

Practical Parallel Computing (実践的並列コンピューティング)

Part 1: OpenMP

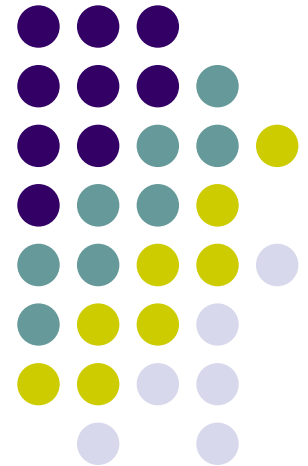
No 1: Introduction to OpenMP

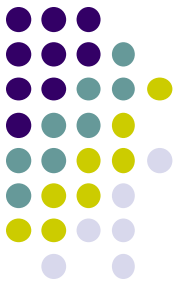
Apr 15, 2024

Toshio Endo

School of Computing & GSIC

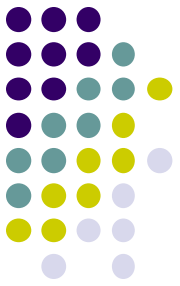
endo@is.titech.ac.jp





Overview of This Course

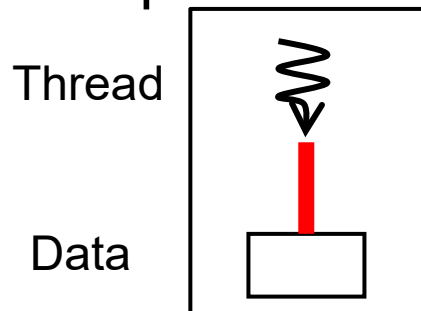
- Part 0: Introduction
 - 2 classes
- Part 1: **OpenMP** for shared memory programming
 - 4 classes **← We are here (1/4)**
- Part 2: **GPU** programming
 - OpenACC and CUDA
 - 4 classes
- Part 3: **MPI** for distributed memory programming
 - 3 or 4 classes



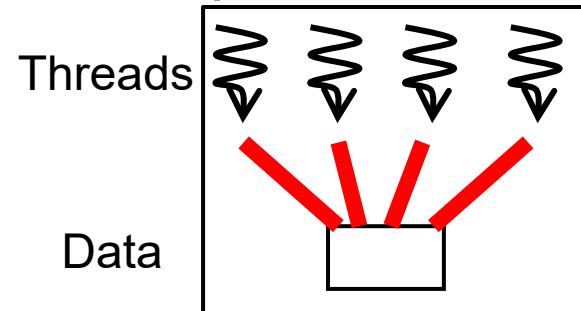
What is OpenMP?

- One of programming APIs based on **shared-memory** parallel model
 - Multiple threads work cooperatively using multiple cores
 - Threads can share data
 - Threads (in a program) are on a single node

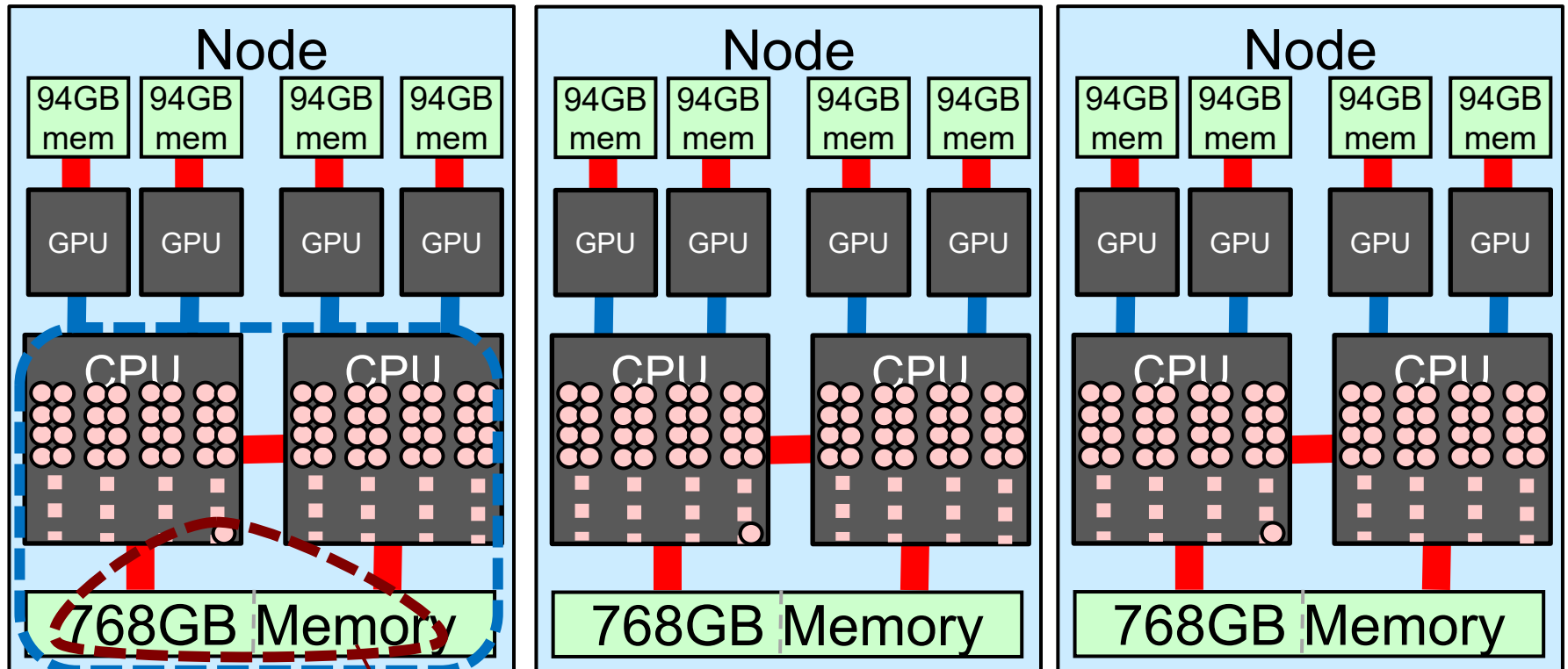
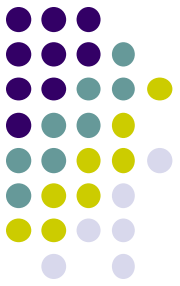
Simple C software



