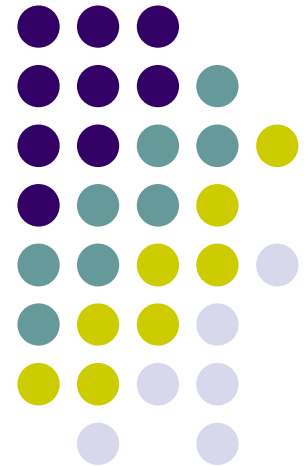


# 2021 Practical Parallel Computing (実践的並列コンピューティング) No. 2

Introduction (2)  
Apr 15, 2021

Toshio Endo  
School of Computing & GSIC  
[endo@is.titech.ac.jp](mailto:endo@is.titech.ac.jp)





# Overview of This Course

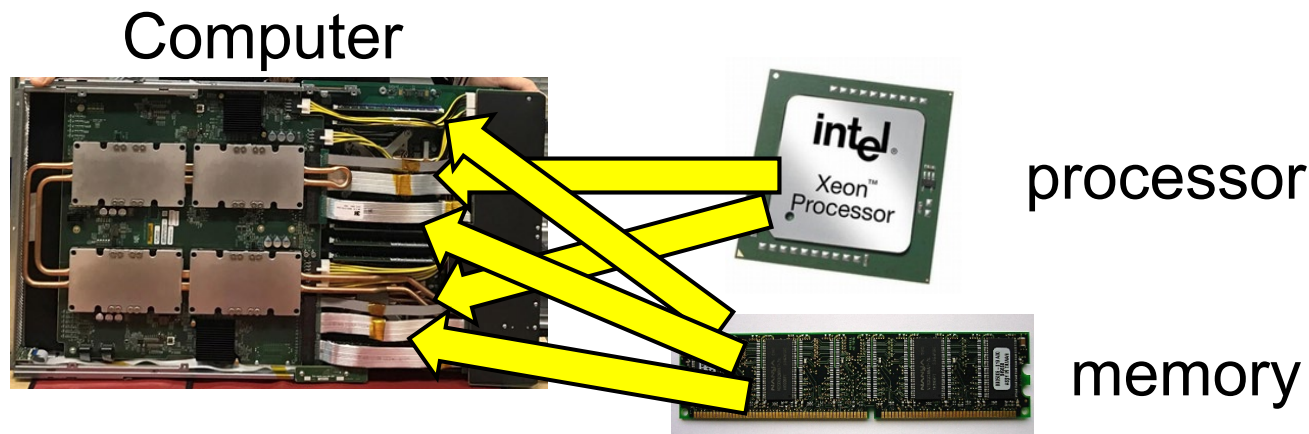
- Part 0: Introduction
  - 2 classes ← We are here (2/2)
- Part 1: **OpenMP** for shared memory programming
  - 4 classes
- Part 2: **GPU** programming
  - OpenACC and CUDA
  - 4 classes
- Part 3: **MPI** for distributed memory programming
  - 3 classes

# Different Parallel Programming Methods



- Why do we learn several programming methods?
  - OpenMP, OpenACC/CUDA, MPI in this lecture

Reason: Programming methods depend on **structure of computer hardware** (or **computer architecture**) we will use



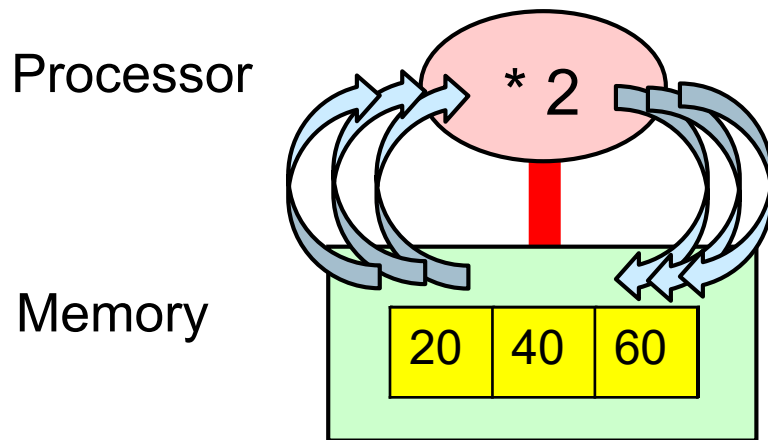


# Software Runs on Hardware

- Software = Algorithm + Data
- Hardware (architecture)  $\doteq$  Processor + Memory

*Note: This is so simplified discussion*

## Hardware



## Software Example

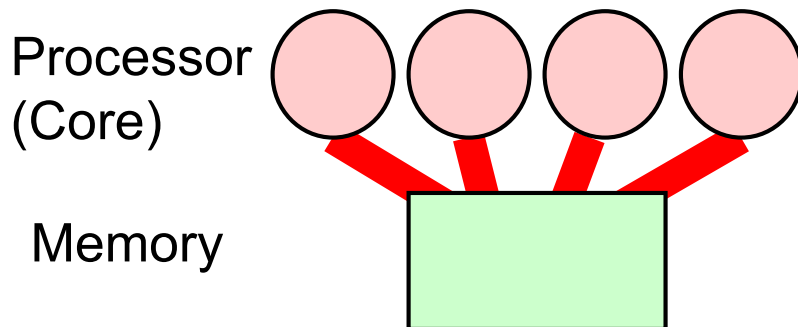
```
int a[3] = {10, 20, 30};  
int i;  
  
for (i = 0; i < 3; i++) {  
    a[i] = a[i] * 2;  
}
```



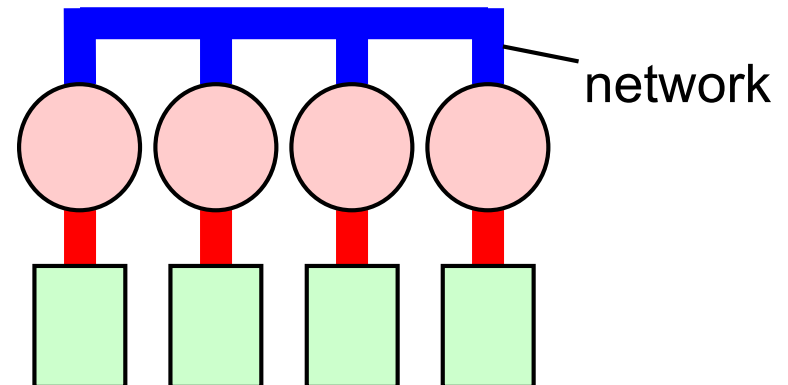
# What is Parallel Architecture?

