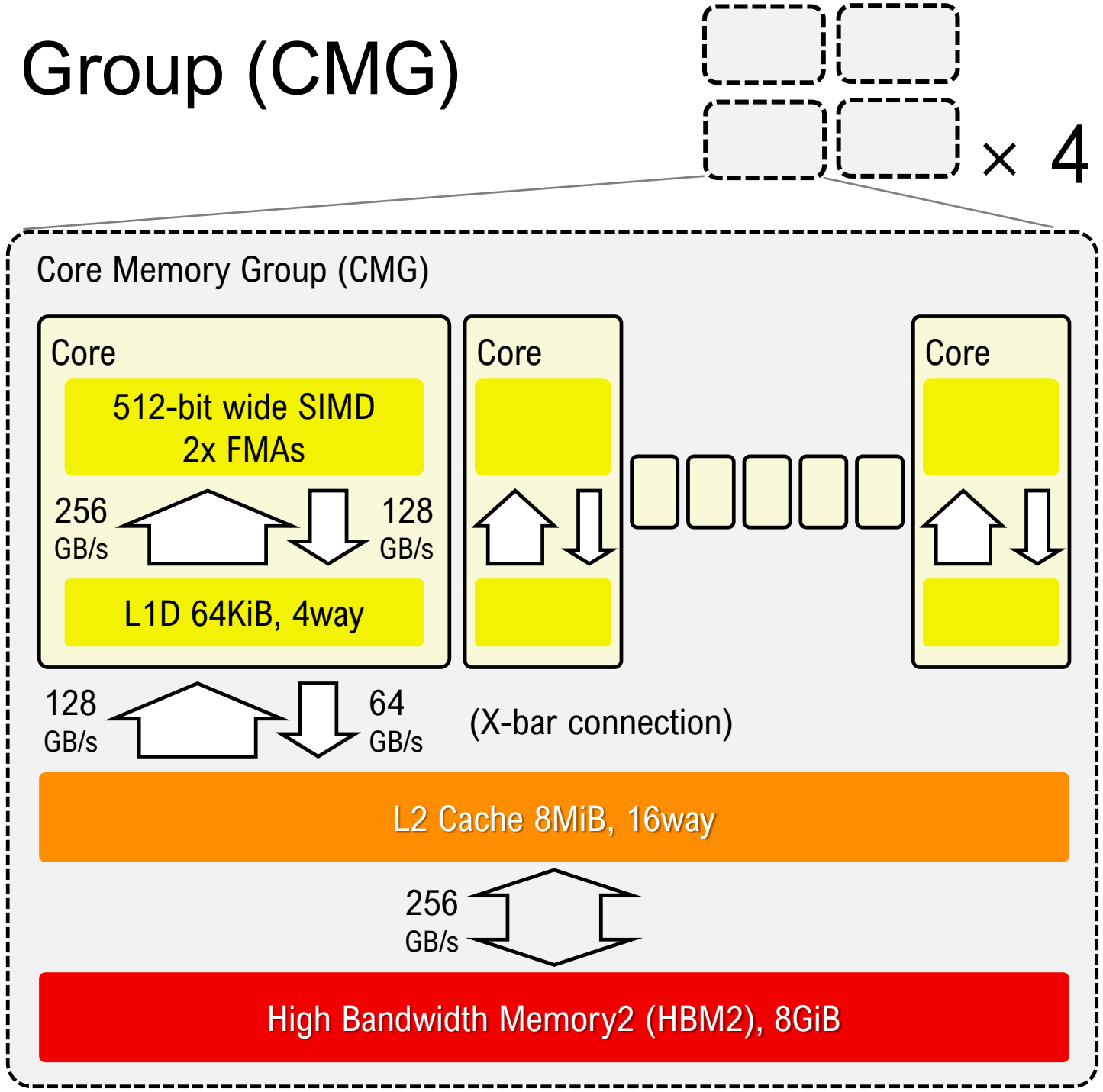
- SVE 512-bit wide SIMD
- FP64/FP32/FP16
- 12+1 cores in a CMG
  - an assistant core in each CMG
- 4 CMG in a Node
  - 48 + 2/4 core
- HBM2 32GiB/Node
  - 8GiB/CMG
  - 1024 GB/s
- Ring Network, 128 GB/s x 2
- TofuD NIC, 6.8 GB/s x 6
  - 6 simultaneously transmit directions



A64FX: Core Memory Group (CMG)

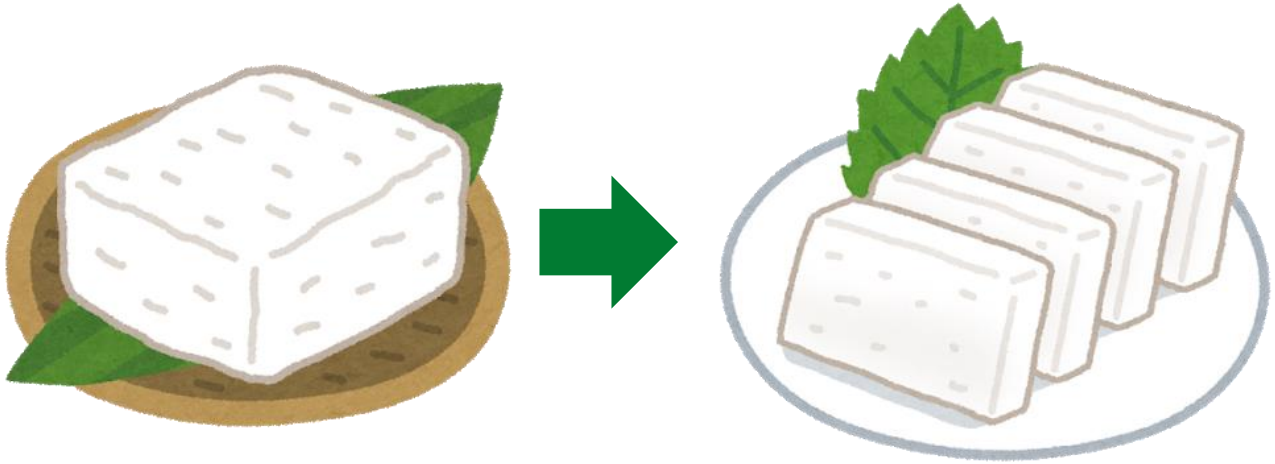
- 2x 512-bit wide SIMD, 4x ALUs, Predicate Operation
- 2x 512-bit wide SIMD load or 512-bit wide SIMD store
- L1D cache (/core) : 64 KiB, 4 ways, “Combined Gather” on L1
- L2D cache (/CMG): 8MiB
  - X-bar connection in a CMG
- 4 CMGs support cache coherency by ccNUMA with on-chip directory (256 GB/s x 2 for inter-CMGs)
- Memory controller for HBM2

※ L1/L2 cache bandwidths are for 2.0GHz

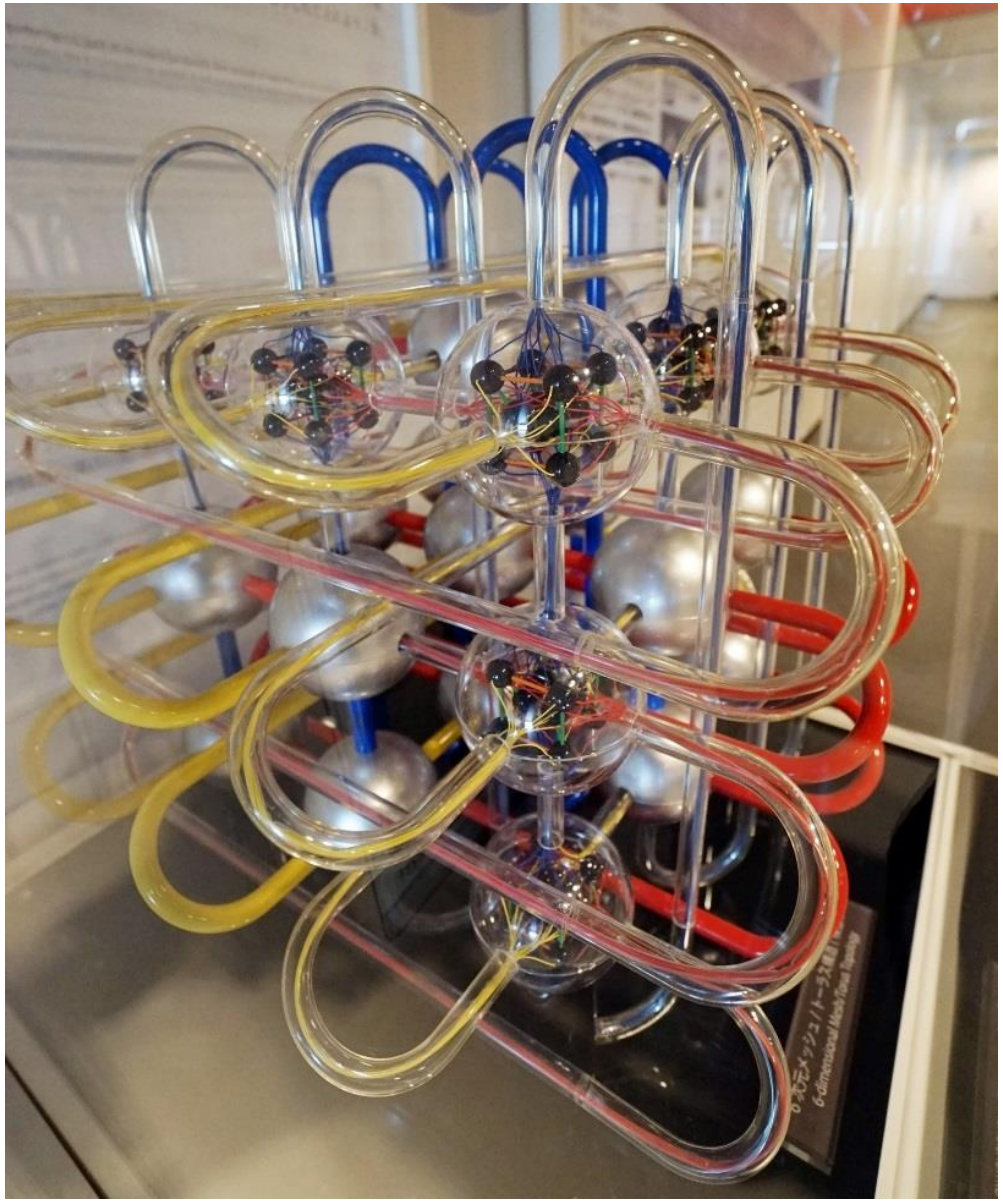


Tofu Interconnect D (TofuD)

- Tofu interconnect series
  - High scalable and fault tolerant 6D mesh/torus network for large scale systems
  - Arbitrary 3-Dimentional **torus** topology can be extracted from 48x36x48 (current maximum for a user), whereas a general torus network will be **mesh** if you extract a part of it



Tofu (豆腐) is Tofu even after cut as you like.
TofuD is torus even after cut ☺



Summary: parallelism in modern HPC systems

- Multi-nodes connected by network
  - multi-processors (multi-sockets) in a single node
    - multi-cores in a single processor
      - SIMD instructions in a single core
        - pipelined

} automatically parallelized by compilers and hardware !

