



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# 并行程序设计 with 算法

## OpenMP 程序设计

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

中山大学 计算机学院  
国家超级计算广州中心

- 什么是OpenMP?
- 语法
- 同步机制
- 变量作用域
- 线程调度
- 生产者-消费者问题及其他

## • 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

### – 支持共享内存的多处理器/多核系统

- 与Pthreads相似；与MPI、CUDA不同（讲义1, 3, 4）

Compiler directives

OpenMP library

Environment variables

OpenMP runtime library

OS support for shared memory and threading

## • 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指令，否则将被忽略
- 不属于C/C++标准的一部分

### – 举例：#pragma once

- 指定头文件只被编译一次

#### #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

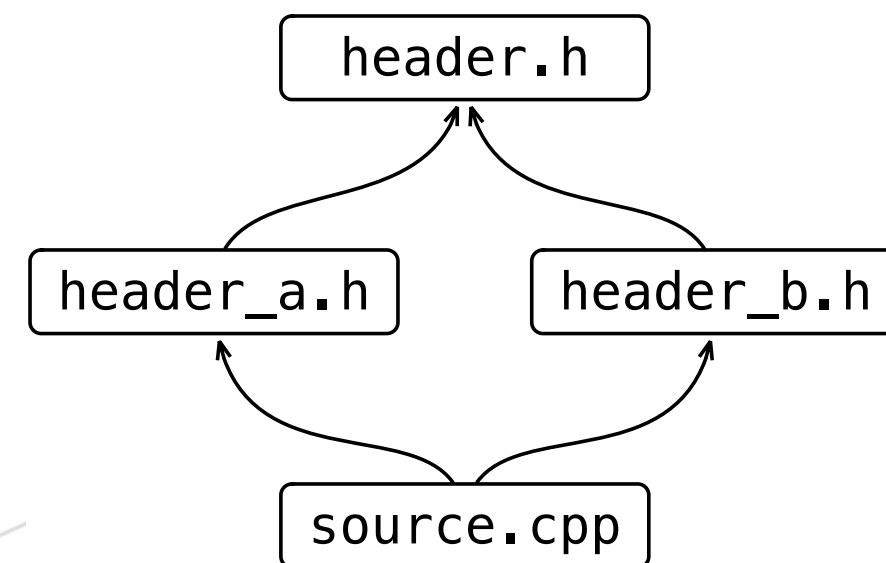
#### #ifndef

- 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

```
#ifndef HEADER_H  
#define HEADER_H
```

...

```
#endif //HEADER_H
```



- 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;  
}
```

## • MacOS/Linux

– 对于支持OpenMP的编译器

- gcc: 在编译时增加 **-fopenmp** 标记

- **gcc** -g -Wall **-fopenmp** -o omp\_hello omp\_hello . c

## • Windows

– 项目属性

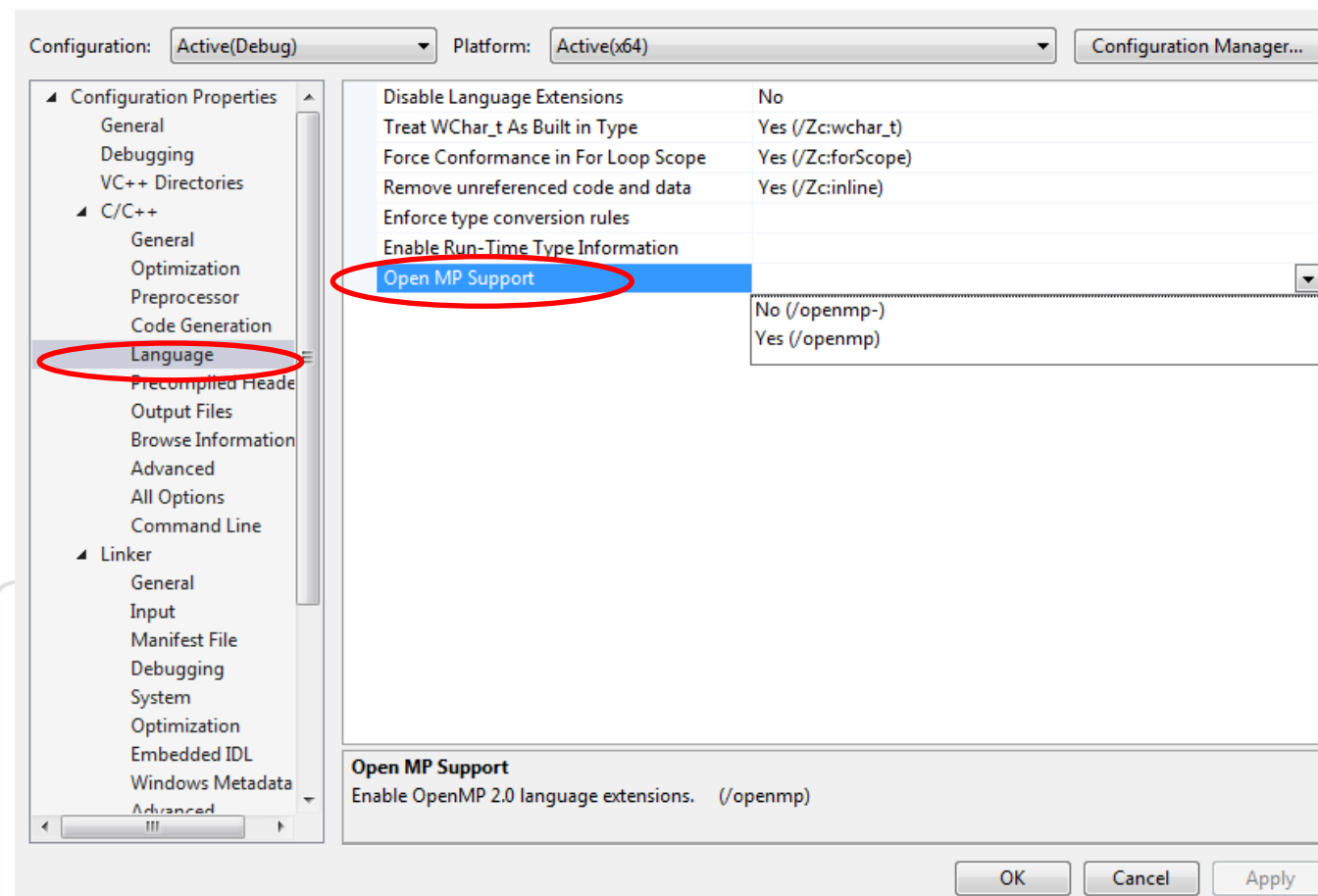
->C/C++

->Language

->Open MP Support

## • 使用库函数

– `#include <omp.h>`



## ● 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

## ● 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.



## 查看OpenMP版本

### – 使用\_OPENMP宏定义

