



OpenMP 并行编程 (一)

- 并行编程介绍
- 并行域与工作共享



目录页

Contents

1

并行编程介绍

2

OpenMP 概述

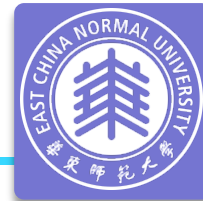
3

并行域创建

4

工作共享结构

- ❑ OpenMP Specifications, <https://www.openmp.org/specifications>
- ❑ Using OpenMP – The Next Step, van der Pas et al, 2017
- ❑ Using OpenMP, Chapman, Jost, and Van Der Pas, 2007



1

并行编程介绍

- 1 并行编程介绍
- 2 OpenMP 概述
- 3 并行域创建
- 4 工作共享结构

- 并行程序设计模型
- 并行编程模型
- 并行化方法
- 并行算法设计原则

并行程序设计模型



■ 并行程序设计模型

- 隐式并行 (Implicit Parallel)
- 数据并行 (Data Parallel)
- 共享变量 (Shared Variable)
- 消息传递 (Message Passing)



■ 隐式并行

- 编写串行程序
- 通过编译器和运行支持系统将串行程序自动并行化
- 特点：语义简单，可移植性好，易于调试和验证
- 缺点：细粒度并行，效率很低

■ 数据并行

- **SIMD**（单指令流多数据流）
- 同一操作同时作用到一组数据上
- 特点：单一地址空间，编程简单，
松散同步，隐式交互，隐式数据分配
- 缺点：并行粒度局限于数据级并行，粒度小
- 典型代表：**Fortran 90, HPF**

SIMD: Single Instruction Multiple Data

■ 共享变量

- 适用于 SMP 和 DSM

- 特点：松散同步，多线程（SPMD，MPMD）

单一地址空间，显式同步，隐式通信，隐式数据分布

- 典型代表：OpenMP，Pthreads

SMP: Shared Memory Processors

DSM: Distributed Shared Memory

SPMD: Single Program Multiple Data

MPMD: Multiple Program Multiple data

OpenMP: Open Multi-Processing

Pthreads: POSIX threads

■ 消息传递

- **MPP、COW 的自然模型**
- **特点：异步并行，多线程，多地址空间，
显式同步，显式通信，显式数据映射和负载分配**
- **典型代表：MPI，PVM**

MPP: Massively parallel processing

COW: Cluster of Workstations

MPI: Message Passing Interface

PVM: Parallel Virtual Machine



■ 并行编程模型标准

- 数据并行: **Fortran 90, HPF**; 适用于 **SMP, DSM**
- 共享内存: **OpenMP, Pthreads**; 适用于 **SMP, DSM**
- 消息传递: **MPI, PVM**; 适用于所有并行机
- 三者可混合使用



■ 基本并行化方法

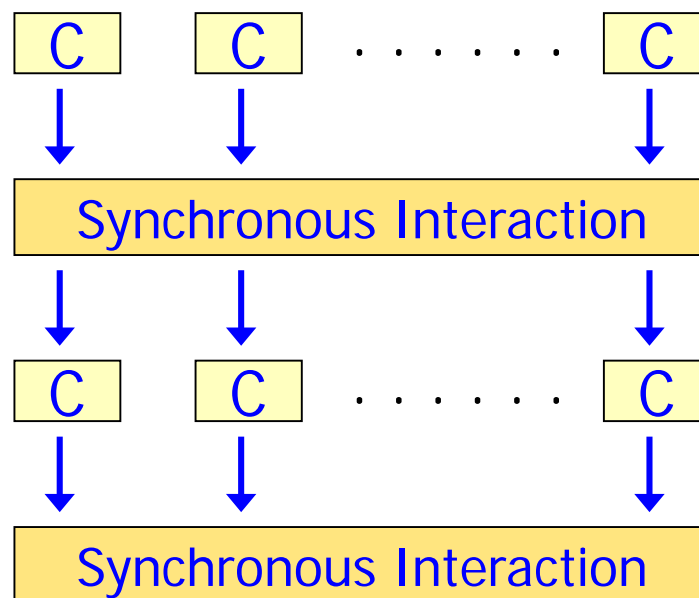
- 相并行 (Phase Parallel)
- 流水线并行 (Pipeline Parallel)
- 主从并行 (Master-Slave Parallel)
- 分而治之并行 (Divide and Conquer Parallel)
- 工作池并行 (Work Pool Parallel)

并行化方法



● 相并行

- 一组超级步（相）
- 步内各自计算
- 步间通信同步
- 方便差错和性能分析
- 计算和通信不能重叠



● 流水线并行

- 将分成一系列子任务 t_1, t_2, \dots, t_m ，一旦 t_1 完成，后继的子任务就立即开始，并以同样的速率进行计算
- 一组进程，流水线作业，流水线设计技术

