# Lecture 2-1 : OpenMP

## Lecturer : Atseng

# Course Information

Join the Discord First !

All the class information call discuss in Discord

https://discord.gg/wEGVXNT7Mh

# Course Contents

01 Thread intro

02 Pthread

03 OpenMP

04 Hands on time !

# 01 Core vs. Thread

# What is Core ?

> A core is a physical processing unit inside the cpu.

> Each core can handle one task or process at a time.

> More cores mean a CPU can perform multiple tasks simultaneously.

- a quad-core CPU has 4 cores that work independently.

# What is Thread ?

> A thread is a virtual component that represent a task handled by a core.

> Modern CPUs use ‘Hyper-Threading’(intel), ‘Simultaneous Multi-Threading’(AMD)

> This increases efficiency and performance.

# Core vs. Threads

> Cores are physical unit; Threads are virtual.

> Threads rely on the cores to execute tasks.

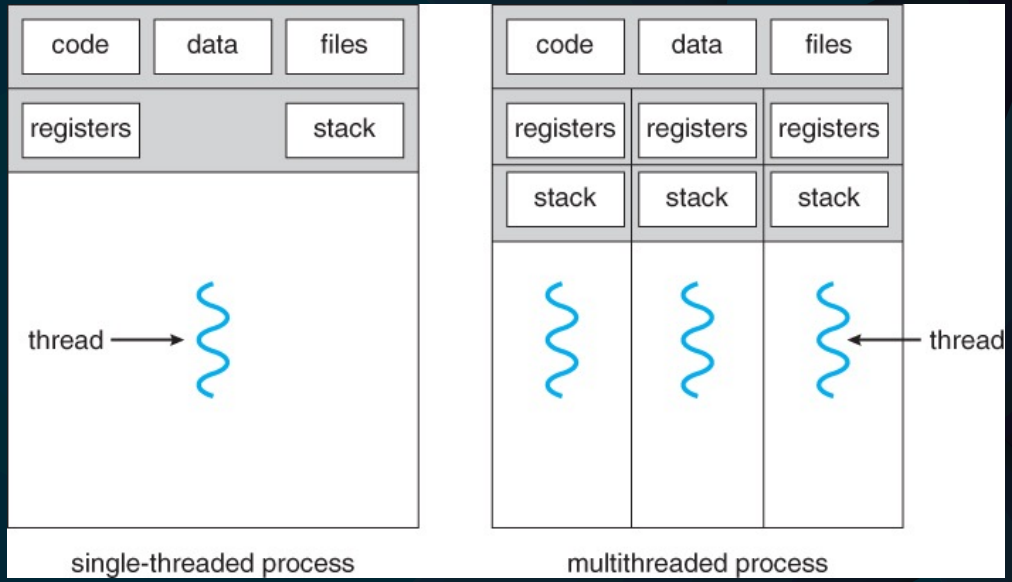
> A CPU with 4 cores and 8 threads can handle up to 8 tasks at the same time

> More threads improve multitasking but don't always double performance.

# Process vs. Threads

> Process: Each process has its own independent memory space

> Threads : Threads within the same process share the same memory space and resources



# Shared Memory Programming

## Definition

- Multiple processes or threads execute within the same memory space.

## Pros :

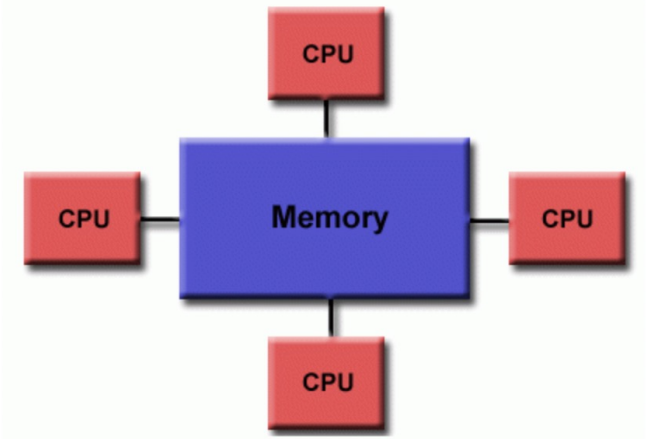
- Fast communication
- Low latency
- Efficient resource sharing

## Cons :

- may have issue with Synchronization
- Deadlock
- Cache coherence

## Uniform Memory Access (UMA)

- Most commonly represented today by ***Symmetric Multiprocessor (SMP)*** machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

