High Performance Computing

高性能計算論

Volume 5

Cyberscience Center, Tohoku Univ

Hiroyuki Takizawa
<takizawa@tohoku.ac.jp>

Class Schedule

1-Oct	Introduction to HPC (1)
1-Oct	Introduction to HPC (2)
8-Oct	Parallel Architectures
8-Oct	How to use Supercomputer AOBA
15-Oct	Parallel Algorithm Design (1)
15-Oct	Parallel Algorithm Design (2)
5-Nov	MPI Programming (1)
5-Nov	MPI Programming (2)
12-Nov	OpenMP Programming (1)
12-Nov	OpenMP Programming (2)
26-Nov	Hybrid Programming
26-Nov	Performance Modeling and Analysis

Pre-recorded video

Today's Topic

- **Introduction to OpenMP Programming**
  - How to parallelize a for loop.
  - Shared data and private data.
  - Task-parallel processing
  - Offloading
- **Some keys to achieve better scalability**

What’s Parallel Computer (1/3)

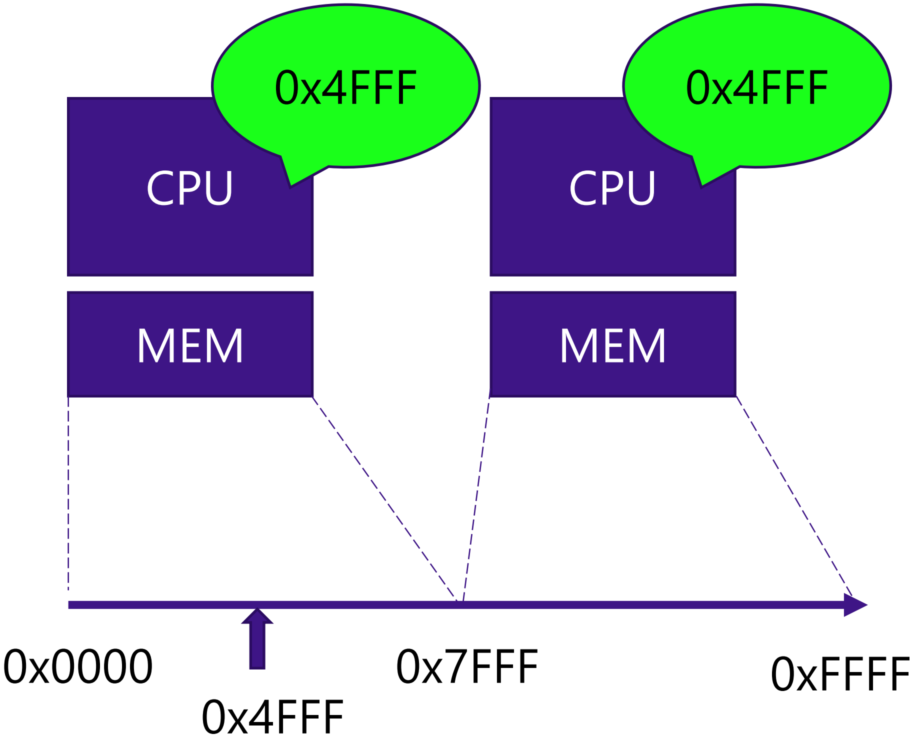
■ Parallel Computer

- A multi-processor computer system supporting parallel programming
- Two major categories of parallel computers
  - Distributed-memory parallel computers
    - Multiple computers and their interconnection network.
    - Employed to build a large-scale system
  - Shared-memory parallel computers
    - Symmetric multi-processor(SMP) and multicore/manycore.
    - Employed by most of current processors.

What’s Parallel Computer (2/3)

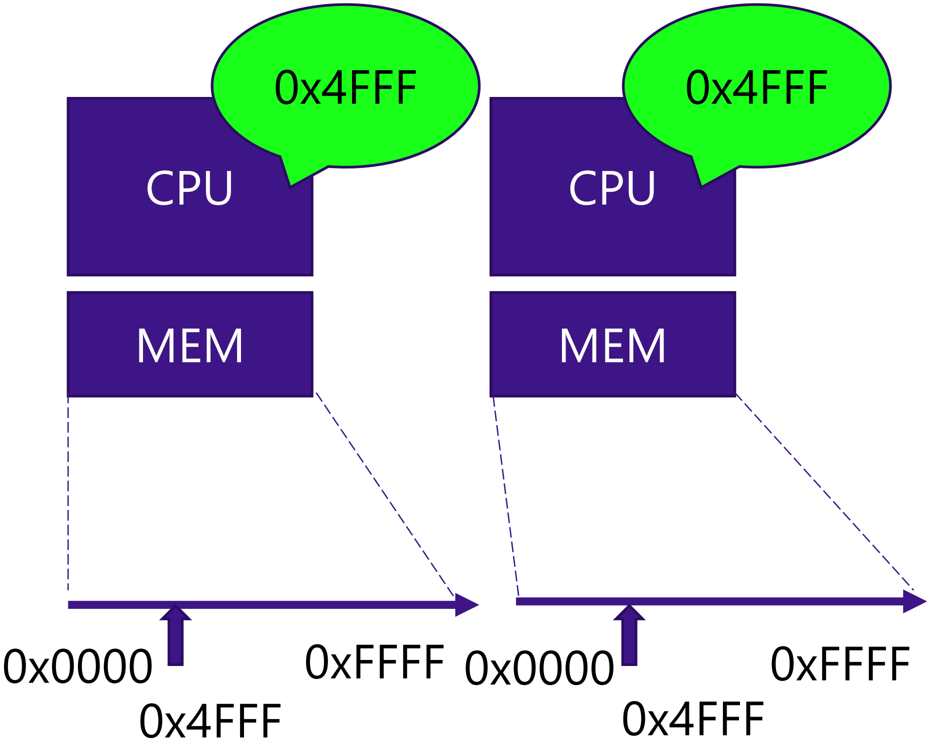
Shared-memory (NUMA)

Each memory device is mapped to a part of the memory space.



Distributed-memory

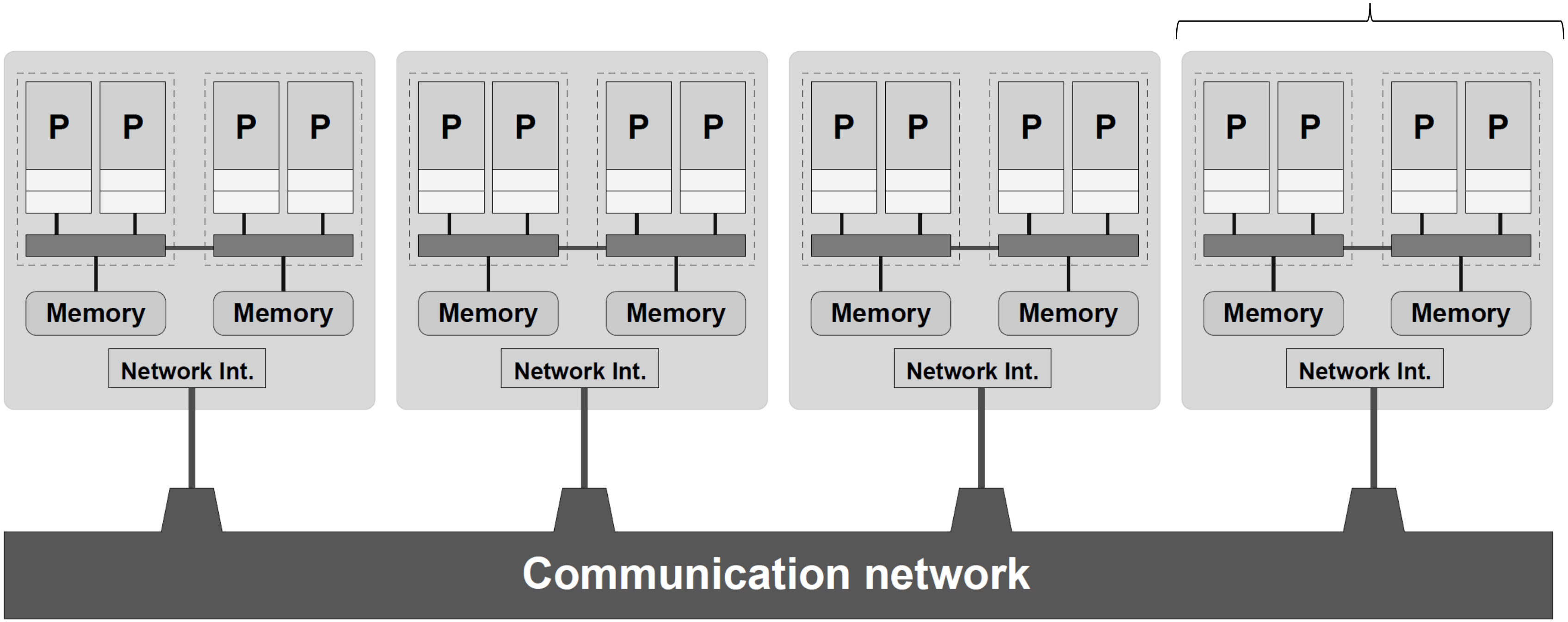
Each memory device has its own memory space.