※ you can help compilers to generate more efficient program, even you can parallelize your code on these levels

- A programming model for a level of parallel architectural level
- Hybrid of parallel programming models for a whole system

- Discuss parallel programming model for these levels+

Some important concepts about parallel programming

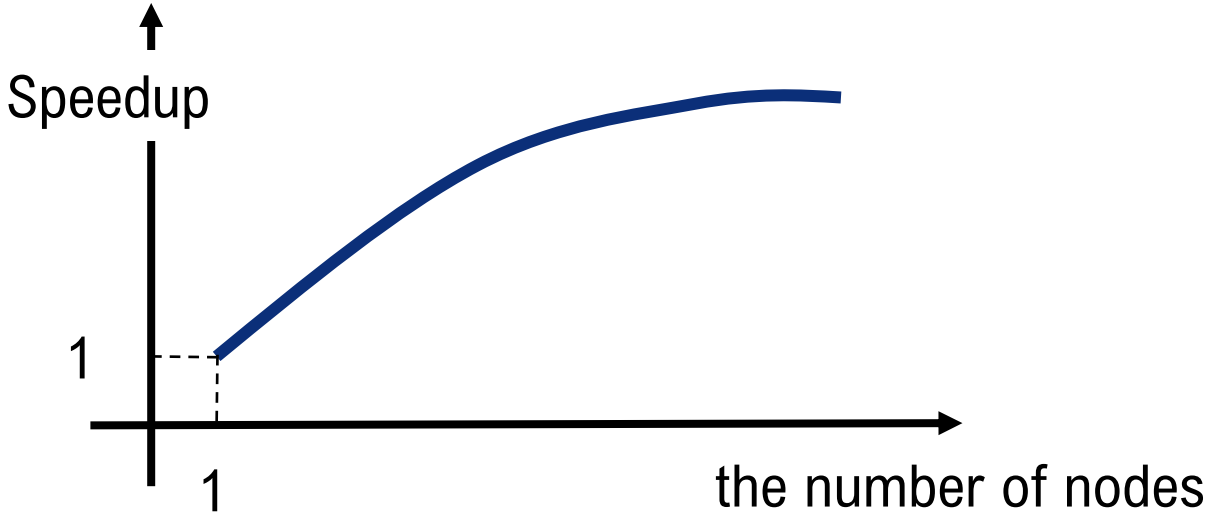
- Speedup
- Amdahl's law
- Weak scaling vs Strong scaling

Speedup

- the relative performance of two systems processing the same program
  - typically, the relative performance of parallel and serial executions

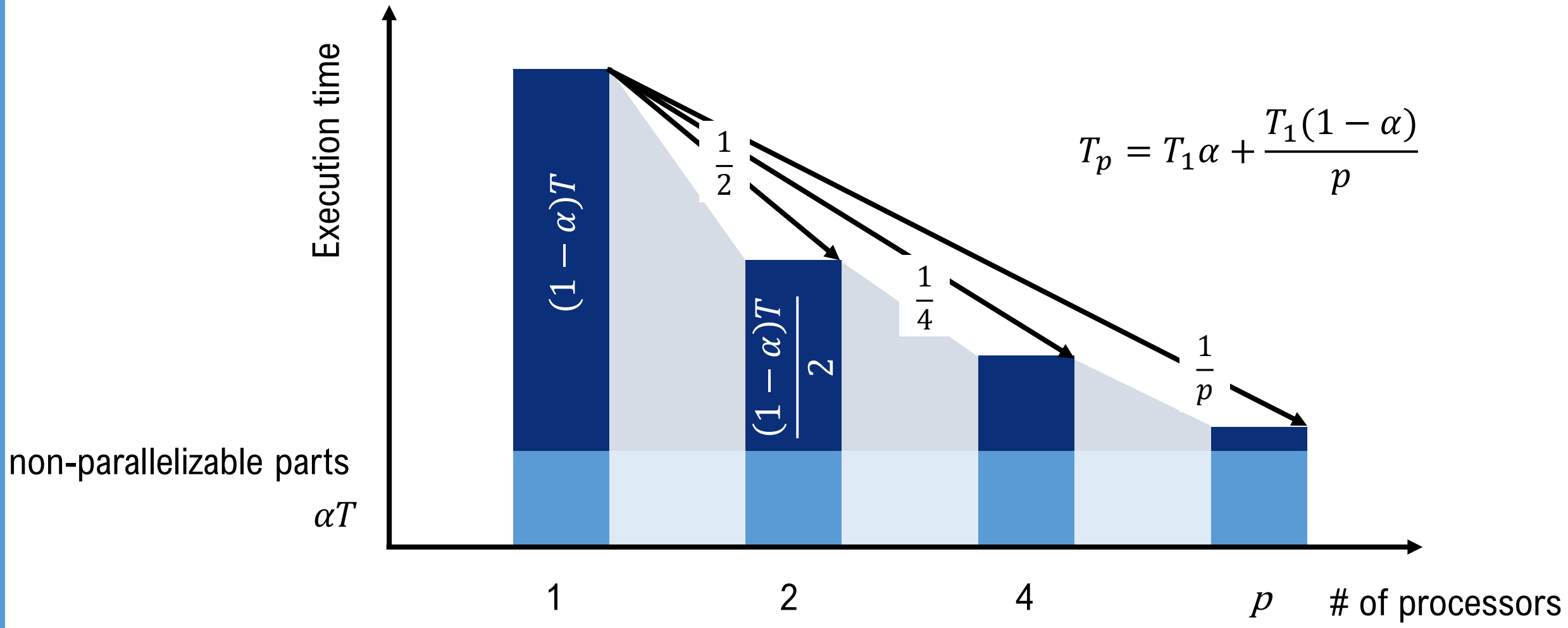
$\frac{T_1}{T_p}$ execution time w/ 1 processor / execution time w/ p processors

- If we use 2 processors and the execution time becomes the half of the 1 processor case, then the speedup is “2”



Amdahl's law

- Total execution time: T_1 on a single processor
- Total execution time: T_p on p processors
- α is the ratio of non-parallelizable part
- P processors take $T_1\alpha$ (sec) for the part
- The rest can be parallelized: $\frac{T_1(1-\alpha)}{p}$



$$T_p = T_1\alpha + \frac{T_1(1-\alpha)}{p}$$

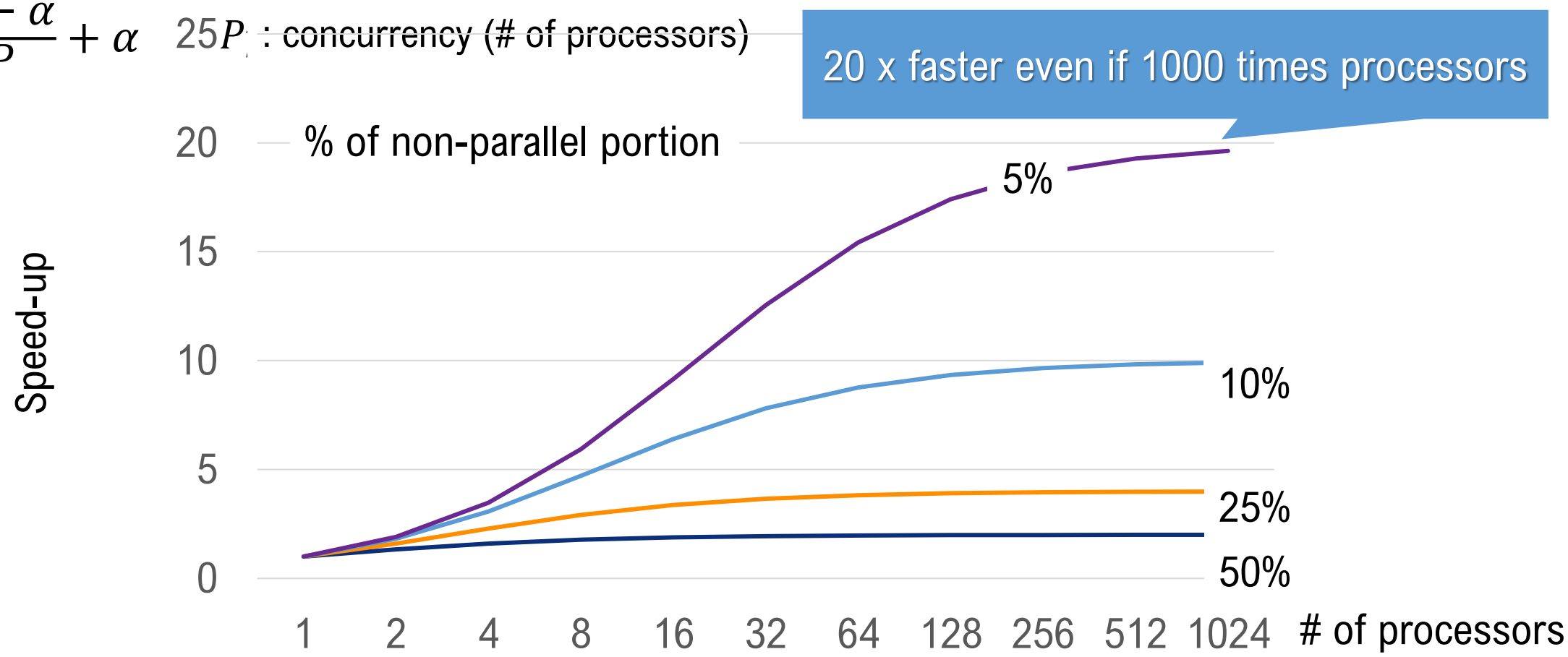
Amdahl's law

- The speed-up from parallelization
  - ✗ 2 processors ⇒ 2 times faster

$$S(s) = \frac{1}{\frac{1-\alpha}{P} + \alpha}$$

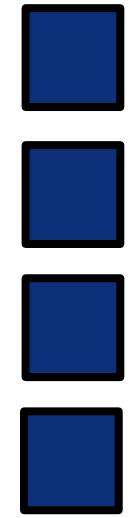
α the fraction of running time a program spends on non-parallelizable parts

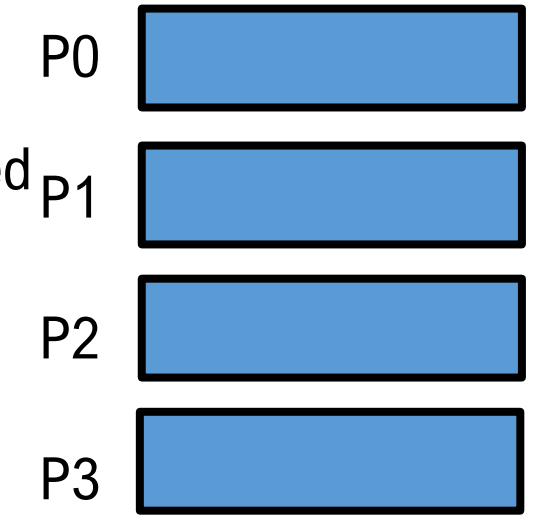
P : concurrency (# of processors)



Weak scaling vs Strong scaling

- We do not have to solve the program size of a small system for large systems
  - Parallelism : Faster time to solution / **larger computing problems(scalability)**
- **Strong scaling**
  - Fixed problem size while the number of processors are increased
  - problem size for each processor becomes smaller
  - Amount of communication between processors may be smaller or stay constant or grow
  - Limited scalability due to Amdahl's low
- **Weak scaling**
  - Fixed problem size for each processor
  - (Total) problem size increases when the number of processors are increased
  - Amount of communication between processors remains constant or grows
    - Note: Communication overhead may grow even the amount remains constant, because of synchronization overhead, etc..





