多核程序设计与实践

OpenMP入门

陶钧

taoj23@mail.sysu.edu.cn

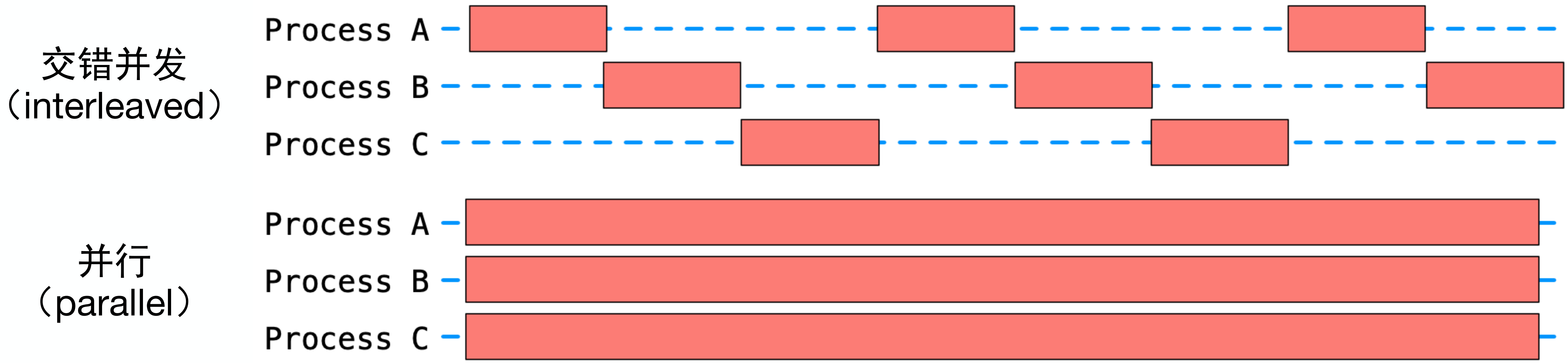
中山大学 数据科学与计算机学院
国家超级计算广州中心

课程概要

- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度

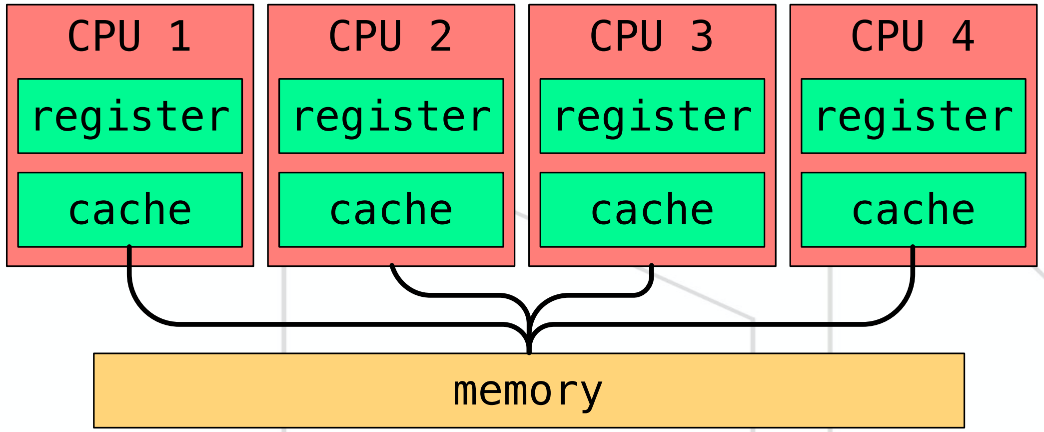
什么是并行？

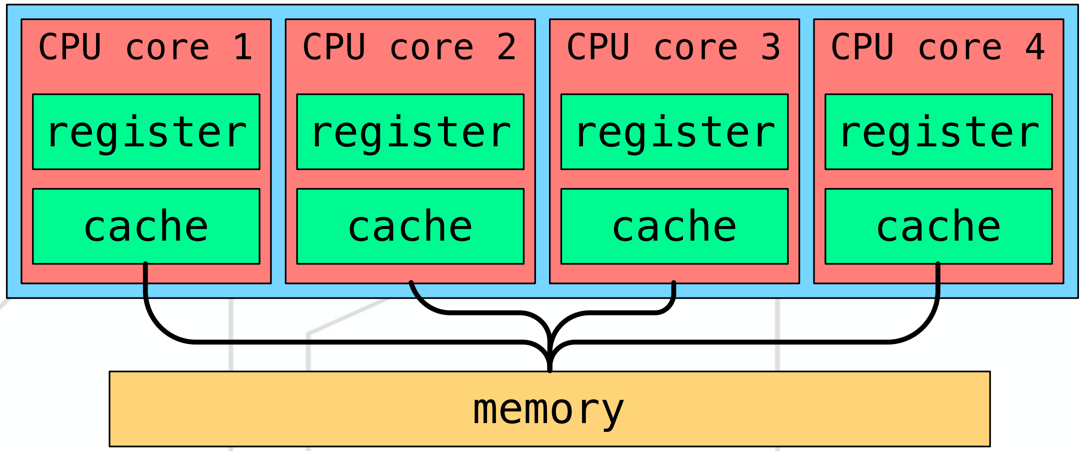
- 多个任务同时进行
  - 生活中的多任务与并行
    - 边吃饭边看电视边聊天，边听音乐边回邮件
    - 边上课边睡觉，边开车边发短信
    - 写论文、写代码、完成作业（？）
  - 计算机上的多任务与并行



什么是并行？

- 多任务实现方式
  - 并发（concurrent）
    - 多个进程或线程“同时”进行
    - 交错并发（interleaved）：可通过系统调度由单核完成
    - 并行（parallel）：需多个运算核心同时完成
      - 多处理器 vs 单处理器多核
        - » 同一芯片上的多核通信速度更快
        - » 同一芯片上的多核能耗更低