What's Parallel Computer (3/3)

- Large-scale parallel computers
  = mixture of shared and distrib.-parallel.

One OS instance manages a node.



In addition, each node may have accelerators such as GPUs.

What’s Parallel Programming

Parallel Programming

- Programming in a language that allows you to explicitly indicate parallel portions of a program.
  - Those portions can run concurrently on different processors
- Why required?
  - Automatic parallelization is difficult, even though there have been many studies on parallelizing compilers.
- **OpenMP** and **MPI**
  - Standards for shared- and distributed-memory systems, respectively

Block in C/C++ Language

- In C/C++ language, a **block** is the code between **{** and **}**.

EX) body of a for loop.

```
for (i=0;i<10;i++)
  printf(”hello1¥n”);
printf(”hello2¥n”);
```

```
for (i=0;i<10;i++){
  printf(”hello1¥n”);
  printf(”hello2¥n”);
}
```

OpenMP is used to specify how to execute a block.

Processes and Threads

- **Both processes and threads are execution flows of a program.**
  - When a program is launched, OS reserves a set of some computing resources for the execution, a so-called **process**.
    - CPU time
    - Memory space
    - File descriptors
  - A **thread** is created inside a process
    - CPU time is assigned to each thread
    - The other resources are **shared** with the other threads of the process.

Compiler Directives

```
#include <stdio.h>

#define N 1000

int main(){

#ifdef DEBUG
  printf(“debug mode”);
#else
  printf(“normal mode”);
#endif
  return 0;
}
```

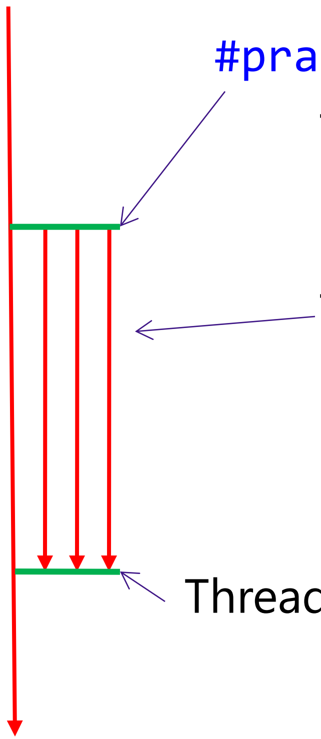
The line starting with # is not a statement, but a compiler directive to specify compiler’s behavior.

What’s OpenMP?

■ Threads are created/deleted on demand.

- Thread: an execution flow

Fork-join model



```
#pragma omp parallel
```

Threads are created to organize a team.

The block following the directive is multi-threaded.

Threads are deleted at the end of the block.

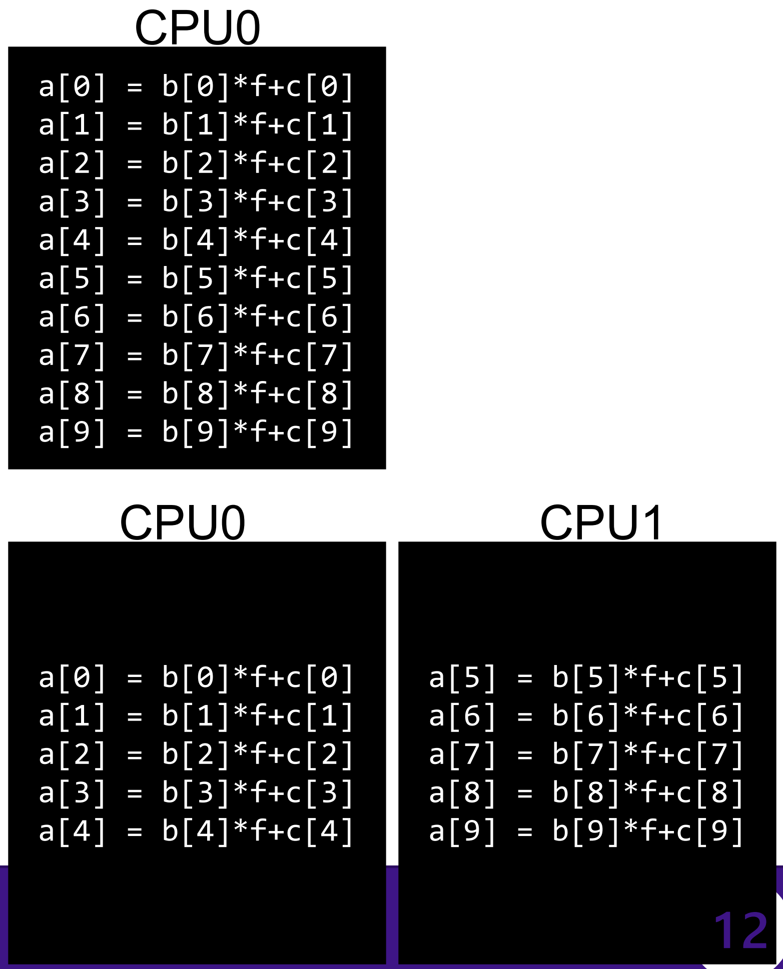
Data-Parallel Processing

- **Work-sharing by inserting directives into a sequential code.**
  - C code

```
for(i=0;i<10;i++)
    a[i] = b[i]*f + c[i];
```

  - OpenMP code

```
#pragma omp parallel
{
#pragma omp for
   for(i=0;i<10;i++)
    a[i] = b[i]*f + c[i];
}
```



CPU0

```
a[0] = b[0]*f+c[0]
a[1] = b[1]*f+c[1]
a[2] = b[2]*f+c[2]
a[3] = b[3]*f+c[3]
a[4] = b[4]*f+c[4]
a[5] = b[5]*f+c[5]
a[6] = b[6]*f+c[6]
a[7] = b[7]*f+c[7]
a[8] = b[8]*f+c[8]
a[9] = b[9]*f+c[9]
```

CPU0

```
a[0] = b[0]*f+c[0]
a[1] = b[1]*f+c[1]
a[2] = b[2]*f+c[2]
a[3] = b[3]*f+c[3]
a[4] = b[4]*f+c[4]
```

CPU1

```
a[5] = b[5]*f+c[5]
a[6] = b[6]*f+c[6]
a[7] = b[7]*f+c[7]
a[8] = b[8]*f+c[8]
a[9] = b[9]*f+c[9]
```

