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

# CS 110
# Computer Architecture
# Thread-Level Paralellism

Instructors: Siting Liu & Yuan Xiao
Course website: https://faculty.sist.shanghaitech.edu.cn/liust/courses/CS110.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2026/5/19

# Administratives

- Mid-term II May 21st 8am-10am; you can bring 2-page A4-sized double-sided cheat sheet, **handwritten** only! (Teaching center 301/SIST 1D-107/108); From start to May 19th lecture (Thread-level parallelism/TLP included).
- Project 2.2 to released, ddl June 4th.
- HW6 released, ddl May 28th!
- Lab 12 to be released.
- Discussion May 22nd on Cache.

# Parallelism Overview

*Software*

- 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

Harness Parallelism & Achieve High Performance

*Hardware*

Warehouse Scale Computer

Smart Phone




**Today’s Lecture**

# Motivation

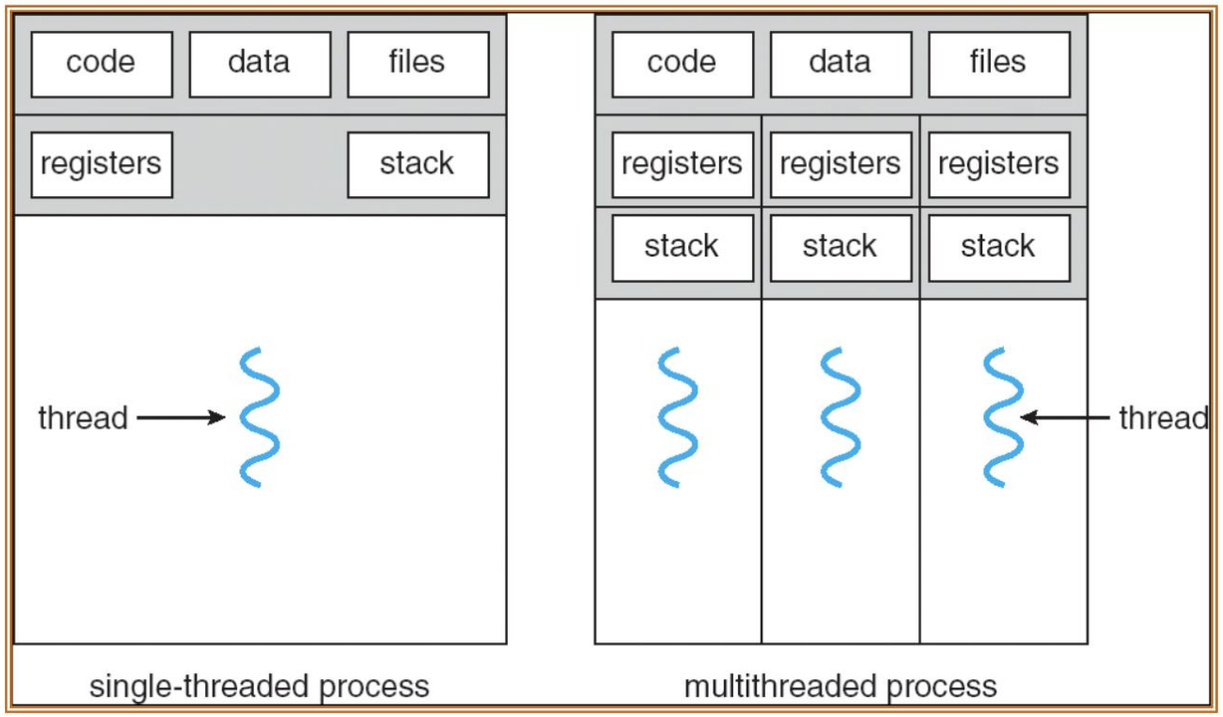
进程(P) 资源(R) 文件系统(F)

| 进程名 | 用户 | % CPU | ID | 内存 | 磁盘读取总计 | 磁盘写入总计 | 磁盘读取 | 磁盘写入 | 优先级 |
|---|---|---|---|---|---|---|---|---|---|
| promecefpluginh | siting | 2.12 | 38213 | 216.4 MB | 3.9 MB | 8.4 MB | *不适用* | *不适用* | 普通 |
| gnome-shell | siting | 1.21 | 3767 | 223.4 MB | 52.9 MB | 196.6 kB | 25.3 KiB/s | *不适用* | 普通 |
| wpsoffice | siting | 0.62 | 37774 | 46.9 MB | 115.1 MB | 3.8 MB | *不适用* | *不适用* | 普通 |
| ibus-daemon | siting | 0.42 | 4141 | 13.8 MB | 221.2 kB | 8.2 kB | *不适用* | *不适用* | 普通 |
| wps | siting | 0.20 | 38439 | 177.4 MB | 148.1 MB | 42.8 MB | *不适用* | 1.3 KiB/s | 普通 |
| wpp | siting | 0.20 | 38231 | 555.8 MB | 126.4 MB | 671.8 MB | *不适用* | *不适用* | 普通 |
| ibus-extension-gtk3 | siting | 0.19 | 4319 | 16.6 MB | 1.6 MB | *不适用* | *不适用* | *不适用* | 普通 |
| ibus-engine-libpinyin | siting | 0.19 | 4521 | 22.9 MB | 40.3 MB | 905.2 kB | 33.3 KiB/s | *不适用* | 普通 |
| wpsoffice | siting | 0.19 | 37681 | 89.3 MB | 212.6 MB | 40.1 MB | *不适用* | *不适用* | 普通 |
| wpscloudsvr | siting | 0.17 | 37709 | 87.9 MB | 68.6 MB | 60.8 MB | *不适用* | 1.3 KiB/s | 普通 |
| promecefpluginh | siting | 0.17 | 37870 | 29.4 MB | 14.1 MB | *不适用* | *不适用* | *不适用* | 普通 |
| gnome-system-monitor | siting | 0.13 | 344605 | 56.7 MB | 7.8 MB | *不适用* | 2.7 KiB/s | *不适用* | 普通 |
| nautilus | siting | 0.08 | 5577 | 112.0 MB | 27.4 MB | 1.6 MB | 90.7 KiB/s | *不适用* | 普通 |
| gnome-characters | siting | 0.08 | 344848 | 57.7 MB | 4.6 MB | 4.1 kB | *不适用* | *不适用* | 普通 |
| Isolated Web Co | siting | 0.05 | 15487 | 131.4 MB | 9.2 MB | *不适用* | *不适用* | *不适用* | 普通 |
| gjs | siting | 0.05 | 132796 | 23.0 MB | *不适用* | *不适用* | *不适用* | *不适用* | 普通 |
| gnome-clocks | siting | 0.05 | 344846 | 10.6 MB | 426.0 kB | *不适用* | *不适用* | *不适用* | 普通 |
| java | siting | 0.03 | 4376 | 252.8 MB | 171.1 MB | 4.7 MB | *不适用* | *不适用* | 普通 |
| gnome-keyring-daemon | siting | 0.02 | 3484 | 1.2 MB | 2.9 MB | 4.1 kB | *不适用* | *不适用* | 普通 |
| dbus-daemon | siting | 0.02 | 3495 | 2.6 MB | *不适用* | *不适用* | *不适用* | *不适用* | 普通 |
| Xwayland | siting | 0.02 | 4528 | 41.0 MB | 1.2 MB | *不适用* | *不适用* | *不适用* | 普通 |
| tracker-miner-fs-3 | siting | 0.02 | 4597 | 21.7 MB | 101.1 MB | 19.8 MB | 172.0 KiB/s | *不适用* | 非常低 |
| aTrustAgent | siting | 0.02 | 4883 | 46.8 MB | 84.1 MB | 4.0 MB | *不适用* | *不适用* | 普通 |
| Isolated Web Co | siting | 0.02 | 116700 | 59.0 MB | 16.4 kB | *不适用* | *不适用* | *不适用* | 普通 |
| Web Content | siting | 0.02 | 341781 | 16.7 MB | *不适用* | *不适用* | *不适用* | *不适用* | 普通 |
| Web Content | siting | 0.02 | 344278 | 16.6 MB | *不适用* | *不适用* | *不适用* | *不适用* | 普通 |
| Isolated Web Co | siting | 0.00 | 129673 | 121.3 MB | 471.0 kB | *不适用* | *不适用* | *不适用* | 普通 |
| (sd-pam) | siting | 0.00 | 3463 | 1.9 MB | *不适用* | *不适用* | *不适用* | *不适用* | 普通 |
| pipewire | siting | 0.00 | 3476 | 8.0 MB | 86.0 kB | *不适用* | *不适用* | *不适用* | 非常高 |
| pipewire | siting | 0.00 | 3477 | 901.1 kB | *不适用* | *不适用* | *不适用* | *不适用* | 普通 |
| wireplumber | siting | 0.00 | 3481 | 4.6 MB | 434.2 kB | 131.1 kB | *不适用* | *不适用* | 非常高 |
| pipewire-pulse | siting | 0.00 | 3482 | 10.2 MB | *不适用* | *不适用* | *不适用* | *不适用* | 非常高 |
| xdg-document-portal | siting | 0.00 | 3544 | 798.7 kB | 147.5 kB | *不适用* | *不适用* | *不适用* | 普通 |
| xdg-permission-store | siting | 0.00 | 3548 | 540.7 kB | *不适用* | *不适用* | *不适用* | *不适用* | 普通 |
| gdm-wayland-session | siting | 0.00 | 3598 | 528.4 kB | 1.2 MB | *不适用* | *不适用* | *不适用* | 普通 |
| gnome-session-binary | siting | 0.00 | 3605 | 2.0 MB | 4.1 MB | *不适用* | *不适用* | *不适用* | 普通 |
| gcr-ssh-agent | siting | 0.00 | 3645 | 712.7 kB | 65.5 kB | *不适用* | *不适用* | *不适用* | 普通 |
| gnome-session-ctl | siting | 0.00 | 3646 | 471.0 kB | 20.5 kB | *不适用* | *不适用* | *不适用* | 普通 |

