COMP4007: 并行处理和体系结构

第四章: 基于OpenMP的并行编程

授课老师：王强、施少怀
助　　教：刘虎成、田超

哈尔滨工业大学（深圳）

大纲

- 基于OpenMP的并行编程

- OpenMP简介
  - 创建并行区域
  - 并行循环
  - 同步
  - 数据共享

C++多线程

- 针对多任务的多线程使你的计算机可以同时运行两个或更多程序

- 用法
  - #include <pthread.h>
  - pthread_create (thread, attr, start_routine, arg)
    - thread: an opaque, unique identifier for the new thread
    - attr: an opaque attribute object that may be used to set thread attributes
    - start_routine: the C++ routine that the thread will execute once it is created
    - arg: Aa single argument that may be passed to start_routine
  - pthread_exit (status)

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
struct thread_data {
  int  thread_id;
  char *message;
};
void *PrintHello(void *threadarg) {
  struct thread_data *my_data;
  my_data = (struct thread_data *) threadarg;
  cout << "Thread ID : " << my_data->thread_id ;
  cout << " Message : " << my_data->message << endl;
  pthread_exit(NULL);
}

int main () {
  pthread_t threads[NUM_THREADS];
  struct thread_data td[NUM_THREADS];
  int rc, i;
  for( i = 0; i < NUM_THREADS; i++ ) {
    cout <<"main() : creating thread, " << i << endl;
    td[i].thread_id = i;
    td[i].message = "This is message";
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *)&td[i]);
    if (rc) {
      cout << "Error:unable to create thread," << rc << endl;
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

常用 Linux 命令

cmd	Description
ping	检查与一个服务器的连通情况
ssh	登录远程服务器
scp	与服务器传递文件

cmd	Description
pwd	当前工作路径
cd	进入一个目录 $cd folder; $cd - $cd $cd ..
ls	列出路径下的项目

cmd	Description
cat	打印文件的内容
cp	复制 $cp file.txt file-copy.txt; $cd -r folder folder-copy;
mv	改变文件路径
mkdir	创建一个新的目录 $mkdir new-directory $mkdir -p d1/d2/d3
rm	删除 $rm file.txt $rm -r d1
touch	创建新的空白文件
find	搜索文件或目录 $find . -name notes.txt $find / -type d -name notes
grep	在输出中搜索过滤关键词 $grep blue notepad.txt

cmd	Description
kill	结束一个进程
wget	下载文件
top	显示当前活动的进程
lscpu	显示 CPU 信息
df	检查磁盘信息 $df -h
ifconfig	显示网络配置
free	显示系统内存 $free -h
vi	Unix/Linux 的默认文本编辑器

Ubuntu is easy to use!