Data-Parallel Processing

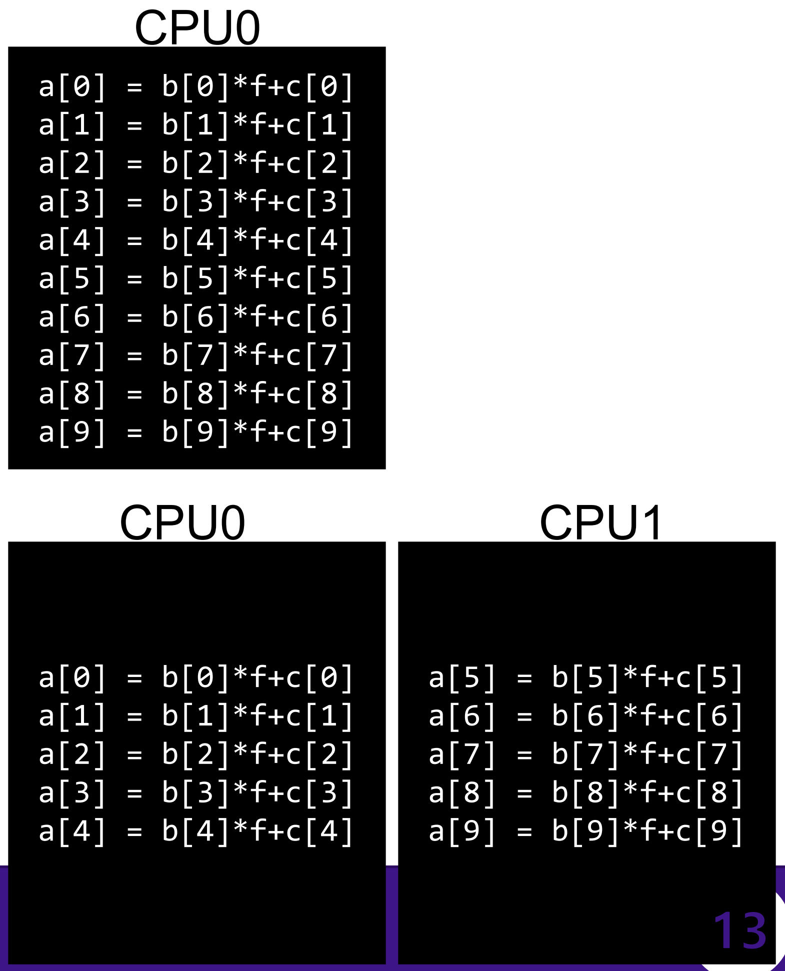
- **Work-sharing by inserting directives into a sequential code.**
  - C code

```
for(i=0;i<10;i++)
    a[i] = b[i]*f + c[i];
```

  - OpenMP code (simplified)

```
#pragma omp parallel for
   for(i=0;i<10;i++)
    a[i] = b[i]*f + c[i];
```

parallel and for can be written together in one line



CPU0

```
a[0] = b[0]*f+c[0]
a[1] = b[1]*f+c[1]
a[2] = b[2]*f+c[2]
a[3] = b[3]*f+c[3]
a[4] = b[4]*f+c[4]
a[5] = b[5]*f+c[5]
a[6] = b[6]*f+c[6]
a[7] = b[7]*f+c[7]
a[8] = b[8]*f+c[8]
a[9] = b[9]*f+c[9]
```

CPU0

```
a[0] = b[0]*f+c[0]
a[1] = b[1]*f+c[1]
a[2] = b[2]*f+c[2]
a[3] = b[3]*f+c[3]
a[4] = b[4]*f+c[4]
```

CPU1

```
a[5] = b[5]*f+c[5]
a[6] = b[6]*f+c[6]
a[7] = b[7]*f+c[7]
a[8] = b[8]*f+c[8]
a[9] = b[9]*f+c[9]
```

How many threads?

- **Set environment variable, OMP_NUM_THREADS, to configure the number of threads for running the program**
  - B shell
    - `export OMP_NUM_THREADS=4` ← On AOBA, use this one.
  - C shell
    - `setenv OMP_NUM_THREADS 4`

Compilation and Execution

Environment Variable OMP_NUM_THREADS

- specifies the number of threads launched in OpenMP
- EX) **`export OMP_NUM_THREADS=4`**
  - 4 threads are created for a team.

GNU C/C++ Compilers (ver. 4.2 or later)

- gcc/g++ options source

```
g++ -fopenmp sample.cc
```

[command] [options] [source code file]

Exercise: simple code

```
#include <stdio.h>
#include <unistd.h>
int main(int ac,char* av)
{
  int i;
  for(i=0;i<16;i++){
    sleep(1);
  }
  return 0;
}
```

```
  [How to run]
  % gcc hoge.c
  % time ./a.out
```

```
#include <stdio.h>
#include <unistd.h>
int main(int ac,char* av)
{
  int i;
  #pragma omp parallel for
  for(i=0;i<16;i++){
    sleep(1);
  }
  return 0;
}
```

```
[How to run]
% gcc -fopenmp hoge.c
% time ./a.out
```

Shared Variables

- **Threads share a single memory space.**
  - **All data are shared by threads by default.**

    The loop index variable is an exception (not shared).
    - EX) Finding the maximum in an array

```
max = 0;
#pragma omp parallel for
for(i=0;i<10;i++){
  if( max < a[i] ) max=a[i];
}
```

NOTE!

Race condition

Don't write a code like this.
Multiple threads may compete to write values to variable "max."

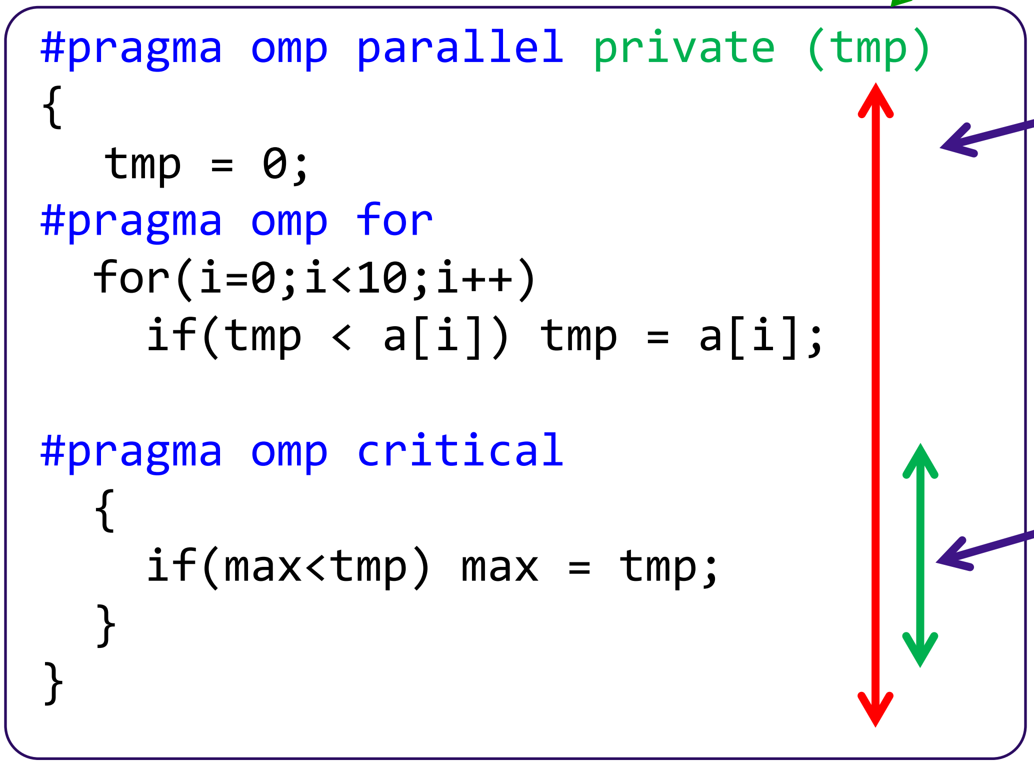
Private Variables and Critical Section

■ Threads share a single memory space.

- How can each thread hold a unique value?

