

An Introduction to Parallel Programming

Peter Pacheco

Chapter 5

用OpenMP进行共享内存编程





Roadmap

- 学习**OpenMP**的基础知识。
- 学习如何使用**OpenMP**编写程序，以及如何编译和运行**OpenMP**程序。
- 了解**OpenMp**强大的功能
 - 只需要对源程序进行少量改动就可以并行化许多串行的**for**循环。
- 了解**OpenMP**的一些特征：
 - 任务并行化。
 - 显式线程同步。
- 了解在共享内存中编程中的标准问题
 - 缓存对共享内存编程的影响
 - 当串行代码被一个共享内存程序使用时遇到的问题。



5.1 OpenMP编程简介

- 诞生于1997年，目前已经推出OpenMP 3.0版本。
- 标准版本3.0，2008年5月，支持Fortran/C/C++。
- 一种编译指导语句，能够显式指导多线程、共享内存并行的应用程序编程接口（API）
- 具有良好的可移植性，支持多种编程语言。
- 支持多种平台
- 大多数的类UNIX系统以及Windows NT系统（Windows 2000，Windows XP，Windows Vista等）。



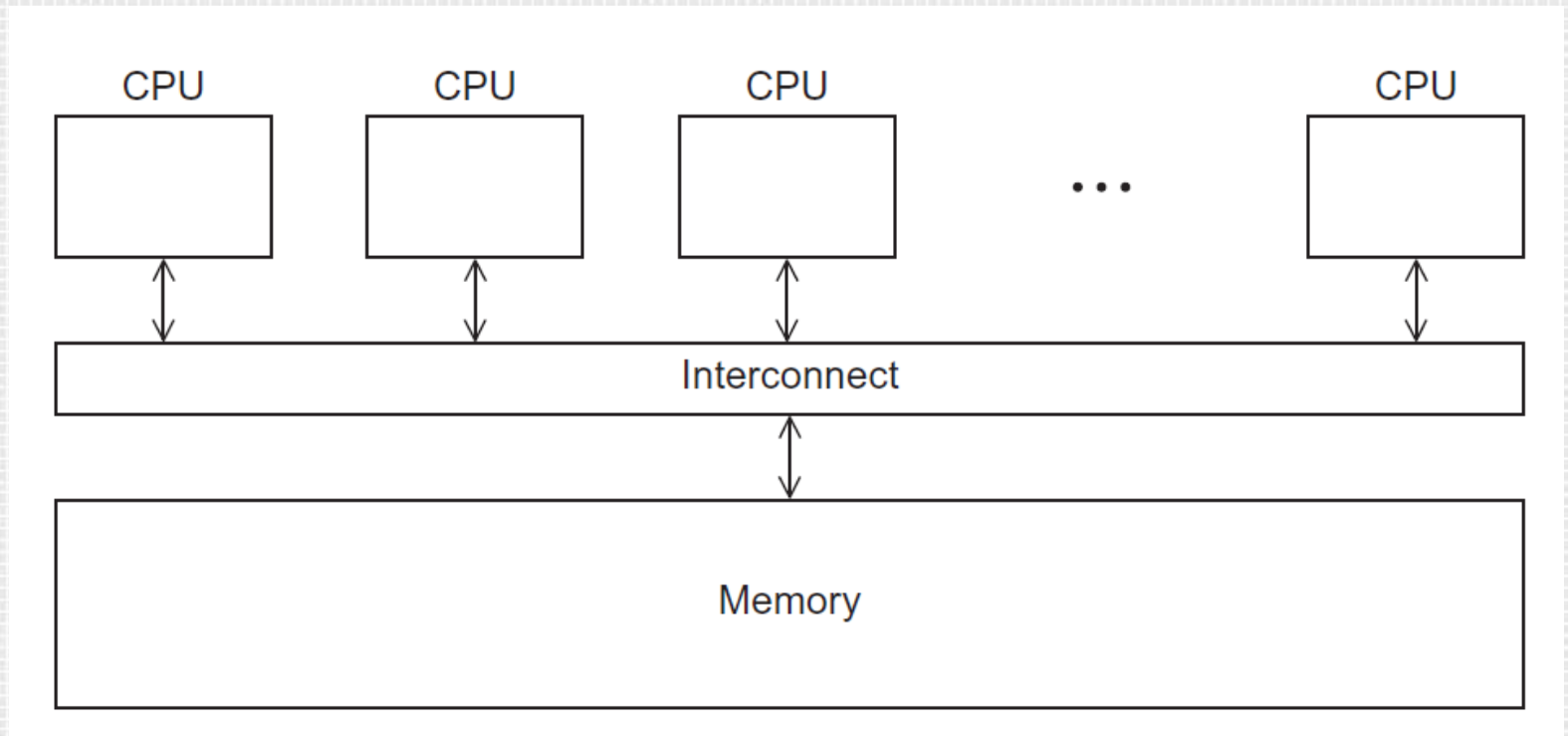
OpenMP

- **OpenMP**是一个针对共享内存并行编程的API。
- **MP = 多处理 (multiprocessing)**
- 在系统中每个线程或进程都有可能访问所有可访问的内存区域。
- 当使用**OpenMP**编程时，我们将系统看做一组核或**CPU**的集合，他们都能访问主存。



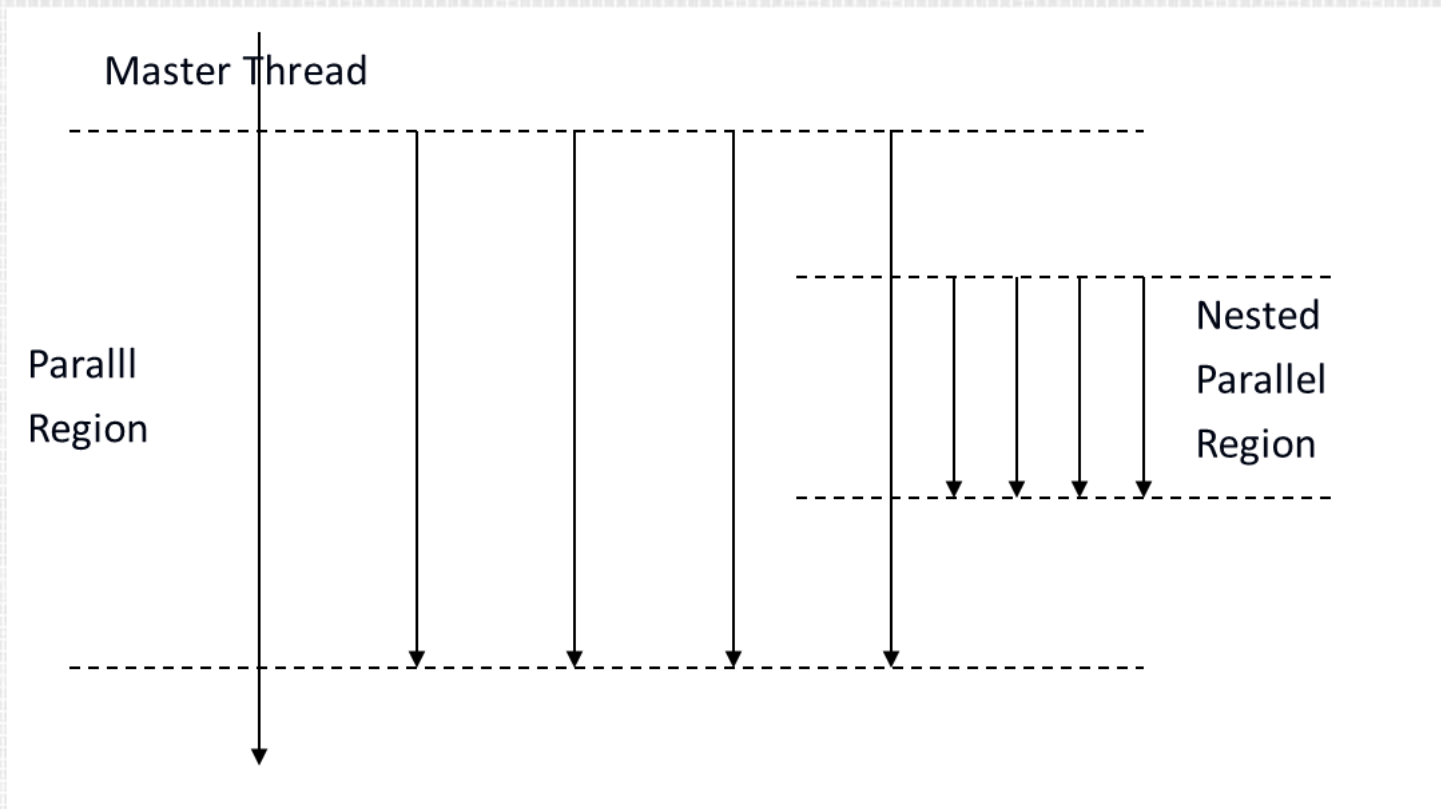
一个共享内存的系统

- 共享内存多处理器
- 内存是共享的，某一个处理器写入内存的数据会立刻被其它处理器访问到。





- 以线程为基础，通过编译指导语句来显式地指导并行化，为编程人员提供对并行化的完整控制。
- 采用**Fork-Join**的执行模式

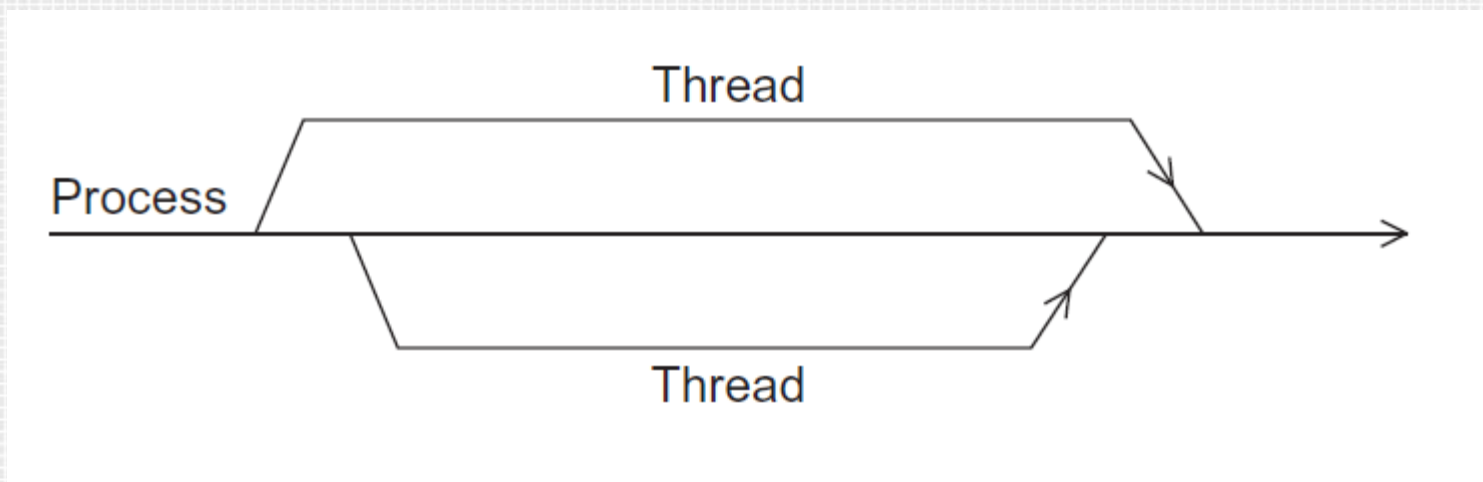




Fork-Join执行模式

- 在开始执行的时候，只有主线程的运行线程存在
- 主线程在运行过程中，当遇到需要进行并行计算的时候，派生出（**Fork**，创建新线程或者唤醒已有线程）线程来执行并行任务
- 在并行执行的时候，主线程和派生线程共同工作
- 在并行代码结束执行后，派生线程退出或者挂起，不再工作，控制流程回到单独的主线程中（**Join**，即多线程的会和）。

一个派生和合并两个线程的进程





Pragma

- 特殊的预处理指令
- 在系统中加入预处理器指令一般是用来允许使用不规范的基本C语言行为。
- 那些不支持 **pragmas** 的编译器忽略 **pragma**指令指示的那些语句。

#pragma



- **#Pragma 指令的作用**
 - 设定编译器的状态或者是指示编译器完成一些特定的动作。
- **#Pragma 指令对每个编译器给出了一个方法,在保持与C和C++语言完全兼容的情况下,给出主机或操作系统专有的特征。**
- 依据定义,编译指示是机器或操作系统专有的,且对于每个编译器都是不同的。
- 其格式一般为: **#Pragma Para**
 - 其中**Para**为参数。



- OpenMP的格式一般为: **#Pragma omp Para**
 - 其中**Para**部分就包含了具体的编译制导语句, 包括 **parallel, for, parallel for, section, sections, single, master, critical, flush, ordered**和**atomic**。
- 下面来看一些常用的参数。



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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```

编译和运行OpenMP程序

```
gcc -g -Wall -fopenmp -o omp_hello  
omp_hello.c  
./omp_hello 4
```

↑ **compiling**

↑ **running with 4 threads**

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

**possible
outcomes**

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



OpenMp pragma

- **# pragma omp parallel**
 - 最基本的**parallel**指令形式。
 - 运行结构化代码块的线程数将由系统决定。

子句

- 一些用来修改指令的文本。
 - **num_threads** 子句
 - 允许程序员指令执行后面代码块的线程数
- # pragma omp parallel num_threads (thread_count)**



需要注意的！

- 程序可以启动的线程数可能会受到系统定义的限制。
- **OpenMP**标准并不保证实际情况下能够启动`thread_count`个线程。
- 目前，大部分的系统能够启动数百数千个线程，因此除非你试图启动许多线程，否则一般情况下所需的线程数都能满足。



- 当程序到达**parallel**指令时，究竟会发生什么？
 - 在**parallel**之前，程序只使用一个线程
 - 在到达**parallel**时，原来的线程继续执行，**thread_count-1**个线程被启动。
- 线程组
- 主线程
- 从线程

编译制导

■ 语句格式

#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。



编译制导

■ 作用域

■ 静态扩展

- 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

■ 动态扩展

- 包括静态范围和孤立语句



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

```
#pragma omp parallel
{
    ...
    #pragma omp for
    for(...) {
        ...
        sub1();
        ...
    }
    ...
    sub2();
    ...
}
```

孤立语句

critical和**sections**语句出现在封闭的并行域之外

```
void sub1()
{
    ...
    #pragma omp critical
    ...
}
void sub2()
{
    ...
    #pragma omp sections
    ...
}
```



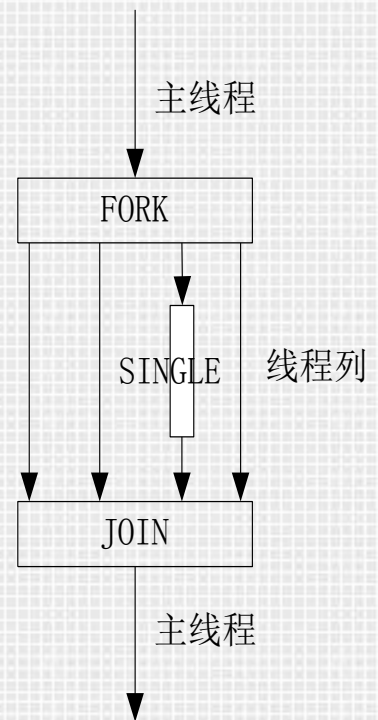
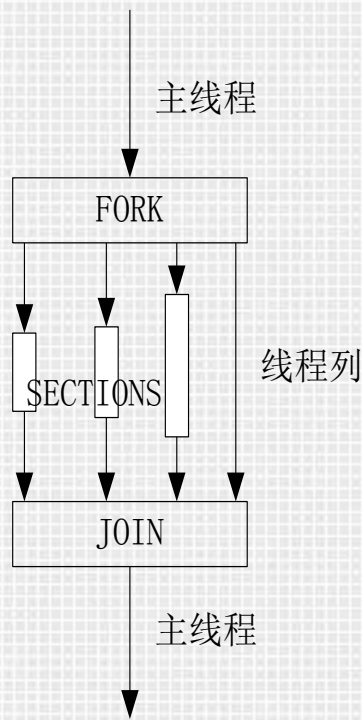
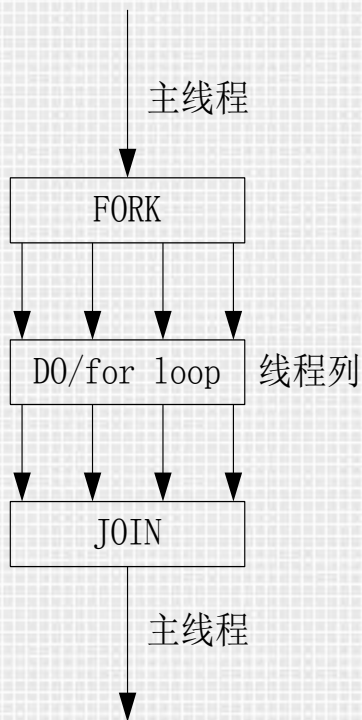

并行域结构

- 并行域中的代码被所有的线程执行
- 具体格式
 - **#pragma omp parallel [clause[[,]clause]...]newline**
 - **clause=**
 - **if (scalar_expression)**
 - **private (list)**
 - **shared (list)**
 - **default (shared | none)**
 - **firstprivate (list)**
 - **reduction (operator: list)**
 - **copyin (list)**



共享任务结构

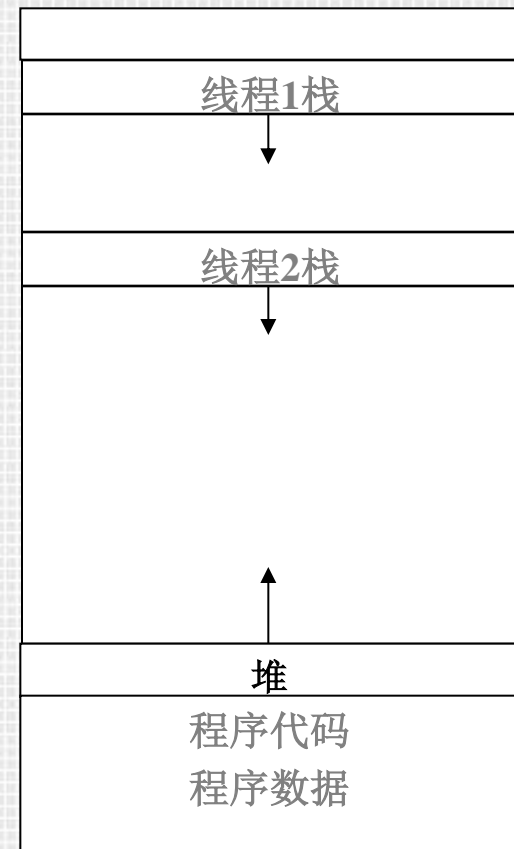
- 共享任务结构将它所包含的代码划分给线程组的各成员来执行
 - 并行for循环
 - 并行sections
 - 串行执行





控制数据的共享属性

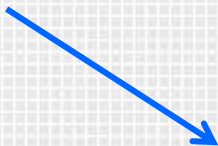
- OpenMP程序在同一个共享内存空间上执行
 - 可以任意使用这个共享内存空间上的变量进行线程间的数据传递
- 内存分布结构如图
 - 每一个线程的栈空间都是私有的
 - 全局变量以及程序代码都是全局共享
 - 动态分配的堆空间也是共享的
 - 通过threadprivate指出的数据结构在每一个线程中都会有一个副本
 - shared定义变量作用域是共享的
 - private私有的



错误检查

- 编译器不支持OpenMP时:

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



```
#ifdef _OPENMP  
# include <omp.h>  
#endif
```



```
# ifdef _OPENMP
```

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

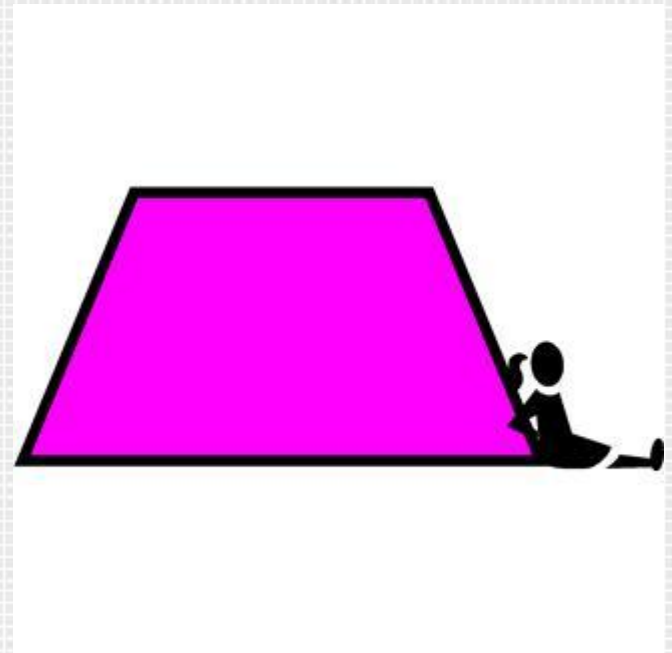
```
    int thread_count = omp_get_num_threads ( );
```

```
# else
```

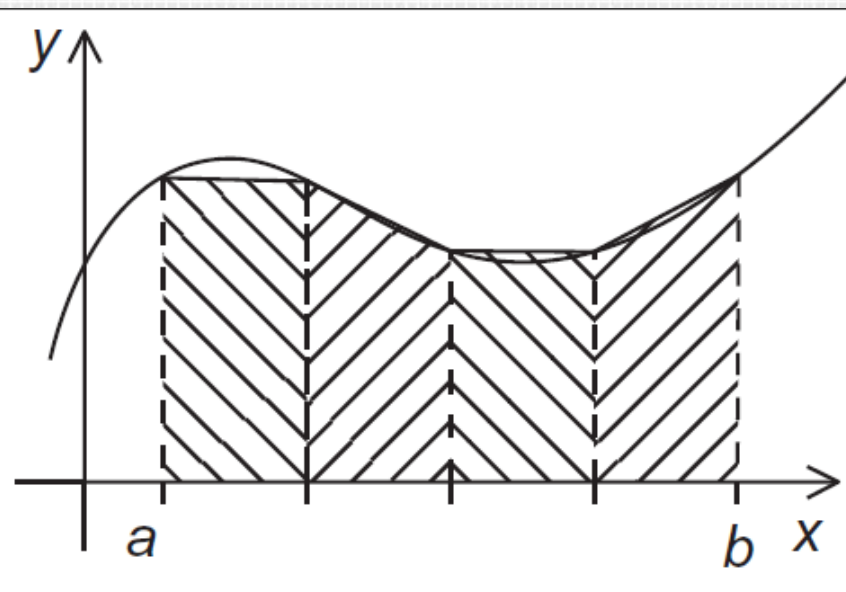
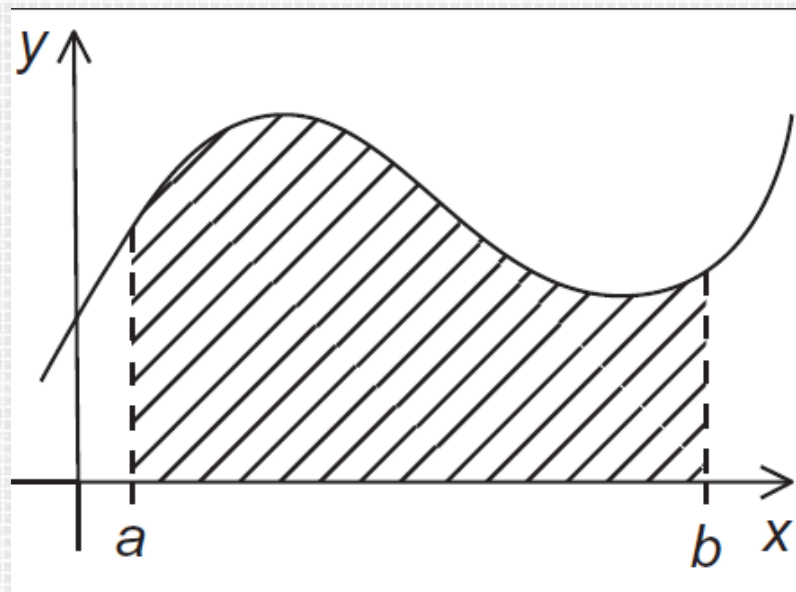
```
    int my_rank = 0;
```

```
    int thread_count = 1;
```

```
# endif
```



5.2 梯形积分法





串行程序

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



第一个OpenMP 版本

1) 识别定义两类任务:

- a) 单个梯形面积的计算
- b) 梯形面积的求和.

2) 在1 (a) 的任务中, 没有任务间的通信, 但这一组任务中的每一个任务都与1 (b) 中的任务通信。



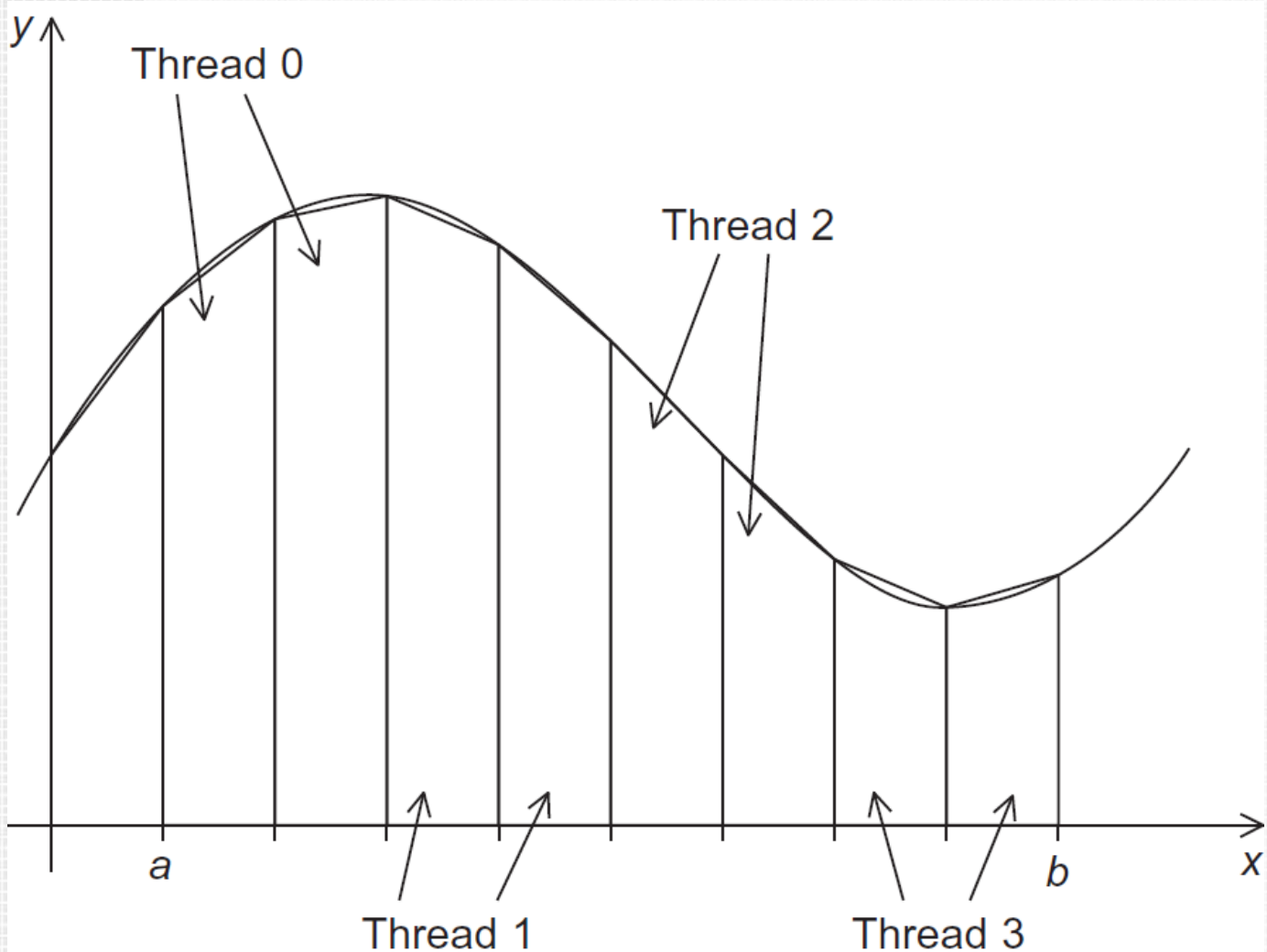
第一个OpenMP 版本

3) 假设梯形的数量远大于核的数量.

- 于是通过给每个线程分配连续的梯形块（和每个核一个线程）来聚集任务。



将梯形分配给各个线程





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

如果两个线程试图同时执行这条语句，
那么结果将是不可预计的：

`global_result += my_result ;`


竞争条件
临界区
互斥



互斥现象

```
# pragma omp critical  
global_result += my_result ;
```

一次只有一个线程执行这段代码，并且第一个线程完成操作前，没有其他的线程能开始执行这段代码。



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

```
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;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
# pragma omp parallel num_threads(thread_count)
    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();

    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
        *global_result_p += my_result;
}... /* Trap */
```




5.3 变量的作用域



作用域

- 在串行编程中，变量的作用域由程序中的变量可以使用的那部分组成。
- 在**OpenMP**中，变量的作用域设计在**parallel**块中能够访问该变量的线程集合。



在OpenMP中的作用域

- 一个能够被线程组中的所有线程访问的变量拥有共享作用域。
- 一个只能被单个线程访问的变量拥有私有作用域。
- 在**parallel** 指令前已经被声明的变量，拥有共享作用域；而在**parallel**块中声明的变量拥有私有作用域。



5.4 归约子句

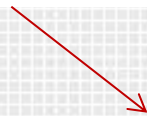


我们需要更复杂的版本将每个线程的局部
计算结果加到***global_result***。

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

我们可能这样定义：

```
double Trap(double a, double b, int n);
```



```
global_result = Trap(a, b, n);
```



我们可能更倾向于以下的函数原型：

```
double Local_trap(double a, double b, int n);
```

如果做出这样的改变...

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

... 这就相当于强制各个线程顺序执行梯形积分法



我们可以通过在**parallel**块中声明一个私有变量和将临界区移到函数调用之后来避免这个问题。

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0;  /* private */

    my_result += Local_trap(double a, double b, int n);
# pragma omp critical
    global_result += my_result;
}
```



**I think we
can do
better.**

**Neither
do I.**

**I don't
like it.**





归约操作符

- 归约操作符（**reduction operator**）是一个二元操作（例如：加减法）。
- 归约（**reduction**）就是将相同的归约操作符重复地应用到操作数序列来得到一个结果的。
- 所有的操作的中间结果都存储在同一个变量里：**归约变量（the reduction variable）**。



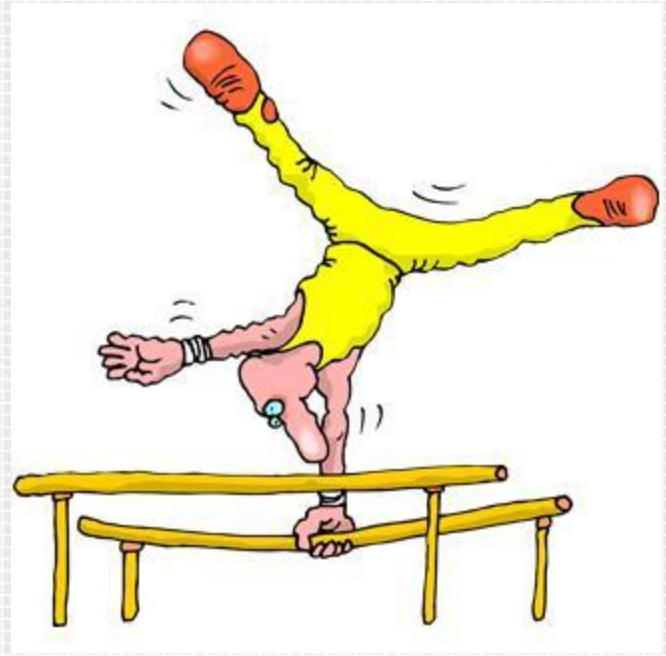
reduction子句可以被添加到一个**parallel**指令。

```
reduction(<operator>: <variable list>)
```



+, *, -, &, |, ^, &&, ||

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




5.5 “PARALLEL FOR” 指令

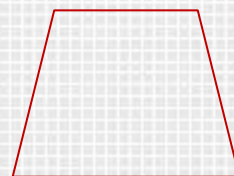


Parallel for

- **parallel for**指令生成一组线程来执行后面的结构化代码块。
- 然而，在**parallel for**指令之后的结构化块必须是一个**for**循环。
- 另外，运行**parallel for**指令，系统通过在线程间划分循环迭代来并行化**for**循环。



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



```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
# pragma omp parallel for num_threads(thread_count) \  
    reduction(+: approx)  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```



可并行化的for语句的合法表达形式

for $\left(\begin{array}{l} \text{index} = \text{start} ; \text{index} >= \text{end} ; \\ \text{index} < \text{end} \\ \text{index} <= \text{end} \\ \text{index} > \text{end} \end{array} \right. \begin{array}{l} \text{index}++ \\ ++\text{index} \\ \text{index}-- \\ --\text{index} \\ \text{index} += \text{incr} \\ \text{index} -= \text{incr} \\ \text{index} = \text{index} + \text{incr} \\ \text{index} = \text{incr} + \text{index} \\ \text{index} = \text{index} - \text{incr} \end{array} \right)$



注意事项

- 变量`index` 必须是整型或指针类型。
- 表达式`start`, `end`, 和 `incr` 必须有一个兼容的类型。例如：如果`index`是一个指针，那么`incr`必须是整型。



注意事项

- 表达式 `start`, `end`, 和 `incr` 不能够在循环执行期间改变。
- 在循环执行期间, 变量 `index` 只能够被 **for** 语句中的“增量表达式”修改。

数据依耐性

```
fibonacci[0] = fibonacci[1] = 1;  
for (i = 2; i < n; i++)  
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
```

note 2 threads

```
fibonacci[0] = fibonacci[1] = 1;  
# pragma omp parallel for num_threads(2)  
for (i = 2; i < n; i++)  
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
```

1 1 2 3 5 8 13 21 34 55

这是正确的结果

1 1 2 3 5 8 0 0 0 0

我们也可能
偶然得到



What happened?



1. **OpenMP**编译器不检查被 **parallel for** 指令并行化的循环所包含的迭代间的依赖关系。
2. 一个或多个迭代结果依赖于其他迭代的循环，一般不能被**OpenMP**并行化



π 值估计

$$\pi = 4 \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right] = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;  
double sum = 0.0;  
for (k = 0; k < n; k++) {  
    sum += factor/(2*k+1);  
    factor = -factor;  
}  
pi_approx = 4.0*sum;
```



OpenMP solution #1

循环依赖

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



OpenMP solution #2

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

保证**factor** 拥有
私有作用域



default 子句

- 让程序员明确块中每个变量的作用域。

```
default (none)
```

- 有了这个子句，编译器将要求我们明确在这个块中使用的每个变量和已经在块外声明的变量的作用域。



default 子句

```
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    default(none) reduction(+:sum) private(k, factor) \
    shared(n)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor/(2*k+1);
}
```



5.6 更多关于OPENMP的循环： 排序



冒泡排序

```
for (list_length = n; list_length >= 2; list_length--)  
    for (i = 0; i < list_length - 1; i++)  
        if (a[i] > a[i+1]) {  
            tmp = a[i];  
            a[i] = a[i+1];  
            a[i+1] = tmp;  
        }
```





奇偶变换排序

```
for (phase = 0; phase < n; phase++)  
    if (phase % 2 == 0)  
        for (i = 1; i < n; i += 2)  
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);  
    else  
        for (i = 1; i < n-1; i += 2)  
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```



奇偶变换排序

Phase	Subscript in Array			
	0	1	2	3
0	9	\leftrightarrow 7	8	\leftrightarrow 6
	7	9	6	8
1	7	9	\leftrightarrow 6	8
	7	6	9	8
2	7	\leftrightarrow 6	9	\leftrightarrow 8
	6	7	8	9
3	6	7	\leftrightarrow 8	9
	6	7	8	9



奇偶排序的第一个 OpenMP 实现

```
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;
            }
        }
}
```



奇偶排序的第二个 OpenMP实现

```
# 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;  
                }  
            }  
    }  
}
```



用两条**parallel for** 语句或**for** 语句运行奇偶排序的时间
(单位: 秒)

thread_count	1	2	3	4
Two parallel for directives	0.770	0.453	0.358	0.305
Two for directives	0.732	0.376	0.294	0.239





5.7 循环调度



我们要并行化这个循环。

```
sum = 0.0;  
for (i = 0; i <= n; i++)  
    sum += f(i);
```

Thread	Iterations
0	$0, n/t, 2n/t, \dots$
1	$1, n/t + 1, 2n/t + 1, \dots$
\vdots	\vdots
$t - 1$	$t - 1, n/t + t - 1, 2n/t + t - 1, \dots$

循环划分



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

我们定义的函数**f**



结果

- 每次函数 $f(i)$ 调用 i 次 \sin 函数。
- 执行 $f(2i)$ 的时间几乎是执行 $f(i)$ 的时间的两倍。
- $n = 10,000$
 - one thread
 - run-time = 3.67 seconds.

结果

- **$n = 10,000$**
 - 两个线程
 - 缺省分配
 - **run-time = 2.76 seconds**
 - **speedup = 1.33**
- **$n = 10,000$**
 - 两个线程
 - 循环分配
 - **run-time = 1.84 seconds**
 - **speedup = 1.99**





Schedule 子句

■ 缺省调度:

```
sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum)
    for (i = 0; i <= n; i++)
        sum += f(i);
```

■ Schedule子句:

```
sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum) schedule(static,1)
    for (i = 0; i <= n; i++)
        sum += f(i);
```




schedule (type , chunksize)

- **Type** 可以是下面任意一个：
 - **Static**: 迭代能够在循环执行前分配给线程。
 - **Dynamic**或**guided**: 迭代在循环执行时被分配给线程，因此在一个线程完成了它的当前迭代集合后，它能从运行时系统中请求更多。
 - **Auto**: 编译器和运行时系统决定调度方式。
 - **Runtime**: 调度在运行时决定。
- **Chunksize**是一个正整数。



Static调度类型

12个迭代, 0, 1, ..., 11, 和3个线程

```
schedule(static, 1)
```

Thread 0: 0, 3, 6, 9

Thread 1: 1, 4, 7, 10

Thread 2: 2, 5, 8, 11



Static调度类型

12个迭代, 0, 1, ..., 11, 和3个线程

```
schedule(static, 2)
```

Thread 0 : 0, 1, 6, 7

Thread 1 : 2, 3, 8, 9

Thread 2 : 4, 5, 10, 11



Static调度类型

12个迭代, 0, 1, ..., 11, 和3个线程

```
schedule(static, 4)
```

Thread 0 : 0, 1, 2, 3

Thread 1 : 4, 5, 6, 7

Thread 2 : 8, 9, 10, 11



Dynamic调度类型

- 迭代被分成**chunksize**个连续迭代的块。
- 每个线程执行一块，并且当一个线程完成一块时，它将从运行时系统请求另一块。
- 这一直持续到所有的迭代完成。
- **chunksize** 可以被忽略，当它被忽略时，**chunksize**为1。



Guided调度类型

- 每个线程也执行一块，并且当一个线程完成一块时，将请求另一个。
- 然而，在**guided**调度中，当块完成后，新块的大小会变小。
- 在**guided**调度中，如果没有指定，那么块的大小为1。
- 如果指定了**chunksize**，那么块的大小就是 **chunksize**，除了最后一块可以比 **chunksize** 小。



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	1	0

使用**guided**调度为两个线程分配梯形积分法的1–9999次迭代



Runtime调度类型

- 当**schedule(runtime)**指定是，系统使用环境变量**OMP_SCHEDULE** 在运行时来决定如何调度循环。
- **OMP_SCHEDULE**环境变量会呈现任何能被**static**，**dynamic**，或**guided**调度所使用的值。



5.8 生产者和消费者问题



5.8.1 队列

- 队列可以看作是在超市中等待付款的消费者的抽象，队列中的元素是消费者。
- 队列是在多线程应用程序中经常使用到的数据结构。
- 例如，我们有几个“生产者”的线程和几个“消费者”线程。
 - 生产者线程“产生”对服务器数据的请求。
 - 消费者线程通过发现和生成数据，来“消费”请求。



5.8.2 消息传递

- 每一个线程有一个共享消息队列，当一个线程要向另一个线程“发送消息”时，它将信息放入目标线程的消息队列中。
- 一个线程接收消息时只需从它的消息队列的头部取出消息。



消息传递

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {  
    Send_msg();  
    Try_receive();  
}  
  
while (!Done())  
    Try_receive();
```




5.8.3 发送消息

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

为什么要加入临界区的概念？




5.8.4 接收消息

```
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);
```



5.8.5 终止检查

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



每个线程在**for**循环结束后将该计数器加**1**。



5.8.6 启动(1)

- 当程序开始执行时，主线程将得到命令行参数并分配一个数组空间给消息队列，每个线程对应着一个消息队列。
- 由于每个线程可以向其他任意的线程发送消息，所以这个数组应该被所有线程共享，而且每个线程可以向任何一个消息队列插入一条消息。



启动(2)

- 一个或多个线程可能在其它线程之前完成它的队列分配。
- 我们应当使用显式路障，当线程遇到路障时，它将被阻塞，直到组中所有线程都达到了障碍。
- 当组中所有线程都达到了这个障碍时，这些线程可以接着往下执行。

```
# pragma omp barrier
```



5.8.7 Atomic指令(1)

- 和**critical**指令不同，它只能保护由一条**C**语言赋值语句所形成的临界区。

```
# pragma omp atomic
```

- 此外，语句必须是下列形式之一：

```
x <op>= <expression>;  
x++;  
++x;  
x--;  
--x;
```




Atomic指令(2)

- **<op>**可以是以下任意的二元操作符：

`+, *, -, /, &, ^, |, <<, or >>`

- 许多处理器提供了专门的装载-修改-存储（**load-modify-store**）指令。
- 使用这种专门的指令而不使用保护临界区的通用结果，可以更高效地保护临界区。

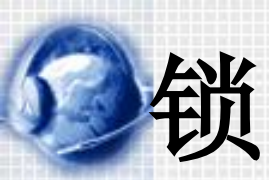


5.8.8 临界区

- OpenMP 提供了向**critical** 指令添加名字的添加：

```
# pragma omp critical(name)
```

- 采取这种方式，两个用不同名字的**critical** 指令保护的代码块就可以同时执行。
- 因此，我们需要在程序执行的过程中设置临界区的名字。



锁

- 锁是由一个数据结构和定义在这个数据结构上的函数组成，这些函数使得程序员可以显式地强制对临界区进行互斥访问





锁

```
/* Executed by one thread */
```

```
Initialize the lock data structure;
```

```
. . .
```

```
/* Executed by multiple threads */
```

```
Attempt to lock or set the lock data structure;
```

```
Critical section;
```

```
Unlock or unset the lock data structure;
```

```
. . .
```

```
/* Executed by one thread */
```

```
Destroy the lock data structure;
```

5.8.9 在消息传递函数中使用锁

在消息传递程序中，我们想要确保的是对每个消息队列进行互斥访问，而不是对于一个特定的代码块。

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

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, msg);
omp_unset_lock(&q_p->lock);
```

5.8.9 在消息传递函数中使用锁

```
# pragma omp critical
/* q_p = msg_queues[my_rank] */
Dequeue(q_p, &src, &msg);
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &msg);
omp_unset_lock(&q_p->lock);
```




5.8.10 **critical**指令、**atomic**指令、锁的比较

- 三种机制可以实现对实现对临界区的访问：
 - **atomic**指令是实现互斥访问最快的方法
 - 当程序中有多多个不同的由**atomic**指令保护的临界区，则应当使用命名的**critical**指令或者锁。
 - 锁机制适用于需要互斥的是某个数据结构而不是代码块的情况。



5.8.11 经验

1. 对同一个临界区不应当混合使用不同的互斥机制。
2. 互斥执行不保证公平性。
 - a. 也就是说可能某个线程会被一直阻塞以等待对某个临界区的执行。
3. “嵌套”互斥结构可能会产生意料不到的结果。



5.9 缓存、缓存一致性、伪共享



矩阵-向量乘法

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

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}

```
for (i = 0; i < m; i++) {  
    y[i] = 0.0;  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]*x[j];  
}
```



矩阵-向量乘法

```
# 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][j]*x[j];
}
```

矩阵向量乘法的运行
时间和效率（时间是
秒）

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 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



```
void Tokenize(  
    char*  lines[]      /* in/out */,  
    int    line_count   /* in      */,  
    int    thread_count /* in      */) {  
    int my_rank, i, j;  
    char *my_token;  
  
#   pragma omp parallel num_threads(thread_count) \  
        default(none) private(my_rank, i, j, my_token) \  
        shared(lines, line_count)  
    {  
        my_rank = omp_get_thread_num();  
#   pragma omp for schedule(static, 1)  
        for (i = 0; i < line_count; i++) {  
            printf("Thread %d > line %d = %s", my_rank, i, lines[i]);  
            j = 0;  
            my_token = strtok(lines[i], " \t\n");  
            while ( my_token != NULL ) {  
                printf("Thread %d > token %d = %s\n", my_rank, j, my_token);  
                my_token = strtok(NULL, " \t\n");  
                j++;  
            }  
        } /* for i */  
    } /* omp parallel */  
  
} /* Tokenize */
```




5.11 小结(1)

- **OpenMP**是一个共享存储系统的编程标准。
- **OpenMP**使用专门的函数和预处理指令 **pragmas**.
- **OpenMP**使用多线程而不是多进程。
- 许多**OpenMP**指令可以被子句修改。



小结(2)

- 共享内存程序开发时的一个主要问题是可能存在竞争条件。
- **OpenMP**提供了多种机制实现临界区的互斥访问。
 - **Critical** 指令
 - 程序中可以使用命名的**Critical** 指令
 - **Atomic**指令
 - 简单锁



小结(3)

- 缺省情况下，绝大多数系统在并行化的**for**循环中对迭代使用块划分。
- **OpenMP**提供许多调度选项。
- 在**OpenMP**中，变量的作用域是可以访问该变量的线程集合



小结(4)

- 归约操作符是一个二元操作符，而一次归约计算对一个操作数列重复使用相同的归约操作符，从而得到一个唯一的结果。