并行计算
Parallel Computing

主讲 孙经纬

2024年 春季学期

概要

- 第三篇 并行编程
  - 第十三章 并行程序设计基础
  - **第十四章 共享存储系统并行编程**
  - 第十五章 分布存储系统并行编程
  - 补充章节1 GPU并行编程
  - 补充章节2 关于并行编程的更多话题

第十四章 并行程序设计基础

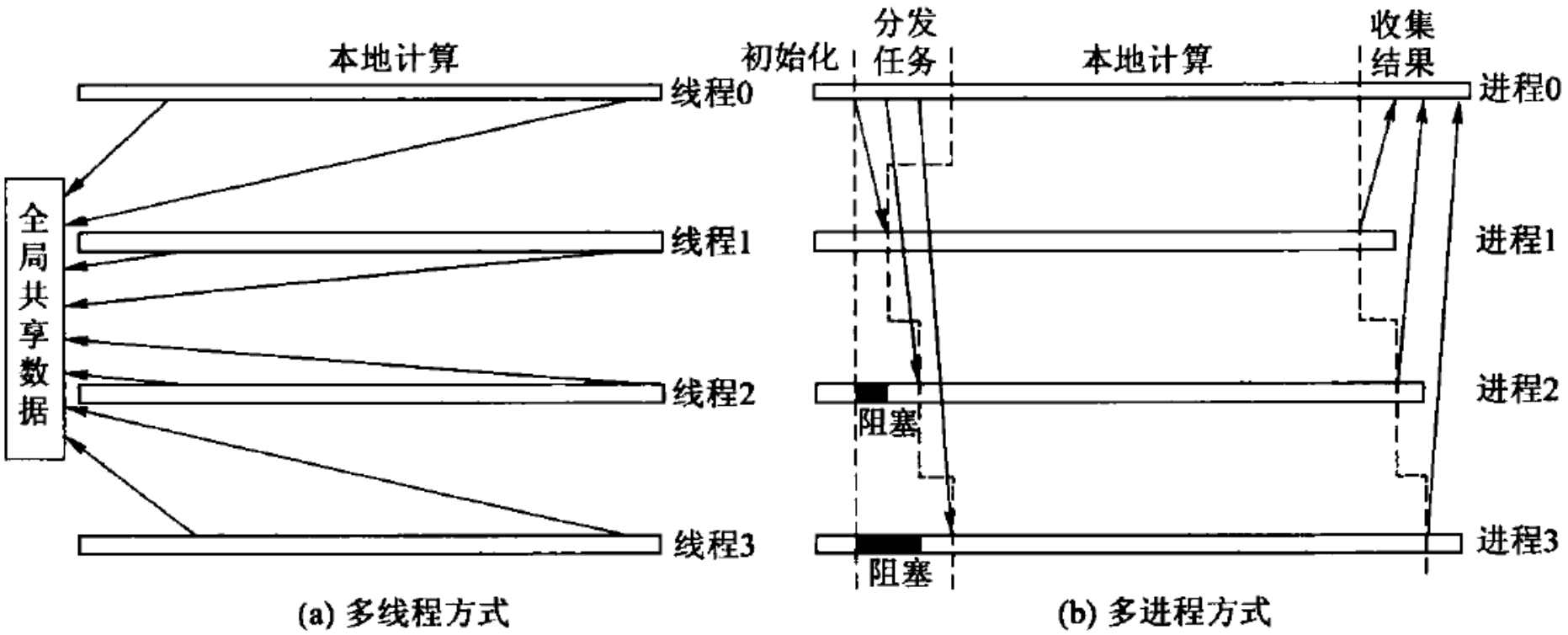
- **14.1 POSIX线程模型**
- 14.2 OpenMP

进程和线程

- 进程（Process）是正在运行的程序，是该程序对应的指令和数据的运行实例。操作系统进行资源分配的基本单位。

- 线程（Thread）是进程中的一个可调度实体，是操作系统进行运行调度的基本单位。也叫轻量级进程。
- 与进程 fork() 相比，线程带来的开销很小。内核无需单独复制进程的内存空间或文件描述符等等。这就节省了大量的CPU 时间。

进程和线程


本地计算
线程0
全局共享数据
线程1
线程2
线程3
(a) 多线程方式
初始化
分发任务
本地计算
收集结果
进程0
进程1
进程2
进程3
阻塞
阻塞
(b) 多进程方式

POSIX线程模型

- Portable Operating System Interface of UNIX
- POSIX线程简称为Pthread
- 提供了在多个操作系统上一致的程序设计接口

POSIX线程模型

- 提供超过100种线程操作函数，以pthread_开头
- 主要包含以下分类
  - Thread management – create, join threads, etc
  - Mutexes
  - Condition variables
  - Locks and barriers
  - Spinlocks

- Pthread Monte Carlo求π示例 calculate_pi.c

```
~/pc_course ❯ gcc -o calculate_pi calculate_pi.c -pthread
~/pc_course ❯ ./calculate_pi
Estimated Pi: 3.140590
~/pc_course ❯
```

记得加上
pthread flag

calculate_pi.c

pc_course > calculate_pi.c > main()

```c
int main() {
    pthread_t threads[NUM_THREADS];
    ThreadData data;
    data.hits = 0;
    data.iterations = NUM_ITERATIONS / NUM_THREADS;
    pthread_mutex_init(&data.mutex, NULL);

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, compute_pi, &data)) {
            fprintf(stderr, "Error creating thread\n");
            return 1;
        }
    }

    // Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], NULL)) {
            fprintf(stderr, "Error joining thread\n");
            return 2;
        }
    }

    // Calculate pi
    double pi_estimate = 4.0 * data.hits / NUM_ITERATIONS;
    printf("Estimated Pi: %f\n", pi_estimate);

    // Clean up
    pthread_mutex_destroy(&data.mutex);

    return 0;
}
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define NUM_THREADS 4
#define NUM_ITERATIONS 10000000
// Shared data structure for the threads
typedef struct {
    int hits;
    int iterations;
    pthread_mutex_t mutex;
} ThreadData;

void *compute_pi(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int hits = 0;
    double x, y;

    unsigned int seed = time(NULL) ^ (pthread_self() << 16);

    for (int i = 0; i < data->iterations; i++) {
        x = (double)rand_r(&seed) / RAND_MAX;
        y = (double)rand_r(&seed) / RAND_MAX;
        if (x * x + y * y <= 1.0) {
            hits++;
        }
    }

    // Lock the mutex before updating the shared hits count
    pthread_mutex_lock(&data->mutex);
    data->hits += hits;
    pthread_mutex_unlock(&data->mutex);

    return NULL;
}
```