OpenMP software



With OpenMP, We Can Use Multiple Cores



OpenMP

Sequential



OpenMP Programs Look Like

- OpenMP defines extensions to C/C++/Fortran
- Directive syntaxes & library functions
 - Directives look like: `#pragma omp` ~~

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

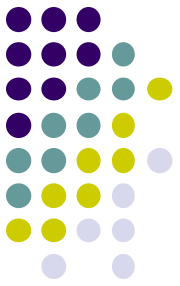
An example of OpenMP
directive

In this case, a directive has
an effect on the following
block/sentence



Sample Programs

- [/gs/bs/tga-ppcomp/24/](#) directory
 - You have to be a member of [tga-ppcomp](#) group
 - There are sub-directories per sample
- Samples related to today's class
 - [hello-omp](#)
 - matrix multiplication
 - [mm](#): sequential version
 - [mm-omp](#): OpenMP version
 - [mm-mkl](#): Using Intel MKL library



Using hello-omp Sample

[make sure that you are at an interactive node (r4nX)]

```
cd ~/ppc24
```

```
cp -r /gs/bs/tga-ppcomp/24/hello-omp .
```

```
cd hello-omp
```

```
make
```

[this creates an executable file “hello”]

```
./hello
```



Compiling OpenMP Programs

All famous compilers support OpenMP (fortunately☺), but require different options (unfortunately☹)

- gcc
 - `-fopenmp` option in compiling and linking
- NVIDIA HPC SDK (called PGI compiler)
 - `module load nvhpc`, and then use `pgcc`
 - `-mp` option in compiling and linking
- Intel compiler
 - `module load intel`, and then use `icx`
 - “icc” was deprecated
 - `-qopenmp` option in compiling and linking

} Our samples
use gcc

Check outputs of “make” in OpenMP sample directory

A Sequential Example

```
int main()
{
    A;

    {
        B;
    }
    C;

    D;
    E;
}
```

Flow of
execution

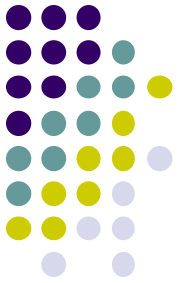
A

B

C

D

E



Basic Parallelism in OpenMP: Parallel Region



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

```
int main()  
{
```

```
  A;
```

```
  #pragma omp parallel
```

```
  {
```

```
    B;
```

```
  }
```

```
  C;
```

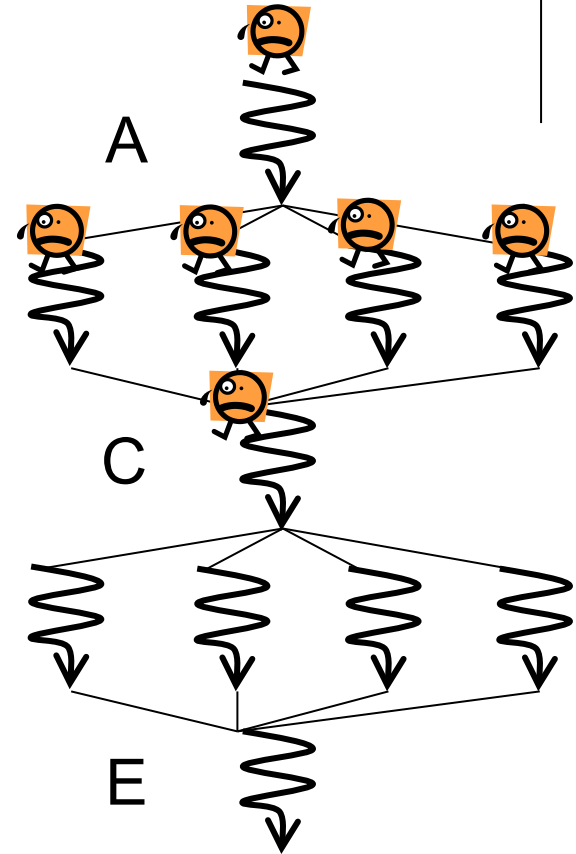
```
  #pragma omp parallel
```

```
  D;
```

```
  E;
```

```
}
```

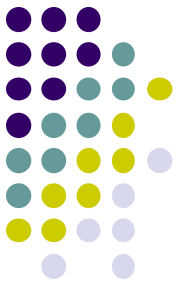
Parallel
region



Sentence/block immediately after **#pragma omp parallel** is called **parallel region**, executed by multiple threads

- Here a “block” is a region surrounded by braces `{ }`
- Functions called from parallel region are also in parallel region

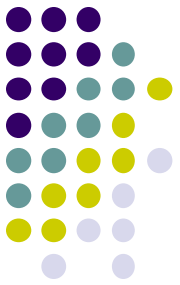
Number of Threads 🤪 🤪 🤪 🤪



How is the number of threads in parallel region determined?