- Parallel architecture has MULTIPLE components
- Two basic types:

**Shared** memory  
parallel architecture



**Distributed** memory  
parallel architecture



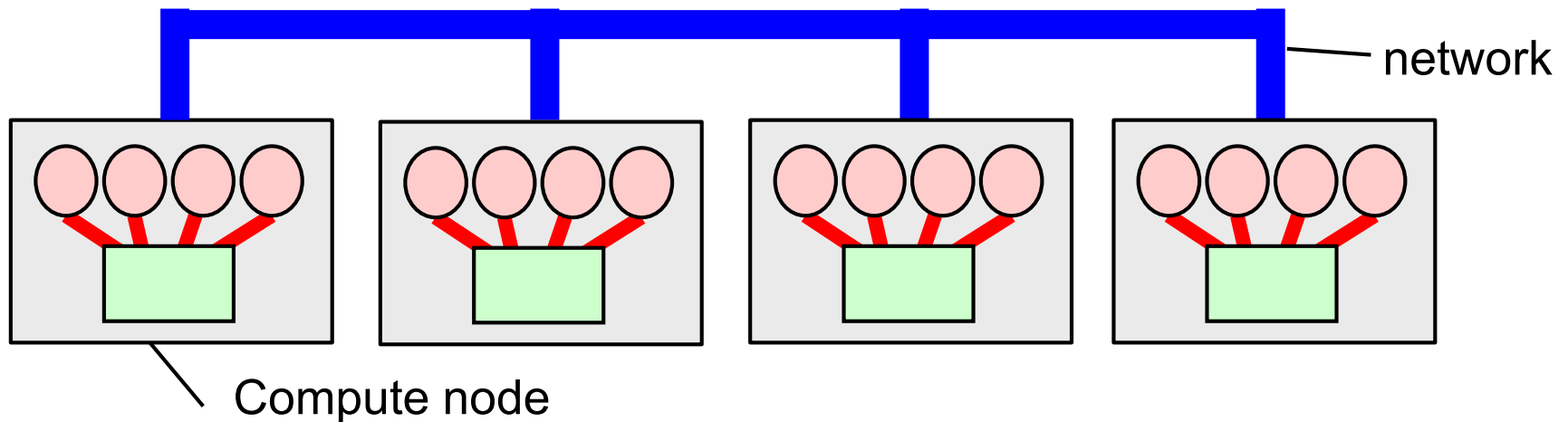
- Different programming methods are used for different architecture

# Modern SCs use Both!



Modern SCs are combination of “shared” and “distributed”  
“shared memory” in a node

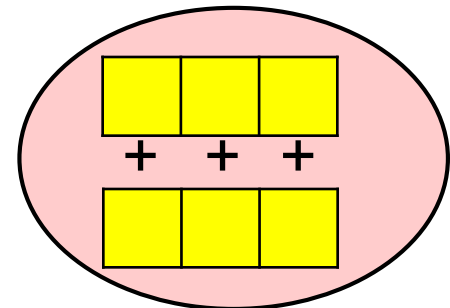
“distributed memory” among nodes, connected by network



✂ Moreover, each processor (core) may have *SIMD parallelism*, such as SSE, AVX...