```
#include <unordered_map>
#include <string>
#include <stdio>
#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

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

```
#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

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

```
#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)

- 同一线程的多个语句是否连续执行？

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

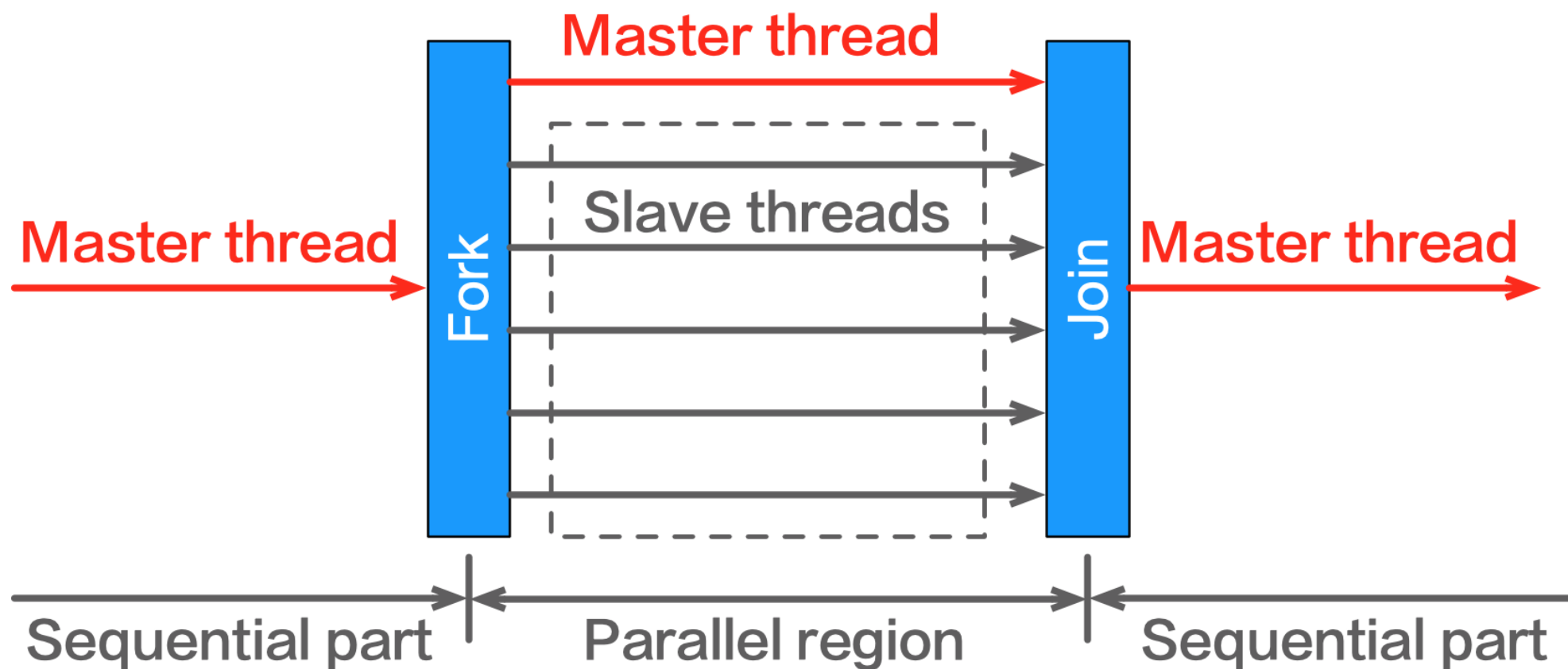
int main()
{
    #pragma omp parallel
    {
        int thread = omp_get_thread_num();
        printf("hello(%d) ", thread);
        printf("world(%d) ", thread);
    }
    return 0;
}
```

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)
```

## 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



- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度
- 生产者-消费者问题及其他

## ◉ 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);
}
```



## • 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



## • 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
```

...

## • 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

## • 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

思考：

n=?

