信息科学与技术学院

School of Information Science and Technology

CS 110
Computer Architecture
Thread-Level Paralellism III

Instructors:

Chundong Wang, Siting Liu & Yuan Xiao

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2025/5/13

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 today's lecture (Thread-level parallelism).
- Project 2.2 released, ddl approaching, May 19th.
- Project 3 released. Speed Competition! ddl May 29th.
- HW 7 released, ddl May 23rd.
- Lab 13 released, related to Project 3. Prepare in advance!
  - To check May 13th, 15th & 19th;
- Discussion May 16th & 19th on *Profiling*.

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

Today’s Lecture

Pragma and Scope - Review

- Basic OpenMP construct for parallelization:

```
#pragma omp parallel
{
        /* code goes here */
}
```

  - *Each* thread runs a copy of code within the block
  - Thread scheduling is *non-deterministic*
- To make private, need to declare with pragma:

```
#pragma omp parallel private (x)
```

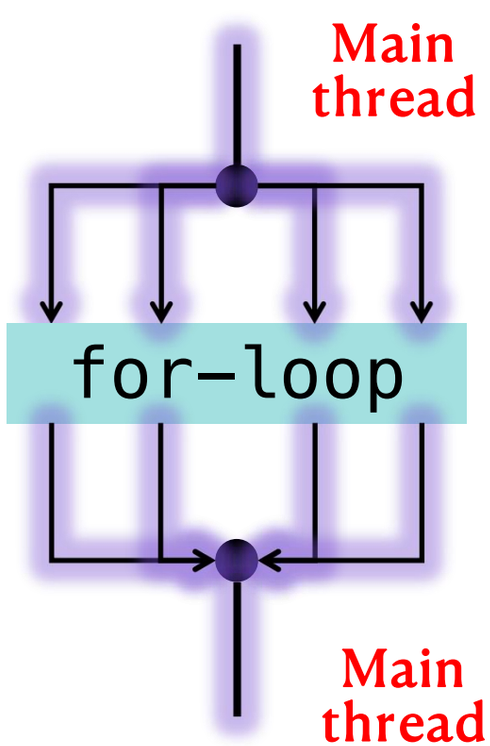
- Automatic work-sharing (for-loop):

```
#pragma omp parallel
{
#pragma omp for
    /* for loop */
}
```

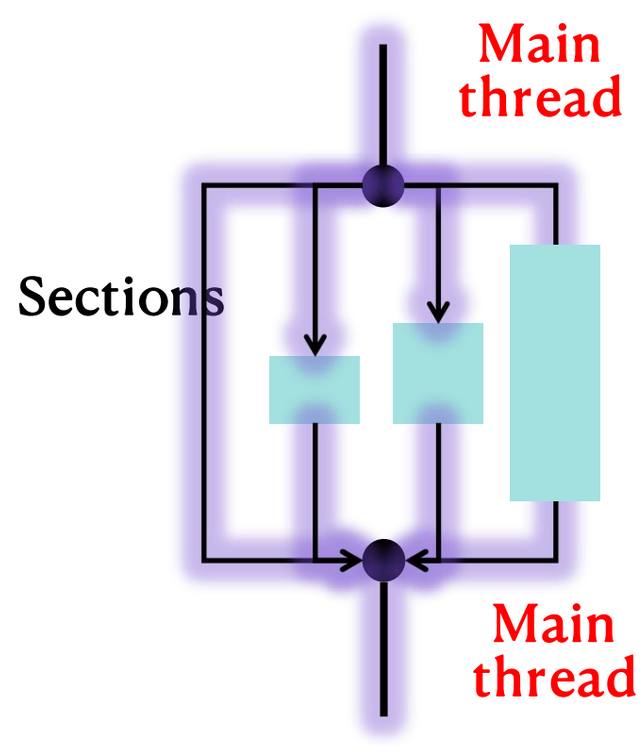
  - Can be shortened as: `#pragma omp parallel for`
  - Implicit “barrier” synchronization at end of the for loop
  - `for-loop` index private for each thread, variables declared outside `for-loop` shared

OpenMP Directives (Work-sharing)

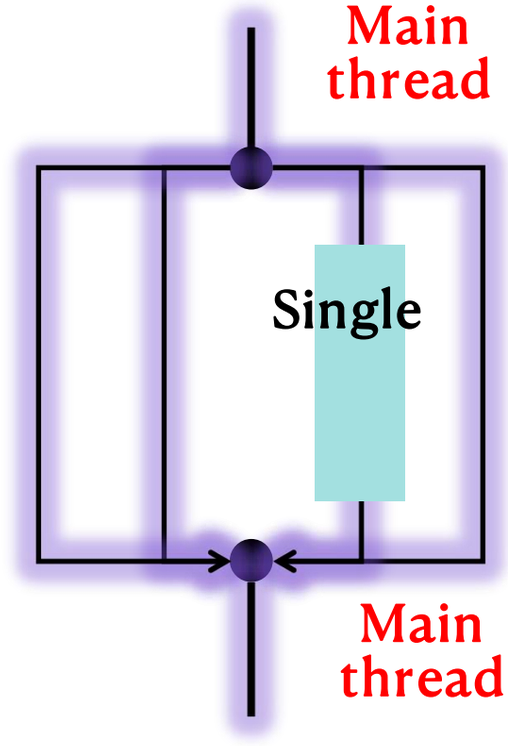
- Defined within a `parallel` section



Shares iterations of a loop across the threads



Each section is executed by a separate thread



Serializes the execution of a thread

OpenMP `section` usage:
https://www.cnblogs.com/wzyj/p/4501348.html

OpenMP Timing

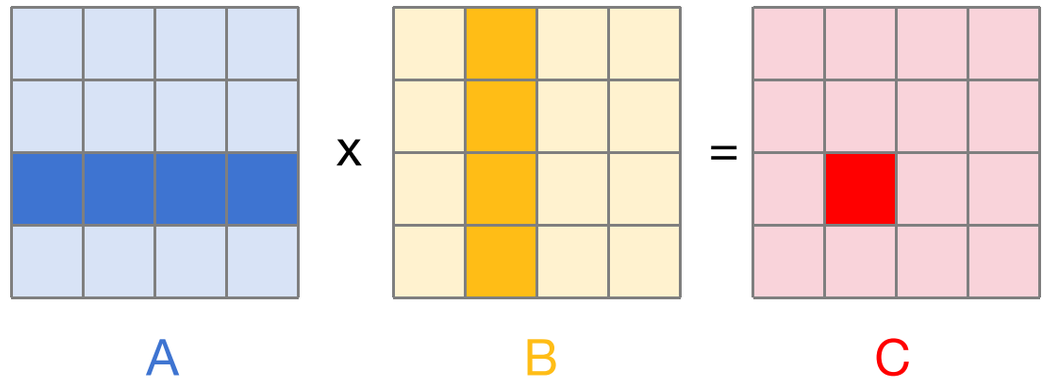
- Elapsed wall clock time:

```
double omp_get_wtime(void);
```