A processor (core) can do several computations at once

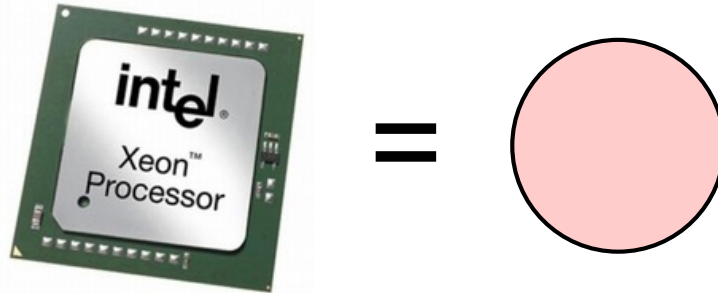
*SIMD is out of scope of this class*





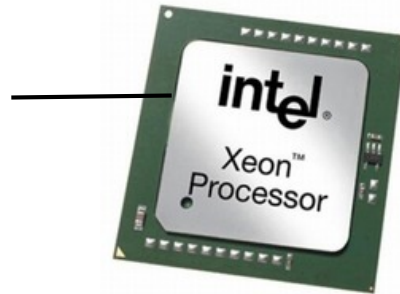
# (Confusing) Terminology

- In old days, definition of “processor” was simple

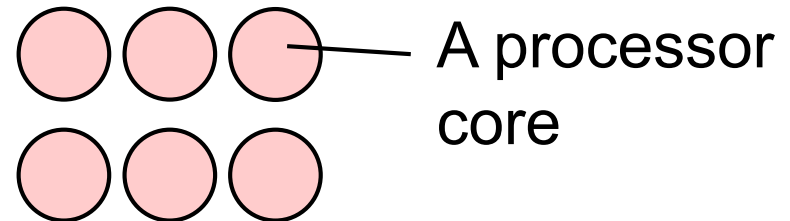


- Since around 2005, “multicore processor” became popular

A processor package



=



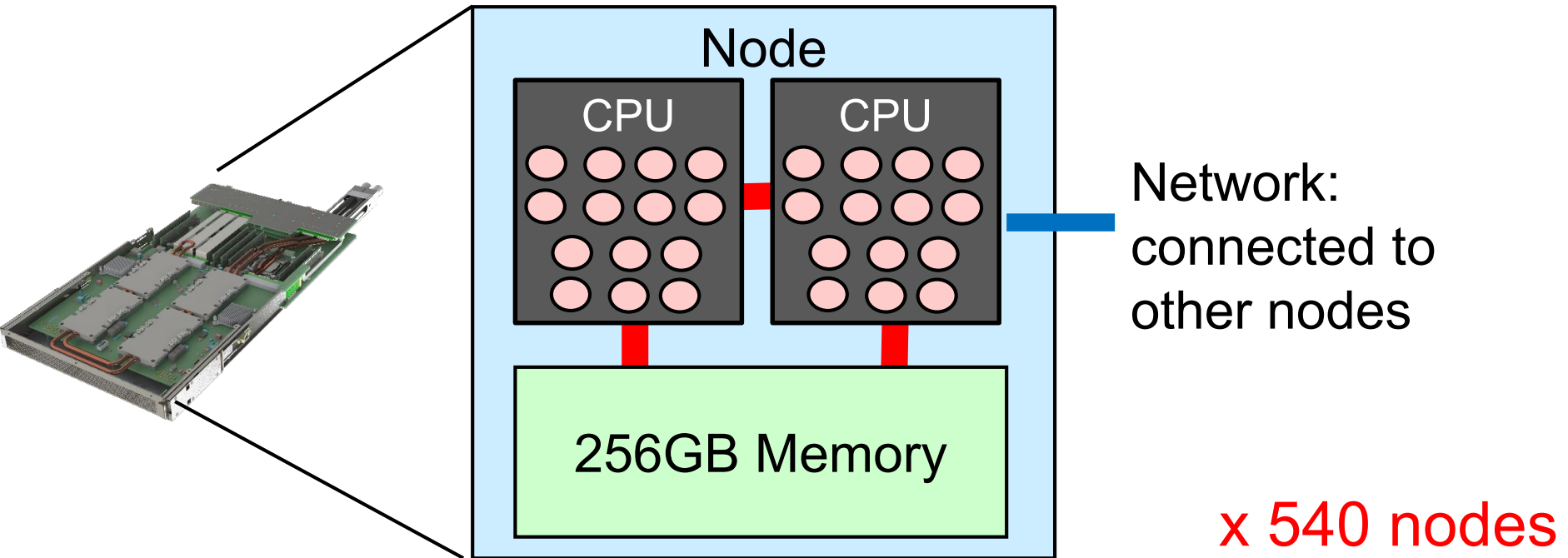
A processor core

✂ *Hyperthreading* makes discussion more complex, but skipped here



# A TSUBAME3 Node (1)

- 2 processor packages (CPU) × 14 cores  
→ A TSUBAME3 node has **28 cores**



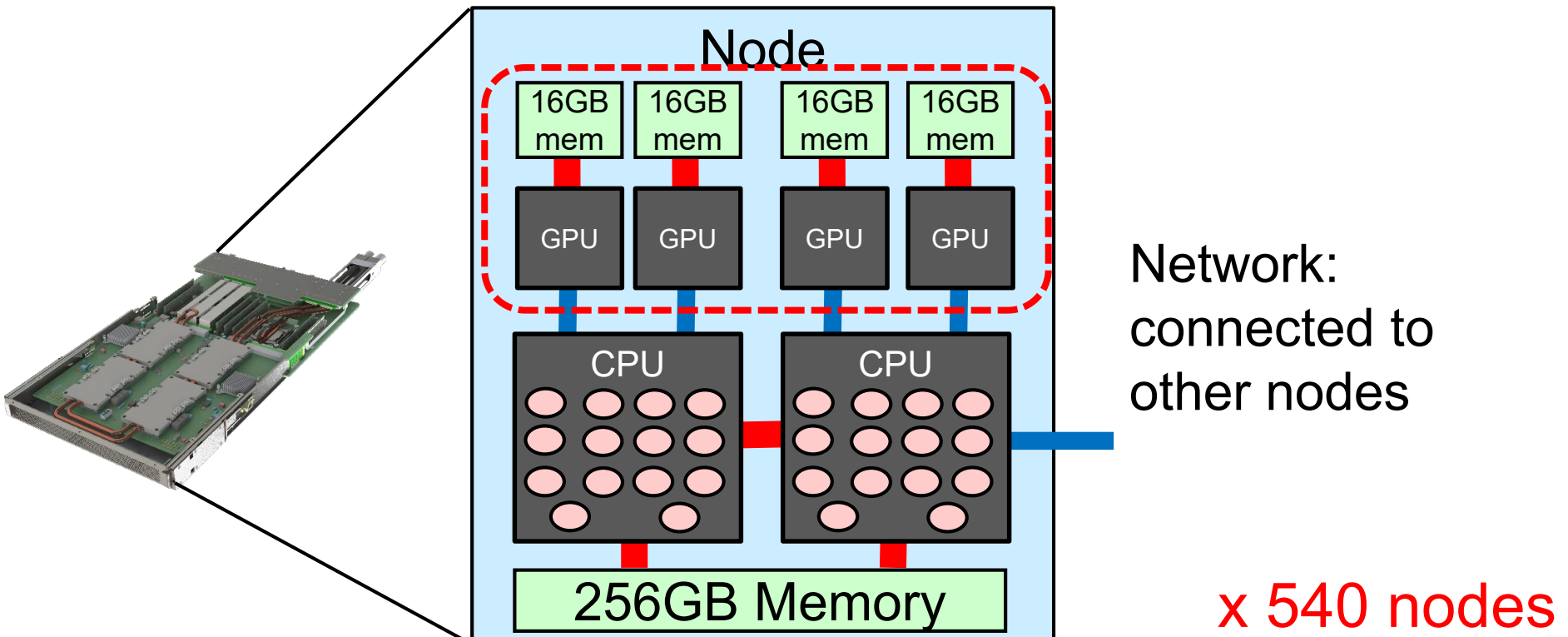
- GPUs are (still) omitted in this figure