1. Default: number of cores (including HyperThreads) are used
 - On a TSUBAME4 interactive node, it is $24 \times 2 = 48$
2. We can specify number of threads by **OMP_NUM_THREADS** environment variable
 - this is done out of program execution

```
export OMP_NUM_THREADS=4  
./hello [executed with 4 threads]
```
3. We can overwrite it inside the program
 - cf) `omp_set_num_threads(6);`



Outputs of hello-omp

Before the parallel region

```
Hello OpenMP World
I'm 8-th thread out of 48 threads
I'm 6-th thread out of 48 threads
I'm 9-th thread out of 48 threads
I'm 1-th thread out of 48 threads
I'm 0-th thread out of 48 threads
I'm 7-th thread out of 48 threads
:
Good Bye OpenMP World
```

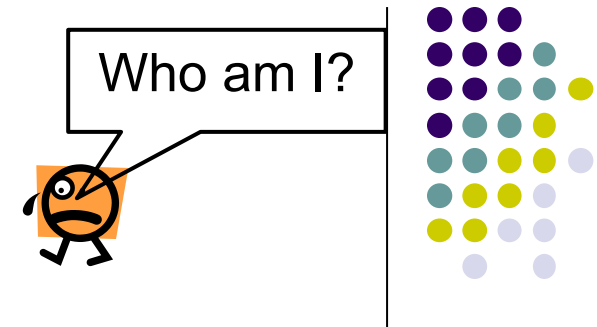
Inside the parallel region,
each thread prints a message
for several (5) times

omp_get_num_threads()

omp_get_thread_num()

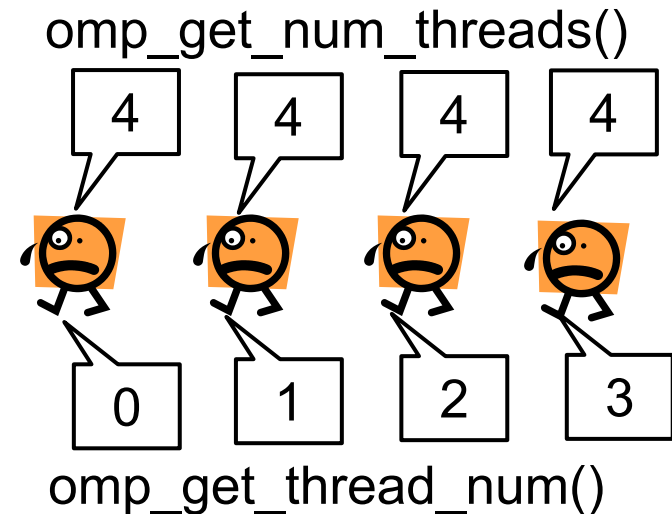
After the parallel region

To Know ID of Threads



In some programs, threads can distinguish each other
Each thread has an “ID number” (0, 1, 2 ...)

- To obtain number of threads
 - cf) `n = omp_get_num_threads();`
- To obtain my ID of calling thread
 - cf) `id = omp_get_thread_num();`
 - $0 \leq id < n$ (total number)



Executing a Sample with Various Number of Threads



[make sure that there is an executable file “hello”]

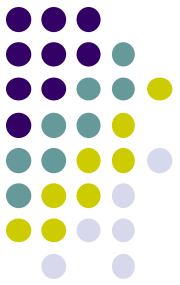
```
export OMP_NUM_THREADS=1  
./hello
```

```
export OMP_NUM_THREADS=8  
./hello
```

```
export OMP_NUM_THREADS=48  
./hello
```

```
export -n OMP_NUM_THREADS  
[This deletes variable]  
./hello
```

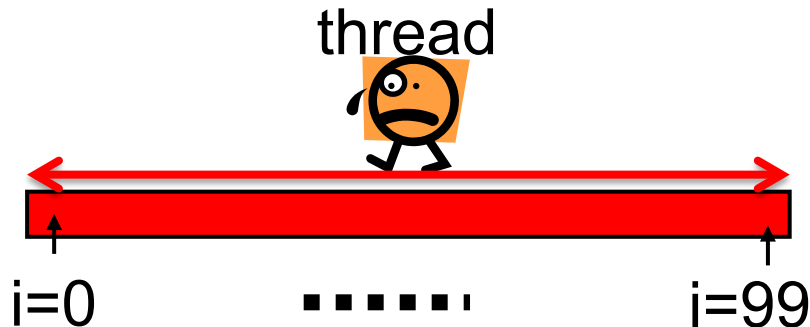
How Can We Make a Program Faster?



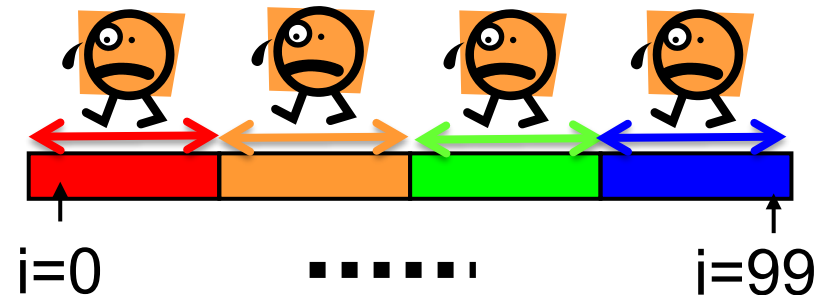
```
for (i = 0; i < 100; i++) { some computation; }
```

assumption: 100 tasks are independent with each other

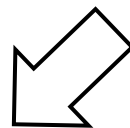
Only with one thread



With 4 threads



thread 0: for (i = 0 ; i < 25; ...
thread 1: for (i = 25; i < 50; ...
thread 2: for (i = 50; i < 75; ...
thread 3: for (i = 75; i < 100; ...



Fortunately, OpenMP makes this program very easy!

