



信息科学与技术学院

School of Information Science and Technology

# CS 110

# Computer Architecture

# Thread-Level Parallelism

**Instructors:**

**Chundong Wang, Siting Liu & Yuan Xiao**

Course website: <https://toast->

[lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html](https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html)

**School of Information Science and Technology (SIST)**

**ShanghaiTech University**

2025/5/6

# Administratives

- Mid-term II May 15th 8am-10am; you can bring 2-page A4-sized double-sided cheat sheet, **handwritten** only! (**Teaching center 201/301/303**); From start to May 13th lecture (Thread-level parallelism/TLP).
- Project 2.2 released, ddl May 19th.
- HW5 ddl **approaching**, May 7th.
- HW6 released, ddl May 12th!
- Lab 12 to be released. **Prepare in advance!**
  - To check May 13th, 15th & 19th;
- Discussion May 9th & 12th on SIMD.

# Parallelism Overview

- **Parallel Requests**  
Assigned to computer  
e.g., Search “CS110”
- **Parallel Threads**  
Assigned to core  
e.g., Lookup, Ads
- **Parallel Instructions**  
>1 instruction @ one time  
e.g., 5 pipelined instructions
- **Parallel Data**  
>1 data item @ one time  
e.g., Add of 4 pairs of words
- **Hardware descriptions**  
All gates @ one time
- **Programming Languages**