  = private variables

clause: optional component to pragma



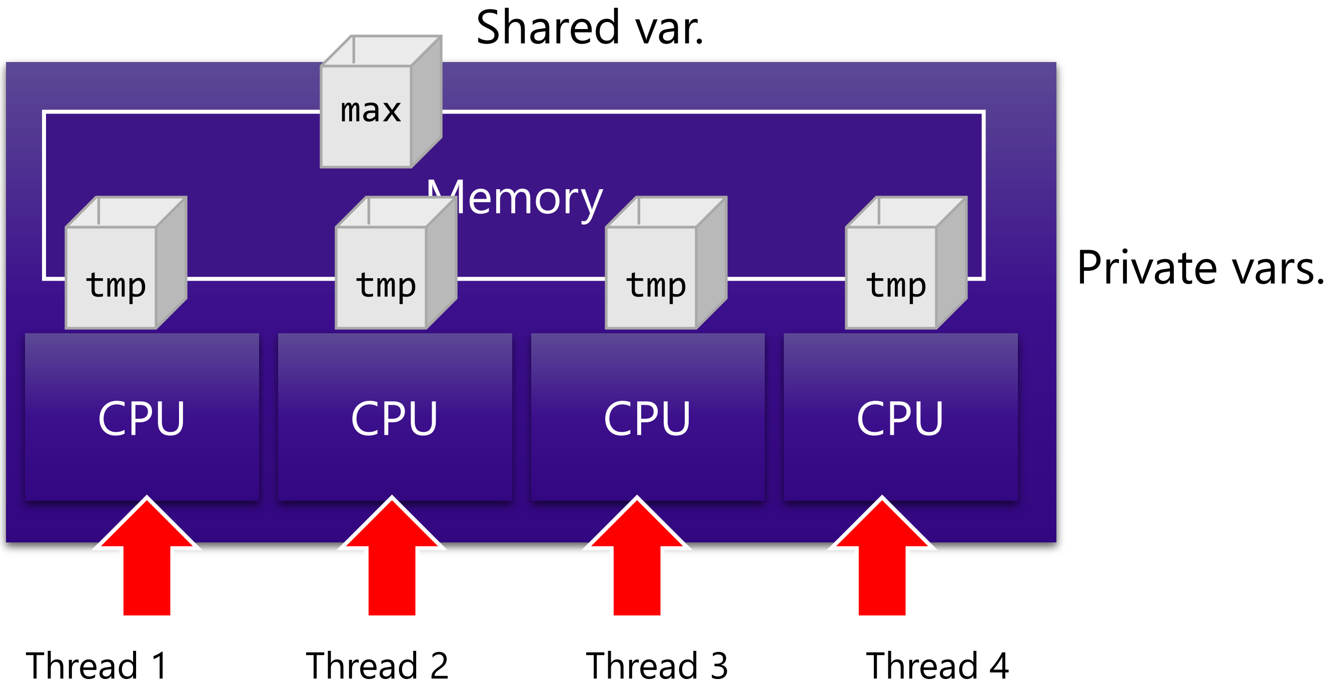
```
#pragma omp parallel private (tmp)
{
   tmp = 0;
#pragma omp for
  for(i=0;i<10;i++)
    if(tmp < a[i]) tmp = a[i];

#pragma omp critical
  {
    if(max<tmp) max = tmp;
  }
}
```

Multi-threaded
Each thread holds a unique value in tmp.
(tmp keeps the maximum value in a subarray)

Critical Section

Once a thread enters the section,
the others cannot enter it.
(See Next Slide)

Private Variables and Critical Section



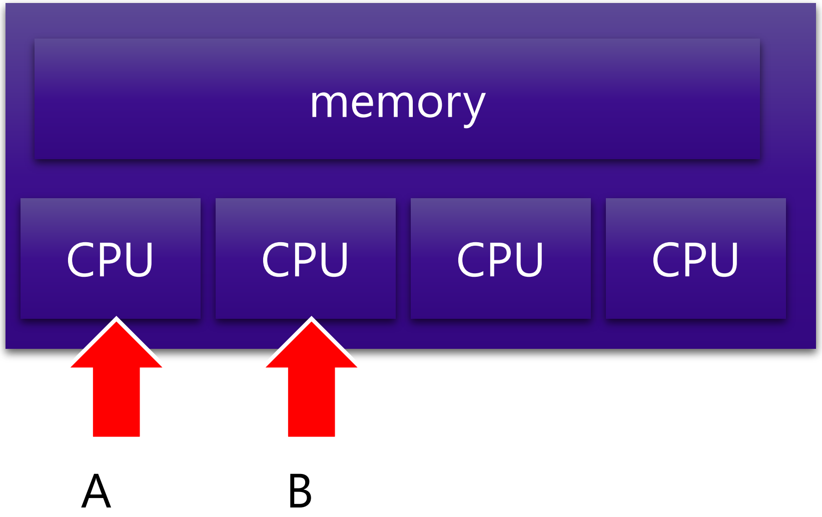
Even if one thread updates its own tmp, tmp in view from the others does not change.

If one thread updates max, max in view from the others also changes.
While one thread is updating max, we have to inhibit the others from accessing it.
⇒ #pragma omp critical

Sections

- **If there are multiple sections that can be done in parallel…**

```
#pragma omp sections
{
#pragma omp section
   {
      A
   }
#pragma omp section
   {
      B
   }
}
```



They are executed in parallel by different threads

Major OpenMP Directives

- **#pragma omp parallel**
  - directs parallelizing the following block
- **#pragma omp for**
  - directs work-sharing the following for loop
- **#pragma omp sections**
  - directs the following block is a set of parallel sections
- **#pragma omp section**
  - directs the following block is a section
- **#pragma omp critical**
  - directs the following block is a critical section
    - Just one thread can enter it

Clauses

shared (default)

- The variables are shared by threads

private/firstprivate/lastprivate/copyprivate

- The variables can have different values for different threads. The value can be copied from/to the outside of the block.

nowait

- Each thread does not wait for the others at the end of the block.

reduction

- Reduction operations are applied to the variables.

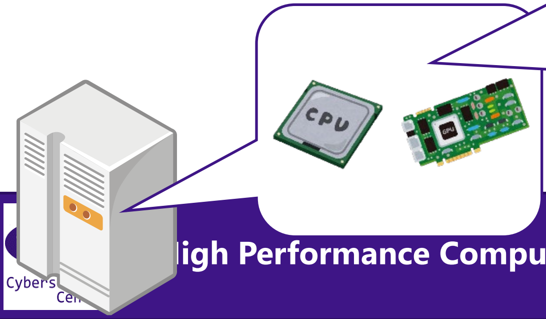
```
#pragma omp for reduction(+:x), private(j)
for(i=0;i<N;i++) {
  for(j=0;j<N;j++) {
    x+=j;
  }}
```

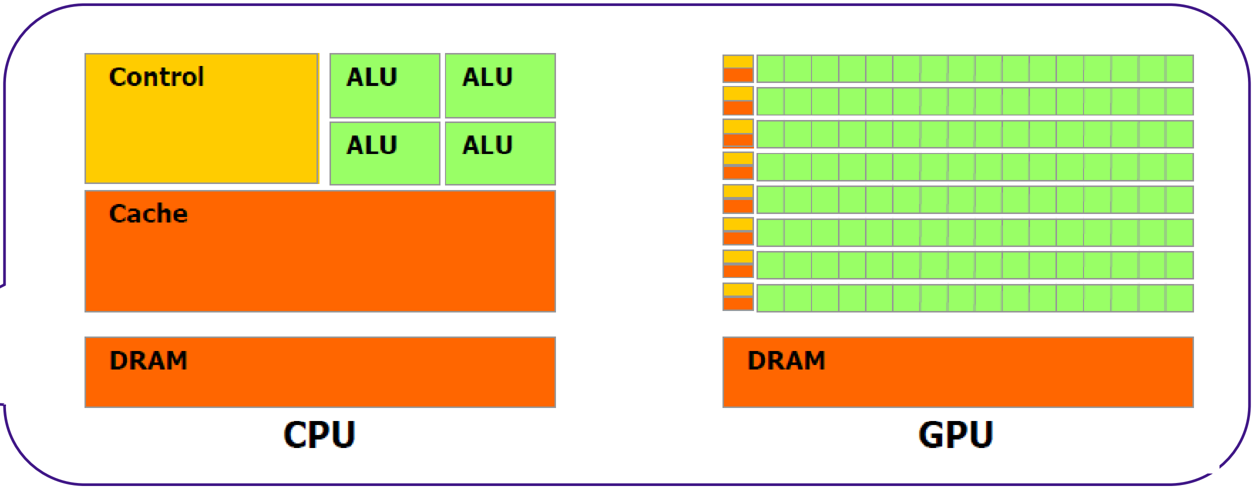
Homework: Confirm this code fragment works correctly.
(You need to add the prolog and epilog codes to make this work.)