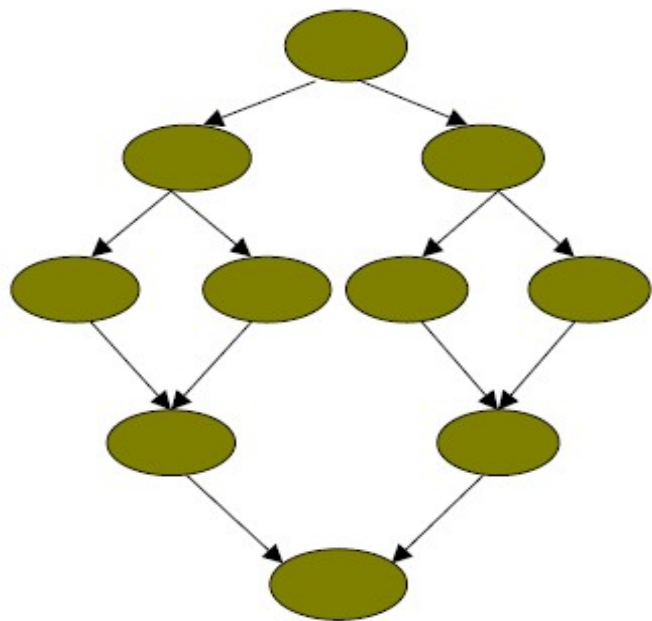
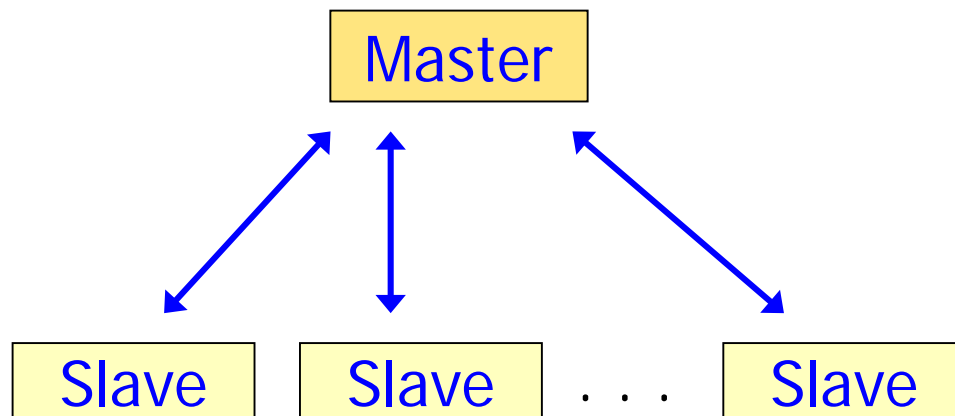