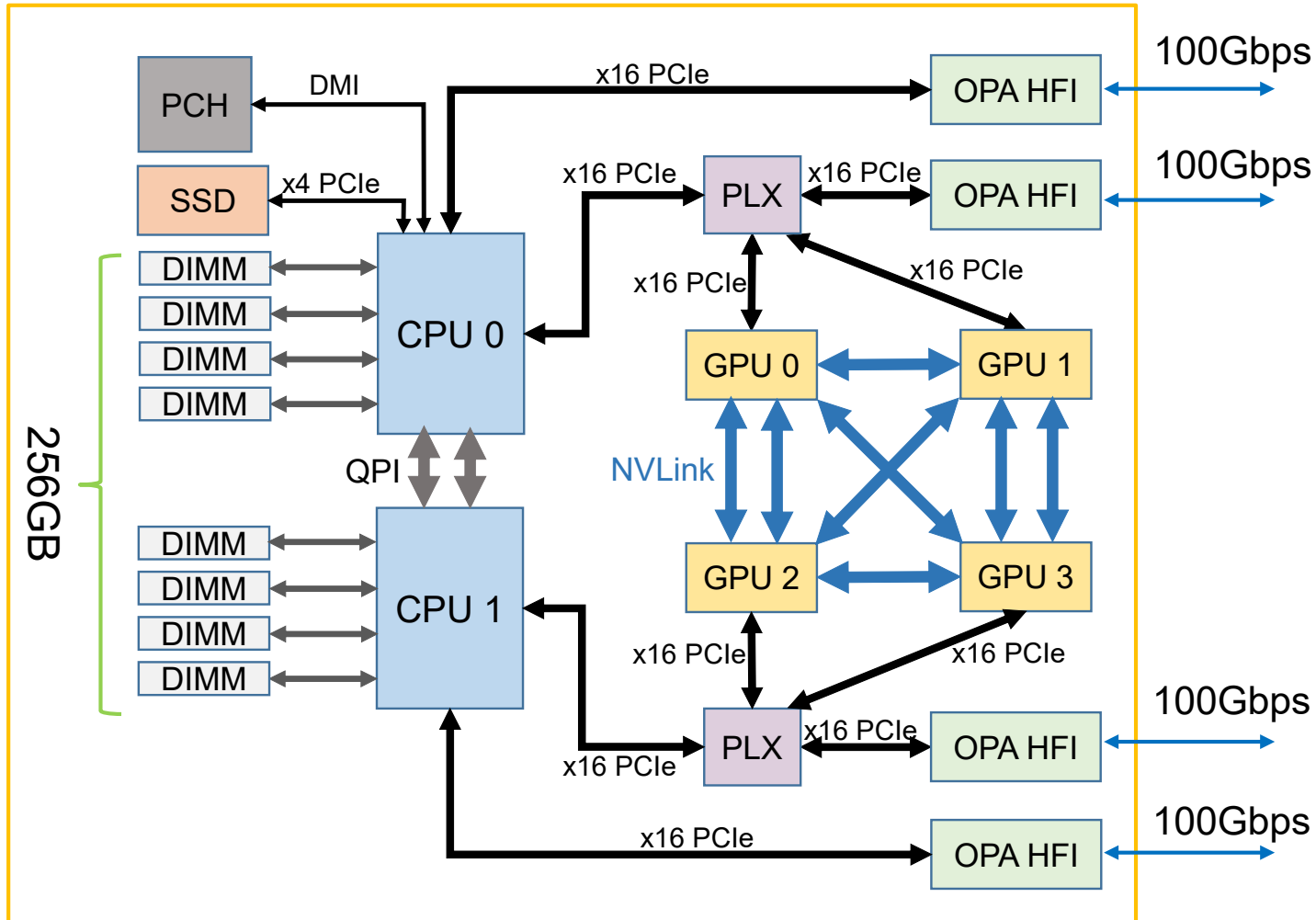
# A TSUBAME3 Node (2)

- A node has 2 CPUs + 4 GPUs
  - Each GPU (Tesla P100) has 3,584 cores

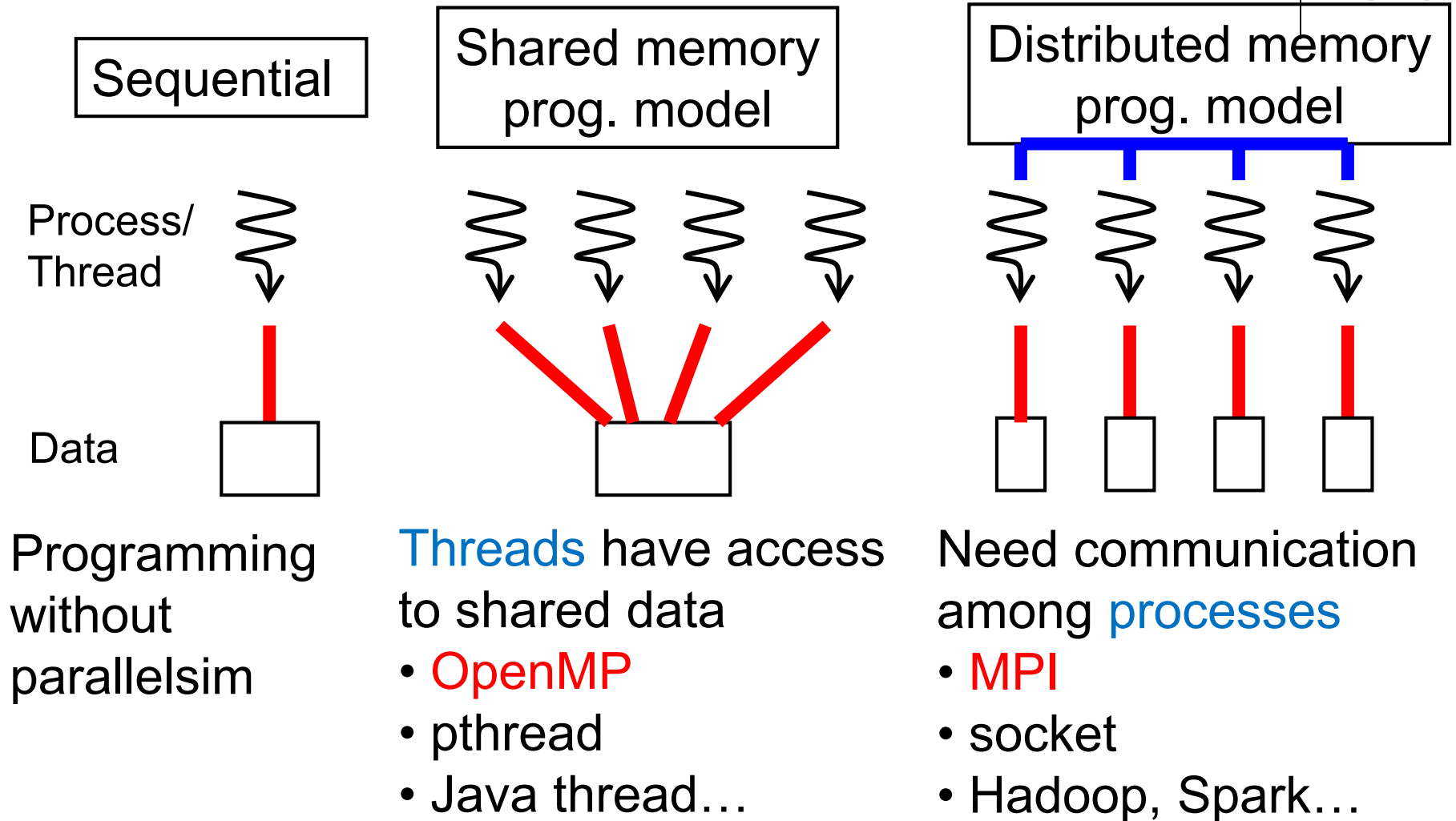
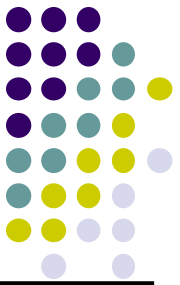