*Software*

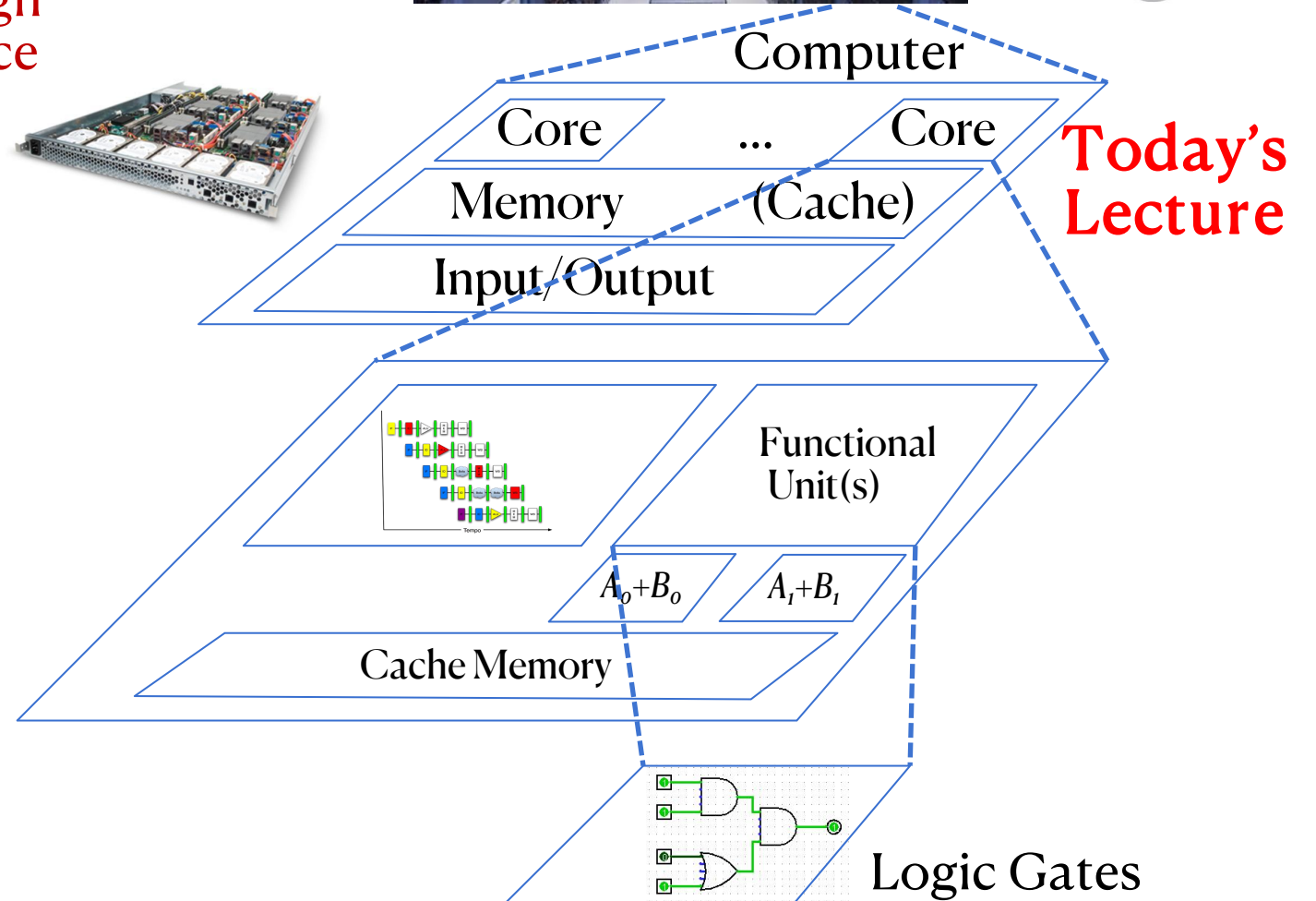
*Hardware*

**Harness  
Parallelism &  
Achieve High  
Performance**

Warehouse  
Scale  
Computer

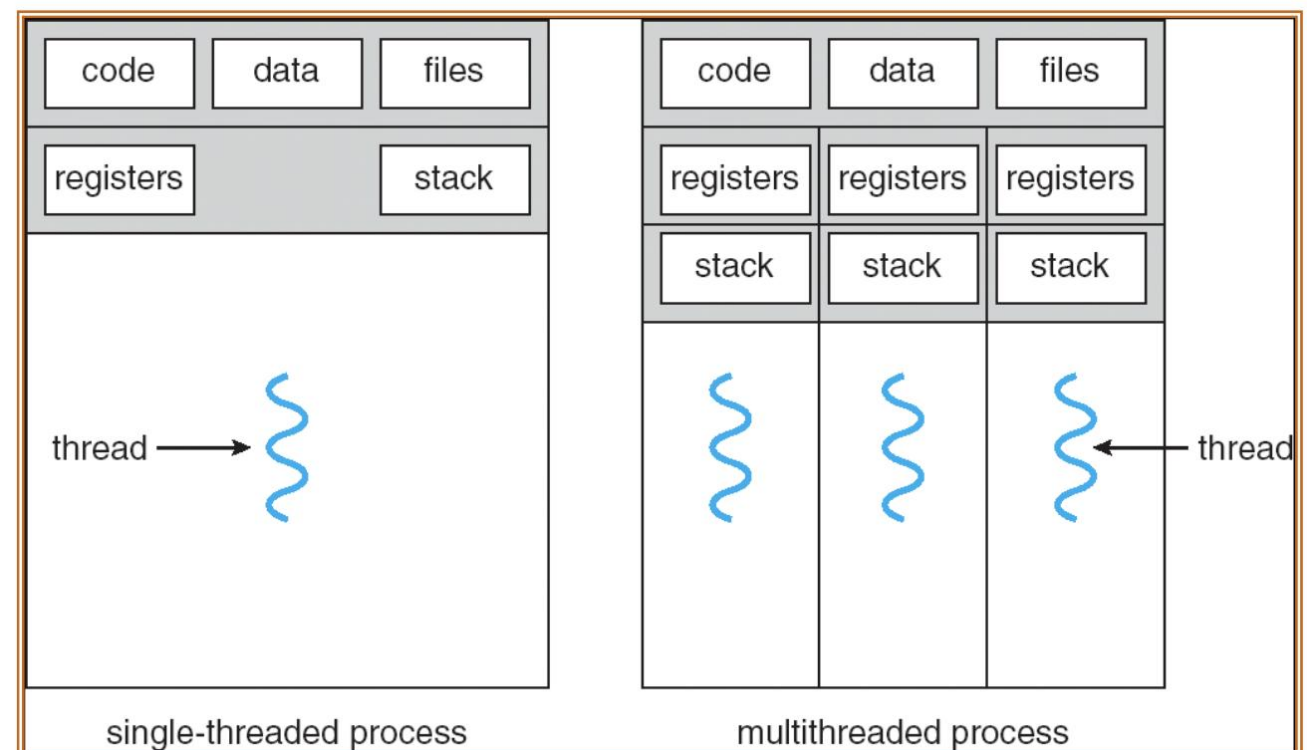


Smart  
Phone



# Threads

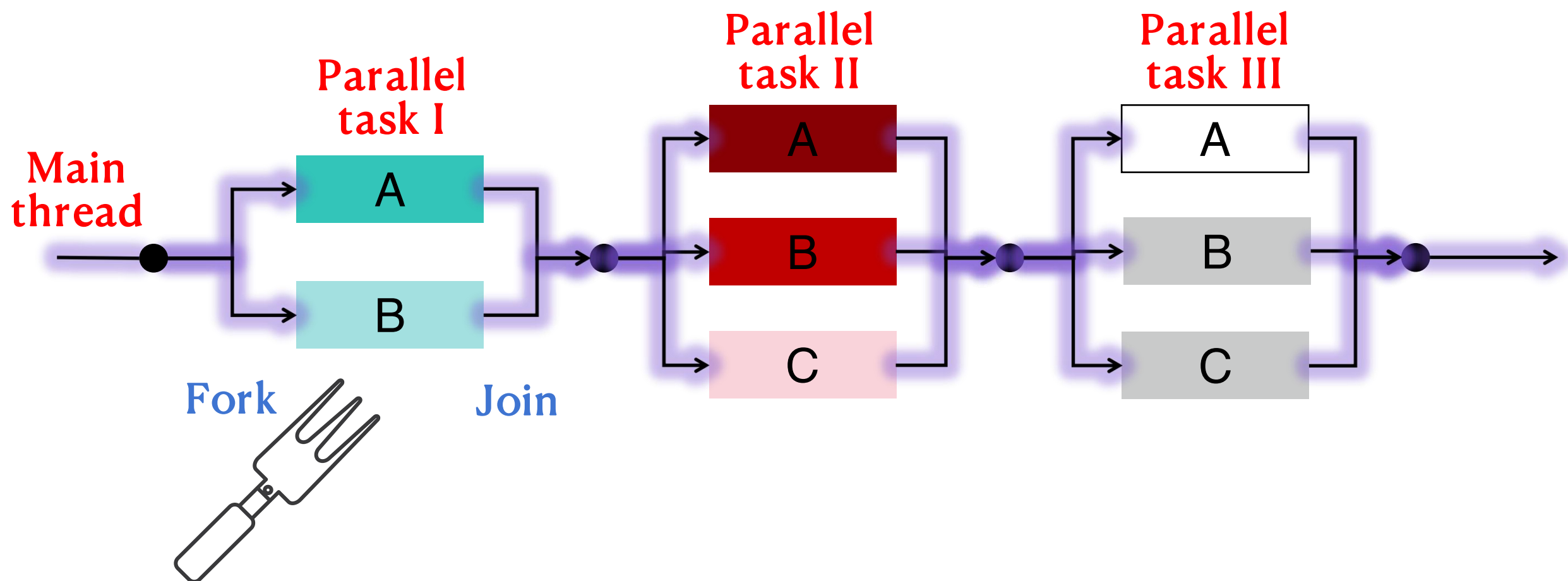
- Threads (short for threads of execution) is a single stream of instructions.
- Each thread has:
  - Its own registers (including stack pointer)
  - Its own program counter (PC)
  - Shared memory (heap, global variables) with other threads
- Each processor provides one (or more) hardware threads (through **multi-core** or **single-core multithreading**, later) that actively execute instructions
- Within a given program's process, threads can run concurrently.
- Operating system (OS) multiplexes multiple **software** threads onto the available hardware threads



