分布式计算综合实践

线程级的并行：OpenMP编程

Chapter 2 – Thread Level Parallel Computing: OpenMP

线程级并行：基础概念

- 当一个处理器不足以满足计算需求时，除了增强单个核心的计算性能（但这很难），最直观的方法就是增加核心数量（线程级并行，TLP）
- 我们称这种拥有多个处理机的结构为多处理机，其特点是多个处理机共用一个共有的内存，也称为共享内存模型

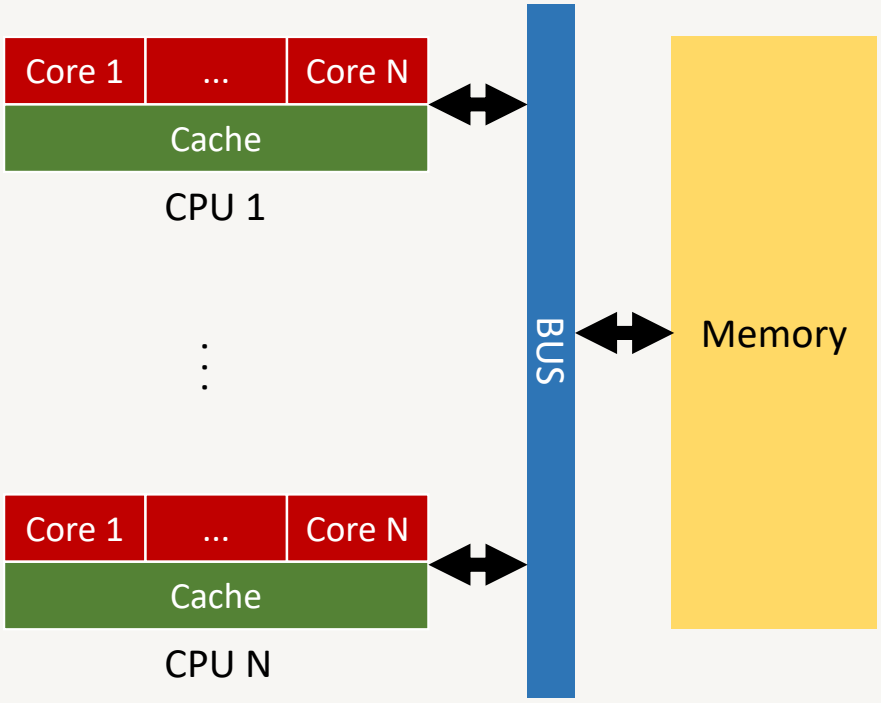
- 我们先从存储的角度尝试对共享内存模型的多处理机进行分类

术语：访存模型（共享内存）

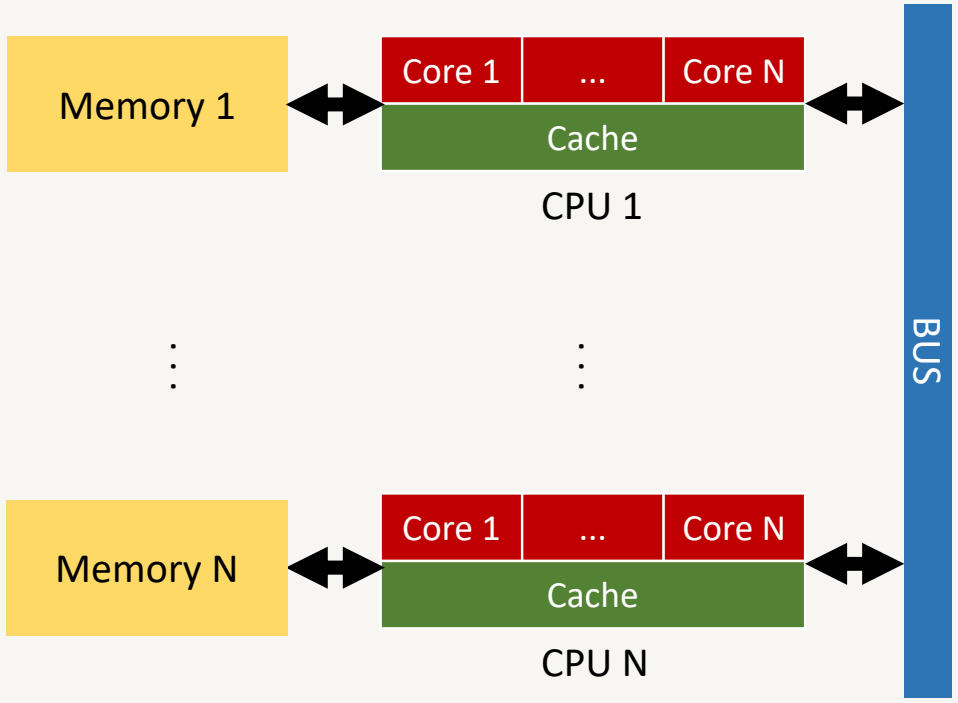
- UMA：Uniform Memory Access，一致访存，所有处理器对内存的访问是一致的，可以有私有cache
- NUMA：Non uniform Memory Access，非一致访存，处理器有各自的存储器