并行化方法



- 主从并行

- 主进程：串行，协调任务
- 子进程：计算子任务
- 与相并行结合
- 主进程易成为瓶颈



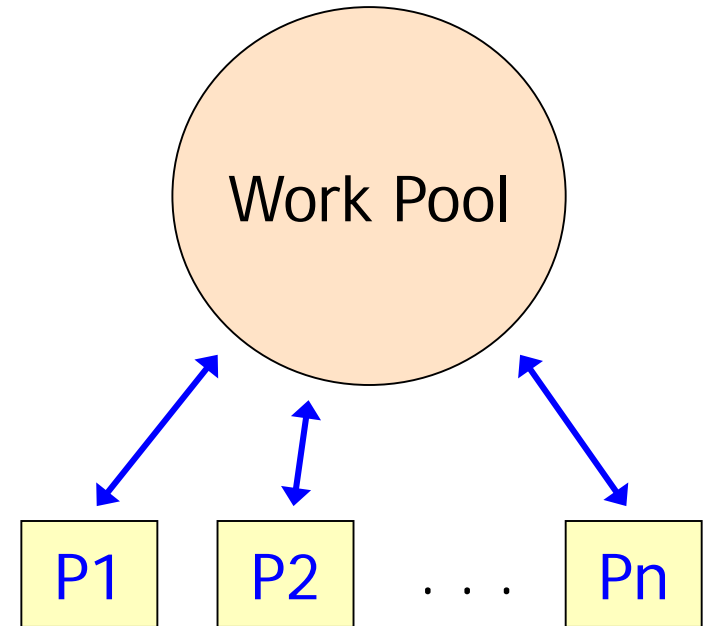
- 分而治之

- 将问题分解成若干特征相同的子问题，分而治之
- 父进程把负载分割并指派给子进程
- 重点在于归并
- 难以负载平衡

并行化方法



- 工作池并行
 - 初始状态：一件工作
 - 进程从池中取任务执行
 - 可产生新任务放回池中
 - 直至任务池为空
 - 易于负载均衡



■ 并行算法设计基本原则 Parallel Programming Paradigms

- 与体系结构相结合
- 具有可扩展性
- 粗粒度
- 减少通信
- 优化性能

粒度是指各个线程可以独立并行执行的任务的大小，是一个相对的概念，与并行度和并行机相关。一般可理解为：

- 细粒度：基于向量和循环系级并行
- 中粒度：较大的循环级并行
- 大粒度：任务级并行（如：区域分解）

■ 并行程序设计步骤

● 划分（**Partitioning**）

将计算任务划分成尽可能多的小任务

划分方法主要有：数据分解（区域分解）和功能分解

● 通信（**Communication**）

确认各任务间的数据交流，评估任务划分的合理性

● 组合（**Agglomeration**）

依据任务的局部性，将小任务组合成大任务，减少通信

● 映射（**Mapping**）

将组合后的任务分配到各个线程，力争负载平衡



2

OpenMP 概述

- 1 并行编程介绍
- 2 OpenMP 概述
- 3 并行域
- 4 工作共享结构

- OpenMP 发展
- OpenMP 并行编程模式
- 编译指导指令
- OpenMP 子句

OpenMP 简介



- **OpenMP** 是基于共享存储体系结构的一个并行编程标准。
- **OpenMP** 通过在源代码（串行）中添加 **OpenMP** 指令和调用 **OpenMP** 库函数来实现在共享内存系统上的并行执行。
- **OpenMP** 为共享内存并行程序员提供了一种简单灵活的开发并行应用的接口模型，使程序既可以在台式机上执行，也可以在超级计算机上执行，具有良好的可移植性。

Jointly defined by a group of major computer hardware and software vendors and major parallel computing user facilities, the OpenMP API is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from embedded systems and accelerator devices to multicore systems and shared-memory systems. <https://www.openmp.org>

API: Application Programming Interface

几点注记

- 编译时不打开 OpenMP 编译选项，则编译器将忽略 OpenMP 指令，从而生成串行可执行程序（串行等价性）
- 打开 OpenMP 编译选项，编译器将对 OpenMP 指令进行处理，编译生成 OpenMP 并行可执行程序
- 并行线程数可以在程序启动时利用环境变量等方法进行动态设置
- 编程方式：增量并行
- 支持与 MPI 混合编程

OpenMP 的使用



OpenMP

- 起源于 ANSI X3H5 (1994) 草案
- 1997年, 部分设备商和编译器开发商组成 ARB (架构审查委员会), 着手制定 OpenMP 标准化规范
- 编程简单, 增量化并行, 移植性好, 可扩展性好
- 主流 Fortran, C/C++ 编译器都支持 OpenMP
(<https://www.openmp.org/resources/openmp-compilers-tools/>)
- 支持 Unix, Linux, Windows 等操作系统
- AMD, Intel, IBM, Cray, NEC, HP, NVIDIA,

ARB: Architecture Review Board



发展进程

- **FORTTRAN version 1.0 – (October 1997)**
- **C/C++ version 1.0 – (October 1998)**
- **FORTTRAN version 2.0 – (November 2000)**
- **C/C++ version 2.0 – (March 2002)**
- **OpenMP 2.5 – (May 2005)**
- **OpenMP 3.0 – (May 2008)**
- **OpenMP 3.1 – (July 2011) // 本讲义以此版本为主**
- **OpenMP 4.0 – (July 2013)**
- **OpenMP 4.5 – (Nov 2015)**
- **OpenMP 5.0 – (Nov 2018)**

OpenMP 并行编程模式



OpenMP 是基于**线程**的并行编程模型。

OpenMP 编程三要素

- 编译指导 (Compiler Directive)
- 运行库函数 (Runtime Library Routines)
- 环境变量 (Environment Variables)