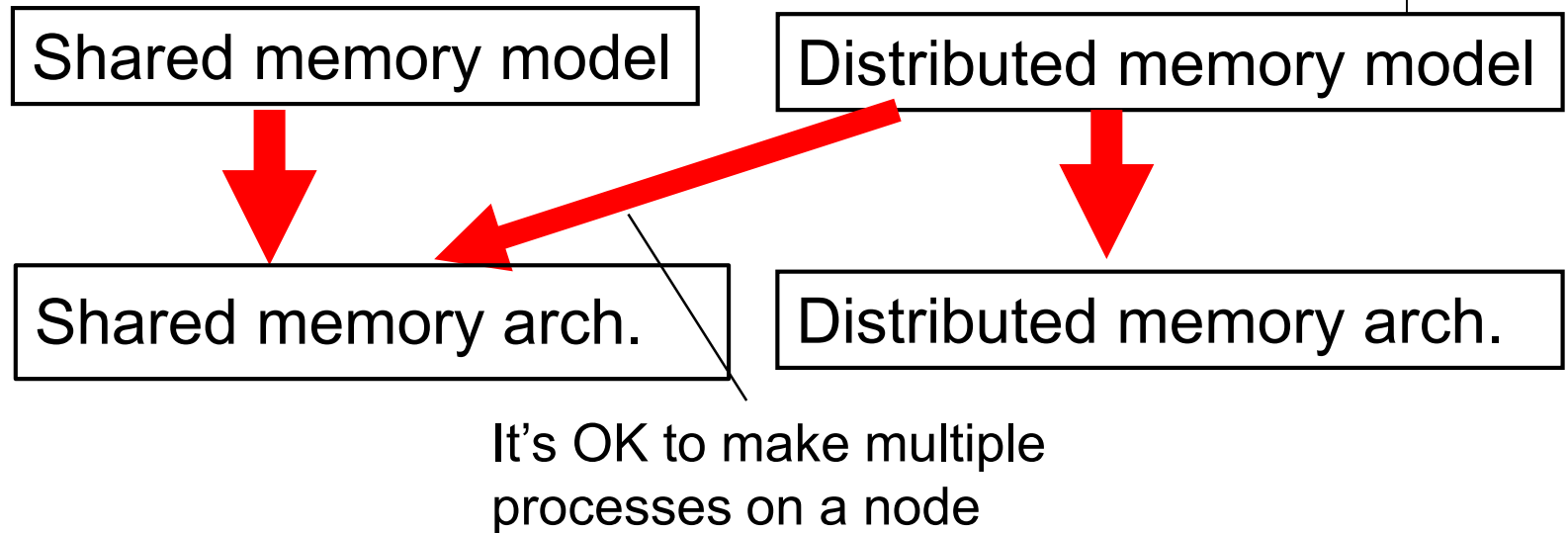
# A TSUBAME3 Node in More Detail



# Classification of Parallel Programming Models

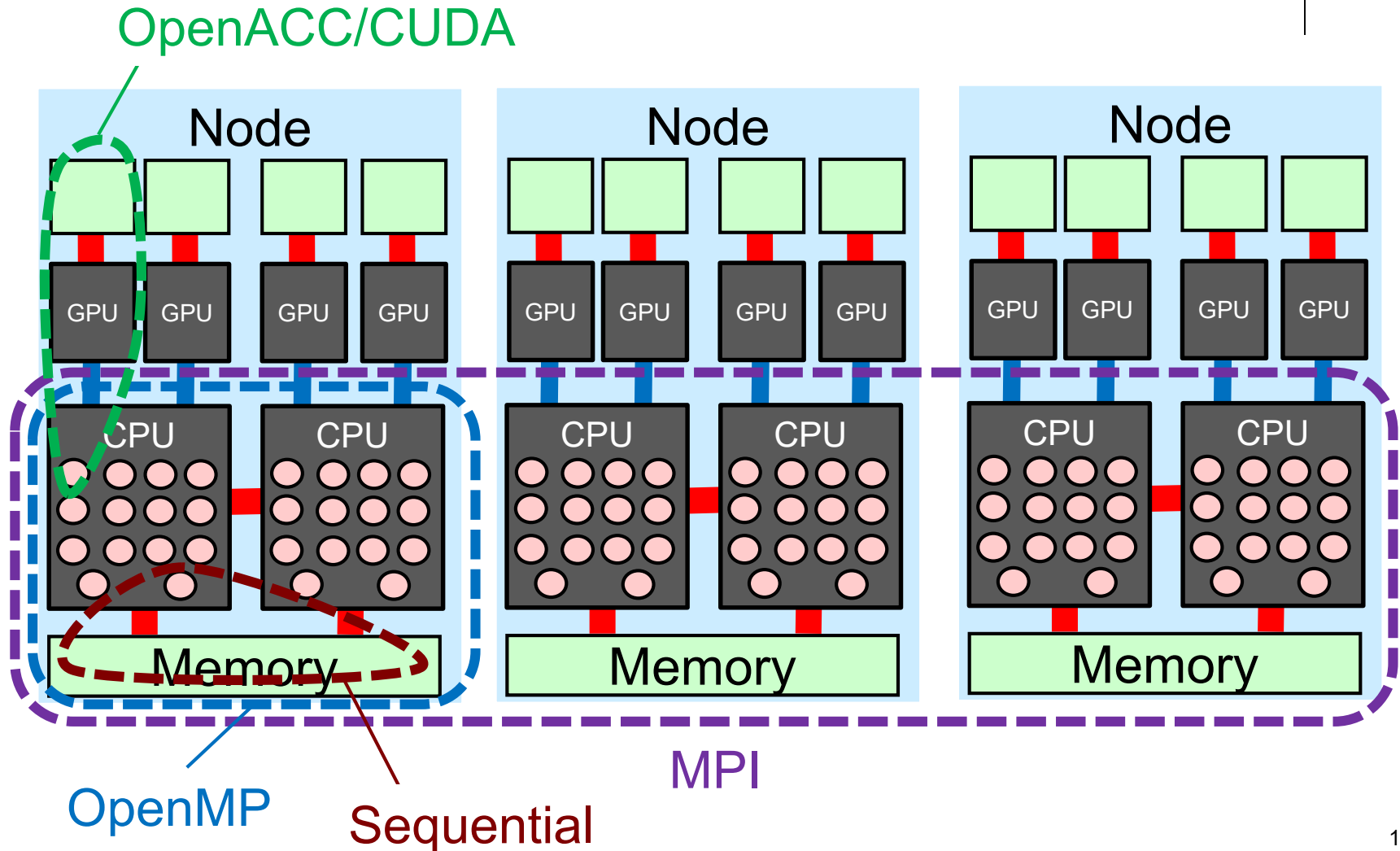
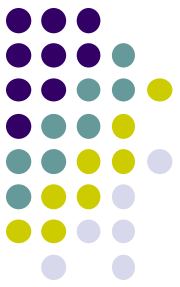


# Programming Models on Architecture



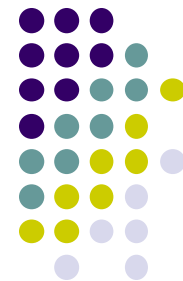
- Shared memory model (Part 1) can use only cores in a single node (up to 28 cores on TSUBAME3)