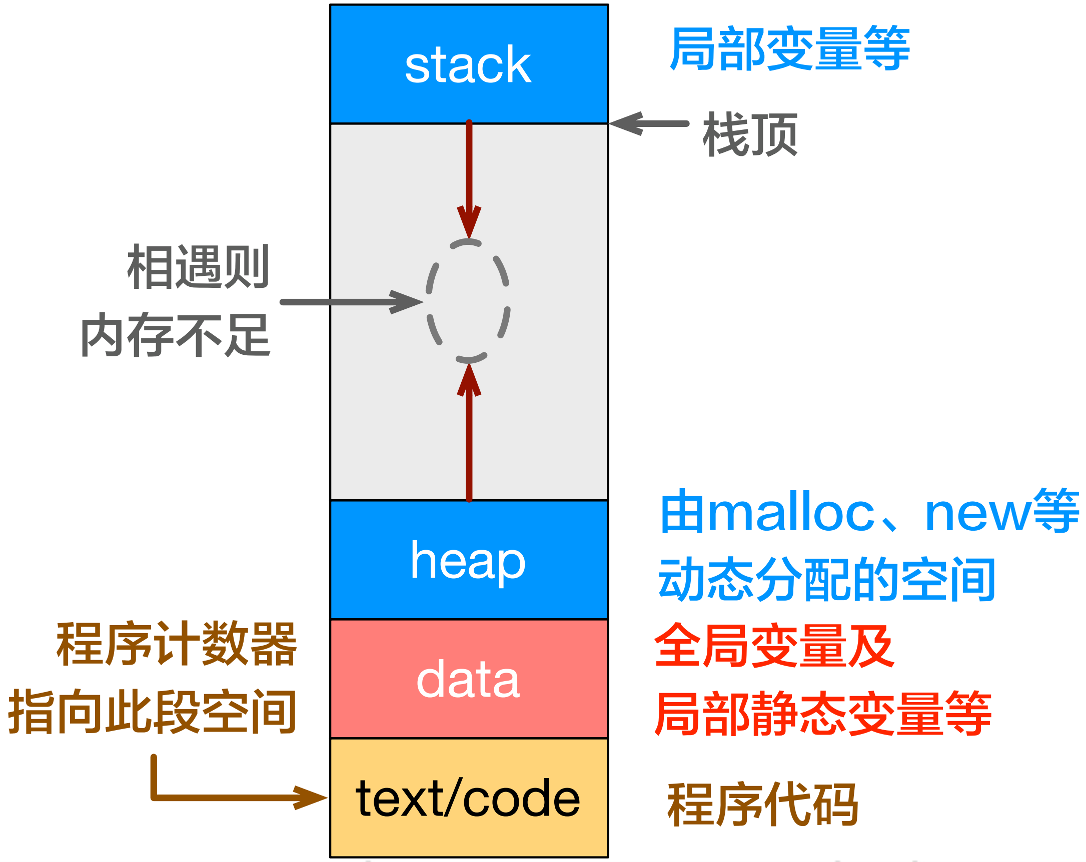


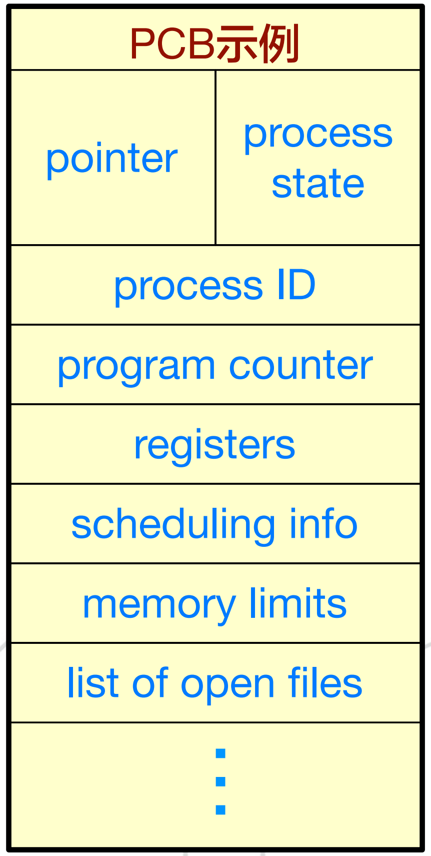
什么是并行？

多任务实现方式

– 进程：一个执行中的程序即为一个进程

- 每个进程有独立的程序计数器（program counter）、堆（heap）、栈（stack）、数据段（data section）、代码段（code section）等
- 程序运行状态由进程控制器（process control block）记录





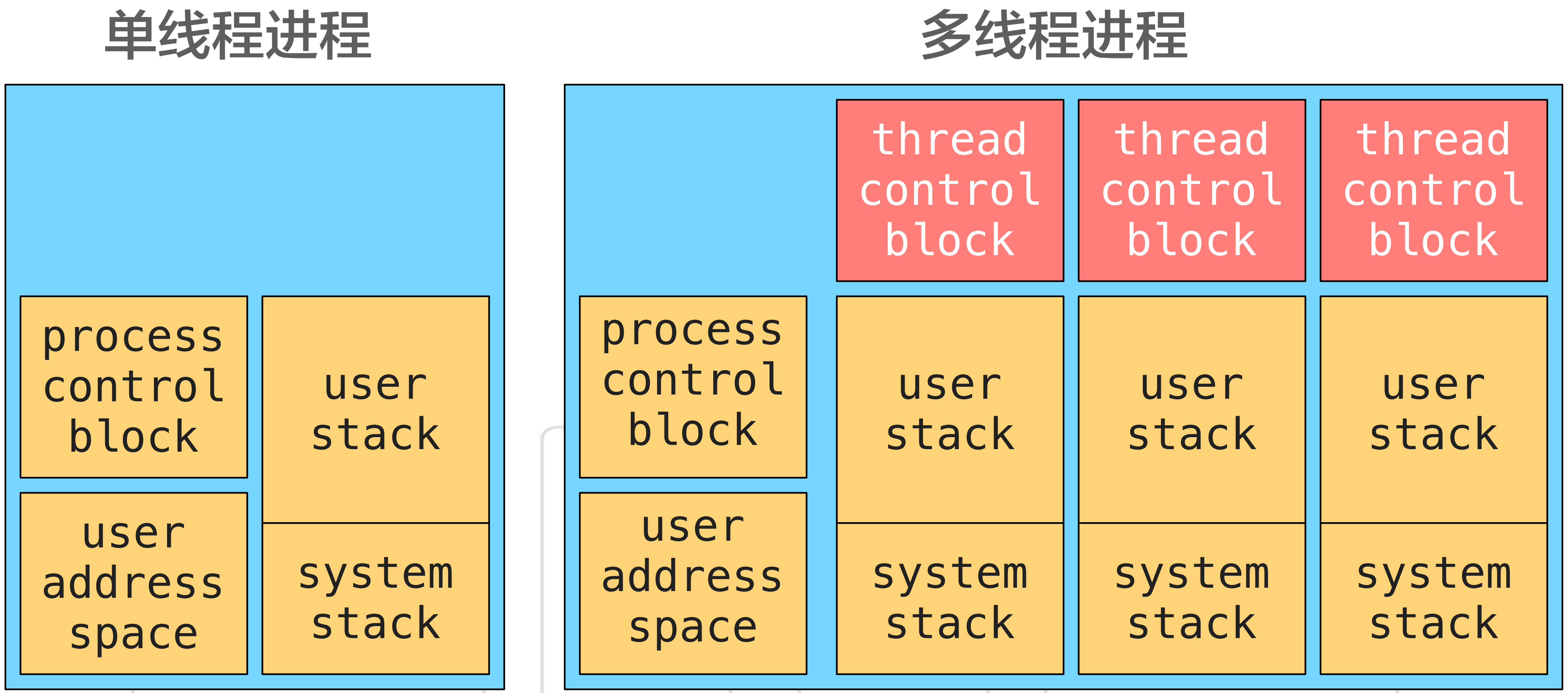
进程状态：
new，running，waiting，ready，terminated

寄存器：如累加器、堆栈指针等
用于进程中断后恢复

调度信息：如优先级等

什么是并行？

- 多任务实现方式
  - 线程：常被称为轻量级进程（lightweight process）
    - 与进程相似：每个线程有线程ID、程序计数器、寄存器、栈等
    - 与进程不同：所有线程共享代码段、数据段及其他系统资源（如文件等）



什么是OpenMP?