  - Returns elapsed wall clock time in seconds;
  - Time is measured per thread, no guarantee can be made that two distinct threads measure the same time;
  - Time is measured from “some time in the past,” so subtract results of two calls to `omp_get_wtime` to get elapsed time within a parallel section;

OpenMP Matrix Multiplication Example

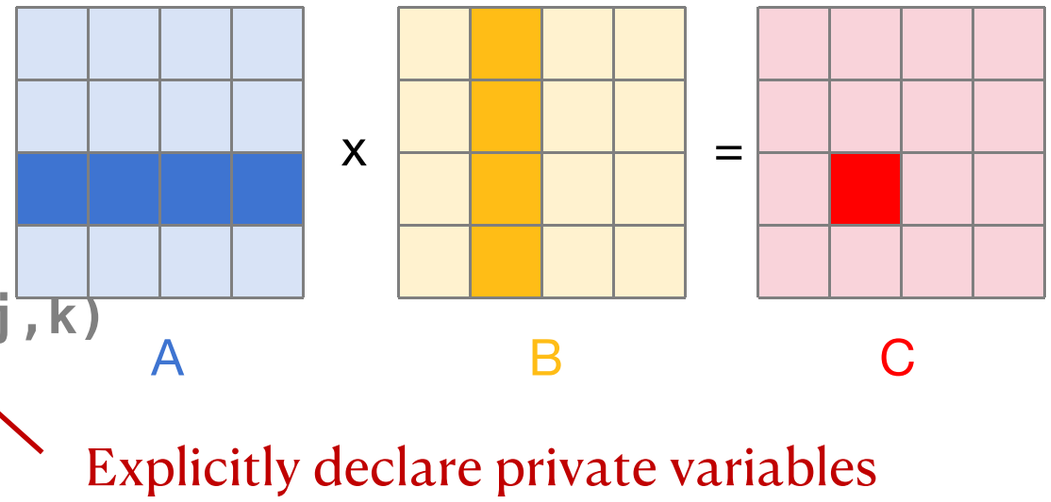
```
double dgemm_scalar(int N, double *A,
                    double *B, double *C) {
  double start_time = omp_get_wtime();
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      double Cij = 0;
      for (int k = 0; k < N; k++) {
        Cij += A[i+k*N] * B[k+j*N];
      }
      C[i+j*N] = Cij;
    }
  }
  double run_time = omp_get_wtime()-start_time;
  return run_time;
}
```



Outer loop spread across x threads;
inner loops inside a single thread

OpenMP Matrix Multiplication Example

```
double dgemm_scalar(int N, double *A,
                    double *B, double *C) {
  double start_time = omp_get_wtime();
  double Cij;
  int i,j,k;
  #pragma omp parallel for private (Cij,i,j,k)
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      Cij = 0;
      for (k = 0; k < N; k++) {
        Cij += A[i+k*N] * B[k+j*N];
      }
      C[i+j*N] = Cij;
    }
  }
  double run_time = omp_get_wtime()-start_time;
  return run_time;
}
```



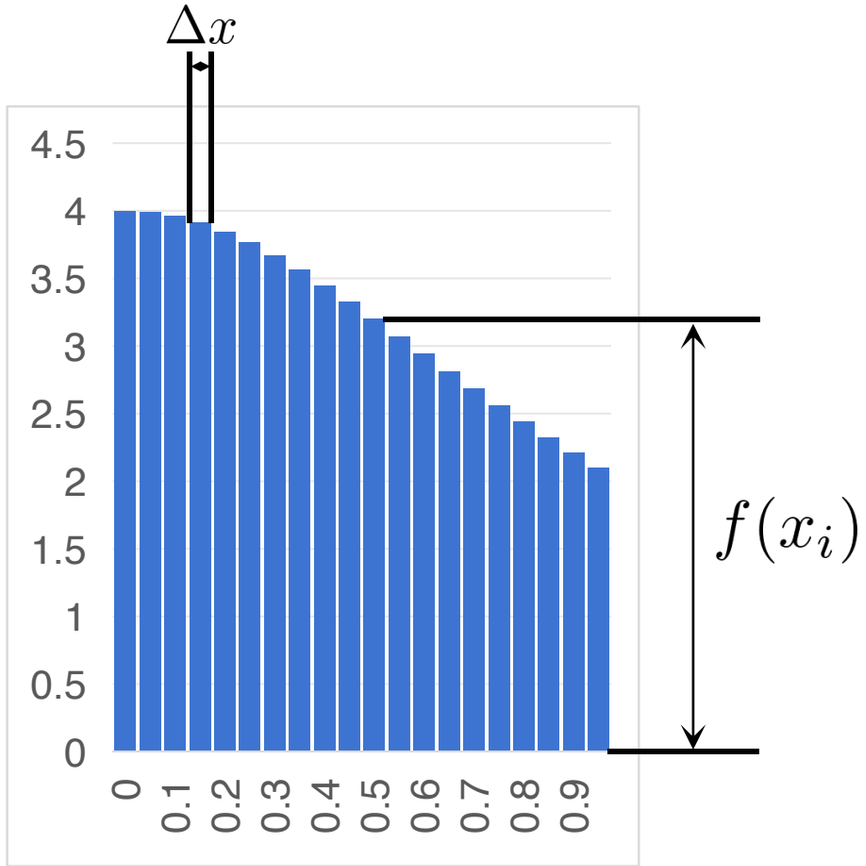
Notes on Matrix Multiplication Example

- More performance optimizations available:
  - Higher compiler optimization (-O2, -O3) to reduce number of instructions executed
  - Cache blocking to improve memory performance
  - Using SIMD SSE instructions to raise floating point computation rate (using DLP)

Example: Calculating π

$$\int_0^1 f(x)dx = \int_0^1 \frac{4.0}{(1+x^2)}dx \approx \pi$$

- Can be approximated by numerical integration (Reimann sum)