Heterogenous Computing

■ CPU and GPU are very different processors.

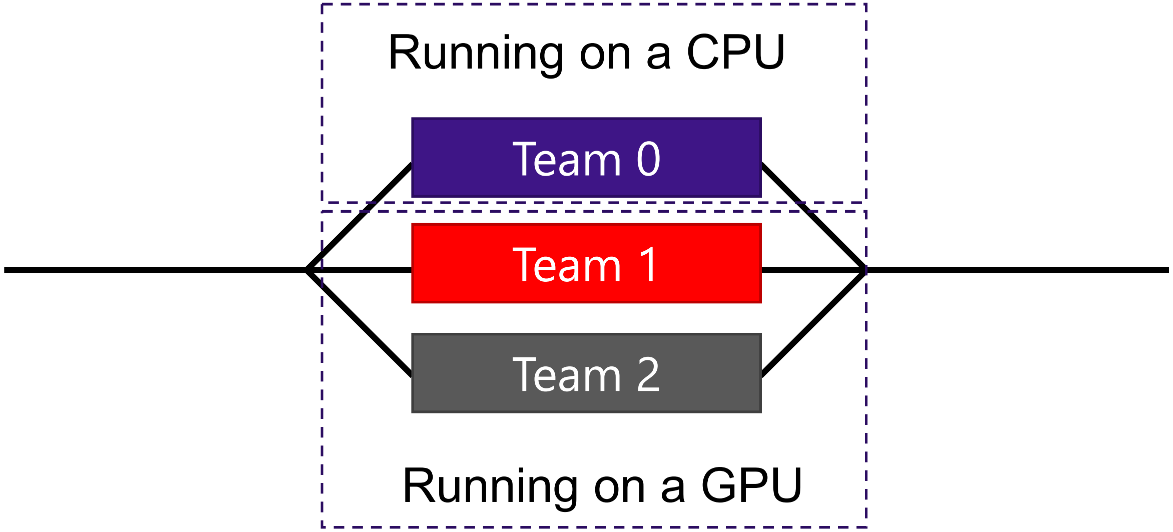
- Latency-oriented design (=speculative)
  - CPU has a large cache memory and control unit.
- Throughput-oriented design (=parallel)
  - GPUs devote more hardware resources to ALUs.





GPU Programming with OpenMP

- A task can be executed on a different processor



Running on a CPU

Team 0

Team 1

Team 2

Running on a GPU

```
#pragma omp target map (alloc:Anew) map(tofrom:A)
{
#pragma omp teams distribute
  for(j=0;j<M;j++){
    #pragma omp parallel for
    for(i=;i<N;i++){
      /* time-consuming data-parallel computation */
    }}}
```

Target Directives

- **#pragma omp target**
- **#pragma omp target data**

Defining only data mapping

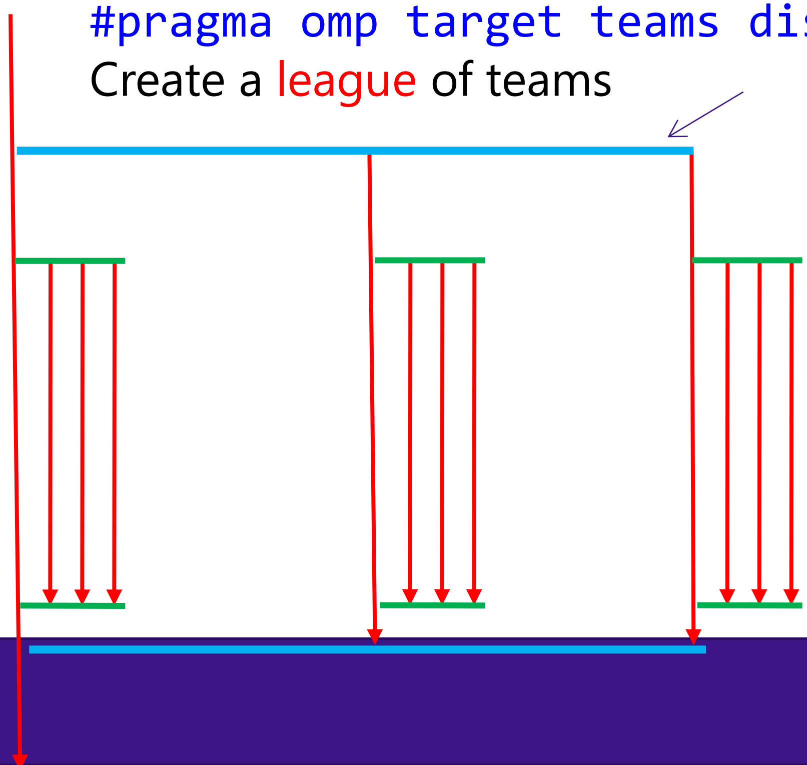
  - New features of OpenMP 4.0 or later
  - The block specified by the target directive can be **offloaded to another processor such as GPU**.
  - A map clause is used to send/retrieve data to/from the GPU.

```
/* Aray Anew is created, and A is transferred from/to GPU */
#pragma omp target map(alloc:Anew) map(tofrom:A)
{
#pragma omp teams distribute parallel for
  for(i=;i<N;i++){
    /* time-consuming data-parallel computation */
  }
}
```

This part is executed on GPU

Target Directives (Cont’d)

- `#pragma omp teams distribute`
  - By definition, all threads in a team have to be synchronizable.
  - A many-core processor needs to run many threads, but the synchronization overhead increases with the number of participating threads.