**Software threads**

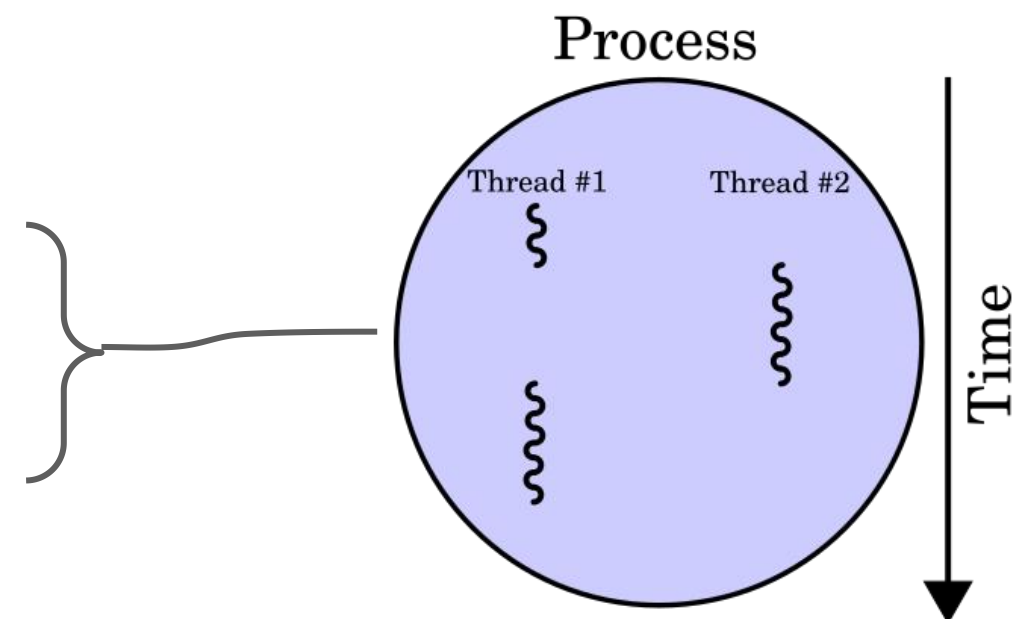
# Fork-Join Model

- Fork-Join model: A program / process can split, or fork itself into separate threads, which can (in theory) execute concurrently.
  - Main thread executes sequentially until first parallel task region.
  - Fork: Main thread then creates a team of parallel subthreads.
  - Join: When subthreads complete their parallel task region, they synchronize and terminate, leaving only the main thread.



# Thread-Level Parallelism & MIMD

- MIMD: Multiple Instruction, Multiple Data
  - Examples: Multicore systems, compute clusters, etc.
- A program / process can split, or fork itself into separate threads, which can (in theory) execute simultaneously.
- In hardware:
  - Single core: Multiple threads execute instructions on a single core, concurrently (time-multiplexing, giving the illusion of many active threads)
  - Multicore: Each thread executes on a separate core, simultaneously/in parallel
  - Can combine the above two...(more later)



A process with two threads of execution, running on a single processor [\[wiki\]](#)

Multithreaded programs can run on both SISD (time-shared) and MIMD systems (in parallel).

# Operating System (OS) Threads

- The operating system, or OS, is responsible for (among other tasks) managing which threads get run on which CPUs.
- On most modern computers, number of active threads  $\gg$  number of available cores, so most threads are idle at any given time;
- Context switches: The OS can choose whichever threads it wants to run, and change threads at any time;
  - Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory;
  - Threads from the same program share memory;
  - Can make a different software thread active by loading its registers into a hardware thread's registers and jumping to its saved PC, e.g., if one thread is blocked waiting for network access or user input.



# “Hello, world!” to OpenMP

- OpenMP is a language extension used for multi-threaded, shared-memory parallelism
  - Generally follows the fork-join model
  - “Open Multi-Processing”
- Portable & standardized
- Easy to compile
  - `#include <omp.h>`
  - Use pragma, e.g., `#pragma omp parallel`
  - `cc -fopenmp name.c`
- Key ideas:
  - Shared vs. private variables
  - OpenMP directives for:
    - Parallelization and work sharing
    - Synchronization



# “Hello, world!” to OpenMP

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

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

```
int main() {
```

```
    /* Fork team of threads with private variable tid */
```

```
    #pragma omp parallel
```

```
{
```

```
    int tid = omp_get_thread_num(); /* get thread id */
```

```
    printf("Hello World from thread = %d\n", tid);
```

```
    /* Only main thread does this */
```

```
    if (tid == 0) {
```

```
        printf("Number of threads = %d\n",  
              omp_get_num_threads());
```

```
    }
```

```
} /* All threads join main and terminate */
```