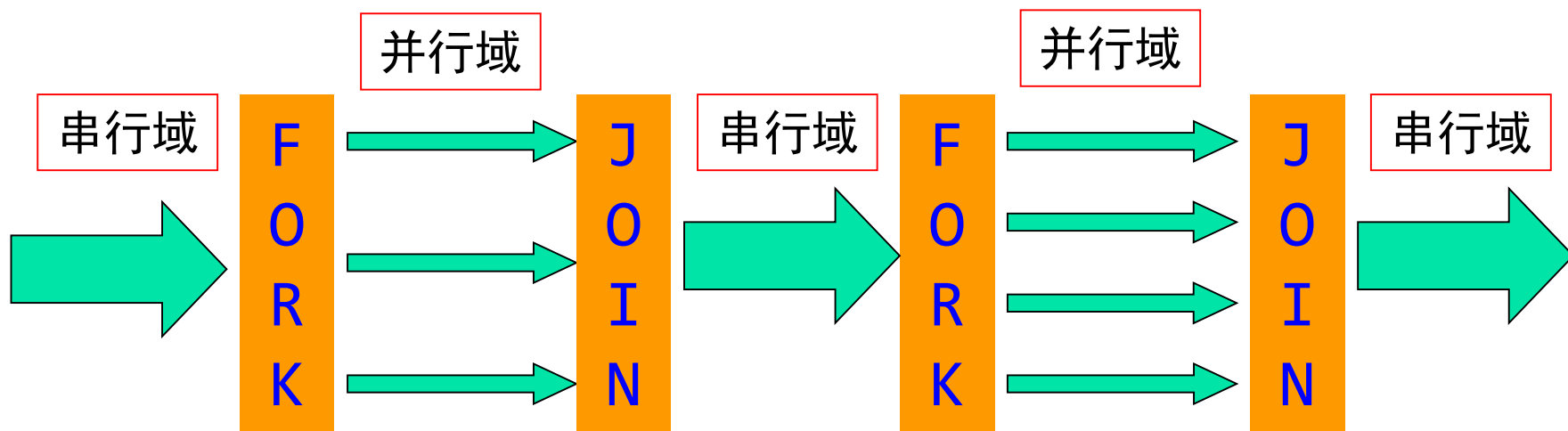
OpenMP 并行方式



OpenMP 采用 **Fork-Join** 并行执行方式

- OpenMP 程序开始于一个单独的主线程（Master Thread），然后主线程一直串行执行，直到遇见第一个**并行域**（Parallel Region），然后开始并行执行并行域。并行域代码执行完后再回到主线程，直到遇到下一个并行域，以此类推，直至程序运行结束。

Fork-Join



- **Fork:** 主线程创建一个并行线程队列，然后，并行域中的代码在不同的线程上并行执行
- **Join:** 当并行域执行完之后，它们或被同步，或被中断，最后只有主线程继续执行

† 并行域可以嵌套

C 举例



```
#include <omp.h>
#include <stdio.h>
int main()
{
    int nthreads, tid;
    #pragma omp parallel private(nthreads,tid)
    {
        tid=omp_get_thread_num(); // 获取线程号
        printf("Hello world from OpenMP thread %d\n", tid);
        if (tid==0)
        {
            nthreads=omp_get_num_threads(); // 获取线程个数
            printf("Number of threads %d\n", nthreads);
        }
    }
    return 0;
}
```

- 头文件: `omp.h`
- OpenMP 编译指导标识符 `#pragma omp`

- 编译
`gcc -fopenmp hello.c`



■ 几点说明 (C/C++)

- C/C++ 的 OpenMP 指令标识符为 `#pragma omp`
- C/C++ 程序中, OpenMP 指令区分大小写
- 每个 OpenMP 指令后是一个结构块 (用大括号括起来)

■ OpenMP 并行程序编写方法

增量并行：逐步改造现有的串程序，每次只对部分代码进行并行化，这样可以逐步改造，逐步调试。

■ OpenMP 源程序的编译

```
gcc -fopenmp hello.c  
icc -openmp hellp.c    // Intel C Compiler
```

OpenMP 通过对串行程序添加**编译指导指令**实现并行化

编译指导指令

- **并行域指令**：生成并行域，即产生多个线程以并行执行任务，所有并行任务必须放在并行域中才可能被并行执行
- **工作共享指令**：负责任务划分，并分发给各个线程，工作共享指令不能产生新线程，因此必须位于并行域中
- **同步指令**：负责并行线程之间的同步
- **数据环境**：负责并行域内的变量的属性（共享或私有），以及边界上（串行域与并行域）的数据传递

● 并行域指令 **Parallel Constructs**

<code>parallel</code>	创建一个并行域
-----------------------	---------

```
#pragma omp parallel private(tid)
{
    tid=omp_get_thread_num(); // Obtain 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);
    }
}
```

OMP_hello.c

● 工作共享结构 Work-Sharing Constructs

for	创建循环共享结构，代表典型的数据并行
sections /section	创建 sections 结构，将任务划分成独立的子任务(section)，每个子任务由一个线程执行，典型的任务并行
single	创建仅由一个线程执行的任务，先到先执行，其他线程等待其执行结束后再一起执行后面的任务
master	与 single 类似，但指定由主线程执行，而且其他线程无需等待
task taskyield	创建一个显式任务，可以立即被执行，也可以挂起并推迟执行，便于实现一些复杂结构，如递归。
workshare	仅适用 Fortran

● 同步结构 Synchronization Constructs

<code>critical</code>	避免线程竞争，其包含的代码同一时刻只能有一个线程执行
<code>barrier</code>	障碍同步：用在并行域内，所有线程执行到 <code>barrier</code> 都要停下等待，直到所有线程都执行到 <code>barrier</code> ，然后再继续往下执行
<code>atomic</code>	确保一个特殊存储单元只能原子更新，即不允许许多线程同时去写，只能用于单一赋值语句等特殊情况
<code>flush</code>	确保线程存储的临时视图与共享存储中的数据一致，并且保证一个变量在共享存储中的读/写顺序
<code>ordered</code>	指定并行域的循环按迭代顺序执行
<code>taskwait</code>	可配合 <code>task</code> 结构使用，创建任务调度点

● 数据环境指令 Data Environment Constructs

<code>threadprivate(list)</code>	<p>将一个或多个私有变量声明为全局的，即在多个并行域中使用时，保留私有变量在上次并行域中的值；</p> <p>可以与 <code>copyin</code> 子句联合使用，将主线程的值广播给其他线程。</p>
----------------------------------	---