第十四章 并行程序设计基础

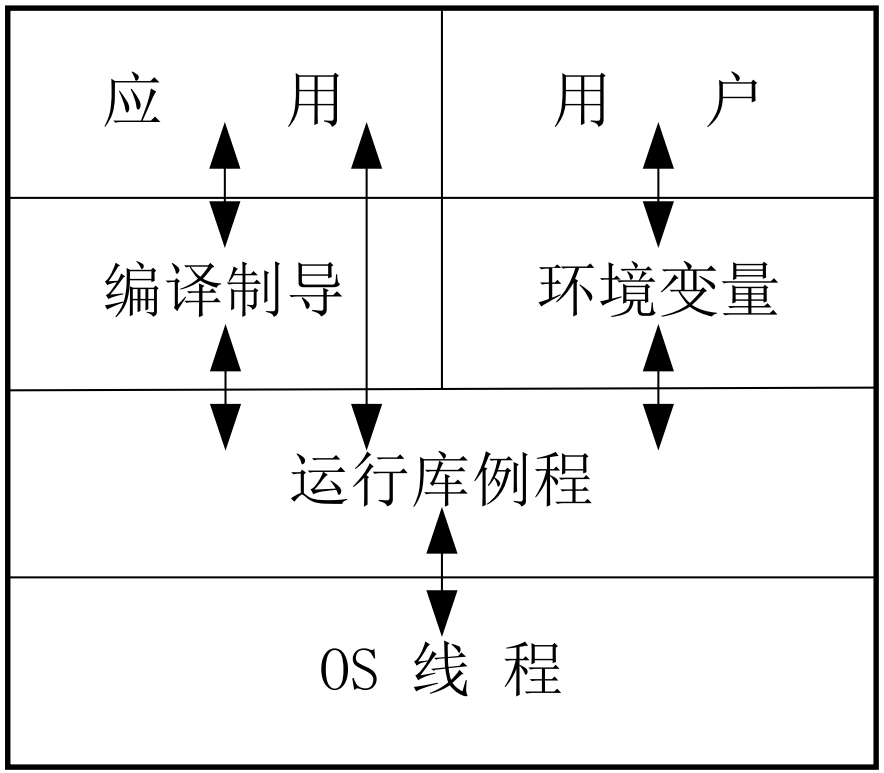
- 14.1 POSIX线程模型
- **14.2 OpenMP**

OpenMP (Open Multi-Processing)

- OpenMP是在共享存储体系结构上的编程模型
- 是C/C++ 和Fortran等的应用编程接口
- 已经被大多数计算机硬件和软件厂商支持

- 由三个基本部分构成:
  - 编译制导(Compiler Directive)
  - 运行库例程(Runtime Library)
  - 环境变量(Environment Variables)

OpenMP



OpenMP

- OpenMP不具备的性质
  - 不是一种编程语言，不能自动进行并行化
  - 不是建立在分布式存储系统上的
  - 不是在所有的环境下都是一样的
  - 不能保证让共享存储器均能有效的利用
  - 不进行错误检查，即使用户给出的编译制导有错误

- OpenMP的目标
  - 标准性
  - 简洁实用
  - 使用方便
  - 可移植性

OpenMP历史

- 1994年，第一个ANSI X3H5草案提出，被否决
- 1997年，OpenMP标准规范代替原先被否决的ANSI X3H5，被人们认可
- 1997年10月公布了与Fortran语言捆绑的第一个标准规范
- 1998年11月9日公布了支持C和C++的标准规范

- 网站：http://www.openmp.org

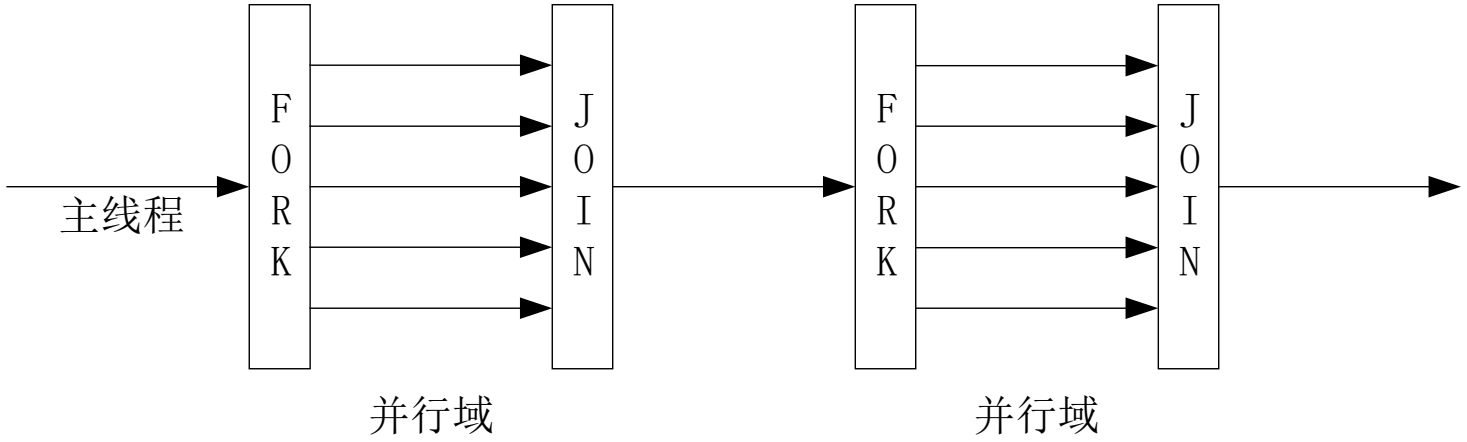
OpenMP历史

- 1997年 OpenMP 1.0
- 2000年 OpenMP 2.0
- 2008年 OpenMP 3.0
- 2013年 OpenMP 4.0
- 2018年 OpenMP 5.0
- 2024年 OpenMP 6.0（预计）