# Threads

- Threads (short for threads of execution) is a single stream of instructions.
- Each thread has (its own state):
  - 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

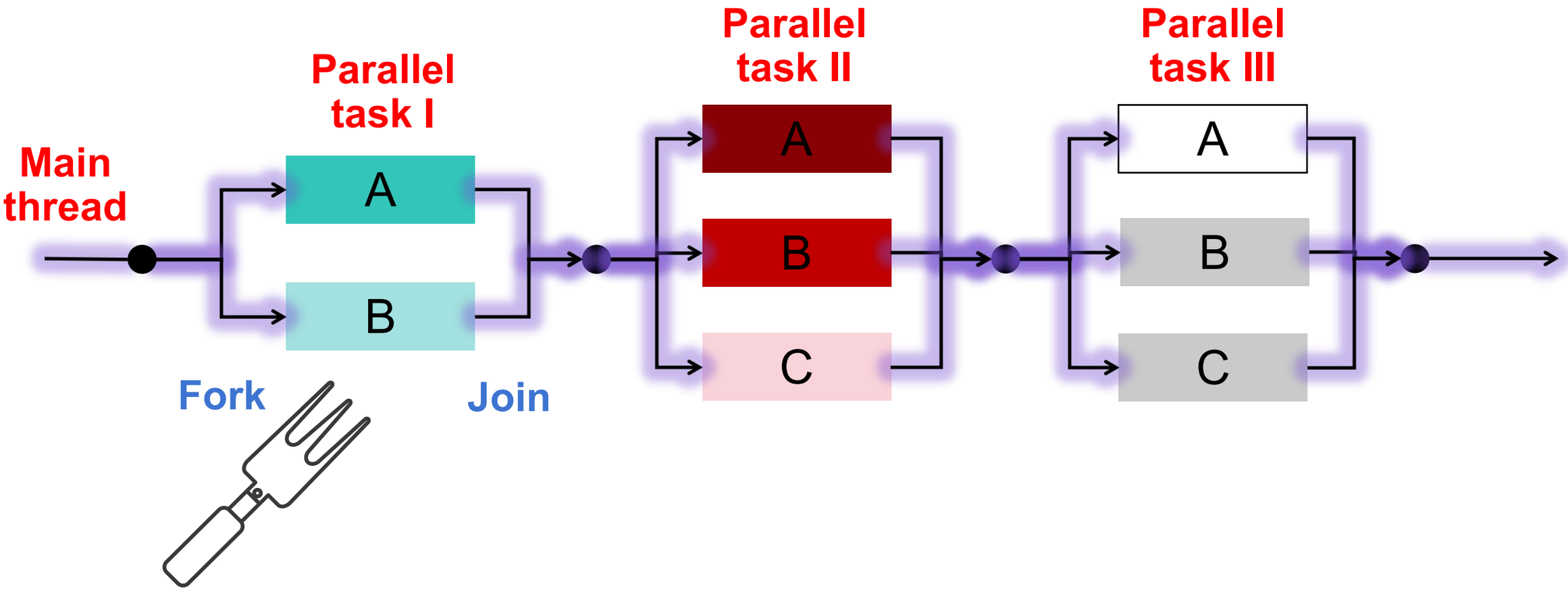
- Within a given program's process, threads can run concurrently.
- Operating system (OS) multiplexes multiple **software** threads onto the available hardware threads.




**Software** threads

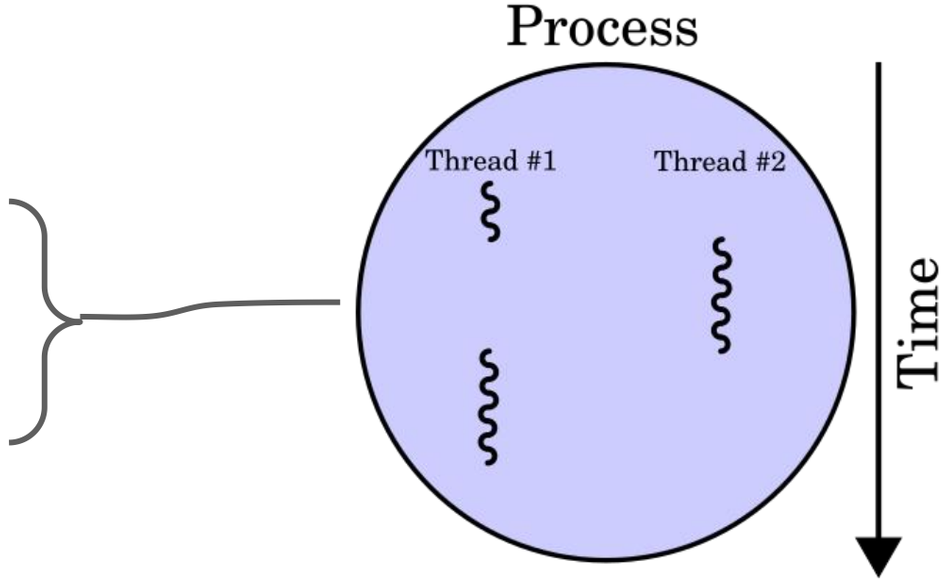
# Fork-Join Model

- 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 >> 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

Pragmas are a preprocessor mechanism C provides for language extensions.