- 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);
}
```

## • 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);

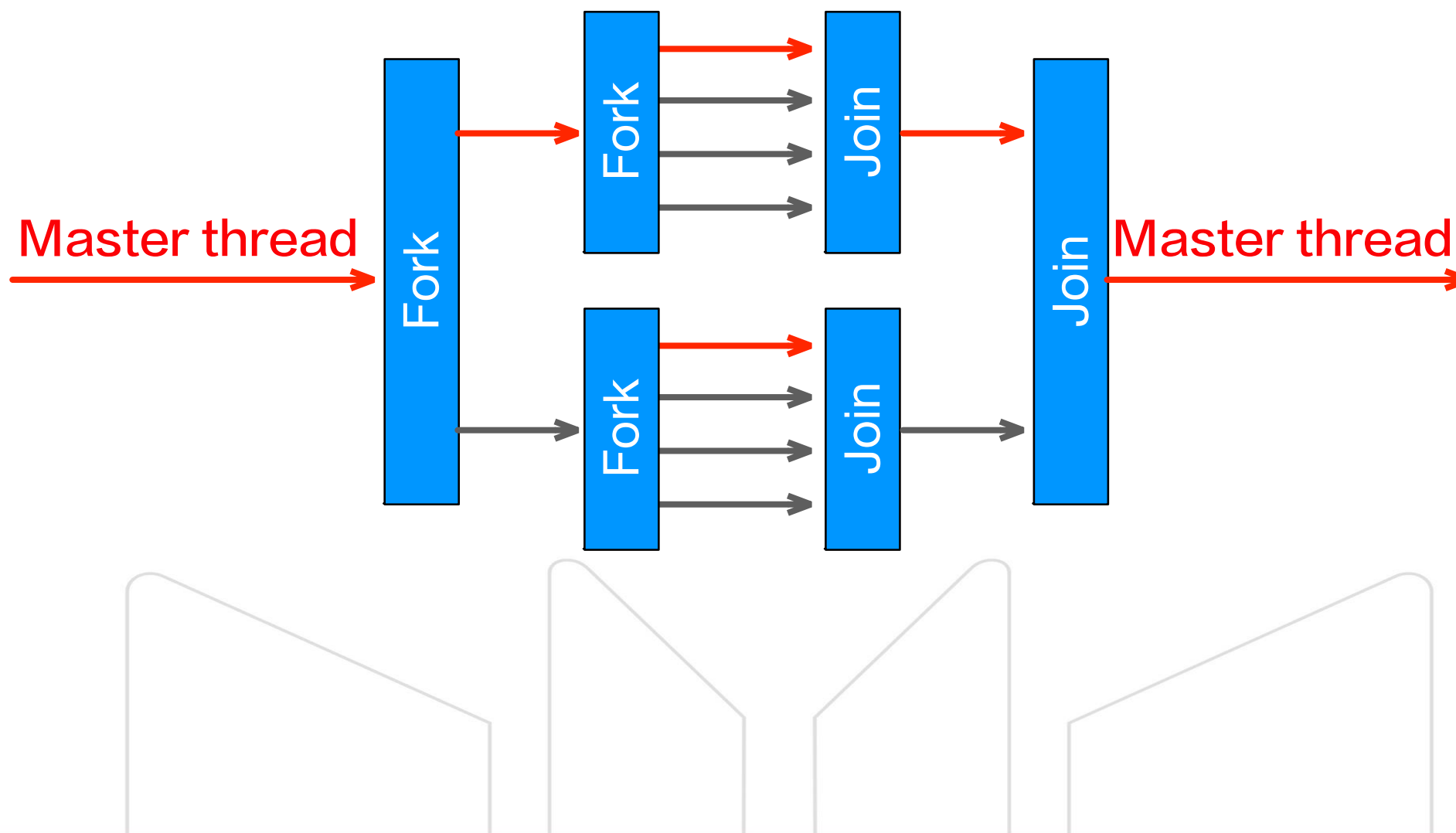
#pragma omp parallel for
for (int i = 0; i < 2; i++){
    int outer_thread = omp_get_thread_num();

    #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)
```

- 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



## ◉ 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）

## 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]  $\rightarrow$  S1[i+1]: 循环相关  
S2[i]  $\rightarrow$  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]  $\rightarrow$  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]  $\rightarrow$  S4[i+1,j]:  
i循环相关, j循环无关

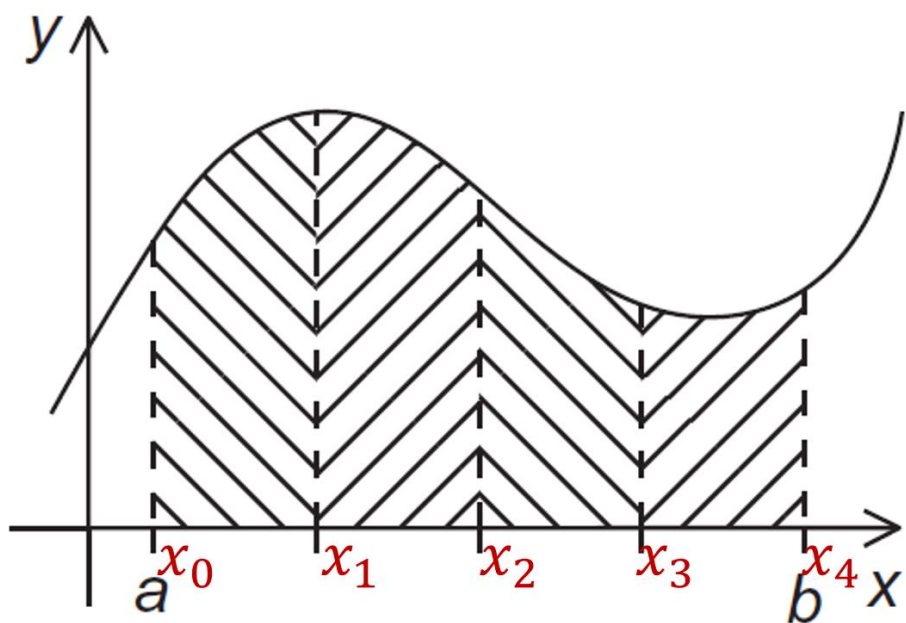


- 什么是并行?
- 什么是OpenMP?
- 语法
- 同步机制
- 变量作用域
- 线程调度
- 生产者-消费者问题及其他

- 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

- 梯形积分法估算 $y = f(x)$ 在 $[a, b]$ 区间上的积分

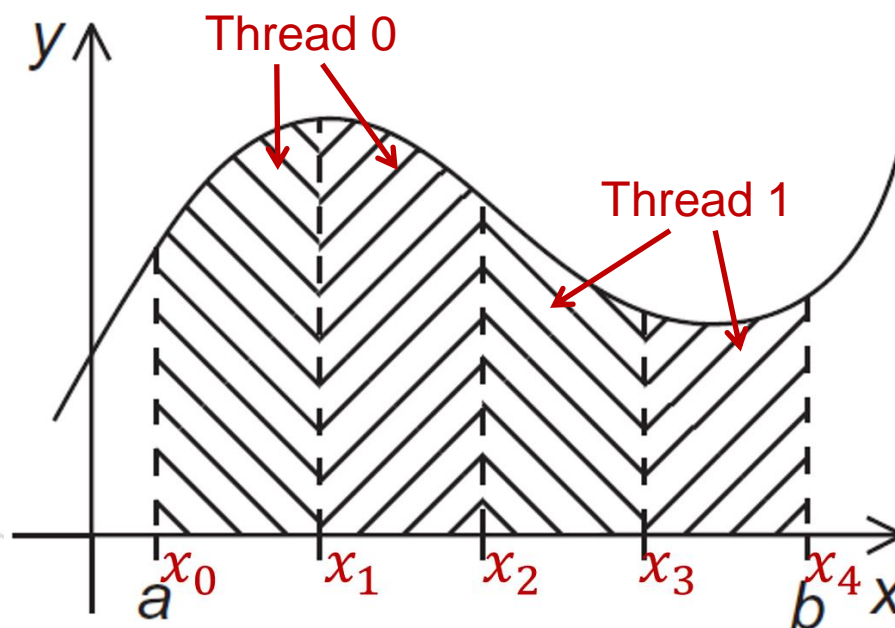
– 总面积为 $h(\frac{1}{2}f(x_0) + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n))$



```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i < n; ++i){  
    x_i = a + i*h;  
    approx += f(x_i);  
}  
approx = h*approx;
```