- OpenMP：Open Multi-Processing
  - 多线程编程API
    - 编译器指令（#pragma）、库函数、环境变量
    - 极大地简化了C/C++/Fortran多线程编程
    - 并不是全自动并行编程语言
      - 其并行行为仍需由用户定义及控制
  - 支持共享内存的多核系统
    - 与CUDA、MPI所支持的硬件比较（讲义1）

Compiler directives	OpenMP library	Environment variables
OpenMP runtime library		
OS support for shared memory and threading		

课程概要

- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度

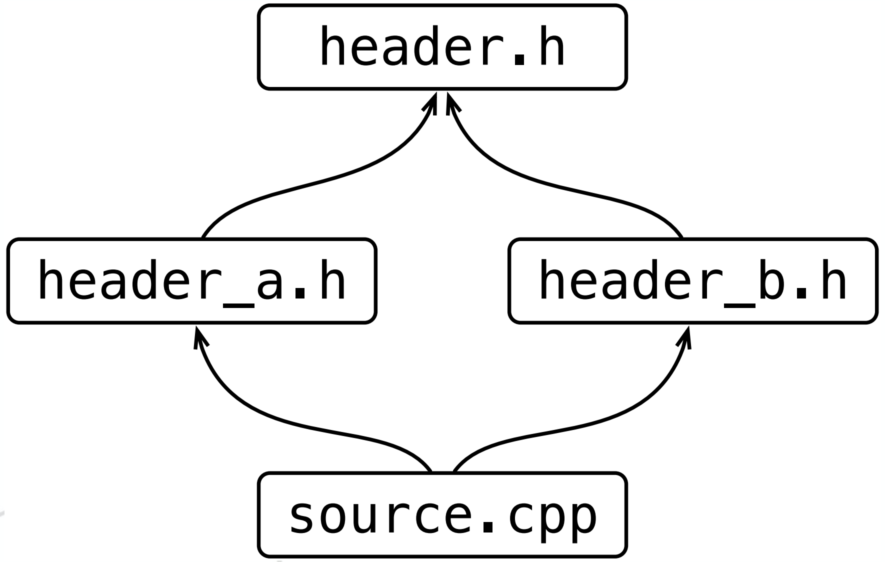
#pragma

- 预处理指令
  - 设定编译器状态或指定编译器完成特定动作
    - 需要编译器支持相应功能
    - 否则将被忽略
  - 举例：`#pragma once`
    - 指定头文件只被编译一次

```
#ifndef HEADER_H
#define HEADER_H

...

#endif //HEADER_H
```

`#pragma once`
- 需要编译器支持
- 针对物理文件
- 需要用户保证头文件没有多份拷贝

`#ifndef`
- 不需要特定编译器
- 不针对物理文件
- 需要用户保证不同文件的宏名不重复



#pragma

其他#pragma指令

- `#pragma GCC poison printf`
- `#pragma warning (disable : 4996)`

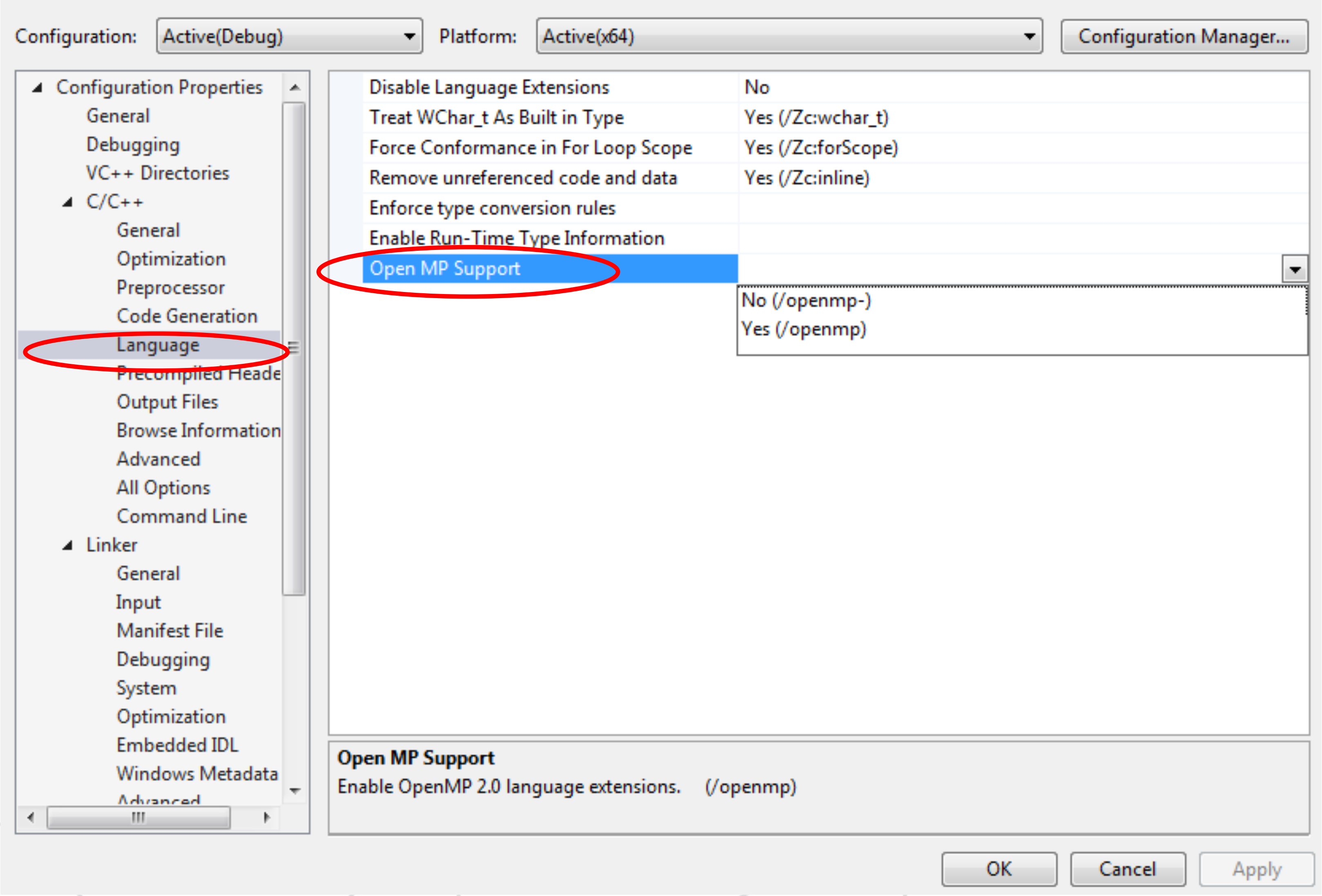
OpenMP中的并行化声明由#pragma完成

- 格式为`#pragma omp construct [clause [clause]...]`
  - 如`#pragma omp parallel for`
  - 编译器如果不支持该指令则将直接忽略
- 其作用范围通常为一个代码区块

```
#pragma omp parallel for
for (int i=0; i<10; ++i){
    std::cout << i << std::endl;
}
```

OpenMP环境配置

- **Windows**
  - 项目属性
    ->C/C++
    ->Language
    ->Open MP Support
- **macOS/Linux**
  - 对于支持OpenMP的编译器
    - gcc：在编译时增加 –fopenmp 标记
- **使用库函数**
  - #include <omp.h>



OpenMP环境配置

查看OpenMP版本

– 使用_OPENMP宏定义

```
#include <unordered_map>
#include <string>
#include <cstdio>
#include <omp.h>

int main(int argc, char *argv[]) {
    std::unordered_map<unsigned,std::string> map{
        {200505, "2.5"}, {200805, "3.0"},
        {201107, “3.1”}, {201307, "4.0"},
        {201511, "4.5"}};
    printf("OpenMP version: %s.\n", map.at(_OPENMP).c_str());
    return 0;
}
```

编译：g++ -fopenmp openmp.cpp -o openmp_example

OpenMP环境配置

- 查看OpenMP版本
  - 使用_OPENMP宏定义

```
#include <unordered_map>
#include <string>
#include <cstdio>
#include <omp.h>

int main(int argc, char *argv[]) {
    std::unordered_map<unsigned,std::string> map{
        {200505, "2.5"}, {200805, "3.0"},
        {201107, "3.1"}, {201307, "4.0"},
        {201511, "4.5"}};
    printf("OpenMP version: %s.\n", map.at(_OPENMP).c_str());
    return 0;
}
```