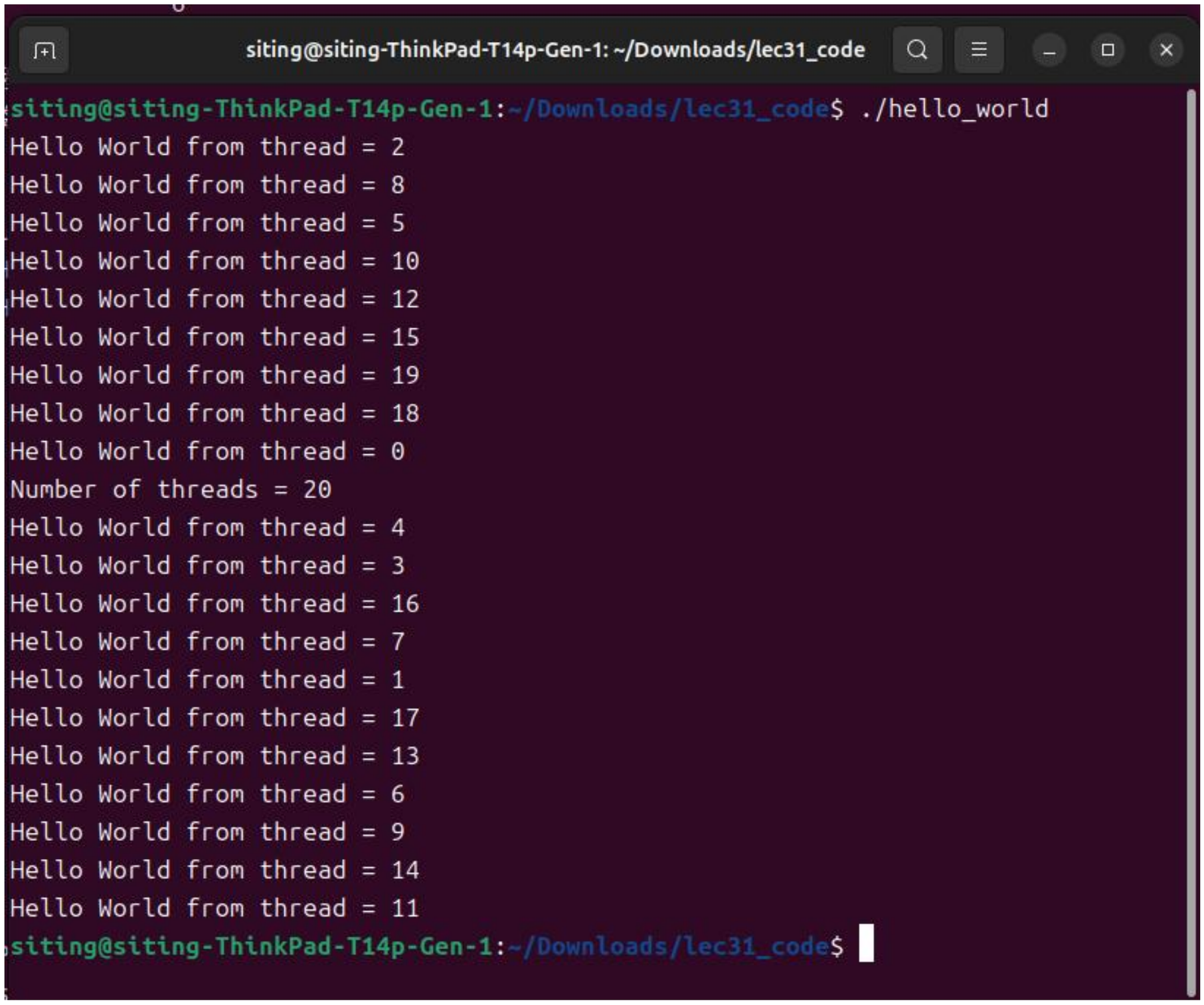
```
#include <stdio.h>
#include <omp.h>
int main() {
  /* Fork team of threads with private variable tid */
  #pragma omp parallel
  {
    int tid = omp_get_thread_num(); /* get thread id */
    printf("Hello World from thread = %d\n", tid);
    /* Only main thread does this */
    if (tid == 0) {
 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

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

- Basic OpenMP construct for parallel regions:
  - Each thread runs the instructions within the block
  - Thread scheduling is non-deterministic

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

# “Hello, world!” to OpenMP



# 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 multiplex-ed onto available hardware threads.

| OpenMP Intrinsic | Description |
|---|---|
| `omp_set_num_threads(x);` | Set number of threads to x. |
| `num_th = omp_get_num_threads();` | Get number of threads. |
| `th_ID = omp_get_thread_num();` | 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)

# OpenMP: Thread-Based Parallelism

- 1. Explicit programming model with full programmer control over parallelization
- Pros:
  - Compiler directives are simple and easy to use;
  - Legacy serial code does not need to be rewritten;
- Cons:
  - Compiler must support OpenMP (e.g. gcc 4.2);
  - Amdahl's law is gonna get you after not too many cores…;
- 2. Multiple threads in a **shared memory environment**
- Pros:
  - Takes advantage of shared memory, programmer need not worry (that much) about data placement;
- Cons:
  - Code can only be run in shared memory environments;
  - **Synchronizing use of shared resources is hard** (more later).

# 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 ~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] = …;
  }
}
```

Equivelant

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

# 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)
```

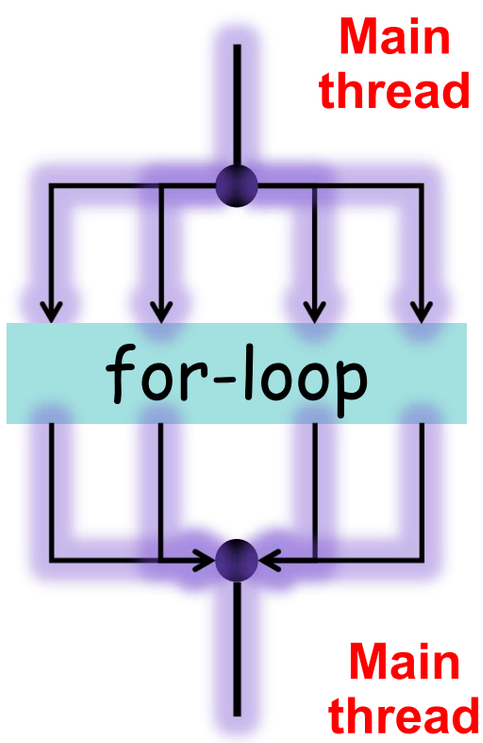
- Automatic work-sharing (for-loop):

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

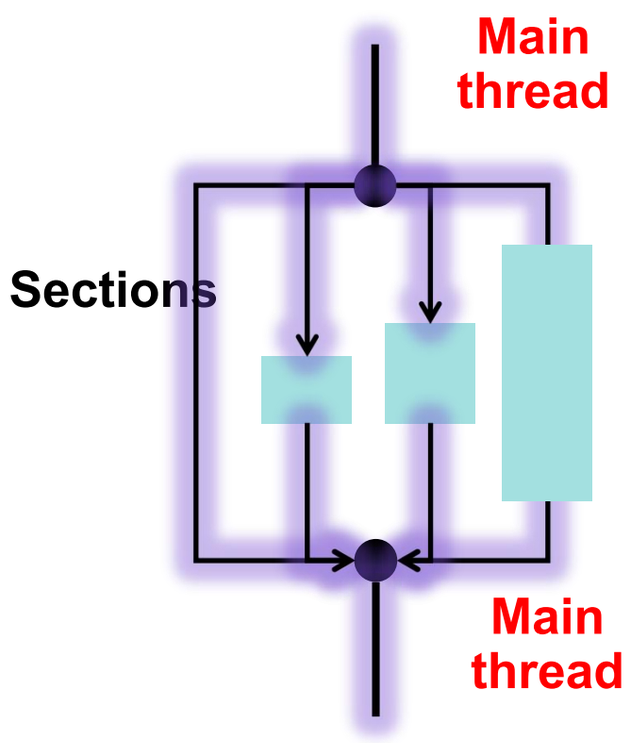
  - Can be shortened as: `#pragma omp parallel for`
  - Implicit “barrier” synchronization at end of the for loop
  - `for-loop` index private for each thread, variables declared outside `for-loop` shared

# OpenMP Directives (Work-sharing)

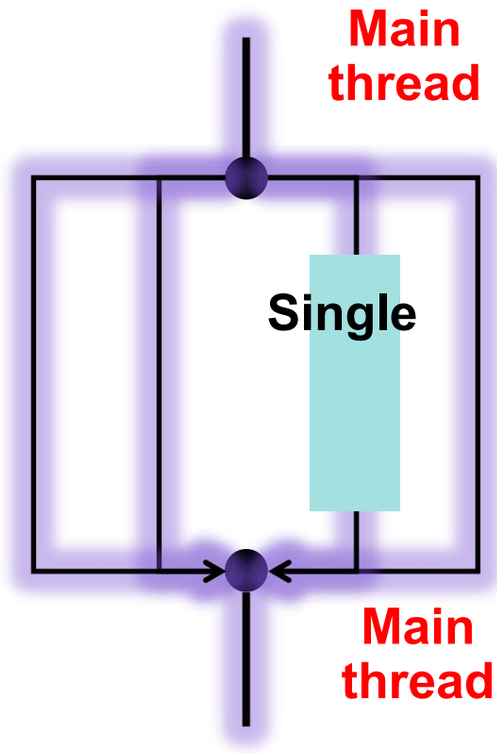
- Defined within a *parallel* section




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

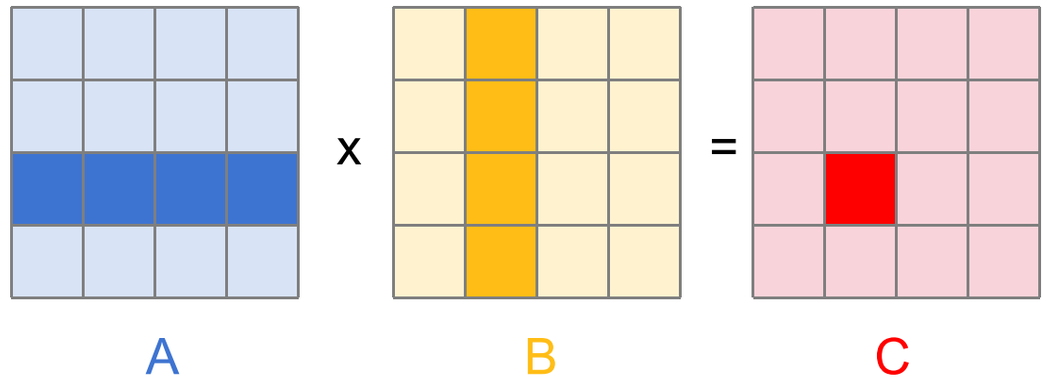
- Elapsed wall clock time:

```
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

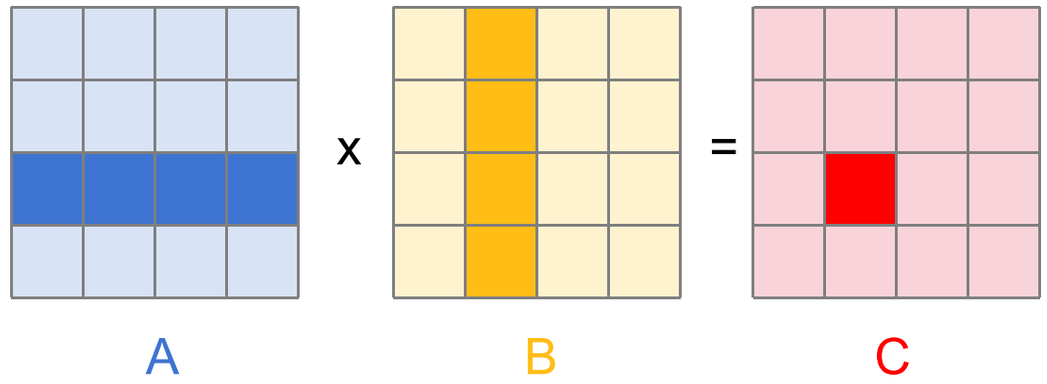
```
double dgemm_scalar(int N, double *A,
                    double *B, double *C) {
  double start_time = omp_get_wtime();
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      double Cij = 0;
      for (int k = 0; k < N; k++) {
        Cij += A[i+k*N] *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



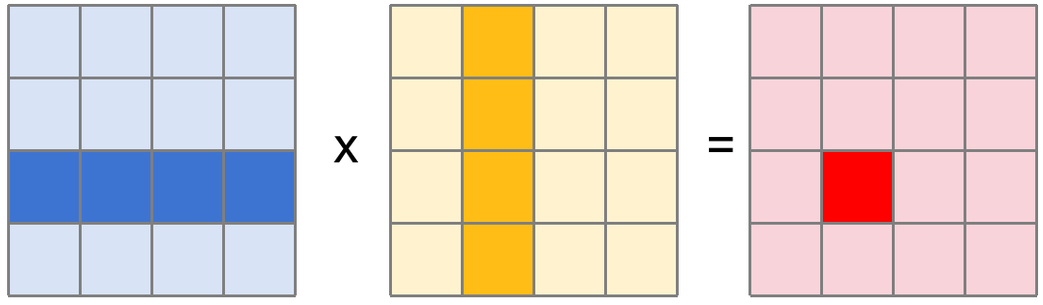

Explicitly declare private variables

# Notes on Matrix Multiplication Example

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

# OpenMP Matrix Multiplication with SIMD

```
double dgemm_simd(int N, double *A, double *B, double *C) {
    memset(C, 0, N * N * sizeof(double));
    double start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            __m256d sum_vec = _mm256_setzero_pd();
            for (int k=0; k <= N - 4; k += 4) {
                __m256d a_vec = _mm256_set_pd(
                    A[i + (k+3)*N], A[i + (k+2)*N],A[i + (k+1)*N],A[i + k*N]);
                __m256d b_vec = _mm256_loadu_pd(&B[k + j*N]);
                #ifdef __FMA__    {sum_vec = _mm256_fmadd_pd(a_vec, b_vec, sum_vec);}
                #else  {sum_vec = _mm256_add_pd(_mm256_mul_pd(a_vec, b_vec), sum_vec);}
                #endif
            }
            double sum = sum_vec[0] + sum_vec[1] + sum_vec[2] + sum_vec[3];
            for (; k < N; k++) {sum += A[i + k*N] * B[k + j*N];}    // trailing elements
            C[i + j*N] = sum;
        }
    }
    double run_time = omp_get_wtime() - start_time;
    return run_time;
}
```



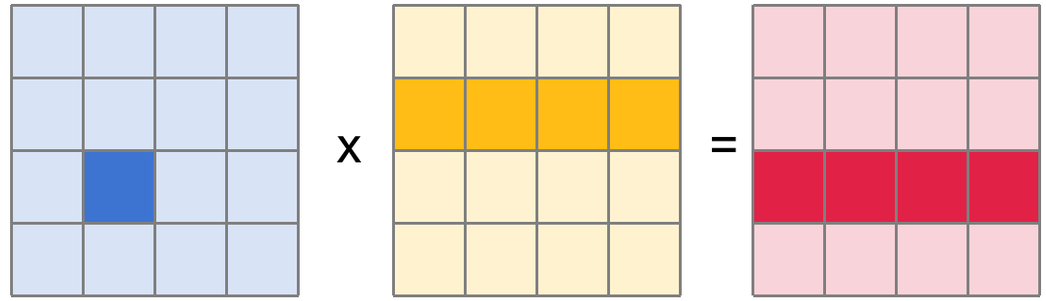

SIMD intrinsics inside each thread

Compile options: gcc -O3 -march=native -mfma -fopenmp -o dgemm dgemm.c

# OpenMP Matrix Multiplication with More

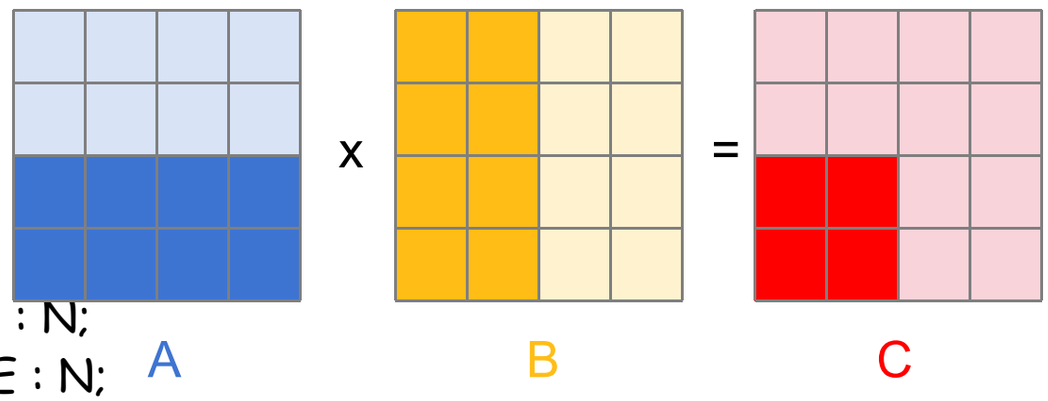
```
double dgemm_reordered(int N, double *A, double *B, double *C) {
    memset(C, 0, N * N * sizeof(double));

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

Loop reordered



# OpenMP Matrix Multiplication with CB

```
#define BLOCK_SIZE 64
double dgemm_blocked_simd(int N,double*A,double*B,double*C){
    memset(C,0, N * N *sizeof(double));
    double start_time =omp_get_wtime();
      #pragma omp parallel for collapse(2)
    for(int i0 =0; i0 < N; i0 += BLOCK_SIZE){
       for(int j0 =0; j0 < N; j0 += BLOCK_SIZE){
          int i_max =(i0 + BLOCK_SIZE < N)? i0 + BLOCK_SIZE : N;
          int j_max =(j0 + BLOCK_SIZE < N)? j0 + BLOCK_SIZE : N;
        // Outer loop for blocks, inner loop for inside-block data processing with SIMD
       for(int i = i0; i < i_max; i++){
          for(int j = j0; j < j_max; j++){
             __m256d sum_vec =_mm256_setzero_pd();
                for(int k =0; k <= N -4; k +=4){
                __m256d a_vec =_mm256_set_pd(A[i +(k+3)*N], A[i +(k+2)*N], A[i +(k+1)*N], A[i + k*N]);
                __m256d b_vec =_mm256_loadu_pd(&B[k + j*N]);
                #ifdef __FMA__  {sum_vec =_mm256_fmadd_pd(a_vec, b_vec, sum_vec);}
                #else {sum_vec =_mm256_add_pd(_mm256_mul_pd(a_vec, b_vec), sum_vec);}
                #endif}
                double sum = sum_vec[0]+ sum_vec[1]+ sum_vec[2]+ sum_vec[3];
             for(int k =(N /4)*4; k < N; k++){ sum += A[i + k*N]* B[k + j*N];}
             C[i + j*N]= sum;}}}}
             double run_time =omp_get_wtime()- start_time;
return run_time;}
```



# 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

| | | | |
|---|---|---|---|
| ● **Grey** thread load | read | x: 0 | load |
| ● **Grey** thread store | write | x: 1 | addi |
| ● **Green** thread load | read | x: 1 | store |
| ● **Green** thread store | write | x: 2 | load |
| ● **Blue** thread load | read | x: 2 | addi |
| ● **Blue** thread store | write | x: 3 | store |
| ● **Red** thread load | read | x: 3 | load |
| ● **Red** thread store | write | x: 4 | addi |
| | | | store |
| | | | load |
| | | | 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 |
| | | | load |
| | | | addi |
| | | | store |