- 梯形积分法估算 $y = f(x)$ 在 $[a, b]$ 区间上的积分

- 梯形数量远超线程数量，因此每个线程负责一段连续的数据
- 任务1：每个线程计算一段子区间上的积分
- 任务2：对线程局部积分的结果求和



- 梯形积分法估算  $y = f(x)$  在  $[a, b]$  区间上的积分
  - 任务2：对线程局部积分的结果求和
    - 使用 `global_result += my_result` 显然会造成竞争条件

Time	Thread 0	Thread 1
0	<code>global_result = 0</code> to register	finish <code>my_result</code>
1	<code>my_result = 1</code> to register	<code>global_result = 0</code> to register
2	add <code>my_result</code> to <code>global_result</code>	<code>my_result = 2</code> to register
3	store <code>global_result = 1</code>	add <code>my_result</code> to <code>global_result</code>
4		store <code>global_result = 2</code>

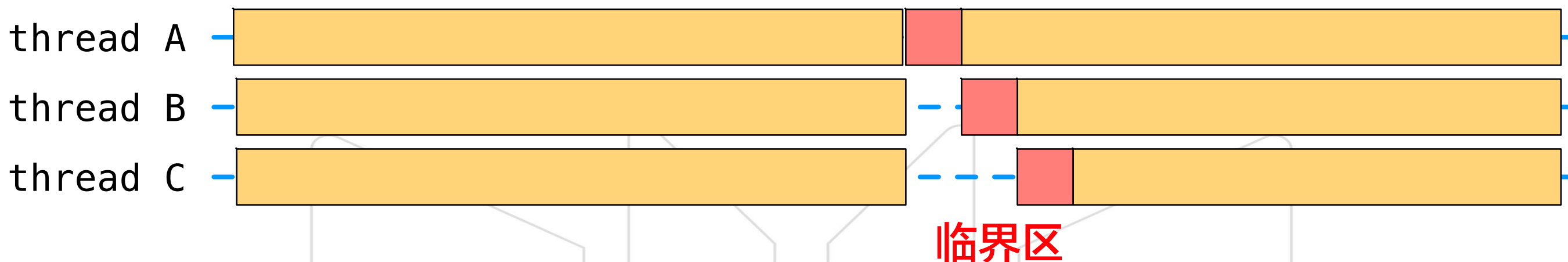


## ● 临界区 (critical section)

— `#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



## ◉ 临界区 (critical section)

- #pragma omp critical
- 与此前Pthreads临界区实现对比 (讲义4)

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

```
Semaphore a;  
wait(a);  
...  
critical section;  
...  
post(a);
```

```
Mutex mutex;  
lock(mutex);  
...  
critical section;  
...  
unlock(mutex);
```



```
void Trap(double a, double b, int n, double* global_result_p);
```

```
int main(int argc, char* argv[]) {  
    double global_result = 0.0; /* Store result in global_result */  
    double a, b; /* Left and right endpoints */  
    int n; /* Total number of trapezoids */  
    int thread_count;
```

```
    if (argc != 2) Usage(argv[0]);  
    thread_count = strtol(argv[1], NULL, 10);  
    printf("Enter a, b, and n\n");  
    scanf("%lf %lf %d", &a, &b, &n);  
    if (n % thread_count != 0) Usage(argv[0]);
```

```
# pragma omp parallel num_threads(thread_count) ← 指明函数Trap并行执行  
    Trap(a, b, n, &global_result);
```

```
    printf("With n = %d trapezoids, our estimate\n", n);  
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);  
    return 0;  
} /* main */
```



```
void Trap(double a, double b, int n, double* global_result_p) {
```

```
    double h, x, my_result;
```

```
    double local_a, local_b;
```

```
    int i, local_n;
```

```
    int my_rank = omp_get_thread_num();
```

```
    int thread_count = omp_get_num_threads();
```

获得线程编号及总线程数量

```
    h = (b-a)/n;
```

```
    local_n = n/thread_count;
```

```
    local_a = a + my_rank*local_n*h;
```

```
    local_b = local_a + local_n*h;
```

```
    my_result = (f(local_a) + f(local_b))/2.0;
```

```
    for (i = 1; i <= local_n-1; i++) {
```

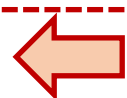
```
        x = local_a + i*h;
```

```
        my_result += f(x);
```

```
    }
```

```
    my_result = my_result*h;
```

```
# pragma omp critical
```



指明下一语句为临界区

```
    *global_result_p += my_result;
```

```
} /* Trap */
```

- 使用并行规约进行求和
  - 首先改进此前的编程风格

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
#     pragma omp critical
    global_result += Local_trap(a, b, n);
}
```



- 使用并行规约进行求和
  - 首先改进此前的编程风格

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0;
    my_result += Local_trap(a, b, n);
# pragma omp critical
    global_result += my_result; 只将求和放到临界区
}
```



## • 使用并行规约进行求和

- #pragma omp ... reduction(+: variable name)
- 支持的操作: +, -, \*, &, |, && and ||

并行规约

```
global_result = 0.0;  
# pragma omp parallel num_threads(thread_count) reduction(+: global_result)  
    global_result += Local_trap(a, b, n);
```