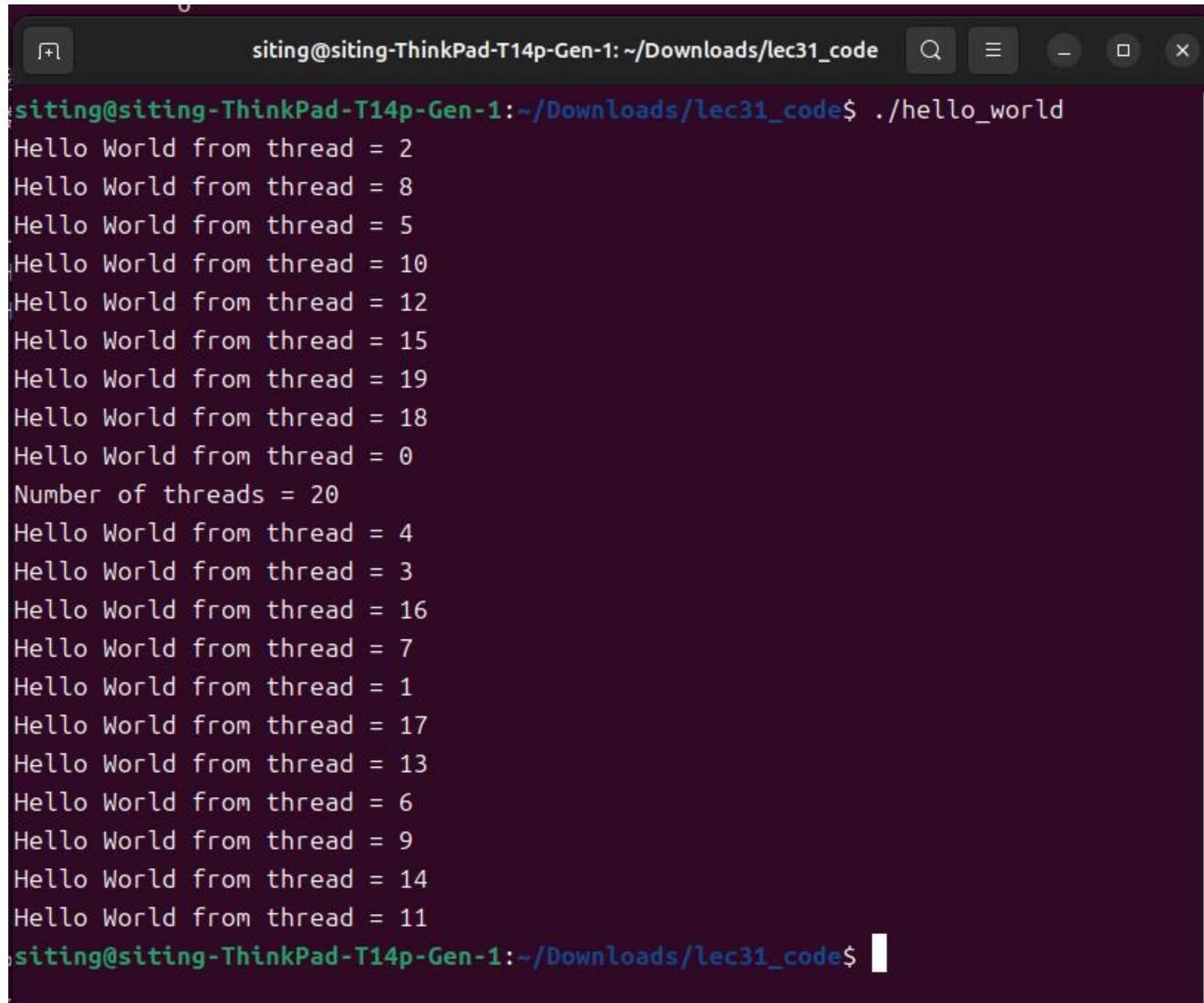
```
return 0;
```

```
}
```

Pragmas are a preprocessor mechanism C provides for language extensions.

This is annoying, but curly brace **MUST** go on separate line from #pragma

# “Hello, world!” to OpenMP

A terminal window with a dark background and light-colored text. The window title bar shows the user 'siting' on a 'siting-ThinkPad-T14p-Gen-1' machine, with the current directory being '~/Downloads/lec31\_code'. The terminal shows the execution of a program that prints 'Hello World from thread = X' for 20 different thread IDs (0-19) in a non-sequential order, demonstrating parallel execution. The output lines are: 'Hello World from thread = 2', 'Hello World from thread = 8', 'Hello World from thread = 5', 'Hello World from thread = 10', 'Hello World from thread = 12', 'Hello World from thread = 15', 'Hello World from thread = 19', 'Hello World from thread = 18', 'Hello World from thread = 0', 'Number of threads = 20', 'Hello World from thread = 4', 'Hello World from thread = 3', 'Hello World from thread = 16', 'Hello World from thread = 7', 'Hello World from thread = 1', 'Hello World from thread = 17', 'Hello World from thread = 13', 'Hello World from thread = 6', 'Hello World from thread = 9', 'Hello World from thread = 14', 'Hello World from thread = 11'. The prompt is ready for the next command.

```
siting@siting-ThinkPad-T14p-Gen-1: ~/Downloads/lec31_code
siting@siting-ThinkPad-T14p-Gen-1:~/Downloads/lec31_code$ ./hello_world
Hello World from thread = 2
Hello World from thread = 8
Hello World from thread = 5
Hello World from thread = 10
Hello World from thread = 12
Hello World from thread = 15
Hello World from thread = 19
Hello World from thread = 18
Hello World from thread = 0
Number of threads = 20
Hello World from thread = 4
Hello World from thread = 3
Hello World from thread = 16
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 17
Hello World from thread = 13
Hello World from thread = 6
Hello World from thread = 9
Hello World from thread = 14
Hello World from thread = 11
siting@siting-ThinkPad-T14p-Gen-1:~/Downloads/lec31_code$
```