http://mally.stanford.edu/~sr/computing/basic-unix.html

什么是OpenMP

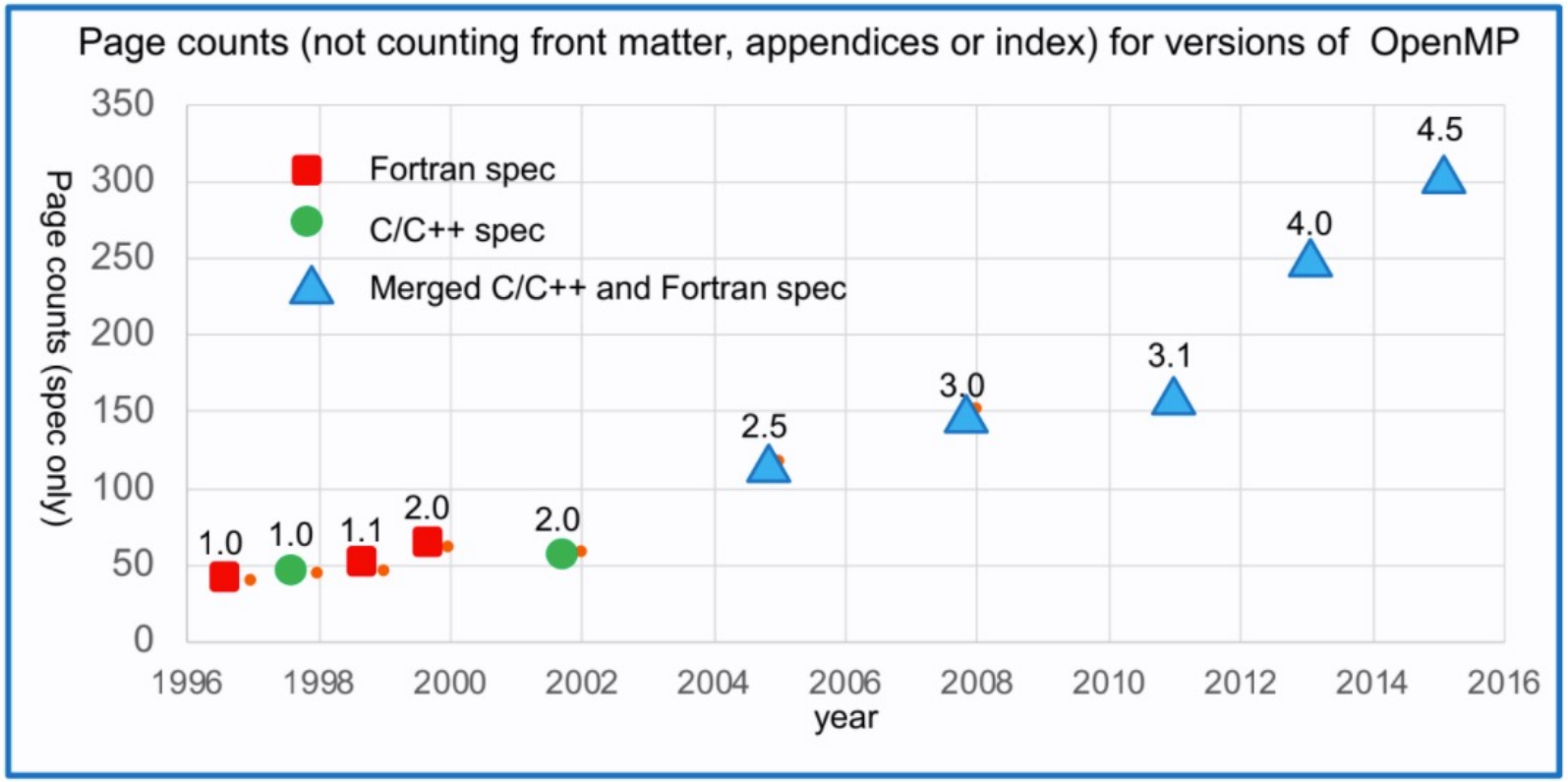
- OpenMP = 多进程开放式规范(Open specification for Multi-Processing)
  - www.openmp.org – 课程, 示例, 论坛, 等.
  - OpenMP 架构审查委员会(OpenMP Architecture Review Board, ARB)控制的规范
    - 控制 OpenMP 规范的非营利组织
    - 最新规范: OpenMP 5.2 (2021年11月)
- 动机: 致力于简化常规并行场景的编程
  - 针对每个线程或进程都有可能访问所有可用内存的系统而设计
- 基于线程的并行
  - OpenMP 程序仅通过线程来实现并行运行
  - 显式并行
    - 程序员可全面控制并行过程
- 数据域
  - 并行区域中的所有线程都能同时访问共享数据

程序员眼中的 OpenMP

- OpenMP 是一种可移植、线程式、共享内存的编程规范，采用“轻型”语法
  - 需要编译器支持(C, C++ or Fortran)
- OpenMP可以
  - 允许将程序分为串行区域和并行区域，而不仅仅是多个并发执行的线程。
  - 隐藏堆栈管理
  - 提供同步构造
- OpenMP不可以
  - 自动并行
  - 保证加速
  - 避免数据访问竞争