## 原子 (atomic) 操作

- `#pragma omp atomic`
- 保证对内存的读写更新等操作在同一时间只能被一个线程执行
  - 常用来做计数器、求和等
- 原子操作通常比临界区执行更快
  - 不需要阻塞其他线程
- `atomic`指令只支持特定单一赋值语句

### 性能对比

	critical	atomic
4 threads	0.917	0.188
10 threads	4.109	0.153
1000 threads	8.802	0.644

```
#pragma omp parallel for
for(int i=0; i<10000000; ++i){
    int value = rand()%20;
    #pragma omp atomic
    histogram[value]++;
}
```

扩展阅读: <https://www.ibm.com/docs/es/xl-c-and-cpp-linux/16.1.0?topic=parallelization-pragma-omp-atomic>

## ◉ 原子 (atomic) 操作

– #pragma omp atomic

– atomic操作只支持特定单一赋值语句

- 自增/自减:  $++x$ ,  $x++$ ,  $--x$ ,  $x--$

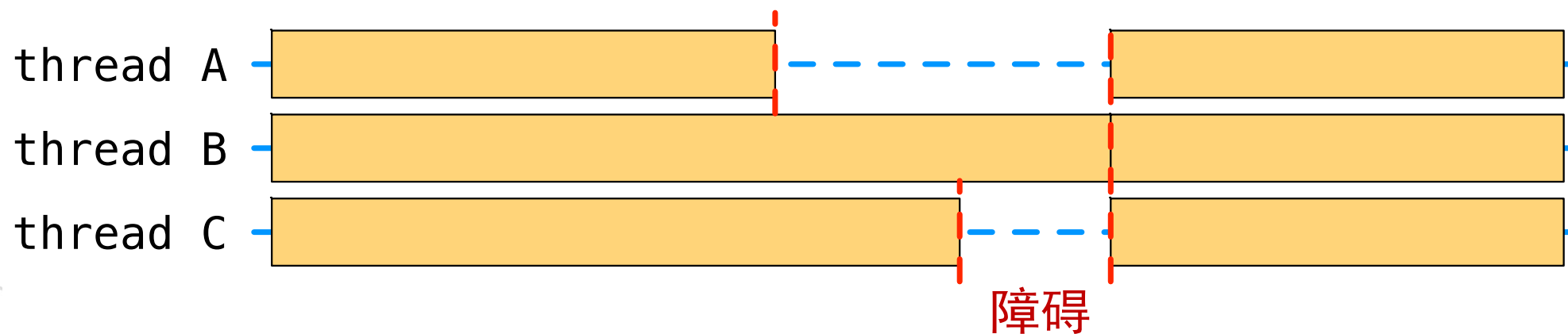
- 复合赋值运算符:  $+=$ ,  $*=$ ,  $-=$ ,  $/=$ ,  $\&=$ ,  $\^=$ ,  $|=$ ,  $<<=$ ,  $>>=$

– 临界区的作用范围更广, 能够实现的功能更复杂

## 障碍 (barrier)

- `#pragma omp barrier`
- 在障碍点处同步所有线程
  - 先运行至障碍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();  
}
```



## 障碍 (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



## 障碍 (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];
}
```

← 使用障碍同步线程

输出:

total: 1000

## 障碍 (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];
}
```

- `#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

- `#pragma omp master {}`
  - 在下面代码中与atomic结果相同

```
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];
}
```

- 什么是并行?
- 什么是OpenMP?
- 语法
- 同步机制
- 变量作用域
- 线程调度
- 生产者-消费者问题及其他

- 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

## Shared 与 private

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

int total = 0;      ← shared

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

## ◦ 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/firstprivate/lastprivate)
  - 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



## ◦ 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
```

## ◦ 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
```

## • 前述循环计数器值问题（见并行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

思考：

n=?

- 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);
}
```

## • 前述循环计数器值问题（见并行for循环）

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

```
thread 0 0 140732889458360
thread 1 1 123145476930232
thread 3 3 123145485343416
thread 2 2 123145481136824
```

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

```
thread 0 0 140732889458348
thread 3 3 123145485343404
thread 2 2 123145481136812
thread 1 1 123145476930220
```

## • 前述循环计数器值问题（见并行for循环）

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

thread 0 140732827743964

...

thread 7 123145346783964

thread 0 0 140732827743932

thread 3 3 123145329957564

thread 2 2 123145325750972

thread 1 1 123145321544380

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

140732889459336

thread 0 0 140732889458348

thread 3 3 123145485343404

thread 2 2 123145481136812

thread 1 1 123145476930220

## • 估算 $\pi$ 值

– 莱布尼茨级数  $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + (-1)^n \cdot \frac{1}{2n+1}$

- 求和问题

```
double factor = 1.0;
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+: sum) private(factor) ←
for (i=0; i<n; ++i){
    sum += factor/(2*i+1);
    factor =- factor;
}
pi = 4.0*sum;
```

`reduction(+: sum)` 保护对sum的更新不丢失  
`private(factor)` 保护factor不受其他线程影响

然而，循环间的依赖关系依然存在

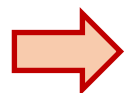
## • 估算 $\pi$ 值

– 莱布尼茨级数  $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \dots + (-1)^n \cdot \frac{1}{2n+1}$

- 求和问题

```
double factor = 1.0;
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+: sum) private(factor)
for (i=0; i<n; ++i){
    sum += factor/(2*i+1);
    factor =- factor;
}
pi = 4.0*sum;
```

factor 依赖于上次迭代



```
factor = (i % 2 == 0) ? 1.0 : -1.0;
sum += factor/(2*i+1);
```