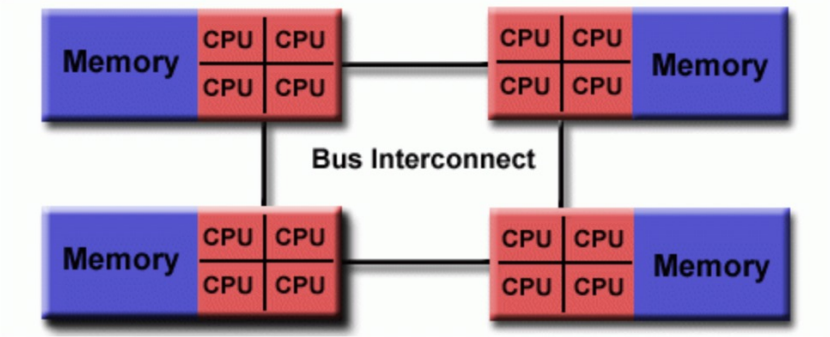



*Uniform memory access*

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##SharedMemory

# Non-Uniform Memory Access (NUMA)

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA




*Non-uniform memory access*

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##SharedMemory

# 02 Pthread

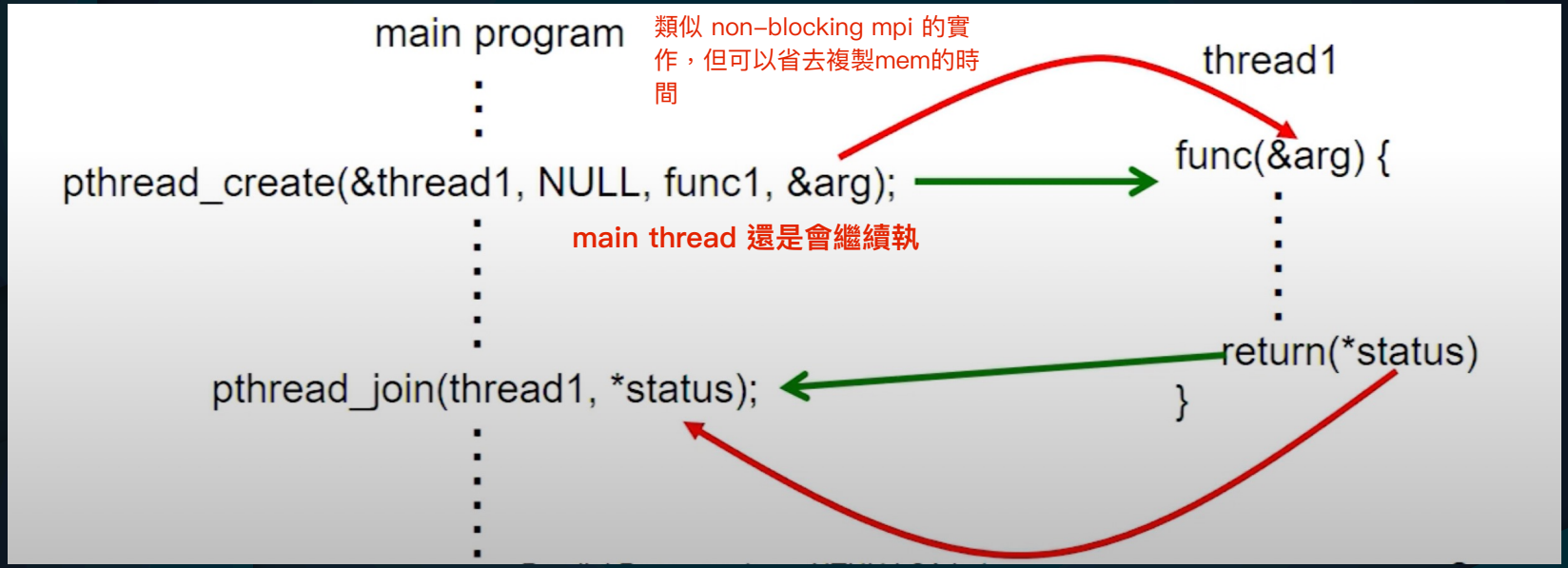
# Pthread

> POSIX Thread = Pthread

- POSIX (Potable Operating System Interface) standard is specified for portability across Unix-like systems
  - Similar concept as MPI for message passing libraries
- Pthread is the implementation of POSIX standard for thread
  - Same relation between MPICH and MPI

https://www.youtube.com/watch?v=IXXImEGZ7ZU&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=19