成功编译运行（学院GPU集群）：

```
./openmp_example
OpenMP version: 4.5.
```

OpenMP环境配置

查看OpenMP版本

– 使用_OPENMP宏定义

```cpp
#include <unordered_map>
#include <string>
#include <cstdio>
#include <omp.h>

int main(int argc, char *argv[]) {
    std::unordered_map<unsigned,std::string> map{
        {200505, "2.5"}, {200805, "3.0"},
        {201107, "3.1"}, {201307, "4.0"},
        {201511, "4.5"}};
    printf("OpenMP version: %s.\n", map.at(_OPENMP).c_str());
    return 0;
}
```

- macOS默认编译器不支持OpenMP报错:
```
clang: error: unsupported option '-fopenmp’
```
- 解决方案 – 安装 llvm clang:
```
brew install llvm
brew install libomp
echo 'export PATH="/usr/local/opt/llvm/bin:$PATH"' >> ~/.bash_profile
```
- 编译:
```
clang++ -fopenmp openmp.cpp -o openmp_example
```

OpenMP Hello world

- 通过#pragma omp parallel指明并行部分
- 无需改变串行代码

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

int main()
{
    #pragma omp parallel
    {
        printf("Hello World\n");
    }
    return 0;
}
```

输出：

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

OpenMP Hello world

- 在输出中增加线程编号
  - omp_get_thread_num();

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

int main()
{
    #pragma omp parallel
    {
        int thread = omp_get_thread_num();
        int max_threads = omp_get_max_threads();
        printf("Hello World (Thread %d of %d)\n", thread, max_threads);
    }
    return 0;
}
```

输出：

```
Hello World (Thread 0 of 8)
Hello World (Thread 4 of 8)
Hello World (Thread 1 of 8)
Hello World (Thread 7 of 8)
Hello World (Thread 3 of 8)
Hello World (Thread 2 of 8)
Hello World (Thread 6 of 8)
Hello World (Thread 5 of 8)
```

OpenMP Hello world

同一线程的多个语句是否连续执行?

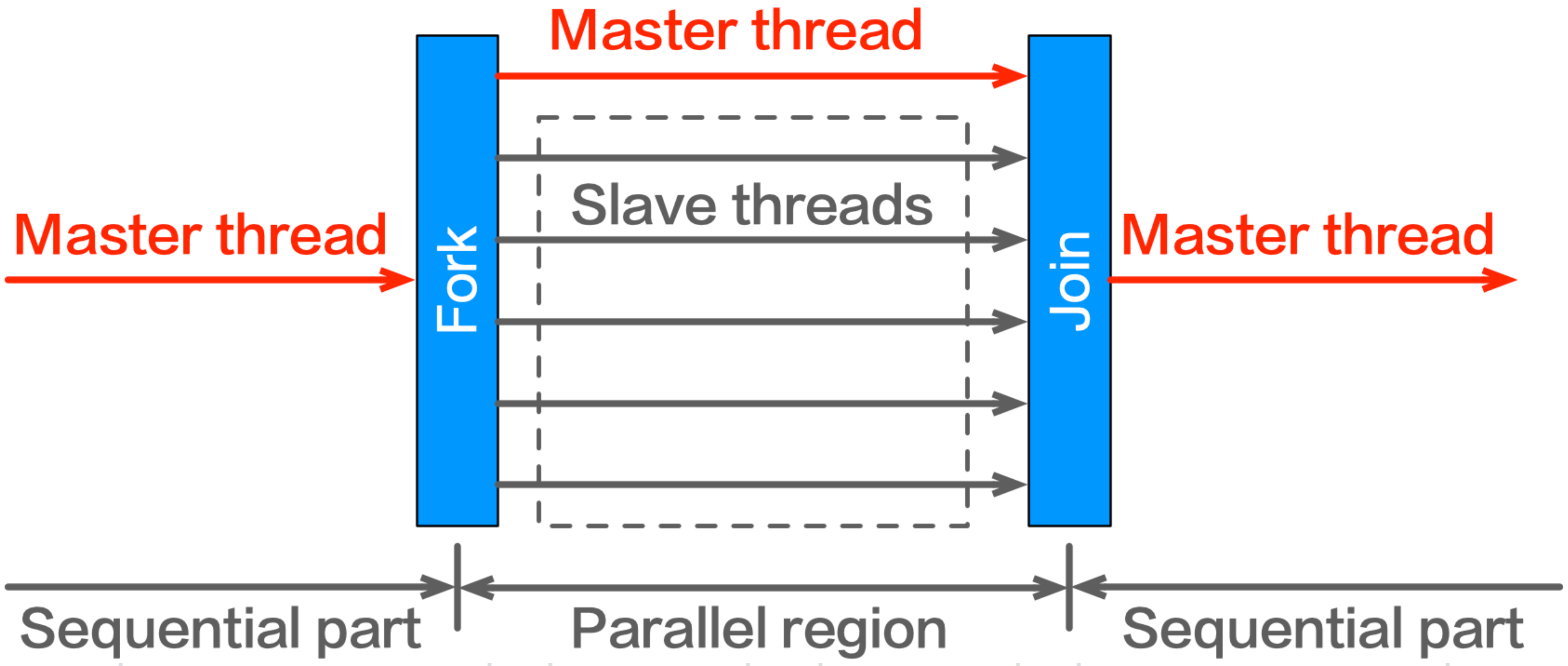
```c
#include <stdio.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
        int thread = omp_get_thread_num();
        printf("hello(%d) ", thread);
        printf("world(%d) ", thread);
    }
    return 0;
}
```

输出:

```
hello(0) world(0)
hello(4) hello(1)
world(1) hello(7)
hello(3) world(7)
world(3) hello(6)
world(6) hello(5)
world(5) hello(2)
world(4) world(2)
```

OpenMP运行机制

- 使用分叉（fork）与交汇（join）模型
  - Fork：由主线程（master thread）创建一组从线程（slave threads）
    - 主线程编号永远为0（thread 0）
    - 不保证执行顺序
  - Join：同步终止所有线程并将控制权转移回至主线程



课程概要

什么是并行？

什么是OpenMP？

语法

同步机制

变量作用域

线程调度

OpenMP语法

- 编译器指令
  - #pragma omp construct [clause [clause]...]{structured block}
  - 指明并行区域及并行方式
  - clause子句
    - 指明详细的并行参数
      - 控制变量在线程间的作用域
      - 显式指明线程数目
      - 条件并行

```
#pragma omp parallel num_threads(16)
{
    int thread = omp_get_thread_num();
    int max_threads = omp_get_max_threads();
    printf("Hello World (Thread %d of %d)\n", thread, max_threads);
}
```

OpenMP语法

- num_threads(int)
  - 用于指明线程数目
  - 当没有指明时，将默认使用OMP_NUM_THREADS环境变量
    - 环境变量的值为系统运算核心数目（或超线程数目）
    - 可以使用omp_set_num_threads(int)修改全局默认线程数
    - 可使用omp_get_num_threads()获取当前设定的默认线程数
    - num_threads(int)优先级高于环境变量
  - num_threads(int)不保证创建指定数目的线程
    - 系统资源限制

OpenMP语法

- 并行for循环
  - 将循环中的迭代分配到多个线程并行

```
#pragma omp parallel
{
    int n;
    for (n = 0; n < 4; n++){
        int thread = omp_get_thread_num();
        printf("thread %d \n", thread);
    }
}
```

输出是?