#pragma omp for for Easy Parallel Programming



“for” loop with simple forms can parallelized easily

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

#pragma omp for must be

- inside a parallel region
- right before a “for” loop

→ Computations in the loop are distributed among threads (work distribution)

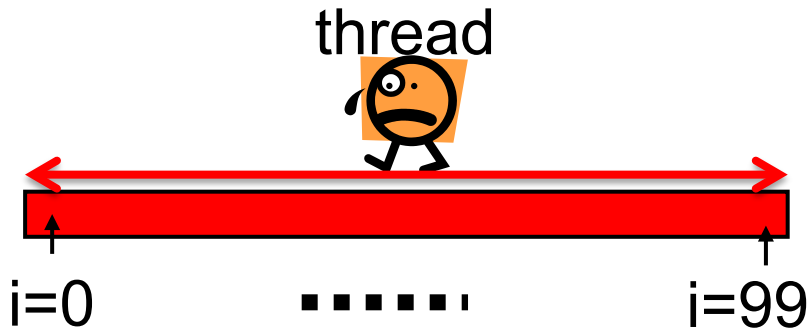
- With 4 threads, each thread take $100/4=25$ iterations → **speed up!!**
 - Indivisible cases are ok, such as 7 threads

- Abbreviation: omp parallel + omp for = omp parallel for

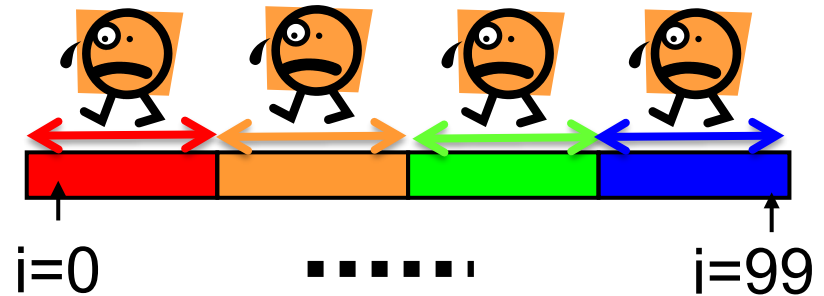
Why “omp for” Reduces Execution Time



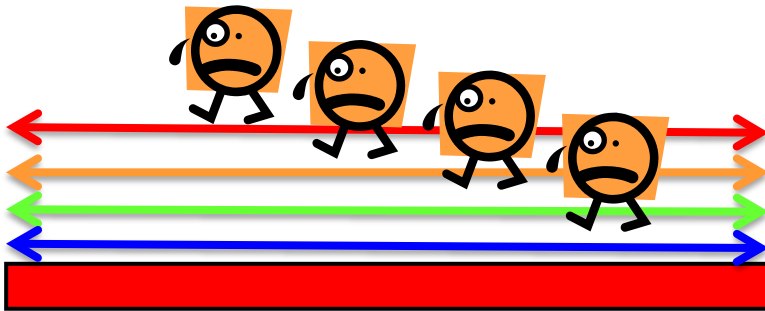
Only with one thread



With several threads



- What if we use “omp parallel”, but **forget** to write “omp for”?



Every thread would work for all iterations

→ No speed up ☹️

→ Answer will be wrong ☹️

“mm” sample: Matrix Multiply



In previous class, we tested </gs/bs/tga-ppcomp/24/mm/>, which is sequential

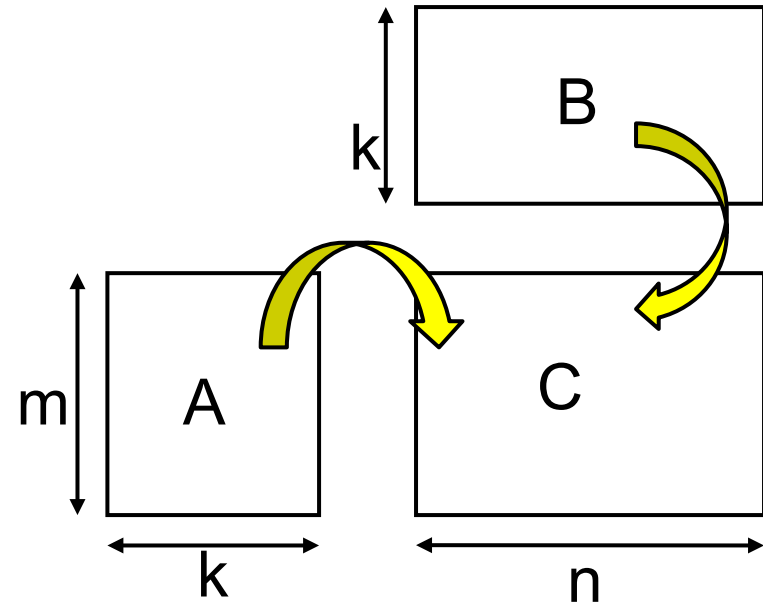
A: a $(m \times k)$ matrix

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]`



A parallel version is at </gs/bs/tga-ppcomp/24/mm-omp/>

- Please copy it to your home directory as usual

OpenMP Version of mm (mm-omp)

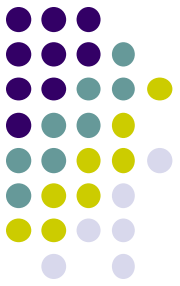


- matmul() function uses 3 loops
- Out of them, j loop is parallelized

```
#pragma omp parallel private(i,l)
#pragma omp for           ← j loop is parallelized
for (j = 0; j < n; j++) {
    for (l = 0; l < k; l++) {
        for (i = 0; i < m; i++) {
            C[i+j*ldc] += A[i+l*lda] * B[l+j*ldb];
        } } }
```