# Pthread




https://www.youtube.com/watch?v=IXXImEGZ7ZU&list=PLS0SUwlYe8cxqw70UHOE5n4Lm-mXFXbZT&index=19

# Pthread_create

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   typeof(void *(void *)) *start_routine,
                   void *restrict arg);
```

https://man7.org/linux/man-pages/man3/pthread_create.3.html

```
pthread_t t1, t2;
int t1_num = 1;
int t2_num = 2;
pthread_create(&t1, NULL, thread_func, &t1_num);
```

# Pthread_join

```c
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

https://man7.org/linux/man-pages/man3/pthread_join.3.html

```c
36      pthread_join(t1, NULL);
37      pthread_join(t2, NULL);
```

這裡有 blocking

# Pthread code

```c
int main() {
    pthread_t t1, t2;
    int t1_num = 1;
    int t2_num = 2;

    // create first thread
    if (pthread_create(&t1, NULL, thread_func, &t1_num) != 0) {
        perror("Failed to create thread 1");
        return 1;
    }

    // create second thread
    if (pthread_create(&t2, NULL, thread_func, &t2_num) != 0) {
        perror("Failed to create thread 2");
        return 1;
    }

    // wait for threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("All threads finished.\n");
    return 0;
}
```

# Pthread code

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// threqd function
void* thread_func(void* arg) {
    int thread_num = *(int*)arg;
    printf("Hello from thread %d\n", thread_num);

    // thread work
    for (int i = 0; i < 5; i++) {
        printf("Thread %d is working... (i=%d)\n", thread_num, i);
    }

    // return
    pthread_exit((void*)0);
}
```

```c
int main() {
    pthread_t t1, t2;
    int t1_num = 1;
    int t2_num = 2;

    // create first thread
    if (pthread_create(&t1, NULL, thread_func, &t1_num) != 0) {
        "Failed to create thread 1");
        1;

    econd thread
    _create(&t2, NULL, thread_func, &t2_num) != 0) {
    'Failed to create thread 2");
    1;

     threads to finish
    ı(t1, NULL);
    ı(t2, NULL);