- 以上两种模型均具有统一的地址空间，也就是基于**共享内存模型**（本章只探讨这种情况）

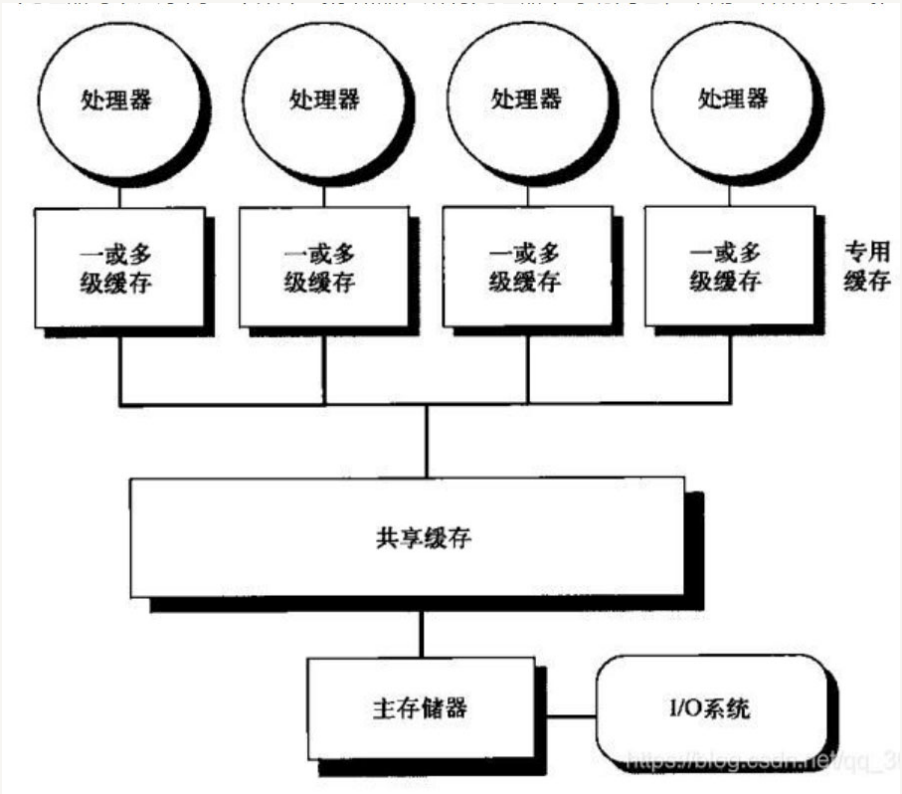
UMA


Core 1
...
Core N
Cache
CPU 1
Core 1
...
Core N
Cache
CPU N
BUS
Memory

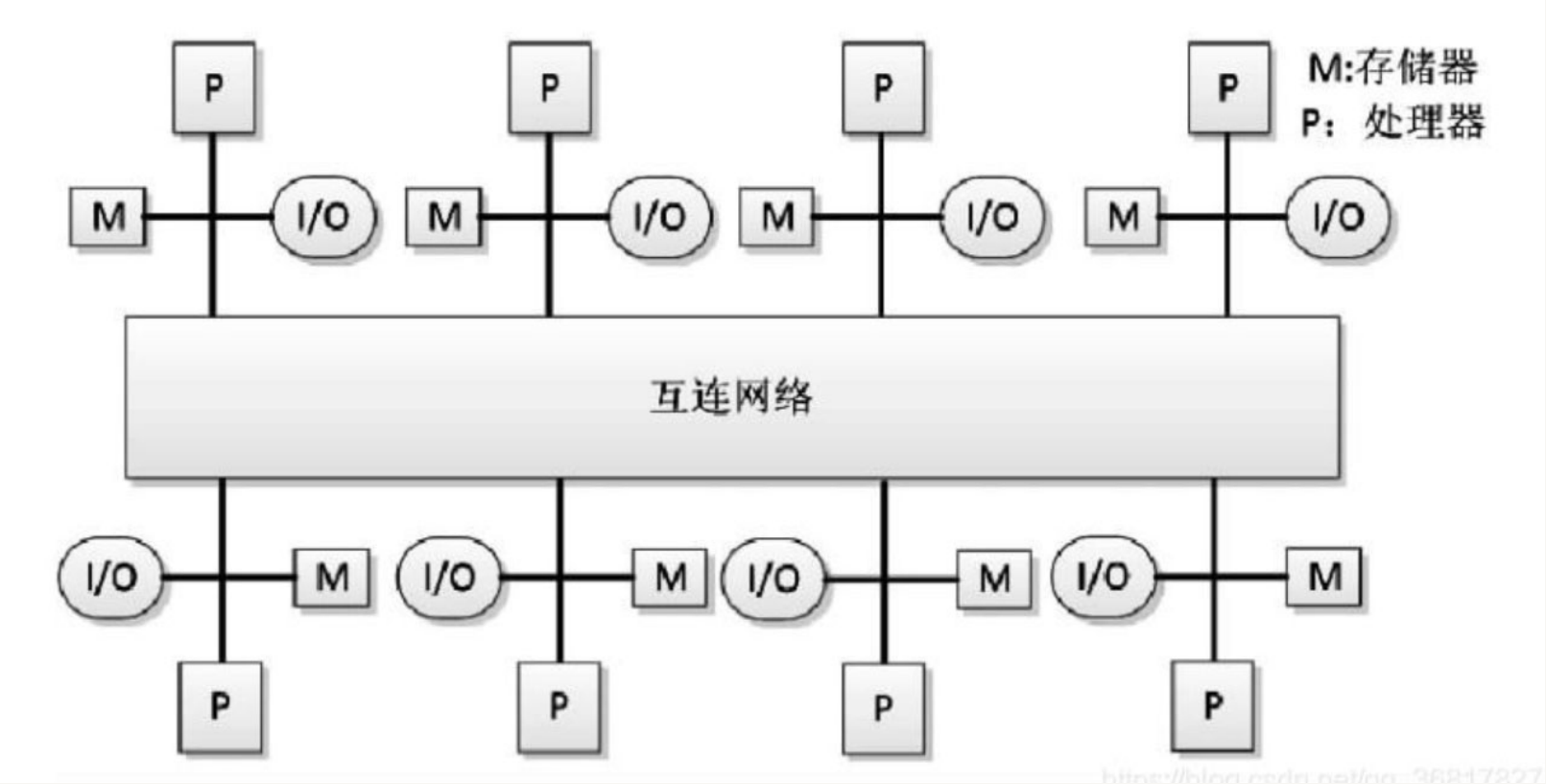
NUMA


Memory 1
Core 1
...
Core N
Cache
CPU 1
Memory N
Core 1
...
Core N
Cache
CPU N
BUS

- 典型的UMA：集中式共享存储器SMP



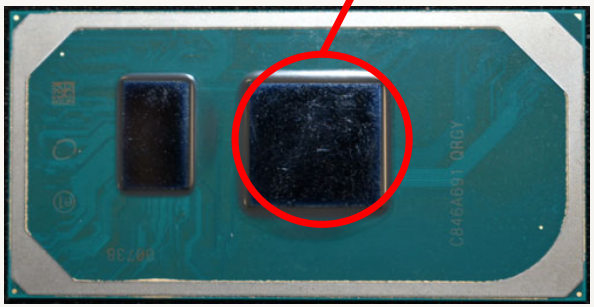
- 典型的NUMA：分布式共享存储器DSM



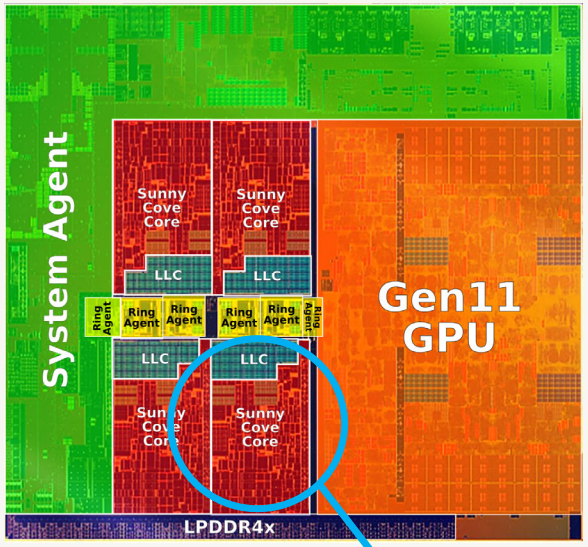
术语：关于CPU

- Core：一个计算核心
- Die：一颗晶片（可以有一或多个Core）
- Package：一个封装的处理器芯片（可以有一或多个Die）
- Socket：主板上的一个插槽，对应一个Package
- CPU / Processor：多重含义。在描述硬件时，一般指一个物理处理器也就是一个Package；在操作系统中，一般指一个逻辑处理器也就是一个Core

motherboard



package



die

core

举例

- 在1块消费级主板上往往只有1颗CPU（package），其中可能包含多个core，共用主存，因此属于UMA
- 在一些企业级主板上，可以同时安装多颗CPU，每颗CPU拥有自己的内存控制器，CPU之间使用外部总线连接；所有CPU共用一个内存地址空间，但其各自管理着属于自己的内存，因此属于NUMA

线程与进程

回顾在操作系统中的线程与进程概念：

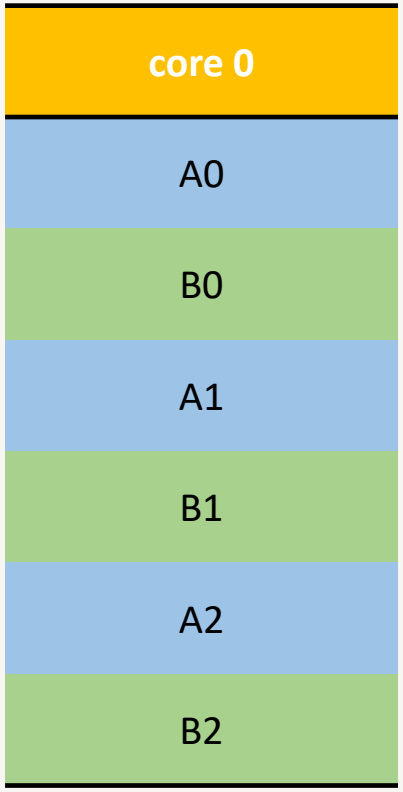
- 进程：一个正在执行程序的实例，包括程序计数器、寄存器和变量的当前值(更独立，有独自的地址空间)
- 线程：轻量级进程，共享地址空间，但各有一套堆栈

- 在操作系统中，线程与进程可以在单一处理器上(通过操作系统内核)实现，基本概念是对多道程序的快速切换(分配时间片)，是一种软件方式的**伪并行**，也被称为并发（concurrency）
- 显然，这并不会使得程序运行得更快

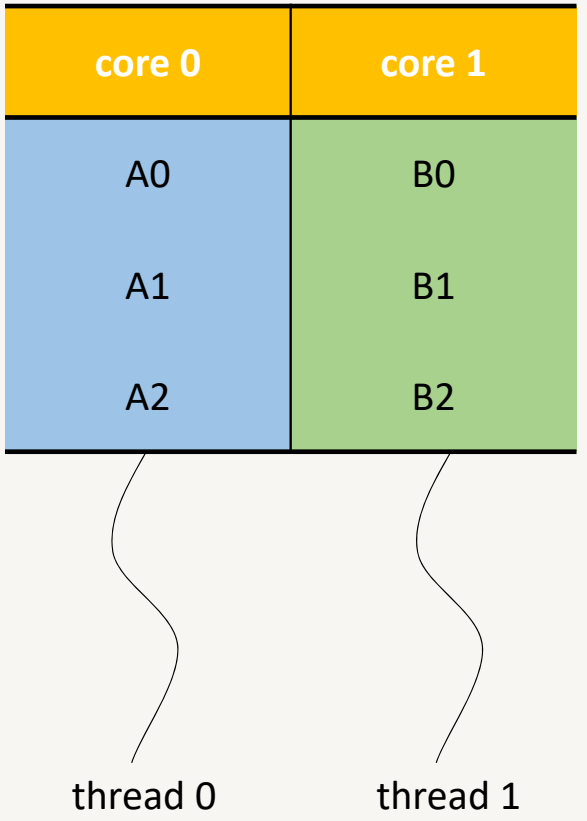
- 本章所述“线程级并行”指的更多是在多处理器上的**硬件级**线程并行，而非操作系统中实现的软件控制的并发

- 即，线程是在一个处理机上运行着的一段程序

- 并发 concurrency



- 并行 parallelism



并行计算编程模型

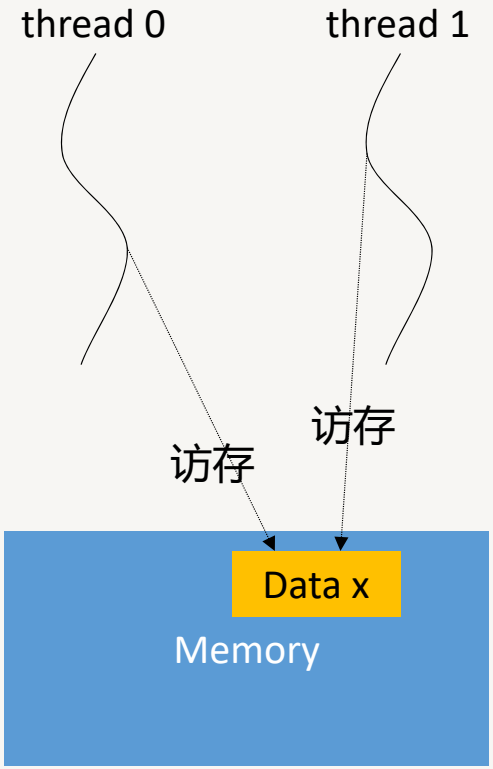
- 使用多线程协同进行计算需要专门的编程方式，在这一章及下一章我们探究在多处理器（SMP、DSM）和多机（MPP、COW）环境下的多线程（进程）编程模型
- 在这些场景下，作为计算资源的数据的共用是主要的矛盾点，而数据需要通过进程（线程）交互进行分享，因此我们先从进程交互的角度去进行分析

