并行计算

（Parallel Computing）

共享内存编程 - OpenMP

学习内容：

- 使用 OpenMP 编写程序
- 使用 OpenMP 并行化 for 循环
- 任务并行
- 显式的线程同步
- 共享内存编程中的标准问题

1.使用 OpenMP 编写程序

- OpenMP
  - 共享内存并行编程 API
  - MP = multiprocessing
  - 系统被看作是 CPU 或者 cores 的集合，它们都可以访问主存
  - 系统中的线程或者进程都可以访问所有可用的内存

1.使用 OpenMP 编写程序

- OpenMP

Date	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5
Nov 2018	OpenMP 5.0

1.使用 OpenMP 编写程序

- Pthreads vs. OpenMP
  - 都是共享内存编程的 API
  - Pthreads：要求程序员显式地指定每个线程的行为
  - OpenMP：有时允许程序员简单地声明一段代码应该并行执行，而任务的划分以及由哪个线程来执行，留给编译器和运行时系统决定

1.使用 OpenMP 编写程序

- Pthreads vs. OpenMP
  - Pthreads：和MPI一样，是一个可以链接到 C 程序的函数库，因此只要系统有 Pthreads 库，任何 Pthreads 程序都可以与任何 C 编译器一起使用
  - OpenMP：需要编译器对某些操作的支持，因此完全有可能在某个 C 编译器上无法将 OpenMP 程序编译为并行程序

1.使用 OpenMP 编写程序

- Pthreads vs. OpenMP
  - Pthreads：低级别的，它为我们提供了操作线程行为的能力。但需要程序员说明每个线程的实现细节
  - OpenMP：允许编译器和运行时系统确定线程行为的一些细节，因此使用OpenMP编写并行程序更为简单。但底层线程交互的细节难以编程

1.使用 OpenMP 编写程序

- Pragmas
  - OpenMP提供了基于指令的（“directives-based”）共享内存 API
  - pragma：特殊的预处理器指令，提供不属于 C 语言基本规范的行为
  - 不支持 pragma 的编译器忽略它们

#pragma

1.使用 OpenMP 编写程序

- Pragmas

#pragma compiler specific extension

```
#pragma once
// header file code
```

=

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_
/* code */
#endif
```

#pragma once 可以被一些主流的编译器支持，
包括：Clang, GCC, Intel C++ compiler and MSVC.

1.使用 OpenMP 编写程序

- Pragmas

```
#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 */
```

1.使用 OpenMP 编写程序

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

compiling

```
./omp_hello 4
```

running with 4 threads

possible outcomes

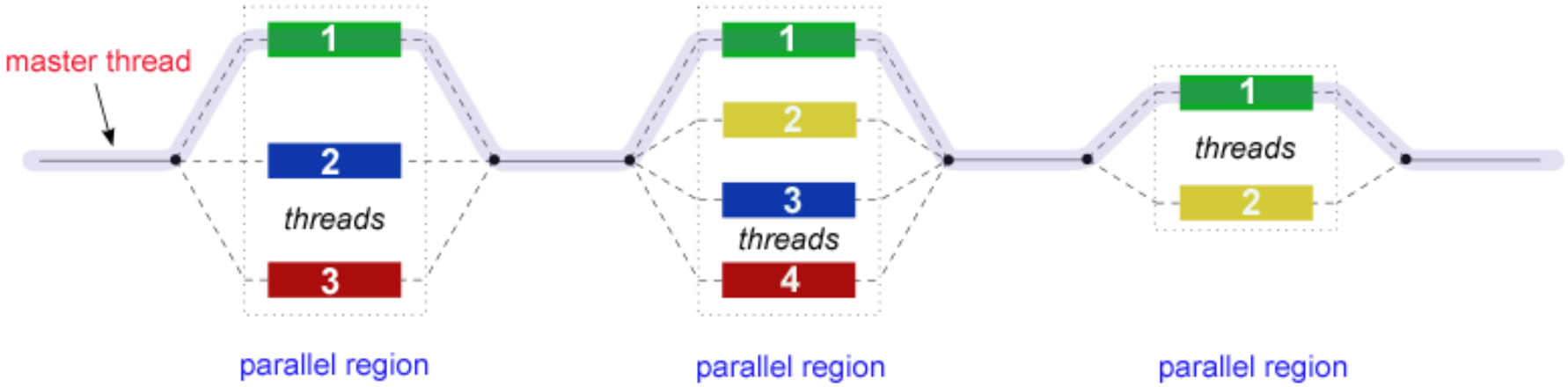
```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
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
```

1.使用 OpenMP 编写程序

- Pragmas
  - **# pragma omp parallel**
    - 最基本的并行指令
    - 运行紧跟着的结构化代码块的线程数由运行时系统决定



1.使用 OpenMP 编写程序

- Clause （从句）
  - 修改指令的文本
  - num_threads 从句可以添加到 parallel 指令中
  - 它允许程序员说明执行后面代码块的线程数量