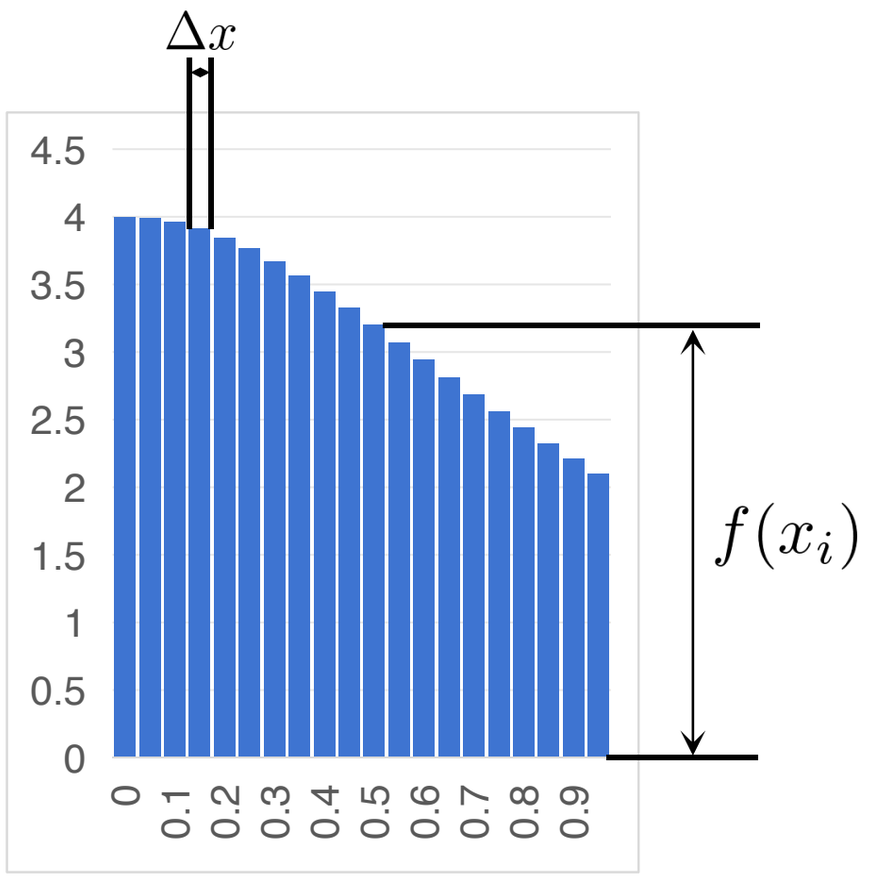


$$\sum f(x_i)\Delta x \approx \pi$$

Example: Calculating π

- Serial version

$$\sum f(x_i)\Delta x \approx \pi$$




```
#include <stdio.h>
void main(){
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double sum = 0.0;
  for (int i=0; i<num_steps; i++){
      double x = (i+0.5)*step;
      sum += 4.0*step/(1.0+x*x);
  }
  printf("pi=%6.12f\n",sum);
}
```

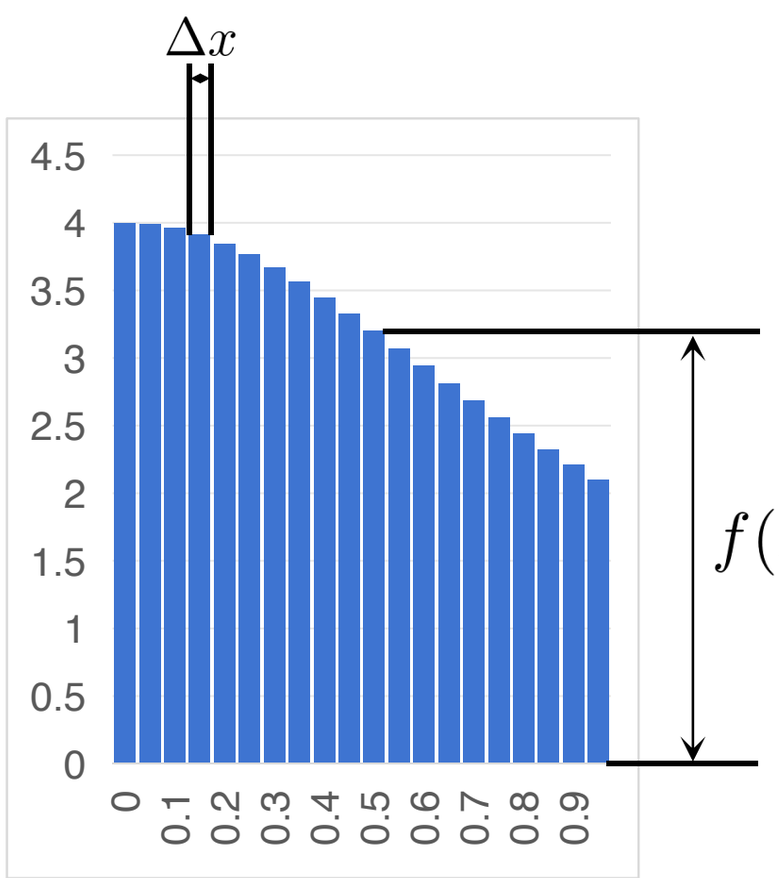
```
pi=3.141800986893.
```

- Resembles π, but not very accurate
- Let's increase **num_steps** and parallelize

Example: Calculating π

- OpenMP version 1

$$\sum f(x_i)\Delta x \approx \pi$$



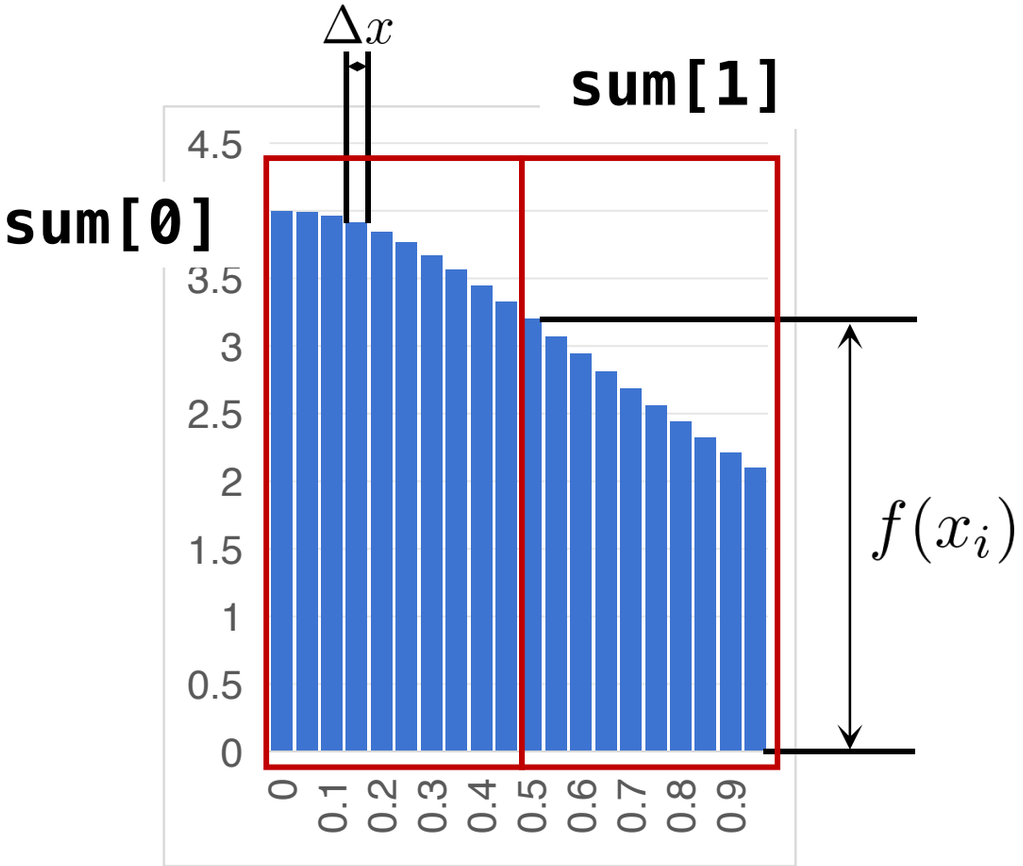

```
#include <stdio.h>
#include <omp.h>
void main(){
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double sum = 0.0;
  #pragma omp parallel for
  for (int i=0; i<num_steps; i++){
      double x = (i+0.5)*step;
      sum += 4.0*step/(1.0+x*x);
  }
  printf("pi=%6.12f\n",sum);
}
```