# OpenMP: Threads

- OpenMP creates as many threads as specified in the environment variable `OMP_NUM_THREADS`.
  - Set this variable to max number of threads you want to use
  - Generally, default: ( $\#$  physical cores) \* ( $\#$  threads/core). Use `lscpu` command to obtain the numbers (e.g. 6 physical cores \* 2 threads/core = **12 threads**)
- OpenMP threads are OS (software) threads, which are then multiplexed onto available hardware threads.

OpenMP Intrinsic	Description
<code>omp_set_num_threads(x);</code>	Set number of threads to x.
<code>num_th = omp_get_num_threads();</code>	Get number of threads.
<code>th_ID = omp_get_thread_num();</code>	Get Thread ID number.



# lscpu

```
siting@siting-ThinkPad-T14p-Gen-1: ~/Downloads
siting@siting-ThinkPad-T14p-Gen-1:~/Downloads$ lscpu
架构:                x86_64
CPU 运行模式:        32-bit, 64-bit
Address sizes:        46 bits physical, 48 bits virtual
字节序:              Little Endian
CPU:                  20
  在线 CPU 列表:      0-19
厂商 ID:              GenuineIntel
型号名称:             13th Gen Intel(R) Core(TM) i9-13900H
  CPU 系列:           6
  型号:               186
  每个核的线程数:     2
  每个座的核数:       14
  座:                 1
  步进:               2
CPU(s) scaling MHz:   39%
CPU 最大 MHz:         2600.0000
CPU 最小 MHz:         400.0000
BogoMIPS:             5990.40
标记:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx f
xsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monit
or ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb ssbd i
brs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1
avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb intel_pt sha_ni xsaveopt xsavec xg
etbv1 xsaves split_lock_detect user_shstk avx_vnni dtherm ida arat pln pts hwp hwp_notify hwp_
act_window hwp_epp hwp_pkg_req hfi vnmi umip pku ospke waitpkg gfni vaes vpclmulqdq tme rdpid
movdiri movdir64b fsrm md_clear serialize pconfig arch_lbr ibt flush_l1d arch_capabilities
```

# OpenMP: Shared/Private Variables

- **Shared variables:** all threads read/write the same variable.
  - Variable declared outside of parallel region
  - Heap-allocated variables
  - Static variables
- **Private variables:** Each thread has its own copy of the variable.
  - Variables declared inside parallel region (recall separate stack frames)

```
int var1, var2;  
char *var3 = malloc(...);  
#pragma omp parallel private(var2)  
{  
    int var4;  
    // var1 shared (default)  
    // var2 private  
    // var3 shared (heap)  
    // var4 private (thread's stack)  
    ...  
}
```

# Example

```
#include <stdio.h>
#include <omp.h>
int main() {
    /* Fork team of threads with private variable tid */
    #pragma omp parallel
    {
        int tid = omp_get_thread_num(); /* get thread id */
        printf("Hello World from thread = %d\n", tid);
        /* Only main thread does this */
        if (tid == 0) {
            printf("Number of threads = %d\n",
                omp_get_num_threads());
        }
    } /* All threads join main and terminate */
    return 0;
}
```

Private  
variable

Parallel  
region  
executed  
by each  
subthread  
(with  
OpenMP  
API)

# Parallelizing Loop Work

- Problem: You have to do some work over an array of  $2^{27}$  numbers, with 8 people. How would you split the work?
- Assumptions
  - We need to decide before running the code!
  - Each element of the array is independent, so the tasks can be done in any order
  - The threads are about equally fast, so we want to assign each of them  $\sim 11$  million numbers