`#pragma omp target teams distribute`

Create a league of teams

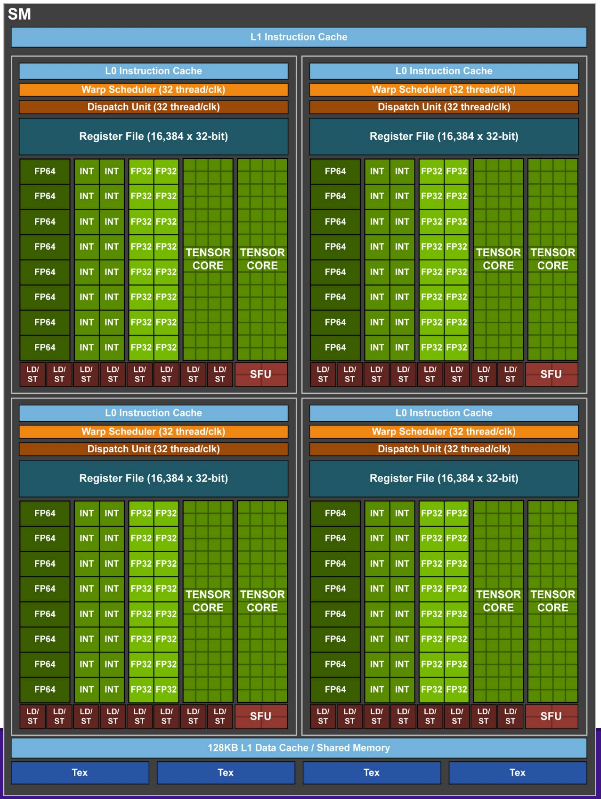
`#pragma omp parallel`

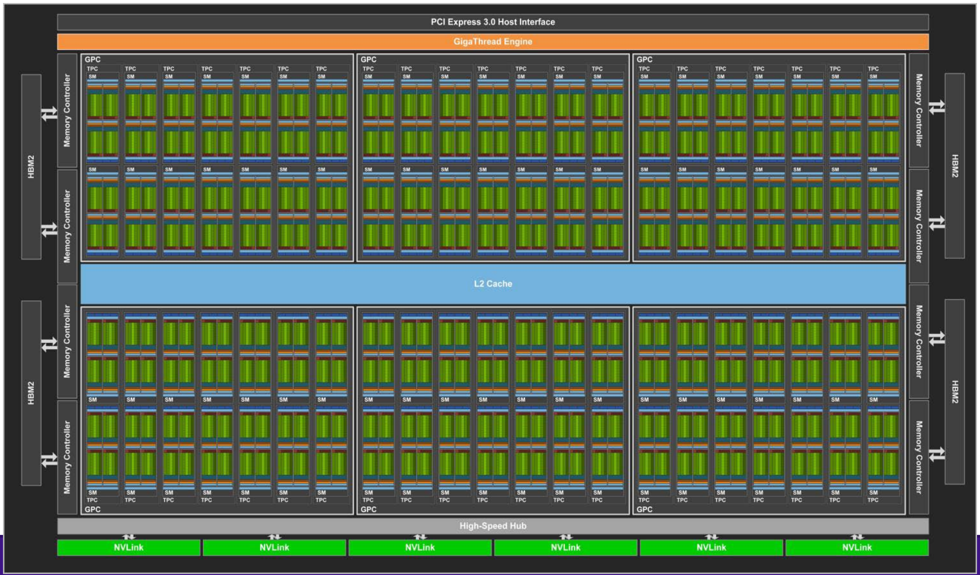
Creates a team of (synchronizable) threads.

GPU Architecture (Volta)

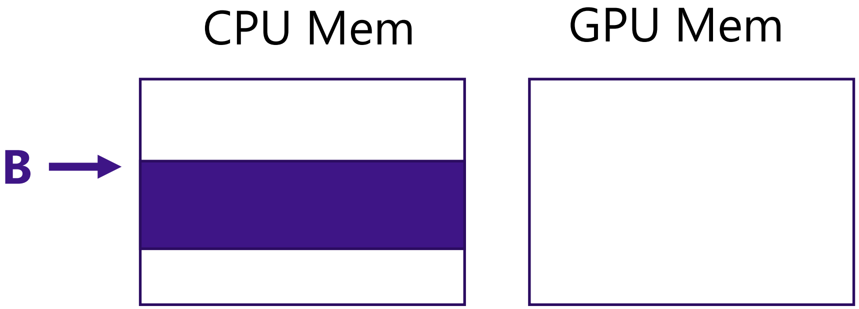
Processor Array

- executing many parallel threads efficiently by using many simple cores (CUDA cores)
- organizing streaming multiprocessors (SMs) each consisting of …
    - 64 CUDA cores (INT+FP32)
    - 8 Tensor cores (4x4 matmul and acc)
    - Instruction and constant caches


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex



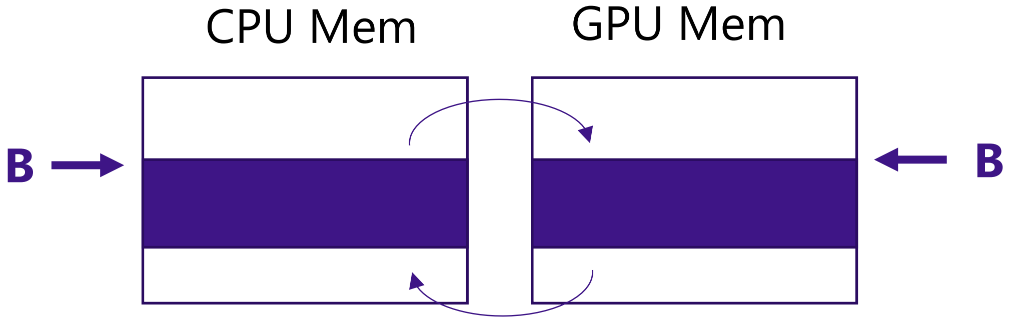
Mapping Data to GPU Mem



```
int N = 1000;

int main()
{
  int A = 0;
  int* B = (int*)malloc(sizeof(int)*N);
  #pragma omp target
  #pragma omp teams distribute parallel for
  for(int i = 0;  i< N; i++){
    // A, B, N, and i exit here
    // B is a pointer – the data B point to DOES NOT exist here
  }
}
```

Mapping Data to GPU Mem

```
int N = 1000;

int main()
{
 int A = 0;
 int* B = (int*)malloc(sizeof(int)*N);
 #pragma omp target map(tofrom:B[0:N])
 #pragma omp teams distribute parallel for
 for(int i = 0;  i< N; i++){
   // A, B, N, and i exit here
   // B is a pointer – the data B point to DOES exist here
 }
}
```