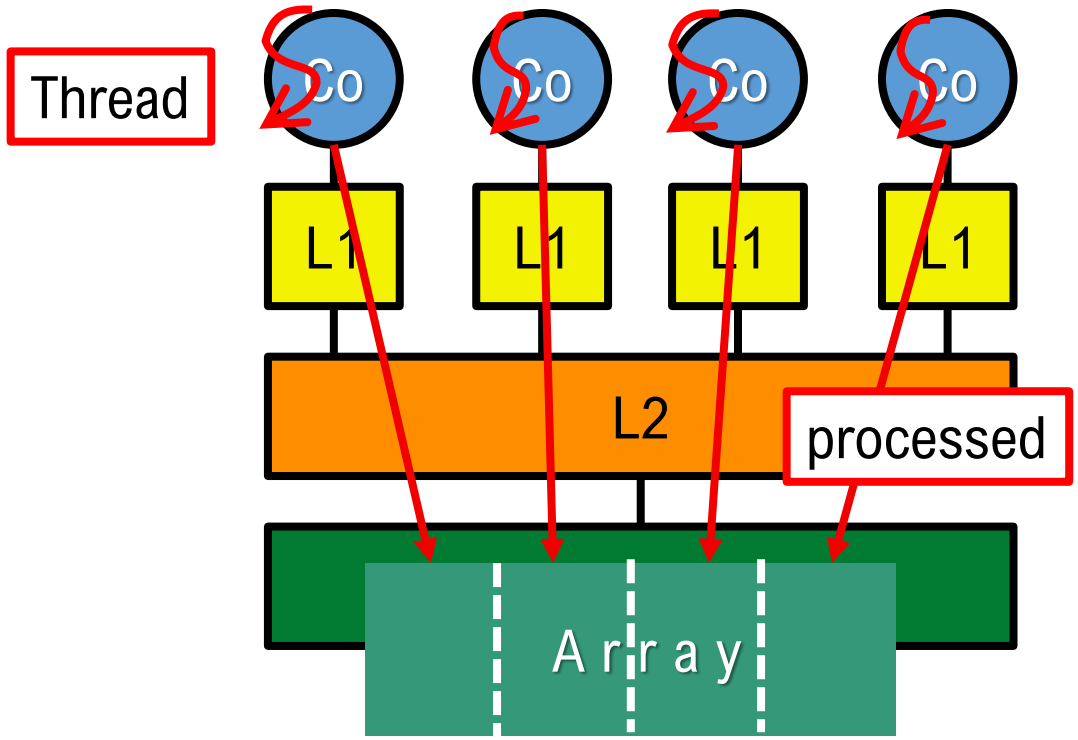
Parallelization and parallel programming

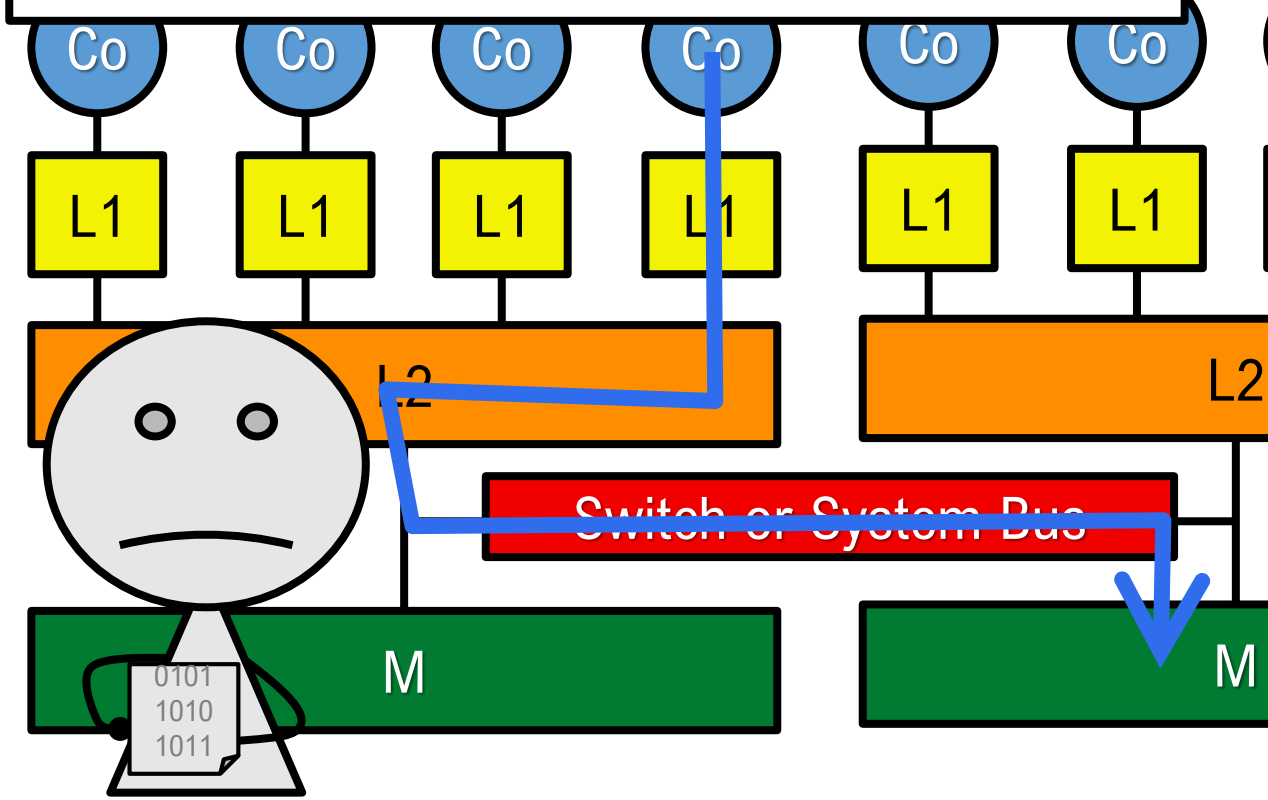
- Shared memory programming
  - Overview
  - OpenMP
- Distributed parallel programming, Message Passing
  - Overview
  - MPI
- Hybrid programming
  - OpenMP+MPI

Shared memory programming model

- Threads share a common address space
- on the shared memory architectures
- Easy to program (/extend) from a serial code

※ Note:
Process: independent, separate memory space
Thread: subsets of a process, shared memory space. Two or more threads can share a core (Hyperthreading, simultaneous multithreading)

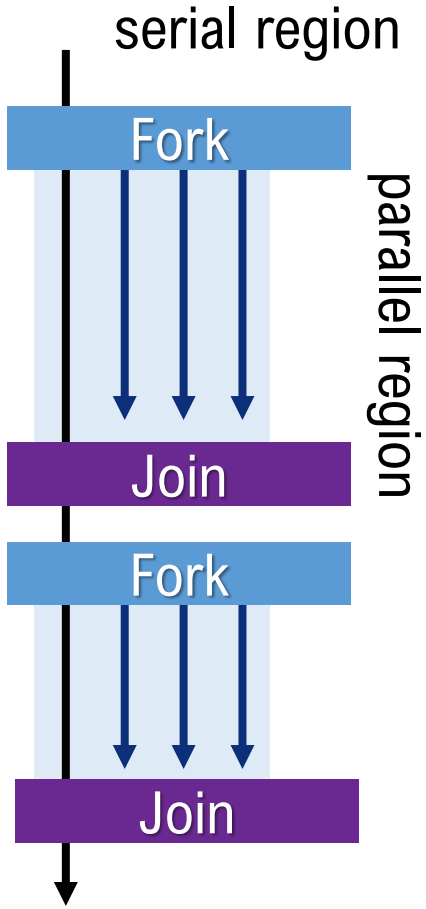




Shared memory programming model: OpenMP

- Most popular parallel programming language (and library) for the shared memory programming
- "Fork-Join" execution model
  - A *parent* thread calls "**Fork**" to create new threads
    - The parent thread continues operation, the children threads also start operation
  - "**Join**" is called by both the parent and children threads
    - Children exit
    - Parent waits until all children join
    - Parent continues operation (serial)
- Directive based
  - insert directives into C/fortran

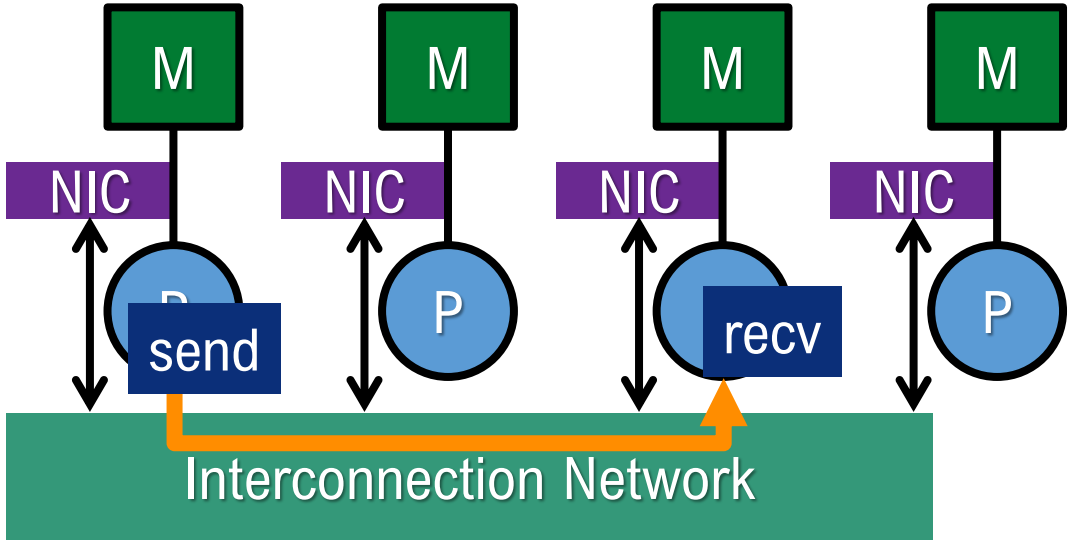
```
PROGRAM TEST
print *,"Thanks"
!$OMP PARALLEL
print *,"Many Thanks"
!$OMP END PARALLEL
END PROGRAM TEST
```



Distributed parallel programming model, Message Passing

- In distributed parallel systems, each processor can not access all data
- Processors must access non-local data by *communication*
  - Message passing Interface (MPI)
  - Parallel Virtual Machine (PVM)
  - etc..
- Applications must be parallelized explicitly
  - work mapping
  - data distribution
- Scalable from the viewpoint of construction
  - Just increase the number of nodes

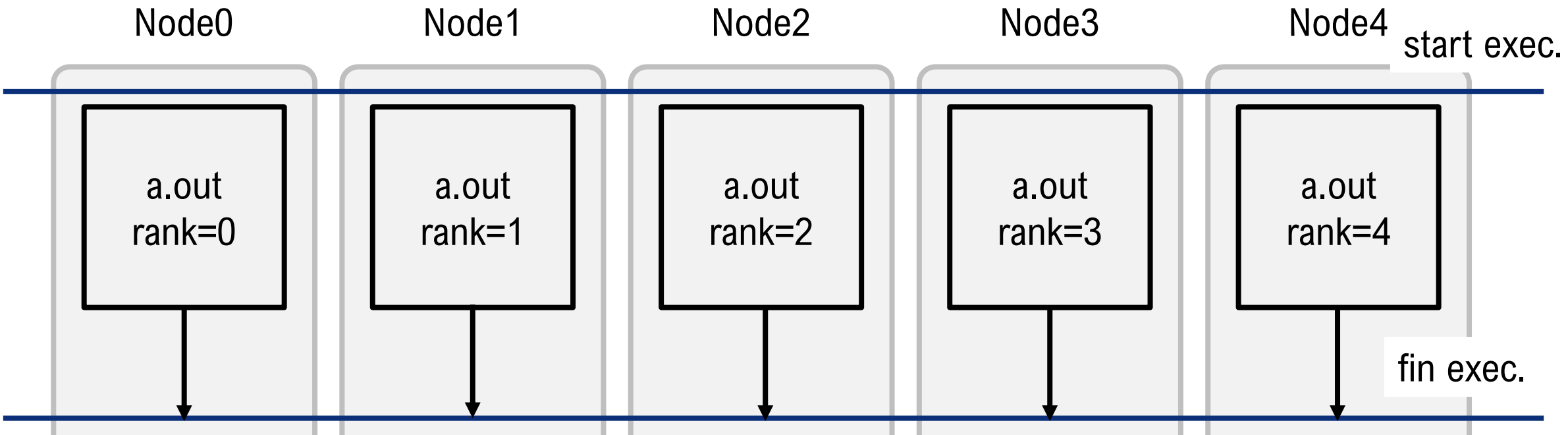
- Note: Distributed parallel programming model is available on shared memory systems



MPI : Message Passing Interface

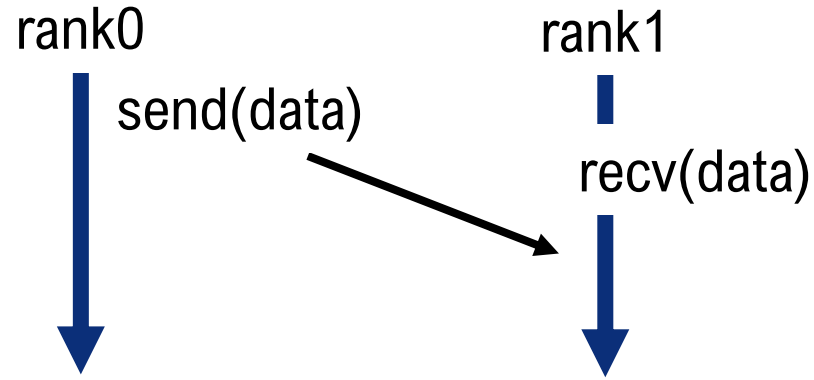
Note: MPI is not a programming language. MPI is a message passing interface specification.