子句（**Clause**）：

出现在编译制导指令之后，负责添加一些补充设置

数据共享属性子句



● 数据作用域属性子句 Data Sharing Attribute Clauses

<code>private(list)</code>	<p>创建一个或多个变量的私有拷贝，即在每个线程中都创建一个同名局部变量，但没有初始值；</p> <p>列表中的变量必须已定义，且不能是常量和引用；</p> <p>列表中的多个变量用逗号隔开。</p>
<code>firstprivate(list)</code>	<p>private 的扩展，创建私有拷贝的同时，将主线程中的同名变量的值作为初值。</p>
<code>lastprivate(list)</code>	<p>退出并行域时，将指定的私有拷贝的“最后”值复制到主线程中的同名变量中；</p> <p>“最后”：循环的最后一次迭代（按串行方式），或 sections 的最后一个 section（代码中）；</p> <p>可能会增加额外开销，一般不建议使用，可以用共享变量等方式实现。</p>

数据共享属性子句



- 数据作用域属性子句 Data Sharing Attribute Clauses

<code>shared(list)</code>	指定一个或多个变量为共享变量，即所有线程都可以访问这些变量
<code>default(...)</code>	指定并行域内的变量的缺省属性，C 语言支持 shared 和 none

- 如何决定哪些变量是共享哪些是私有？
 - 通常循环变量、临时变量、写变量一般应设置成私有的；
 - 数组变量、仅用于读的变量通常是共享的；
 - 能设置成共享的变量建议设置成共享的。
 - `default(none)` : 所有变量必须显式指定是私有或共享

† 紧跟 for 结构后面的循环变量默认是私有的，其他循环的循环变量需显式声明成私有的。

数据缺省属性



● 结构内变量的缺省属性

threadprivate 指定的变量	私有
结构内声明的自动存储持续变量	私有
动态存储持续变量	共享
静态数据成员	共享
紧跟 for 或 parallel for 的循环变量	私有
不包含可变成员的常量	共享
结构内声明的静态存储持续变量	共享

数据缺省属性



- 在并行域内，但不在结构内

threadprivate 指定的变量	私有
声明在调用函数中的静态存储持续变量	共享
声明在调用函数中的常量	共享
动态存储持续变量	共享
静态数据成员	共享
引用方式的形参	与实参相同
声明在调用函数中的其他变量	私有

数据共享属性子句



● 数据作用域属性子句（续）

<code>copyin(list)</code>	配合 threadprivate ，用主线程同名变量的值对 threadprivate 的私有拷贝进行初始化
<code>copyprivate(list)</code>	配合 single ，将 single 块中串行计算得到的变量值广播到并行域中其它线程的同名变量中
<code>reduction(op:list)</code>	创建一个或多个变量的私有拷贝，在并行结束后对这些变量执行指定的归约操作（如求和），并将结果返回给主线程中的同名变量

REDUCTION



● 规约操作 reduction

```
#pragma omp parallel reduction(+:mysum) private(i,tid)
{
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();

    for (i=tid+1; i<=n; i=i+nthreads)
    { mysum = mysum + i;    }
}
```

OMP_reduction.c

- 在 **reduction** 子句中，编译器为每个线程创建变量 **mysum** 的私有拷贝
- 退出并行域时，将这些值加在一起并把结果加到原始变量 **mysum** 中
- **reduction** 中的 **op** 操作可以是：**+**，**-**，*****，**&&**，**||**，**&**，**|**，**^** 和内置函数 **max**，**min**

REDUCTION

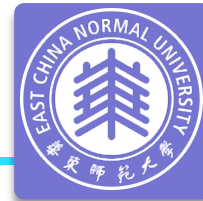


- 规约操作时私有变量的初始值

+	0
*	1
-	0
&&	1
	0
&	~0
	0
^	0
max	相应变量类型的最小值
min	相应变量类型的最大值

● 其它字句

<code>if(log_expr)</code>	条件并行，满足指定条件时才执行相关操作
<code>num_threads(int)</code>	指定并行域内线程的个数
<code>nowait</code>	忽略并行线程或其它制导指令中暗含的障碍同步，使用时需小心
<code>schedule(type,chunk)</code>	指定循环任务的分配规则
<code>ordered</code>	指定循环内的代码按循环顺序执行
<code>collapse(int)</code>	将多重循环转换为一层循环，然后进行任务划分



3

并行域

- 1 并行编程介绍
- 2 OpenMP 概述
- 3 并行域
- 4 工作共享结构

- 并行域创建
- 并行域举例
- 并行域嵌套

■ 并行域的创建

```
#pragma omp parallel // 创建并行域
```

- 产生多个线程，即生成一个并行域
- 并行域中的所有代码默认都将被所有线程并行执行
- 可以通过线程 **id** 给不同线程手工分配不同的任务
- 也可以利用**工作共享指令**给每个线程分配任务
- 并行域可以嵌套
- 并行域结束后，将回到主线程

PARALLEL



Fortran	<code>!\$omp parallel</code> [clause clause ...] <i>structured-block</i> <code>!\$omp end parallel</code>
C/C++	<code>#pragma omp parallel</code> [clause clause ...] { <i>structured-block</i> }

● 结尾处有隐式同步，可用的子句包括：

`if (scalar-logical-expression)`
`num_threads(scalar-integer-expression)`
`default(shared | none)`
`private(list), firstprivate(list)`
`shared(list)`
`copyin(list)`
`reduction(op : list)`