```
# pragma omp parallel num_threads ( thread_count )
```

1.使用 OpenMP 编写程序

- 程序到达 parallel 指令时发生了什么？
  - 在 parallel 指令之前，程序为一个单独线程
  - 当到达 parallel 指令后，原始线程继续执行，额外的 （thread_count -1）个线程被启动
  - 在 OpenMP 的术语中，执行 parallel 块的线程集合（原始线程和新线程）称为一个团队（***team***），原始线程称为主线程（***master***），其他线程称为从线程（***slaves***）
  - team 中的每个线程执行 parallel 指令后的代码块，在我们的例子中，为 Hello 函数

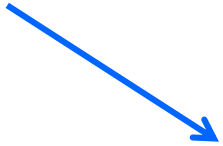
1.使用 OpenMP 编写程序

- 程序到达 parallel 指令时发生了什么？
  - 当代码块执行结束后（在我们的例子中，当每个线程从 Hello 函数返回），有一个隐式的屏障（implicit barrier）
  - 这意味着先完成代码块的线程将等待 team 中的其他线程
  - 当所有的线程完成代码块，slave 线程将终止，master 线程将继续执行代码块后面的代码（在我们的例子中，master 线程将执行 Hello 函数后面的 return 语句，程序终止）

1.使用 OpenMP 编写程序

- 对于不支持 OpenMP 的编译器

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



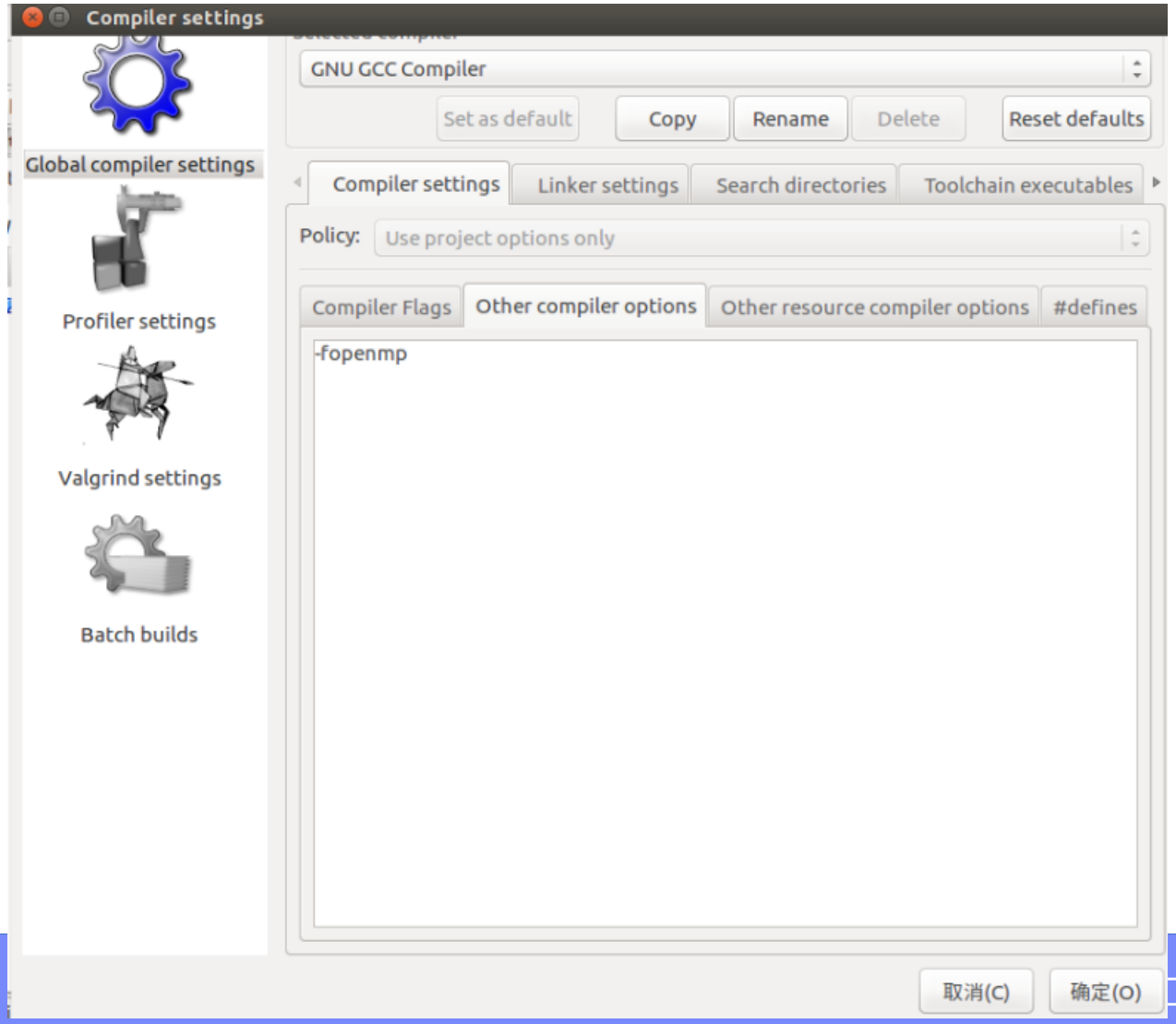
```
#ifdef _OPENMP
# include <omp.h>
#endif
```

1.使用 OpenMP 编写程序

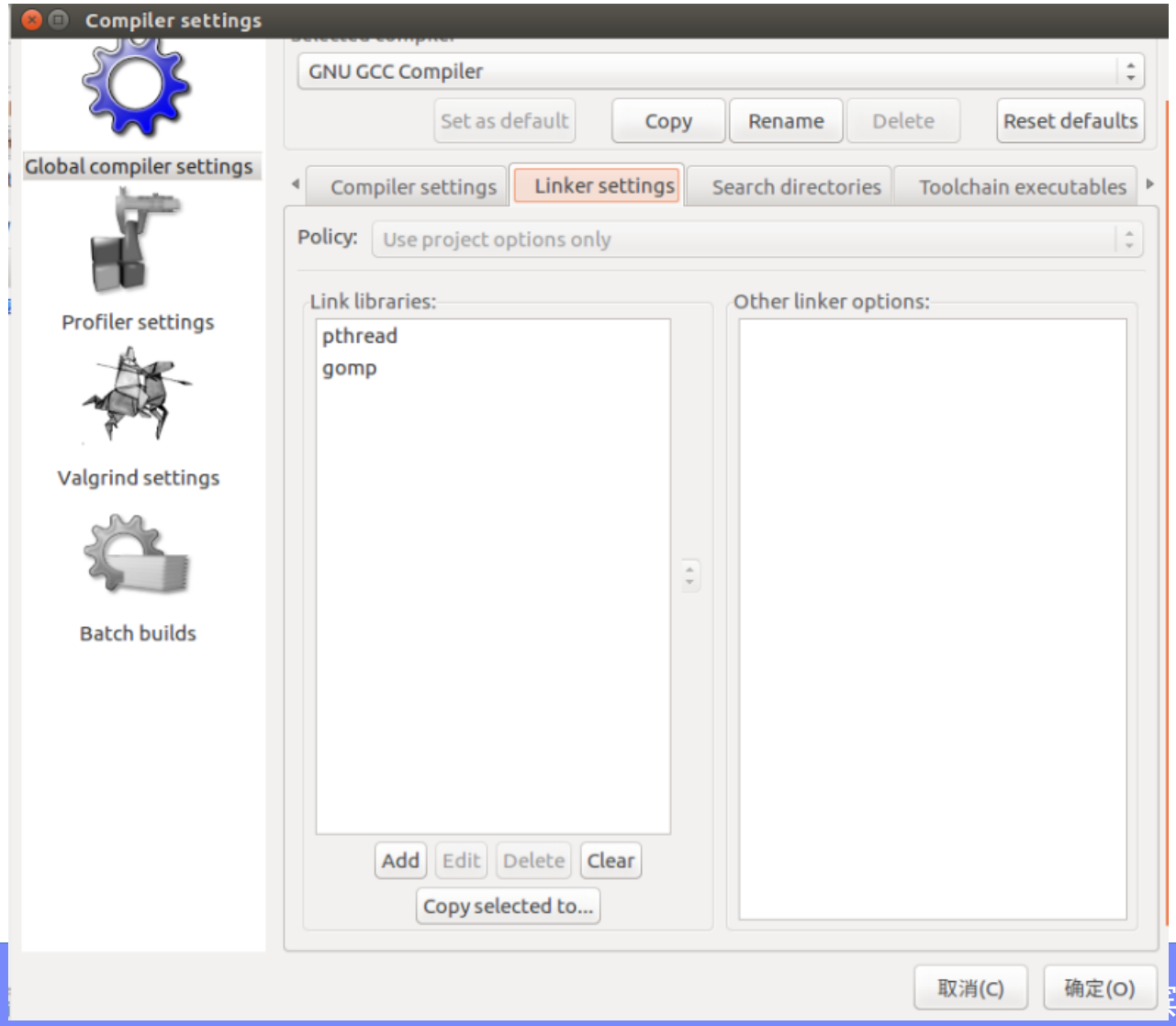
- 对于不支持 OpenMP 的编译器