并行for循环

– 将循环中的迭代分配到多个线程并行

```
#pragma omp parallel
{
    int n;
    for (n = 0; n < 4; n++){
        int thread = omp_get_thread_num();
        printf("thread %d \n", thread);
    }
}
```

输出：

```
thread 3
thread 3
thread 0
thread 0
thread 0
thread 0
thread 1
thread 1
thread 1
thread 1
thread 3
thread 3
thread 5
thread 5
…
```

OpenMP语法

- 并行for循环
  - 将循环中的迭代分配到多个线程并行
    - 风格1：在并行区域内加入#pragma omp for

```
#pragma omp parallel
{
    int n;
    #pragma omp for
    for (n = 0; n < 4; n++){
        int thread = omp_get_thread_num();
        printf("thread %d \n", thread);
    }
}
```

在并行区域内，for循环外还可以加入其它并行代码

    - 风格2：合并为#pragma omp parallel for

```
int n;
#pragma omp parallel for
for (n = 0; n < 4; n++) {
    int thread = omp_get_thread_num();
    printf("thread %d \n", thread);
}
```

写法更简洁

OpenMP语法

并行for循环

– 将循环中的迭代分配到多个线程并行

- 风格1：在并行区域内加入#pragma omp for

```
#pragma omp parallel
{
    int n;
    #pragma omp for
    for (n = 0; n < 4; n++){
        int thread = omp_get_thread_num();
        printf("thread %d \n", thread);
    }
}
```

输出：

```
thread 0
thread 2
thread 3
thread 1
```

- 风格2：合并为#pragma omp parallel for

```
int n;
#pragma omp parallel for
for (n = 0; n < 4; n++) {
    int thread = omp_get_thread_num();
    printf("thread %d \n", thread);
}
```

嵌套并行

- OpenMP中的每个线程同样可以被并行化为一组线程
  - OpenMP默认关闭嵌套
    - 需要使用omp_set_nested(1)打开

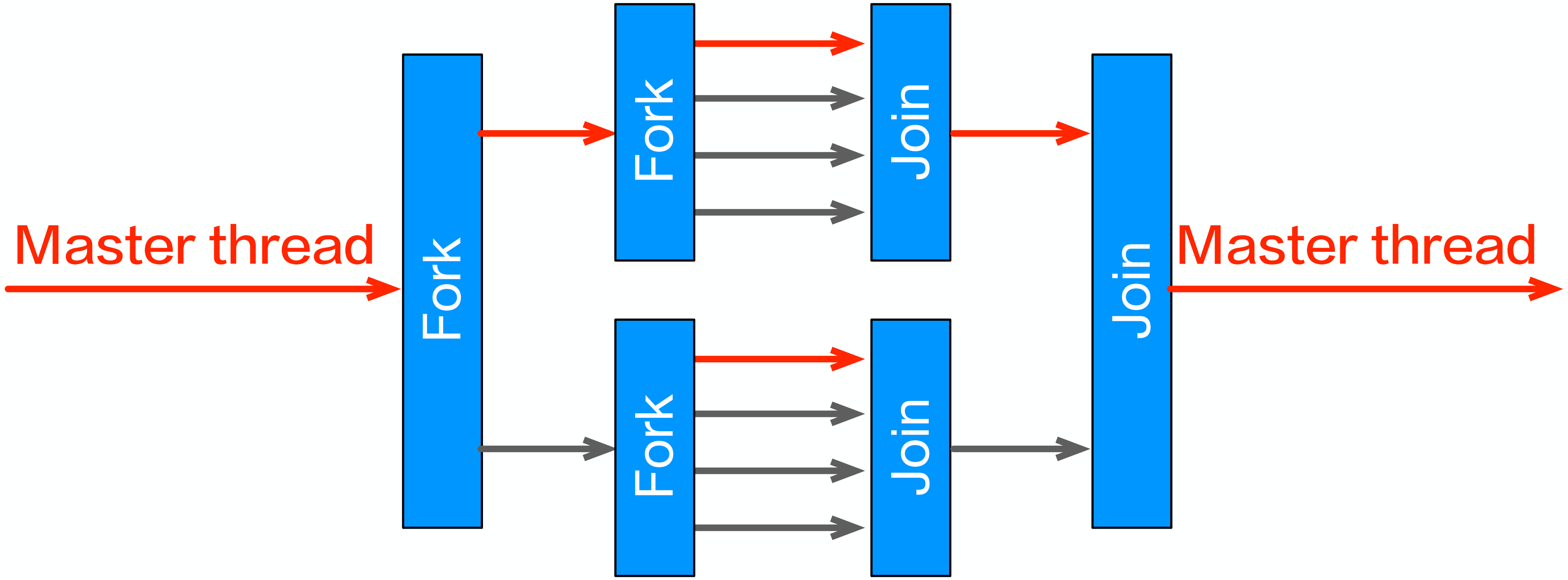
```
omp_set_nested(1);

#pragma omp parallel for
for (int i = 0; i < 2; i++){
    int outer_thread = omp_get_thread_num();

    #pragma omp parallel for
    for (int j = 0; j < 4; j++){
        int inner_thread = omp_get_thread_num();
        printf("Hello World (i = %d j = %d)\n",
               outer_thread, inner_thread);
    }
}
```

输出：

```
Hello World (i = 0 j = 0)
Hello World (i = 1 j = 0)
Hello World (i = 0 j = 2)
Hello World (i = 0 j = 3)
Hello World (i = 1 j = 1)
Hello World (i = 1 j = 2)
Hello World (i = 1 j = 3)
Hello World (i = 0 j = 1)
```

嵌套并行

- OpenMP中的每个线程同样可以被并行化为一组线程
  - 仍然使用fork and join



不能并行的循环

- 语法限制
  - 不能使用!=作为判断条件
    - `for (int i = 0; i!=8; ++i){`
    - error: condition of OpenMP for loop must be a relational comparison (‘<’, ‘<=’, ‘>’, or ‘>=’) of loop variable ‘i’
  - 循环必须为单入口单出口
    - 不能使用break、goto等跳转语句
    - error: 'break' statement cannot be used in OpenMP for loop
  - （以上错误提示来自OpenMP 3.1）

不能并行的循环

数据依赖性

- 循环迭代相关（loop-carried dependence）
  - 依赖性与循环相关，去除循环则依赖性不存在
- 非循环迭代相关（loop-independent dependence）
  - 依赖性与循环无关，去除循环依赖性仍然存在

```
for (i = 1; i < n; i++){
    S1: a[i] = a[i - 1] + 1;
    S2: b[i] = a[i];
}
```

S1[i] → S1[i+1]: 循环相关
S1[i] → S2[i]: 循环无关

```
for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
        S3: a[i][j] = a[i][j - 1] + 1;
```

S3[i,j] → S3[i,j+1]:
i循环无关，j循环相关

```
for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
        S4: a[i][j] = a[i - 1][j] + 1;
```

S4[i,j] → S4[i+1,j]:
i循环相关，j循环无关

课程概要

- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度

线程互动与同步

- OpenMP是多线程共享地址架构
  – 线程可通过共享变量通信
- 线程及其语句执行具有不确定性
  – 共享数据可能造成竞争条件（race condition）
  – 竞争条件：程序运行的结果依赖于不可控的执行顺序
- 必须使用同步机制避免竞争条件的出现
  – 同步机制将带来巨大开销
  – 尽可能改变数据的访问方式减少必须的同步次数

竞争条件

- 语句执行顺序造成结果不一致

```
int a[3] = { 3, 4, 5};

thread 1                          thread 2
a[1] = a[0] + a[1];               a[2] = a[1] + a[2];
```

a = { 3, ?, ? }

竞争条件

- 语句执行顺序造成结果不一致

```
int a[3] = { 3, 4, 5};

thread 1                          thread 2
a[1] = a[0] + a[1];               a[2] = a[1] + a[2];
```

a = { 3, ?, ? }

- 先执行 thread 1 再执行 thread 2
  - a[1]=a[0]+a[1]=3+4=7; a[2]=a[1]+a[2]=7+5=12;
  - a = { 3, 7, 12}
- 先执行 thread 2 再执行 thread 1
  - a[2]=a[1]+a[2]=4+5=9; a[1]=a[0]+a[1]=3+4=7;
  - a = { 3, 7, 9}

竞争条件

- 高级语言的语句并非原子操作

```
int count=10;

thread 1          thread 2

count++;          count--;
```

count = 9, 10, 11?