Final value of x: 1

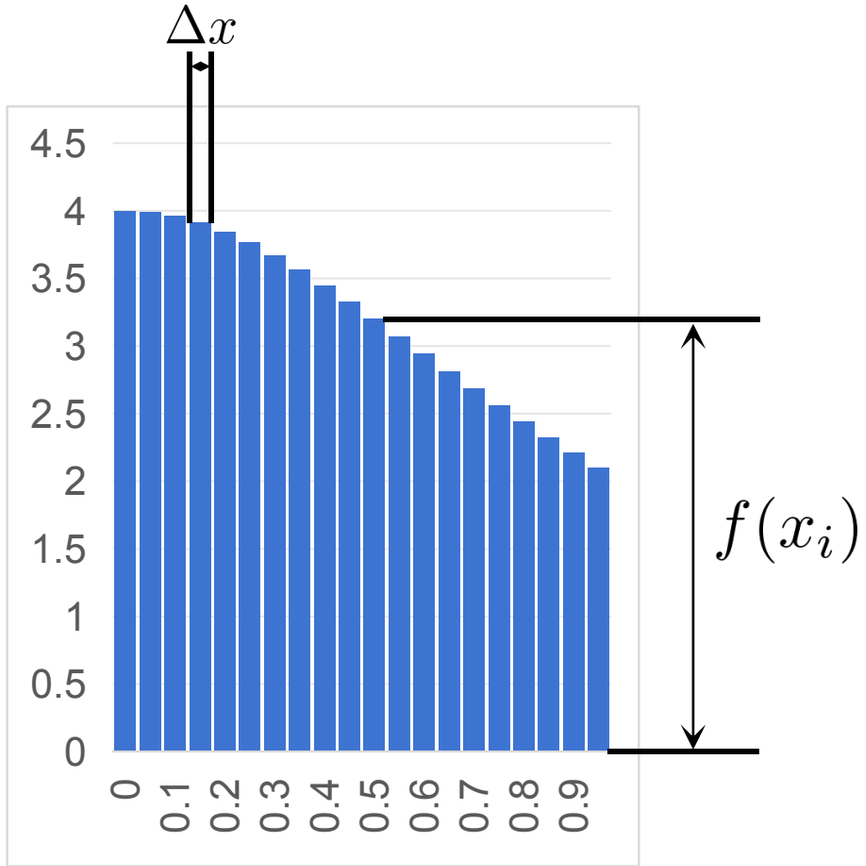
# Data Race: Cases ...

- **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.

# 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)

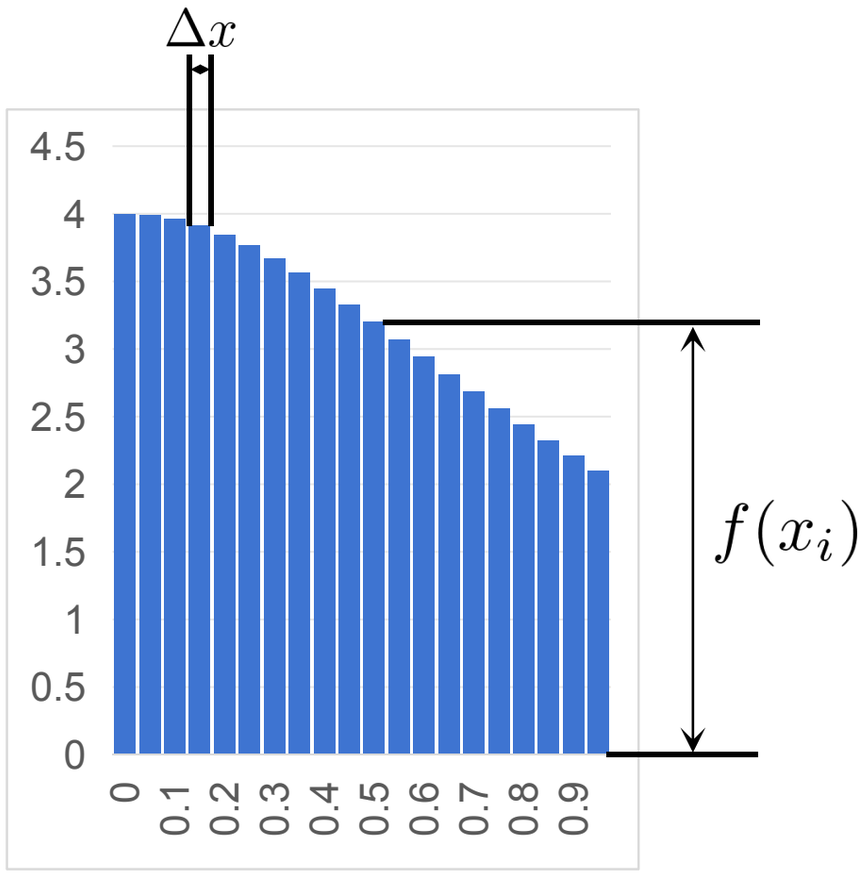



$$\sum f(x_i)\Delta x \approx \pi$$

# 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);
}
```

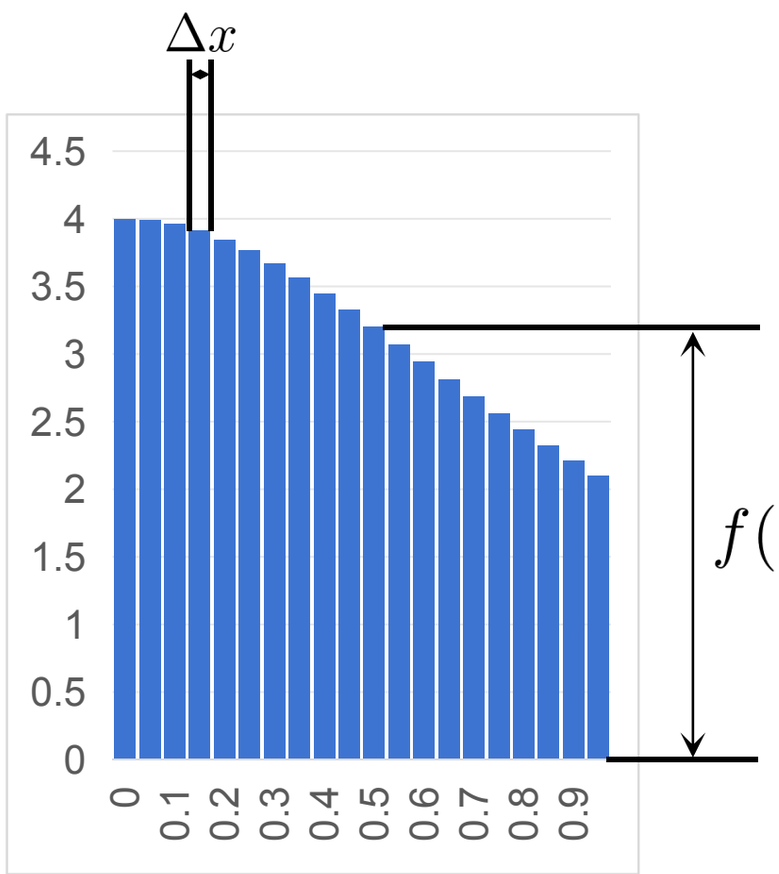
pi=3.141800986893.

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

# Example: Calculating π

- OpenMP version 1

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



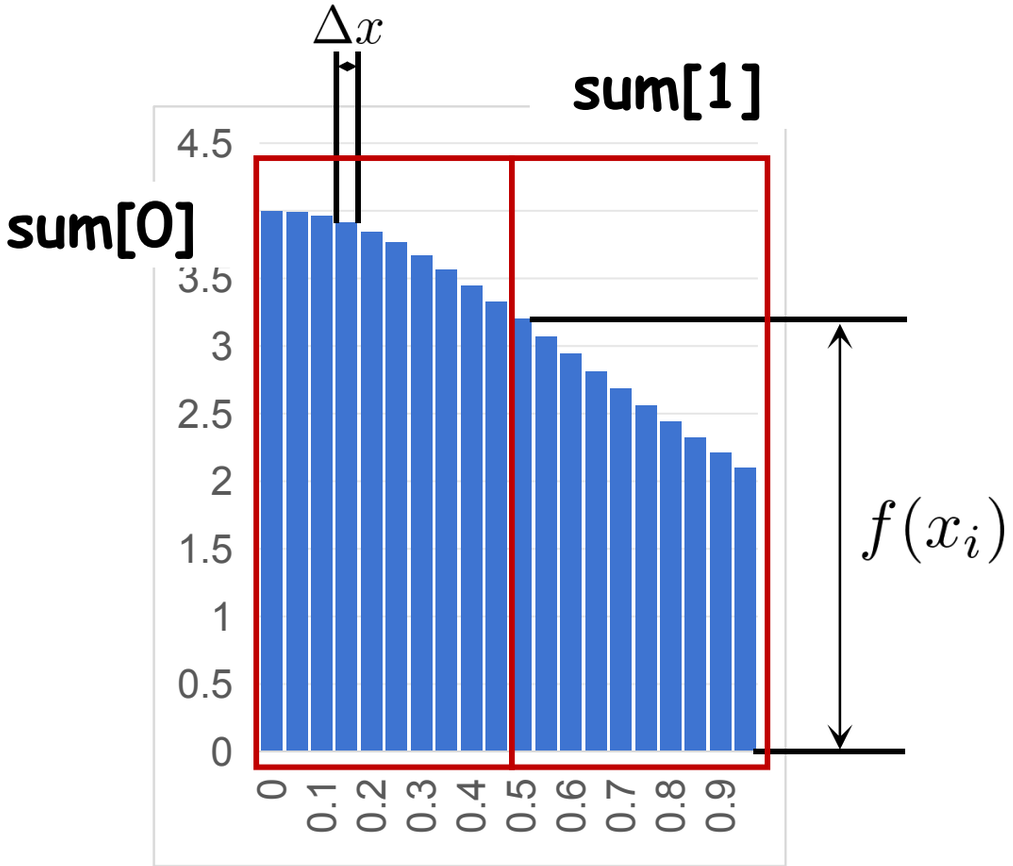