     threads finished.\n");
```

# Pthread

```
gcc -o thread_program yourfilename.c -lpthread
```

```
./thread_program
```

```
[atseng@TsengdeMacBook-Pro GPUtest % gcc -o thread_program thread.c -lpthread
[atseng@TsengdeMacBook-Pro GPUtest % ./thread_program
Hello from thread 1
Thread 1 is working... (i=0)
Thread 1 is working... (i=1)
Thread 1 is working... (i=2)
Thread 1 is working... (i=3)
Thread 1 is working... (i=4)
Hello from thread 2
Thread 2 is working... (i=0)
Thread 2 is working... (i=1)
Thread 2 is working... (i=2)
Thread 2 is working... (i=3)
Thread 2 is working... (i=4)
All threads finished.
```

# 03 OpenMP

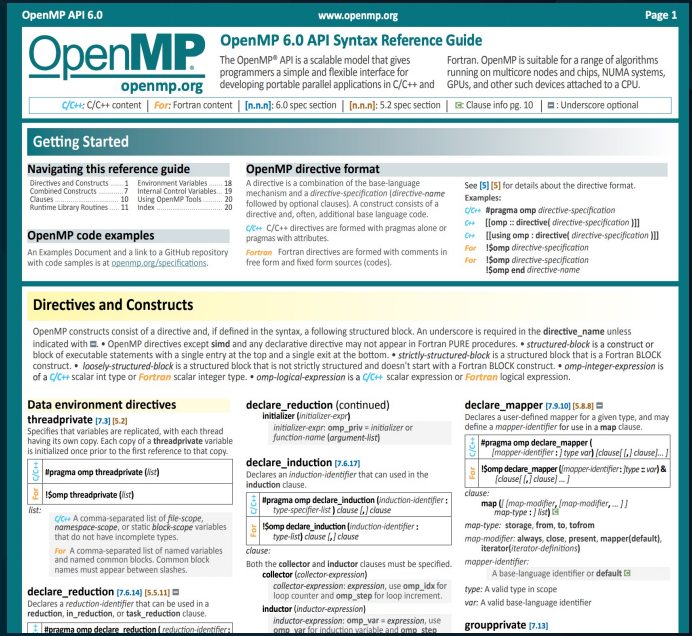
# OpenMP

> Easy Parallelization

> Shared memory model

> Compiler Directives

https://www.openmp.org/resources/refguides/



OpenMP API 6.0 www.openmp.org Page 1

OpenMP 6.0 API Syntax Reference Guide

openmp.org

The OpenMP® API is a scalable model that gives programmers a simple and flexible interface for developing portable parallel applications in C/C++ and Fortran. OpenMP is suitable for a range of algorithms running on multicore nodes and chips, NUMA systems, GPUs, and other such devices attached to a CPU.

C/C++: C/C++ content | For: Fortran content | [n.n.n]: 6.0 spec section | [n.n.n]: 5.2 spec section | : Clause info pg. 10 | : Underscore optional

Getting Started

Navigating this reference guide



OpenMP code examples

An Examples Document and a link to a GitHub repository with code samples is at openmp.org/specifications.

OpenMP directive format

A directive is a combination of the base-language mechanism and a *directive-specification* (*directive-name* followed by optional clauses). A construct consists of a directive and, often, additional base language code.

C/C++ C/C++ directives are formed with pragmas alone or pragmas with attributes.

Fortran Fortran directives are formed with comments in free form and fixed form sources (codes).

See [5] [5] for details about the directive format.

Examples:

C/C++ #pragma omp *directive-specification*

C++ [[omp :: directive( *directive-specification* )]]

C++ [[using omp : directive( *directive-specification* )]]

For !$omp *directive-specification*

For !$omp *directive-specification*
!$omp end *directive-name*

Directives and Constructs

OpenMP constructs consist of a directive and, if defined in the syntax, a following structured block. An underscore is required in the **directive_name** unless indicated with . • OpenMP directives except **simd** and any declarative directive may not appear in Fortran PURE procedures. • *structured-block* is a construct or block of executable statements with a single entry at the top and a single exit at the bottom. • *strictly-structured-block* is a structured block that is a Fortran BLOCK construct. • *loosely-structured-block* is a structured block that is not strictly structured and doesn't start with a Fortran BLOCK construct. • *omp-integer-expression* is of a C/C++ scalar int type or Fortran scalar integer type. • *omp-logical-expression* is a C/C++ scalar expression or Fortran logical expression.

Data environment directives

**threadprivate** [7.3] [5.2]

Specifies that variables are replicated, with each thread having its own copy. Each copy of a **threadprivate** variable is initialized once prior to the first reference to that copy.

C/C++ #pragma omp threadprivate (*list*)

For !$omp threadprivate (*list*)

*list:*

C/C++ A comma-separated list of *file-scope*, *namespace-scope*, or static *block-scope* variables that do not have incomplete types.

For A comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

**declare_reduction** [7.6.14] [5.5.11]

Declares a *reduction-identifier* that can be used in a **reduction**, **in_reduction**, or **task_reduction** clause.

C/C++ #pragma omp declare_reduction ( *reduction-identifier* :

**declare_reduction** (continued)

**initializer** (*initializer-expr*)

*initializer-expr*: **omp_priv** = *initializer* or *function-name* (*argument-list*)

**declare_induction** [7.6.17]

Declares an *induction-identifier* that can used in the **induction** clause.

C/C++ #pragma omp declare_induction (*induction-identifier* : *type-specifier-list* ) *clause* [,] *clause*

For !$omp declare_induction (*induction-identifier* : *type-list*) *clause* [,] *clause*

*clause:*

Both the **collector** and **inductor** clauses must be specified.

**collector** (*collector-expression*)

*collector-expression*: *expression*, use **omp_idx** for loop counter and **omp_step** for loop increment.

**inductor** (*inductor-expression*)

*inductor-expression*: **omp_var** = *expression*, use omp_var for induction variable and omp_step

**declare_mapper** [7.9.10] [5.8.8]

Declares a user-defined mapper for a given type, and may define a *mapper-identifier* for use in a **map** clause.

C/C++ #pragma omp declare_mapper ( [*mapper-identifier* :] *type var*) *clause*[ [,] *clause*]... )

For !$omp declare_mapper ([*mapper-identifier*:]*type* :: *var*) & *clause*[ [,] *clause*] ... )

*clause:*

**map** ( [ *map-modifier*, [*map-modifier*, ... ] ] *map-type* : ) *list*)

*map-type:* **storage**, **from**, **to**, **tofrom**

*map-modifier:* **always**, **close**, **present**, **mapper(default)**, **iterator**(*iterator-definitions*)

*mapper-identifier:*

A base-language identifier or **default**

*type:* A valid type in scope

*var:* A valid base-language identifier

**groupprivate** [7.13]

https://www.openmp.org/wp-content/uploads/OpenMP-RefGuide-6.0-OMP60SC24-web.pdf

# Compiler Directives

