高性能计算及应用
OpenMP 导论

SKL

ZJUSCT

目录

- OpenMP 概述
- 并行指导语句
  - 并行区结构体
  - 任务分配结构体
- 数据共享与线程同步
  - 数据共享
  - 线程同步
  - 规约
- 编译运行命令

OpenMP 概述

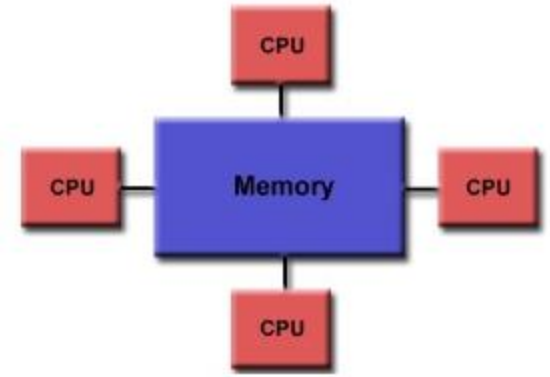
OpenMP 概述

- OpenMP
  - Open Multi-Processing 的缩写
- 一种应用程序接口(API)，可用于显式地指示多线程、共享内存并行性
  - 轻量级、可移植的语法标准
  - 增量式并行
  - 需要编译器支持（C/C++ 或 Fortran）
- 三类主要API
  - 编译器指导指令
  - 运行时库函数
  - 环境变量

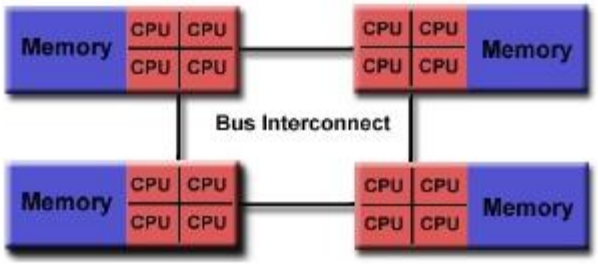
共享内存模型(Shared Memory Model)

- 共享内存含义：所有处理单元与共享内存相连，处于同一个地址空间下，任何处理单元可通过地址直接访问任意内存位置

- OpenMP是为多处理器或多核共享内存机器设计的。底层架构可以是共享内存 UMA 或 NUMA。



Uniform Memory Access
一致内存访问



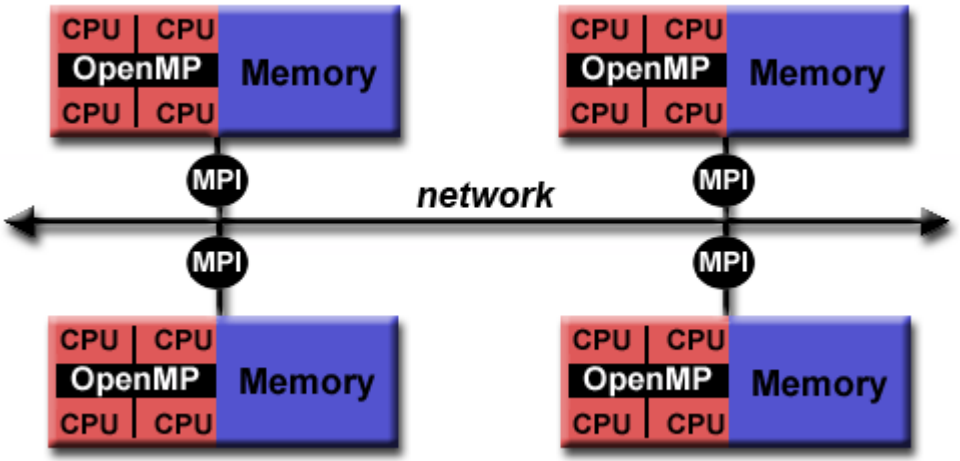
Non-Uniform Memory Access
非一致内存访问

- 由于是共享内存架构，往往只能将 omp 应用在单节点上

共享内存模型(Shared Memory Model)

- 在HPC中往往是四机集群的配置（也可以认为是 lab1 中的四台虚拟机），不同机器之间通过 MPI 与 OMP 相结合实现分布式内存并行，这通常被称为**混合并行编程**。

- OpenMP 用于每个节点上的计算密集型工作。
- MPI 用于实现节点之间的通信和数据共享。



Hybrid OpenMP-MPI Parallelism

- 这使得并行性可以在集群的整个范围内实现。

OpenMP 设计理念：增量式并行

- 先编写串行程序，然后添加并行化指导语句
  - 如同写了一句注释

```
int main() {

  /** 并行化下面的代码 */
  printf( "Hello, World!\n" );

  return 0;
}
```

```
int main() {
  /** 并行化下面的代码 */
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }
  return 0;
}
```

