

## 第五章

# 用OpenMP进行共享内存编程

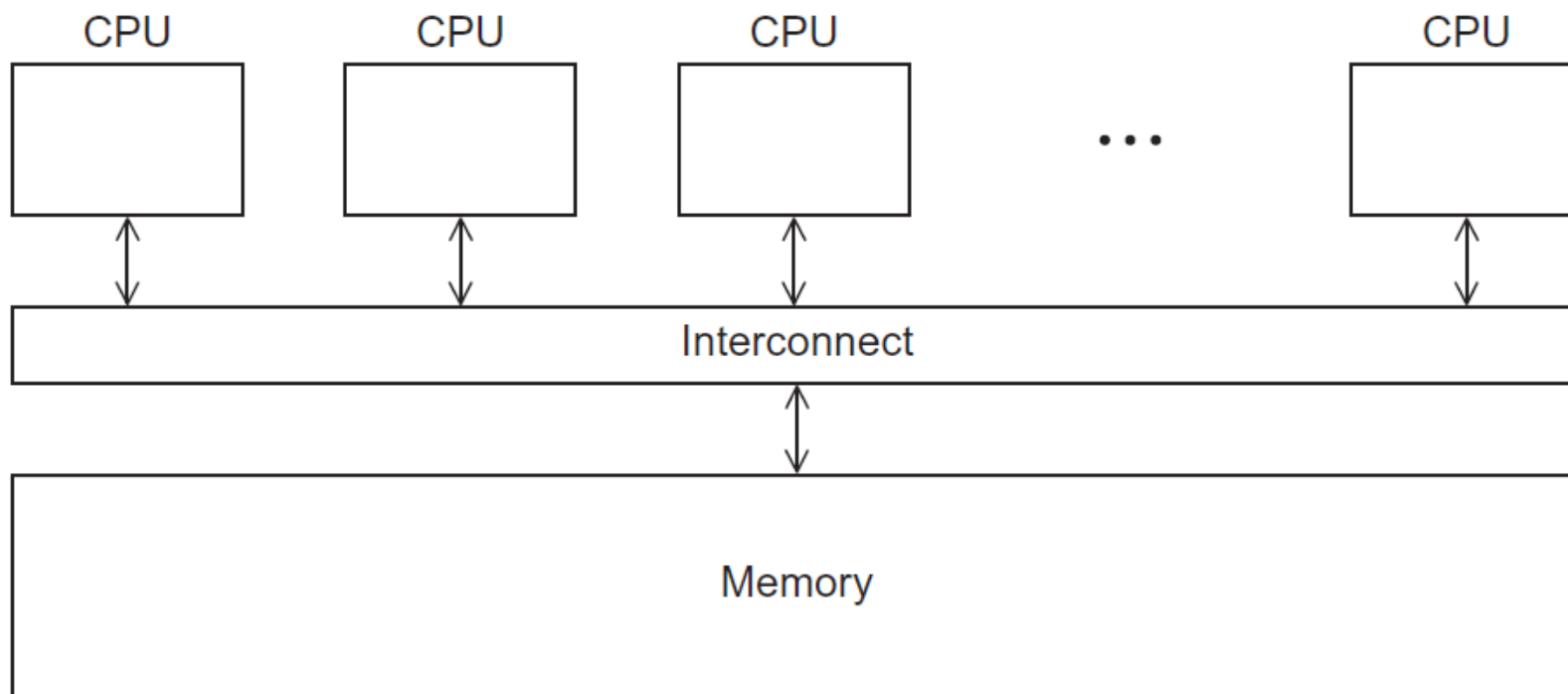
# 目录

- 编写使用OpenMP的程序.
- 使用OpenMP并行化串行for循环
- 任务并行.
- 显式线程同步.
- 共享内存编程中的标准问题.

# OpenMP

- 用于共享内存并行编程的API。
- MP = multiprocessing
- 为每个线程或进程都有可能访问所有可用内存系统而设计。
- 系统被看作是一个核心或CPU的集合，所有的核心或CPU都可以访问主存。

# 共享内存系统



# Pragmas

- 特殊的预处理指令。
- 通常添加到系统中是为了允许不属于基本C规范的行为。
- 不支持pragmas的编译器会忽略它们。

#pragma

```
#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 */
```

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

```
./ omp_hello 4
```

running with 4 threads

compiling

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

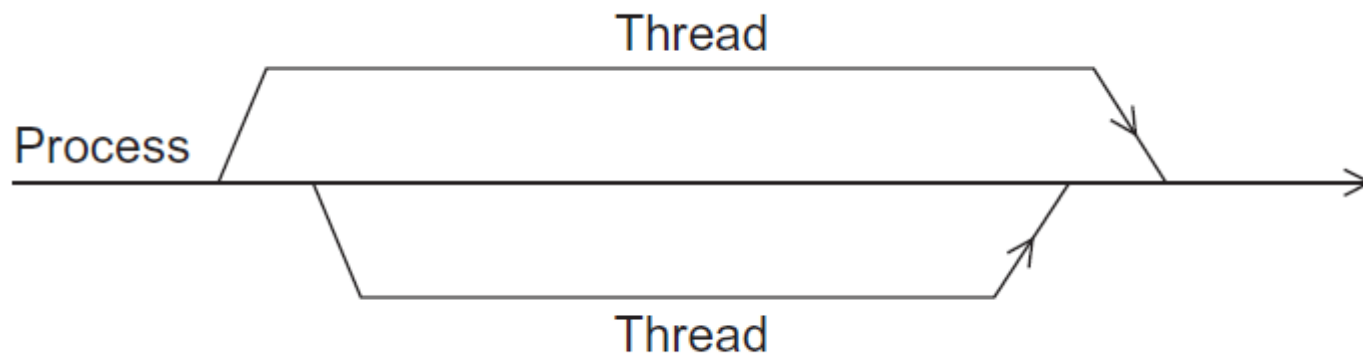
Hello from thread 3 of 4  
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 0 of 4

# OpenMP pragmas

- # pragma omp parallel
  - 最基本的 parallel 指令.
  - 运行结构化代码块的线程数由运行时系统决定.



# 一个派生和合并两个线程的进程



# 子句

- 子句是一些用来修改指令的文本.
- `num_threads`子句添加到 `parallel` 指令中。
- 允许程序员指定执行代码块的线程数.

`# pragma omp parallel num_threads ( thread_count )`

# 注意:

- 程序可以启动的线程数可能受系统定义的限制。
- OpenMP标准并不能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甚至数千个线程。
- 除非我们试图启动大量线程，否则我们几乎总是会得到所需要的线程数。

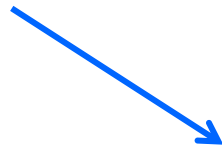
# 专业术语

- 在OpenMP的语法中，执行并行块的线程集合（原始线程和新线程）被称作线程组，原始线程被称为主线程，额外的线程被称为从线程。



# 条件编译：以防编译器不支持OpenMP

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



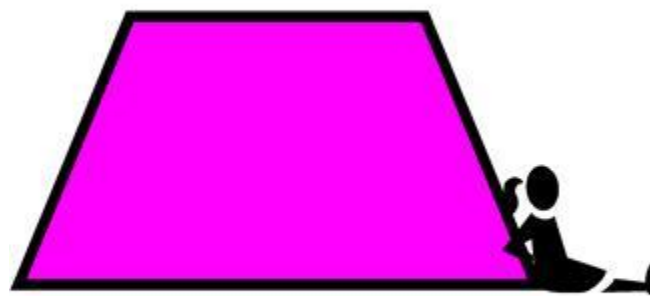
```
#ifdef _OPENMP
```

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

```
#endif
```

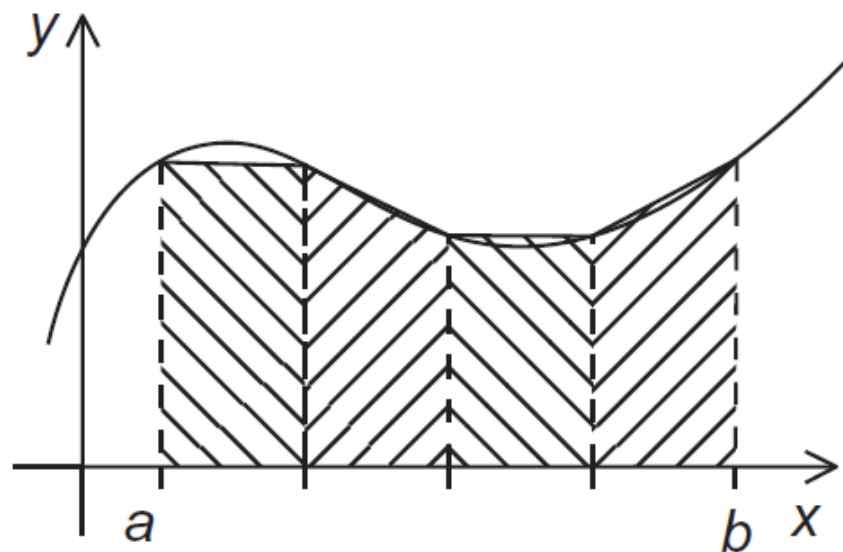
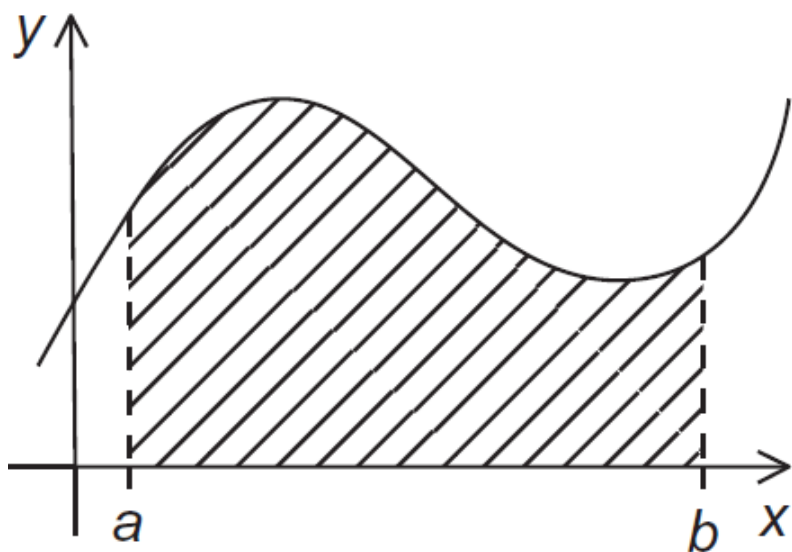
# 条件编译：以防编译器不支持OpenMP

```
# 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
```



# 梯形积分法

# 梯形积分法





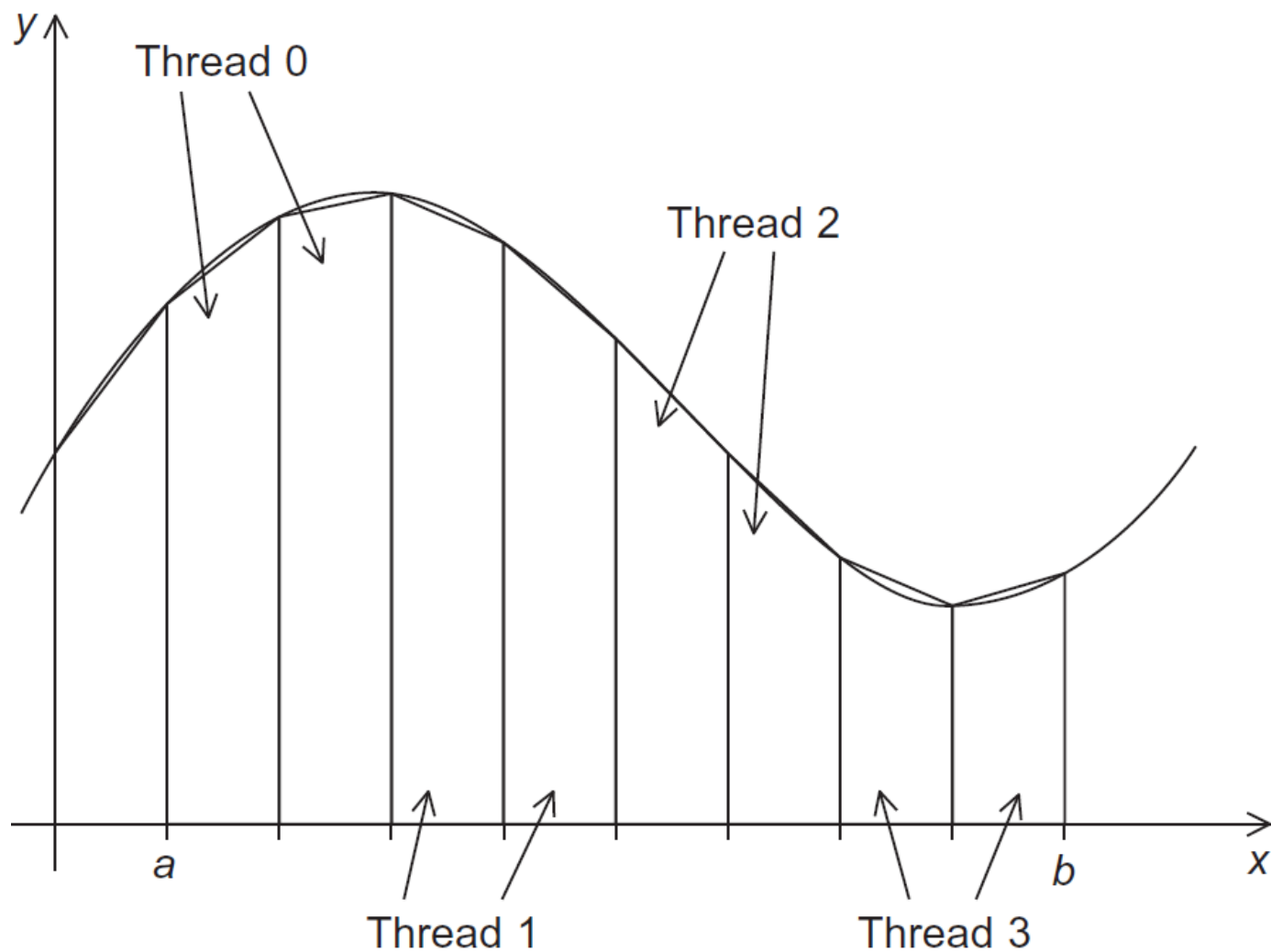
# 串程序

```
/* 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, 任务间没有通信, 但这一组任务中的每一个任务都与1(b)中的任务通信。
- 3) 我们假设梯形的数量远大于核的数量.
- 因此, 我们通过给每个线程分配连续的梯形块 (每个核心分配一个线程) 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>

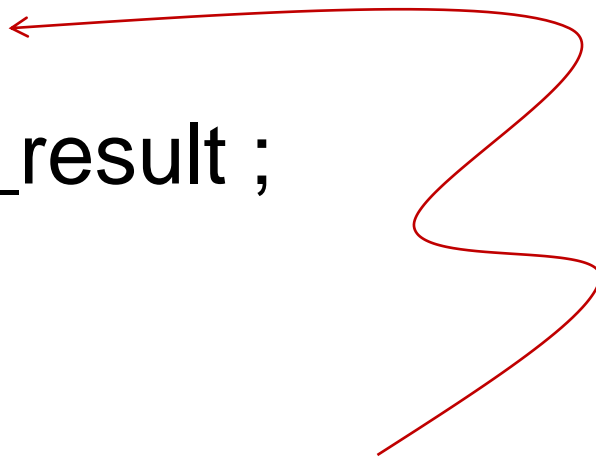
当两个(或多个)线程试图同时执行时，将导致不可预测的结果：

`global_result += my_result ;`



# 互斥锁

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



一次只能有一个线程执行下面的结构化代码

```

#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 */

```



# 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被使用的那些部分组成.
- 在OpenMP中，变量的作用域涉及在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 **paralle** 指令前已经被声明的变量，默认作用域是共享的。





# 规约子句

我们需要一个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);
```

我们可能更倾向于以下函数原型.

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



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

如果我们这样使用，就没有临界区！

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

如果我们这样修改它...

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

...我们强制线程按顺序执行.

我们可以通过在 **parallel** 块中声明一个私有变量和将临界区移动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;
}
```



# 归约操作符

- 归约操作符是一种二元操作(如加法或减法)
- 规约是一种重复地对操作数序列，应用相同的归约操作符以获得单个结果的计算
- 所有操作的中间结果都应该存储在同一个变量中:归约变量.



归约子句可以添加到 **parallel** 指令中。

```
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);
```

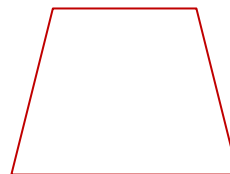


# PARALLEL FOR 指令

# Parallel for

- `fork` 一组线程来执行后面的结构化代码块
- 然而，`parallel for` 指令后面的结构化代码块必须是一个 `for` 循环。
- 此外，使用 `parallel for` 指令，系统通过在线程之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

<b>for</b>	{	index = start ;		index++
				++index
			index < end	index--
			index <= end	--index
			index >= end ;	index += incr
			index > end	index -= incr
				index = index + incr
			index = incr + index	
			index = index - incr	
	}			

# 警告

- 变量**index**必须是整形或指针类型(例如，它不能是浮点数)。
- 表达式**start**、**end**和**incr**必须具有兼容的类型。例如，如果**index**是一个指针，则**incr**必须为整型。
- 表达式**start**、**end**和**incr**在循环执行期间不能改变。
- 在循环执行过程中，变量**index**只能被**for**语句中的“增量表达式”修改。

# 数据依赖

```
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

this is correct

1 1 2 3 5 8 0 0 0 0

but sometimes  
we get this

# 发生了什么？

- OpenMP编译器不会检查被parallel for 指令并行化的循环所包含的迭代间依赖关系，而由程序员来识别这些依赖关系。
- 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正确地并行化。





# π的估计

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

```
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

循环依赖关系

```
# 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** 有私有作用域

# default 子句

- 让程序员指定块中每个变量的作用域.
- **default**(none)
- 使用这个子句，编译器将要求我们指定在块中使用的每个变量的作用域和已经在块之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);
}
```



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;  
        }
```



# 串行奇偶交换排序

```
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]);
```



# 串行奇偶交换排序

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

# 第一个 OpenMP 奇偶排序

```
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;
            }
        }
}
```

## 第二个 OpenMP 奇偶排序

```
# 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语句或两条 for 语句运行奇偶排序的时间  
(单位：秒)

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





## 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	$1, n/t + 1, 2n/t + 1, \dots$
$\vdots$	$\vdots$
$t - 1$	$t - 1, n/t + t - 1, 2n/t + t - 1, \dots$

使用循环划分分配任务

```
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 */
```

函数 **f** 的定义

# 结果

- $f(i)$  调用 $\sin$ 函数 1 次.
- 假设执行 $f(2i)$ 的时间大约是执行 $f(i)$ 的时间的两倍。
- $n = 10,000$ 
  - 一个线程
  - 运行时间 = 3.67 秒.



# 结果

- $n = 10,000$ 
  - 两个线程
  - 缺省分配
  - 运行时间 = 2.76 秒
  - 加速比 = 1.33
- $n = 10,000$ 
  - 两个线程
  - 循环分配
  - 运行时间 = 1.84 秒
  - 加速比 = 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);
```

## ■ 循环调度:

```
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 可以是下列任意一个:
  - static:迭代能够在循环执行之前分配给线程.
  - dynamic or guided:迭代在循环执行时, 被分配给线程.
  - auto:编译器和运行时系统决定调度方式.
  - runtime:调度是在运行时决定。
- chunksize 是一个正整数.

# Static 调度类型

12个迭代  $0, 1, \dots, 11$

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  $0, 1, \dots, 11$

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  $0, 1, \dots, 11$

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

- 迭代还被分解成 **chunksize** 个连续迭代的块
- 每个线程执行一个块，当一个线程完成一个块时，它从运行时系统请求另一个块.
- 持续到所有的迭代完成.
- **chunksize** 可以被忽略。当省略它时，**chunksize** 为1。

# Guided 调度类型

- 每个线程执行一个块，当一个线程完成一个块时，它会请求另一个块。
- 然而，在 **guided** 调度中，当块完成时，新块的大小会减少。
- 如果没有指定 **chunksize**，则块的大小为1。
- 如果指定了 **chunksize**，那么块的大小就是 **chunksize**，除了最后一个块的大小可以小于 **chunksize**。



使用**guided** 调度为两个线程分配梯形积分法的1-9999次迭代。

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

# Runtime 调度类型

- 系统使用环境变量OMP\_SCHEDULE在运行时确定如何调度循环.
- OMP\_SCHEDULE环境变量可以采用任何能被static、dynamic或guided调度所用的值。
  -



## 生产者和消费者

# 队列

- 是一个抽象数据结构，可以被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。
- 在许多多线程应用程序中经常使用的数据结构。
- 例如，假设我们有几个“生产者”线程和几个“消费者”线程。
  - 生产者线程可能“产生”数据请求。
  - 消费者线程可能通过查找或生成所请求的数据来“消费”请求。

# 消息传递

- 每个线程都可以有一个共享的消息队列，当一个线程想要向另一个线程“发送消息”时，它可以将消息加入目标线程的队列中。
- 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();
```

# 发送消息

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

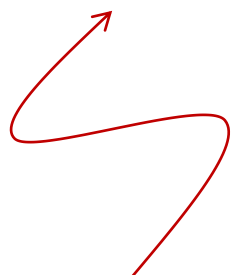
# 接收消息

```
    if (queue_size == 0) return;  
    else if (queue_size == 1)  
#        pragma omp critical  
            Dequeue(queue, &src, &msg);  
    else  
        Dequeue(queue, &src, &msg);  
    Print_message(src, msg);
```



# 终止检测

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



每个线程在完成它的for循环后增加这个值

# 启动 (1)

- 当程序开始执行时，一个单独的线程，即主线程，将获得命令行参数并分配一个数组空间给消息队列，每个线程对应一个消息队列
- 这个数组应该被所有线程共享，因为任何线程都可以向任何其他线程发送消息，因此任何线程都可以在任何队列中加入消息队列。

## 启动 (2)

- 一个或多个线程可能在其他线程之前完成队列分配.
- 我们需要一个路障，这样当一个线程遇到路障时，它就会阻塞，直到线程组中的所有线程都到达路障。
- 当所有线程都到达路障后，线程组中的所有线程都可以继续前进.

```
# pragma omp barrier
```

# Atomic 子句 (1)

- 与critical指令不同，它只能保护由单个C语言赋值语句所形成的临界区.

```
# pragma omp atomic
```

- 此外，该语句明必须具有以下形式之一：

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

```
x++;
```

```
++x;
```

```
x--;
```

```
--x;
```

# Atomic 子句(2)

- 这里 <op> 可以是以下任意的二元操作符:

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

- 许多处理器提供专门的load-modify-store指令.
- 只执行load-modify-store操作的临界区段可以通过使用这个特殊指令, 而不是使用保护临界区的通用结构, 可以更有效地保护临界区。

# 临界区

- OpenMP提供了向临界指令添加名称的选项

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

- 当我们这样做的时候，两个不同名称的 **critical** 指令保护的代码可以同时执行。
- 但是，名称是在编译过程中设置的。如果我们需要程序执行过程中设置名称，**critical** 指令不能满足我们的需要。

# Locks

- 锁是一个由数据结构和定义在这个数据结构上的函数组成，程序员可以显式地强制对临界区互斥访问。



# Locks

```
/* 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;
```



# 在消息传递程序中使用锁

```
# 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);
```

# 在消息传递程序中使用锁

```
# 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);
```

# 经验

- 对同一临界区不应当混合使用不同的互斥机制
- 互斥机制不保证公平性。可能某个线程会被一直阻塞以等待对某个临界区的执行。
- “嵌套”互斥结构可能会产生意料不到的结果

# Matrix-vector multiplication

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

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

$x_0$
$x_1$
$\vdots$
$x_{n-1}$

=

$y_0$
$y_1$
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
$\vdots$
$y_{m-1}$

```

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

# Matrix-vector multiplication

```
# 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(时间以秒为单位)

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 */

```

# 小结(1)

- OpenMP是一种共享内存系统的编程标准.
- OpenMP使用专门的函数和预处理器指令 pragmas.
- OpenMP程序启动多个线程而不是多个进程
- 许多OpenMP指令可以通过子句进行修改.

## 小结(2)

- 开发共享内存程序中的一个主要问题是可能存在竞争条件
- OpenMP提供了多种机制来实现对临界区的互斥访问：
  - Critical 指令
  - 命名的 critical指令
  - Atomic 指令
  - 简单 locks



# 小结(3)

- 默认情况下，大多数系统在并行for循环中对迭代使用块划分区.
- OpenMP提供了多种调度选项.
- 在OpenMP中，变量的作用域是可以访问该变量的线程的集合.

# 小结(4)

- 归约是一种重复地对操作数序列应用相同的归约操作符以获得一个唯一结果的计算.