- Distributed memory model (Part 3) supports large scale parallelism (~15,000 cores on TSUBAME3)

# Parallel Programming Methods on TSUBAME

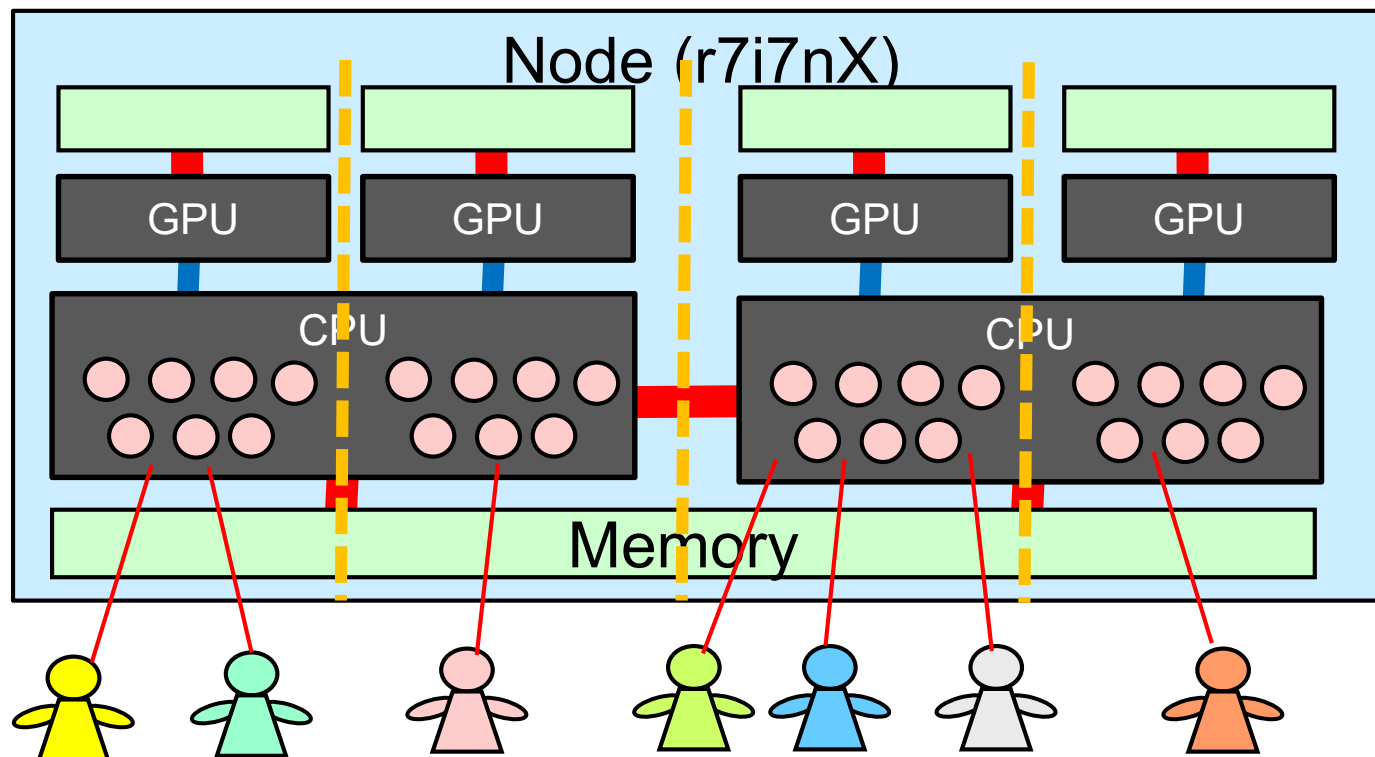


Standard route

Web-only route



# TSUBAME Interactive Node



A node is partitioned into 4. Each user can use

- $\frac{1}{4}$  node = 7 CPU cores + 60GB memory + 1 GPU (3584cores+16GB mem)
- Only one partition simultaneously