```
# 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
```

Linux/Ubuntu下配置Code::blocks



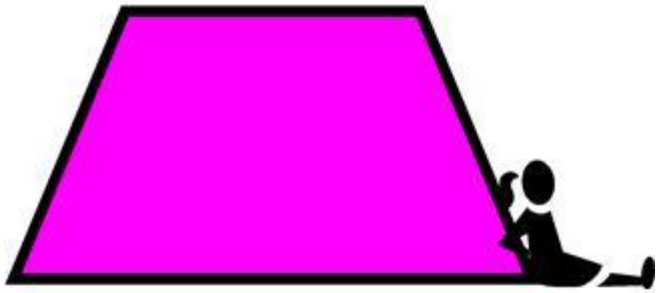
Linux/Ubuntu下配置Code::blocks



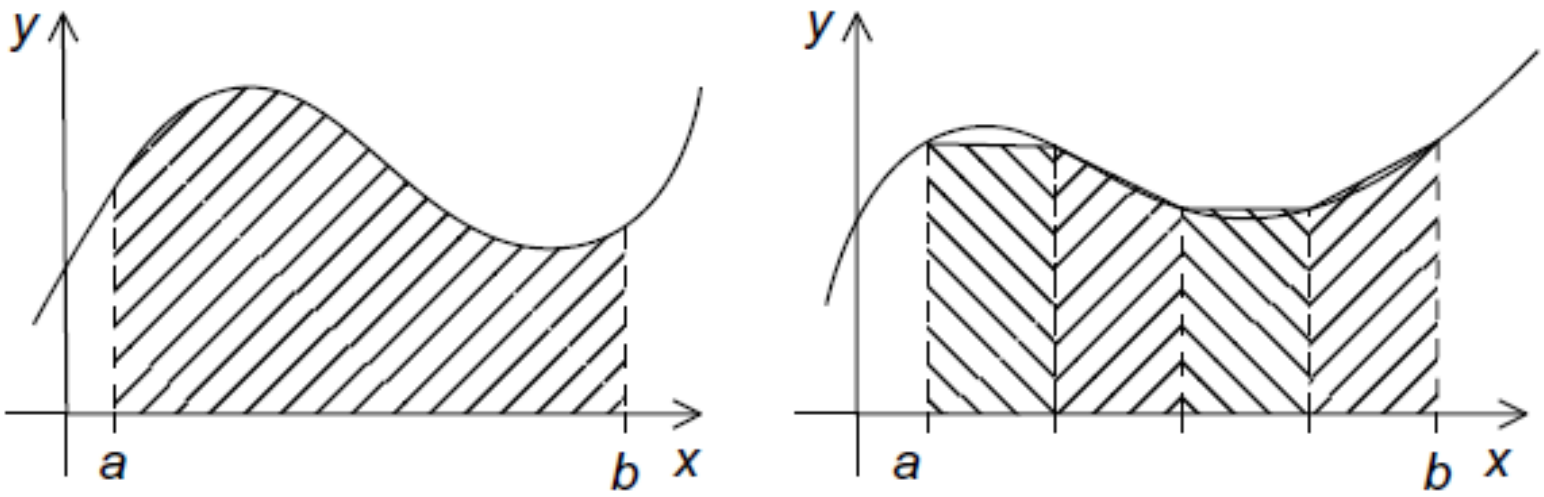
Visual Studio下配置OpenMP

https://blog.csdn.net/qq_35012681/article/details/80004509

2.梯形法则



2.梯形法则



(a) (b)

2.梯形法则

●串行算法 $h[f(x_0)/2+f(x_1)+f(x_2)+\cdots+f(x_{n-1})+f(x_n)/2]$