每次迭代独立计算factor

## 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();
```

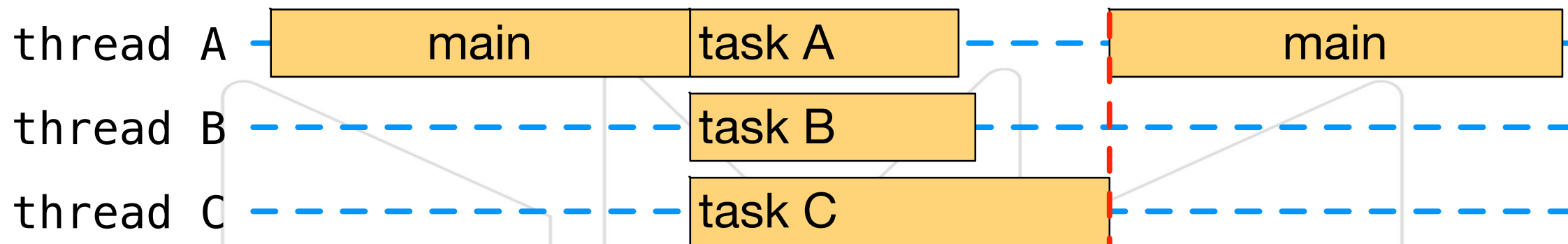
```
    #pragma omp section
```

```
    task_B();
```

```
    #pragma omp section
```

```
    task_C();
```

```
}
```

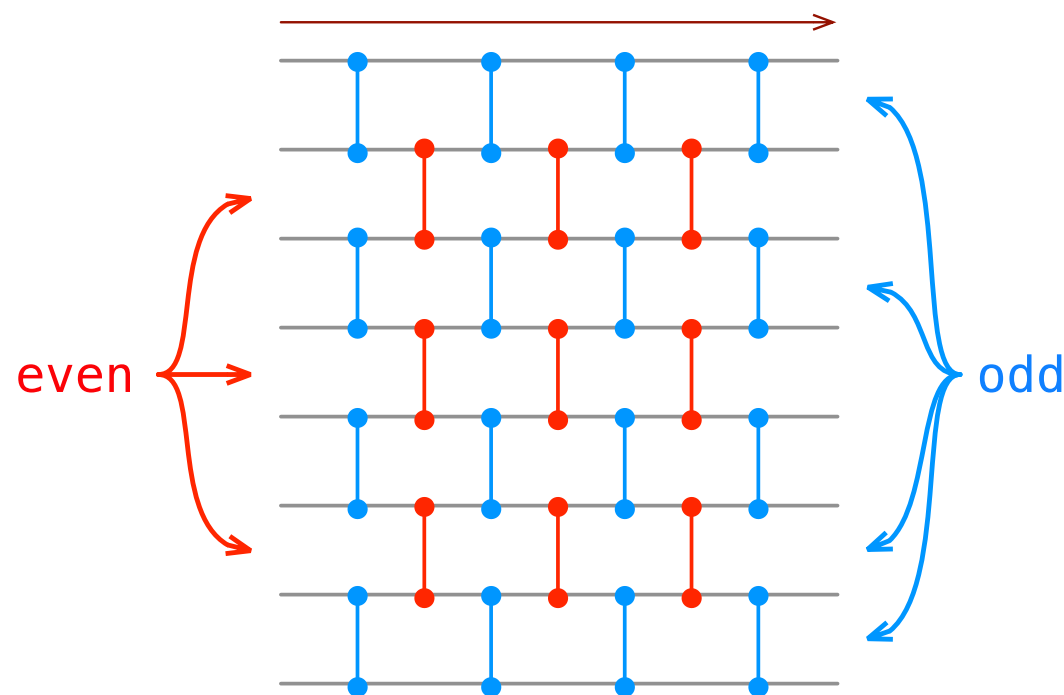


隐式障碍



## 奇偶移项排序

- 外层循环迭代间有依赖关系
- 内层循环可并行



```
void Odd_even_sort(int a[], int n) {  
    int phase, i, temp;  
  
    for (phase = 0; phase < n; phase++)  
        if (phase % 2 == 0) { /* Even phase */  
            for (i = 1; i < n; i += 2)  
                if (a[i-1] > a[i]) {  
                    temp = a[i];  
                    a[i] = a[i-1];  
                    a[i-1] = temp;  
                }  
        } else { /* Odd phase */  
            for (i = 1; i < n-1; i += 2)  
                if (a[i] > a[i+1]) {  
                    temp = a[i];  
                    a[i] = a[i+1];  
                    a[i+1] = temp;  
                }  
        }  
    } /* Odd_even_sort */  
}
```

## 奇偶移项排序

- 对奇偶phase分别构造并行区块
  - 两个parallel for
- 每次构造需重新创建线程

```
void Odd_even_sort(int a[], int n) {
    int phase, i, tmp;

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0)
            #pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    tmp = a[i-1];
                    a[i-1] = a[i];
                    a[i] = tmp;
                }
            else
                #pragma omp parallel for num_threads(thread_count) \
                default(none) shared(a, n) private(i, tmp)
                for (i = 1; i < n-1; i += 2)
                    if (a[i] > a[i+1]) {
                        tmp = a[i+1];
                        a[i+1] = a[i];
                        a[i] = tmp;
                    }
            }
    } /* Odd_even */
}
```

## 奇偶移项排序

### 一次并行区块构造

- 一个parallel指令
- 两个for指令

```
void Odd_even_sort(int a[], int n) {  
    int phase, i, tmp;  
  
    # pragma omp parallel num_threads(thread_count) \  
      default(none) shared(a, n) private(i, tmp, phase)  
    for (phase = 0; phase < n; phase++) {  
        if (phase % 2 == 0)  
        # pragma omp for  
        for (i = 1; i < n; i += 2)  
            if (a[i-1] > a[i]) {  
                tmp = a[i-1];  
                a[i-1] = a[i];  
                a[i] = tmp;  
            }  
        else  
        # pragma omp for  
        for (i = 1; i < n-1; i += 2)  
            if (a[i] > a[i+1]) {  
                tmp = a[i+1];  
                a[i+1] = a[i];  
                a[i] = tmp;  
            }  
    }  
} /* Odd_even */
```

并行区块内指明并行任务

## 奇偶移项排序：性能对比

线程数量	1	2	3	4
两个parallel for 指令	0.770	0.453	0.358	0.305
两个for指令	0.732	0.376	0.294	0.239

### 两个parallel for指令

```
void Odd_even_sort(int a[], int n) {
    int phase, i, tmp;