- Problem: each thread needs access to the shared variable **sum**
- Code demands synchronization…

Example: Calculating π

- OpenMP version 2

$$\sum f(x_i)\Delta x \approx \pi$$



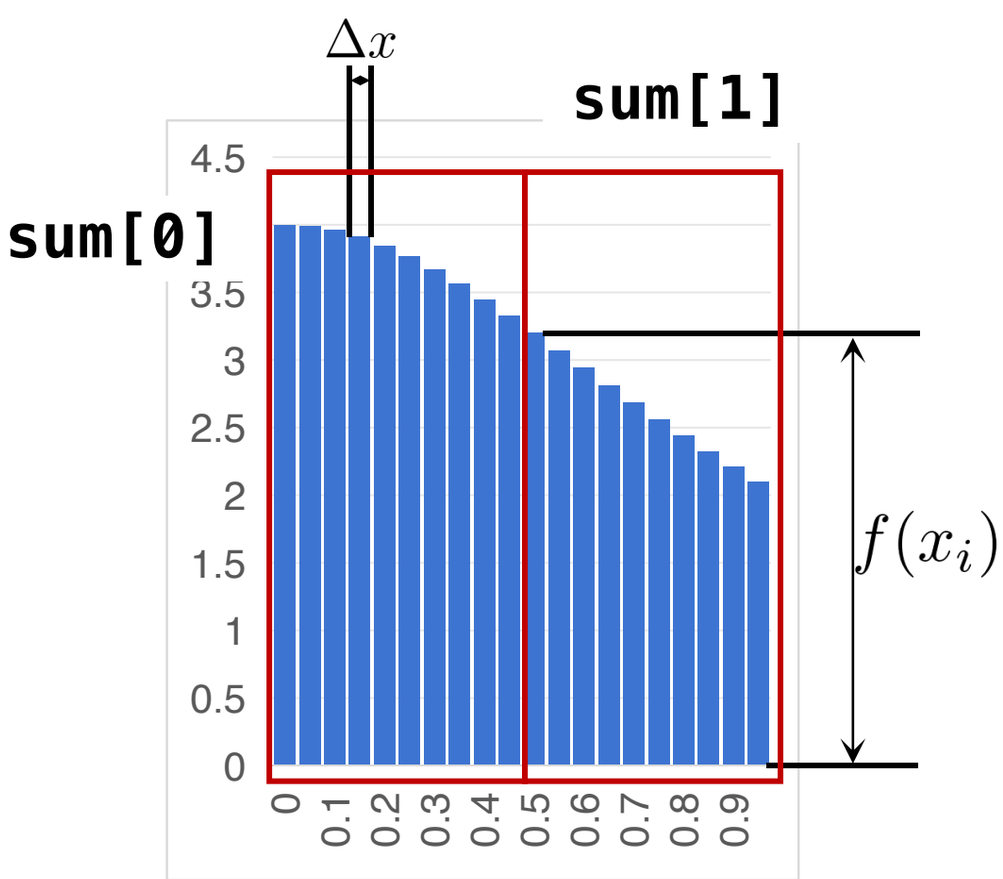
Divide & conquer

1. Compute **sum[0]** and **sum[1]** in parallel
2. Compute **sum=sum[0]+sum[1]** sequentially

Example: Calculating π

- OpenMP version 2

$$\sum f(x_i)\Delta x \approx \pi$$




```
#include <stdio.h>
#include <omp.h>
void main(){
  const int NUM_THREADS = 4;
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double sum[NUM_THREADS];
  for (int i=0;i<NUM_THREADS;i++)
    sum[i]=0;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    for (int i=id;i<num_steps;i+=NUM_THREADS){
      double x = (i+0.5)*step;
      sum[id] += 4.0*step/(1.0+x*x);
    }
  }
  double pi=0;
  for (int i=0;i<NUM_THREADS;i++)
    pi += sum[i];
  printf("pi=%6.12f\n",pi);
}
```

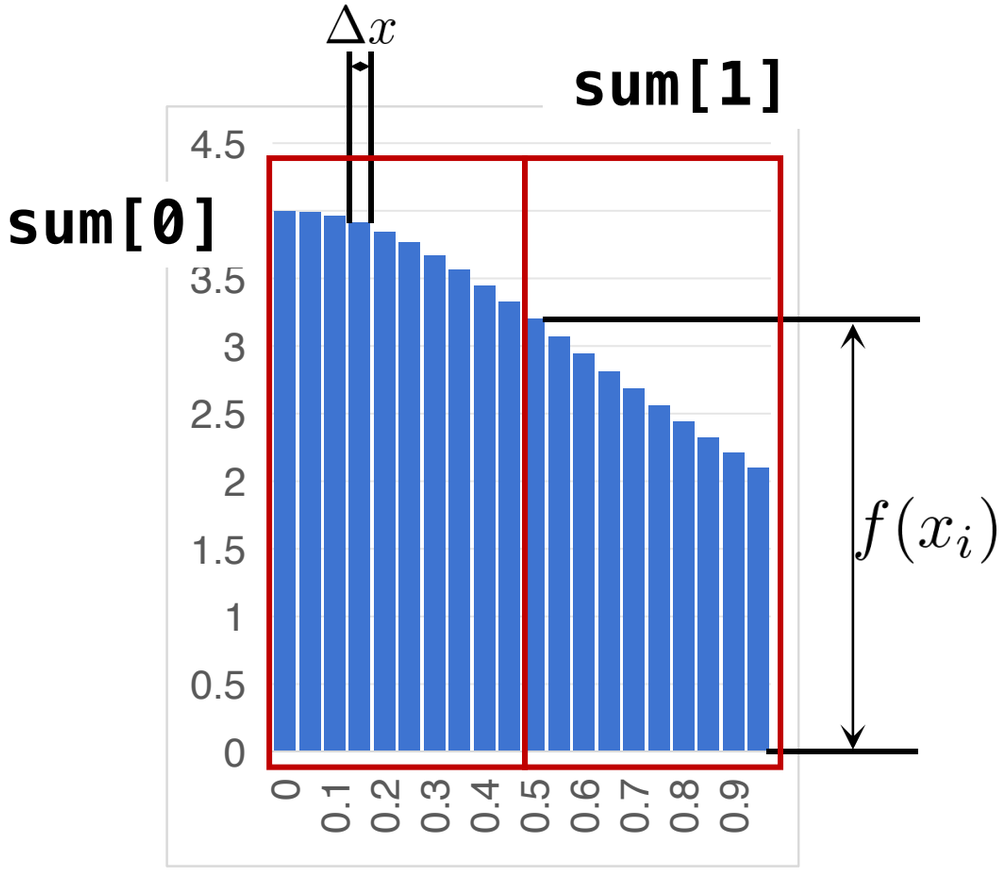
Demo

Increase `num_step` **to obtain higher accuracy.**

Question

- OpenMP version 2

$$\sum f(x_i)\Delta x \approx \pi$$




```
#include <stdio.h>
#include <omp.h>
void main(){
  const int NUM_THREADS = 4;
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double pi=0;
  double sum[NUM_THREADS];
  for (int i=0;i<NUM_THREADS;i++)
    sum[i]=0;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    for (int i=id;i<num_steps;i+=NUM_THREADS){
      double x = (i+0.5)*step;
      sum[id] += 4.0*step/(1.0+x*x);
    }
    pi += sum[id];
  }
  printf("pi=%6.12f\n",pi);
}
```

Parallelize the final summation?