竞争条件

- 高级语言的语句并非原子操作

```
int count=10;
```

thread 1	**thread 2**
LOAD Reg, count	LOAD Reg, count
ADD #1	SUB #1
STORE Reg, count	STORE Reg, count

count = 9, 10, 11?

竞争条件

- 高级语言的语句并非原子操作

```
int count=10;
```

thread 1	Reg	count	Reg	thread 2
LOAD	10	10		
ADD	11	10		
	11	10	10	LOAD
	11	10	9	SUB
	11	9	9	STORE
STORE	11	11		

count = 11

竞争条件

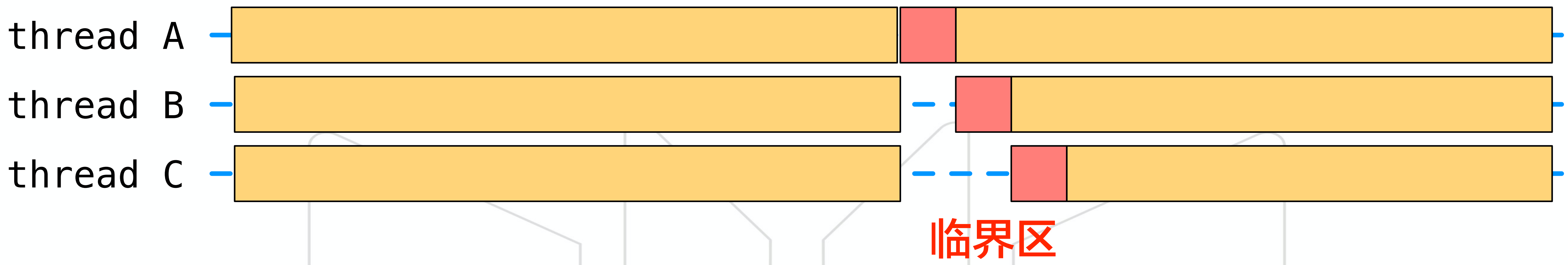
- 高级语言的语句并非原子操作

```
               int count=10;
      thread 1                  thread 2
            Reg      count      Reg
                       10        10     LOAD
LOAD        10         10        10
ADD         11         10        10
            11         10         9     SUB
STORE       11         11         9
            11          9         9     STORE
```

count = 9

如何保证执行结果正确？

- 临界区（critical section）
  - `#pragma omp critical`
  - 指的是一个访问共用资源（例如：共用设备或是共用存储器）的程序片段，而这些共用资源又无法同时被多个线程访问的特性
    - 同一时间内只有一个线程能执行临界区内代码
    - 其他线程必须等待临界区内线程执行完毕后才能进入临界区
    - 常用来保证对共享数据的访问之间互斥



如何保证执行结果正确？

- 临界区（critical section）
  - #pragma omp critical
  - 比照操作系统中信号量（semaphore）与P、V操作

```
#pragma omp critical
{
    ...
    critical section;
    ...
}
```

```
Semaphore a;
P(a);
...
critical section;
...
V(a);
```

如何保证执行结果正确？

临界区（critical section）

– 举例：统计随机数分布
  - 随机产生1000个[0-20)之间的整数
  - 统计每个数字出现频率

无临界区：

```
#pragma omp parallel for
for(int i=0; i<1000; ++i){
    int value = rand()%20;
    histogram[value]++;
}

int total = 0;
for(int i = 0; i < 20; i++){
    total += histogram[i];
    cout<<histogram[i]<<" ";
}
cout<<endl<<"total: "<<total<<endl;
```

输出：

```
25 31 26 34 40 47 24 29 44 44
31 26 41 38 32 45 26 54 45 27
total: 709
```

如何保证执行结果正确？

- 临界区（critical section）
  - 举例：统计随机数分布
    - 随机产生1000个[0-20)之间的整数
    - 统计每个数字出现频率

有临界区：

```
#pragma omp parallel for
for(int i=0; i<1000; ++i){
    int value = rand()%20;

    #pragma omp critical
    {
        histogram[value]++;
    }
}
```

输出：

```
60 47 28 54 52 50 33 56 44 53
61 58 43 47 52 54 50 52 53 53
total: 1000
```

如何保证执行结果正确？

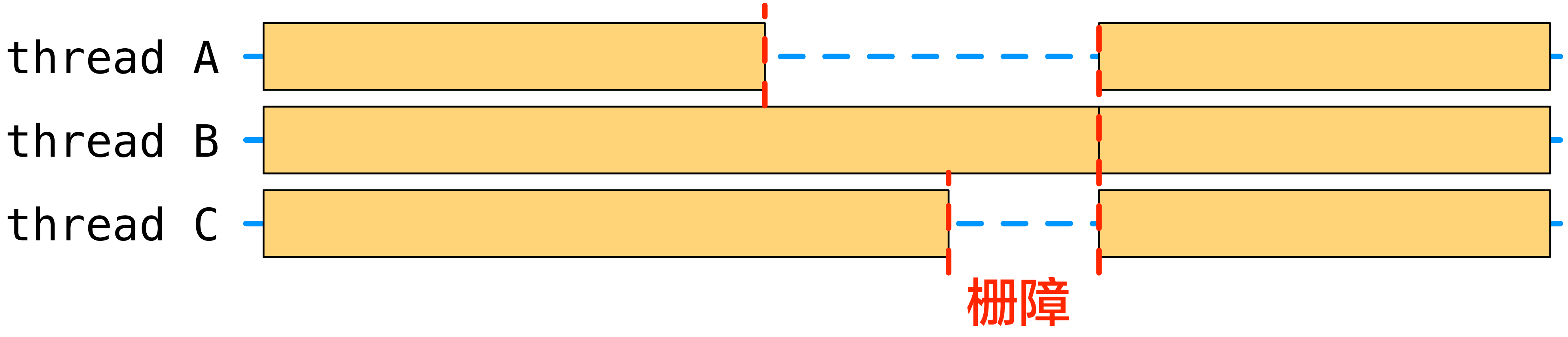
- 原子（atomic）操作
  - #pragma omp atomic
  - 保证下一语句在同一时间只能被一个线程执行
    - 常用来做计数器、求和等
  - 原子操作通常别临界区执行更快
  - 临界区的作用范围更广，能够实现的功能更复杂

```
#pragma omp parallel for
for(int i=0; i<1000; ++i){
    int value = rand()%20;
    #pragma omp atomic
    histogram[value]++;
}
```

其他同步机制

- 栅障（barrier）
  - #pragma omp barrier
  - 在栅障点处同步所有线程
    - 先运行至栅障点处的线程必须等待其他线程
    - 常用来等待某部分任务完成再开始下一部分任务
    - 每个并行区域的结束点默认自动同步线程

```
#pragma omp parallel
{
    function_A()
    #pragma omp barrier
    function_B();
}
```



其他同步机制

- 栅障（barrier）
  - 并行随机数统计及并行求和

```
int total = 0;

#pragma omp parallel num_threads(20)
{
    for(int i=0; i<50; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }

    int thread = omp_get_thread_num();
    #pragma omp atomic
    total += histogram[thread];
}
```

输出：

```
total: 619
```

求和时可能其他线程还没完成统计

其他同步机制

- 栅障（barrier）
  - 并行随机数统计及并行求和

```
int total = 0;

#pragma omp parallel num_threads(20)
{
    for(int i=0; i<50; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }
    #pragma omp barrier
    int thread = omp_get_thread_num();
    #pragma omp atomic
    total += histogram[thread];
}
```