- “private” option is explained later

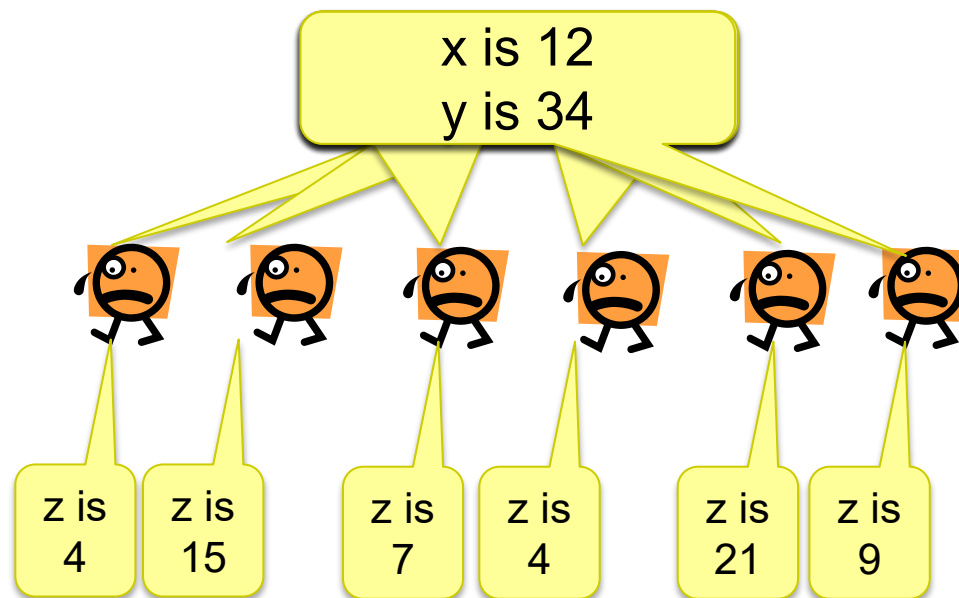
Shared Variables & Private Variables (1)



While OpenMP uses “shared memory model”, **not all are shared**

Two types of variables: **shared variables** and **private variables**

cf) Here variables *x, y* are shared, and *z* is private

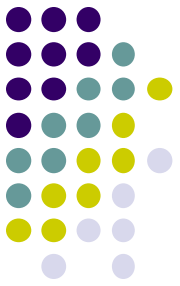


Shared →
*Single instance
for each *x, y* are
shared by threads*

Private →
*Each thread has
its own instance for *z**

- When a thread updates a shared variable, other threads are affected
 - We should be careful

Shared Variables & Private Variables (2)



In the default rule, variables are classified as follows

- Variables declared **out of** parallel region \Rightarrow **Shared variables**
- Global variables \Rightarrow **Shared variables**
- Variables declared **inside** parallel region \Rightarrow **Private variables**

```
{  
    int s = 1000; shared  
    #pragma omp parallel  
    {  
        int i; private  
        i = func(s, omp_get_thread_num());  
        printf( "%d\n" , i);  
    }  
}
```

```
int func(int a, int b)  
{  
    int rc = a+b; private  
    return rc;  
}
```



Pitfall in Nested Loops (1)

- The following code looks ok, but it has a bug
 - We do not see compile errors, but answers would be wrong ☹️

```
int i, j;  
#pragma omp parallel  
#pragma omp for  
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        ...  
    }  
}
```

Both i, j are declared
outside parallel region
→ Considered “shared”
It is a problem to share **j**

cf)

Thread A is executing i=5 loop
Thread B is executing i=8 loop

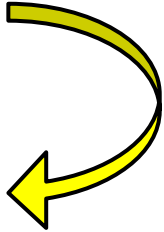
} The executions should be independent
Each execution must include
j=0, j=1...j=n-1 correctly
j must be private



Pitfall in Nested Loops (2)

Two modifications (Either is ok)

```
int i;  
#pragma omp parallel for  
for (i = 0; i < m; i++) {  
    int j;    // j is private  
    for (j = 0; j < n; j++) {  
        ...  
    } }  
}
```



```
int i, j;  
#pragma omp parallel for private(j)  
// j is forcibly private  
for (i = 0; i < m; i++) {  
    for (j = 0; j < n; j++) {  
        ...  
    } }  
}
```

OpenMP Version of mm (Again)



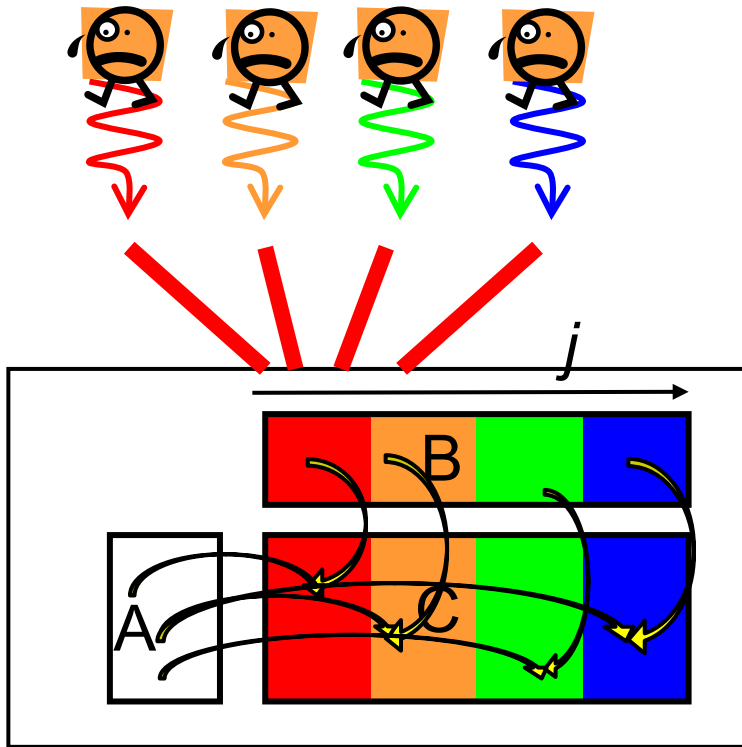
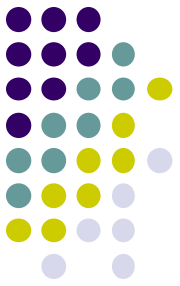
`#pragma omp parallel private(i,l)` → i, l made private forcibly

`#pragma omp for`

```
for (j = 0; j < n; j++) {  
    for (l = 0; l < k; l++) {  
        for (i = 0; i < m; i++) {  
            C[i+j*ldc] += A[i+l*lda] * B[l+j*ldb];  
        } } }  
}
```

In this sample, *j* loop is parallelized
→ Each thread executes computations
only for subset of [0, n)

How Arrays are Accessed in mm-omp



- It is programmers responsibility to make each thread does independent computation

j loop is parallelized

→ Each thread executes computations only for subset of $[0, n)$

[Q] What if we parallelize other loops?