OpenMP Reduction

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp parallel for private (sum)
for (i = 0; i <= MAX ; i++)
          sum += A[i];
avg = sum/MAX;  // bug
```

- *Problem is that we really want sum over all threads!*

- *Reduction*: specifies that, 1 or more variables that are private to each thread, are subject of reduction operation at end of parallel region:

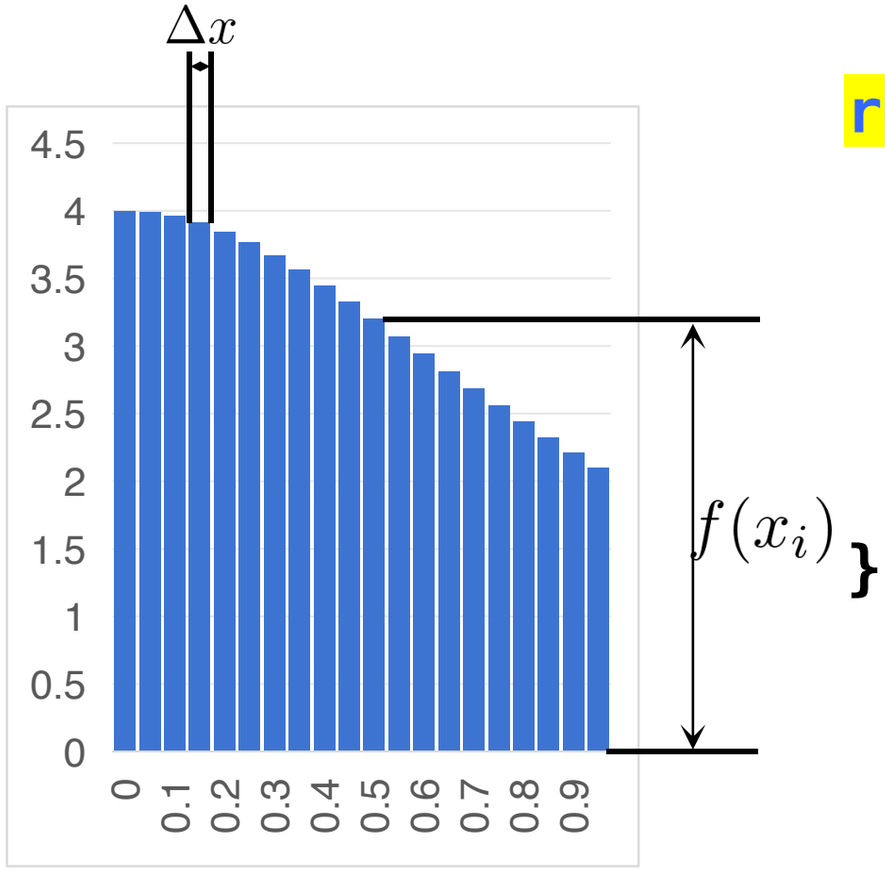
  reduction(operation:var) where

  - *Operation*: operator to perform on the variables (var) at the end of the parallel region: +, *, -, &, ^, |, &&, or ||.
  - *Var*: One or more variables on which to perform scalar reduction.

OpenMP Reduction Example

- OpenMP reduction

$$\sum f(x_i)\Delta x \approx \pi$$