```
/* 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;
```

2.梯形法则

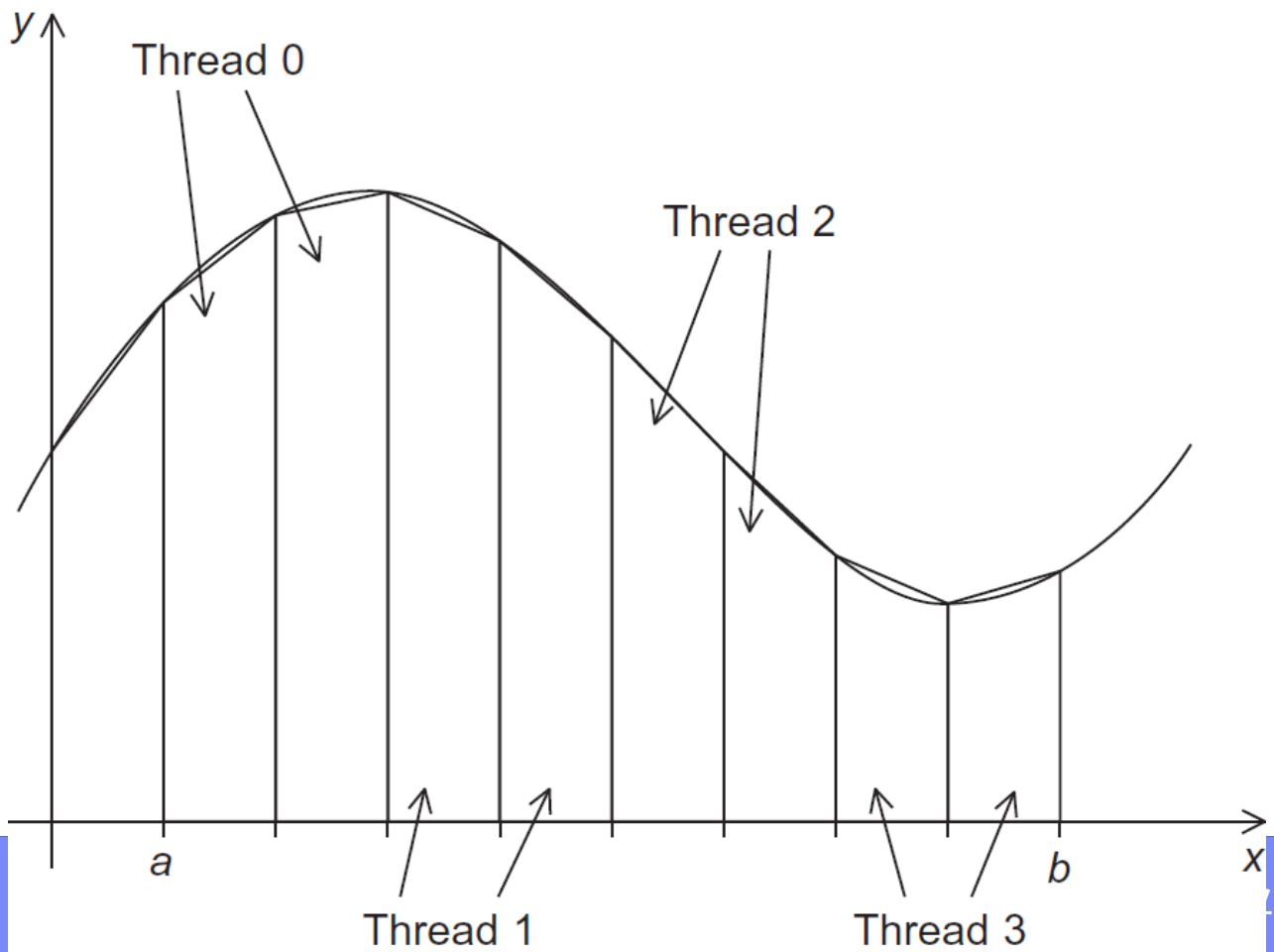
- 第一个 OpenMP 版本
  - 应用 Foster 并行程序设计方法
    - 确定两类任务
      a. 计算单个梯形的面积
      b. 累加梯形的面积
    - 第一类任务之间不需要通信，但第一类的任务都需要与第二类任务通信

2.梯形法则

- 第一个 OpenMP 版本
  - 应用 Foster 并行程序设计方法
    - 假设梯形数量多于 core 的数量
    - 通过给每个线程分配一个连续的梯形块（给每个核心分配一个线程）来聚合任务

2.梯形法则

●第一个 OpenMP 版本



2.梯形法则

- 第一个 OpenMP 版本
  - 应用 Foster 并行程序设计方法
    - 需要累加每个线程的结果
    - *每个线程执行* `global_result += my_result;` *?*

2.梯形法则

●第一个 OpenMP 版本

Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

当两个或多个线程试图同时执行下面的语句，结果不可预测

global_result += my_result ;

2.梯形法则

- 第一个 OpenMP 版本：临界区域（critical section）
  - Pthreads：mutexes 或 semaphores
  - OpenMP：critical 指令

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

一次只能有一个线程执行后面的代码块

2.梯形法则

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <omp.h>
4
5   void Trap(double a, double b, int n, double* global_result_p);
6
7   int main(int argc, char* argv[]) {
8      double  global_result = 0.0;
9      double  a, b;
10     int     n;
11     int     thread_count;
12
13     thread_count = strtol(argv[1], NULL, 10);
14     printf("Enter a, b, and n\n");
15     scanf("%lf %lf %d", &a, &b, &n);
16  #  pragma omp parallel num_threads(thread_count)
17     Trap(a, b, n, &global_result);
18
19     printf("With n = %d trapezoids, our estimate\n", n);
20     printf("of the integral from %f to %f = %.14e\n",
21        a, b, global_result);
22     return 0;
23  }  /* main */
24
```