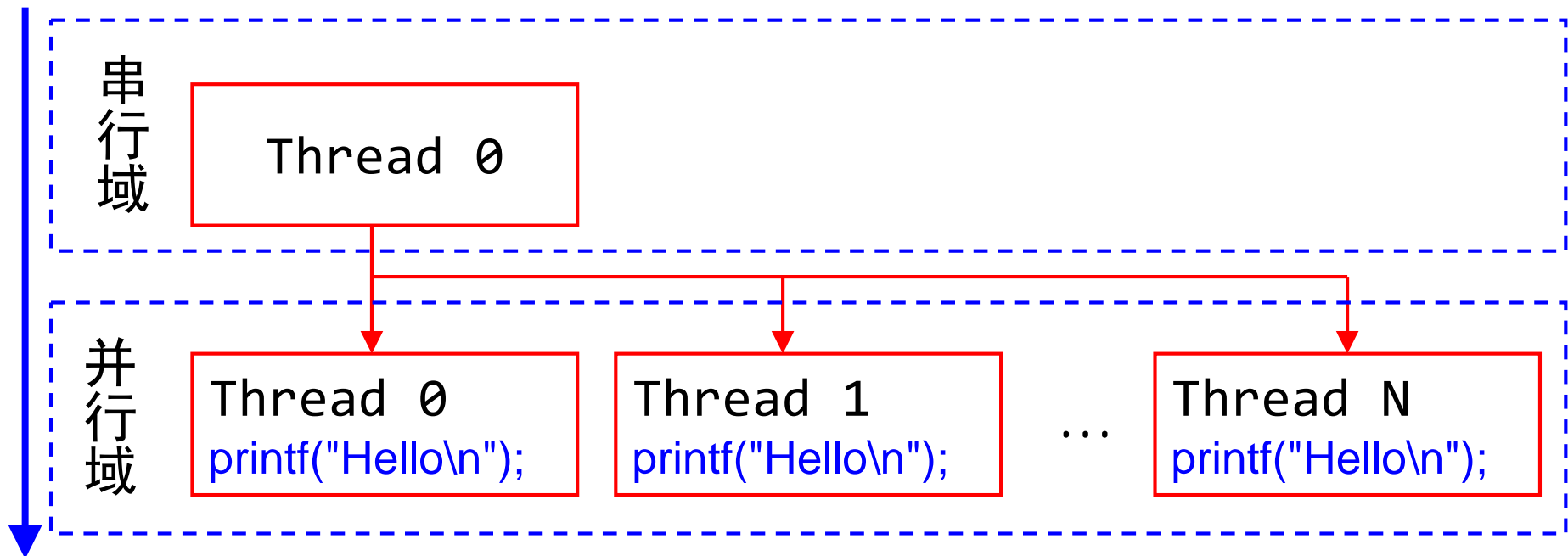
子句用来添加一些补充信息。
若有多个，则用空格隔开。

若没有指定线程个数，则产生最大可能的线程个数。

PARALLEL 举例



```
#pragma omp parallel
{
    printf("Hello\n");
}
```



PARALLEL举例



例：指定线程个数，设置变量属性

```
#pragma omp parallel private(tid) num_threads(3)
{
    . . . . .
}
```

OMP_parallel_01.c

PARALLEL 嵌套



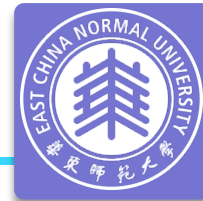
● PARALLEL 可以嵌套

OMP_parallel_02.c

```
omp_set_nested(1); // 打开并行域嵌套功能
#pragma omp parallel private(tid) num_threads(2)
{
    tid=omp_get_thread_num();
    printf("Hello world from OpenMP thread %d\n", tid);

    #pragma omp parallel private(tid) num_threads(3)
    {
        tid=omp_get_thread_num();
        printf("Hello math from OpenMP thread %d\n", tid);
    }
}
```

† 缺省不支持嵌套，需要利用 OpenMP 的 API 过程 **omp_set_nested** 开启嵌套功能（该过程的缺省值是 **false**）



4

工作共享结构

- 1 并行编程介绍
- 2 OpenMP 简介
- 3 并行域
- 4 工作共享结构

- 工作共享指令分类
- 循环共享结构
- SCHEDULE 任务调度
- 数据的共享和私有
- 规约操作

■ 工作共享指令

- 负责任务的划分和分配,
 - 在每个工作分享结构入口处无需同步
 - 每个工作分享结构结束处会隐含障碍同步
-
- `for` 指令: 自动划分和分配循环任务
 - `sections` 指令: 手动划分任务
 - `single` 指令: 指定并行域中的串行任务
 - `master` 指令: 指定仅由主线程执行的串行任务

循环共享



Fortran	<pre>!\$omp do [clause clause ...] <i>do-loops</i> !\$omp end do</pre>
C/C++	<pre>#pragma omp for [clause clause ...] { <i>for-loops</i> }</pre>