```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000;
double step;
void main (){
  int i; double x, pi, sum = 0.0;
  step = 1.0 / (double)num_steps;
  #pragma omp parallel for private(x)
reduction(+:sum)
  for (i=1; i<= num_steps; i++){
    x = (i - 0.5) * step;
    sum = sum + 4.0 / (1.0+x*x);
    }
  pi = sum * step;
  printf("pi = %6.12f\n", pi);
}
```

OpenMP Other Usage

- **`#pragma omp single`**
  - Code block executed by one thread only;
  - Other threads will wait;
  - Useful for thread-unsafe code & I/O operations.
- **`#pragma omp master`**
  - Only the master threads executes instructions in the block.
  - There is no implicit barrier, so other threads will not wait for master to finish
- ... ...
- Learn more at https://www.openmp.org/specifications/

Summary

- Thread-level parallelism (TLP)
  - Fork-join model
  - Software/hardware threads, context switching, etc.
  - OpenMP as simple parallel extension to C
    - Pragma as compiler directives
    - Parallel for, private, reductions, barrier, critical …
  - Synchronization and atomic instructions in RISC-V
  - Much we didn’t cover – including other synchronization mechanisms, TLP models, to be explored in advanced courses

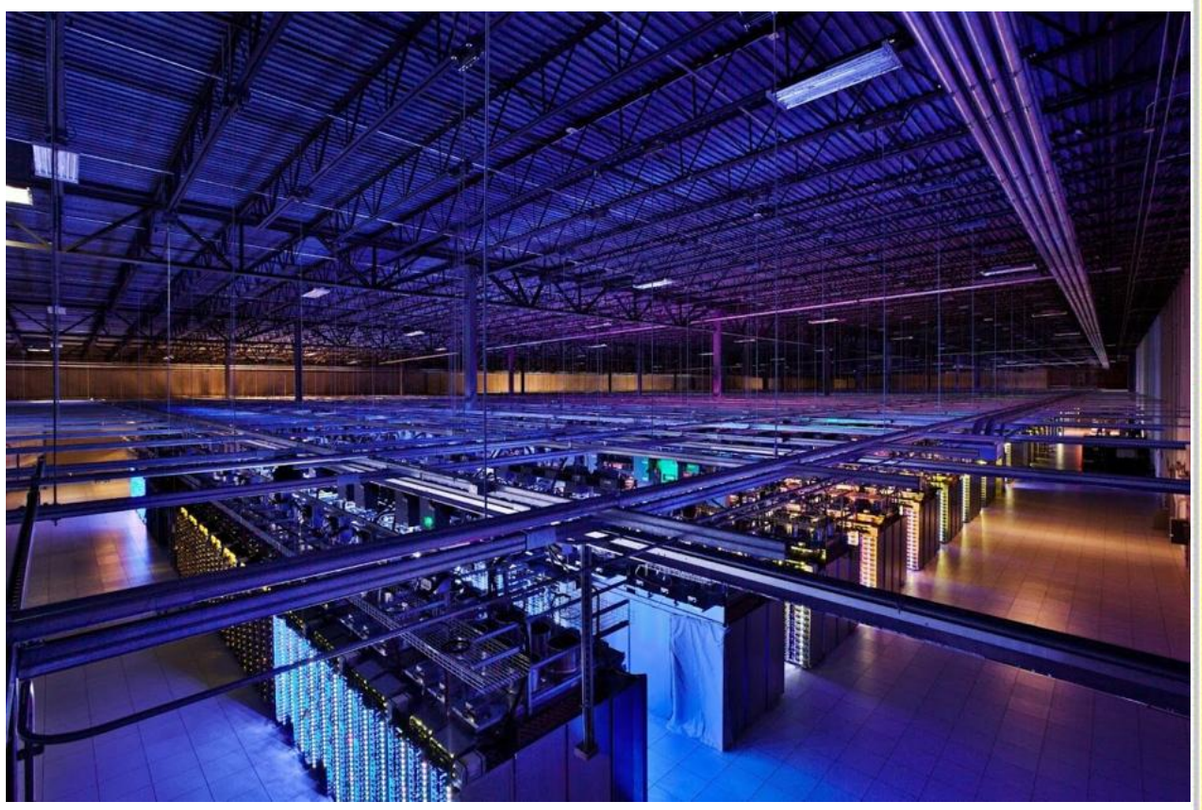
信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
Hardware Multithreading

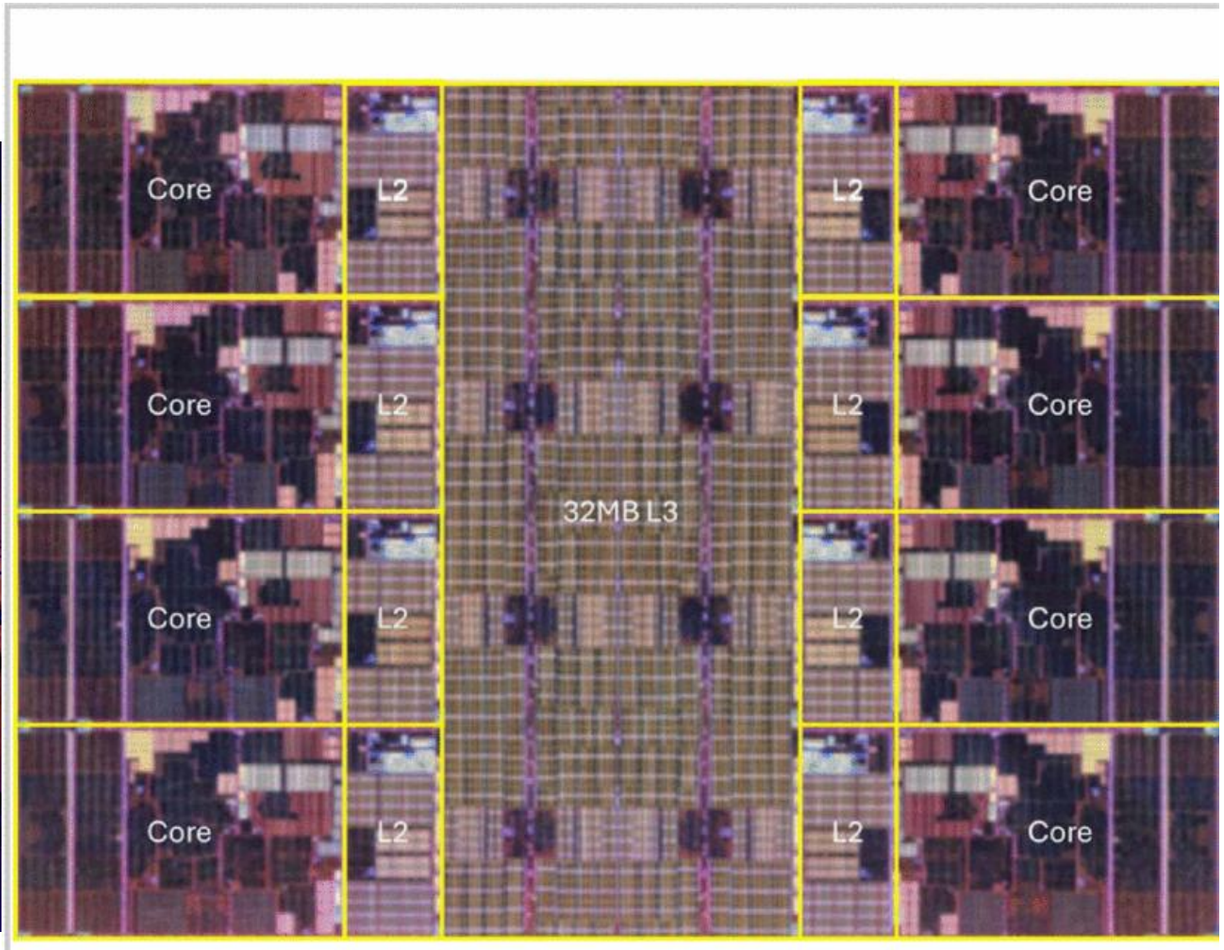
Instructors:
Chundong Wang, Siting Liu & Yuan Xiao
Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2025/5/13

Parallel Computer Architecture



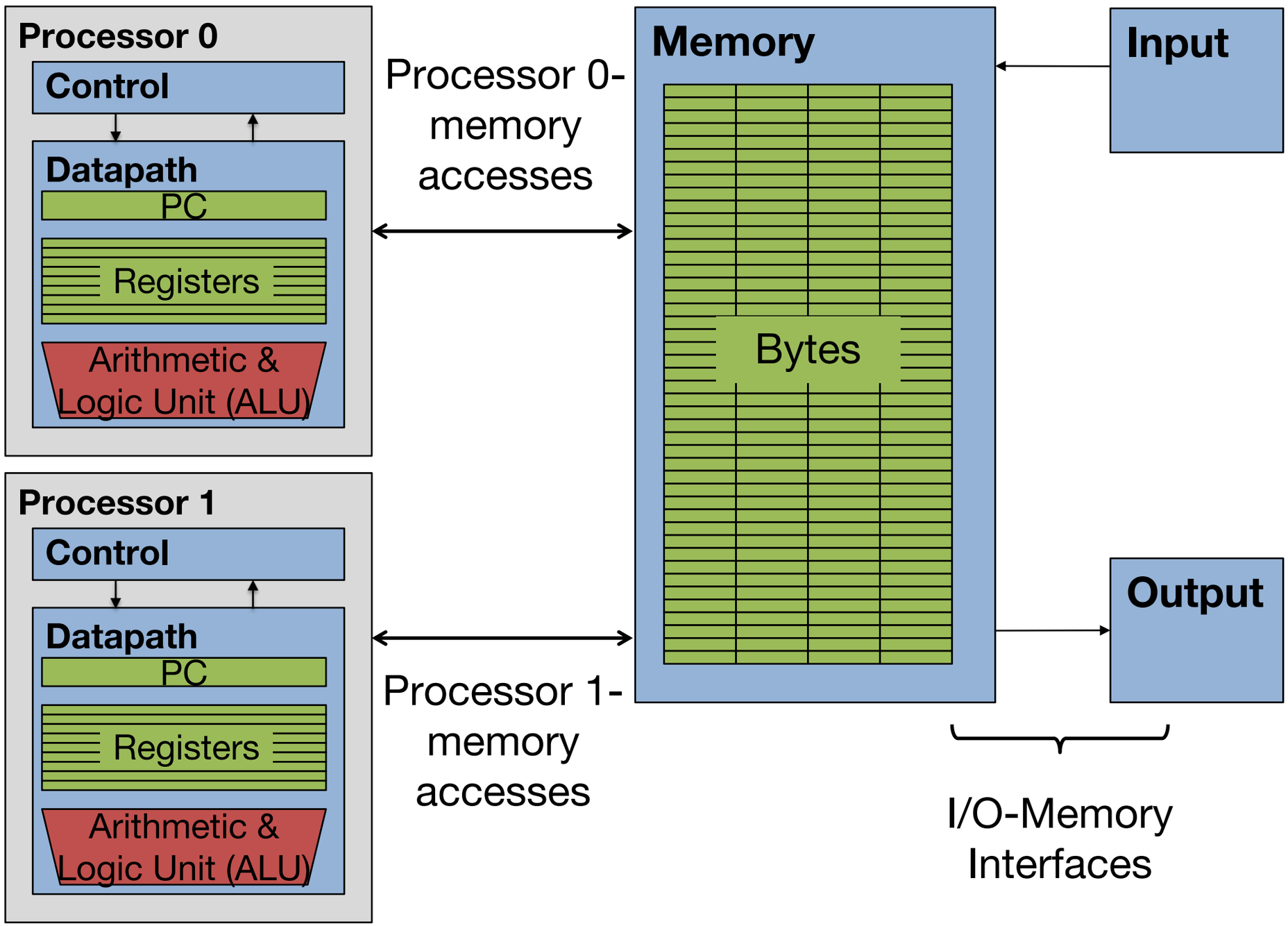
Warehouse-scale computers



AMD Zen 5 Die photo [ISSCC 2025]

Multicore Processor

- A multicore processor contains multiple processors (“cores”) in a single integrated circuit.



Multicore Processor

- A multicore processor contains multiple processors (“cores”) in a single integrated circuit.
- Execution Model:
  - Each processor executes an independent stream of instructions.
    - Also separate: high-level caches (e.g., L1 & L2 cache)
  - All processors access the same shared memory.
    - Shared: DRAM and perhaps L3 cache