2.梯形法则

```
25  void Trap(double a, double b, int n, double* global_result_p) {
26     double  h, x, my_result;
27     double  local_a, local_b;
28     int  i, local_n;
29     int my_rank = omp_get_thread_num();
30     int thread_count = omp_get_num_threads();
31
32     h = (b-a)/n;
33     local_n = n/thread_count;
34     local_a = a + my_rank*local_n*h;
35     local_b = local_a + local_n*h;
36     my_result = (f(local_a) + f(local_b))/2.0;
37     for (i = 1; i <= local_n-1; i++) {
38       x = local_a + i*h;
39       my_result += f(x);
40     }
41     my_result = my_result*h;
42
43  #  pragma omp critical
44     *global_result_p += my_result;
45  }  /* Trap */
```

2.梯形法则

- 变量作用域范围（Scope of variables）
  - 在串行编程中，变量的作用域：程序其他部分是否可以使用该变量
  - 在 OpenMP 中，变量的作用域： 可以被线程访问到的变量
    - team中所有线程都可以访问的变量具有共享作用域（shared scope）
    - 只能由单个线程访问的变量具有私有作用域（private scope）
    - 在 parallel 代码块前声明的变量默认的作用域为 shared
    - 变量的默认作用域可以被OpenMP中的指令所修改

2.梯形法则

- Reduction 从句
  - 在梯形法则的串行实现中：

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

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

  - 在并行版本中，为了累加每个线程的结果，获得 *global_result*

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

2.梯形法则

- Reduction 从句
  - 如果采用和串行版本相似的形式

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

  - 可以做如下修正：

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

  - *问题：强制每个线程顺序执行！*

2.梯形法则

- Reduction 从句
  - 可以通过声明私有变量，在函数调用后进入临界区来避免上面的问题

```
    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;
    }
```

2.梯形法则

- Reduction 从句
  - reduction 运算符：二元运算符（如：加法、乘法）
  - reduction 是将相同的 reduction 运算符重复应用于操作数序列以获得单个结果的运算
  - 操作的所有中间结果都存储在同一个变量中：reduction 变量