# Which Runs Fastest?

```
/* A. */
#pragma omp parallel
{
    for (int i = 0;
         i < LENGTH;
         i++) {
        arr[i] = ...;
    }
}
```

Duplicates  
work

```
/* B. */
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    int thread_start = tid*LENGTH/num_threads;
    int thread_end = (tid+1)*LENGTH/num_threads;
    for (int i = thread_start;
         i < thread_end; i++) {
        arr[i] = ...;
    }
}
```

“Chunks” array  
sections

```
/* C. */
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int num_threads =
    omp_get_num_threads();
    for (int i = tid;
         i < LENGTH;
         i += num_threads) {
        arr[i] = ...;
    }
}
```

“Interweaves”  
array access  
between threads

```
/* D. */
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0;
         i < LENGTH;
         i++) {
        arr[i] = ...;
    }
}
```

Like C, but  
planned via  
OpenMP

# OpenMP Work-sharing for Syntax

- `#pragma omp for`
  - must be written inside an already existing parallel segment.
  - If a parallel segment consists only of one for loop, we can combine the two declarations with `#pragma omp parallel for`.
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e. No `break`, `return`, `exit`, `goto`

```
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0;
        i < LENGTH;
        i++) {
        arr[i] = ...;
    }
}
```



```
#pragma omp parallel for
for (int i = 0;
    i < LENGTH;
    i++) {
    arr[i] = ...;
}
```

# Non-deterministic Outcomes

- Suppose we run the below code on 4 threads.

```
int x = 0;  
#pragma omp parallel  
{  
    x = x + 1;  
}
```

What are possible values of x after running this code? Select all that apply.

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4
- F. 5 or more

# Data Race

- Two memory accesses form a **data race** if:
  - They are from different threads to the same location
  - At least one is a write, and
  - They occur one after another.
- Recall thread model: **shared memory**.
  - For a given thread, these two operations don't necessarily happen together...! Thread scheduling is *non-deterministic*.
    - Read current value of x
    - Write new value of x
- **Not a data hazard!**
  - Data hazard: Sequential instructions have data dependencies during concurrent execution (instruction-level parallelism via pipelining).
  - Here, even with no ILP, can have nondeterministic results. This results from lack of synchronization on which thread accesses memory first.

# Data Race: RISC-V Instructions

- Instructions from different threads have their execution on the CPU interleaved.
- Assume (for ease of analysis):
  - Non-pipelined, single-cycle datapath → all **atomic instructions**, meaning that no nothing else interposes itself while the instruction is executing.

```
// four threads  
int x = 0;  
#pragma omp parallel  
{  
    x = x + 1;  
}
```



```
lw t0 0(sp)  
addi t0 t0 1  
sw t0 0(sp)
```



```
load  
addi  
store
```

```
load  
addi  
store
```

```
load  
addi  
store
```

```
load  
addi  
store
```

# Data Race: Case 1

• <b>Grey</b> thread load	read	x: 0	load
• <b>Grey</b> thread store	write	x: 1	addi
• <b>Green</b> thread load	read	x: 1	store
• <b>Green</b> thread store	write	x: 2	load
• <b>Blue</b> thread load	read	x: 2	addi
• <b>Blue</b> thread store	write	x: 3	store
• <b>Red</b> thread load	read	x: 3	load
• <b>Red</b> thread store	write	x: 4	addi
			store

Final value of x: 4

# Data Race: Case 1

• Grey thread load	read	x: 0	load
• Green thread load	read	x: 0	addi
• Blue thread load	read	x: 0	store
• Red thread load	read	x: 0	load
• Grey thread store	write	x: 1	addi
• Green thread store	write	x: 1	store
• Blue thread store	write	x: 1	load
• Red thread store	write	x: 1	addi
			store

Final value of x: 1



# Data Race: Cases ...

- **Many possible permutations!**
  - Cannot go over 4;
  - Cannot go below 1;
- Formally, a multithreaded program is only considered correct if **ANY** interlacing of threads yield the same result.
  - Here, we have an incorrect program!
  - But if each thread works on independent data (no thread accesses same data location another thread wrote to), you can guarantee correctness.

# Summary

- Basics on thread-level parallelism
- One implementation: OpenMP (extension for C/C++ and Fortran)
- Data race and how shall we solve it?