    - Communicate via shared memory by storing to/loading from common locations.

Multicore Processor Use Cases

- A multicore processor contains multiple processors (“cores”) in a single integrated circuit.

- Parallel-processing program
  - Improve the runtime of a single program that has been specially crafted to run on a multiprocessor
  - Example: Multithreaded program
- Process-level parallelism (i.e., job-level parallelism, not covered in CS 110):
  - Deliver high throughput for independent jobs
  - Example: Your operating system and different programs

Key Design Questions

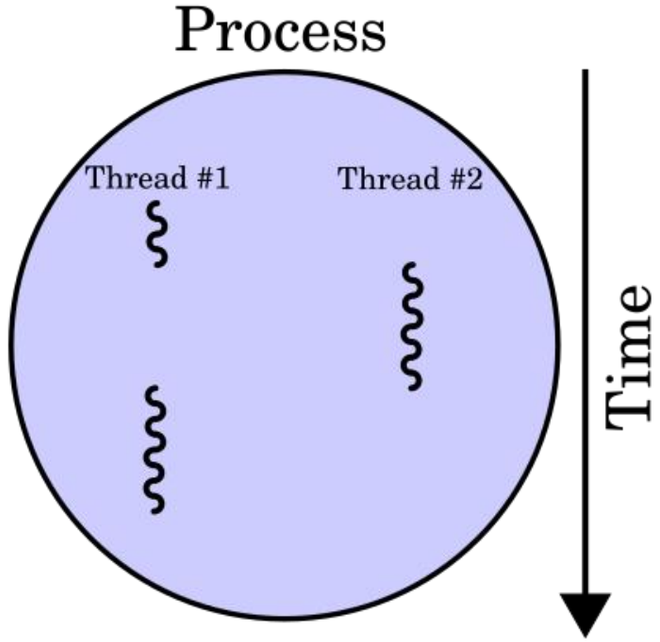
- How many processors (or cores) should be supported in this multiprocessor?
  - Depends on the target workload!
  - Most systems: Multiple “best available single core within constraints”
  - Power-critical systems (e.g., phones): “some of the best available single cores” and “some of the most power efficient single cores”

- How do different processors coordinate/communicate?
  - Shared variables in memory and load/store instructions
  - Coordinated access to shared data through synchronization primitives (e.g., locks) that restrict access to one processor at a time

- How do different processors (cores) share data?
  - Via shared-memory multiprocessor (SMP) (later this lecture)

Hardware vs. Software Thread Review

- Each core provides one (or more) **hardware threads** that actively execute instructions.
- The Operating System multiplexes multiple **software threads** onto the available hardware threads.
  - Only those mapped onto hardware threads are executing; **all the others are waiting**.
  - The OS uses **context switching** to give the illusion of many active, concurrently executing threads.
- **Context switching:**
- To remove old SW thread from HW thread:
  - Interrupt execution
  - Save registers to memory, including PC
- To activate new SW thread onto HW thread:
  - Load register values for new thread, including PC
  - Transfer control to new thread

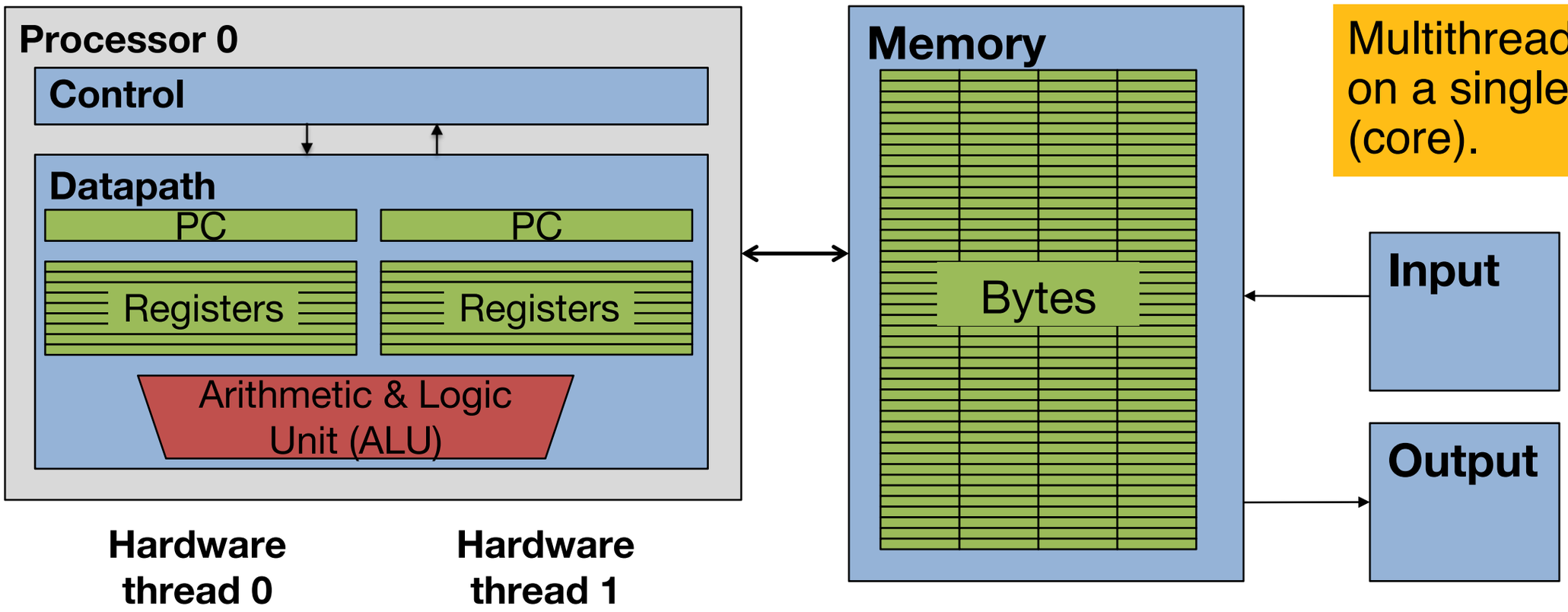


Simultaneous Multithreading (SMT)

- Processor resources are expensive; should not be left idle