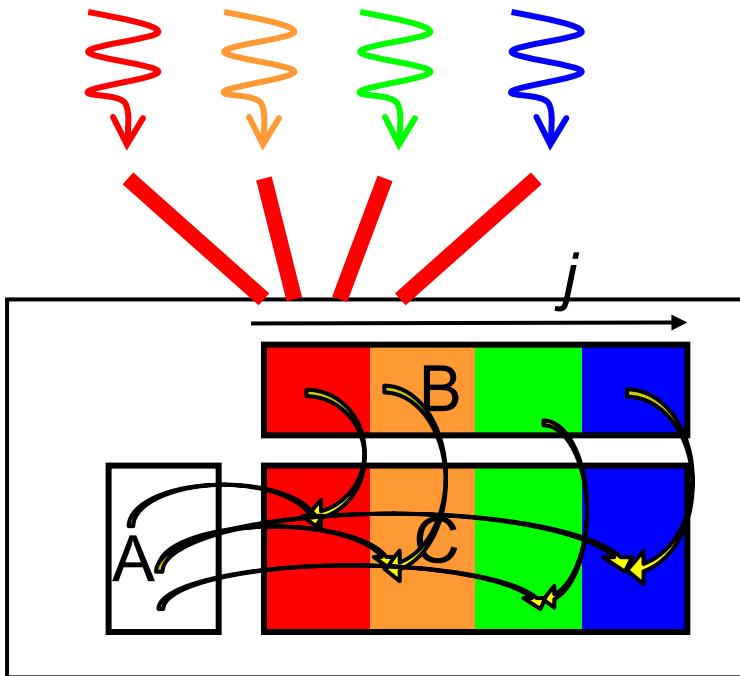
→ *i* loop is ok for correct answers, but may be slow

→ *l* loop causes wrong answers!

Correct Parallelization and Bad Parallelization



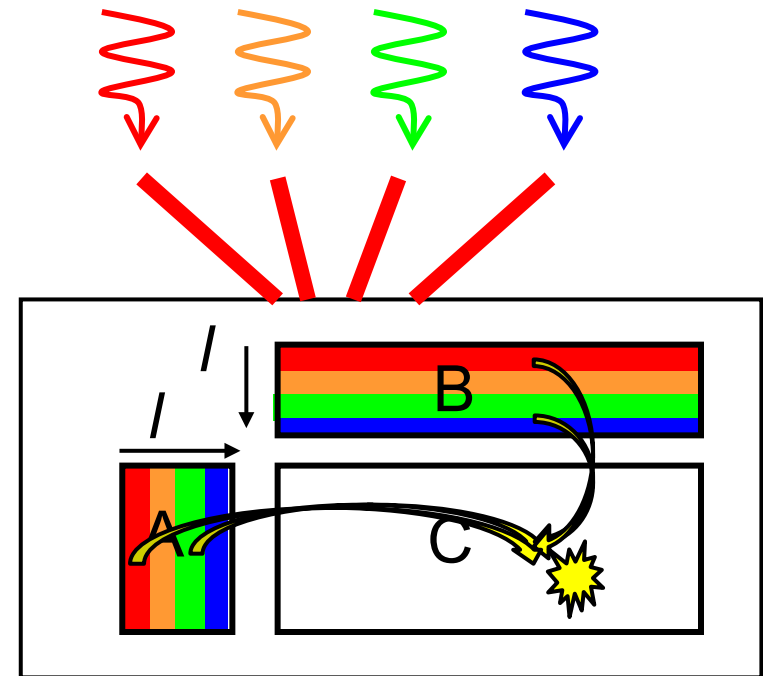
Parallelizing j loop



Simultaneous read from same data
(in this case, A) is OK

Similarly, parallelizing
 i loop is ok

Parallelizing i loop (??)

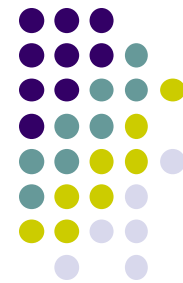


Possible simultaneous write to
same data

→ “Race condition” problem
may occur.

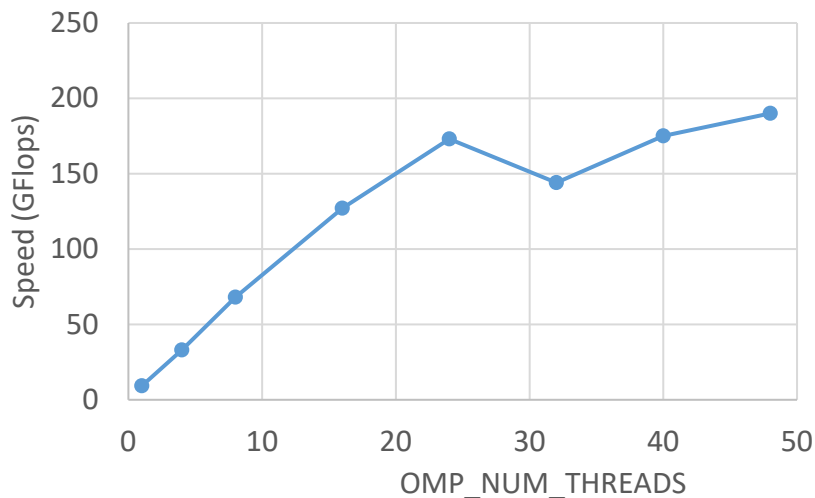
Answers may be wrong !!