OpenMP 组成

- 编译器预处理指导语句

#pragma omp	指导语句名	[子句,...]	换行
必须	parallel, for 等	可选，可以有若干个，无顺序	必须

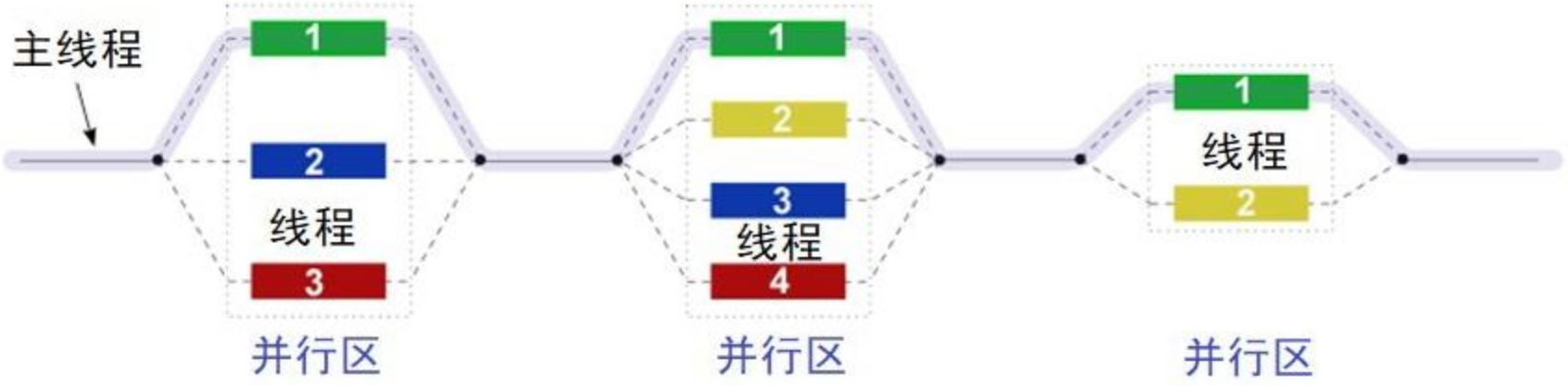
- 例

```
#pragma omp parallel default(shared) private(beta,pi)
```

指导语句名　　子句　　子句

- 基本规则
  - 区分大小写
  - 指导语句作用于语句之后的语句块（例如循环）

Fork-Join 模型：




```
int main() {
  omp_set_num_threads(3);
  #pragma omp parallel
    printf( "Hello, World-1!\n" );

  omp_set_num_threads(4);
  #pragma omp parallel
    printf( "Hello, World-2!\n" );

  return 0;
}
```

并行指导语句

并行指导语句

—并行区结构体

并行区结构体

- 并行区结构体：创建一组 OpenMP 线程以执行一个并行区

```
#pragma omp parallel [子句[ [,]子句] …]
    语句块
```

子句：

- private(列表)
- firstprivate(列表)
- shared(列表)
- copyin(列表)
- reduction([归约修饰符,] 归约操作符: 列表)
- proc_bind(master | close | spread)
- allocate([分配器: 列表)
- if([parallel:] 标量表达式)
- num_thread(整数表达式)
- default(shared | none)

*详细说明及 FORTRAN 用法请参见文档

```
int main() {
  omp_set_num_threads(16);

  /** 并行区 */
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

并行区结构体 – 语义

- 程序运行至 **parallel** 时，创建一组线程
- 所有线程均执行并行区内的代码
- 并行区结束时，所有线程会同步（有一个隐式 barrier）
- 限制：
  - 并行区须为函数内的一个语句块
  - 不能跳转（goto）出入并行区
  - 但并行区内可以调用其他函数

```
int main() {
  omp_set_num_threads(16);

  /** 并行区 */
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

并行区：线程数量

- 调整并行区内线程数量的方式

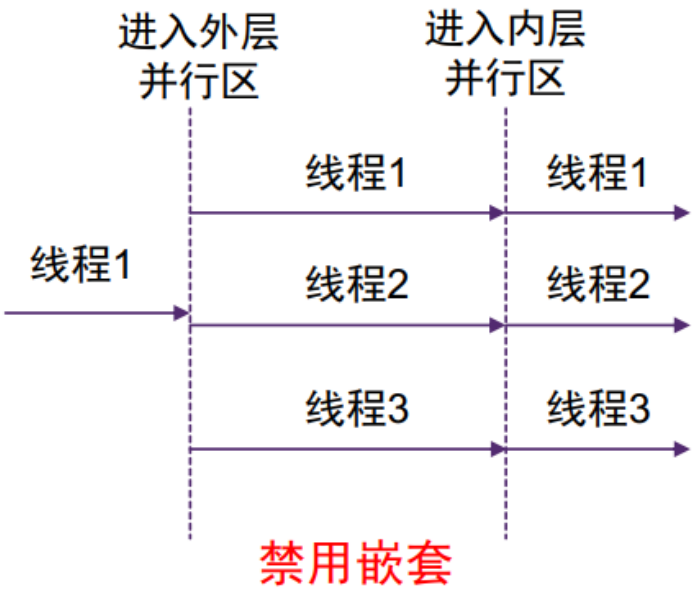
1. 添加 **num_threads** 子句
   - 例：**#pragma omp parallel num_threads(10)**
2. 调用 **omp_set_num_threads()** 函数
   - 在并行区前调用
3. 设置 **OMP_NUM_THREADS** 环境变量
   - 在并行区前调用
4. 添加 **if** 子句
   - 条件为 **false** 时，仅用主线程串行执行
   - 例：**#pragma omp parallel if (para == true)**

- 优先级：
  - pragma 里面指定 > 函数API设置 > 环境变量设置

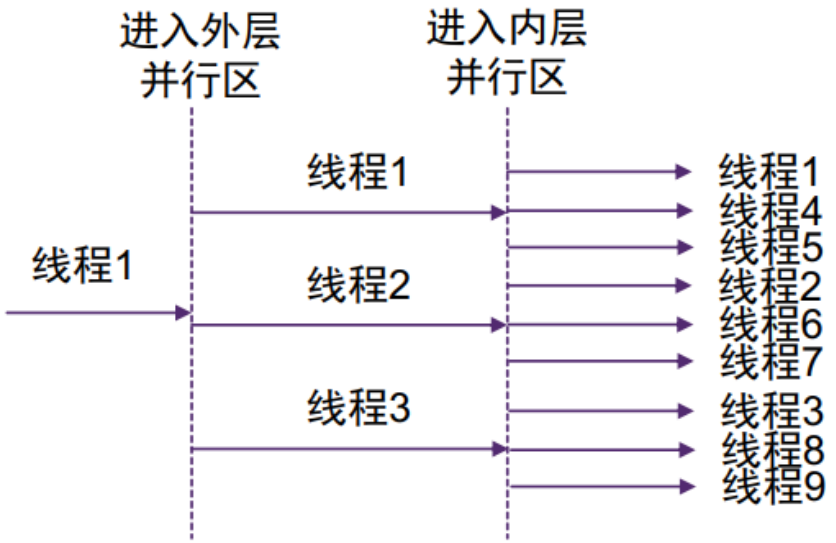
并行区结构体：嵌套并行区

- 除非启用嵌套并行区，内层并行区**只用一个线程**执行
- 启用/禁用嵌套并行区：
  - `omp_set_nested(bool)`
  - 设置 `OMP_NESTED` 环境变量
- 检查嵌套并行区是否开启：
  - `omp_get_nested()`

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



禁用嵌套



启用嵌套

并行指导语句

—任务分配结构体

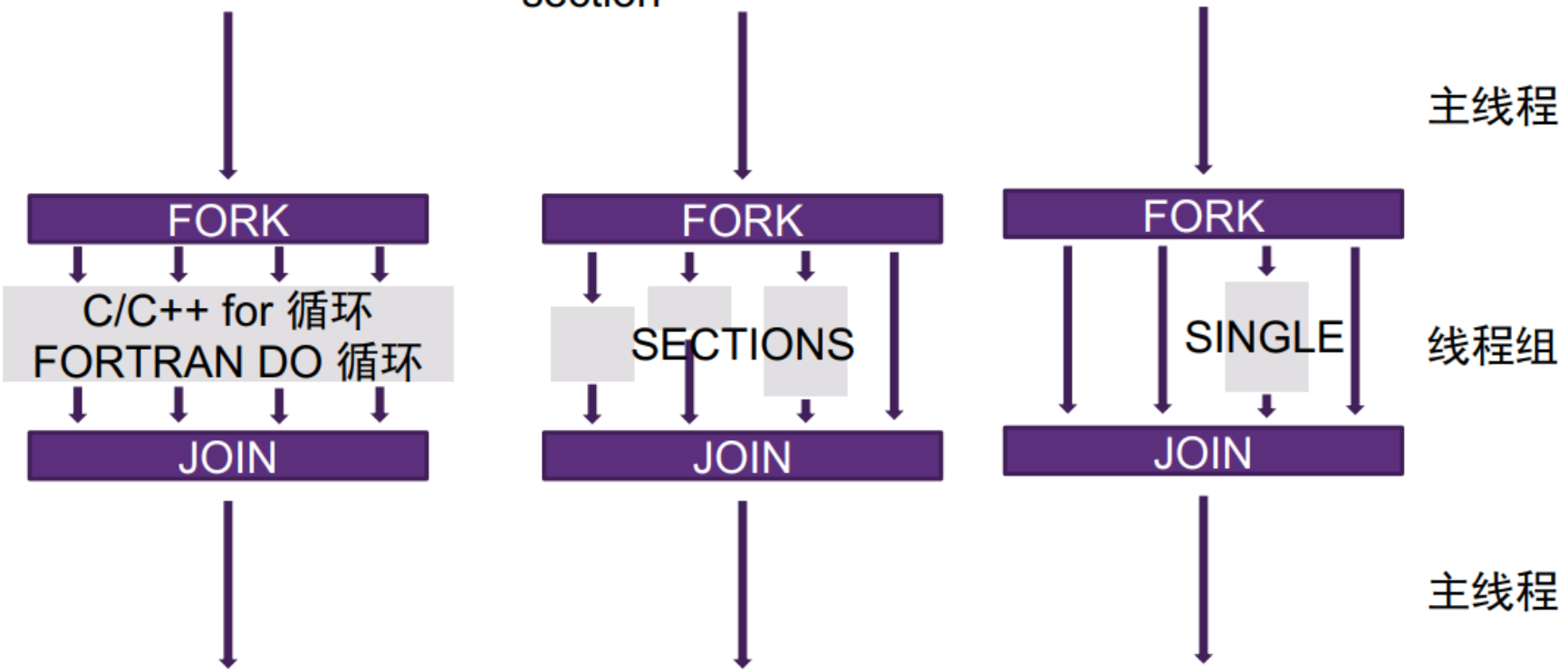
任务分配结构体

- 三种典型的任务分配结构体
  - 注意：任务分配结构体需要包含在并行区内部

DO / for: 将循环的各次迭代分配给不同线程。是一种数据并行

sections: 将任务划分成若干个 section，每个线程执行各自的 section

single: 仅使用并行区中的一个线程执行



任务分配结构体

- 将循环的各次迭代自动分配给并行区线程组中的各个线程执行

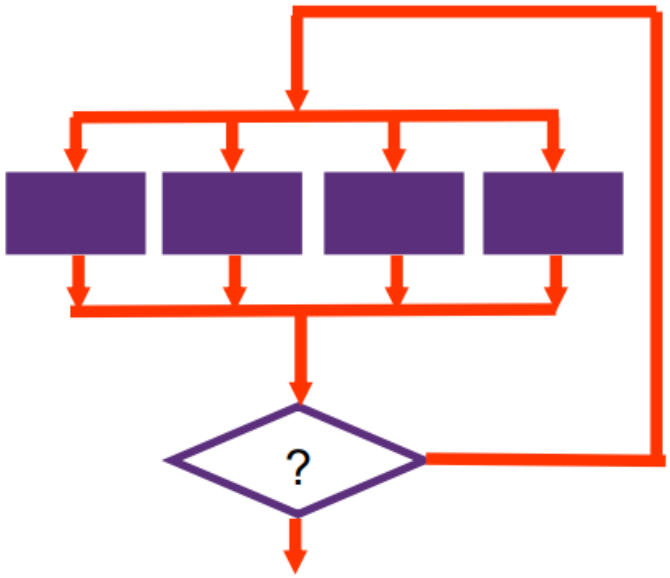
```
#pragma omp for [子句[ [,]子句] …]
    for 循环
```

子句:

- private(列表)
- firstprivate(列表)
- lastprivate([lastprivate修饰符:]列表)
- linear(列表[: 步长])
- schedule([调度修饰符 [,调度修饰符]:] 调度策略 [,块大小])
- collapse(n)
- ordered[(n)]
- allocate([分配器: 列表)
- order(concurrent)
- reduction([归约修饰符,] 归约操作符: 列表)
- nowait

*详细说明及FORTRAN用法请参见文档

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < 25; i++) {
  printf("foo");
}
```



任务分配结构体

- 简便写法：以下两种写法是等价的

```
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < 25; i++) {
    printf("Foo");
  }
}
```

```
#pragma omp parallel for
for (i = 0; i < 25; i++) {
  printf("Foo");
}
```

任务分配结构体：调度方式

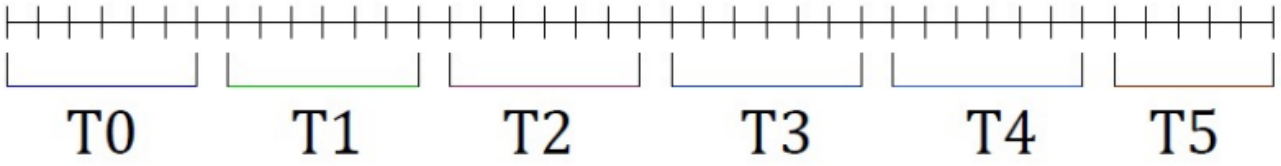
- 调度策略（ **schedule** 子句）：任务分配方式

1. **static**
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. guided
   - 与 dynamic 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：调度方式

static 调度

- 设有N_i次迭代和N_t个线程，为每个线程分配包含N_i/N_t次迭代的一块



- 线程 T_0：迭代 $0 \sim \frac{N_i}{N_t} - 1$
- 线程 T_1：迭代 $\frac{N_i}{N_t} \sim 2\frac{N_i}{N_t} - 1$
- 线程 T_2：迭代 $2\frac{N_i}{N_t} \sim 3\frac{N_i}{N_t} - 1$
- ……
- 线程 $T_{N_t - 1}$：迭代 $(\mathrm{N_t} - 1)\frac{N_i}{N_t} \sim N_i - 1$

任务分配结构体：调度方式

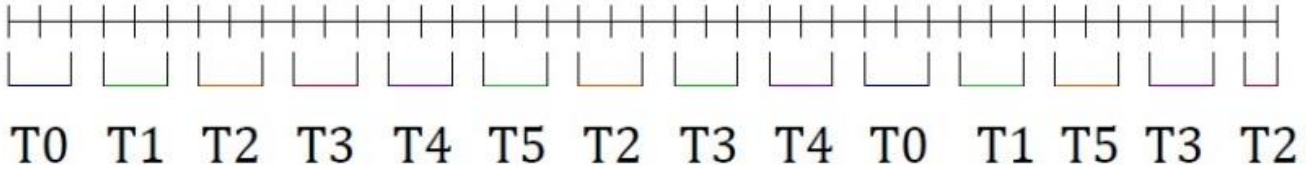
- 调度策略（schedule 子句）

1. static
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. guided
   - 与 dynamic 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：调度方式

dynamic 调度

- 设有N_i次迭代和N_t个线程，每个线程被分配若干个包含 k 次迭代的块



- 当某个线程完成了一块时，该线程会被立即分配新的一块
- 因此，迭代与线程的对应关系是不确定的
- 优点：灵活
- 缺点：高开销—分配过程耗时很长

任务分配结构体：调度方式

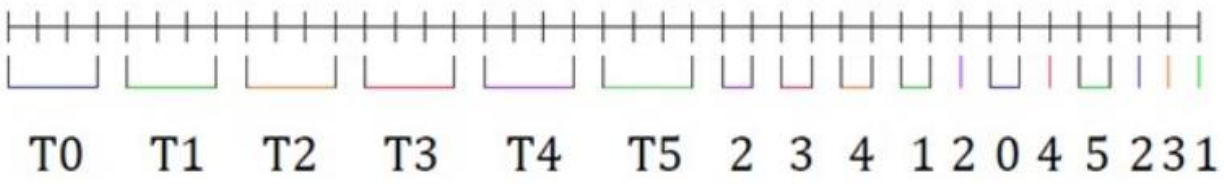
- 调度策略（ schedule 子句）

1. static
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. guided
   - 与 dynamic 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：调度方式

guided 调度

- 设有N_i次迭代和N_t个线程，一开始每个线程被分配一个包含 k 次迭代的块；
- 某个线程完成后，该线程再被分配包含 $k/2$ 个迭代的块；
- 再完成后，会被分配包含 $k/4$ 个迭代的块；以此类推



- 特点：
- 相比 **static**：更好的负载均衡、更多开销
- 相比 **dynamic**：更少开销、没那么好的负载均衡

任务分配结构体：调度方式

- 调度策略（ schedule 子句）

1. static
   - 将各循环迭代组成“块”，“块”数与线程数相等，各“块”包含的迭代次数尽量平均
   - 每个线程执行一“块”
2. dynamic
   - 将各循环迭代分割为“块”，每“块”默认只含一次迭代
   - 每当某线程执行完一“块”，会动态地再请求一块
3. guided
   - 与 dynamic 类似，但块的大小会从大到小动态变化，以改善负载均衡
4. runtime
   - 运行时再通过环境变量 OMP_SCHEDULE 设定
5. auto
   - 由编译器或系统自动决定

任务分配结构体：嵌套循环

- `collapse` 子句：将嵌套循环的迭代统一调度，而不是串行执行内层循环

```
#pragma omp parallel num_thread(6)
#pragma omp for schedule(dynamic)
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
      printf("i=%d, j=%d, thread = %d\n",
             i, j, omp_get_thread_num());
```

```
i=1, j=0, thread = 1
i=2, j=0, thread = 2
i=0, j=0, thread = 0
i=1, j=1, thread = 1
i=2, j=1, thread = 2
i=0, j=1, thread = 0
i=1, j=2, thread = 1
i=2, j=2, thread = 2
i=0, j=2, thread = 0
```

仅仅外层循环被并行、只有3个线程

```
#pragma omp parallel num_thread(6)
#pragma omp for schedule(dynamic)
                    collapse(2)
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
      printf("i=%d, j=%d, thread = %d\n",
             i, j, omp_get_thread_num());
```

```
i=0, j=0, thread = 0
i=0, j=2, thread = 1
i=1, j=0, thread = 2
i=2, j=0, thread = 4
i=0, j=1, thread = 0
i=1, j=2, thread = 3
i=2, j=2, thread = 5
i=1, j=1, thread = 2
i=2, j=1, thread = 4
```

内层和外层循环统一调度、6个线程

任务分配结构体：sections

- 任务分配方式
- 将不同片段的代码组成不同 section，再分配给不同线程
- 各 **section** 命令需要包含在一个 **sections** 命令中
- 每个 section 只被一个线程执行一次
- 线程和 section 的对应关系是不确定的

```
#pragma omp sections [子句[ [,]子句] …]
    {
    [#pragma omp section]
        语句块
    [#pragma omp section]
        语句块
    …
    }
```

子句：

private(列表)
firstprivate(列表)
lastprivate([lastprivate修饰符：]列表)
allocate([分配器： 列表)
reduction([归约修饰符,] 归约操作符：列表)
nowait

*详细说明及FORTRAN用法请参见文档

任务分配结构体：sections

```
#pragma omp parallel num_threads(2)
{
#pragma omp sections
  {
#pragma omp section // 第一个 section
    {
      for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
    }
#pragma omp section // 第二个 section
    {
      for (int i = 0; i < n; i++)
        d[i] = a[i] + b[i];
    }
  }
}
```

任务分配结构体：single

- 令某一段代码仅被并行区中的一个线程执行
- 可用于执行非线程安全的操作（例如 I/O）
- 不执行此代码段的线程将等待此代码段执行结束（除非用了 **nowait** 子句）

```
#pragma omp parallel num_threads(10)
{
#pragma omp single
  {
    scanf("%d", &input);
  }
  printf("input is %d", input);
}
```

数据共享与线程同步

数据共享与线程同步

—数据共享

数据共享

- 理解数据的作用域十分重要
  - OpenMP 属于共享内存编程模型，大多数变量默认是共享的
- 默认线程间共享变量：
  - 全局变量、static 变量
  - 除了并行循环的循环下标外，所有并行区内、任务分配结构体外局部变量
- 默认线程内私有变量：
  - 并行循环的循环下标
  - 任务分配结构体内的局部变量
  - 并行区中调用的函数中的局部变量
- 作用域可以通过子句显式控制
  - **private**, **shared**, **firstprivate**, **lastprivate**
  - **default**, **reduction**, **copyin**

数据共享

- **private(var_list)**
  - 将列表中的变量声明为线程私有
  - 变量在进入并行区时不会初始化，离开并行区后不会保留
- **shared(var_list)**
  - 将列表中的变量声明为线程间共享
- **firstprivate(var_list)**
  - 与 **private** 类似，但变量在进入并行区时会被初始化成进入并行区前的值
- **lastprivate(var_list)**
  - 与 **private** 类似，但变量在离开并行区后，其在最后一次迭代时的值会被保留

数据共享

- **private** 子句举例

```
void work(flaot *c, int N) {
  float x, y;
#pragma omp parallel for private(x, y)
  for (int i = 0; i < N; i++) {
    x = a[i];
    y = b[i];
    c[i] = x + y;
  }
}
```

数据共享

- 点乘
- 有什么错误？

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for shared(sum)
  for (int i = 0; i < N; i++) {
    sum = sum + a[i] * b[i];
  }
  return sum;
}
```

数据竞争 Data race

数据共享与线程同步

—线程同步

线程同步

- 对存在数据竞争（data race）的变量访问需要保护
  - 同时只有1个线程对共享变量进行修改

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for shared(sum)
  for (int i = 0; i < N; i++) {
#pragma omp critical
    sum += a[i] * b[i];
  }
  return sum;
}
```

线程同步

线程同步指导语句：

- **#pragma omp critical**
  - 只有一个线程执行临界区

```
#pragma omp critical
{ /** 临界区代码*/ }
```

- **#pragma omp barrier**
  - 等待所有线程到达此处后继续运行

```
#pragma omp barrier
```

- **#pragma omp single**
  - 任务分配结构体的一种：只用一个线程执行代码
  - 也可以用 **#pragma omp master**：只用主线程执行一段代码
    - 执行完毕后没有隐式 barrier
    - 比 **single** 命令更高效

```
#pragma omp single // 或 master
{ /** 仅被一个线程执行 */ }
```

- **#pragma omp atomic**
  - 使用原子指令访问共享变量

```
#pragma omp atomic
a[x] += b[y]; // 支持原子操作的运算
```

线程同步

- OpenMP 的锁函数：

```
omp_set_lock(l);
/* 临界区代码 */
omp_unset_lock(l);
```

- **void omp_init_lock(omp_lock_t *lock)**
  - 初始化锁
- **void omp_destroy_lock(omp_lock_t *lock)**
  - 销毁锁
- **void omp_set_lock(omp_lock_t *lock)**
  - 获取锁。当锁已被其他线程获取时，则等待其他线程释放锁
- **void omp_unset_lock(omp_lock_t *lock)**
  - 释放锁
- **int omp_test_lock(omp_lock_t *lock)**
  - 测试某个锁是否已被获取。此函数不会阻塞

线程同步

- OpenMP 对共享变量维持“放松的一致性”
  - 各线程中，某一共享变量的值不总是最新的
- 当需要获取一个共享变量的最新值时，程序员须保证变量被 flush
- 下列命令会自动进行 flush：
  - **parallel**（进出）、**critical**（进出)、**ordered**（进出）、**for**（出）、**sections**（出）、**single**（出）
- 手动 flush：

```
#pragma omp flush [内存序] [(变量列表)]
```

内存序：

acq_rel, release, acquire

*详细说明及FORTRAN用法请参见文档

线程同步

- 若干 OpenMP 结构体具有隐式 barrier，例如
  - **parallel**, **for**, **single**
- 若已知不需要 barrier，可通过 **nowait** 子句取消 barrier

```
#pragma omp for nowait
  for (int i = 0; i < n; i++)
    a[i] = bigFunc1(i);
#pragma omp for nowait
  for (int j = 0; j < m; j++)
    b[j] = bigFunc2(j);
```

线程同步

OpenMP 同步方法：

- Barrier
  - 等待所有线程到达
  - 适用于等待一个计算步骤结束以开始下一步骤
- 临界区（底层采用锁实现）
  - 保护共享资源的简便方法
  - 可以多行代码
- 显式的锁操作
  - 保护共享资源
  - 比临界区灵活但繁琐，实现复杂同步操作时可能需要
- 原子操作 Atomic
  - 保护共享资源
  - 轻量级，适用于单个操作：单个语句
- 单线程任务分配结构体
  - 保护共享资源
  - 用于 `omp parallel` 语句块中（`omp for` 之外）

数据共享与线程同步

—规约

归约

- 问题是什么？是否有更高效的方法？

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for shared(sum)
  for (int i = 0; i < N; i++) {
#pragma omp critical
    sum += a[i] * b[i];
  }
  return sum;
}
```

归约

- 使用 **reduction** 子句

```
float dot_prod(float *a, float *b, int N) {
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < N; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

是否想到 MPI_Reduce 函数

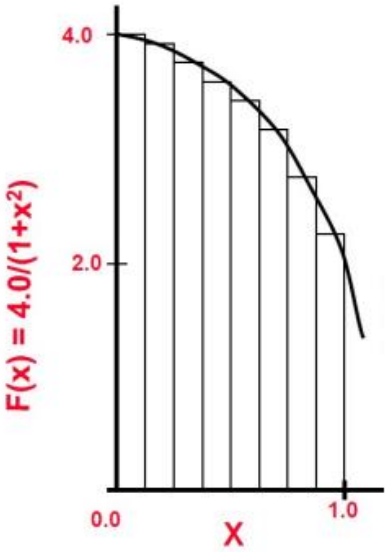
归约

- `reduction(op: list)`
- `list` 中的变量需为共享变量
- 实现原理：
  1. 首先，各线程各自在私有变量上进行局部归约
  2. 任务结束时，局部结果再被跨线程地归约成全局结果

归约

- 性能对比

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




```
static long num_steps = 100000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step; //中点
  sum += 4.0 / (1.0 + x * x); //计算函数值
}
double pi = step * sum; //乘以高度
```

串行伪代码

归约

- 方案1: **reduction**

```
static long num_steps = 100,000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step;
  sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
```

归约

- 方案2：critical

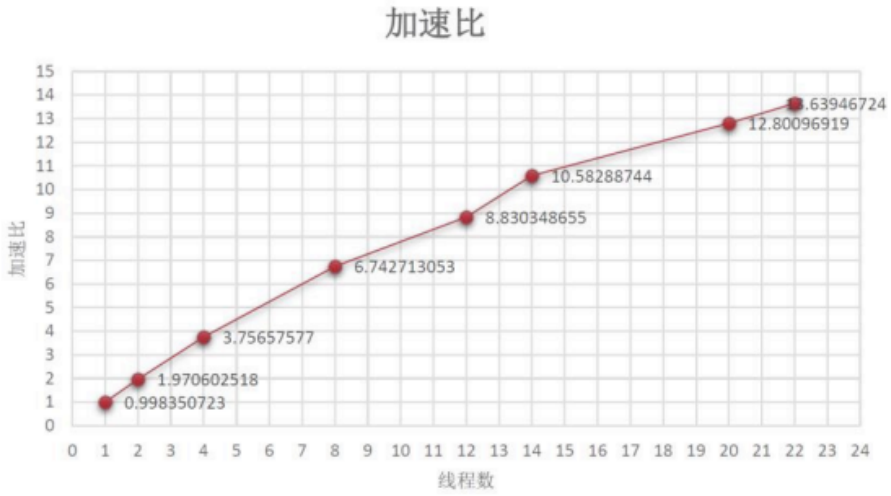
```
static long num_steps = 100,000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
#pragma omp parallel
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step;
#pragma omp critical
  sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
```

归约

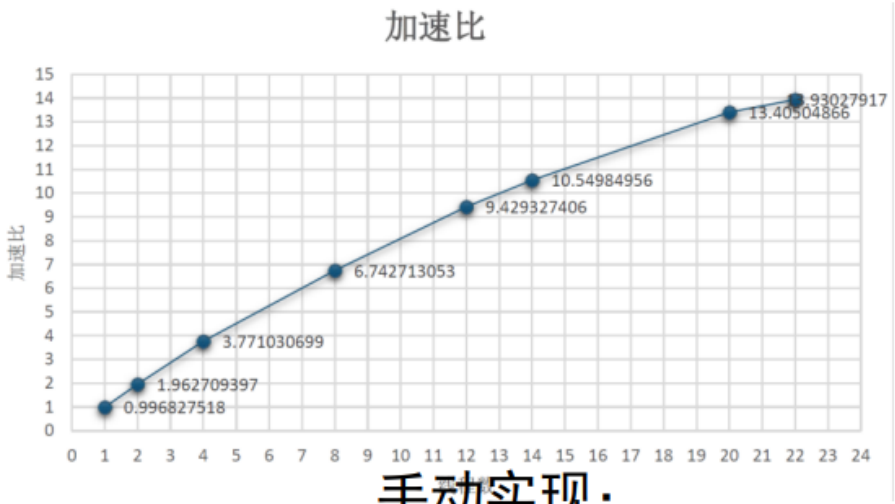
- 方案3： **atomic**