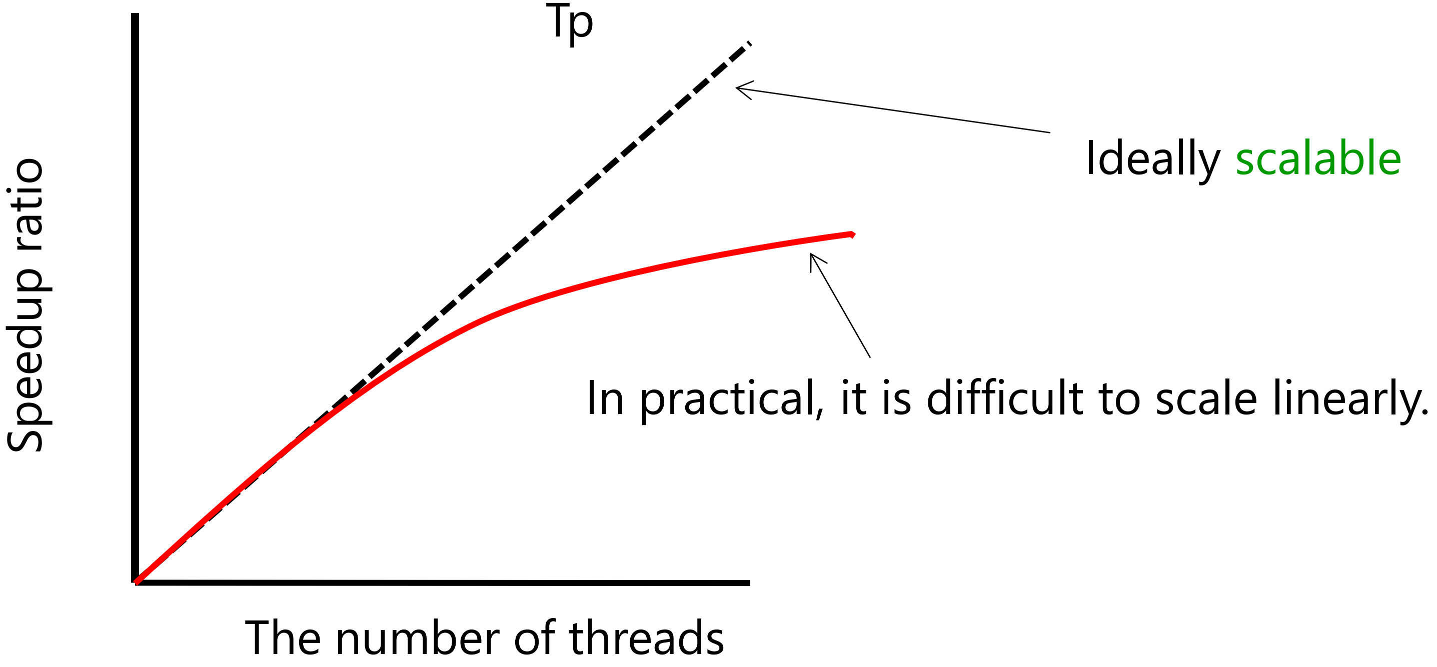


Copy to/from GPU mem

Parallelization Scalability

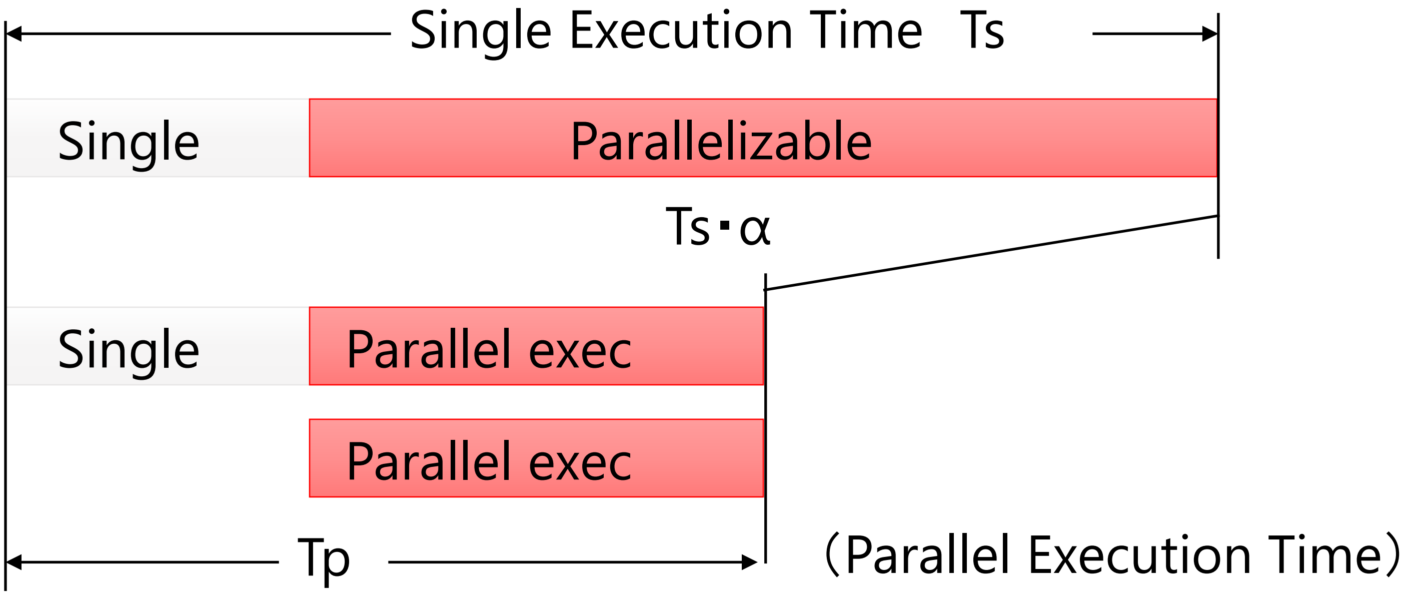
- **Speedup ratio: the ratio of parallel exec. to sequential exec.**
  - How many times does it become faster?

$$\text{Speedup ratio} = \frac{Ts}{Tp}$$



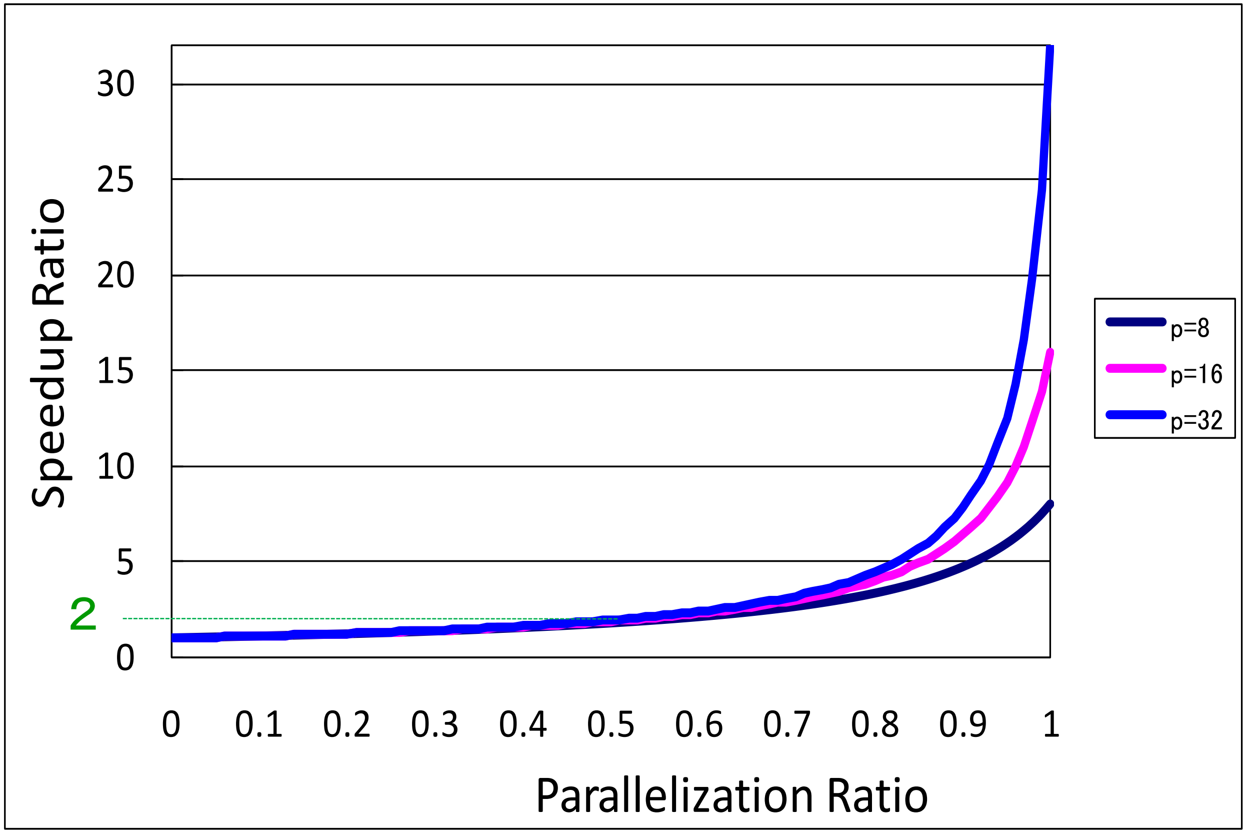
Parallelization Ratio

Parallelization Ratio α： The ratio of parallel exec.



$$\text{Speedup ratio} = \frac{Ts}{Tp} = \frac{1}{1 - \alpha + \alpha/n}$$ (Amdahl’s law)

Amdahl’s Law



In the case of α=0.5 (50%), the speedup ratio does not exceed 2.

＝ It is very important to select an algorithm with high parallelism.

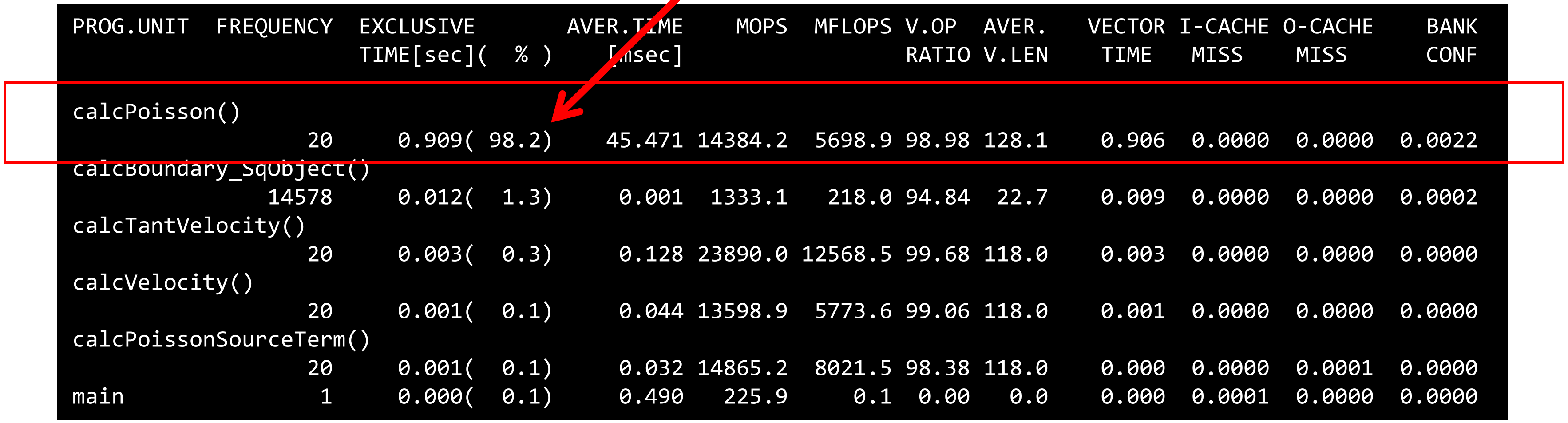
What part should be accelerated?