A partition may be shared by several users → you may suffer from slow down

# Sample Programs in this Lecture



- Samples are at </gs/hs1/tga-ppcomp/21/> directory
  - You have to be a member of [tga-ppcomp](#) group
    - If “[ls /gs/hs1/tga-ppcomp/21](#)” works well, you are a member
  - There are sub-directories per sample
- Sequential sample programs are
  - [mm](#): matrix multiplication
  - [pi](#): approximation of pi ( $\pi$ )
  - [diffusion](#): simple simulation of diffusion phenomena
  - [fib](#): Fibonacci number
  - [sort](#): quick-sort sample

# Using Sample Programs (1)

## Make Copies



- Samples in /gs/... are “*read-only*”, so make copies of samples into somewhere in your home directory
  - Where is somewhere? If you are using **web-only route**, **~/t3workspace** may be good
  - In the case of “**mm**” sample

*[make sure that you are at an interactive node (r7i7nX) ]*

```
cd ~/t3workspace    [In web-only route]
cp -r /gs/hs1/tga-ppcomp/21/mm .
cd mm
```

**don't forget  
space & dot**



# Using Sample Programs (2)

## Executing mm



- In the case of “mm” sample

[make sure that you are at mm directory]

ls

[you will see 3 files of mm.c, Makefile, job.sh]

make

[this creates an executable file “mm”]

./mm 1000 1000 1000

[this is the execution of mm sample]

# Using Sample Programs (3)

## Executing Samples



Before execution, please do “copy” and “make” for each sample

- mm

```
./mm 1000 1000 1000
```

Options are matrix sizes  $m, n, k$

- pi

```
./pi 10000000
```

Option is number of samples  $n$

- diffusion

```
./diffusion 20
```

Option is number of time steps  $nt$

- fib

```
./fib 40
```

Option is sequence index  $n$

- sort

```
./sort 10000000
```

Option is array length  $n$  to be sorted



# How Do We Edit C Programs?

There are many ways. The best way is up to you

## 1. Using editors on Linux

[1a] vim

[1b] emacs

Standard route

emacs is not good on web route, since Ctrl+s does not work well

## 2. Using editors on your PC

- You will repeat copy the file into PC, edit on PC, and copy it to TSUBAME again
- scp command on your PC, or WinSCP can be used
- Drag&drop Web-only route

## 3. Using Jupyter's editor Web-only route

# “mm” sample: Matrix Multiply



Available at </gs/hs1/tga-ppcomp/21/mm/>

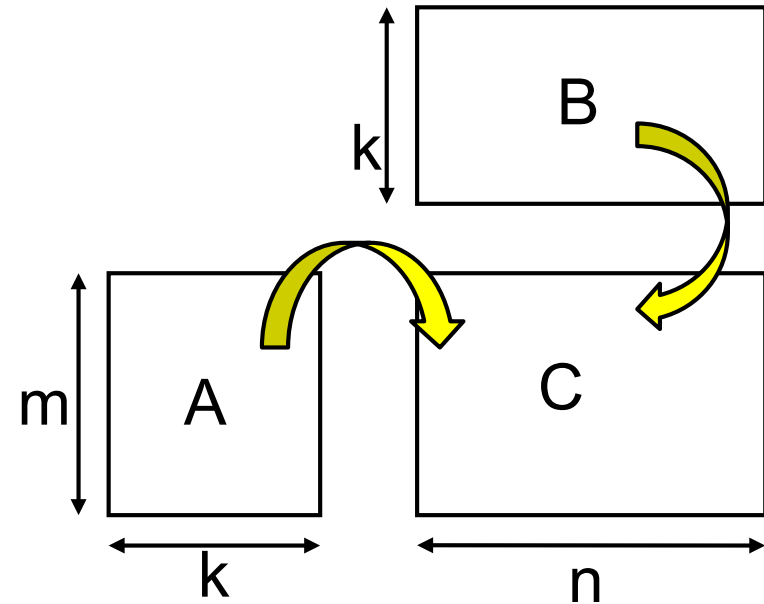
A: a  $(m \times k)$  matrix

B: a  $(k \times n)$  matrix

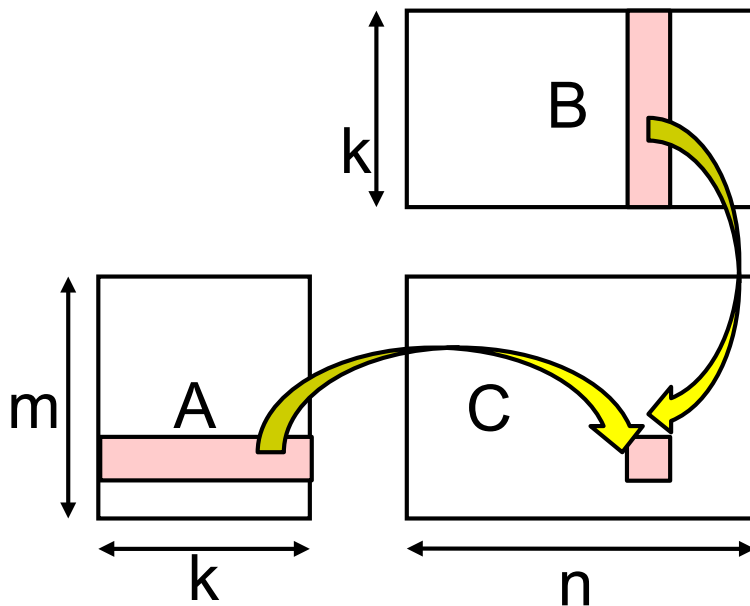
C: a  $(m \times n)$  matrix

$C \leftarrow A \ B$

- This sample supports variable matrix sizes
- Execution: `./mm [m] [n] [k]`



# Matrix Multiply Algorithm (1)



$C_{i,j}$  is defined as the dot product of

- A's i-th row
- B's j-th column

The algorithm uses triply-nested loop

```
for (i = 0; i < m; i++) {  
  for (j = 0; j < n; j++) {  
    for (l = 0; l < k; l++) {  
       $C_{i,j} += A_{i,l} * B_{l,j};$   
    }  
  }  
}
```