使用栅障同步线程

输出：
```
total: 1000
```

其他同步机制

- 栅障（barrier）
  - 并行随机数统计及并行求和
    - 这两段代码结果是否相同？

```
int total = 0;

#pragma omp parallel num_threads(20)
{
    for(int i=0; i<50; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }
    #pragma omp barrier
    int thread = omp_get_thread_num();
    #pragma omp atomic
    total += histogram[thread];
}
```

```
int total = 0;

#pragma omp parallel num_threads(20)
{
    #pragma omp for
    for(int i=0; i<1000; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }
    int thread = omp_get_thread_num();
    #pragma omp atomic
    total += histogram[thread];
}
```

single & master

- `#pragma omp single {}`
  - 用于保证{}内的代码由一个线程完成
  - 常用于输入输出或初始化
  - 由第一个执行至此处的线程执行
  - 同样会产生一个隐式栅障
    - 可由`#pragma omp single` **`nowait`**去除
- `#pragma omp master {}`
  - 与single相似，但指明由主线程执行
  - 与使用IF的条件并行等价
    - `#pragma omp parallel IF(omp_get_thread_num() == 0) nowait`
    - 默认不产生隐式栅障

single & master

- `#pragma omp master {}`
  - 在下面代码中与atomic等价

```
int total = 0;

#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<1000; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }
    #pragma omp master
    {
        for(int i=0; i<20; ++i){
            total += histogram[i];
        }
    }
}
```

```
int total = 0;

#pragma omp parallel num_threads(20)
{
    #pragma omp for
    for(int i=0; i<1000; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }

    int thread = omp_get_thread_num();
    #pragma omp atomic
    total += histogram[thread];
}
```

并行Reduction

- 指明如何将线程局部结果汇总
  - 如#pragma omp for **reduction(+: total)**
  - 支持的操作： +, -, *, & , |, && and ||

```
int total = 0;

#pragma omp parallel num_threads(20)
{
    for(int i=0; i<50; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }
    #pragma omp barrier
    int thread = omp_get_thread_num();
    #pragma omp atomic
    total += histogram[thread];
}
```

```
int total = 0;

#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<1000; ++i){
        int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }
    #pragma omp for reduction(+: total)
    for(int i=0; i<20; ++i){
        total += histogram[i];
    }
}
```

课程概要

- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度

变量作用域

- OpenMP与串行程序的作用域不同
  - OpenMP中必须指明变量为shared或private
    - Shared：变量为所有线程所共享
      - 并行区域外定义的变量默认为shared
    - Private：变量为线程私有，其他线程无法访问
      - 并行区域内定义的变量默认为private
      - 循环计数器默认为private

变量作用域

- Shared 与 private

```
int histogram[20];  ← shared
init_histogram(histogram);

int total = 0;      ← shared

#pragma omp parallel
{
    int i;  ← 循环计数器为private!
    #pragma omp for
    for(i=0; i<1000; ++i){
private →   int value = rand()%20;
        #pragma omp atomic
        histogram[value]++;
    }
}
```

变量作用域

显式作用域定义

- 显式指明变量的作用域
- shared（var）
  - 指明变量var为shared
- default(none/shared/private)
  - 指明变量的默认作用域
  - 如果为none则必须指明并行区域内每一变量的作用域

```
int a, b = 0, c;
#pragma omp parallel default(none) shared(b)
{
    b += a;
}
```

```
error: variable 'a' must have explicitly specified data
sharing attributes
```

变量作用域

显式作用域定义

- private（var）
  - 指明变量var为private

```
int i = 10;
#pragma omp parallel for private(i)
for (int j=0; j<4; ++j) {
    printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
}
printf("i = %d\n", i);
```

输出：
```
Thread 0: i = 1
Thread 1: i = 0
Thread 3: i = 0
Thread 2: i = 0
i = 10
```

- firstprivate(var)
  - 指明变量var为private，同时表明该变量使用master thread中变量值初始化

```
int i = 10;
#pragma omp parallel for firstprivate(i)
for (int j=0; j<4; ++j) {
    printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
}
printf("i = %d\n", i);
```

输出：
```
Thread 0: i = 10
Thread 3: i = 10
Thread 2: i = 10
Thread 1: i = 10
i = 10
```

变量作用域

显式作用域定义

- private (var)
  - 指明变量var为private

```
int i = 10;
#pragma omp parallel for private(i)
for (int j=0; j<4; ++j) {
    printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
}
printf("i = %d\n", i);
```

输出：

```
Thread 0: i = 1
Thread 1: i = 0
Thread 3: i = 0
Thread 2: i = 0
i = 10
```

- lastprivate(var)
  - 指明变量var为private，同时表明结束后一层迭代将结果赋予该变量

```
int i = 10;
#pragma omp parallel for lastprivate(i)
for (int j=0; j<4; ++j) {
    printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
}
printf("i = %d\n", i);
```

输出：

```
Thread 0: i = 1
Thread 3: i = 0
Thread 1: i = 0
Thread 2: i = 0
i = 0
```

数据并行与任务并行

数据并行

– 同样指令作用在不同数据上

– 前述例子均为数据并行

任务并行

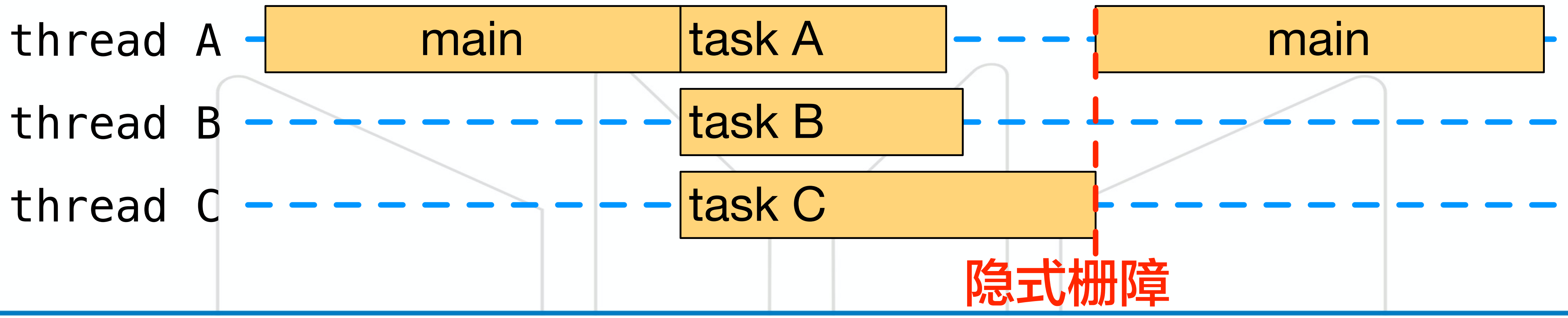
– 线程可能执行不同任务

– `#pragma omp sections`

– 每个section由一个线程完成

– 同样有隐式栅障（可使用nowait去除）

```
#pragma omp parallel

#pragma omp sections
{
    #pragma omp section
    task_A();
    #pragma omp section
    task_B();
    #pragma omp section
    task_C();
}
```



课程概要

- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度

线程调度

- 当迭代数多于线程数时，需要调度线程
  - 某些线程将执行多个迭代
  - #pragma omp parallel for schedule(type,[chunk size])
    - type 包括 static, dynamic, guided, auto, runtime
    - 默认为static

```
#pragma omp parallel for num_threads(4)
for (int i=0; i<6; ++i)
{
    int thread = omp_get_thread_num();
    printf("thread %d\n", thread);
}
```