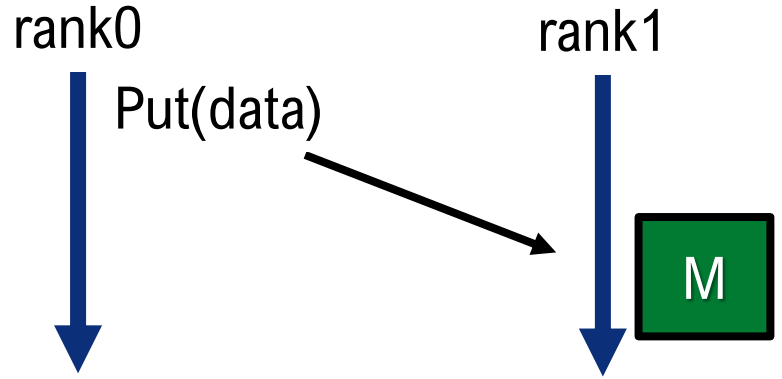
- de fact standard for parallel programming for distributed parallel systems
- SPMD programming model
  - SI**MD** : Single Instruction Multiple Data
  - MI**MD** : Multiple Instruction Multiple Data
  - SPMD: Single **Program** Multiple Data
    - a same binary runs on multiple nodes to process multiple data
    - use if (rank=**) to assign a special work on a certain process



MPI: communication types

- Cooperative operations
  - cooperatively exchanged in message passing
    - receiver explicitly allocate memory space to receive
  - explicitly sent by a process and received by another
  - communication and synchronization are combined
- One-sided operations
  - remote memory reads/writes
  - only one process needs to explicitly participate
  - communication and synchronization are not combined
    - faster
  - Programmers must take care about local memory control





MPI: Communication types, from a different perspective

- Pairwise (point-to-point) communication
  - communication between 2 processes
    - Send/Recv
- Collective communication
  - communication between multiple processes
    - a group of all processes
    - a group of some processes
  - ex: send a data from rank-0 to all processes
  - rank-0 receives data from all other processes

MPI : point-to-point communication

- rank=0 sends a number to rank=1
- rank=1 receives a number from rank=0, and print the number

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int nprocs, myrank;
    double number;

    MPI_Init(&argc, &argv);                    // Initialize the MPI execution environment
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // Get the number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // Get the rank of the calling process
    if (myrank == 0) {
        number = 1.0;
        MPI_Send(&number, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    }else if (myrank == 1) {
        MPI_Recv(&number, 1, MPI_DOUBLE, 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("recv: %f\n", number);
    }
    MPI_Finalize();                            // Finalize the MPI execution environment
}
```

MPI : point-to-point communication

```
MPI_Send(&number,          const void*   initial address of send buffer
         1,                int           the number of elements
         MPI_DOUBLE,       MPI_Datatype  datatype of each element
         1,                int           rank of destination
         0,                int           message tag, will be used to differentiate messages
         MPI_COMM_WORLD);  MPI_Comm      communicator, which describes a group of processors
                                         MPI_COMM_WORLD is the group of all processors in your job

MPI_Recv(&number,          const void*   initial address of recieve buffer
         1,                int           the number of elements
         MPI_DOUBLE,       MPI_Datatype  datatype of each element
         0,                int           rank of source
         0,                int           message tag
         MPI_COMM_WORLD,   MPI_Comm      communicator
         MPI_STATUS_IGNORE); MPI_Status  a structure including a status of a reception
```

MPI: collective communication

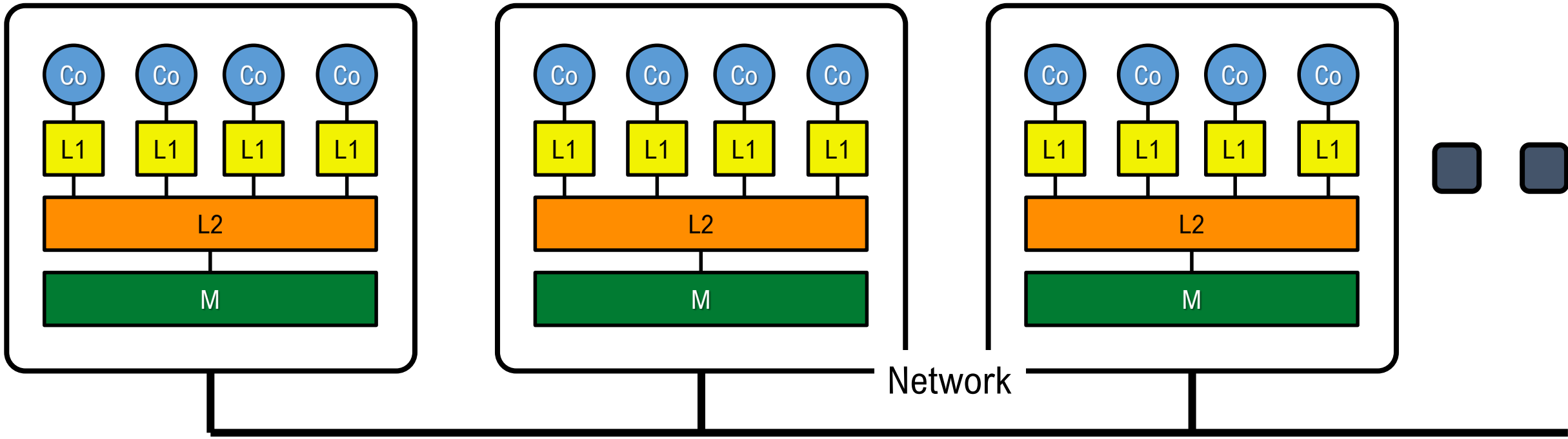
- rank=0 sends “number” to all other processes

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
   int nprocs, myrank;
   double number;

   MPI_Init(&argc, &argv);                  // Initialize the MPI execution environment
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // Get the number of processes
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // Get the rank of the calling process

   if(myrank == 0) number = 1.23;
   else            number = 3.21;

   MPI_Bcast(&number, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   printf(“recv: %f\n”, number);

   MPI_Finalize();                          // Finalize the MPI execution environment
}
```

MPI: collective communication

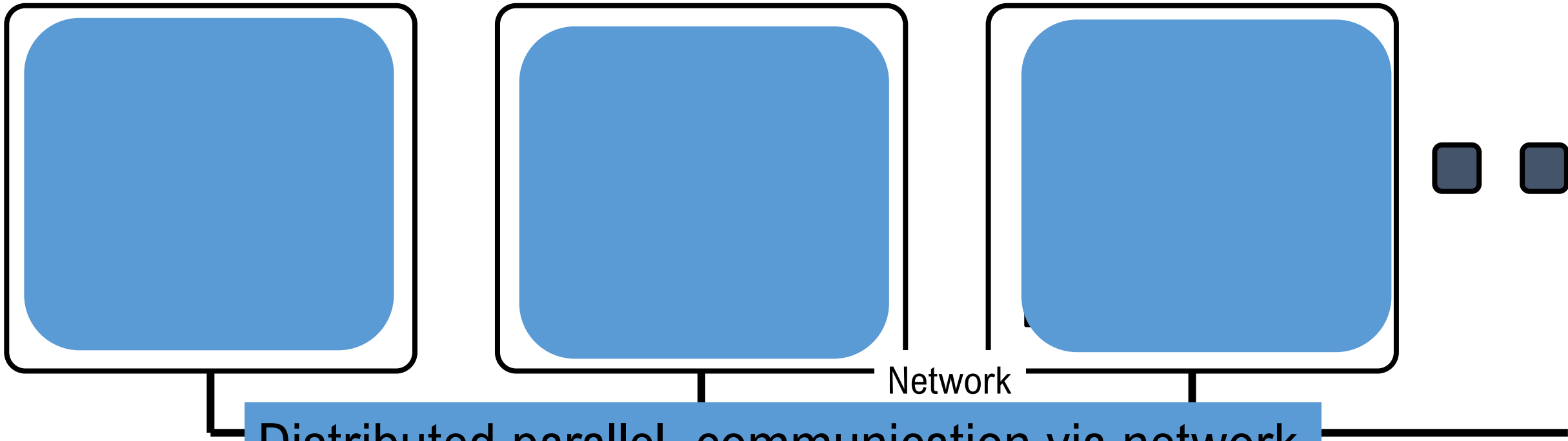
`MPI_Bcast(&number,`	`const void*`	initial address of send buffer
`1,`	`int`	the number of elements
`MPI_DOUBLE,`	`MPI_Datatype`	datatype of each element
`1,`	`int`	rank of root (any process can be a root)
`MPI_COMM_WORLD);`	`MPI_Comm`	communicator

Hybrid of shared memory and distributed parallel



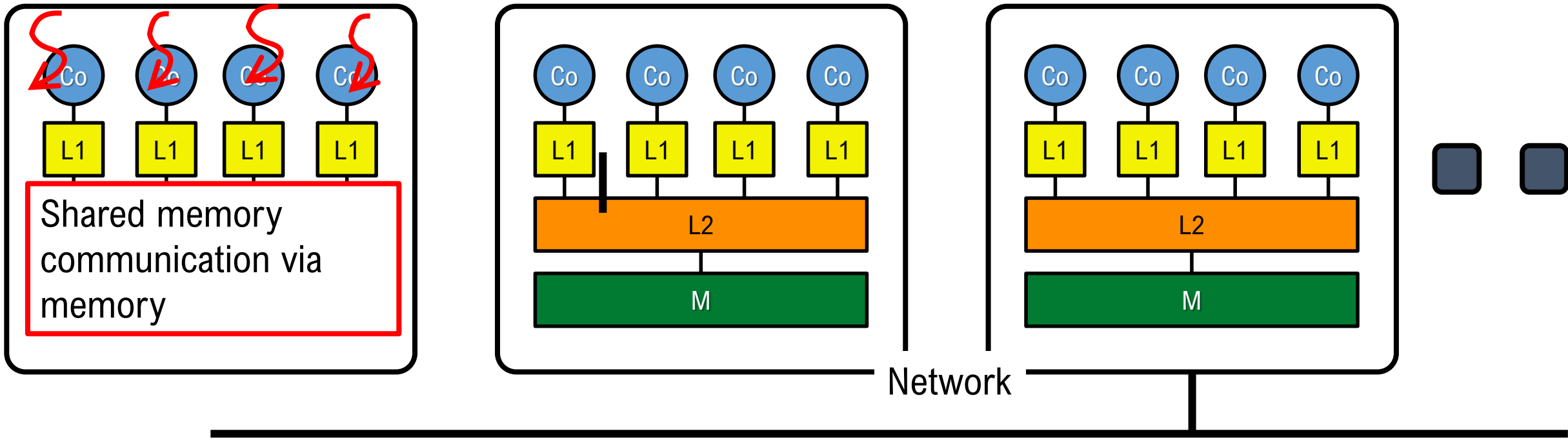
- 2 possible choices of programming models
  - **Hybrid of shared memory and distributed parallel programming models**
    - **OpenMP + MPI (Today's focus)**
  - distributed parallel programming model
    - flat-MPI (remember! sometimes, this is not bad choice)