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

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

2.梯形法则

- Reduction 从句
  - reduction 从句可以添加到 parallel 指令中

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

  - global_result 为 reduction 变量，reduction 运算符为“+”

2.梯形法则

- Reduction 从句

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

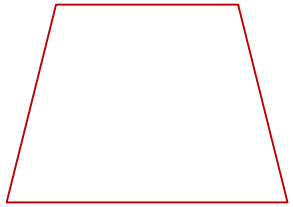
```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */

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

3.parallel for 指令

- parallel for 指令

```
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;
```

3.parallel for 指令

- parallel for 指令
  - 生成一组线程来执行后面的代码块
  - parallel for 后面的代码块必须为 for 循环
  - 使用 parallel for 指令，系统将循环分配给各个线程，来完成对 for 循环的并行化
  - 大多数系统采用块划分，假设有 m 次迭代，将第一个 m / thread_count 次迭代分配给线程0，下一个 m / thread_count 次迭代分配给线程1，…
  - parallel for 指令后的循环中，循环变量默认为私有变量

3.parallel for 指令

●parallel for 指令

- OpenMP 只能并行化 for 循环，不能并行化 whie 或 do-while 循环
- OpenMP 只能并行化循环次数确定的 for 循环

```
for ( ; ; ) {
    . . .
}
```

```
for (i = 0; i < n; i++) {
    if ( . . . ) break;
    . . .
}
```

3.parallel for 指令

- 可以被并行化的 for 语句的合法形式

```
                                        index++
                                        ++index
                     index < end        index--
                     index <= end       --index
for  index = start ; index >= end ;     index += incr
                     index > end        index -= incr
                                        index = index + incr
                                        index = incr + index
                                        index = index - incr
```

3.parallel for 指令

- 可以被并行化的 for 语句的合法形式
  - 变量 index 必须为整型或指针类型
  - 表达式 start，end，incr 必须为兼容类型。如：index 为指针类型，则 incr 必须为整型
  - 表达式 start，end，incr 在循环执行期间不能改变
  - 在循环执行期间，变量 index 只能被 for 语句中的增量表达式所改变

3.parallel for 指令

●parallel for 指令

```
1     int Linear_search(int key, int A[], int n) {
2        int i;
3        /* thread_count is global */
4     #  pragma omp parallel for num_threads(thread_count)
5        for (i = 0; i < n; i++)
6           if (A[i] == key) return i;
7        return -1;  /* key not in list */
8     }
```

➢gcc 编译器报告：

```
Line 6: error: invalid exit from OpenMP structured block
```

3.parallel for 指令

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

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

3.parallel for 指令

- What happened？
  - *数据依赖！（Data dependencies）*
  - OpenMP 编译器不检查被 parallel for 指令并行化的循环中，迭代之间的依赖性
  - 一个或多个迭代的结果依赖于其他迭代的循环通常不能被 OpenMP 正确地并行化

3.parallel for 指令

- 发现循环携带的依赖（loop-carried dependences）

```
for (i = 0; i < n; i++) {
   x[i] = a + i*h;
   y[i] = exp(x[i]);
}
```

有问题？

```
# pragma omp parallel for num_threads(thread_count)
for (i = 0; i < n; i++) {
   x[i] = a + i*h;
   y[i] = exp(x[i]);
}
```

3.parallel for 指令

- 估计 π

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

```
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;
```

3.parallel for 指令

- 估计 π：OpenMP 版本 1

loop dependency

```
        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;
```

3.parallel for 指令

- 估计 π：OpenMP 版本 2

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
sum += factor/(2*k+1);
factor = -factor;
```

↓

```
if (k % 2 == 0)
   factor = 1.0;
else
   factor = -1.0;
sum += factor/(2*k+1);
```

3.parallel for 指令

- 估计 π：OpenMP 版本 2

What's wrong ?

```
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
```

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

3.parallel for 指令

- 估计 π：OpenMP 版本 2
  - 除了循环变量外，在循环前声明的变量为线程共享变量
  - factor 为共享变量

```
        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;
```

3.parallel for 指令

- 估计 π：OpenMP 版本 3

```
        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.

3.parallel for 指令

- default 从句
  - 允许程序员指定块中每个变量的作用域

```
default(none)
```

  - 使用 default 从句，编译器将要求我们指定在块中使用的在块外声明的每个变量的作用域

3.parallel for 指令

- default 从句

```
        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);
        }
```