- Pareto’s laws (aka 80-20 rule)
- A small part of the code consumes a large part of the execution time.

■ Accelerate a time-consuming part!

- Significant reduction in the execution time.

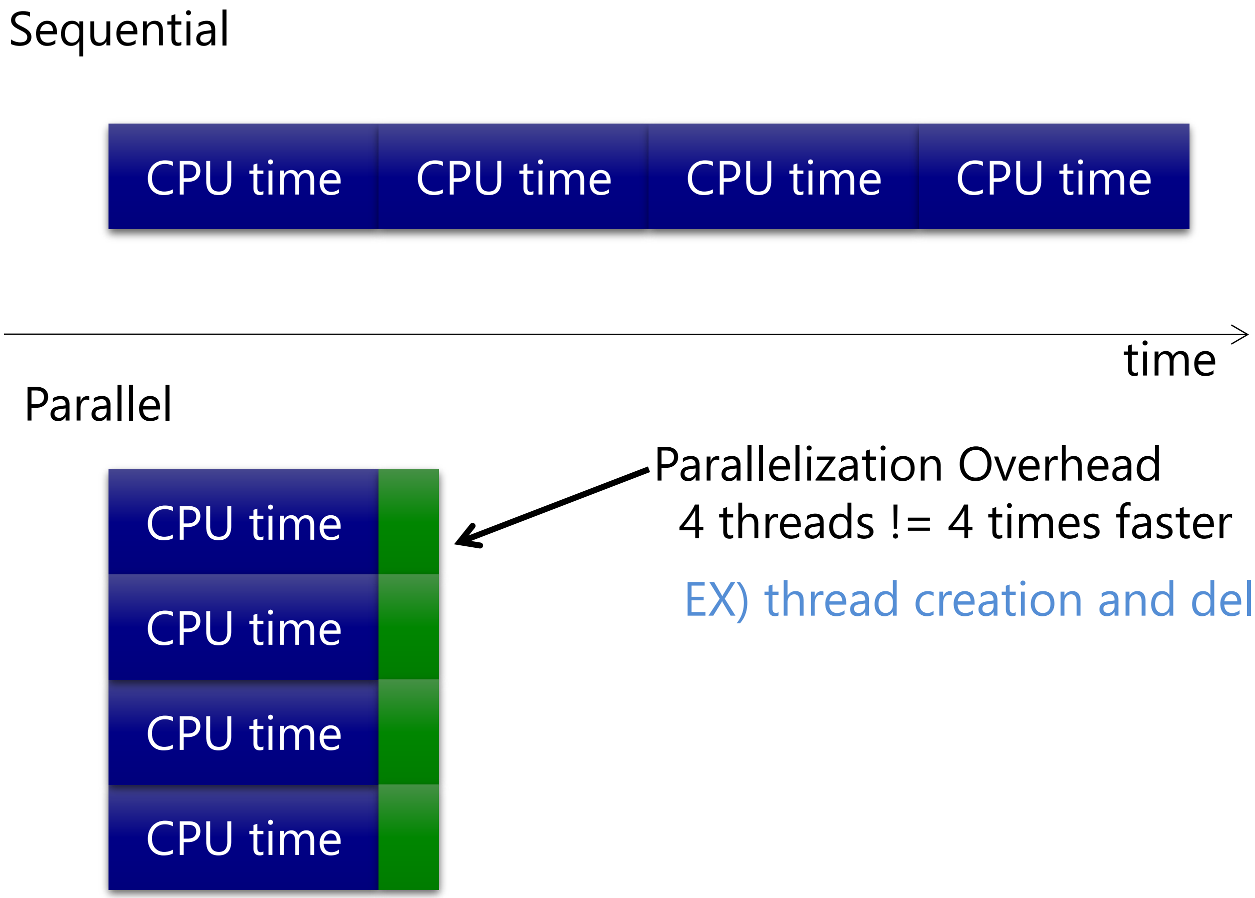
98% of exec time consumed.



```
PROG.UNIT   FREQUENCY   EXCLUSIVE        AVER.TIME     MOPS   MFLOPS V.OP   AVER.    VECTOR I-CACHE O-CACHE      BANK
                        TIME[sec](   % )    [msec]                   RATIO  V.LEN     TIME    MISS     MISS       CONF

calcPoisson()
                   20      0.909( 98.2)    45.471 14384.2   5698.9 98.98 128.1     0.906  0.0000  0.0000   0.0022
calcBoundary_SqObject()
                14578      0.012(  1.3)     0.001  1333.1    218.0 94.84  22.7     0.009  0.0000  0.0000   0.0002
calcTantVelocity()
                   20      0.003(  0.3)     0.128 23890.0 12568.5 99.68 118.0     0.003  0.0000  0.0000   0.0000
calcVelocity()
                   20      0.001(  0.1)     0.044 13598.9   5773.6 99.06 118.0     0.001  0.0000  0.0000   0.0000
calcPoissonSourceTerm()
                   20      0.001(  0.1)     0.032 14865.2   8021.5 98.38 118.0     0.000  0.0000  0.0001   0.0000
main                1      0.000(  0.1)     0.490   225.9      0.1  0.00   0.0     0.000  0.0001  0.0000   0.0000
```

The performance info of a program compiled with –pg can be printed out by using the **gprof** command on a standard Linux system (e.g., AOBA-B).

Parallelization Overhead

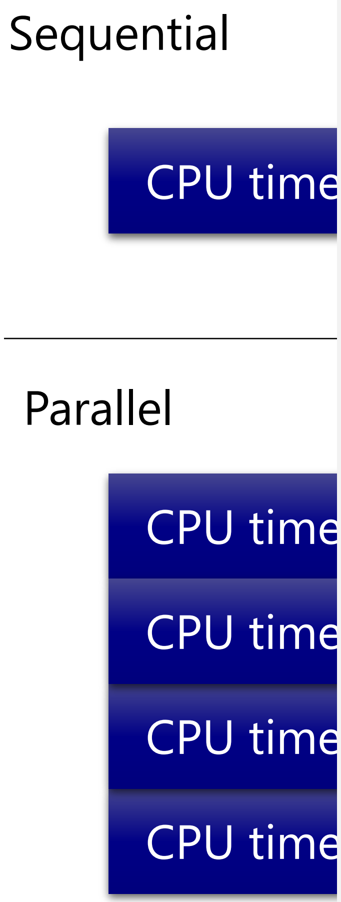


Parallelization Overhead

4 threads != 4 times faster

EX) thread creation and deletion

Parallelization Overhead



```
#pragma omp parallel for
for (int i=0; ···) {}
#pragma omp parallel for
for (int i=0; ···) {}
```

Twice

```
#pragma omp parallel
{
 #pragma omp for
 for (int i=0; ···) {}
#pragma omp for
 for (int i=0; ···) {}
}
```

Once

Summary

■ Introduction to OpenMP Programming

- Data-parallel processing (work-sharing)
    - How to parallelize a for loop.
- Critical section
    - Shared variables and private variables.
- Task-parallel processing
    - Sections
    - Tasks
- Offloading
    - Target

■ Some keys to achieve better scalability

- Amdahl’s law
- Pareto’s law