←For each row in C  
←For each column in C  
←For dot product



# Matrix Multiply Algorithm (2)

```
for (i = 0; i < m; i++) {  
  for (j = 0; j < n; j++) {  
    for (l = 0; l < k; l++) {  
      G,j += Ai,l * B,j;  
    }  
  }  
}
```

← For each row in C  
← For each column in C  
← For dot product

- The innermost statement is executed for  $mnk$  times
- Compute Complexity:  $O(mnk)$ 
  - Computation speed (Flops) is obtained as  $2mnk/t$ , where  $t$  is execution time

The innermost statement includes 2 (floating point) calculations:  $*$ ,  $+$

# Variable Length Arrays in (Classical) C Language



- `double C[n];` raises an error. How do we do?
- `void *malloc(size_t size);`  
⇒ Allocates a memory region of *size* bytes from “heap region”, and returns its head pointer
- When it becomes unnecessary, it should be discarded with `free()` function

*A fixed length array*

```
double C[5];  
  
... C[i] can be used ...
```

*A variable length array*

```
double *C;  
C = (double *)malloc(sizeof(double)*n);  
... C[i] can be used ...  
  
free(C);
```

array length

⌘ Exceptionally, C99 specification includes variable length arrays

# How We Do for Multiple Dimensional Arrays



`double C[m][n];` raises an error. How do we do?

Not in a straightforward way. Instead, we do either of:

(1) Use a pointer of pointers

- We *malloc*  $m$  1D arrays for every row (each has  $n$  length)
- We *malloc* 1D array of  $m$  length to store the above pointers

(2) Use a 1D array with length of  $m \times n$


(*mm sample uses this method*)

- To access an array element, we should use `C[i*n+j]` or `C[i+j*m]`, instead of `C[i][j]`



# Express a 2D array using a 1D array



  
“I want  
to use ...”

a 2D array  $C[m][n]$

$m$

8	3	7	4	1	2
0	2	1	5	0	3
1	8	6	4	2	1
3	4	8	1	0	2

$n$

$C[1][3]$

Expressions in C language (Example)

```
double *C; C = malloc(sizeof(double)*m*n);
```

$n$

8	3	7	4	1	2	0	2	1	5	0	3	.....	8	1	0	2
---	---	---	---	---	---	---	---	---	---	---	---	-------	---	---	---	---

$C[1*n+3]$

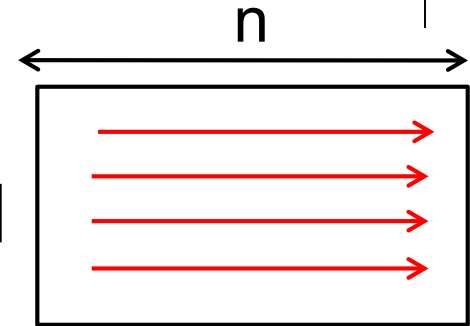
In this case, an element  $C_{i,j}$  is  $C[i*n+j]$

# Two Data Formats

## Row major format

- More natural for C programmers

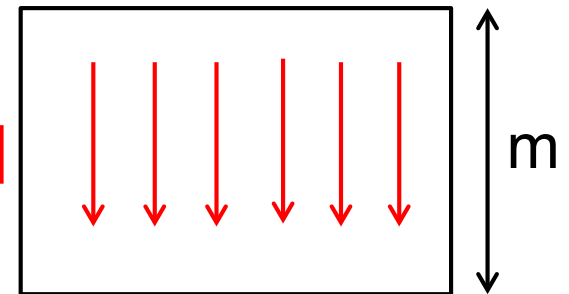
$$C_{i,j} \Rightarrow C[i*n+j]$$



## Column major format

- BLAS library
- mm sample uses this

$$C_{i,j} \Rightarrow C[i+j*m]$$



- We have more choices for 3D, 4D... arrays

[Q] Does the format affect the execution speed?



# Actual Codes in mm Sample

```
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        for (l = 0; l < k; l++) {  
            G,j += A,l * B,j;  
        } } }  
}
```

IJL order



```
for (j = 0; j < n; j++) {  
    for (l = 0; l < k; l++) {  
        double blj = B[l+j * 1 db];  
        for (i = 0; i < m; i++) {  
            double ail = A[i+l * 1 da];  
            Q[i+j * 1 dc] += ail * blj;  
        } } }  
}
```

Changed to JLI order  
(a bit faster)

=k

=m

# Time Measurement in Samples



- `gettimeofday()` function is used
  - It provides wall-clock time, not CPU time
  - Time resolution is better than `clock()`

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

:
{
    struct timeval st, et;
    long us;
    gettimeofday(&st, NULL); /* Starting time */
    ...Part for measurement...
    gettimeofday(&et, NULL); /* Finishing time */
    us = (et.tv_sec-st.tv_sec)*1000000+
        (et.tv_usec-st.tv_usec);
    /* us is difference between st & et in microseconds */
}
```



# If You Have Not Done This Yet

Please do the followings as soon as possible

- Please make your account on TSUBAME
- Please send an e-mail to [ppcomp@el.gsic.titech.ac.jp](mailto:ppcomp@el.gsic.titech.ac.jp)

Subject: [TSUBAME3 ppcomp account](#)

To: [ppcomp@el.gsic.titech.ac.jp](mailto:ppcomp@el.gsic.titech.ac.jp)

Department name:

School year:

Name:

Your TSUBAME account name:

Then we will invite you to the TSUBAME group, [please click URL and accept the invitation](#)

その後、TSUBAMEグループへの招待を送ります。[メール中のURLをクリックして参加承諾してください](#)

# Next Class: Introduction to OpenMP



- Shared memory parallel programming API
- Extensions to C/C++, Fortran
- Includes directives & library functions
  - Directives: `#pragma omp ~`

```
int i;  
#pragma omp parallel for  
for (i = 0; i < 100; i++) {  
    a[i] = b[i] + c[i];  
}
```