Hybrid of shared memory and distributed parallel



- 2 possible choices of programming models
  - **Hybrid of shared memory and distributed parallel programming models**
    - **OpenMP + MPI (Today's focus)**
  - distributed parallel programming model
    - flat-MPI (remember! sometimes, this is not bad choice)

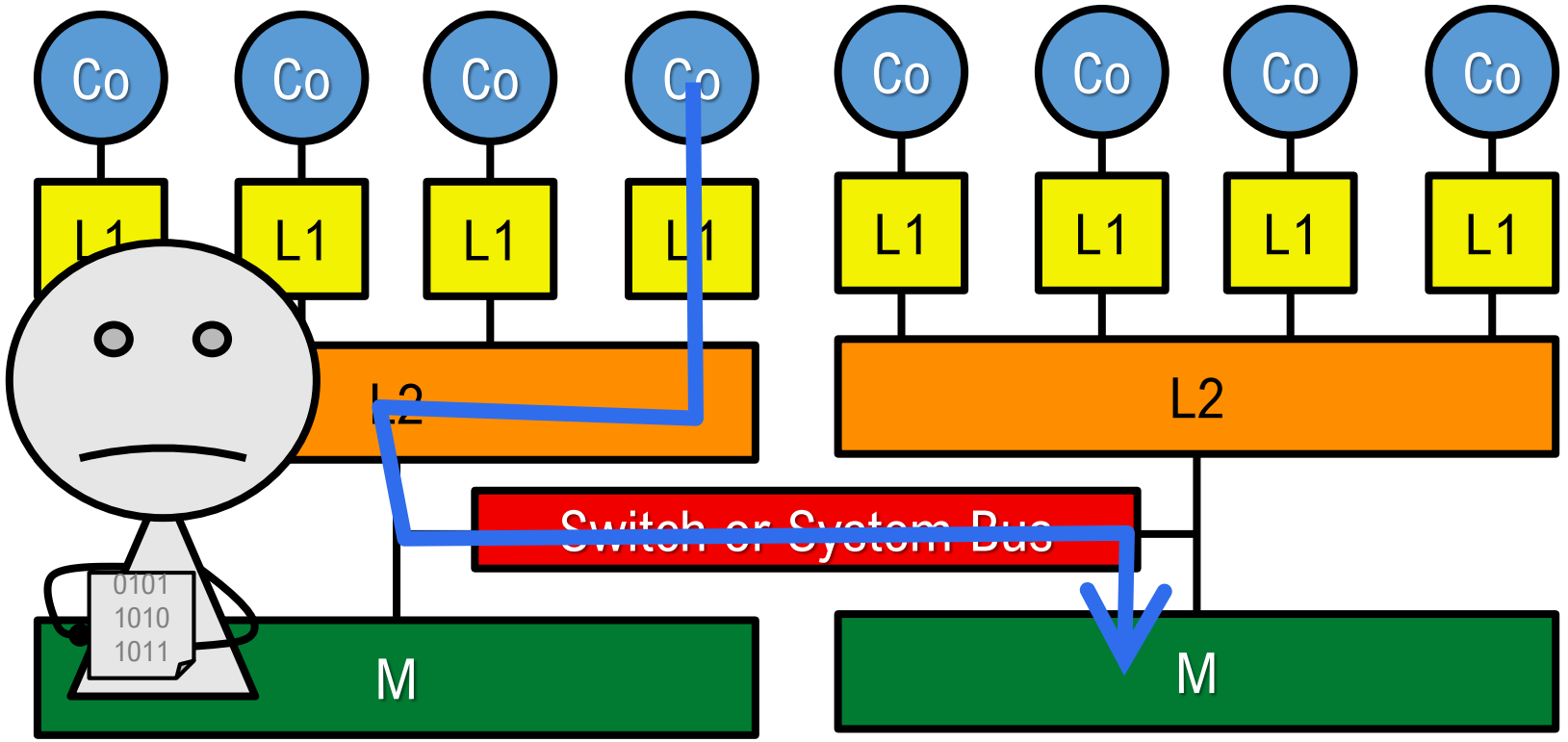
Hybrid of shared memory and distributed parallel



- 2 possible choices of programming models
  - **Hybrid of shared memory and distributed parallel programming models**
    - **OpenMP + MPI**
  - distributed parallel programming model
    - flat-MPI (remember! sometimes, this is not bad choice)

Hybrid of shared memory and distributed parallel

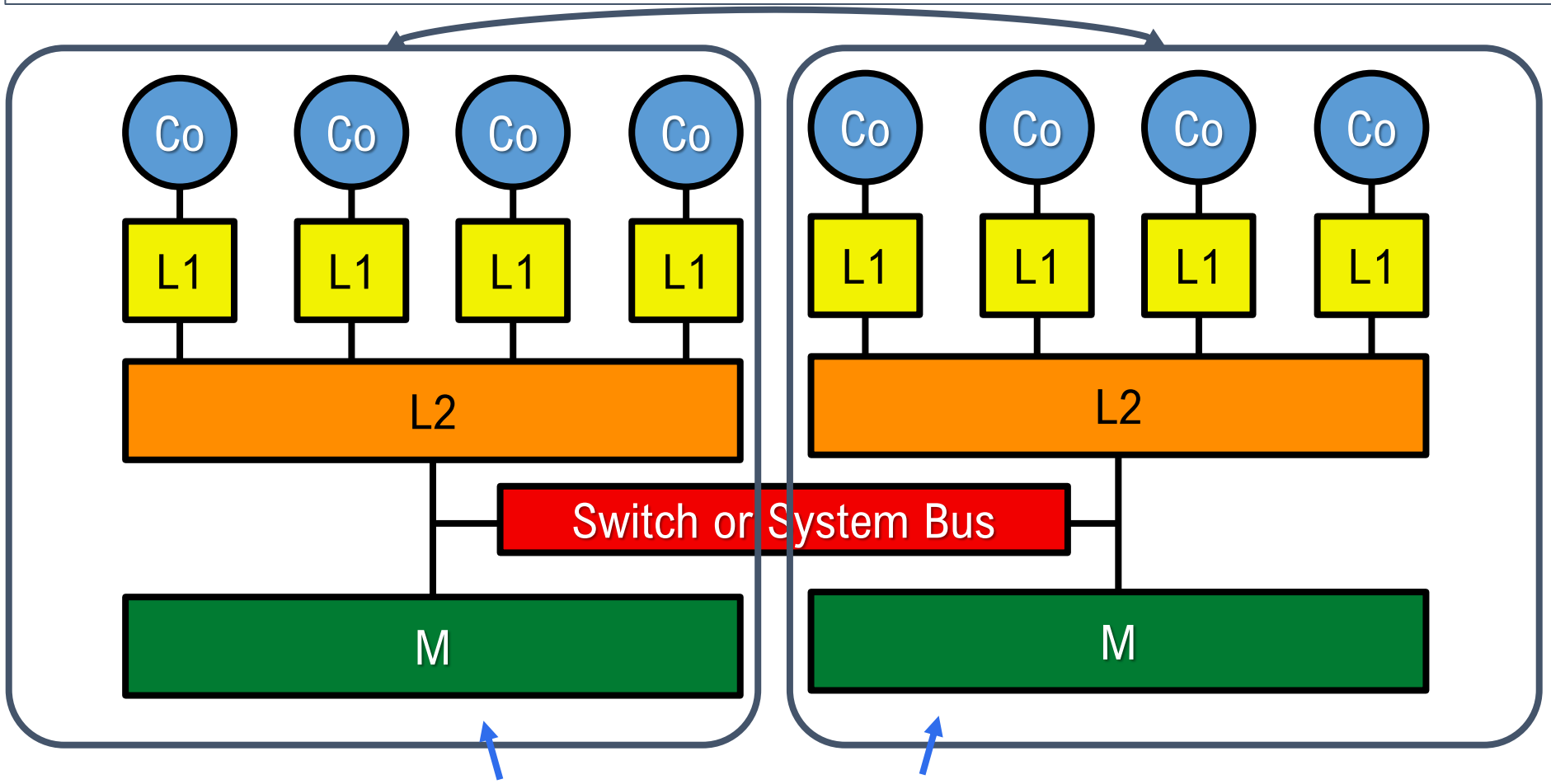
- For NUMA (ex. 2-CPUs in a node), shared memory programming model has “non-local data access” problem

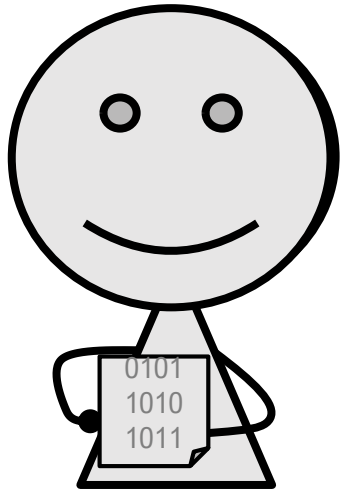


Hybrid of shared memory and distributed parallel

- For NUMA, shared memory programming model has “non-local data access” problem

Distributed parallel programming model between sets of shared memory cores

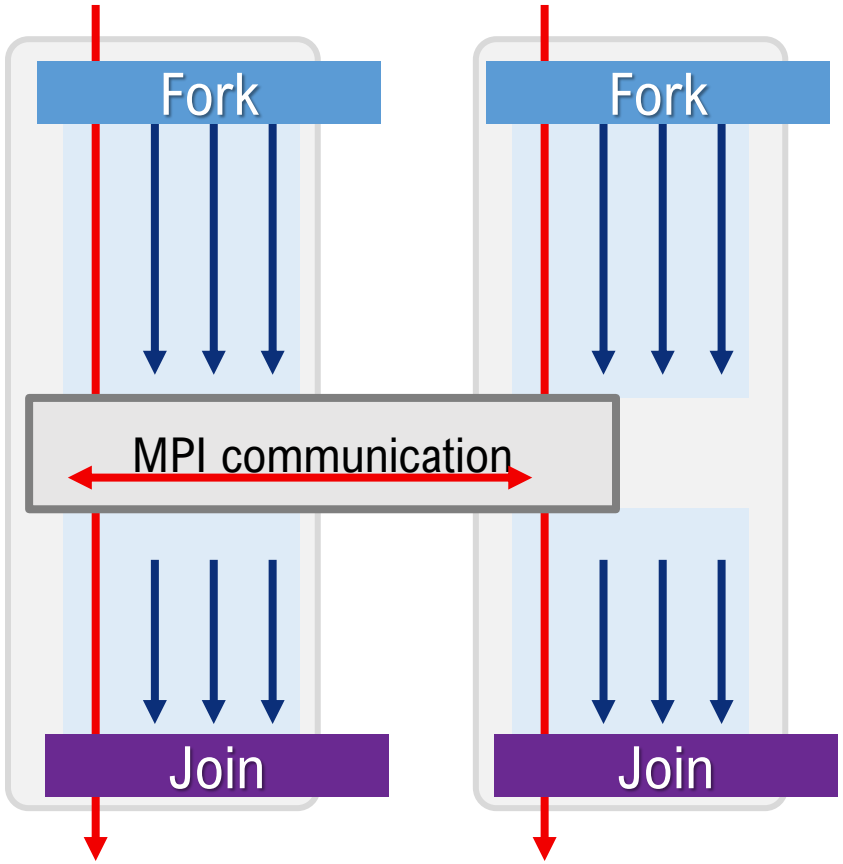




Shared memory programming model inside a set of shared cores

Hybrid of OpenMP+MPI

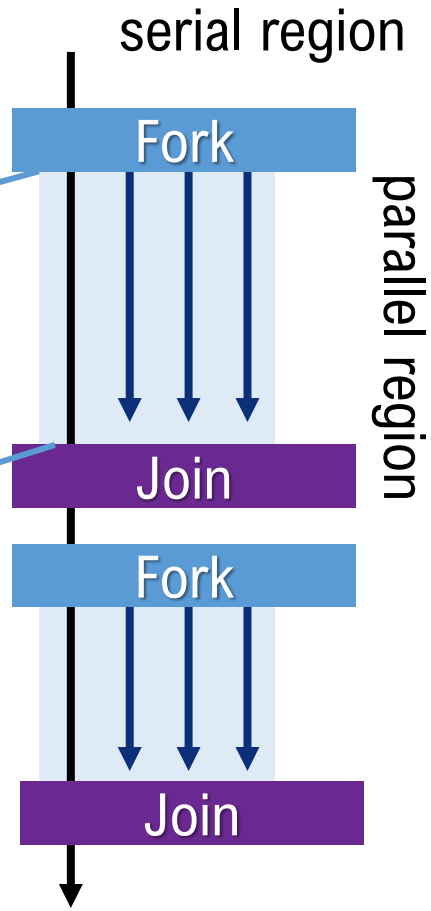
- MPI describes parallelism between processes
- OpenMP provides a shared memory model within a process