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0)
            # pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    tmp = a[i-1];
                    a[i-1] = a[i];
                    a[i] = tmp;
                }
            else
            # pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    tmp = a[i+1];
                    a[i+1] = a[i];
                    a[i] = tmp;
                }
    }
} /* Odd_even */
```

### 两个for指令

```
void Odd_even_sort(int a[], int n) {
    int phase, i, tmp;

    # pragma omp parallel num_threads(thread_count) \
    default(none) shared(a, n) private(i, tmp, phase)
    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0)
            # pragma omp for
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    tmp = a[i-1];
                    a[i-1] = a[i];
                    a[i] = tmp;
                }
            else
            # pragma omp for
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    tmp = a[i+1];
                    a[i+1] = a[i];
                    a[i] = tmp;
                }
    }
} /* Odd_even */
```

- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度
- 生产者-消费者问题及其他

- 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
```

## 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  $n/\text{threads}$  (快划分)

	(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调度的效率

- 如果工作量和迭代相关？

```
# pragma omp parallel for reduction(+:sum)
for ( i = 0; i <= n; ++i )
    sum += f(i);
```

```
double f(int i) {
    int j, start = i*(i+1)/2;
    int finish = start + i;
    double return_val=0.0;

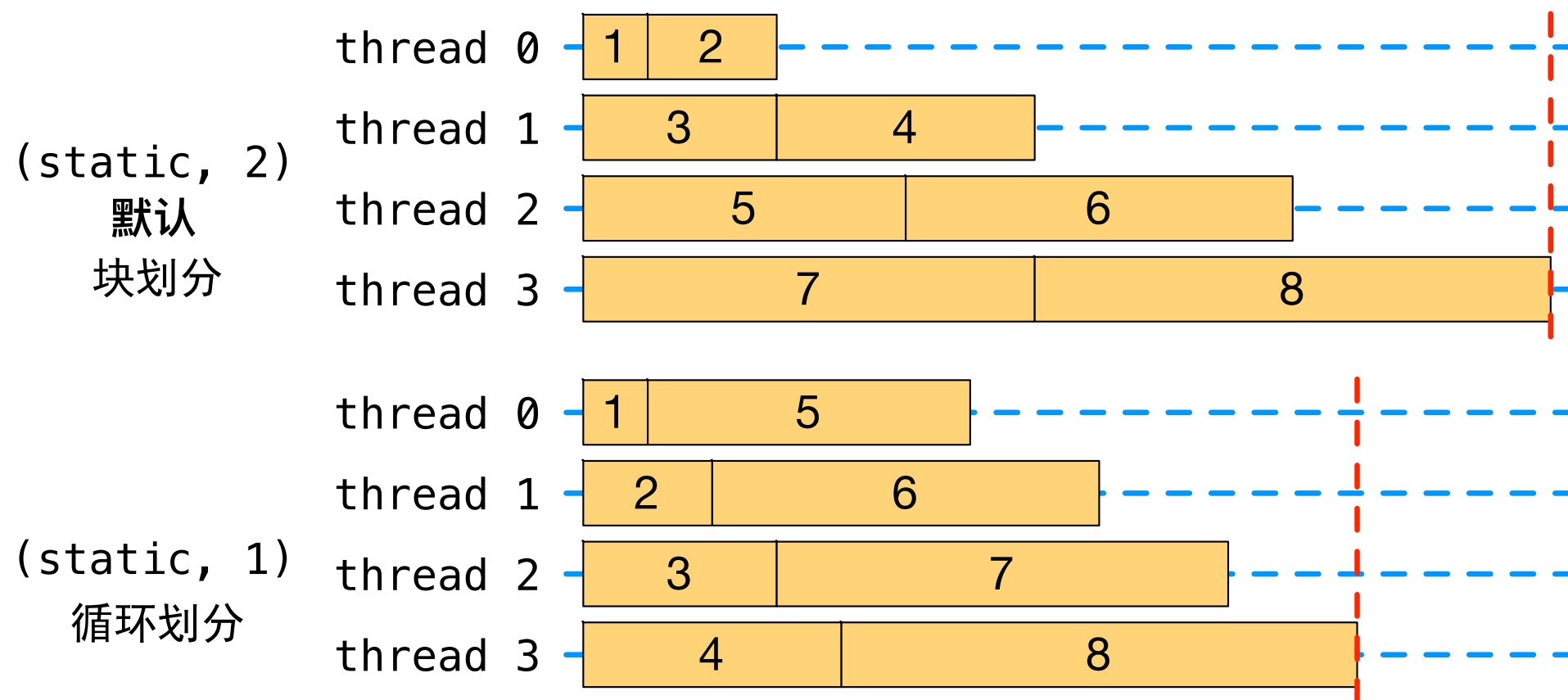
    for( j = start; j <= finish; j++){
        return_val += sin(j);
    }
    return return_val;
}
```



## Static调度的效率

- 默认的块划分：线程的负载可能不均匀
- 循环划分：划分更均匀

- `pragma omp parallel for schedule(static, 1)`



双线程执行时间&加速比

	时间	加速比
块划分	2.76	1.33
循环划分	1.84	1.99

单线程3.67秒  
n=10000

- 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

- auto（编译器决定策略）与 runtime（由环境变量指定策略）

- “Note that keywords auto and runtime aren’t adequate.”
  - [https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP\\_BoothTalk.pdf](https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdf)
- auto: “/\* For now map to schedule (static), later on ...\*/ -libgomp

## guided调度

– 但分配的chunk大小在运行中递减

- 初始块大小根据循环迭代综述、线程数、其他因素共同决定
- 最小不小于指定的chunk size

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999		0

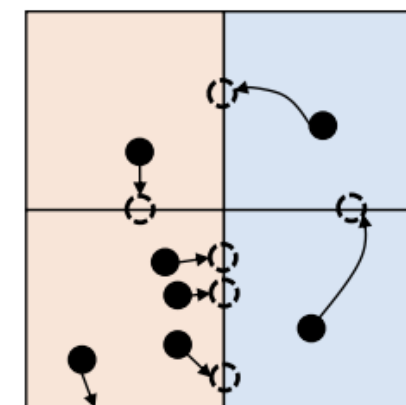
- 什么是并行？
- 什么是OpenMP？
- 语法
- 同步机制
- 变量作用域
- 线程调度
- 生产者-消费者问题及其他