```
// Parallel region
#pragma omp parallel
{
    // Code here will be executed by multiple threads simultaneously
}

// Work distribution
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    // Loop iterations will be automatically distributed among threads
}
```

# Simple example (Vector Addition)

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

int main() {
    int n = 1000000;
    int a[n], b[n], c[n];

    // Initialize arrays
    for (int i = 0; i < n; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

    // Parallel computation of vector addition
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }

    printf("c[500000] = %d\n", c[500000]);
    return 0;
}
```

# Data sharing and privacy

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

- `shared` - Specifies variables shared among threads
- `private` - Specifies variables private to each thread

# Synchronization Mechanisms

```
// Critical section
#pragma omp critical
{
    // Only one thread can execute this code at a time
    sum += local_sum;
}

// Barrier
#pragma omp barrier
// All threads will wait here until everyone arrives

// Atomic operation
#pragma omp atomic
counter++;
```

確保單一執行緒 (Thread) 對變數的存取是原子性 (Atomic) 的，避免多個執行緒同時修改變數導致不一致性。

僅適用於單一變數的簡單操作（只限於下面那行 code），如 +=, -=, *=, /=, ++, --，不適用於更複雜的程式碼區塊。

比 #pragma omp critical 更輕量，執行效率通常較高，因為 atomic 只保護單一變數操作，而 critical 可能會產生額外的同步開銷。

# Task Parallelism

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        task1();

        #pragma omp task
        task2();
    }
    // Continue after all tasks complete
}
```

# Thread Count Control

```
// Set number of threads
omp_set_num_threads(4);

// Get current thread count
int num_threads = omp_get_num_threads();

// Get current thread ID
int thread_id = omp_get_thread_num();
```

# 04 Hands on time

# macOS (install llvm with homebrew)

/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

brew install llvm

### 1. Install Homebrew (if not already installed)

bash Copy

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD
```

### 2. Install LLVM using Homebrew

bash Copy

```
brew install llvm
```

This will install the latest version of LLVM, which includes Clang with full OpenMP support.

# macOS (Apple Silicon Macs : M1,M2,M3)

echo 'export PATH="/opt/homebrew/opt/llvm/bin:$PATH"' >> ~/.zshrc

```
echo 'export PATH="/usr/local/opt/llvm/bin:$PATH"' >> ~/.zshrc
```

source ~/.zshrc

```
source ~/.zshrc  # or ~/.bash_profile if using bash
```

# macOS (other Macs)

echo 'export PATH="/usr/local/opt/llvm/bin:$PATH"' >> ~/.zshrc

```
echo 'export PATH="/usr/local/opt/llvm/bin:$PATH"' >> ~/.zshrc
```

source ~/.zshrc

```
source ~/.zshrc  # or ~/.bash_profile if using bash
```

# Using Make file

```
CC = /opt/homebrew/opt/llvm/bin/clang++
CFLAGS = -fopenmp -O2

all: my_program

my_program: test_openmp.cpp
  $(CC) $(CFLAGS) -o my_program test_openmp.cpp

run: my_program
  OMP_DISPLAY_ENV=VERBOSE ./my_program

clean:
  rm -f my_program
```

# Sample code

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

int main() {
    std::cout << "OpenMP Version: " << _OPENMP << std::endl;

    #pragma omp parallel
    {
        #pragma omp critical
        {
            std::cout << "Thread " << omp_get_thread_num()
                      << " of " << omp_get_num_threads() << std::endl;
        }
    }

    return 0;
}
```

# Sample code2

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

#define SIZE 100000000

int main() {
    double a[SIZE], b[SIZE], c[SIZE];
    int i;
    double start_time, end_time;

    // Initialize arrays
    for (i = 0; i < SIZE; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }

    // Record start time
    start_time = omp_get_wtime();

    // Parallelize the loop
    #pragma omp parallel for
    for (i = 0; i < SIZE; i++) {
        c[i] = a[i] + b[i];
    }

    // Record end time
    end_time = omp_get_wtime();

    printf("Computation completed! Time: %f seconds\n", end_time - start_time);
    printf("Verification: c[0]=%f, c[%d]=%f\n", c[0], SIZE-1, c[SIZE-1]);

    return 0;
}
```

# It’s your turn!

# Course Reference

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##SharedMemory

https://www.youtube.com/watch?v=nE-xN4Bf8XI&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG&index=2

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html