```
#include <stdio.h>
#include <omp.h>
void main(){
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double sum = 0.0;
#pragma omp parallel for
  for (int i=0; i<num_steps; i++){
     double x = (i+0.5)*step;
     sum += 4.0*step/(1.0+x*x);
  }
  printf("pi=%6.12f\n",sum);}
```

- Problem: each thread needs access to the shared variable **sum**
- Code runs sequentially …

# Example: Calculating π

- OpenMP version 2

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



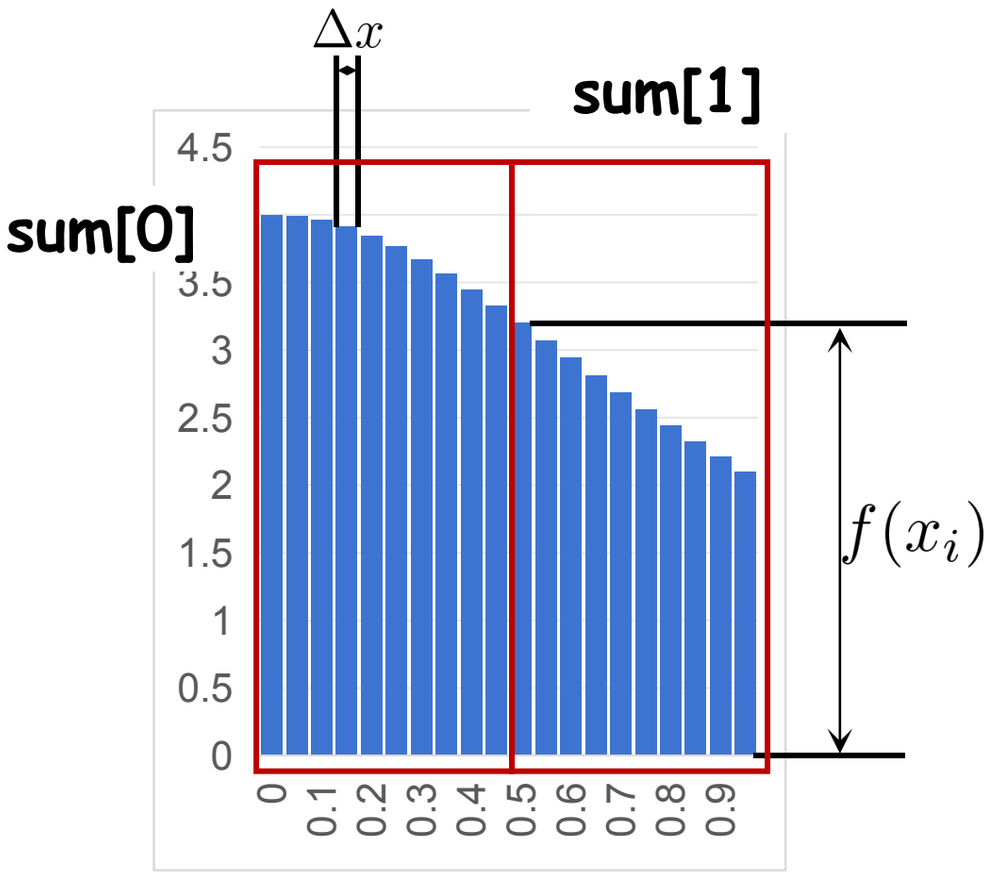

Divide & conquer

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

# Example: Calculating π

- OpenMP version 2

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



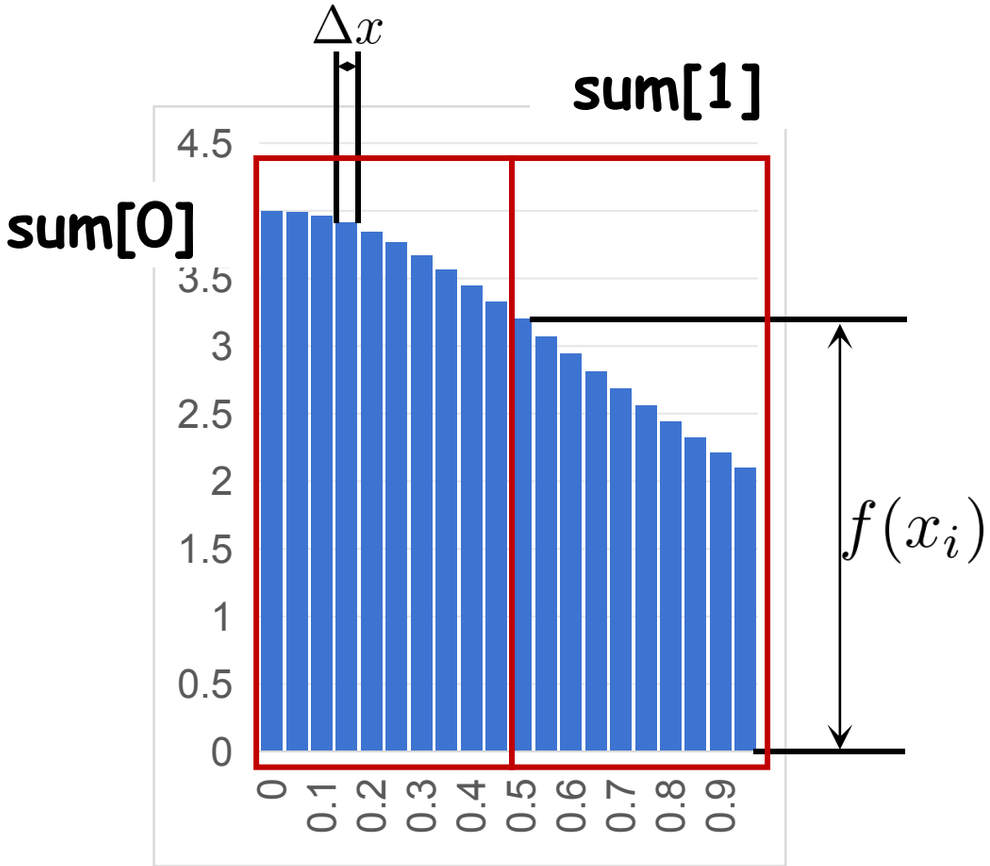

```
#include <stdio.h>
#include <omp.h>
void main(){
  const int NUM_THREADS = 4;
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double sum[NUM_THREADS];
  for (int i=0;i<NUM_THREADS;i++)
    sum[i]=0;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int id = 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);
    printf("i=%3d,id=%3d\n",i,id);
   }
  }
   double pi=0;
  for (int i=0;i<NUM_THREADS;i++)
    pi += sum[i];
  printf("pi=%6.12f\n",pi);
}
```

**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);
    printf("i=%3d,id=%3d\n",i,id);
   }
   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
```