- https://www.openmp.org/specifications/

OpenMP并行编程模型

- 基于线程的并行编程模型(Programming Model)
- OpenMP使用Fork-Join并行执行



OpenMP程序结构

- 基于Fortran语言的OpenMP程序的结构

```
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3
      !Serial code
        …
      !Beginning of parallel section. Fork a team of threads.
      !Specify variable scoping
!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
      !Parallel section executed by all threads
        …
      !All threads join master thread and disband
!$OMP END PARALLEL
      !Resume serial code
        …
END
```

OpenMP程序结构

- 基于c/c++语言的OpenMP程序的结构

```
#include <omp.h>
main (){
        int var1, var2, var3;
        /*Serial code*/
          …
        /*Beginning of parallel section. Fork a team of threads*/
        /*Specify variable scoping */
        #pragma omp parallel private(var1, var2) shared(var3)
        {
                /*Parallel section executed by all threads*/
                …
                /*All threads join master thread and disband*/
        }
        /*Resume serial code */
        …
}
```

一个简单的OpenMP程序实例

- 基于c/c++语言的OpenMP程序的一个具体实现

hello_world_omp.c

pc_course > hello_world_omp.c > main(int, char * [])

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

int main(int argc, char* argv[])
{
    int nthreads, tid;

     /*     Fork a team of threads      */
    #pragma omp parallel private(nthreads, tid)
    {
        /*     Obtain and print thread id  */
        tid = omp_get_thread_num();
        printf("Hello World from OMP thread %d\n", tid);
        /*      Only master thread does this  */
        if (tid==0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads %d\n", nthreads);
        }
    }
    return 0;
}
```

一个简单的OpenMP程序实例

- 运行结果

```
~/pc_course ❯ gcc -o hello_world_omp hello_world_omp.c -fopenmp
~/pc_course ❯ ./hello_world_omp
Hello World from OMP thread 2
Hello World from OMP thread 8
Hello World from OMP thread 6
Hello World from OMP thread 18
Hello World from OMP thread 12
Hello World from OMP thread 13
Hello World from OMP thread 7
Hello World from OMP thread 0
Number of threads 24
Hello World from OMP thread 19
Hello World from OMP thread 3
Hello World from OMP thread 23
Hello World from OMP thread 20
Hello World from OMP thread 4
Hello World from OMP thread 15
Hello World from OMP thread 14
Hello World from OMP thread 11
Hello World from OMP thread 17
Hello World from OMP thread 21
Hello World from OMP thread 1
Hello World from OMP thread 16
Hello World from OMP thread 5
Hello World from OMP thread 22
Hello World from OMP thread 10
Hello World from OMP thread 9
~/pc_course ❯
```

记得加上 fopenmp flag

编译制导 (Compiler Directive)

- 语句格式

所以后面的左大括号“{” 必须换行

#pragma omp	directive-name	[clause, ...]	newline
制导指令前缀。对所有的OpenMP语句都需要这样的前缀。	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。	子句。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。	换行符。表明这条制导语句的终止。

并行域

- 并行域（Parallel Region）：
  - 被多个线程执行的程序代码块
  - OpenMP中的基本并行结构
- 语法格式： #pragma omp parallel [clause[[,]clause]…] newline
  - [clause] =
    - if (scalar_expression)
    - private (list)
    - shared (list)
    - default (shared | none)
    - firstprivate (list)
    - reduction (operator: list)
    - copyin (list)

作用域

- 作用域（Scoping）分类
  - 静态范围（Static Extent）
    - 代码在一个编译制导语句之后，被封装到一个结构块中
    - 这类语句不能跨越多个例程或者代码文件
  - 孤立语句（Orphaned Directive）
    - 这类OpenMP编译制导语句不依赖于其它的语句
    - 存在于其他静态范围语句之外
    - 可跨越多个例程或者代码文件
  - 动态范围（Dynamic Extent）
    - 包括静态范围和孤立语句

https://hpc-tutorials.llnl.gov/openmp/directive_scoping/

作用域

动态范围	
静态范围 for语句出现在一个封闭的并行域中	孤立语句 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
	…
	}

作用域

- 孤立语句示例

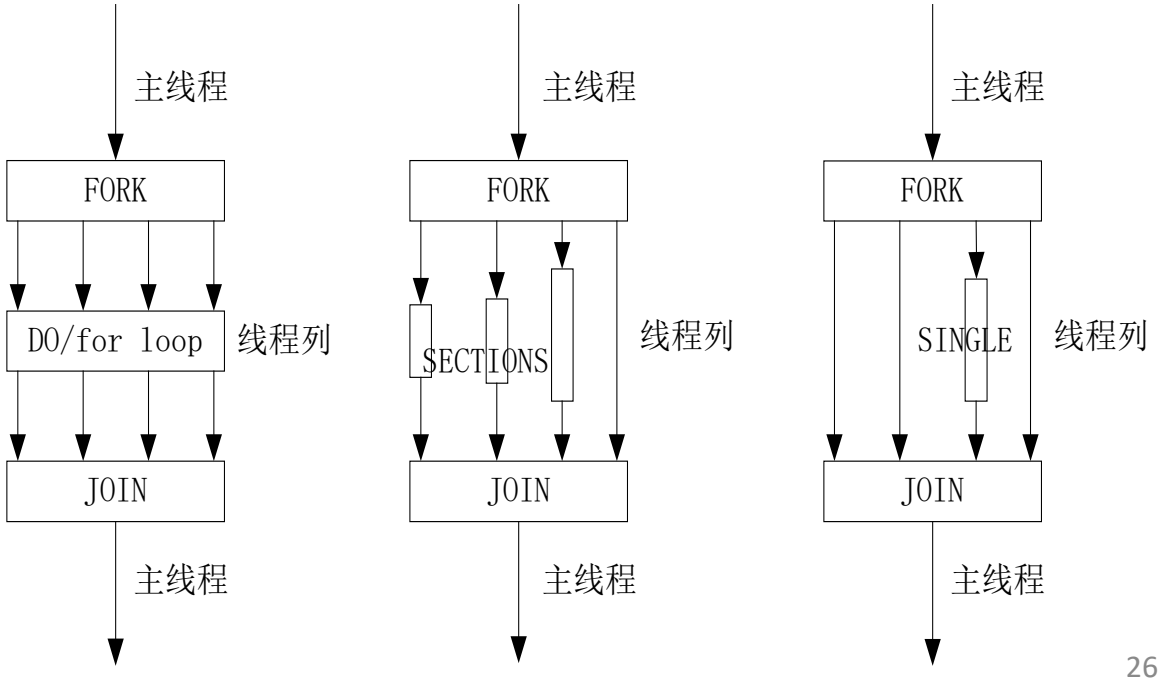
orphaned_omp.cpp

pc_course > orphaned_omp.cpp > ...