Performance of mm-omp sample

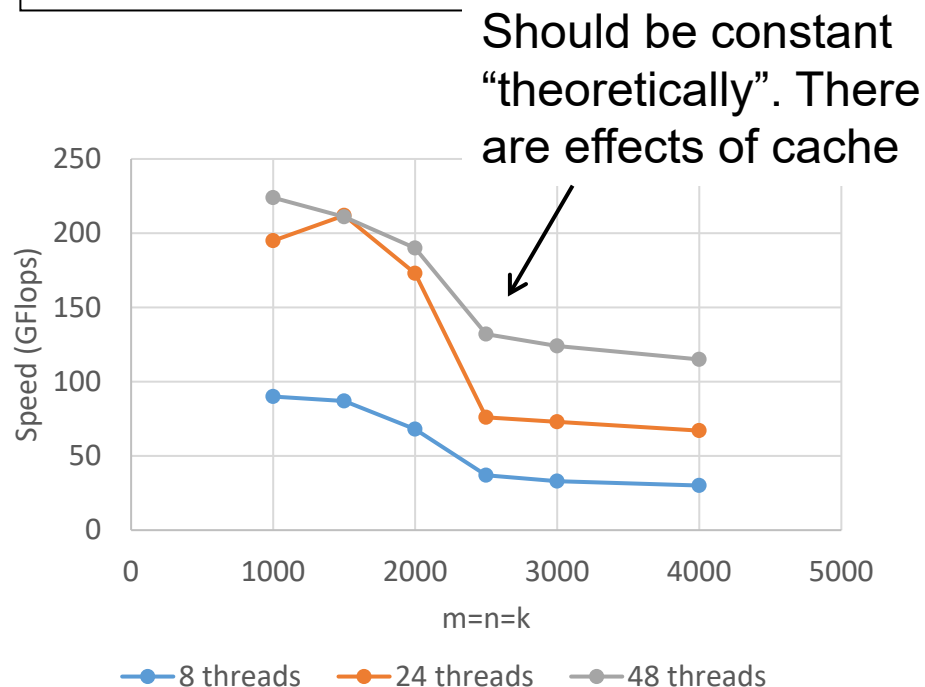


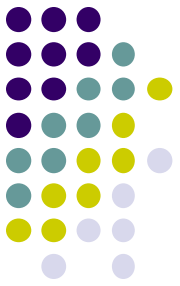
- On a TSUBAME4 interactive node (24 cores)
- Speed is (2mnk/t)

m=n=k=2000,
Varying # of threads



Varying m=n=k

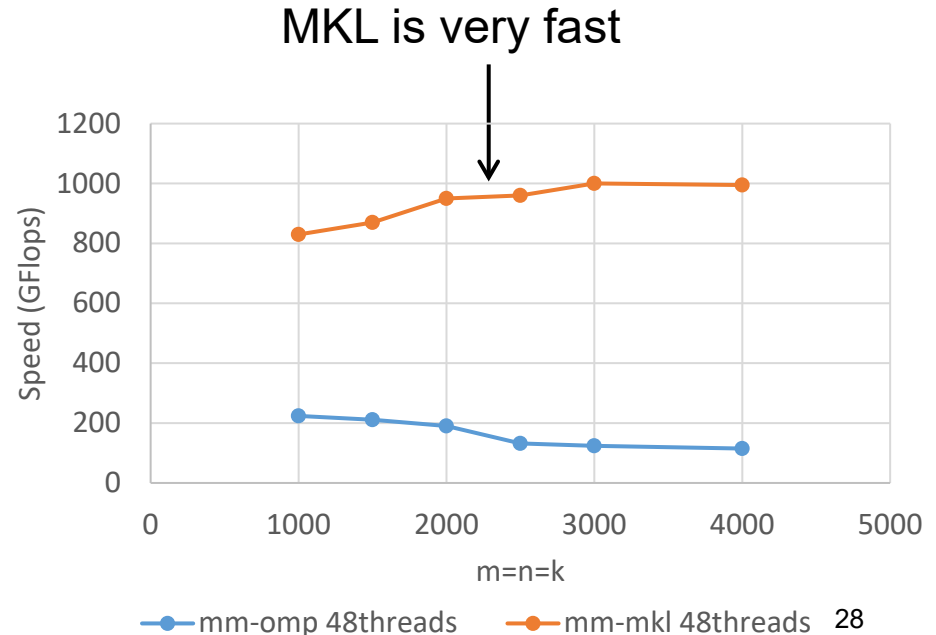




FYI: Using Optimized Libraries

- Each processor vendor has optimized (fast) libraries including matrix operations or deep learning kernels
 - Such as Intel MKL, NVIDIA cuBLAS/cuDNN...
- mm-mkl sample uses MKL library

```
cd ~/ppc24
cp -r /gs/bs/tga-ppcomp/24/mm-mkl .
cd mm-mkl
module load intel
make
export OMP_NUM_THREADS=48
./mm 2000 2000 2000
```





FYI: Why are Fast Libraries Fast?