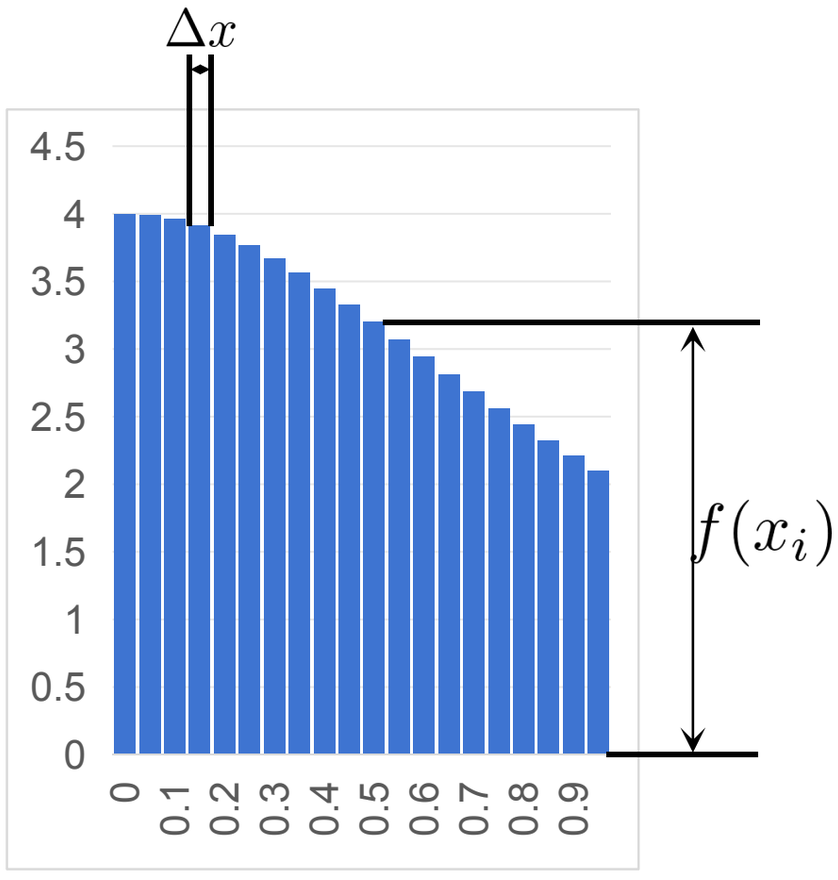
- *Problem is that we really want sum over all threads!*
- ***Reduction***: specifies that, 1 or more variables that are private to each thread, are subject of reduction operation at end of parallel region:

  **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

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



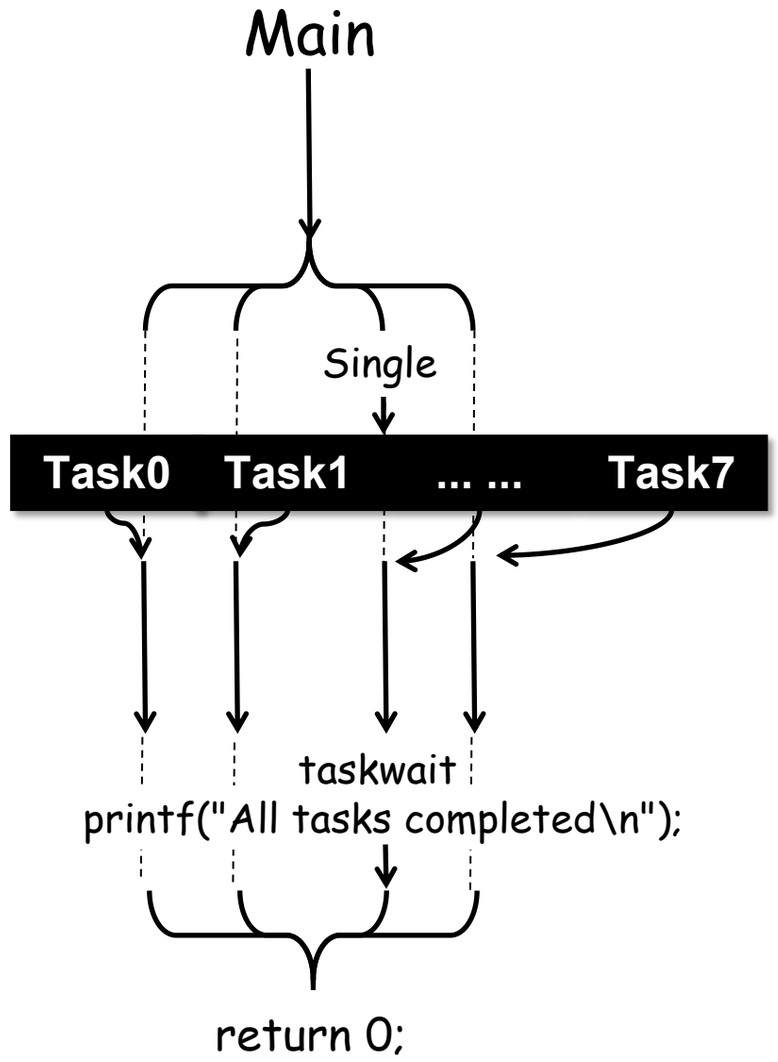

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000;
double step;
void main (){
  int i; double x, pi, sum = 0.0;
  step = 1.0 / (double)num_steps;
  #pragma omp parallel for private(x) reduction(+:sum)
  for (i=1; i<= num_steps; i++){
    x = (i - 0.5) * step;
    sum = sum + 4.0 / (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 (no wait if #pragma omp single nowait);
  - 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
- #pragma omp task/taskwait
  - task create a task in the current thread.
  - but not necessarily executed by the current thread, OpenMP will decide based on hardware thread avaliability.
  - taskwait wait for the task completed and continue the current thread
- ... ...
- Learn more at https://www.openmp.org/specifications/

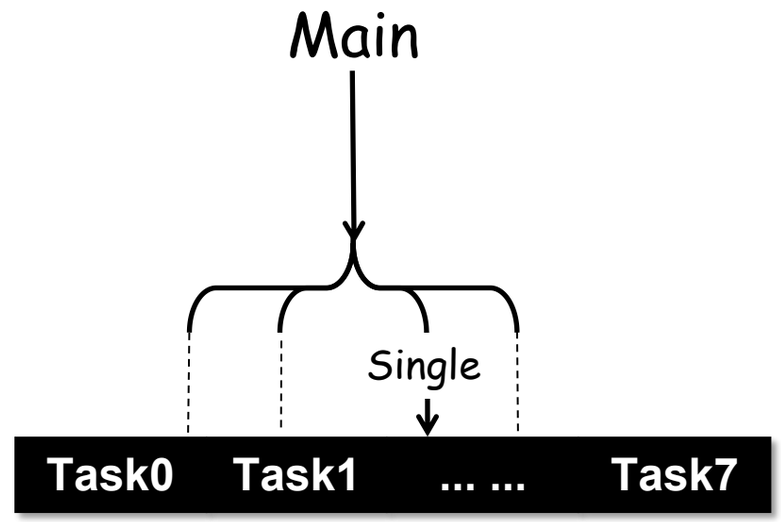
# OpenMP Other Usage: Example

```
#include<stdio.h>
#include<omp.h>
int main(){
    #pragma omp parallel num_threads(4)
     {
     #pragma omp single
      {
          printf("Parent thread (creating tasks):
%d\n",omp_get_thread_num());
          // Create multiple tasks
          for(int i =0; i <8; i++){
            #pragma omp task
             {
                printf("Task %d executed by thread:
%d\n", i, omp_get_thread_num());
             }
          }
          #pragma omp taskwait
          printf("All tasks completed\n");
      }
   }
   return 0;
}
```