```cpp
#include <omp.h>
#include <iostream>
#include <vector>

// process函数实现，其中包含孤立的OpenMP指令
void process(std::vector<double>& vec) {
    // 孤立的OpenMP工作共享指令
    #pragma omp for
    for (size_t i = 0; i < vec.size(); ++i) {
        vec[i] *= 2.0; // 将向量的每个元素乘以2
    }
}

int main() {
    const int n = 1000000;
    std::vector<double> vec(n, 1.0); // 创建一个大小为n并初始化为1.0的向量

    // 开始并行区域
    #pragma omp parallel
    {
        process(vec); // 在并行区域内调用process函数
    }
    // 并行区域结束

    // 输出结果的一部分进行检验
    std::cout << "First elements: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << vec[i] << " ";
    }
    std::cout << "\n";

    return 0;
}
```

共享任务结构

- 共享任务结构将代码划分给线程组的各成员执行
  - 并行do/for循环 —— 适合实现SPMD程序
  - 并行sections —— 适合实现MPMD程序
  - 串行single



for编译制导语句

- for语句指定紧随它的循环语句必须由线程组并行执行
- OpenMP最常见的用法，简单方便
- 语句格式
  - #pragma omp for [clause[[,]clause]…] newline
  - [clause]=
    - Schedule(type [,chunk])
    - ordered
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - shared (list)
    - reduction (operator: list)
    - nowait

for编译制导语句

- schedule子句描述如何将循环迭代划分给线程组中的线程
  - type为static，循环被分成大小为 chunk的块，静态分配