Such libraries use many optimizing techniques

- Using multiple threads to use multiple cores
 - ↑ This lecture focuses on this
- Using SIMD instructions
- Using cache blocking techniques to harness cache memory
 - Unfortunately, MKL is not open-source
 - Some libraries such as OpenBLAS are open-source

“Python is slow” may not be true

- “numpy” linear algebra package may call a fast library automatically
 - `C = numpy.matmul(A, B)`
- Writing 3 loops in Python is very slow

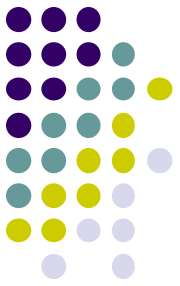


Today's Summary

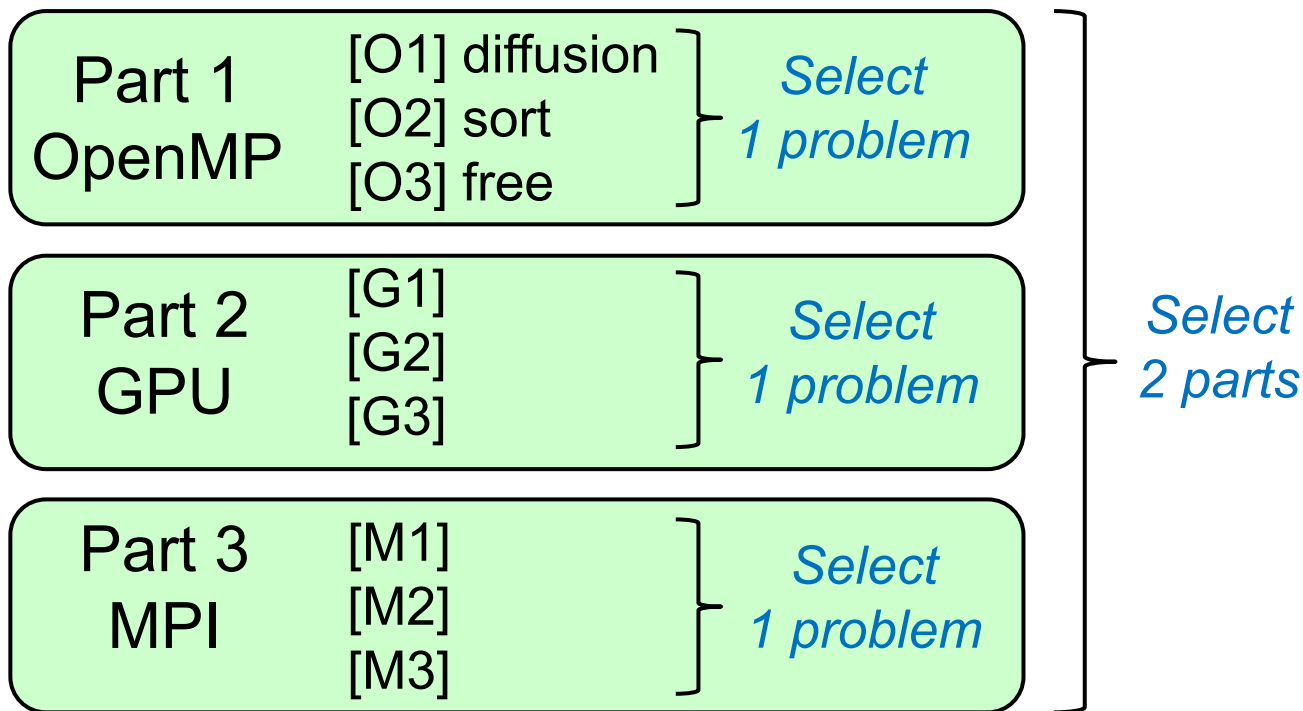
Introduction to OpenMP parallel programming

- Multiple threads work simultaneously with `#pragma omp parallel`
- With `#pragma omp for`, loop-based programs can be parallelized easily
- But it is programmer's responsibility to avoid bugs caused by race conditions

Assignments in this Course



- There is homework for each part. Submissions of reports for **2 parts** are required





Assignments in OpenMP Part (1)

Choose one of [O1]—[O3], and submit a report

Due date: May 9 (Thu)

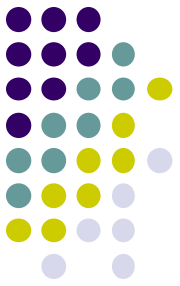
[O1] Parallelize “diffusion” sample program by OpenMP.

(</gs/bs/tga-ppcomp/24/diffusion/> on TSUBAME)

Optional:

- To make array sizes variable parameters, which are specified by execution options. “malloc” will be needed.
- To parallelize it without “omp for”
 - `omp_get_thread_num()`, `omp_get_num_threads()` are needed

Assignments in OpenMP Part (2)



[O2] Parallelize “sort” sample program by OpenMP.
(</gs/bs/tga-ppcomp/24/sort/> on TSUBAME)

Optional:

- Comparison with other algorithms than quick sort
 - Heap sort, Merge sort, Bitonic sort...

Assignments in OpenMP Part (3)



[O3] (Freestyle) Parallelize *any* program by OpenMP.

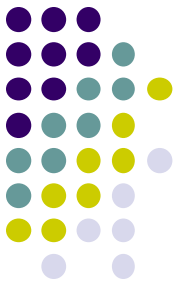
- cf) A problem related to your research
- More challenging one for parallelization is better
 - cf) Partial computations have dependency with each other
 - cf) Uniform task division is not good for load balancing

Notes in Report Submission (1)



- Submit the followings via **T2SCHOLA**
 - (1) **A report document**
 - PDF, MS-Word or text file
 - 2 pages or more
 - in English or Japanese (日本語もok)
 - (2) **Source code files** of your program
 - Try “zip” to submit multiple files

Notes in Report Submission (2)



The report document should include:

- Which problem you have chosen
- How you parallelized
 - It is even better if you mention efforts for high performance or new functions
- Performance evaluation on TSUBAME
 - With varying number of threads
 - On an interactive nodes, $1 \leq \text{OMP_NUM_THREADS} \leq 48$
 - To use more CPU cores, you need to do “job submission”
 - With varying problem sizes
 - Discussion with your findings
 - Other machines than TSUBAME are ok, if available

Announcement of TSUBAME Maintenance



- **TSUBAME4 will be stopped** for maintenance
 - 9:00, Mon, Apr 22 – 17:00, Wed, Apr 24 (plan)
 - TSUBAME4 portal is also stopped
- On Apr 22 class, we can not use TSUBAME



Next Class:

- Part1: OpenMP No. 2
 - diffusion: simple simulation of diffusion phenomena
 - Related to assignment [O1]