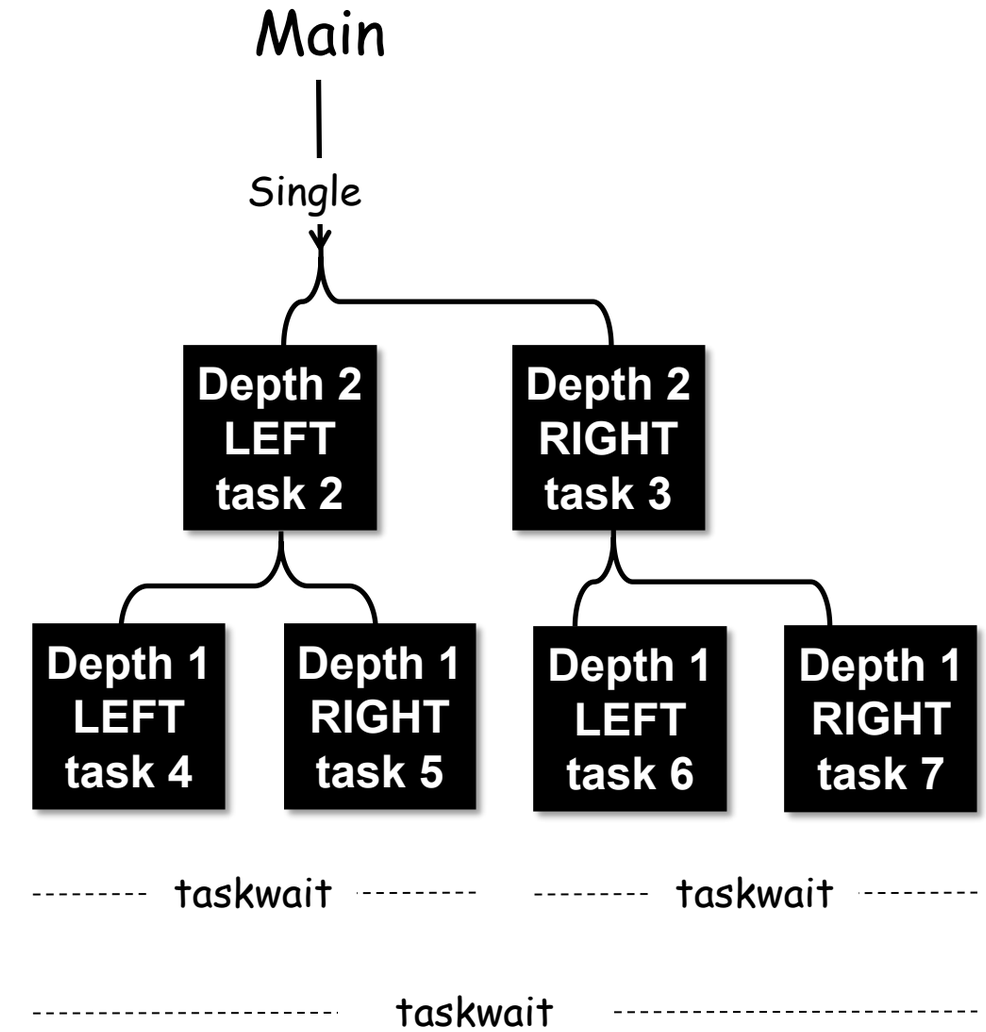
# OpenMP Other Usage: Example

```
#include<stdio.h>
#include<omp.h>
int main(){
    #pragma omp parallel num_threads(4)
     {
     #pragma omp single
      {
          printf("Parent thread (creating tasks):
%d\n",omp_get_thread_num());
        // Create multiple tasks
        for(int i =0; i <8; i++){
          #pragma omp task
           {
              printf("Task %d executed by thread:
%d\n", i, omp_get_thread_num());
           }
          #pragma omp taskwait
        }
        printf("All tasks completed\n");
    }
  }
  return 0;
}
```



# OpenMP Other Usage: Tree Example

```
#include <stdio.h>
#include <omp.h>
void process_tree(int depth, int id) {
    if (depth <= 0) return;
    #pragma omp task
    {
        printf("Creating LEFT task at depth %d (by thread
%d)\n", depth, omp_get_thread_num());
        process_tree(depth - 1, id * 2);
    }
    #pragma omp task
    {
        printf("Creating RIGHT task at depth %d (by thread
%d)\n", depth, omp_get_thread_num());
        process_tree(depth - 1, id * 2 + 1);
    }
    #pragma omp taskwait
    printf("Completed node %d at depth %d (thread %d)\n",
id, depth, omp_get_thread_num());
}
int main() {
     #pragma omp parallel
     #pragma omp single
    process_tree(2, 1);
    return 0;}
```



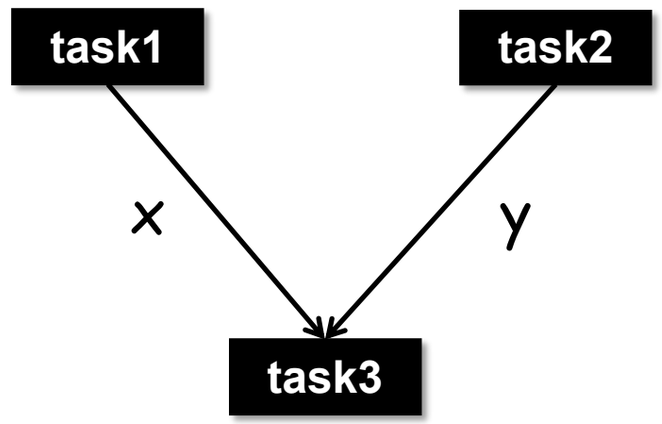
# OpenMP: Task & Dependency

#pragma omp task depend (dependency type: var list)

| Dependency type | Meaning | Explain |
|---|---|---|
| in | Task read these variables | Read after the other tasks write the variables (out/inout) |
| out | Task write these variables | Any read will wait for this variable write |
| inout | Task read&write these variables | in + out |
| mutexinoutset | Task exclusively access variables | Only one task access these variables at a time |

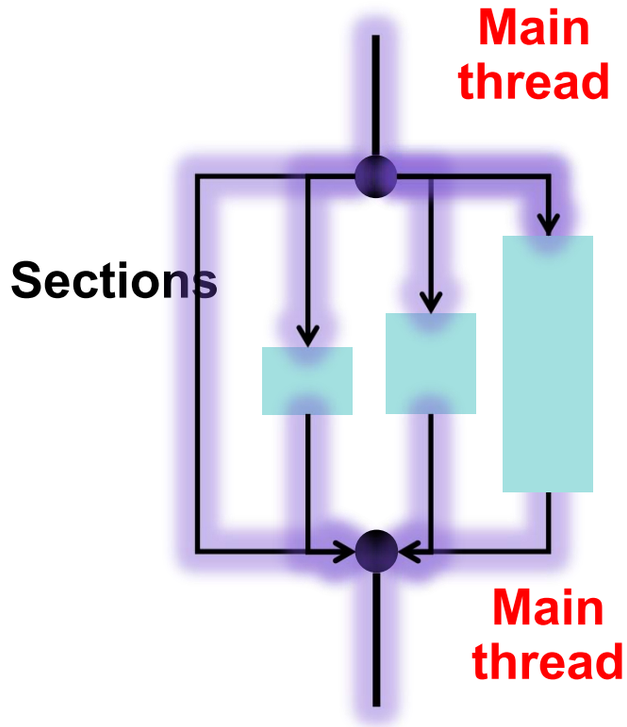
# OpenMP: Dependency Example

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

int main() {
   int x = 0, y = 0;

    #pragma omp parallel
   #pragma omp single
   {
      // task1: write x
        #pragma omp task depend(out: x)
      {
         printf("Task 1: writing x = 10\n");
         x = 10;
      }
      // task2: write y (no dependency with x, execute in parallel)
        #pragma omp task depend(out: y)
         {
         printf("Task 2: writing y = 20\n");
         y = 20;
      }
      // task3: read x & y, depending on tasks 1&2
        #pragma omp task depend(in: x, y)
      {
         printf("Task 3: x + y = %d\n", x + y);
      }
   }
   return 0;
}
```



# OpenMP Sections

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      {
         // Code block 1
      }
      #pragma omp section
      {
         // Code block 2
      }
      #pragma omp section
      {
         // Code block 3
      }
} // Implicit barrier
```



# Synchronization

- To enforce multithreaded program correctness, we often need to **synchronize** threads, i.e., coordinate their execution.
  - Most commonly, know when one task is finished writing so that it is safe for another to read.

  Desired correct outcome:


(or any permutation between these four code segments)

- ⚠ Note: If we enforce the above, then execution effectively becomes sequential…(Amdahl's Law)

# Critical Sections with OpenMP (1/2)

- A critical section is a segment of code that must be executed by a single thread at a time, thereby enforcing synchronization.
  - In OpenMP, you can declare critical sections of code.
  - Each thread can safely execute code in critical section, knowing that it is the only thread that can execute that section at that time
  - This user-level specification (e.g., in OpenMP) relies on hardware synchronization instructions (e.g., in RISC-V) (more later)

- #pragma omp barrier [docs]
  - Forces all threads to wait until all threads have hit the barrier
- #pragma omp critical [docs]
  - Creates a critical section within a parallel code segment; only one thread can run a critical section at a time.

# OpenMP Hello World, Synchronized

```
#include <stdio.h>
#include <omp.h>
int main() {
  int x = 0;                          /* shared variable */
  #pragma omp parallel
  {
    int tid = omp_get_thread_num(); /* private variable */
    #pragma omp critical
    {
      x++;
      printf("Hello World from thread = %d, x = %d\n", tid, x);
    }
    #pragma omp barrier
    if (tid == 0) {
      printf("Number of threads = %d\n", omp_get_num_threads());
    }
  }
}
```

# Critical Sections with OpenMP (2/2)

- A critical section is a segment of code that must be executed by a single thread at a time.
  - In OpenMP, you can declare critical sections of code.
  - Compile to atomic instructions that enable synchronization between threads (more later, RISC-V)

- OpenMP has very restrictive parallelism.
  - Really only good for parallelizing loops…
  - Beyond that:
    - If your critical section is too large, then effectively serial program
      → Amdahl's law quickly rears its ugly head
    - If critical sections not defined well, can run into deadlock.

# Summary

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