输出：

```
thread 1
thread 1
thread 3
thread 0
thread 0
thread 2
```

线程调度

- Static调度
  - 调度由编译器静态决定
  - #pragma omp parallel for schedule(type,[chunk size])
    - 每个线程轮流获取 chunk size 个迭代任务
    - 默认chunk size 为 n/threads

(static, 1)

thread 0	0	4	8	12
thread 1	1	5	9	13
thread 2	2	6	10	14
thread 3	3	7	11	15

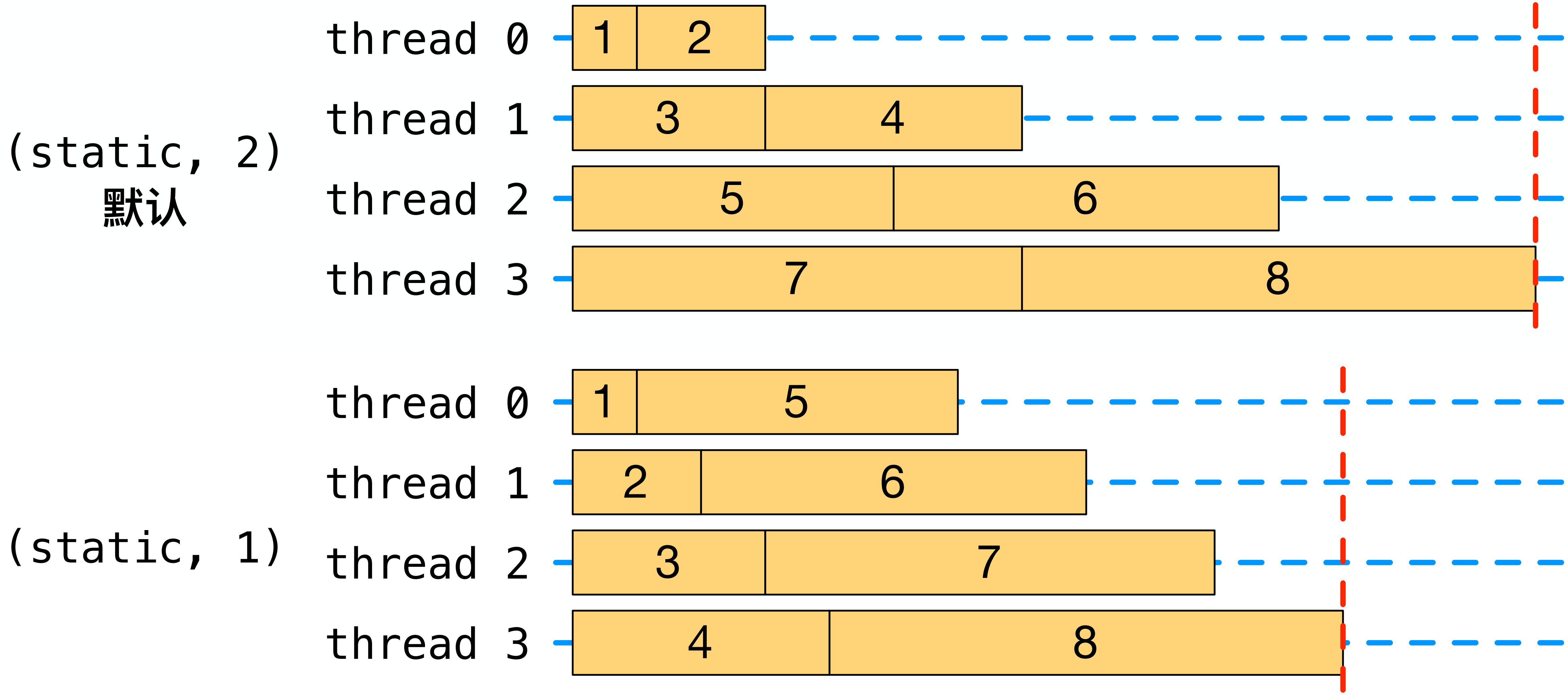
(static, 2)

thread 0	0	1	8	9
thread 1	2	3	10	11
thread 2	4	5	12	13
thread 3	6	7	14	15

(static, 4)

thread 0	0	1	2	3
thread 1	4	5	6	7
thread 2	8	9	10	11
thread 3	12	13	14	15

- Static调度
  - 线程的负载可能不均匀



线程调度

- Dynamic调度
  - 在运行中动态分配任务
  - 迭代任务依然根据chunk size划分成块
  - 线程完成一个chunk后向系统请求下一个chunk
- Guided调度
  - 与dynamic类似
  - 但分配的chunk大小在运行中递减
    - 最小不能小于chunk size参数
- Auto 与 runtime
  - “Note that keywords auto and runtime aren’t adequate.”

https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdf

OpenMP陷阱

- 当#pragma指令无法为编译器理解时
  - 不会报错!
  - 错在哪儿?
    - `#pragma omg parallel`
- 参考OpenMP的32个常见陷阱
  - https://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers

资料

- OpenMP Reference Guide
  - https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf

OpenMP API 4.5 C/C++ Page 1

OpenMP C/C++

OpenMP 4.5 API C/C++ Syntax Reference Guide

OpenMP Application Program Interface (API) is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications. OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows platforms. See www.openmp.org for specifications.

- Text in this color indicates functionality that is new or changed in the OpenMP API 4.5 specification.
- [n.n.n] Refers to sections in the OpenMP API 4.5 specification.
- [n.n.n] Refers to sections in the OpenMP API 4.0 specification.

Directives and Constructs for C/C++

An OpenMP executable directive applies to the succeeding structured block or an OpenMP construct. Each directive starts with **#pragma omp**. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. A *structured-block* is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

parallel [2.5] [2.5]
Forms a team of threads and starts parallel execution.

#pragma omp parallel *[clause[[,]clause] ...]*
structured-block

clause:
if(*[* **parallel :** *] scalar-expression*)
num_threads(*integer-expression*)
default(**shared** | **none**)
private(*list*)
firstprivate(*list*)
shared(*list*)
copyin(*list*)
reduction(*reduction-identifier*: *list*)
proc_bind(**master** | **close** | **spread**)

sections [2.7.2] [2.7.2]
A noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

#pragma omp sections *[clause[[,] clause] ...]*
{
*[***#pragma omp section***]*
structured-block
*[***#pragma omp section**
structured-block]
...
}

clause:
private(*list*)
firstprivate(*list*)

for simd [2.8.3] [2.8.3]
Specifies that a loop that can be executed concurrently using SIMD instructions, and that those iterations will also be executed in parallel by threads in the team.

#pragma omp for simd *[clause[[,]clause] ...]*
for-loops

clause:
Any accepted by the **simd** or **for** directives with identical meanings and restrictions.

task [2.9.1] [2.11.1]
Defines an explicit task. The data environment of the task is created according to data-sharing attribute clauses on **task** construct and any defaults that apply.

OpenMP小结

- 软硬件环境
  - CPU多线程并行库
    - 编译器指令、库函数、环境变量
  - 共享内存的多核系统
- 基本语法
  - `#pragma omp construct [clause [clause]...]{structured block}`
  - 指明并行区域：`#pragma omp parallel`
  - 循环：`#pragma omp (parallel) for`
  - 嵌套：`omp_set_nested(1)`
  - 常用函数： `omp_get_thread_num();num_threads(int);`
  - 同步： `#pragma omp critical/atomic/barrier`、`nowait`
  - 变量作用域：`default(none/shared/private), shared(), private(), firstprivate(), last private()`
  - 调度：`schedule(static/dynamic/guided, [chunk_size])`

Questions?