Hybrid of OpenMP+MPI

- MPI describes parallelism between processes
- OpenMP provides a shared memory model within a process
- After the MPI_Init_thread, you can fork threads wherever you need

```
MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &prov);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel default(shared) private(iam, np)
{
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hybrid: Hello from thread %d out of %d from process %d out of %d¥n",
           iam, np, rank, numprocs);
}
```



Hybrid of OpenMP+MPI

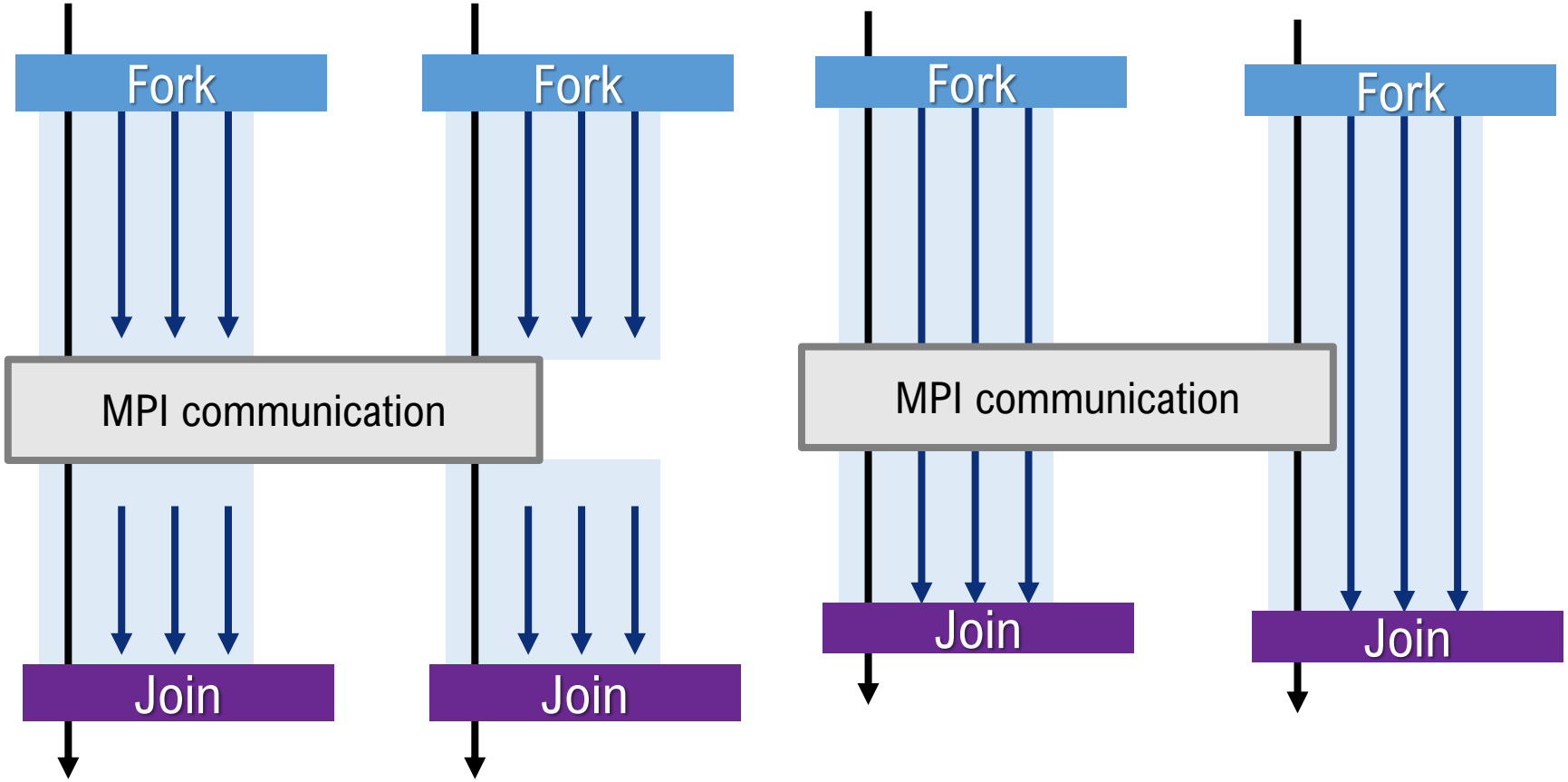
- MPI specification defines four levels of hybrid parallelism to be used with OpenMP-programming
  - MPI implementation do NOT always support all of them ☹

  - MPI_THREAD_SINGLE
    - Only one thread will call communication interface at once
  - MPI_THREAD_FUNNELED
    - The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
  - MPI_THREAD_SERIALIZED
    - The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
  - MPI_THREAD_MULTIPLE
    - Multiple threads may call MPI, with no restrictions.

https://www.mpich.org/static/docs/latest/www3/MPI_Init_thread.html

Hybrid of OpenMP+MPI: Overlapping communication and computation

- The hybrid of OpenMP+MPI allows us to overlap communication and computation
- The threads which do not call MPI functions can continue to other works (not communication)



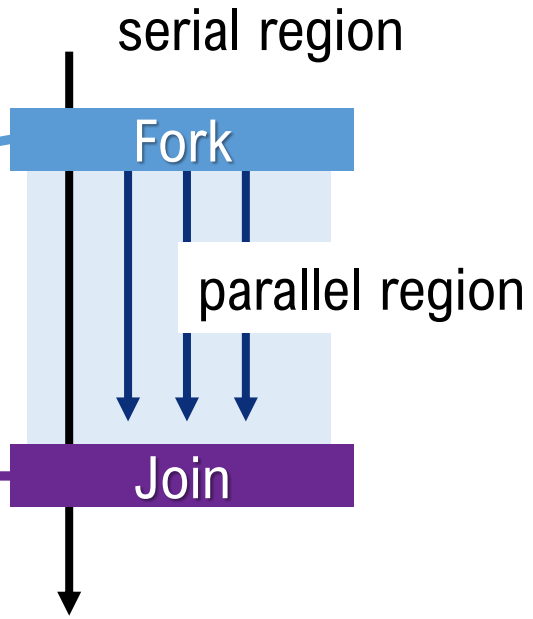

```
# pragma omp parallel
{
 // computation
    …..
#pragma omp single
  MPI_function();

….
}
```

Hands-on Practice

OpenMP: Hello world

- The directive to fork threads is #pragma omp parallel
- The threads join at the end of the region

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

int main(int argc, char **argv)
{
#pragma omp parallel
{
  printf("Hello world\n");
}
}
```



Exercise: Hello world (OpenMP)

- write the hello world code, compile and run with 12 threads
  - compile:

    $ fccpx -Kfast,openmp <your-source-code.c>
  - execution: see next page
- To set the number of threads, set the environmental variable OMP_NUM_THREADS
  - for example, **export OMP_NUM_THREADS=12**
  - Edit your job-script and insert the command to set the environmental variable
- Check the result
  - *may be,* you can find “Hello world” repeated 12 times

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

int main(int argc, char **argv)
{
#pragma omp parallel
{
  printf("Hello world\n");
}
}
```

Exercise: Hello world (OpenMP)

- write a job script

```
#!/bin/bash
#PJM -L "node=1"
#PJM -L "rscunit=rscunit_ft01"
#PJM -L "rscgrp=small"
#PJM -L "elapse=0:10:00"
#PJM -S

#------- Program execution -------#
export OMP_NUM_THREADS=<your number of threads>
./<your binary>
```

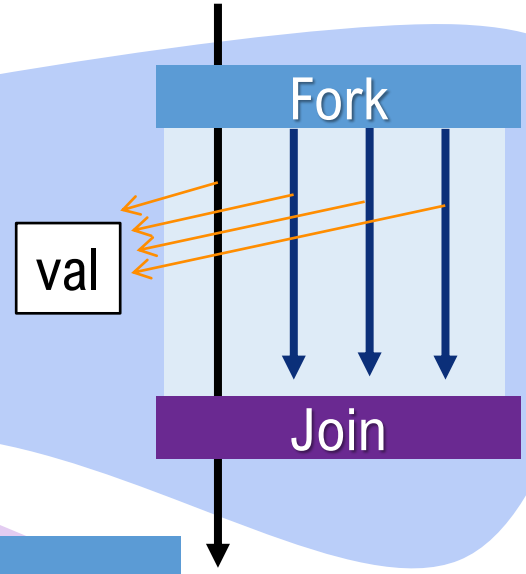
- submit your job

$ pjsub <your-job-script-file.sh>

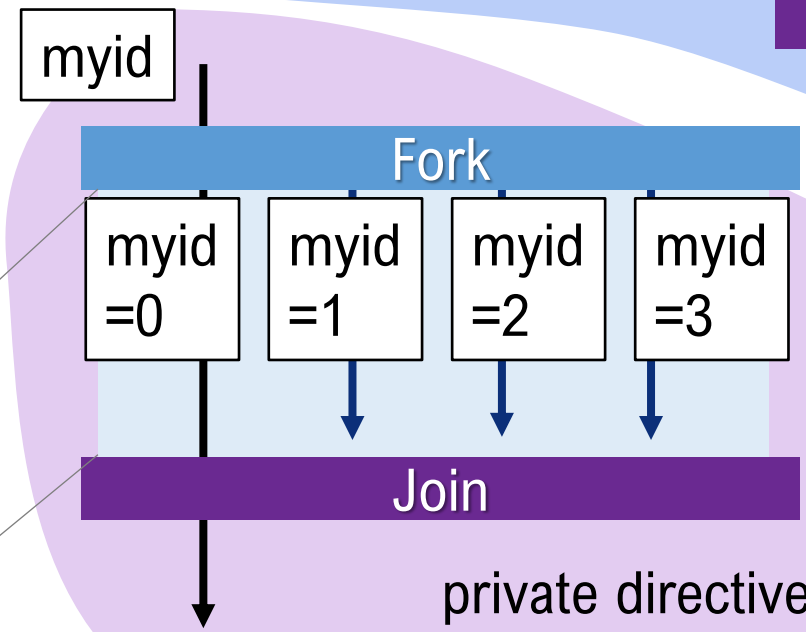
OpenMP: private data and shared data

- The basic idea of the shared programming model is that variables are shared by default, i.e. thread can read/write arbitrary variables
  - sometimes, threads need their own “private” workspace
- By using *private* clause, you can make a separate copy for each thread

default: all threads access all variables




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

int main(int argc, char **argv)
{
  int myid;

#pragma omp parallel private(myid)
{
  myid = omp_get_thread_num();
  printf("Hello world I'm %d\n",myid);
}
}
```



private directive enable us to make the copies of the variables

OpenMP: private data and shared data (FYI)

- private: each thread has its own instance of a variable
- firstprivate: each thread has its own instance of a variable. The variable must be initialized before the parallel region. At the beginning of the parallel region, the variables in all threads have a same initial value.
- lastprivate: : each thread has its own instance of a variable. The final value can be transmitted to the shared variable outside the parallel region.
- shared(default) : variable(s) can be shared among all threads. If you don’t specify any data type, then the variables should be shared in the parallel region.

Exercise: Hello world from who?

- write, compile and run the following code w/ 12 threads

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

int main(int argc, char **argv)
{
  int myid;

#pragma omp parallel private(myid)
{
  myid = omp_get_thread_num();
  printf("Hello world I'm %d\n",myid);
}
}
```

Note

- omp_get_thread_num() is a function to obtain its thread-id.
- Another important function is omp_get_num_threads(), which gives the number of threads in the region.