并行计算编程模型

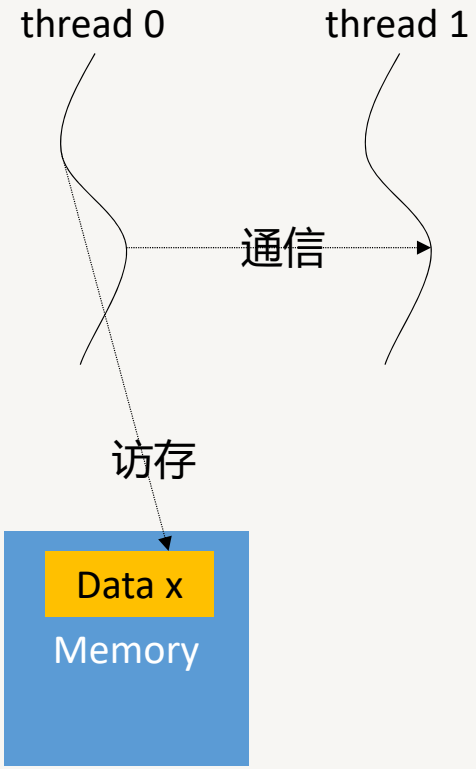
根据进程交互方式，我们有以下几类并行编程模型：

- 隐式交互（完全由编译器实现，这里不展开）
- 共享变量（英特尔Cilk、OpenMP）
- 消息传递（MPI）

- 共享变量


thread 0
thread 1
访存
访存
Data x
Memory

- 消息传递


thread 0
thread 1
通信
访存
Data x
Memory

并行计算编程模型

共享变量

- 适合于SMP、DSM
- 单一地址空间
- 隐式通信
- 在集群中，一般用于一个节点的多个核上

消息传递

- 适合于MPP、COW（不止）
- 多地址空间
- 显式通信
- 一般用于集群中的多个节点上

本章我们关注共享变量模型

共享变量编程存在的隐藏问题

- 在多处理机上使用多线程所面临的主要问题在于线程间如何协调，从而正确操作计算的对象——数据

- 由于数据（从硬件角度看就是内存）属于一种被计算所使用的资源，因此当多个核心共用这些资源（共享变量）时，自然就会因为对数据的竞争（谁先谁后）而产生一些奇怪的问题，归结起来就是：
- *多个线程访问一个变量时，会发生什么？*

多个线程访问一个变量 CASE1

考虑一个最简单的自加代码在多线程运行时的情况

```
// thread 0
// init: x = 0

x = x + 1;
```

```
// thread 1
// init: x = 0

x = x + 1;
```

```
// init: x = 0

x = x + 1;
```

将代码改写为汇编（x初始化为0，a1中为x的内存地址），以下两段代码有几种可能的执行顺序？

```
# thread 0

ld    t0, (a1)
addi  t0, t0, 1
sd    t0, (a1)
```

```
# thread 1

ld    t1, (a1)
addi  t1, t1, 1
sd    t1, (a1)
```

```
// init: x = 0

x = x + 1;
```

我们期待的是，执行了两次x的自加，x变为2

thread 0	thread 1	备注
ld t0, (a1)		t0 = x = 0
addi t0, t0, 1		t0 = 1
sd t0, (a1)		x = t0 = 1
	ld t1, (a1)	t1 = x = 1
	addi t1, t1, 1	t1 = 2
	sd t1, (a1)	x = t1 = 2

```
// init: x = 0

x = x + 1;
```

但也可能出现这种情况

thread 0	thread 1	备注
`ld t0, (a1)`		t0 = x = 0
`addi t0, t0, 1`		t0 = 1
	`ld t1, (a1)`	t1 = x = 0
	`addi t1, t1, 1`	t1 = 1
`sd t0, (a1)`		x = t0 = 1
	`sd t1, (a1)`	x = t1 = 1

- 上述问题被称为：竞态（race condition）
- 当程序的正确运行依赖于程序中各线程的特定时序时（执行顺序不同会产生不同的结果），就会出现竞态
- 这种依赖往往发生在多个线程对同一个资源的竞争中，尤其当其中存在修改资源状态的操作（写操作）时。在内存上，这被称为数据竞争（data race）

- 数据竞争的解决方法：利用同步，确保操作是原子性的，从而对操作进行排序，将资源与操作“保护”起来
- 工具：锁、信号量等
- 注意：“排序”并不意味着顺序是确定的，因此解决了数据竞争并不意味着完全解决了竞态

多个线程访问一个变量 CASE2

再来看一个例子：
看看以下两个线程，你觉得最终z应当是几

```
// thread 0
// init: ready=0, z=0

z = 3;
ready = 1;
```

```
// thread 1
// init: ready=0, z=0

while (!ready);
z++;
```

```
// thread 0                          // thread 1
// init: ready=0, z=0                // init: ready=0, z=0

z = 3;          // (a)               while (!ready);     // (c)
ready = 1;      // (b)               z++;                // (d)
```

- 设想中，由于$b \rightarrow c \implies a \rightarrow d$，z最终结果为4
- 但别忘了，现在的超标量处理器都支持**指令重排序**
- 同时，在多个处理器之间，由于存在cache以及互联网络，你无法保证一个线程的写操作**何时**可以被其他线程看见，因此也可能发生**内存上的重排序**

指令重排序

- 由于对z的数据依赖发生在不同的线程中，处理器无法检测，因此能主动进行重排序

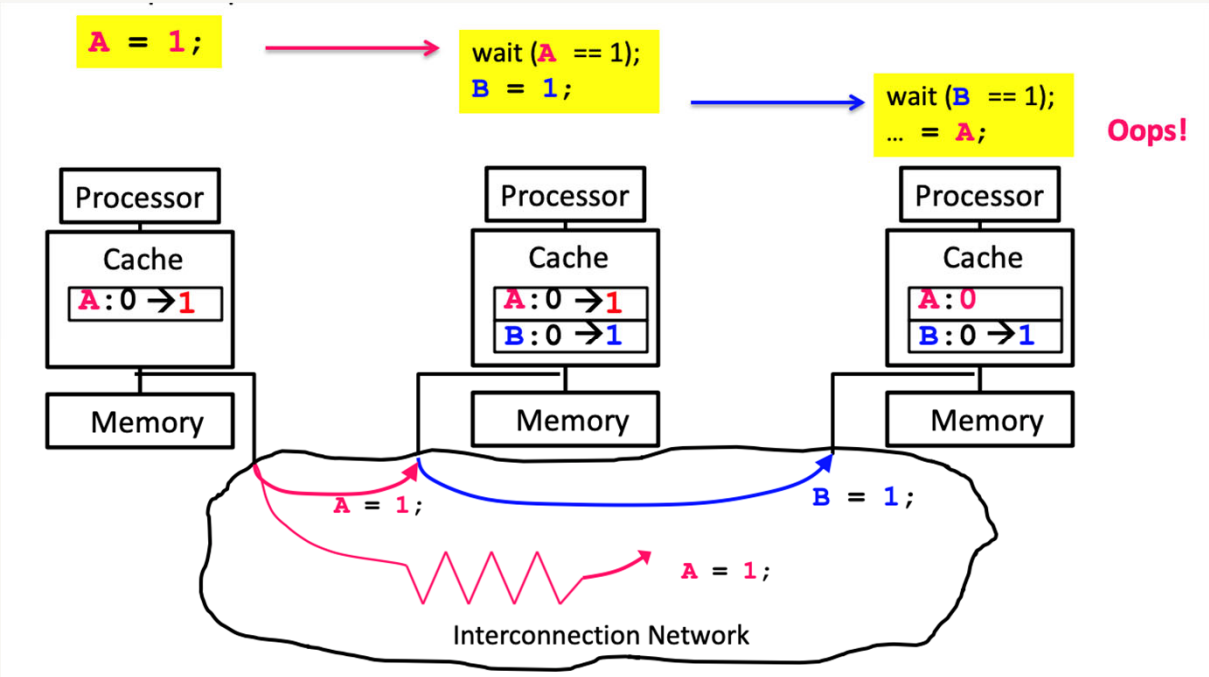
```
// thread 0
// init: ready=0, z=0
//假设z脱靶，ready命中
z = 3;      // cache miss,
            // can't issue
ready = 1; // issue first
```

```
// thread 1
// init: ready=0, z=0

while (!ready);
z++;
```

z=0 -> z=1 -> z=3

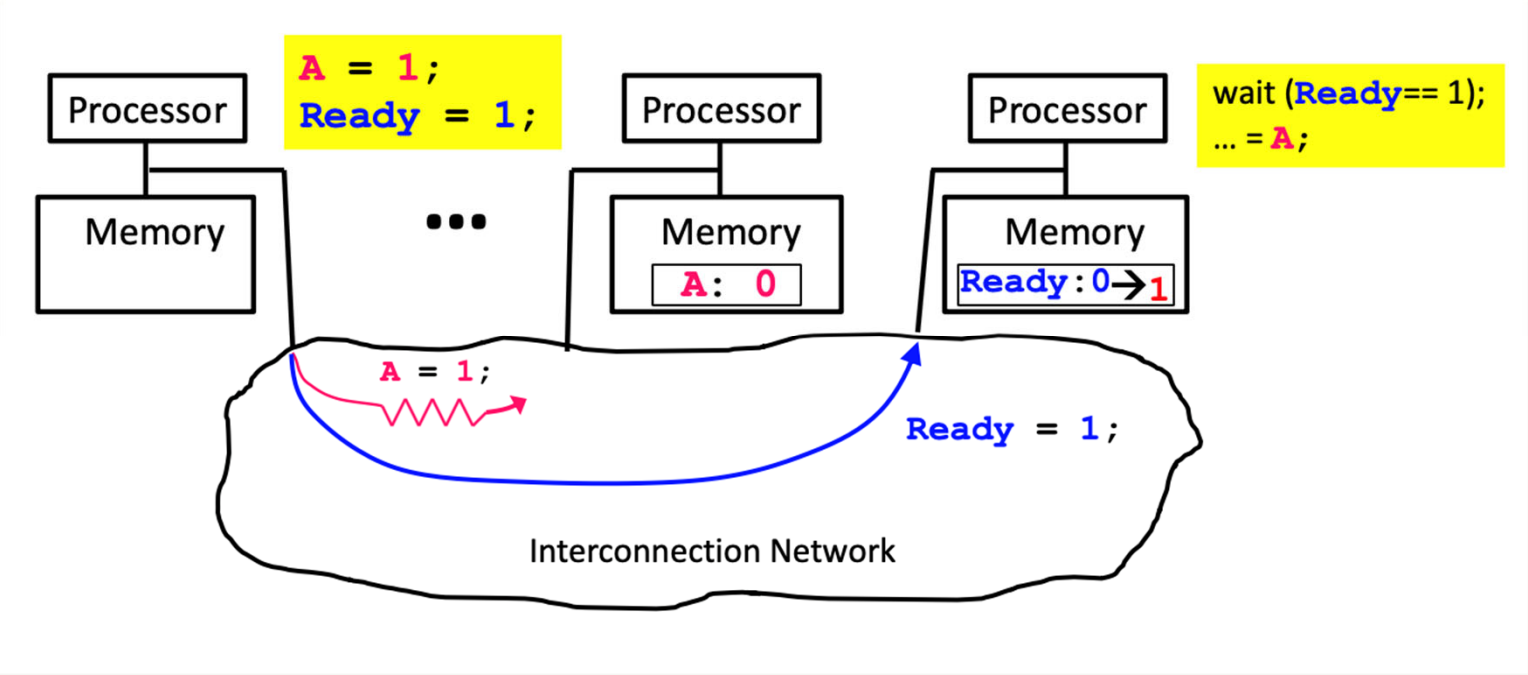
内存重排序

- 缓存导致的问题



内存重排序

- 互联网络导致的问题



- 可以看到，一个线程内的几条指令之间（在对于另一个线程的可见性上）出现了难以预见的重排序情况
- 这种重排序对单线程没有影响，但对于多线程则产生了问题

- 解决方法也是通过同步（锁、栅栏等技术）

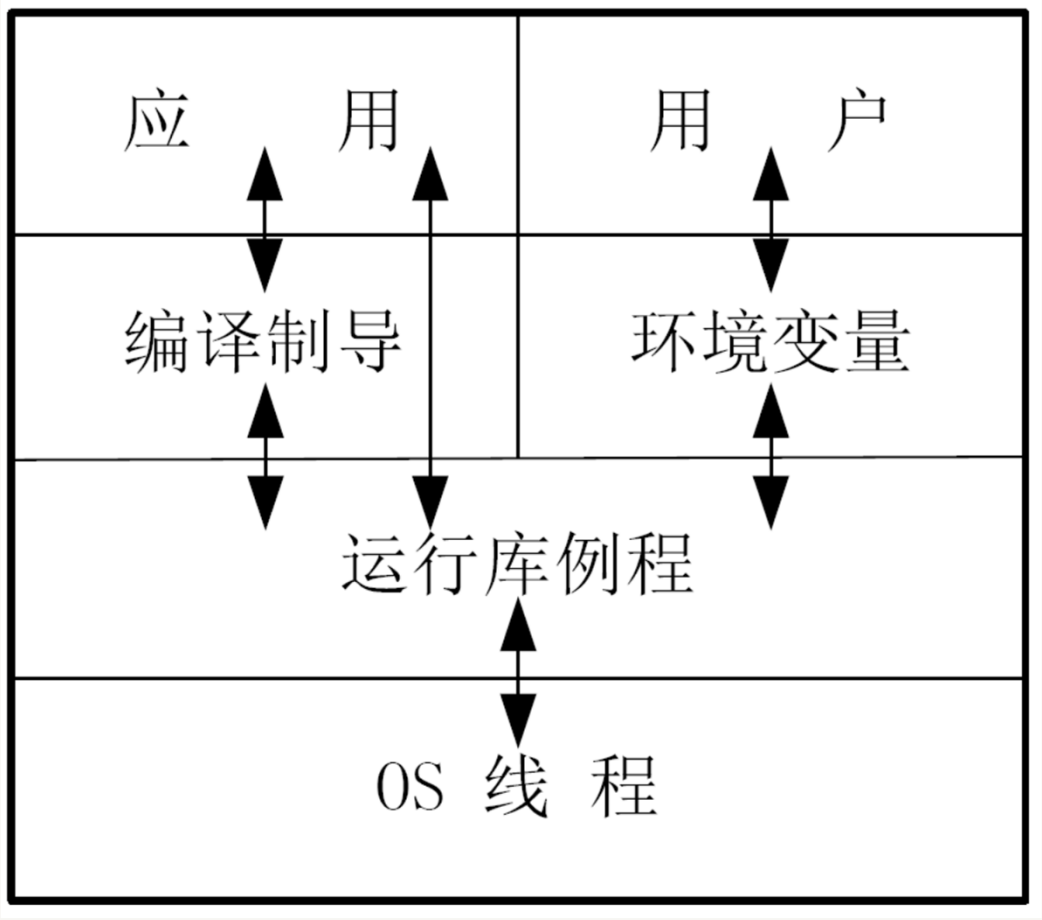
综上所述，在编写多线程代码时，请时刻注意以上问题，
并合理利用同步技术

线程级并行
编程模型：OpenMP

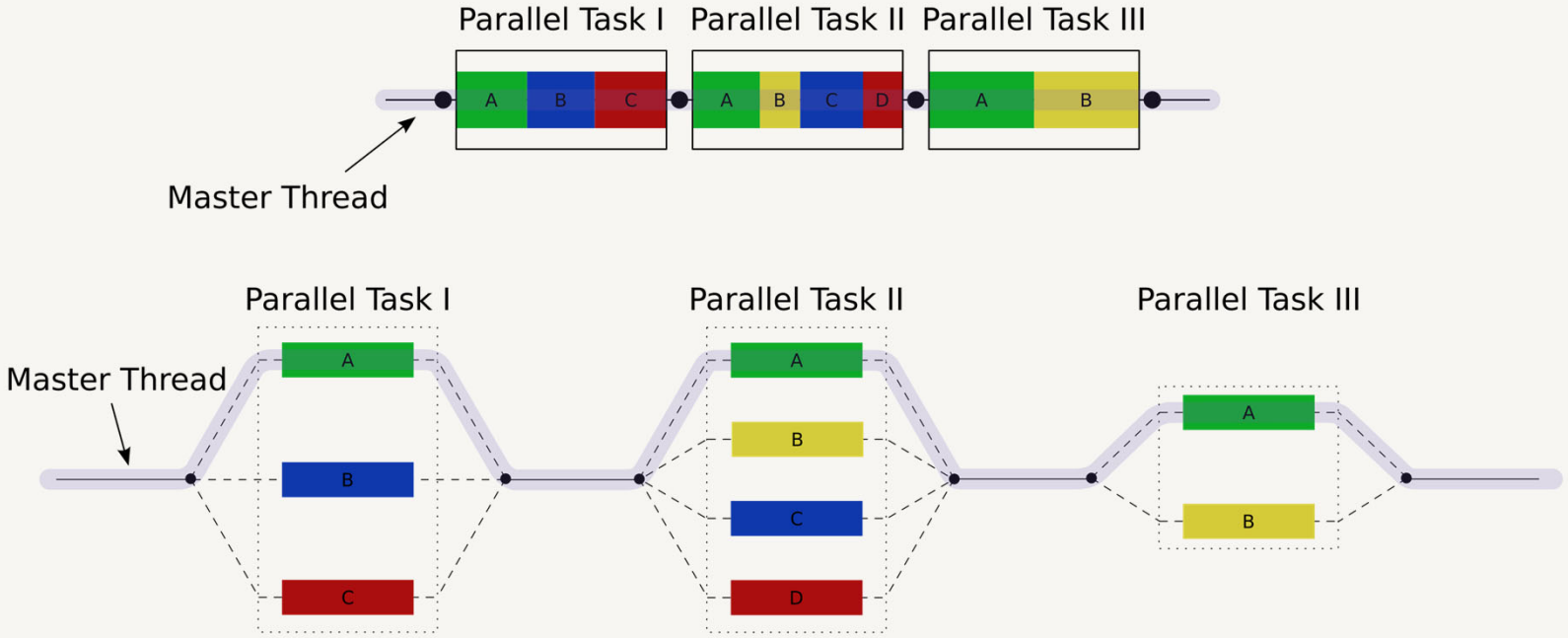
OPENMP概述

- OpenMP API（Open Multi-Processing Application Programming Interface）开放多处理应用编程接口是在共享存储体系结构上的一个编程模型，支持Unix/Linux/Win等多种平台。具有简单、可移植性好、可扩展等优点，是共享存储系统编程的工业标准。
- OpenMP是一个工业标准规范的实现，包含三类基本API，编译制导(Compiler Directive)、运行库例程(Runtime Library)和环境变量(Environment Variables)机制，用户使用这些机制与编译器和运行时系统进行交互，使得用户能够控制并行程序的行为，完成需要的操作。
- OpenMP通过支持用户对程序添加制导，使得用户可以将串行程序增量并行化(Incremental Parallelization)。

OPENMP体系结构



OPENMP并行编程模型：OPENMP是基于线程的并行编程模型，一个共享的进程由多个线程组成。使用FORK-JOIN并行模型，主线程（MASTER THREAD）串行执行，直到编译制导并行域（PARALLEL REGION）出现。

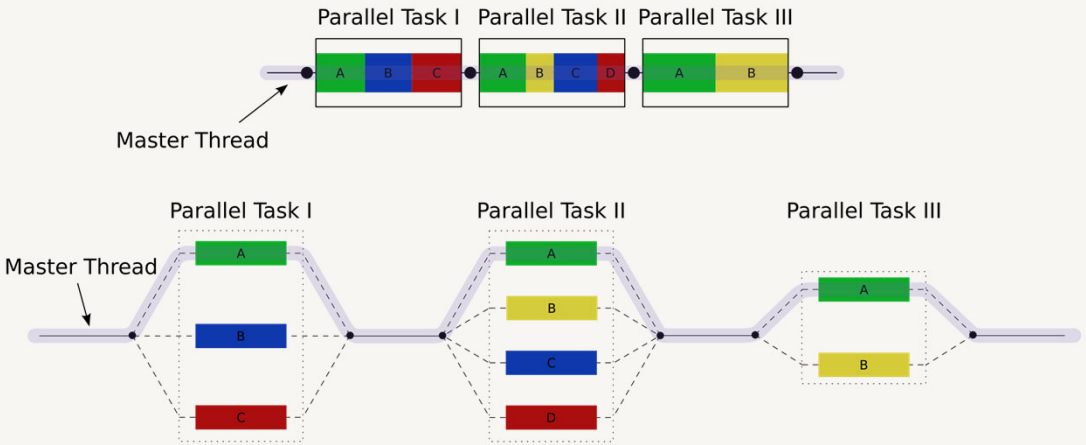


FORK-JOIN 模型

- 将程序划分为许多段的**串行部分**或**并行部分**
- **串行部分**使用单个线程处理
- **并行部分**的各个子任务使用多个线程分而治之
- 并行任务拆分出的子任务可以继续拆分
- 使用fork进入并行(子任务)部分，使用join回到串行(父任务)部分

FORK-JOIN 模型

- 一般情况下，串行部分（父任务）所在线程被视为主线程，该线程在fork时将会被分配给其中一个子任务，而当join时，所有其他线程会被移除，该主线程继续分配给下一段串行部分使用

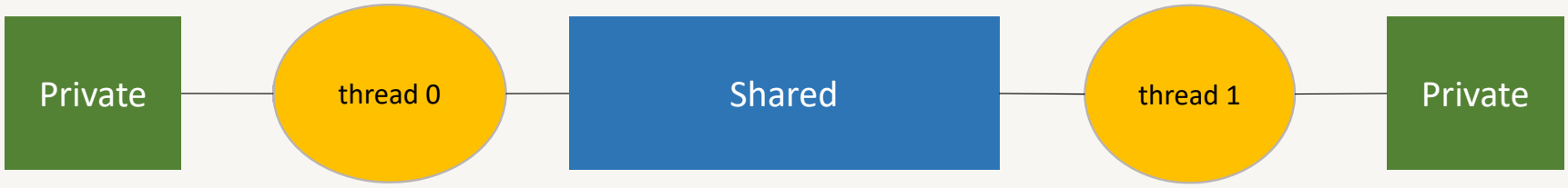


OpenMP

- 一种基于fork-join模型的多线程并行编程API
- 在C、C++、Fortran等语言上提供接口
- 主要适用于共享内存结构的多处理机

OpenMP 存储模型

- OpenMP中将存储分为shared和private两类
- shared变量将在各个线程之间共享（因此在对其进行操作时，请注意竞态和重排序问题，并合理使用同步）
- private变量是各线程独有的，互不影响



```
/* 用OpenMP/C编写Hello World代码段 */
#include <omp.h>
int main(int argc, char *argv[])
{
        int nthreads,tid;
        omp_set_num_threads(4);
        /* Fork a team of threads */
        #pragma omp parallel private(tid)
        {
                tid = omp_get_thread_num(); /* Obtain and print thread id */
                printf("Hello, world from OpenMP thread %d\n", tid);
                if (tid == 0) /*Only master thread does this */
                {
                        nthreads = omp_get_num_threads();
                }
        } //join
        printf(" Number of threads %d\n",nthreads);
        return 0;
}
```

- 编译
  ```
  gcc  -fopenmp  -o hello.o hello.c
  ```
- 运行
  ```
  ./hello.o
  ```
- 结果
  ```
  Hello, world from OpenMP thread 0
  Hello, world from OpenMP thread 3
  Hello, world from OpenMP thread 1
  Hello, world from OpenMP thread 2
   Number of threads 4
  ```

OPENMP的语法主要包含三个部分

- 环境变量
- 运行时库函数
- 编译制导
  - 并行域结构
  - 任务划分结构
  - 同步结构

环境变量(通过对环境变量值的修改，可以对OPENMP进行控制)

- OMP_SCHEDULE
  - 只能用于parallel for 和 for，它的值决定了循环中各个迭代的调度方式
- OMP_NUM_THREADS
  - 执行中所能使用的最大线程数量
- OMP_DYNAMIC
  - 其值为布尔型true或false，确定是否能够动态设定并行域执行部分的线程数
- OMP_NETSTED
  - 其值为布尔型true或false，确定是否允许嵌套并行

```
/* 用OpenMP/C编写Hello World代码段 */
#include <omp.h>
int main(int argc, char *argv[])
{
        int nthreads,tid;
        char buf[32];
        omp_set_num_threads(4);
        /* Fork a team of threads */
        #pragma omp parallel private(tid)
        {
                tid = omp_get_thread_num(); /* Obtain and print thread id */
                printf("Hello, world from OpenMP thread %d\n", tid);
                if (tid == 0) /*Only master thread does this */
                {
                        nthreads = omp_get_num_threads();
                }
        }
        printf(" Number of threads %d\n",nthreads);
        return 0;
}
```

运行时
库函数

编译制导

OPENMP运行时库函数

- OpenMP标准定义了一个应用编程接口来调用库中的多种函数
- 对于C/C++，在程序开头需要引用文件“omp.h”
- 例如：
- omp_set_num_threads(nthread);
- INTEGER omp_get_num_threads();
- INTEGER omp_get_max_threads();
- INTEGER omp_get_thread_num();
- INTEGER omp_get_num_procs();

运行时库函数

- omp_get_num_procs，返回运行本线程的多处理机的处理器个数。
- omp_get_num_threads，返回当前并行区域中的活动线程个数。
- omp_get_thread_num，返回线程号
- omp_set_num_threads，设置并行执行代码的线程个数
- omp_init_lock，初始化一个简单锁
- omp_set_lock，上锁操作
- omp_unset_lock，解锁操作，要和omp_set_lock函数配对使用。
- omp_destroy_lock，omp_init_lock函数的配对操作函数，关闭一个锁

OPENMP 锁

- 在OpenMP中提供一系列库函数用来操作互斥锁：

```
void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_detroy_lock(omp_lock_t *lock);
```

```
#include <stdio.h>
#include <omp.h>
static omp_lock_t lock;
int main()
{
    int i;
    omp_init_lock(&lock);

    #pragma omp parallel for
    for (i = 0; i < 5; ++i)
    {
        omp_set_lock(&lock);
        printf("%d +\n", omp_get_thread_num());
        printf("%d -\n", omp_get_thread_num());
        omp_unset_lock(&lock);
    }

    omp_destroy_lock(&lock);

    return 0;
}
```

Output:

```
1 +
1 -
2 +
2 -
0 +
0 -
4 +
4 -
3 +
3 -
```

OPENMP的语法主要包含三个部分

- 环境变量
- 运行时库函数
- **编译制导**
  - 并行域结构
  - 任务划分结构
  - 同步结构

编译制导

- 编译制导是对程序设计语言的扩展，OpenMP通过对串行程序添加制导语句实现并行化
- 编译制导的语法为

  #pragma omp 指令 子句
- 例如：

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

  C制导前缀　　指令　　子句

编译制导语句具体格式

- #pragma omp parallel [clause,clause,…]newline

语句格式	解释
#pragma omp	制导指令前缀。对所有的OpenMP语句都需要这样的前缀。
directive-name	OpenMP制导指令。在制导指令前缀和子句之间必须有一个正确的OpenMP制导指令。
[clause, ...]	子句。相当于制导指令的修饰参数。在没有其它约束条件下子句可以无序，也可以任意的选择。这一部分也可以没有。
newline	换行符。表明这条制导语句的终止。

编译制导语句作用域

动态范围：包括静态范围和孤立语句	
静态范围（又称词法范围Lexical Extent）：文本代码在一个编译制导语句之后，被封装到一个结构块中	孤立语句：一个OpenMP的编译制导语句不依赖于其它的语句，存在于其他的静态范围语句之外。
例子：for出现在封闭的并行域中间	例子：critical和sections语句出现在封闭的并行域之外

#pragma omp parallel
{
…
#pragma omp for
for(…){
…
sub1();
…
}
…
Sub2();
…
}
	void sub1()
	{
	…
	#pragma omp critical
	…
	}
	
	void sub2()
	{
	…
	#pragma omp sections
	…
	}

制导指令（只列出一部分）

并行结构

- parallel 用在一个代码段之前，表示这段代码将被多个线程并行执行

任务划分结构

- for 用于for循环之前，将循环分配到多个线程中并行执行，必须保证每次循环之间无相关性。
- sections 用在可能会被并行执行的代码段之前
- single 用在一段只被单个线程执行的代码段之前，表示后面的代码段将被单线程执行

同步结构

- critical 用在一段代码临界区之前
- barrier，用于并行区内代码的线程同步，所有线程执行到barrier时要停止直到所有线程都执行到barrier时才继续往下执行。
- Atomic 用于指定一块内存区域被制动更新
- Master 用于指定一段代码块由主线程执行
- Ordered 用于指定并行区域的循环按顺序执行
- threadprivate 用于指定一个变量是线程私有

Atomic 用于指定一块内存区域被制动更新

```
3  #include <stdio.h>
4  #include <omp.h>
5
6  int main()
7  {
8    int data = 1;
9  #pragma omp parallel num_threads(4) shared(data) default(none)
10   {
11 #pragma omp atomic
12     data += data * 2;
13   }
14   printf("data = %d\n", data);
15   return 0;
16 }
```

程序最终结果如下：

```
1  data = 81
```

子句（只列出一部分）

- private，指定每个线程都有它自己的变量私有副本。表示它列出的变量对于每个线程是局部的
- firstprivate，指定每个线程都有它自己的变量私有副本，并且变量要被继承主线程中的初值。申明的变量为线程私有，它的初始值由对应的共享变量的值给出。
- lastprivate，主要是用来指定将线程中的私有变量的值在并行处理结束后复制回主线程中的对应变量。申明的变量为线程私有，在对应的并行任务结束的时候把自己的私有变量的值赋给对应的共享变量。
- reduction，用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的运算。使用指定的操作对其列表中出现的变量进行归约。初始时，每个线程都保留一份私有拷贝，在结构尾部根据指定的操作对线程中的相应变量进行规约，并更新该变量的全局值。
- nowait，忽略指定中暗含的等待
- num_threads，指定线程的个数
- schedule，指定如何调度for循环迭代
- shared，指定一个或多个变量为多个线程间的共享变量，表示它所列出的变量被线程组中所有的线程共享
- ordered，用来指定for循环的执行要按顺序执行
- copyprivate，用于single制导中的指定变量广播到并行区中其它线程
- copyin，用来指定一个threadprivate的变量的值要用主线程的值进行初始化。为线程组中所有线程的threadprivate变量赋相同的值，主线程该变量的值作为初始值。
- default，用来指定并行处理区域内的变量的使用方式，缺省是shared，让用户自行规定在一个并行域的静态范围中所定义的变量的缺省作用范围。

OPENMP的语法主要包含三个部分

- 环境变量
- 运行时库函数
- **编译制导**
  - 1. 并行域结构
  - 2. 任务划分结构
  - 3. 同步结构

1.并行域结构

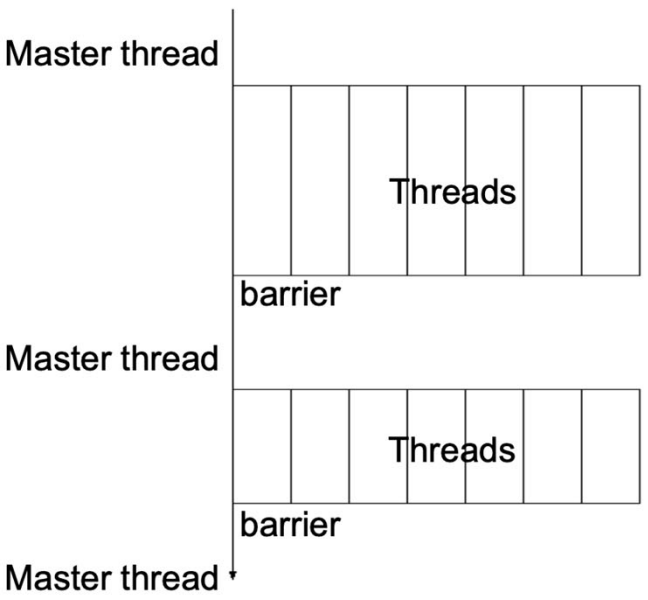
- 一个并行域就是一个能被多个线程并行执行的程序块，它是最基本的OpenMP并行结构。
- 具体格式:　#pragma omp parallel [clause,clause,…]newline
- 其中clause=
- 　if (scalar_expression)
- 　private (list)
- 　shared (list)
- 　default (shared | none)
- 　firstprivate (list)
- 　reduction (operator: list)
- 　copyin (list)

并行域结构PARALLELISM CONSTRUCTS

```
• #pragma omp parallel [clauses]
{
      BLOCK
}
```

- 在并行域结尾有一个隐式同步(barrier)。
- 子句(clause)用来说明并行域的附加信息。
- C/C++子句间用空格分开

并行域结构图




```
      PROGRAM FRED
        .
        .
!$OMP PARALLEL
        .
        .
        .
        .
        .
        .
        .
        .
!$OMP END PARALLEL
        .
        .
        .
        .
!$OMP PARALLEL
        .
        .
        .
        .
!$OMP END PARALLEL
        .
        .
        .
```

并性域结构：IF子句

```
#pragma omp parallel if (n>thr) shared(n) private(i,tid)
{
        #pragma omp for
        for (i=0; i<n; ++i)
        {
                tid = omp_get_thread_num();
                printf("%d %d\n", tid, i);
        }
}
```

并性域结构：SHARED和PRIVATED子句

- 通过shared和privated子句可以指定变量在并行域的各个线程间是公有还是私有
- 通常循环变量、临时变量、写变量一般是私有的;
- 数组变量、仅用于读的变量通常是共享的。

```
shared(list)

private(list)

default(shared|private|none)
```

并性域结构：FIRSTPRIVATE/LASTPRIVATE

- 使用private子句，其指定的私有变量在进入和退出并行域时均为“未定义”的（值带不进来也带不出去）
- firstprivate子句用于在进入并行域之前进行一次初始化，将域外的值继承到并行域内
- lastprivate子句用于在退出并行域时将域内变量值（语法上的最后一个私有变量的值）赋给域外同名变量
  - 语法上的最后一个指的是循环的最后一次迭代，或section的最后一个
- firstprivate和lastprivate可以一起使用

并性域结构：REDUCTION子句

- 通过某种操作（+,*,max或min等），将某个变量在各线程中的值规约为一个值

```
sum=0;
#pragma omp parallel shared(n) private(i) reduction(+:sum)
{
        #pragma omp for // 将循环展开到多个线程
        for (i=0; i<n; ++i)
        {
                sum = sum+i;
        }
}
printf("%d\n", sum);

// reduction 为每个线程创建了sum变量的私有副本,
// 并在并行域结束时把所有sum值加在一起存回原始的sum
```

并性域结构：REDUCTION子句

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

static int data;

int main() {
  #pragma omp parallel num_threads(2) // 使用两个线程同时执行上面的代码块
  {
    for(int i = 0; i < 10000; i++) {
      data++;
      usleep(10);
    }
    // omp_get_thread_num 函数返回线程的 id 号 这个数据从 0 开始, 0, 1, 2, 3, 4, ...
    printf("data = %d tid = %d\n", data, omp_get_thread_num());
  }

  printf("In main function data = %d\n", data);
  return 0;
}
```

并性域结构：REDUCTION子句
（使用数组解决并发程序中的数据竞争问题）

```
1
2  #include <stdio.h>
3  #include <omp.h>
4  #include <unistd.h>
5
6  static int data;
7
8  static int tarr[2];
9
10 int main() {
11   #pragma omp parallel num_threads(2)
12   {
13     int tid = omp_get_thread_num();
14     for(int i = 0; i < 10000; i++) {
15       tarr[tid]++;
16       usleep(10);
17     }
18     printf("tarr[%d] = %d tid = %d\n", tid, tarr[tid], tid);
19   }
20   data = tarr[0] + tarr[1];
21   printf("In main function data = %d\n", data);
22   return 0;
23 }
```

```
1 $./lockfree01.out
2 tarr[1] = 10000 tid = 1
3 tarr[0] = 10000 tid = 0
4 In main function data = 20000
```

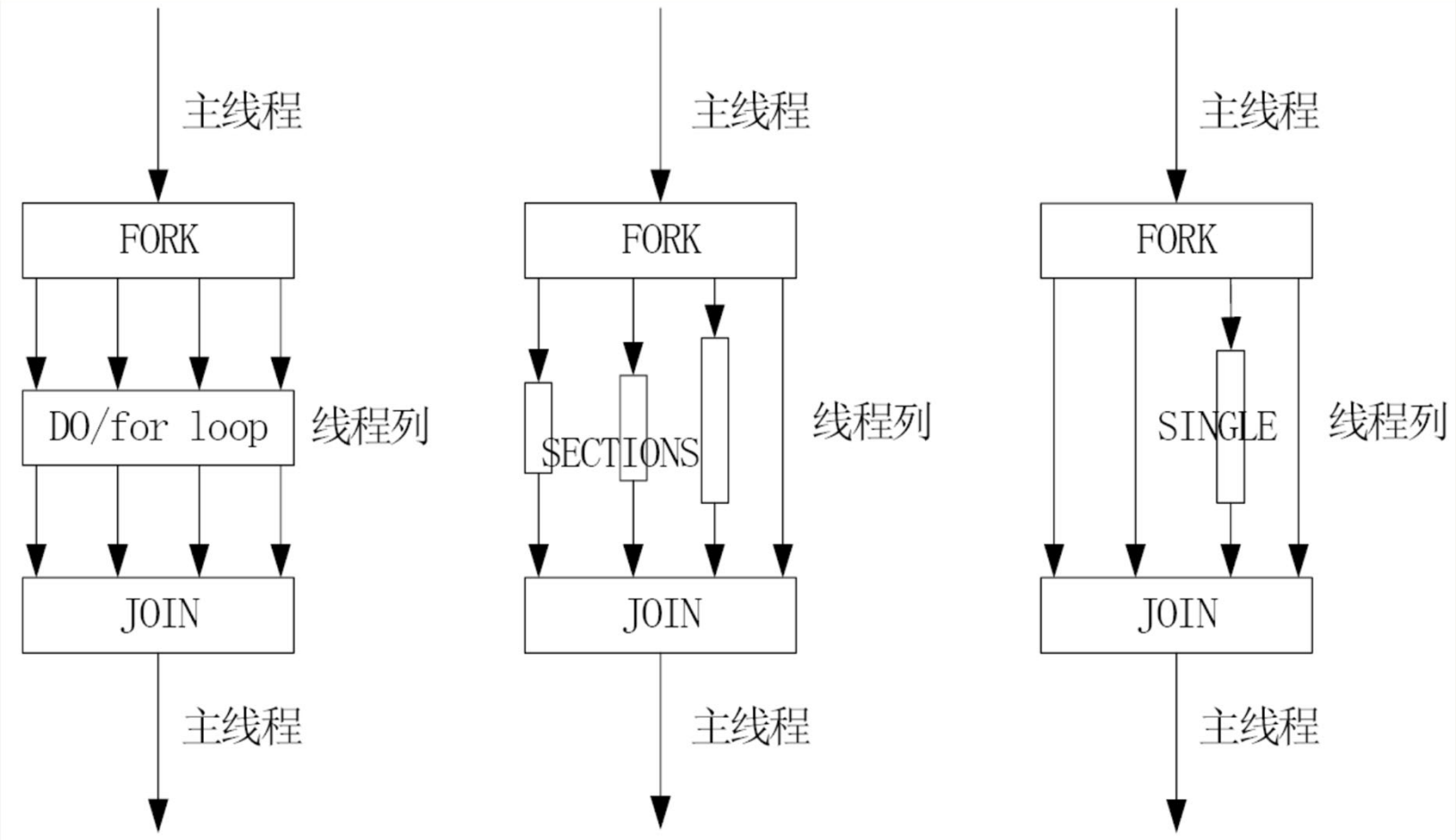
并性域结构：REDUCTION子句
（使用REDUCTION子句并发程序中的数据竞争问题）

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

static int data;

int main() {
  #pragma omp parallel num_threads(2) reduction(+:data)
  {
    for(int i = 0; i < 10000; i++) {
      data++;
      usleep(10);
    }
    printf("data = %d tid = %d\n", data, omp_get_thread_num());
  }

  printf("In main function data = %d\n", data);
  return 0;
}
```

2.任务划分结构WORK-DISTRIBUTION CONSTRUCTS

- 用来表明任务如何在多个线程间分配，任务划分结构将它所包含的代码划分给线程组的各成员来执行。它不产生新的线程，在任务划分结构的入口点没有路障，但在其结束处有一个隐含的路障。一个共享任务结构必须动态地封装在一个并行域中，以使制导语句可以并行执行。包括：
  - 并行DO/for循环制导，用于数据并行
  - 并行SECTIONS制导，用于功能并行
  - SINGLE制导，用于串行执行

2.任务划分结构WORK-DISTRIBUTION CONSTRUCTS



DO/FOR循环制导

用来将循环划分成多个块，并分配给各线程并行执行，在C语言中使用的是for循环制导

```
#pragma omp for [clauses]
for 循环
```

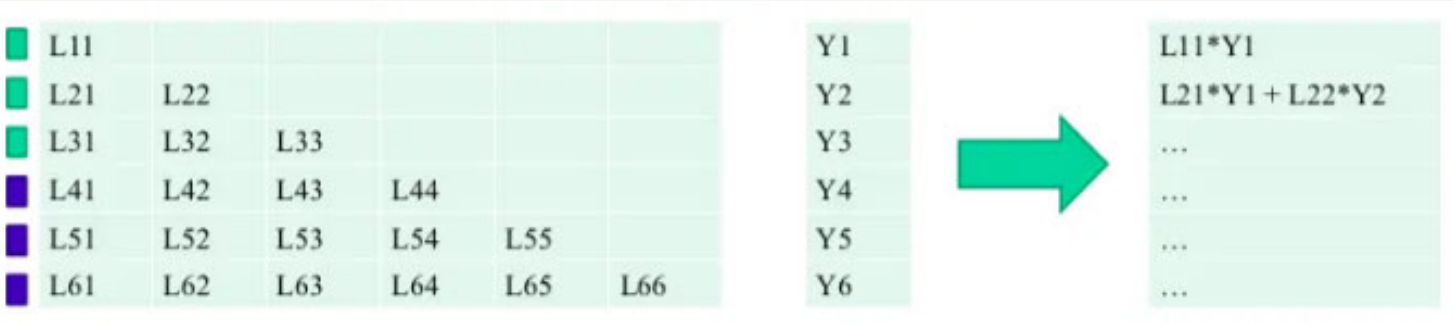
- DO/for循环可以带有PRIVATE和FIRSTPRIVARE等子句
- 循环变量是私有的。

- 可以将do/for制导与并行域制导合并为一句书写

```
#pragma omp parallel for [clauses]
for 循环
```

调度子句SCHEDULE

```
#pragma omp parallel for
for (row=1; row<=6; row++) {
  for (col=1; col<=row; col++) {
    res[row,col] += l[row,col] * y[col];
  }
}
```



调度子句SCHEDULE

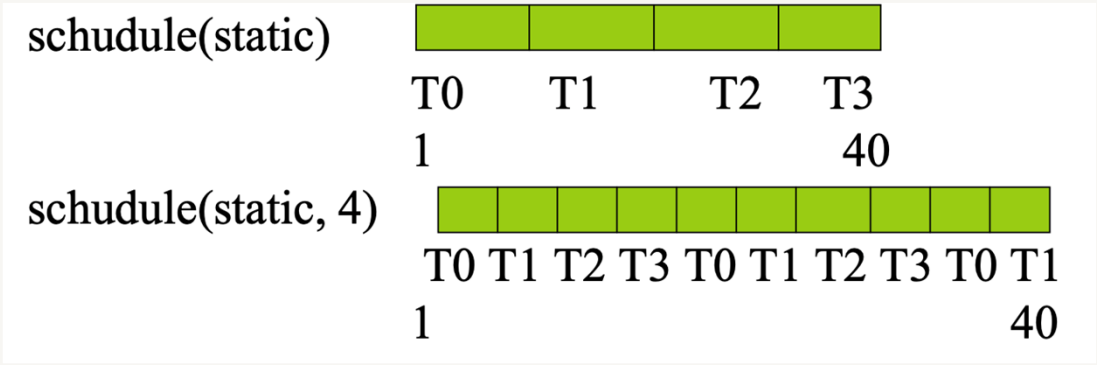
- 控制for循环并行化的任务调度方式（划块方式）

```
schedule(kind[, chunksize])
```

- kind: static, dynamic, guided, runtime
- chunksize是一个整数表达式

SCHEDULE (STATIC [, CHUNKSIZE])

- 省略chunksize，迭代空间被划分成(近似)相同大小的区域，每个线程被分配一个区域;
- 如果chunksize被指明，迭代空间被划分为chunksize大 小，然后被轮转的分配给各个线程
- 例如:假如线程数为4



SCHEDULE(DYNAMIC [,CHUNKSIZE])

- 划分迭代空间为chunksize大小的区间，然后基于先来先服务方式分配给各线程;
- 当省略chunksize时，其默认值为1。

SCHEDULE(GUIDED[,CHUNKSIZE])

- 类似于DYNAMIC调度，但区间开始大，然后迭代区间越来越小
- chunksize说明最小的区间大小。当省略chunksize时，其默认值为1

SCHEDULE(RUNTIME)

- 调度选择延迟到运行时，调度方式取决于环境变量OMP_SCHEDULE的值，例如:

```
export OMP_SCHEDULE=“DYNAMIC, 4”
```

- 使用RUNTIME时，指明chunksize是非法的;

注意数据竞争

错

```
#pragma omp parallel for
for(k=0;k<100;k++) {
        x=array[k];
        array[k]=do_work(x);
}
```

```
#pragma omp parallel for private(x)
for(k=0;k<100;k++) {
        x=array[k];
        array[k]=do_work(x);
}
```

对

- 正确决定变量类型是private还是shared，从而避免数据竞争
- 在parallel结构中声明的变量是私有的

注意循环依赖

- 流依赖
  （跨迭代写后读）

```
for(j=1; j<MAX; j++){
  A[j] = A[j-1];
}
```

- 反依赖
  （跨迭代读后写）

```
for (j=1; j<MAX; j++){
  A[j] = A[j+1];
}
```

- 写依赖
  （跨迭代写相关）

```
for (j=1; j<MAX; j++){
  A[j] = B[j];
  A[j+1] = C[j];
}
```

SECTIONS制导

将一个任务划分为多个互不相关的分段子任务（结构化块），各结构化块在各线程间并行执行

```
#pragma omp sections [clauses]
{
        [#pragma omp section]
        structured-block-sequence
        [#pragma omp section
        structured-block-sequence]
        ...
}
```

- sections制导可以带有PRIVATE、 FIRSTPRIVATE和其它子句
- 每个section必须包含一个结构块block（{}）

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            cout<<"section 1 线程ID: "<<omp_get_thread_num()<<"\n";
        }
        #pragma omp section
        {
            cout<<"section 2 线程ID: "<<omp_get_thread_num()<<"\n";
        }
    }
}
```

- 可以将section制导与并行域制导合并为一句书写

```
#pragma omp parallel sections [clauses]
...
```

SINGLE制导

- 结构体代码仅由一个线程执行;并由首先执行到该代码的线程执行;其它线程等待直至该结构块被执行完 。
- 意义?

```
   #pragma omp single [clauses]
   structured-block
// example
#pragma omp parallel {
        setup(x);
        #pragma omp single {
                input(y);
        }
        work(x,y);
}
```

3.同步结构 SYNCHRONIZATION CONSTRUCTS

- 同步结构用于控制执行过程中各线程的同步。七种典型的同步结构：
- master 制导语句：指定代码段将只由主线程执行，该线程组中的其他线程将忽略此代码段；
- critical制导语句：指定代码段为线程互斥临界区，在同一时刻只能由一个线程执行；
- barrier制导语句：用于同步一个线程组中的所有线程，先执行到达该语句的线程阻塞；
- atomic制导语句：指定特定的存储单元被原子地更新；
- flush制导语句：用于标识一个同步点，以确保所有线程看到一致的存储器视图；
- ordered制导语句：指定代码中所包含的循环以串行方式执行，任何时候只能有一个线程执行这个部分。
- threadprivate制导语句：使一个全局文件作用域的变量在并行域内变成每个线程私有，每个线程对该变量复制一份私有拷贝。

CRITICAL制导

- 由critical指令指定的代码区域（临界区），一次只能由一个线程执行。
- 如果一个线程当前正在一个critical区域内执行，而另一个线程到达该区域并试图执行它，它将阻塞，直到第一个线程退出该区域。

```
#pragma omp critical [(name)]
structured-block
```

CRITICAL VS SINGLE

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
   #pragma omp single
   a++;
   #pragma omp critical
   b++;
}
printf("single: %d -- critical: %d\n", a, b);
```

single: 1 -- critical: 4

- single指定一段代码应由单线程执行（不一定是主线程）
- critical指定代码一次由一个线程执行

所以前者只会执行一次，而后者会执行执行次数。

- 如果为临界区指定了name，该名称充当critical区域全局标识符，相同名称的不同临界区域被视为同一区域
- 所有未命名的critical区域均视为同一区域

```
#pragma omp parallel sections
{
        #pragma omp section
        {
                #pragma omp critical (critical1)
                {
                        printf("section1, critical1");
                }
        }
        #pragma omp section
        {
                #pragma omp critical (critical2)
                {
                        printf("section2, critical2");
                }
        }
        #pragma omp section
        {
                #pragma omp critical (critical2)
                {
                        printf("section3, critical2");
                }
        }
}
```

不同名称的多个critical区域

```
#pragma omp parallel sections
    {
    #pragma omp section
        {
            #pragma omp critical (critical1)
            {
                for (int i=0; i < 5; i++)
                {
                    printf("section1 thread %d excute i = %d\n", omp_get_thread_num(
                    Sleep(200);
                }
            }

        }

    #pragma omp section
        {
            #pragma omp critical (critical2)
            {
                for (int j=0; j < 5; j++)
                {
                    printf("section2 thread %d excute j = %d\n", omp_get_thread_num(
                    Sleep(200);
                }
            }
        }

    }
```

```
section1 thread 3 excute i = 0
section2 thread 1 excute j = 0
section2 thread 1 excute j = 1
section1 thread 3 excute i = 1
section2 thread 1 excute j = 2
section1 thread 3 excute i = 2
section2 thread 1 excute j = 3
section1 thread 3 excute i = 3
section2 thread 1 excute j = 4
section1 thread 3 excute i = 4
```

相同名称的多个critical区域

```
#pragma omp parallel sections
    {
    #pragma omp section
        {
            #pragma omp critical (critical1)
            {
                for (int i=0; i < 5; i++)
                {
                    printf("section1 thread %d excute i = %d\n", omp_get_thread_num
                    Sleep(200);
                }
            }

        }

    #pragma omp section
        {
            #pragma omp critical (critical1)
            {
                for (int j=0; j < 5; j++)
                {
                    printf("section2 thread %d excute j = %d\n", omp_get_thread_num
                    Sleep(200);
                }
            }
        }

    }
```

```
section2 thread 3 excute j = 0
section2 thread 3 excute j = 1
section2 thread 3 excute j = 2
section2 thread 3 excute j = 3
section2 thread 3 excute j = 4
section1 thread 2 excute i = 0
section1 thread 2 excute i = 1
section1 thread 2 excute i = 2
section1 thread 2 excute i = 3
section1 thread 2 excute i = 4
```

BARRIER制导

- barrier指令同步所有线程，组内任何线程到达barrier指令时将在该点等待，直到所有其他线程都到达该barrier处为止。然后所有线程才继续并行执行后续代码
- 在DO/FOR、SECTIONS和SINGLE等制导后，有一个隐式barrier存在

```
#pragma omp barrier
```

ATOMIC制导

- ATOMIC编译制导表明一个特殊的存储单元只能原子的更新，而不允许让多个线程同时去写，一般用于对共享变量的操作
- 提供了一个最小的临界区（critical），其效率比临界区高

```
#pragma omp atomic
statement
```

FLUSH制导

- flush指令标识一个同步点，在该点上list中的变量都要被写回内存，而不是暂存在寄存器中，保证线程读取到的共享变量的最新值，从而保证多线程数据的一致性。

```
#pragma omp flush
```

以下指令隐含flush操作：

- barrier、parallel、critical、ordered
- for、sections、single
- atomic修饰的语句

ORDERED制导

- 在并行化的for循环中，指定一部分代码应当按循环迭代顺序执行
- 只能用于带有ordered子句的for或parallel for结构当中，且在一个循环中只能出现一次ordered制导

```
#pragma omp ordered
structured-block
```

```
#pragma omp parallel
    {
    #pragma omp for ordered
        for (int i = 0; i < 10; ++i)
        {
        #pragma omp ordered
            {
                printf("thread %d excute i = %d\n", omp_get_thread_num(), i);
            }
        }
    }
```

```
thread 0 excute i = 0
thread 0 excute i = 1
thread 0 excute i = 2
thread 1 excute i = 3
thread 1 excute i = 4
thread 1 excute i = 5
thread 2 excute i = 6
thread 2 excute i = 7
thread 3 excute i = 8
thread 3 excute i = 9
```

```
void test(int first, int last)
{
    #pragma omp for schedule(static) ordered
    for (int i = first; i <= last; ++i) {
        // Do something here.
        if (i % 2)
        {
            #pragma omp ordered
            printf_s("test() iteration %d\n", i);
        }
    }
}
```