- 只负责工作分享，不负责并行域的产生和管理，一般需放在并行域中
- 如果不放在并行域内，则只能串行执行
- 结尾处有隐式同步，可用的子句（clause）包括：

private(*list*), firstprivate(*list*), lastprivate(*list*)
reduction(*op* : *list*)
schedule(*kind*[, *chunk_size*])
ordered
nowait

串行循环

- 将循环变量从初始值开始，逐次递增或递减，直至满足结束条件，其间对每个循环变量的取值都将执行一次循环体内的代码，且循环体内的代码是依次串行执行的

OpenMP 并行循环

- 假定总共有 N 次循环，**OpenMP** 对循环的任务分配就是将这 N 次循环进行划分，然后让每个并发线程各自负责其中的一部分循环工作，因此必须确保每次循环之间的数据的相互独立性！

† 循环变量只能是整型或指针

† 将任务划分后分发给并发进程称为“调度” (schedule)

循环共享举例



OMP_for.c

```
#include <omp.h>
#include <stdio.h>
#define N 10000
int main()
{
    int A[N], B[N], i;
    #pragma omp parallel num_threads(4)
    {
        if (!omp_get_thread_num())
            printf("Number of threads: %d\n", omp_get_num_threads());
        #pragma omp for
        for(i=0; i<N; i++)
        {   B[i]=i; A[i]=2*B[i];   }
    }
    printf("A[n]=%d, B[n]=%d\n", A[N-1], B[N-1]);
    return 0;
}
```

for (循环变量赋初值; 循环条件; 循环变量增量)
 循环体 // 循环体中不能修改循环变量的值

- 在循环共享结构中，将任务划分后分发给各个线程称为**调度**(schedule)
- 任务调度的方式直接影响程序的效率：（1）任务的均衡程度；（2）循环体内数据访问顺序与相应的 cache 冲突情况。

循环体任务的调度基本原则

- 分解代价低：分解方法要快速，尽量减少分解任务而产生的额外开销
- 任务计算量要均衡
- 尽量避免高速缓存（cache）冲突，提高 cache 命中率。

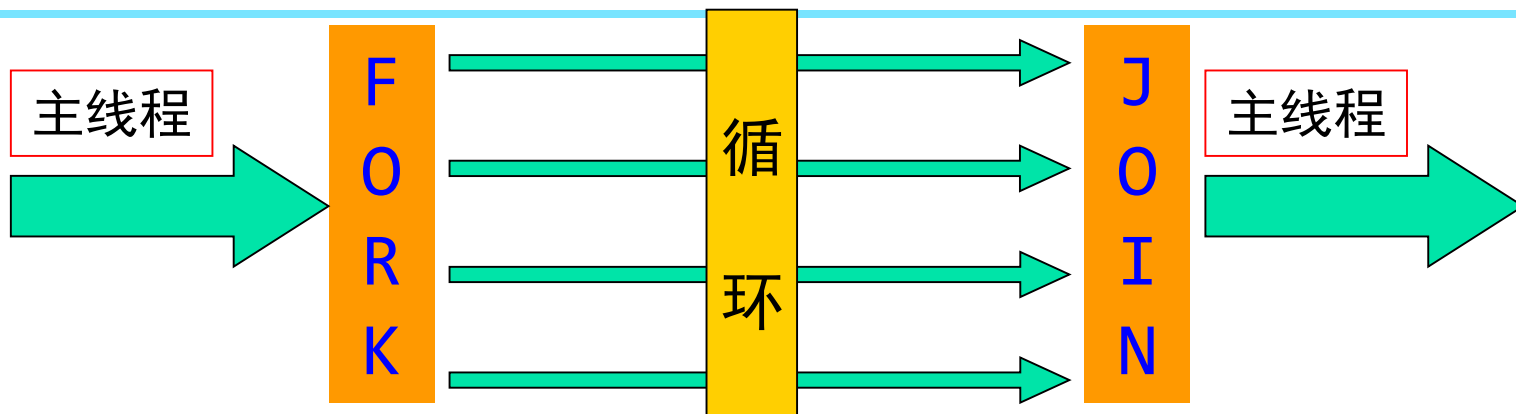
高速缓存 cache



高速缓存（cache）的关键特性是以连续单元的数据块的形式组成的，当处理器需要引用某个数据块的一个或几个字节时，这个块的所有数据就会被传送到高速缓存中。因此，如果接下来需要引用这个块中的其他数据，则不必再从主存中调用它，这样就可以提高执行效率。

在多台处理机系统中，不同的处理器可能需要同一个数据块的不同部分（不是相同的字节），尽管实际数据不共享（处理器有各自的高速缓存），但如果一个处理器对该块的其他部分写入，由于高速缓存的一致性协议，这个块在其他高速缓存上的拷贝就要全部进行更新或者使无效，这就是所谓的“假共享”，它对系统的性能有负面的影响。