OpenMP: set the number of threads

- You've already learned the way to specify the number of threads by using the environmental variable OMP_NUM_THREADS, omp_set_num_threads(num) also allow us to set the number of threads.

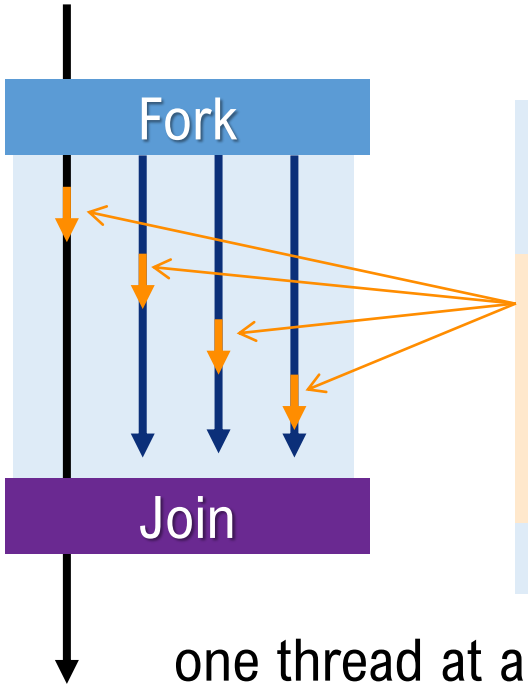
```
#include<omp.h>
#include<stdio.h>

int main(int argc, char **argv)
{
  int myid;
  int nthreads;

  omp_set_num_threads(4);
#pragma omp parallel private(myid, nthreads)
{
  myid     = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf("Hello world I'm %d of %d\n",myid, nthreads);
}
}
```

Exercise: the number of threads

- write, compile and run the code at the previous slide by specifying **OMP_NUM_THREADS=12** at the job-script
- check the results
  - how many threads can you find?

OpenMP: synchronization

- The synchronization directives allow us to control the order of execution of threads
- #pragma omp barrier: synchronizes all threads in the parallel region; all threads pause at the barrier, until all threads execute the barrier
- #pragma omp critical: specifies mutual exclusion. Only one thread at a time can enter a critical region




```
#pragma omp parallel
{
   ...
#pragma omp critical
{
  jobcall(A, B, n);
}
}
```

one thread at a time

- #pragma omp atomic: Only one thread at a time can update the specified variable

Exercise: synchronization

- write, compile and run the following two codes w/ 28 threads, and compare the results

```c
#include<omp.h>
#include<stdio.h>

int main(int argc, char **argv)
{
  int myid;
  int val;

  val = 1000;
#pragma omp parallel private(myid) shared(val)
{
  myid     = omp_get_thread_num();
  printf("myid = %d, val = %d\n",myid,val);
  if(myid==0){
    val = myid;
  }
  printf("myid = %d, val = %d\n",myid,val);
}
}
```

```c
#include<omp.h>
#include<stdio.h>

int main(int argc, char **argv)
{
  int myid;
  int val;

  val = 1000;
#pragma omp parallel private(myid) shared(val)
{
  myid     = omp_get_thread_num();
  printf("myid = %d, val = %d\n",myid,val);
  if(myid==0){
    val = myid;
  }
#pragma omp barrier
  printf("myid = %d, val = %d\n",myid,val);
}
}
```

OpenMP: loop

- loop work-sharing is a typical way to share workloads by threads to speed-up!
- scheduling and work-assignment can be specified by clauses such as *schedule*

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>

#define N 1000

int main(int argc, char **argv)
{
  int i;
  double rmax;
  double A[N], B[N];

  rmax = 1.0/(double)RAND_MAX;

  for(i=0; i<N; i++) A[i] = ((double)random())*rmax;
  for(i=0; i<N; i++) B[i] = ((double)random())*rmax;


#pragma omp parallel private(i)
{
#pragma omp for
  for(i=0; i<N; i++){
    A[i] = A[i]+B[i];
  }
}

}
```

Note: these are equivalent

```
#pragma omp parallel for
  for(i=0; i<N; i++){
    A[i] = A[i]+B[i];
  }
```

loop index “i” is private by default in *parallel for*

OpenMP: reduction

- Standard reduction expressions such as +, max, min, can be defined in the reduction clause. *reduction(op: list)*
- If the reduction is declared, local copies of *"list"* are made in all threads, local results are stored in the local copies, and the local copies are reduced into a single shared value.

```
#pragma omp parallel for reduction(+:total_sum)
for(i=0; i<N; i++){
  total_sum += A[i];
}
```

these are equivalent

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>

#define N 1000

int main(int argc, char **argv)
{
  int i;
  double rmax, sum, total_sum;
  double A[N];

  rmax = 1.0/(double)RAND_MAX;

  for(i=0; i<N; i++) A[i] = ((double)random())*rmax;

  total_sum = 0.0;
#pragma omp parallel private(i, sum) shared(total_sum)
{
  sum = 0.0;
#pragma omp for
  for(i=0; i<N; i++){
    sum += A[i];
  }
  printf("%f\n",sum);
#pragma omp atomic
  total_sum += sum;
}

printf("total sum = %f\n",total_sum);
}
```

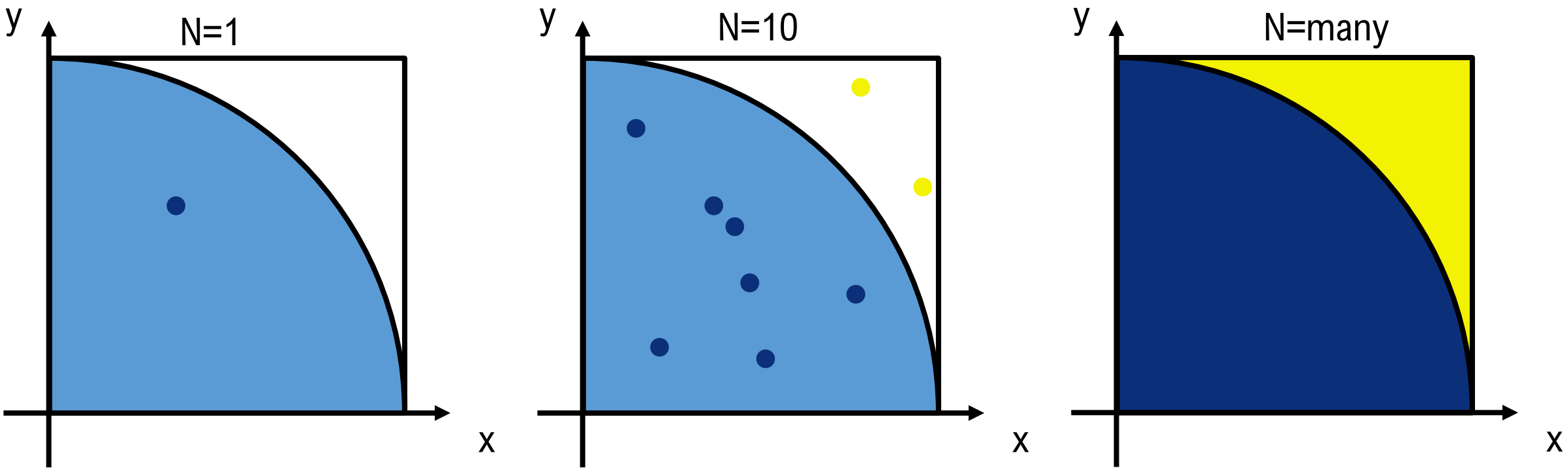
Exercise: parallelize the pi code by using OpenMP

and check execution time by changing the number of threads, 1, 2, 14, 28

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>

#define N 100000000
double mytime();

int main(int argc, char **argv)
{
  int i, n, seed;
  double x, y;
  double t0, t1;
  struct drand48_data drand_buf;

  t0 = mytime();

  seed = 0;
  srand48_r (seed, &drand_buf);

  n = 0;
  for(i=0; i<N; i++){
    drand48_r (&drand_buf, &x);
    drand48_r (&drand_buf, &y);

    if(x*x + y*y < 1.0){
      n++;
    }
  }

  t1 = mytime();
  printf("pi = %f time=%f¥n",4.0*(double)n/(double)N,
t1-t0);

}

double mytime()
{
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + tv.tv_usec*1e-6;
}
```

Exercise: parallelize the pi code (FYI)

- Compute pi using Monte Carlo method
  - generate N sets of two random numbers of [0, 1] : (x,y)
  - if (x, y) is in the inside of quadrant of radius 1, then n++
  - the ratio of n/N approximates the area of the quadrant
    - n/N = (1*1*pi)/4
    - pi = 4*n/N



Exercise: parallelize the pi code

- Hints:
  - drand_buf, seed, must be “private”

MPI (review) : Message Passing Interface

- MPI is not a programming language.
- MPI is a message passing interface specification.
- Programmers call MPI functions to communicate
- MPI_Init function initializes the MPI execution environment. All other MPI functions must be called **after** the MPI_Init
- MPI_Finalize function finalizes the MPI execution environment. All processes must call this before exiting.

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

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  /* starts MPI */

  printf("Hello world\n");

  MPI_Finalize();
  /* exits MPI */
  return 0;
}
```

MPI: Hello world

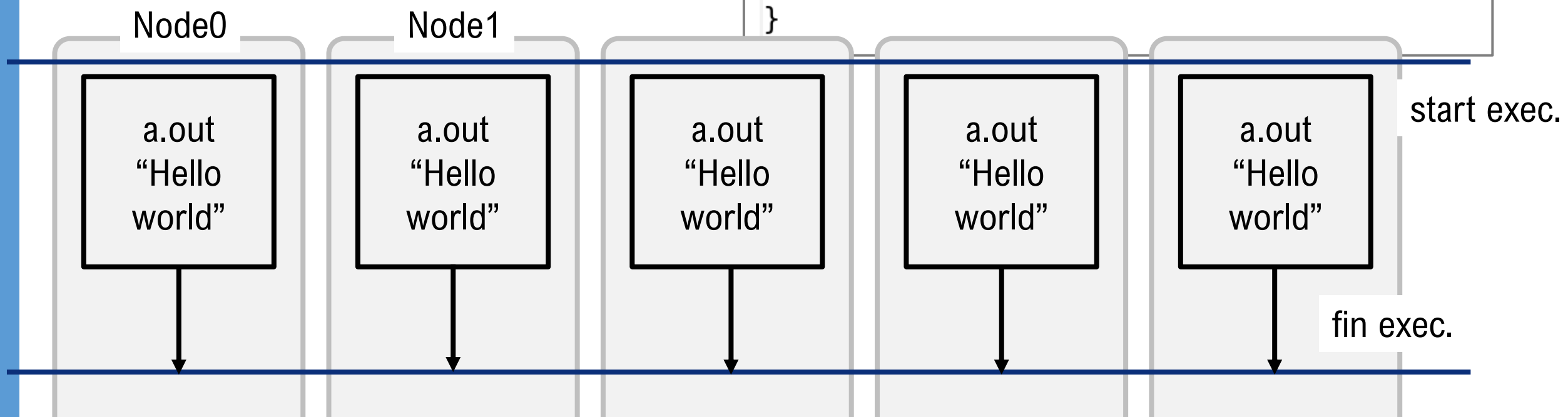
- MPI programs must be launched by a command for execution such as mpirun, mpiexe:

$ mpiexec.hydra –n *num your_binary*

- This runs *num* copies of *your_binary*

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

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  /* starts MPI */

  printf("Hello world\n");

  MPI_Finalize();
  /* exits MPI */
  return 0;
}
```

includes MPI definitions and types



MPI: Hello world from who?

- MPI library provides functions to give
  - ID number (rank) of a specific process
  - the number of processes in a program
- MPI_Comm_rank gives the rank of the calling process in the communicator (MPI_COMM_WORLD)
- MPI_Comm_size gives the number of processes in the communicator.

Note: *Communicator* contains a list of processes attending a program. MPI_COMM_WORLD is the default communicator

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

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

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  /* gets the ID number of "this" process */

  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  /* gets the number of processes */

  printf("I'm number %d of %d processors\n",myrank, nprocs);

  MPI_Finalize();

  return 0;
}
```

Exercise: Hello world from who (MPI)

- write the hello world from who code in the previous page, compile and run with 4 processes

Here is an example of a jobscript

```
#!/bin/bash
#PJM -L "node=1"
#PJM -L "rscunit=rscunit_ft01"
#PJM -L "rscgrp=small"
#PJM -L "elapse=0:10:00“
#PJM --mpi "max-proc-per-node=4"
#PJM -S

#------- Program execution -------#
mpiexec -n 4 ./<your binary>
```

MPI: Synchronization

- MPI also provides a function to synchronize all processes in a communicator, MPI_Barrier

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

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

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  MPI_Barrier(MPI_COMM_WORLD);
  /* Blocks until all processes in the communicator have
     reached this routine. */

  printf("I'm number %d of %d processors\n",myrank, nprocs);

  MPI_Finalize();

  return 0;
}
```

MPI: Pairwise communication

- MPI_Send sends the buffer to the specified rank (dest)
- MPI_Recv receives the buffer from the specified rank (source)

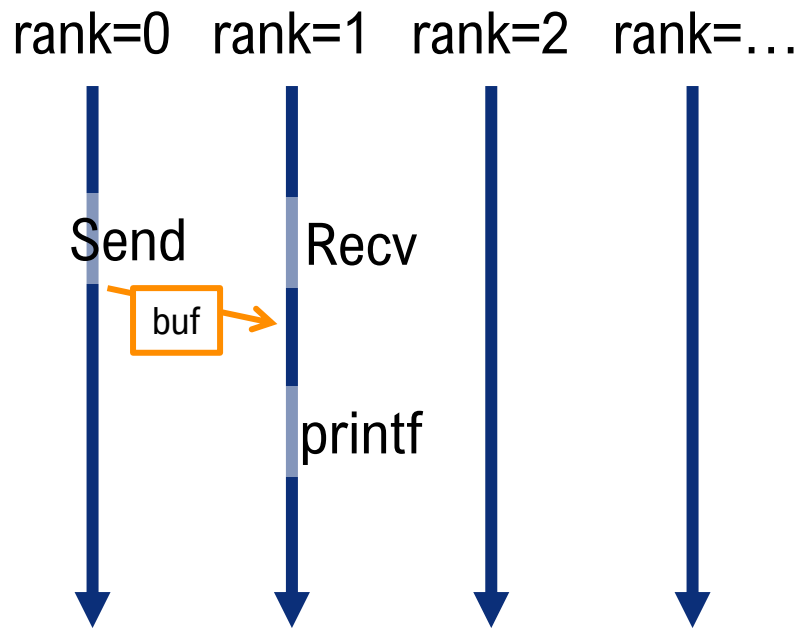
```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

- **buf:** the pointer to buffer to be sent/received
- **count**: the number of elements in the buffer
- **datatype**: MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, …
- **dest/source**: rank of destination/source
- **tag**: message tag
- **comm**: communicator

Remarks: If there is no corresponding MPI_Recv function call in *dest* processes, MPI_Send can never be success, and never finish. vice, sersa

MPI: Pairwise communication

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

int main(int argc, char **argv)
{
  int myrank, nprocs;
  int yourrank;
  MPI_Status stat;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  if(myrank==0){
    MPI_Send(&myrank, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
  }else if(myrank==1){
    MPI_Recv(&yourrank, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &stat);
    printf("I'm number %d, I've received %d\n",myrank, yourrank);
  }

  MPI_Finalize();

  return 0;
}
```



Exercise: Pairwise communication

- write, compile and run the code in the previous page with 2 processes

- modify the code in the previous page to perform the following communication:

  rank-0 sends an integer to rank-1

  rank-2 sends an integer to rank-3

  …

  rank-n sends an integer to rank-(n+1)

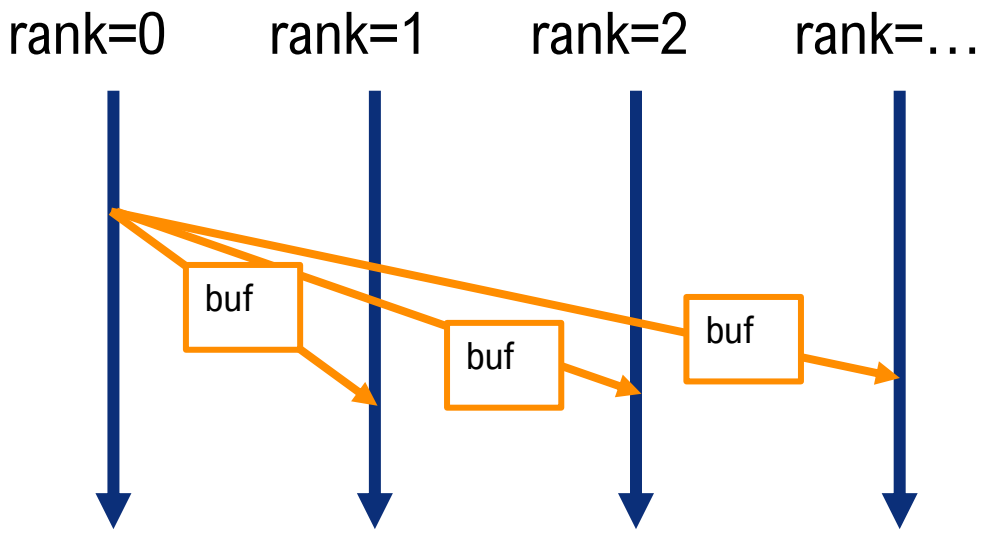
  and run the program with 10, 11, and 24 processes

MPI: Collective communication bcast

- Collective communication involves all processes in a communicator
  - we've already learned MPI_Barrier (MPI_COMM_WORLD), which is a kind of collective communication to make all processes synchronize
- MPI_Bcast broadcasts a buffer from a process to all processes

```
int MPI_Bcast(const void *buf, int count, MPI_Datatype datatype,
              int root, MPI_COMM comm);
```

  - **root** is the rank of broadcast root. All processes can be root



MPI: Collective communication reduction

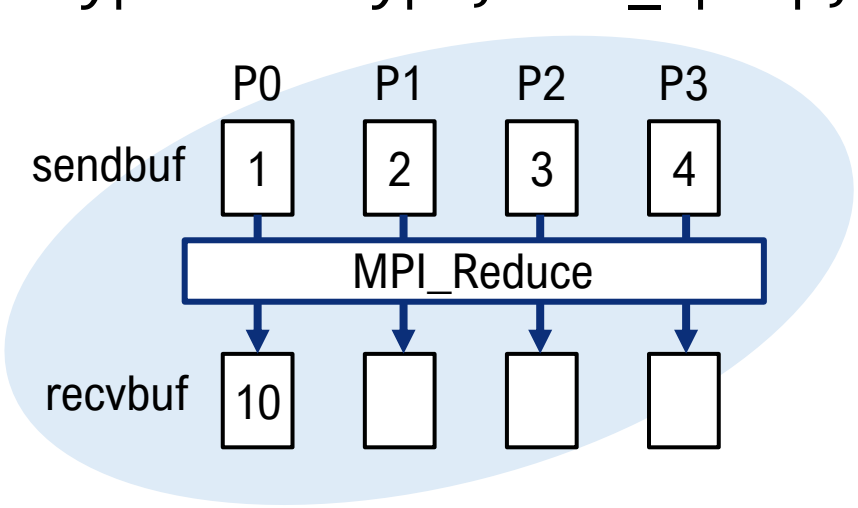
- MPI_Reduce reduces the values on all processes to a single value

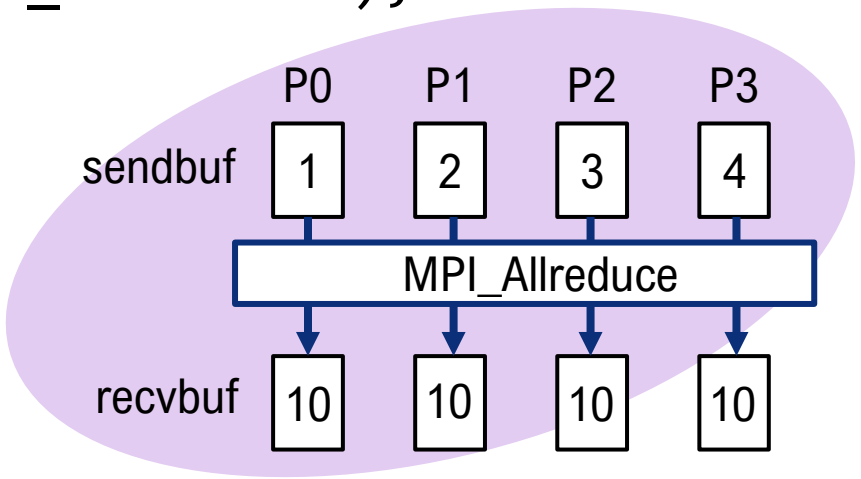
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_COMM comm);
```

  - **sendbuf**, **recvbuf**: address of send/recv buffer
  - **op**: reduce operation (MPI_SUM, MPI_MAX, MPI_MIN, etc)
  - **root**: rank of root process

- MPI_Allreduce reduce the values on all processors to a single value, and share the value among all processors

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_COMM comm);
```





Exercise: Collective communication

- Write, compile and run the code->
- Replace MPI_Reduce with MPI_Allreduce, and compare the results

```
#include<mpi.h>
int main(int argc, char **argv)
{
  int        send[2], recv[2];
  int        myrank;
  MPI_Status stat;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  send[0] = myrank*10+1;
  send[1] = myrank*20+1;

  printf("rank %d sends %d, and %d\n",myrank, send[0], send[1]);

  MPI_Reduce(send, recv, 2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  if(myrank==0){
    printf("recv[0] is %d, recv[1] is %d\n",recv[0],recv[1]);
  }

  MPI_Finalize();

  return 0;
}
```

Exercise:

- prepare an integer array of size 3
- substitute your favorite numbers to the array at rank-0
- share the favorite numbers to all processes
  1) by using MPI_Send/Recv
  2) by using MPI_Bcast
  3) by using MPI_Allreduce
- run the three program (all with 28 processors) and check if the numbers can be shared successfully

References:

- https://ja.wikipedia.org/wiki/フリンの分類
- High Performance Scientific Computing, Marsha J. Berger and Andreas Kloecker
- “Structured Parallel Programming: Patterns for Efficient Computation,” Michael McCool, Arch Robinson, James Reinders
- MPI and Hybrid Programming Models William Gropp
- https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf