



太原理工大学
TAIYUAN UNIVERSITY OF TECHNOLOGY

第5章

OpenMP实例

—梯形积分与 π 值估计

岳俊宏

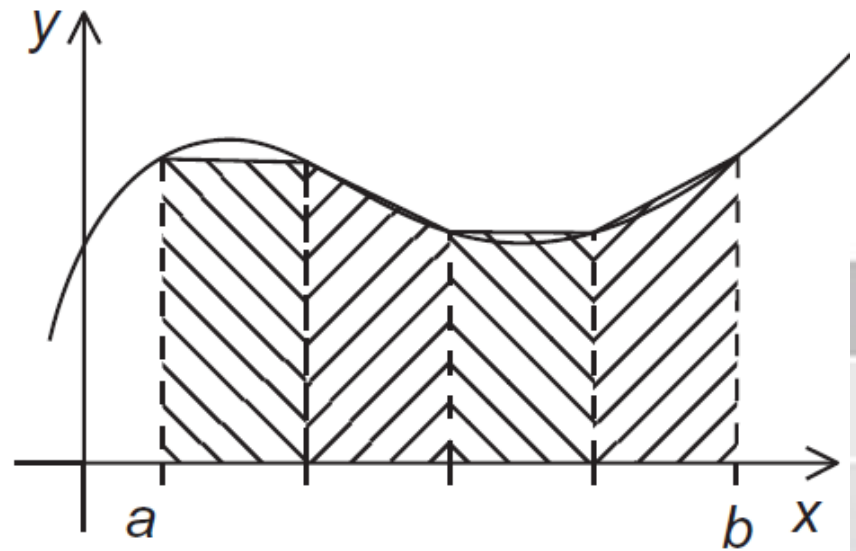
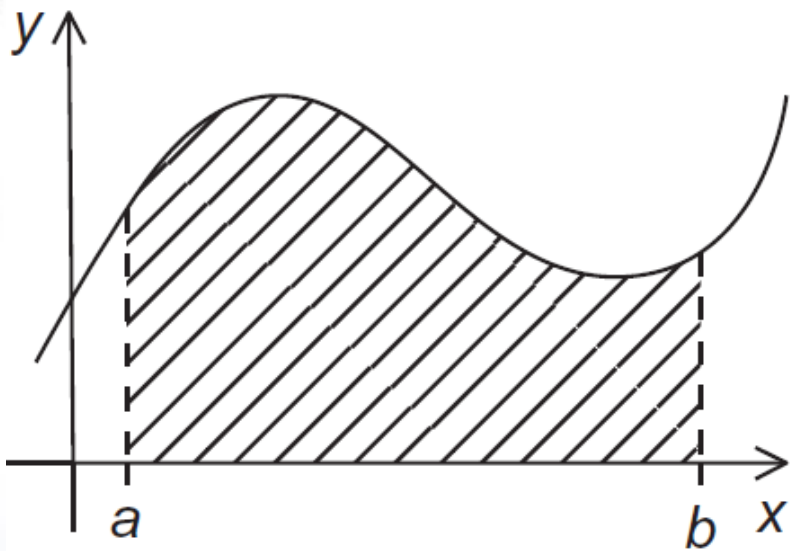
E-mail:yuejunhong@tyut.edu.cn; Tel:18234095983

1. 用OpenMP实现梯形积分
2. 用OpenMP实现 π 值估计



梯形积分法

OpenMP编程



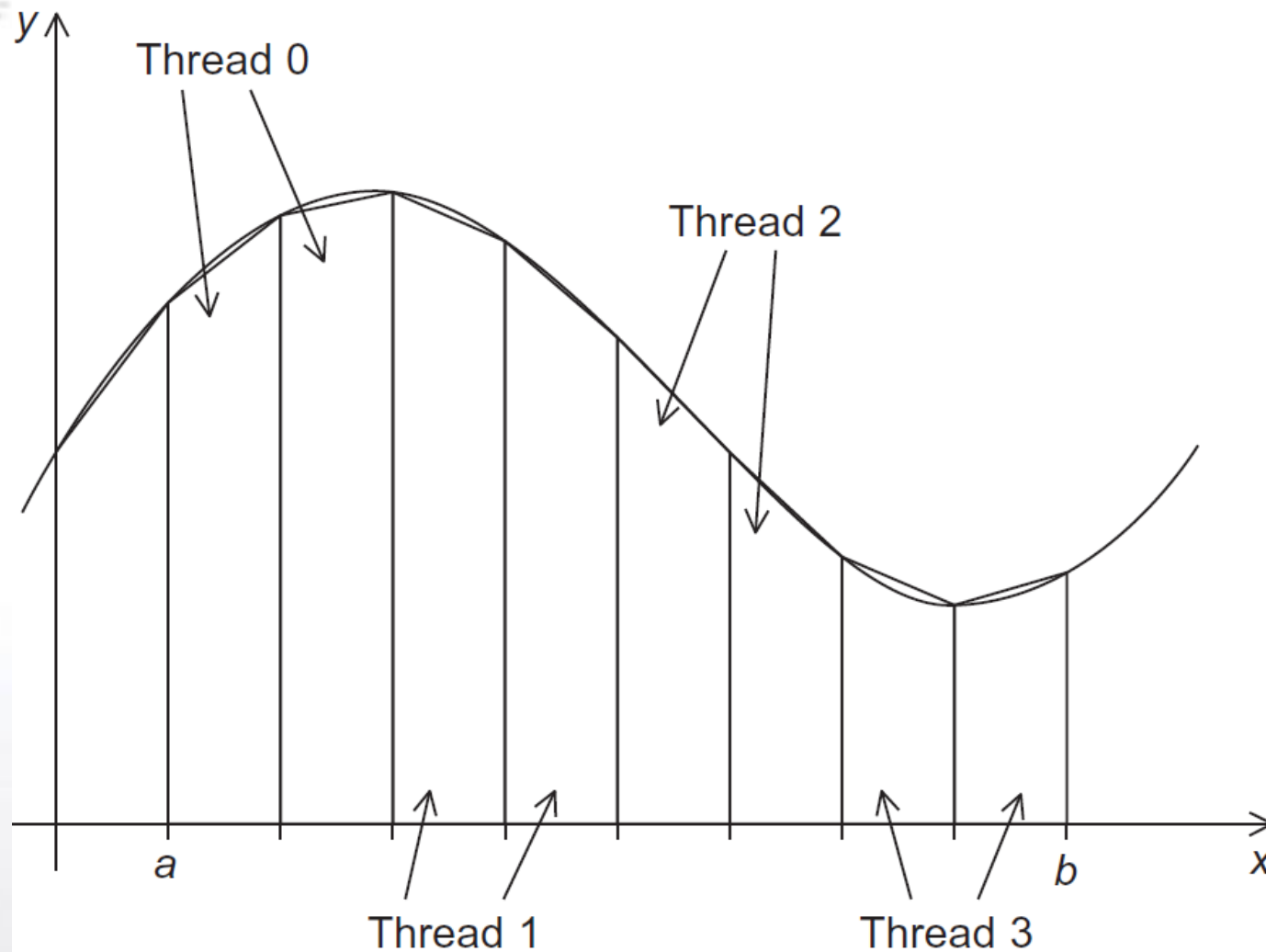
```
int main(int argc, char* argv[]) {
    double  global_result = 0.0; /* Store result in global_result */
    double  a, b;                /* Left and right endpoints   */
    int      n;                  /* Total number of trapezoids */

    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);

    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, approx;
    int i;
    h = (b-a)/n;
    approx = (f(a) + f(b))/2.0;
    for (i = 1; i <= n-1; i++) {
        x = a + i*h;
        approx += f(x);
    }
    *global_result_p = approx*h;
} /* Trap */
```



在trap函数中，每个线程获取它的编号，以及线程总数，然后确定以下的值：

- 1、梯形底长
- 2、给每个线程分配梯形数目
- 3、梯形的左右端点值
- 4、部分和my_result
- 5、将部分和增加到全局和global_result里

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>

当两个或更多的线程加到`global_result`中，可能会出现不可预期的结果：



`global_result += my_result ;`

引起竞争的代码，称为**临界区**

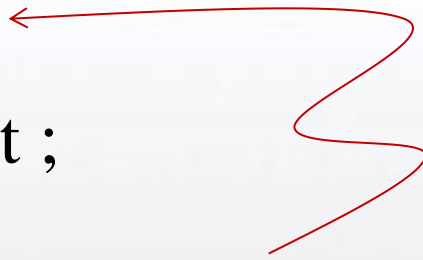
竞争：多个线程试图访问并更新一个共享资源。

因此需要一些机制来确保一次只有一个线程执行临界区，这种操作，我们称为互斥。

- critical指令用在一段代码临界区之前，临界区在同一时间内只能有一个线程执行它，其他线程要执行临界区则需要排队来执行它。

```
#pragma omp critical [(name)] new-line  
structured-block
```

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



确保在一个时间里只有一个线程执行之后的代码块

```

#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 */

```

- 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。
- 共享作用域：能够被线程组中所有线程访问的变量。
- 私有作用域：只能被单个线程访问的变量。
- hello函数和Trap函数中变量的作用域；

```
#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 */
```

```

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 */

```


- 在 main 函数中声明的变量（a,b,n,global_result 和 thread_count）对于所有线程组中被parallel指令启动的线程都可访问。
- parallel块之前被声明的变量缺省作用域是共享的。
 - ✓ 每个线程都能访问a,b,n（Trap的调用）；
 - ✓ 在Trap函数中，虽然global_result_p是私有变量，但它引用了global_result变量，该变量是共享作用域的。因此，*global_result_p拥有共享作用域。
- parallel块中声明的变量有私有作用域（函数中局部变量）



规约子句 reduction

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

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



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

这样是否可以？

- 如果我们选择这种方式，将trap函数内不再有临界区！

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

如果我们这样做...

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

这个结果好么？

可以通过在并行代码块中设置一个私有变量，然后在之后进行互斥操作，这样做可以避免前面的问题

```
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;
}
```

- 规约操作是一个二元操作 (such as addition or multiplication).
- 规约操作就是将一个相同的操作重复应用在一个序列上, 得到一个结果
- 所有的中间结果和最终结果都存储在规约变量里

□ reduction子句:

- ✓ 主要用来对一个或多个参数条目指定一个操作符;
- ✓ 每个线程将创建参数条目的一个私有拷贝, 在区域的结束处, 将用私有拷贝的值进行指定的操作符运算, 初始的参数条目被运算结果更新。用法如下:

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

→ +, *, -, &, |, ^, &&, ||

□ 注意使用减法

```
result = 0;  
for(i=1; i<=4; i++)  
    result -=i;
```

执行过程：

线程0将减1和2, : -3


线程1将减3和4, : -7

而 $(-3) - (-7) = 4$

□ 规约变量是浮点数时，使用不同的线程数，结果可能有些许不同。

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

注意：每个线程的私有拷贝值global_result初始化为0。



parallel for
指令

▣ 梯形积分串行代码

```
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;
```

OpenMP是一种显式并行化方案，能够直接在串行代码的基础上进行修改。

for指令是用来将一个for循环分配到多个线程中执行。

for指令的使用方式：

- ✓ 单独用在parallel语句的并行块中；
- ✓ 与parallel指令合起来形成parallel for指令

用法如下：

```
#pragma omp [parallel] for [子句]  
    for 循环语句
```

□注意：for指令与parallel for指令

- ✓ 派生出一组线程来执行后面的结构化代码块
- ✓ 只能并行for语句

下面将for指令放在parallel并行区域内的代码例子。

```
include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[]){
    int j=0;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp for
        for(j=0;j<4;j++)
            printf("j=%d,ThreadId=%d\n",j,omp_get_thread_num());
    }
    return 0;
}
```

```
[hdusr@Node1 ch5]$ ./omp_for  
j=3,ThreadId=3  
j=0,ThreadId=0  
j=1,ThreadId=1  
j=2,ThreadId=2
```

如果不将for语句放入parallel语句中，是不会创建多个线程来执行的，不妨看看单独使用for语句时的效果。

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

int main(int argc, char* argv[]){
    int j=0;
    #pragma omp for
        for(j=0;j<4;j++)
            printf("j=%d,ThreadId=%d\n",j,omp_get_thread_num());
    return 0;
}
```

```
[hdusr@Node1 ch5]$ ./omp_for1  
j=0,ThreadId=0  
j=1,ThreadId=0  
j=2,ThreadId=0  
j=3,ThreadId=0
```

从结果可以看出四次循环都在一个线程里执行，可见，for指令要与parallel指令结合起来使用。

在一个parallel块中也可以有多个for语句，如一个parallel区域中使用两个for语句：

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv)
{
    int j=0;
    #pragma omp parallel
    {
        #pragma omp for
        for(j=0;j<4;j++)
            printf("First:j=%d,ThreadId=%d\n",j,omp_get_thread_num());
        #pragma omp for
        for(j=0;j<4;j++)
            printf("Second:j=%d,ThreadId=%d\n",j,omp_get_thread_num());
    }
    return 0;
}
```

```
[hdusr@Node1 ch5]$ ./omp_for2
First:j=1,ThreadId=1
First:j=0,ThreadId=0
First:j=3,ThreadId=3
First:j=2,ThreadId=2
Second:j=2,ThreadId=2
Second:j=0,ThreadId=0
Second:j=3,ThreadId=3
Second:j=1,ThreadId=1
```


parallel for: for指令与parallel指令组合起来形成的指令。

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

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

```
int main(int argc, char* argv){
```

```
    int j=0;
```

```
    #pragma omp parallel for num_threads(4)
```

```
        for(j=0;j<4;j++)
```

```
            printf("j=%d,ThreadId=%d\n",j,omp_get_thread_num());
```


```
    return 0;
```

```
}
```

```
[hdusr@Node1 ch5]$ ./omp_for3  
j=3,ThreadId=3  
j=0,ThreadId=0  
j=1,ThreadId=1  
j=2,ThreadId=2
```

可见，循环被分配到四个不同的线程中执行。

```
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;
```

注意： $\text{approx} += f(a+i*h)$ 是一个临界区

需要将 approx 作为一个规约变量



- 在parallel指令中，所有变量的缺省作用域是共享的。
- 在parallel for中，由其并行化的循环中，循环变量的缺省作用域是私有的。

在使用for指令和parallel for指令时，对for循环语句书写是有约束条件的，即for循环圆括号内的语句要按照一定的规范进行书写。

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

□ for循环构造的一些限制如下：

- ✓ 变量index必须是整型或指针类型。
- ✓ 表达式start、end和incr必须是兼容类型。
- ✓ 表达式start、end和incr不能够在循环执行期间改变，也就是迭代次数必须是确定的。
- ✓ 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。
- ✓ for语句中不能含有break。

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



note 2 threads



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



1 1 2 3 5 8 13 21 34 55


this is correct

but sometimes
we get this

1 1 2 3 5 8 0 0 0 0



- 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，而是由程序员来识别这些依赖关系。
- 一个或更多个迭代结果依赖于其它迭代循环，一般不能被OpenMP正确的并行化。



OpenMP的 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;
```

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;
```

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

```
# pragma omp parallel for num_threads(thread_count) \  
    reduction(+: sum)  
for (i = 0; i < n; i++) {  
    factor = (i % 2 == 0) ? 1.0 : -1.0;  
    sum += factor/(2*i+1);  
}
```

□ 两个线程, $n=1000$, 那么运行结果:


- With $n=1000$ terms and 2 threads,
Our estimate of pi = 2.97063289263385
- With $n=1000$ terms and 2 threads,
Our estimate of pi = 3.22392164798593

□ 只用一个线程运行程序:

With $n=1000$ terms and 1 threads,
Our estimate of pi = 3.14059265383979

- 在private子句内列举的变量，在每个线程上都有一个私有副本被创建。
- 即使并行区域外有同名的共享变量，其在并行区域内不起任何作用，且并行区域也不会操作到外面的共享变量

```
# 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);
  }
```



Insures factor has
private scope.

- 出现在reduction子句中的参数不能出现在private子句中。
- 用private子句声明的私有变量的初始值在parallel块或parallel for块的开始处是未指定的，且在完成之后也是未指定的。

```
int main(int argc, char* argv[]){  
    int j=1;  
    #pragma omp parallel for num_threads(4) private(j)  
        for(j=0;j<4;j++)  
            printf("j=%d,ThreadId=%d\n",j,omp_get_thread_num());  
    printf("Last j = %d\n",j);  
    return 0;  
}
```

思考：有无private(j)结果如何？

for循环前的变量j和循环区域内的变量j其实是两个不同的变量。

```
[hdusr@Node1 ch5]$ ./omp_for4  
j=3,ThreadId=3  
j=0,ThreadId=0  
j=1,ThreadId=1  
j=2,ThreadId=2  
Last j = 1
```



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

int main(int argc, char* argv[]){
    int x=5;
    #pragma omp parallel num_threads(4) private(x)
    {
        printf("Thread %d > before initialization, x=%d\n",omp_get_thread_num(),x);
        x = 2*2+2;
        printf("Thread %d > after initialization,x=%d\n",omp_get_thread_num(),x);
    }
    printf("Last x = %d\n",x);
    return 0;
}
```

它不会继承同名共享变量的值

```
[hdusr@Node1 ch5]$ ./omp_private
Thread 3 > before initialization, x=0
Thread 3 > after initialization,x=6
Thread 0 > before initialization, x=0
Thread 0 > after initialization,x=6
Thread 1 > before initialization, x=0
Thread 1 > after initialization,x=6
Thread 2 > before initialization, x=0
Thread 2 > after initialization,x=6
Last x = 5
```

- 通常需要考虑在parallel块或parallel for块中每个变量的作用域。
- 与其让OpenMP决定每个变量的作用域，还不如让程序员明确块中每个变量的作用域。

default子句用来允许用户控制并行区域中变量的共享属性，用法如下：

`default(shared|none)`

使用shared时，缺省情况下，传入并行区域内的同名变量被当做共享变量来处理，不会产生线程私有副本，除非使用private等子句来指定某些变量为私有的才会产生副本。

如果使用none作为参数，那么线程中用到的变量必须显式指定是共享变量还是私有变量，那些有明确定义的除外。

shared子句用来声明一个或多个变量是共享变量，用法如下：

shared(list)

需要注意的是：在并行区域内使用共享变量时，如果存在写操作，必须对共享变量加以保护；否则，不要轻易使用共享变量，尽量将共享变量的访问转化为私有变量的访问。

```
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);
}
```

□指令：

- ✓ critical: 用在一段代码临界区之前，临界区再同一时间内只能有一个线程执行它；（保护临界区）
- ✓ for: 使for循环被多个线程并行执行；
- ✓ parallel for: 使for循环的代码被多个线程并行执行。

□子句

- ✓ reduction: 用来对一个或多个参数条目指定一个操作符。
- ✓ private: 用来声明一个或多个变量是私有变量
- ✓ shared: 用来声明一个或多个变量是共享变量
- ✓ default: 用来允许用户控制并行区域中变量的共享属性

1. 采用Local_trap改写梯形积分的OpenMP并行程序。
2. 采用规约操作改写梯形积分的OpenMP并行程序。
3. 采用parallel for指令改写梯形积分的OpenMP并行程序。
4. 采用OpenMP实现 π 值估计



结束!

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

int main(int argc, char* argv[]){
int j=1;
#pragma omp parallel for num_threads(4) default(shared)
  for(j=0;j<4;j++)
    printf("j=%d,ThreadId=%d\n",j,omp_get_thread_num());
printf("Last j = %d\n",j);
return 0;
}
```

```
[hdusr@Node1 ch5]$ ./omp_for4
j=1,ThreadId=1
j=0,ThreadId=0
j=3,ThreadId=3
j=2,ThreadId=2
Last j = 1
```

```
[hdusr@Node1 ch5]$ ./omp_for4
j=3,ThreadId=3
j=0,ThreadId=0
j=1,ThreadId=1
j=2,ThreadId=2
Last j = 1
```

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

int main(int argc, char* argv[]){
    int j=1;
    #pragma omp parallel for num_threads(4) default(none)
        for(j=0;j<4;j++)
            printf("j=%d,ThreadId=%d\n",j,omp_get_thread_num());
    printf("Last j = %d\n",j);
    return 0;
}
```

```
# pragma omp parallel for num_threads(thread_count) \
    default(none) reduction(+: sum) private(factor)
for (i = 0; i < n; i++) {
    factor = (i % 2 == 0) ? 1.0 : -1.0;
    sum += factor/(2*i+1);
#   ifdef DEBUG
#   printf("Thread %d > i = %lld, my_sum = %f\n", my_rank, i, my_sum);
#   endif
}
```

```
[hdusr@Node1 ch5]$ gcc -g -Wall -fopenmp -o omp_pi omp_pi.c
omp_pi.c: In function 'main':
omp_pi.c:40:11: error: 'n' not specified in enclosing parallel
#   pragma omp parallel for num_threads(thread_count) \
    ^
omp_pi.c:40:11: error: enclosing parallel
```

```
# pragma omp parallel for num_threads(thread_count) \
    default(none) reduction(+: sum) private(factor) shared(n)
for (i = 0; i < n; i++) {
    factor = (i % 2 == 0) ? 1.0 : -1.0;
    sum += factor/(2*i+1);
#   ifdef DEBUG
#   printf("Thread %d > i = %lld, my_sum = %f\n", my_rank, i, my_sum);
#   endif
}
```

总结：parallel for 中缺省循环变量为私有作用域的
权限大于default(none)或default(shared)的设置权限