比如：两个处理器 A 和 B 访问同一个数据块的不同部分，如果处理器 A 修改了数据，则高速缓存一致协议将更新或者使处理器 B 中的高速缓存块无效。而在此时处理器 B 可能也修改了数据，则高速缓存一致协议反过来又要将处理器 A 中的高速缓存块进行更新或者使无效。如此往复，就会导致高速缓存块的乒乓效应（ping-pong effect）。



■ 任务调度 SCHEDULE

- SCHEDULE(**static**, chunk)
静态分配, chunk 为任务块的大小, 每个任务块被轮转分配给各线程
- SCHEDULE(**dynamic**, chunk)
动态分配, chunk 为任务块的大小, 按先来先服务原则分配
- SCHEDULE(**guided**, chunk)
动态分配, 任务块大小可变, 先大后小, chunk 指定最小任务的大小
- SCHEDULE(**runtime**)
具体调度方式到运行时才进行, 由环境变量 OMP_SCHEDULE 确定

SCHEDULE方式



SCHEDULE(static, chunk)

SCHEDULE(static)

- 循环任务被划分为 chunk 大小的子任务，然后被轮转的分配给各个线程
- 省略 chunk，则循环任务被划分成（近似）相同大小的子任务，每个线程被分配一个子任务；

例：假如线程数为 4，总任务量为 40，则

schudule(static)

T0	T1	T2	T3
----	----	----	----

schudule(static, 4)

T0	T1	T2	T3	T0	T1	T2	T3	T0	T1
----	----	----	----	----	----	----	----	----	----

SCHEDULE方式



SCHEDULE(dynamic, chunk)

SCHEDULE(dynamic)

- 基于先来先服务方式分配给各线程；
- 当省略 chunk 时，默认值为 1

SCHEDULE(guided, chunk)

SCHEDULE(guided)

$$S_k = \frac{R_k}{N}$$

- 类似 dynamic，但任务块开始较大，然后变小，划分方式取决于编译器
- chunk 指定最小任务的大小，省略时默认值为 1

GCC: S_k 第 k 块任务大小， N 线程个数， R_k 剩余循环次数

SCHEDULE(runtime)

- 调度延迟到运行时，调度方式取决于环境变量 OMP_SCHEDULE 的值

并行域与循环合并



- 如果并行域中只有循环共享结构，则可以合写在一起

Fortran	<pre>!\$omp parallel do [clause clause ...] <i>do-loops</i> !\$omp end parallel do</pre>
C/C++	<pre>#pragma omp parallel for [clause clause ...] { <i>for-loops</i> }</pre>

循环举例



OMP_parallel_for.c

```
#define N 100000000
int main()
{
    static int A[N], B[N], i;
    clock_t t0, t1;

    t0 = clock();
    #pragma omp parallel for num_threads(4)
        for(i=0; i<N; i++)
            { B[i]=i; A[i]=2*B[i]; }

    t1 = clock();
    printf("A[n]=%d, B[n]=%d\n", A[N-1], B[N-1]);
    printf("Elapsed time: %.2e\n", (double)(t1-t0)/CLOCKS_PER_SEC);
}
```

举例：PI



$$\pi = \int_0^1 \frac{4}{x^2 + 1} dx$$

■ 中点公式： $\int_a^b f(x)dx \approx h \sum_{i=1}^n f\left(\frac{x_{i-1} + x_i}{2}\right)$

$$h = \frac{b-a}{n}$$

$$x_i = a + ih$$

■ 梯形公式： $\int_a^b f(x)dx \approx h \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2}$

$$= \frac{h}{2} (f(x_0) + f(x_n)) + h \sum_{i=1}^{n-1} f(x_i)$$

PI 串行程序



ex_pi.c

```
const int n=1000000;
inline double f(double x)
{ return 4/(x*x+1); }

int main()
{
    double a=0.0, b=1.0, h=(b-a)/n, mypi=0.0;
    int i;

    mypi = f(a) + f(b);
    for(i=1; i<=n; i++)
    { mypi = mypi + f(a+i*h); }
    mypi = h*mypi;
    printf("mypi=%.10f\n", mypi);
    return 0;
}
```



PI 并行: for + critical

```
#pragma omp parallel for private(i) shared(a,h,mypi)
for(i=1; i<=n; i++)
{
    #pragma omp critical
    mypi = mypi + f(a+i*h);
}
mypi = mypi + f(a) + f(b);
mypi = h*mypi;
```

OMP_pi_critical.c

PI 并行: for



```
double mypi[nthreads];
#pragma omp parallel num_threads(nthreads) private(i,tid) shared(a,h,mypi)
{
    tid = omp_get_thread_num();
    #pragma omp for
    for(i=1; i<=n; i++)
    {
        mypi[tid] = mypi[tid] + f(a+i*h);
    }
}
for(i=1; i<nthreads; i++)
    mypi[0] = mypi[0] + mypi[i];
mypi[0] = mypi[0] + f(a) + f(b);
mypi[0] = h*mypi[0];
```

OMP_pi_for.c

OpenMP编程举例：加速比



OMP_parallel_for_speedup_01.c

OMP_parallel_for_speedup_02.c

OMP_parallel_for_speedup_03.c

OMP_parallel_for_speedup_04.c