```
#pragma omp parallel for schedule(static, chunkSize)
for (int i = 0; i < n; i++) {
    // Loop body
}
```

  - type为dynamic,循环被动态划分为大小为chunk的块，动态分配

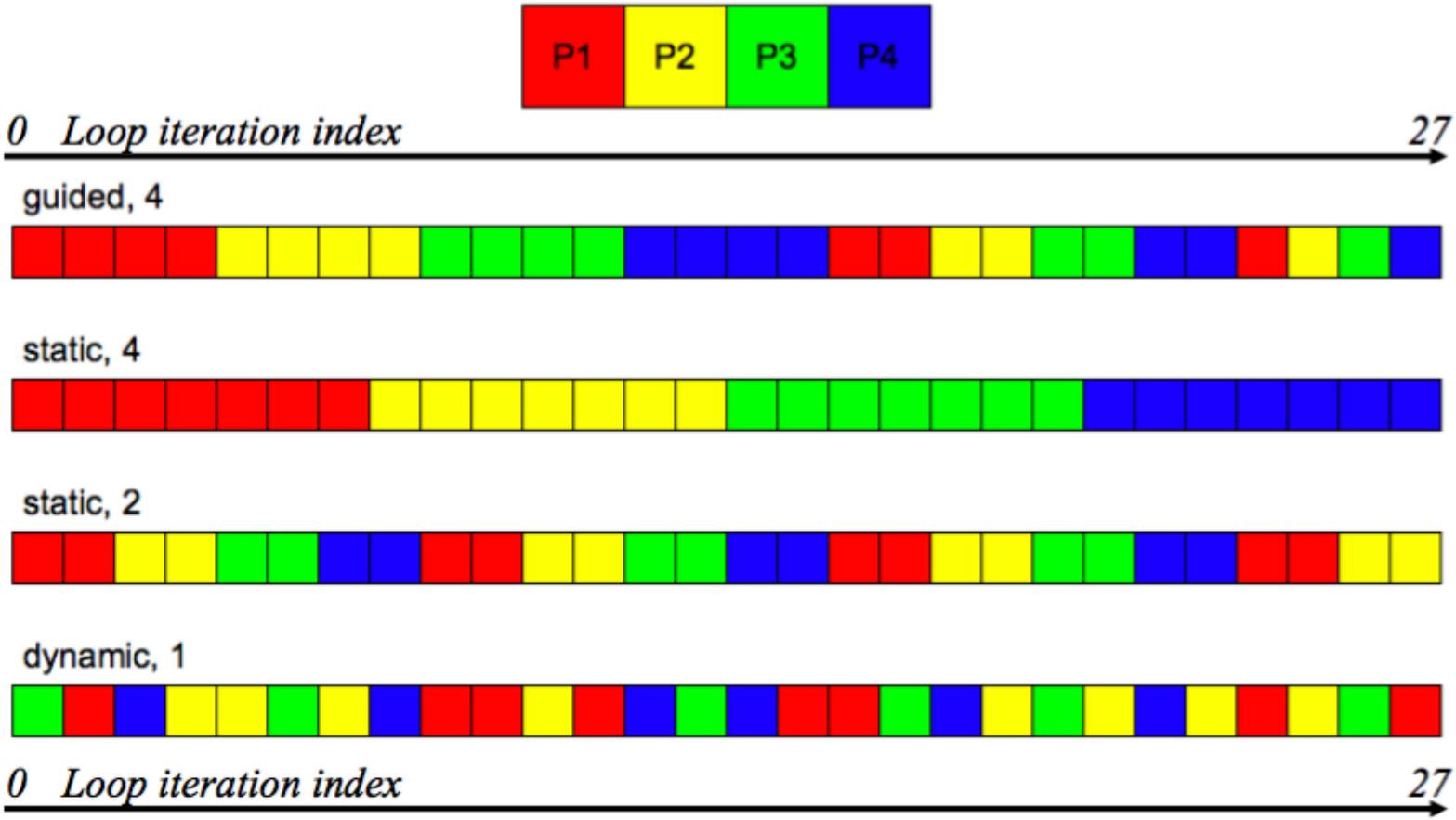
```
#pragma omp parallel for schedule(dynamic, chunkSize)
for (int i = 0; i < n; i++) {
    // Loop body
}
```

  - 如果没有指定chunk大小，默认值为1

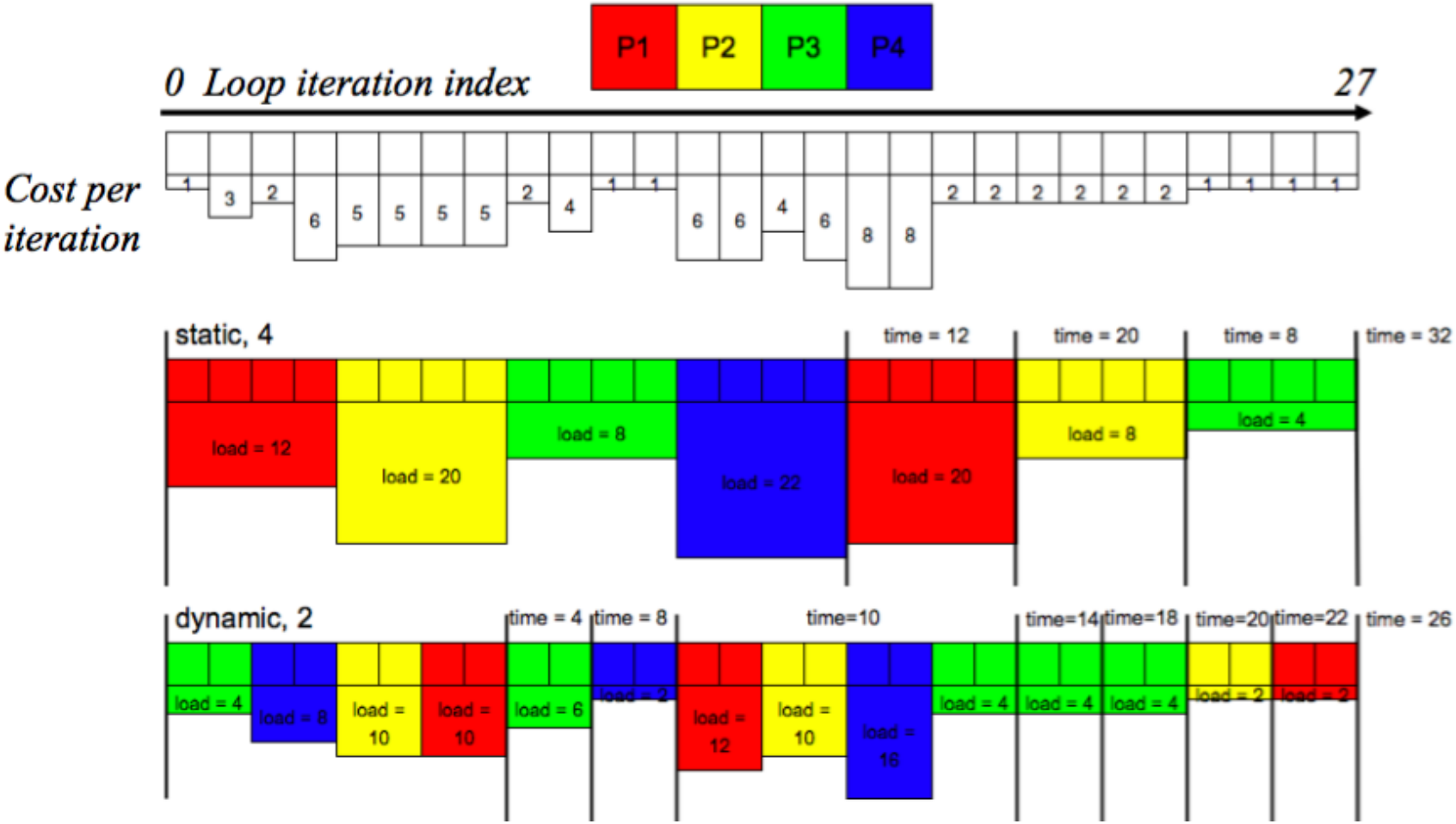
for编译制导语句

- schedule type还可以是guided, auto, runtime
  - runtime并不是一种调度方法，而是让OMP_SCHEDULE环境变量决定调度方法
  - auto也不是一种调度方法，而是让编译器决定
  - guided也是一种动态调度，但是块大小会逐渐减小

for编译制导语句


P1
P2
P3
P4
0 Loop iteration index
27
guided, 4
static, 4
static, 2
dynamic, 1
0 Loop iteration index
27

for编译制导语句


P1
P2
P3
P4
0 Loop iteration index
27
Cost per iteration
1 3 2 6 5 5 5 5 2 4 1 1 6 6 4 6 8 8 2 2 2 2 2 2 1 1 1 1
static, 4
time = 12
time = 20
time = 8
time = 32
load = 12
load = 20
load = 8
load = 22
load = 20
load = 8
load = 4
dynamic, 2
time = 4
time = 8
time=10
time=14
time=18
time=20
time=22
time = 26
load = 4
load = 8
load = 10
load = 10
load = 6
load = 2
load = 12
load = 10
load = 16
load = 4
load = 4
load = 4
load = 2
load = 2

schedule_omp.cpp

pc_course > schedule_omp.cpp > ...

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

void print(int n_thread, std::vector<std::vector<int>> &task){
    for(int i=0;i<n_thread;i++){
        printf("Thread %d:", i);
        for(int j=0;j<task[i].size();j++)
            printf(" %d", task[i][j]);
        printf("\n");
    }
}
//OpenMP static和dynamic调度效果示例
int main() {
    int n_thread = 4;
    int n_task = 10;
    omp_set_num_threads(n_thread);
    std::vector<std::vector<int>> task(n_thread); // 记录每个线程分配到的任务

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        #pragma omp for
        for (int i = 0; i < n_task; ++i) {
            task[tid].push_back(i);
        }
    }
    print(n_thread, task); //输出每个线程分配到的任务
    return 0;
}
```

```cpp
//OpenMP static和dynamic调度效果示例
int main() {
    int n_thread = 4;
    int n_task = 10;
    omp_set_num_threads(n_thread);
    std::vector<std::vector<int>> task(n_thread); // 记录每个线程分配到的任务

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        #pragma omp for
        for (int i = 0; i < n_task; ++i) {
            task[tid].push_back(i);
        }
    }
    print(n_thread, task); //输出每个线程分配到的任务
    return 0;
}
```

啥都不写，默认schedule

OpenMP标准并未规定默认schedule，可以由编译器自行决定

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
~/pc_course ❯ g++ -o schedule_omp schedule_omp.cpp -fopenmp
~/pc_course ❯ ./schedule_omp
Thread 0: 0 1 2
Thread 1: 3 4 5
Thread 2: 6 7
Thread 3: 8 9
~/pc_course ❯
```

连续分块划分 (gcc 9.4.0)

```cpp
//OpenMP static和dynamic调度效果示例
int main() {
    int n_thread = 4;
    int n_task = 10;
    omp_set_num_threads(n_thread);
    std::vector<std::vector<int>> task(n_thread); // 记录每个线程分配到的任务

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        #pragma omp for schedule(static, 1)
        for (int i = 0; i < n_task; ++i) {
            task[tid].push_back(i);
        }
    }
    print(n_thread, task); //输出每个线程分配到的任务
    return 0;
}
```

静态调度，块大小为1

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
~/pc_course ❯ g++ -o schedule_omp schedule_omp.cpp -fopenmp
~/pc_course ❯ ./schedule_omp
Thread 0: 0 4 8
Thread 1: 1 5 9
Thread 2: 2 6
Thread 3: 3 7
~/pc_course ❯
```

按大小为1的块循环分配

```cpp
//OpenMP static和dynamic调度效果示例
int main() {
    int n_thread = 4;
    int n_task = 10;
    omp_set_num_threads(n_thread);
    std::vector<std::vector<int>> task(n_thread); // 记录每个线程分配到的任务

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        #pragma omp for schedule(static, 2)
        for (int i = 0; i < n_task; ++i) {
            task[tid].push_back(i);
        }
    }
    print(n_thread, task); //输出每个线程分配到的任务
    return 0;
}
```

静态调度，块大小为2

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
~/pc_course ❯ g++ -o schedule_omp schedule_omp.cpp -fopenmp
~/pc_course ❯ ./schedule_omp
Thread 0: 0 1 8 9
Thread 1: 2 3
Thread 2: 4 5
Thread 3: 6 7
~/pc_course ❯
```

按大小为2的块循环分配

```cpp
//OpenMP static和dynamic调度效果示例
int main() {
    int n_thread = 4;
    int n_task = 10;
    omp_set_num_threads(n_thread);
    std::vector<std::vector<int>> task(n_thread); // 记录每个线程分配到的任务

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        #pragma omp for schedule(dynamic, 2)
        for (int i = 0; i < n_task; ++i) {
            task[tid].push_back(i);
        }
    }
    print(n_thread, task); //输出每个线程分配到的任务
    return 0;
}
```

动态调度，块大小为2

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
~/pc_course ❯ g++ -o schedule_omp schedule_omp.cpp -fopenmp
~/pc_course ❯ ./schedule_omp
Thread 0: 6 7 8 9
Thread 1: 2 3
Thread 2: 4 5
Thread 3: 0 1
~/pc_course ❯
```

分配顺序具有随机性

sections编译制导语句

- sections语句将代码划分给线程组中的各个线程
- 不同的section由不同的线程执行
- sections语句格式：

```
#pragma omp sections [ clause[[,]clause]…] newline
{
[#pragma omp section newline]
        …
[#pragma omp section newline]
        …
}
```

sections编译制导语句

```
1   #include <omp.h>
2   #define N     1000
3   int main (){
4       int i;
5       float a[N], b[N], c[N];
6       /* Some initializations */
7       for (i=0; i < N; i++)
8           a[i] = b[i] = i * 1.0;
9       #pragma omp parallel shared(a,b,c) private(i)
10          {
11          #pragma omp sections nowait
12          {
13          #pragma omp section
14          for (i=0; i < N/2; i++)
15              c[i] = a[i] + b[i];
16          #pragma omp section
17          for (i=N/2; i < N; i++)
18              c[i] = a[i] + b[i];
19          }  /* end of sections */
20          } /* end of parallel section */
21  }
```

single编译制导语句

- single语句指定代码由线程组中的一个线程执行
- 线程组中没有执行single语句的线程会一直等待代码块的结束，使用nowait子句除外
- 语句格式：
  - #pragma omp single [clause[[,]clause]…] newline
  - [clause] =
    - private(list)
    - firstprivate(list)
    - nowait

组合的并行共享任务结构

- parallel for编译制导语句
- parallel sections编译制导语句

主要是为了代码编写简洁方便

parallel for编译制导语句

- parallel for语句表明一个并行域包含一个独立的for
- 语句格式
  - #pragma omp parallel for [clause…] newline
  - [clause] =
    - if (scalar_logical_expression)
    - default (shared | none)
    - schedule (type [,chunk])
    - shared (list)
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - reduction (operator: list)
    - copyin (list)

parallel for编译制导语句

```
#include <omp.h>
#define N       1000
#define CHUNKSIZE   100
int main () {
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
            a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel for \
            shared(a,b,c,chunk) private(i) \
            schedule(static,chunk)
            for (i=0; i < n; i++)
                    c[i] = a[i] + b[i];
}
```

parallel for编译制导语句

- parallel sections语句表明一个并行域包含单独的一个sections语句
- 语句格式
  - #pragma omp parallel for [clause…] newline
  - [clause] =
    - if (scalar_logical_expression)
    - default (shared | none)
    - schedule (type [,chunk])
    - shared (list)
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - reduction (operator: list)
    - copyin (list)

同步结构

- master 制导语句
- critical制导语句
- barrier制导语句
- atomic制导语句
- flush制导语句
- ordered制导语句

同步结构

- master制导语句指定代码段只有主线程执行
- 语句格式
  - #pragma omp master newline

同步结构

- critical制导语句表明域中的代码一次只能执行一个线程，其他线程被阻塞在临界区
- 语句格式:
  - #pragma omp critical [name] newline

同步结构

```
#include <omp.h>
main()
{
  int x;
  x = 0;
  #pragma omp parallel shared(x)
  {
      #pragma omp critical
            x = x + 1;
  }  /* end of parallel section */
}
```

注释掉这句话，会发生什么？

同步结构

- barrier语句用来同步一个线程组中所有的线程
- 先到达的线程在此阻塞，等待其他线程
- barrier语句最小代码必须是一个结构化的块
- 语句格式
  - #pragma omp barrier newline

同步结构

- barrier正确与错误使用比较

错误	正确

if (x == 0)
#pragma omp barrier
	if (x == 0)
	{
	#pragma omp barrier
	}

同步结构

- atomic语句指定特定的存储单元将被原子更新
- 语句格式
  - #pragma omp atomic newline
- atomic使用的格式：x binop = expr
  - x是一个标量
  - expr是一个不含对x引用的标量表达式，且不被重载
  - binop是+, *, -, /, &, ^, |, >>, <<之一，且不被重载

同步结构

- atomic示例

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

int main() {
    int counter = 0;

    #pragma omp parallel num_threads(4)
    {
        // Each thread will attempt to increment the counter 1000 times.
        for (int i = 0; i < 1000; ++i) {
            #pragma omp atomic
            counter++; // Atomic increment of the shared counter variable
        }
    }

    printf("Counter should be 4000, and it is: %d\n", counter);

    return 0;
}
```

同步结构

- flush制导语句用以标识一个同步点，用以确保所有的线程看到一致的存储器视图
- 语句格式
  - #pragma omp flush (list) newline
- 下面几种情形下隐含flush运行，除非有nowait

barrier
critical:进入与退出部分
ordered:进入与退出部分
parallel:退出部分
for:退出部分
sections:退出部分
single:退出部分

同步结构

- 为什么需要flush?

OpenMP operates under a **relaxed-consistency memory model**. In this model, threads can have a temporary, thread-local view of memory. This means that without synchronization, changes made by one thread to shared data might not be immediately visible to other threads due to these optimizations and caching mechanisms.

- 由于许多常用的omp语句自带flush，因此编程实践中较少需要手动flush

同步结构

- ordered制导语句指出其所包含循环的执行
- 任何时候只能有一个线程执行ordered所限定部分
  - 相当于将该部分代码串行化
- 只能出现在for或者parallel for语句的动态范围中
- 语句格式:
  - #pragma omp ordered newline

ordered, master, single对比

```
#pragma omp parallel for ordered
for (int i = 0; i < N; i++) {
    // Perform some parallel computation here
    #pragma omp ordered
    {
        // This block is executed in the order of the loop iterations
    } ordered结束后不会有隐含同步
}
```

这里的ordered表示域内可能会有代码需要ordered

```
#pragma omp parallel
{
    // Code here is executed by all threads.
    #pragma omp master
    {
        // Only the master thread executes this block
    }
    // Code here is also executed by all threads, without waiting for the master.
}
```

```
#pragma omp single nowait
{
    // Code here is executed by one thread, and other threads do not wait.
}
```

threadprivate编译制导语句

- threadprivate语句使一个全局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
- 语句格式:
  - #pragma omp threadprivate (list) newline

threadprivate编译制导语句

```
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
int main ()
{
    /* First parallel region */
    #pragma omp parallel private(i,beta)
    for (i=0; i < 10; i++)
            alpha[i] = beta[i] = i;
    /* Second parallel region */
    #pragma omp parallel
            printf("alpha[3]= %d and beta[3]=%d\n",alpha[3],beta[3]);
}
```

数据域属性子句

- 变量作用域范围

  默认情况下，并行域外声明的变量是共享的，并行域内声明的变量是私有的

- 数据域属性子句
  - private子句
  - shared子句
  - default子句
  - firstprivate子句
  - lastprivate子句
  - copyin子句
  - reduction子句

private子句

- private子句表示它列出的变量对于每个线程是局部的
- 语句格式
  - private(list)
- private和threadprivate区别

	private	threadprivate
数据类型	变量	变量
声明的位置	在域的开始或共享任务单元	在块或整个文件区域的例程的定义上
是否持久	否	是
范围	只是词法的- 除非作为子程序的参数而传递	动态范围
初始化	使用 firstprivate	使用 copyin

shared子句

- shared子句表示它所列出的变量被线程组中所有的线程共享
- 所有线程都能对它进行读写访问
- 正确性由程序员保证
- 语句格式
  - shared (list)

default子句

- default子句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
- 语句格式
  - default (shared | none)

firstprivate子句

- firstprivate子句是private子句的超集
- 对变量做原子初始化
- 语句格式:
  - firstprivate (list)

lastprivate子句

- lastprivate子句是private子句的超集
- 将变量从最后的循环迭代或section复制给原始的变量
- 语句格式
  - lastprivate (list)

copyin子句

- copyin子句用来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
- 语句格式
  - copyin(list)

reduction子句

- reduction子句使用指定的操作对其列表中出现的变量进行规约
- 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
- 语句格式
  - reduction (operator: list)
- list中的变量必须为标量，且必须为shared类型

reduction子句

```
//示例：求向量内积
#include <omp.h>
int main ()
{
    int   i, n, chunk;
    float a[100], b[100], result;
    /* Some initializations */
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
            a[i] = i * 1.0;
            b[i] = i * 2.0;
    }
    #pragma omp parallel for default(shared) private(i)\
            schedule(static,chunk) reduction(+:result)
    for (i=0; i < n; i++)
            result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
}
```

reduction将result先作为私有变量，for结束时求和写入共享变量

reduction子句

- Reduction子句的格式

---

x=x op expr
x = expr op x (except subtraction)
x binop = expr
x++
++x
x--
--x

---

x是一个标量
expr是一个不含对x引用的标量表达式，且不被重载
binop是+,*,-,/,&,^,|之一，且不被重载
op是+,*,-,/,&,^,|,&&,or||之一，且不被重载

子句/编译制导语句小结

需要注意的是，具体的某种OpenMP实现不一定严格遵循标准

子句	编译制导					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	√				√	√
PRIVATE	√	√	√	√	√	√
SHARED	√	√			√	√
DEFAULT	√				√	√
FIRSTPRIVATE	√	√	√	√	√	√
LASTPRIVATE		√	√		√	√
REDUCTION	√	√	√		√	√
COPYIN	√				√	√
SCHEDULE		√			√	
ORDERED		√			√	
NOWAIT		√	√	√		

语句绑定和嵌套规则

- 语句绑定
  - 语句DO/ FOR 、SECTIONS、SINGLE、MASTER和BARRIER绑定到动态的封装PARALLEL中，如果没有并行域执行，这些语句是无效的;
  - 语句ORDERED指令绑定到动态DO/ FOR封装中;
  - 语句ATOMIC使得ATOMIC语句在所有的线程中独立存取，而并不只是当前的线程;
  - 语句CRITICAL在所有线程有关CRITICAL指令中独立存取，而不是只对当前的线程;
  - 在PARALLEL封装外，一个语句并不绑定到其它的语句中。

绑定和嵌套规则参考：https://hpc-tutorials.llnl.gov/openmp/directive_binding_and_nesting/

语句绑定和嵌套规则

- **语句嵌套**
  - PARALALL 语句动态地嵌套到其它地语句中，从而逻辑地建立了一个新队列，但这个队列若没有嵌套地并行域执行，则只包含当前线程;
  - DO/ FOR 、SECTION和SINGLE语句绑定到同一个PARALLEL 中，则它们是不允许互相嵌套的;
  - DO/FOR、SECTION和SINGLE语句不允许在动态的扩展CRITICAL、ORDERED和MASTER域中;
  - CRITICAL语句不允许互相嵌套;
  - BARRIER语句不允许在动态的扩展DO/ FOR、ORDERED、SECTIONS、SINGLE、MASTER和CRITICAL域中;
  - MASTER语句不允许在动态扩展DO/ FOR、SECTIONS和SINGLE语句中;
  - ORDERED语句不允许在动态的扩展CRITICAL域中;
  - 任何能允许执行到PARALLEL 域中的指令，在并行域外执行也是合法的。当执行到用户指定的并行域外时，语句执行只与主线程有关。

运行库例程与环境变量

- 运行库例程
  - OpenMP标准定义了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”

- 环境变量
  - OMP_SCHEDULE：只能用到for, parallel for中。它的值就是处理器中循环的次数
  - OMP_NUM_THREADS：定义执行中最大的线程数
  - OMP_DYNAMIC：通过设定变量值TRUE或FALSE,来确定是否动态设定并行域执行的线程数
  - OMP_NESTED：确定是否可以并行嵌套

OpenMP计算实例

- C语言写的串行程序：数值积分求π

```
/* Seriel Code */
static long num_steps = 100000;
double step;
void main () {
      int i; double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
      }
      pi = step * sum;
}
```

OpenMP计算实例

- **使用并行域并行化的程序**

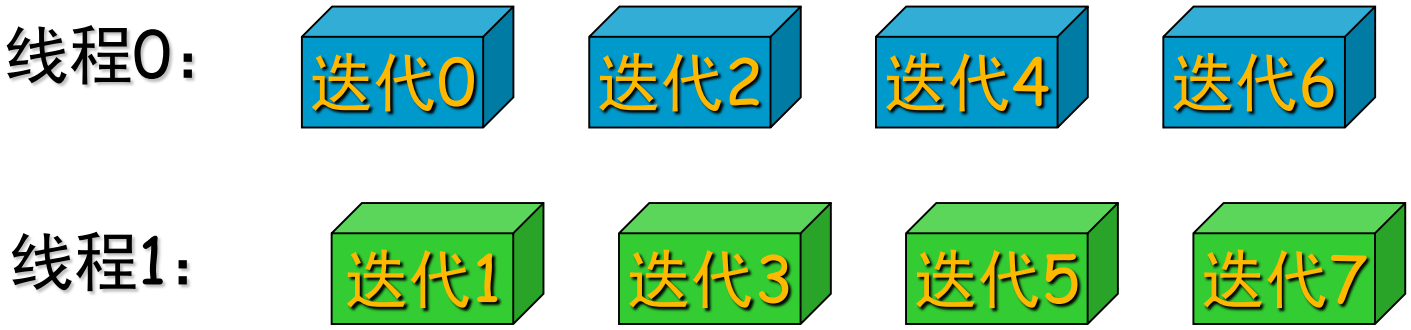
```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main () {
        int i;
        double x, pi, sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
       #pragma omp parallel
       {
                double x;
                int id;
                id = omp_get_thread_num();
                for (i=id, sum[id]=0.0;i< num_steps; i=i+NUM_THREADS){
                        x = (i+0.5)*step;
                        sum[id] += 4.0/(1.0+x*x);
                }
       }
        for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

没有使用omp for，为什么？

这里已经退出并行域，串行执行

OpenMP计算实例

- 使用并行域并行化的程序
- 假设有2个线程参加计算



OpenMP计算实例

- **使用共享任务结构并行化的程序**

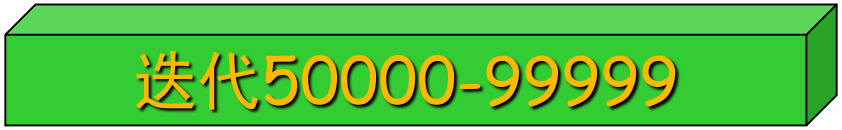
```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main () {
        int i;  double x, pi, sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel
        {
                double x;
                int id;
                id = omp_get_thread_num();
                sum[id] = 0;
                #pragma omp for
                for (i=0;i< num_steps; i++){
                                x = (i+0.5)*step;
                                sum[id] += 4.0/(1.0+x*x);
                }
        }
       for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

OpenMP计算实例

- 使用共享任务结构并行化的程序
- 假设有2个线程参加计算

线程0：

线程1：



OpenMP计算实例

- **使用private子句和critical部分并行化的程序**

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    int i;  double x, sum, pi=0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private (x, sum)
    {
        id = omp_get_thread_num();
        for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
            pi += sum*step
    }
}
```

OpenMP计算实例

- **使用并行归约得出的并行程序**

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main () {
        int i; double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel for reduction(+:sum) private(x)
        for (i=0;i<num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```