  - High memory latency cost on cache miss (~100 cycles)
  - Furthermore, the cost of thread context switch should be much less than cache miss latency!
- HW optimization is to have **redundant hardware** so that not every context switch needs to “save context”
  - Thanks to Moore’s Law, transistors are plenty
  - Add multiple PCs, registers to the same core

Simultaneous Multithreading (SMT)

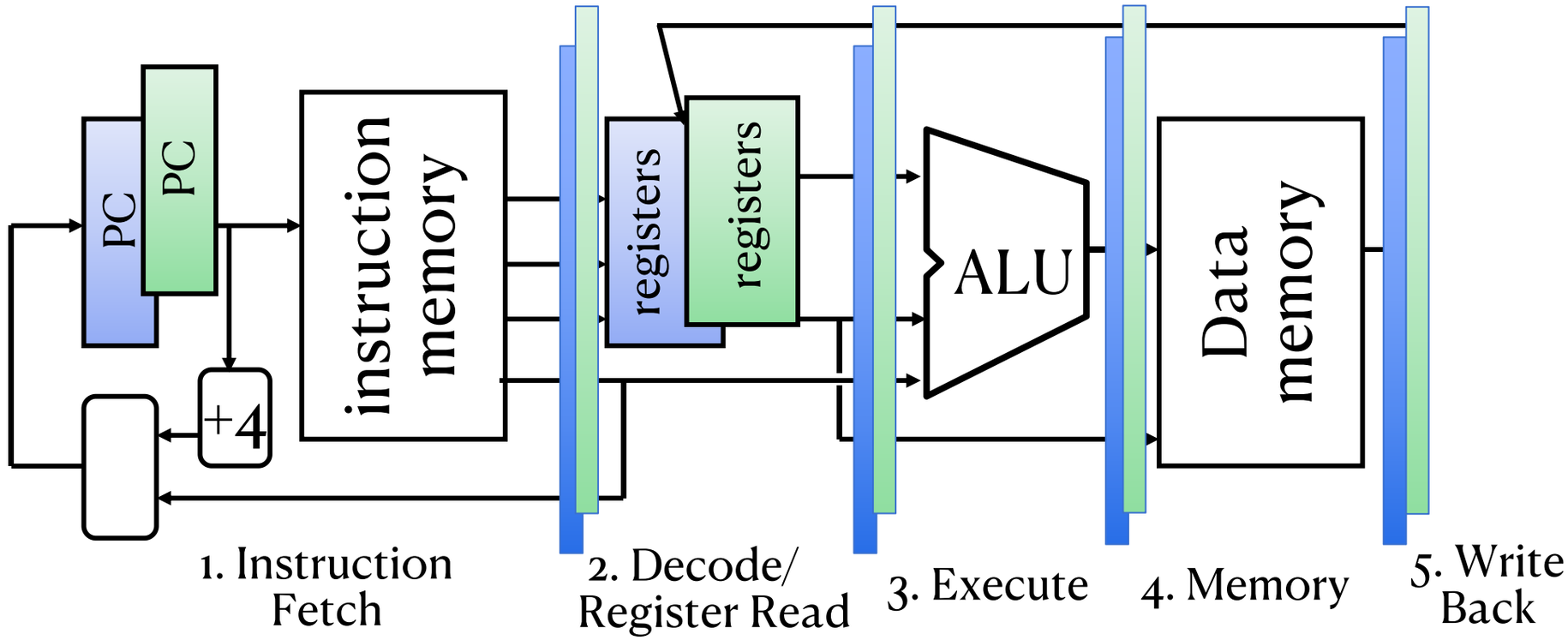
- HW **multithreading** means having multiple thread "active" in the same processor, e.g., by storing thread state (PC, registers).
  - Control logic decides which instruction to issue next
  - Can mix from different threads
- To software, this looks like **multiple processors** (hardware thread 0, hardware thread 1, etc).



Multithreading occurs on a single processor (core).

Simultaneous Multithreading (SMT)

- Simplified implementation



- Use muxes to select which state to use every clock cycle
- Run 2 independent processes
  - No Hazards: registers different; different control flow; different memory; Threads: race condition should be solved by software (e.g., lock sync. …)
- Speedup?
  - No obvious speedup; Complex pipeline: make use of CL blocks in case of unavailable resources (e.g. wait for memory)

More on SMT

- Simultaneous multithreading (SMT) lowers the cost of multithreading by leveraging multiple issue, dynamically scheduled microarchitecture.
  - Superscalar architecture
  - Also called “hyperthreading” in Intel processors
  - Downside: excessive power consumption

Multithreading vs. Multicore

- Modern machines do both:
  - Multiple cores, with multiple threads per core.
- Multithreading (SMT):
  - ~1% more hardware, ~1.10X better performance
  - Shared: integer ALU, floating point units, all caches, memory controller
  - Better utilization from reducing latency of:
    - OS context switches
    - Major stalls like instruction cache misses
- Multiple cores:
  - Duplicate processors entirely
  - ~50% more hardware, ~2X better performance
  - Share: outer caches (e.g., L3 cache), memory controller
  - Better utilization from executing instructions on different processors in parallel