## ◉ 队列：按照先进先出原则管理元素的数据结构

- 待服务的虚拟队列，先处理队首元素，新元素被加入到队尾
  - 可用于生产者-消费者问题，灵活处理各类生产-消费的计算模式
  - 对比此前Pthreads中生成及处理消息的机制（讲义4）
- 常用于任务调度、消息传递、广度优先搜索等

## ◉ 使用队列实现生产者-消费者结构的消息传递

- 生产者线程
  - 产生数据/任务（消息）
  - 发送至目标消费者线程的**共享队列**
- 消费者线程
  - 将队首元素出列（dequeue）并进行处理



blockwise particle  
trajectories

## ● 使用队列实现生产者-消费者结构的消息传递：初始化

### — 主线程

- 创建消息队列数组
- 数组中每个元素为一个队列，对应一个线程
- 每个线程都能访问数组中的任意队列，执行入队操作

### — 所有线程

- 初始化其对应的队列
- 需要使用障碍保证所有线程都已经完成其队列的初始化
  - 否则，先完成初始化并执行消息传递的线程可能对未初始化队列进行入队操作
  - `#pragma omp barrier`

- 使用队列实现生产者-消费者结构的消息传递：基本结构
  - 生产者线程：产生消息，发送至目标共享队列
  - 消费者线程：队首元素出列，处理消息

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++){  
    send_msg();  
    try_receive();  
}  
  
while(!done())  
    try_receive();
```



## ● 使用队列实现生产者-消费者结构的消息传递：基本结构

- 生产者线程：产生消息，发送至目标共享队列
- 消费者线程：队首元素出列，处理消息

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++){  
    send_msg();  
    try_receive();  
}
```

```
while(!done())  
    try_receive();
```

```
mesg = random();  
dest = random() % thread_count;  
#pragma omp critical  
enqueue(queue, dest, my_rank, mesg);
```

生成消息，入队

```
if (queue_size == 0) return;  
else if (queue_size == 1)  
#pragma omp critical  
    dequeue(queue, &src, &mesg);  
else  
    dequeue(queue, &src, &mesg);  
print_message(src, mesg);
```

出队，使用消息



## ● 使用队列实现生产者-消费者结构的消息传递：基本结构

- 生产者线程：产生消息，发送至目标共享队列
- 消费者线程：队首元素出列，处理消息

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++){  
    send_msg();  
    try_receive();  
}
```

```
while(!done())
```

```
    try_receive();
```

检查任务是否完成

```
queue_size = enqueued - dequeued;  
if (queue_size == 0 && done_sending == thread_count)  
    return true;  
else  
    return false;
```

所有线程共同更新  
done\_sending计数器

- 使用队列实现生产者-消费者结构的消息传递：同步
  - 队列数据：消息列表；队尾指针/索引；队首指针/索引； ...
    - 无论入队还是出队，都需要写数据
  - 如何保证对共享队列的访问安全？
  - 解决方案1：临界区
    - 1(a) 在所有入队/出队操作都加上`#pragma omp critical`
      - 不同队列的访问也将互斥
    - 1(b) 对**临界区命名**（不同名字的临界区代码可并行执行）
      - `#pragma omp critical(name)`
      - 然而，名字需要在编译过程中设置，无法动态指定
      - 只能支持固定数量的消息队列

## ● 使用队列实现生产者-消费者结构的消息传递：同步

– 如何保证对共享队列的访问安全？

– 解决方案2：锁（lock）

- 给每个队列配一把对应的锁

- 实现对每个队列访问的独立控制

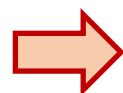
- void **omp\_init\_lock**(omp\_lock\_t\* lock\_p);

- void **omp\_set\_lock**(omp\_lock\_t\* lock\_p);

- void **omp\_unset\_lock**(omp\_lock\_t\* lock\_p);

- void **omp\_destroy\_lock**(omp\_lock\_t\* lock\_p);

```
#pragma omp critical  
/* q_p = msg_queues[dest] */  
enqueue(q_p, my_rank, mesg);
```



```
omp_set_lock(&q_p->lock);  
enqueue(q_p, my_rank, mesg);  
omp_unset_lock(&q_p->lock);
```

- 入队（enqueue）/出队（dequeue）前加锁，完成后解锁

## critical指令 vs atomic指令 vs 锁

- atomic只支持特定单一赋值语句，保证对同一变量的访问互斥
  - 因此，以下两段代码不互斥

```
#pragma omp atomic  
x++;
```

```
#pragma omp atomic  
y++;
```

- critical保证对相同名字的临界区代码访问互斥
- 锁保证对`omp_lock_t`保护的访问互斥
- 不同互斥机制之间不互斥，不应混用

```
#pragma omp atomic  
x += f(y)
```

```
#pragma omp critical  
x = g(x)
```

## ◉ 其他注意事项

- 互斥不保证公平性：线程可能一直阻塞
  - 以下代码中，特定线程可能一直无法执行

```
while (1) {  
    #pragma omp critical  
    x = g(my_rank);  
    ...  
}
```

- “嵌套” 互斥结构需要注意死锁

```
#pragma omp critical ← 增加 (one) 对临界区命名  
y= f(x);  
...
```

```
double f(double x){  
    #pragma omp critical ← 增加 (two) 对临界区命名  
    z = g(x);  
}
```

## 回顾：矩阵向量乘法

– 矩阵维度 ( $8M \times 8$ ,  $8K \times 8K$ ,  $8 \times 8M$ ) 对性能影响依然巨大

```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i, j) shared(A, x, y, m, n)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i*n+j]*x[j];
}
```

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

$x_0$
$x_1$
$\vdots$
$x_{n-1}$

 $=$ 

$y_0$
$y_1$
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
$y_{m-1}$

Threads	Matrix Dimension					
	8,000,000 $\times$ 8		8000 $\times$ 8000		8 $\times$ 8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275


- 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>



# 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 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](http://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.



## ● 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?