```
static long num_steps = 100,000;
double step = 1.0 / (double)num_steps;
double sum = 0.0;
#pragma omp parallel for
for (int i = 0; i < num_steps; i++) {
  double x = (i + 0.5) * step;
#pragma omp atomic
  sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
```

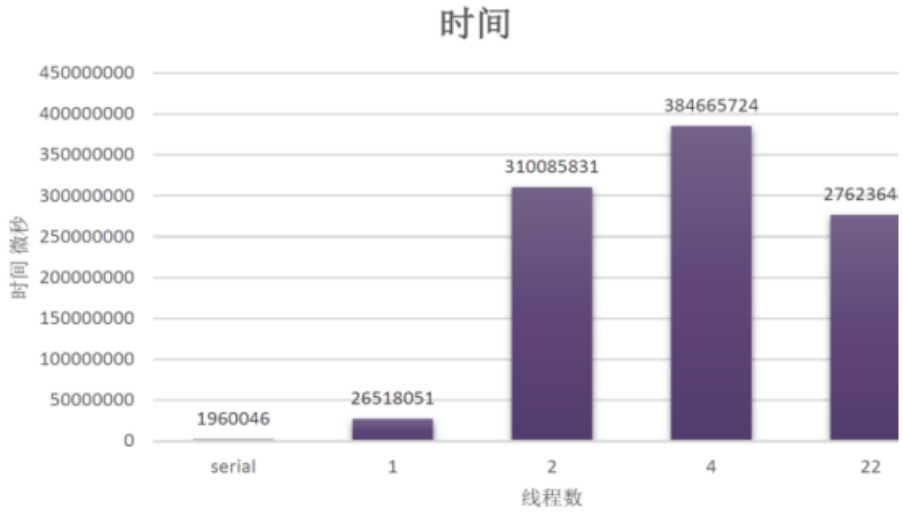
不同归约性能对比



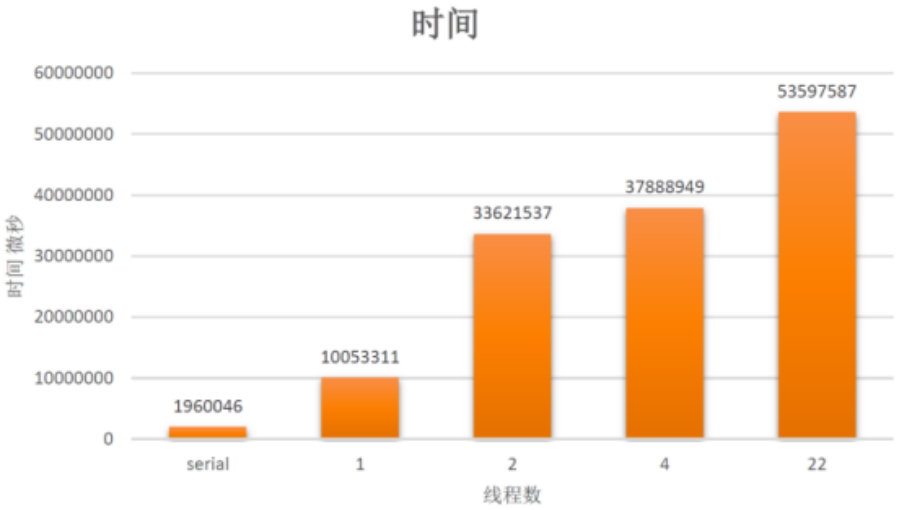
方案1： **reduction**



手动实现：
先求局部和，再串行求全局和



方案2： **critical**



方案3： **atomic**

归约

- 使用 **reduction** 子句性能最优
  - 仅在循环结束后进行同步
  - 与先求局部和，再串行求全局和的手动实现性能相当
- 使用 **critical** 或 **atomic** 使并行性能反而远不如串行
  - 每次迭代均进行同步
  - 同步开销过大
  - 注意：加速比

编译运行命令

OpenMP 示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>  ← 头文件

void Hello(void); /** 线程函数 */

int main(int argc, char* argv[]) {
    /** 从命令行得到线程数 */
    int thread_count = strtol(argv[1], NULL, 10);
    #pragma omp parallel num_threads(thread_count) ← 并行指导语句
    Hello();
    return 0;
} /** main */

void Hello(void) {
    int my_rank = omp_get_thread_num(); ← 函数调用-线程 ID
    int thread_count = omp_get_num_threads(); ← 线程数
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /** Hello */
```

编译运行

```
gcc  -g  -Wall  -fopenmp  -o  main  main.c
```

因编译器不同而不同

```
OMP_NUM_THREADS=4  ./main
```

环境变量设置

可能的输出

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

编译运行

编译器	编译命令	OpenMP 选项
Intel	`icc`	`-qopenmp`

GNU	`gcc`
	`g++`
	`g77`
	`gfortran`
		`-fopenmp`

PGI	`pgcc`
	`pgCC`
	`pgf77`
	`pgf90`
		`-mp`

Clang	`clang` `clang++`	`-fopenmp`

常用的库函数

- `void omp_set_num_threads(int num_threads)`
  - 设置后续并行区的线程数
- `int omp_get_num_threads()`
  - 获取当前设置的线程数
- `int omp_get_thread_num()`
  - 获取当前线程的编号
  - 主线程编号为 0
- `int omp_get_thread_limit()`
  - 获取程序可用的最大线程数
- `int omp_get_num_procs()`
  - 获取程序可用的处理器数量
- `int omp_in_parallel()`
  - 判断当前代码是否在并行区中
- 更多函数请查阅 OpenMP 文档

常用的环境变量

- **OMP_NUM_THREADS**
  - 设置并行区线程数
- **OMP_PROC_BIND**
  - 设置线程与处理器的绑定关系
  - 可设为 **true**（绑定）或 **false**（不绑定）
  - 也可通过 **master**, **close** 和 **spread** 为 NUMA 架构设定更具体的绑定策略（详见文档）
- **OMP_WAIT_POLICY**
  - 设为 **ACTIVE** 表示利用自旋锁进行线程间等待
  - 低延迟，等待需要消耗CPU时间
  - 设为 **PASSIVE** 表示利用操作系统调度进行线程间等待
  - 高延迟，等待不消耗 CPU 时间
- 更多环境变量请查阅 OpenMP 文档

总结

- OpenMP
  - 是一种在共享内存计算机中实现并行的编程方法，尤其适用于并行化循环
  - 采用了 Fork-Join 模型
  - 是基于编译器的编程框架
- 使用 OpenMP 编程注意：
  - 定义并行区（**omp parallel**）
  - 设置并行度
  - 采用合适的并行结构（**for**; **sections**; **single**）
  - 设置调度策略（**schedule**）
  - 数据管理 / 变量分类（ **private** / **shared**; **flush** ）
  - 同步控制（**critical**等）