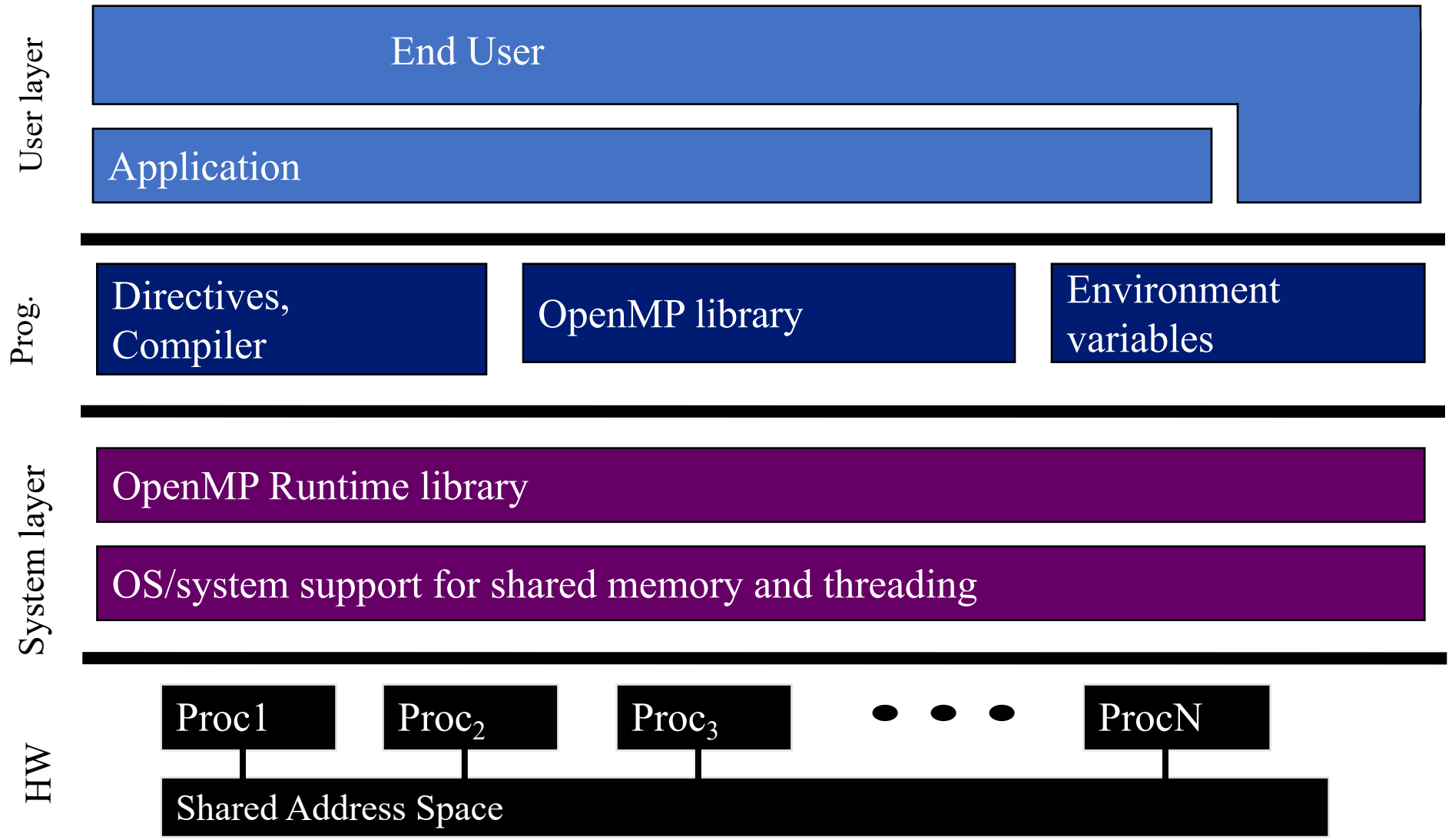
OpenMP的发展历程

- 1997 年诞生时只是一个简单的接口
- 经过多年发展，复杂性大大增加



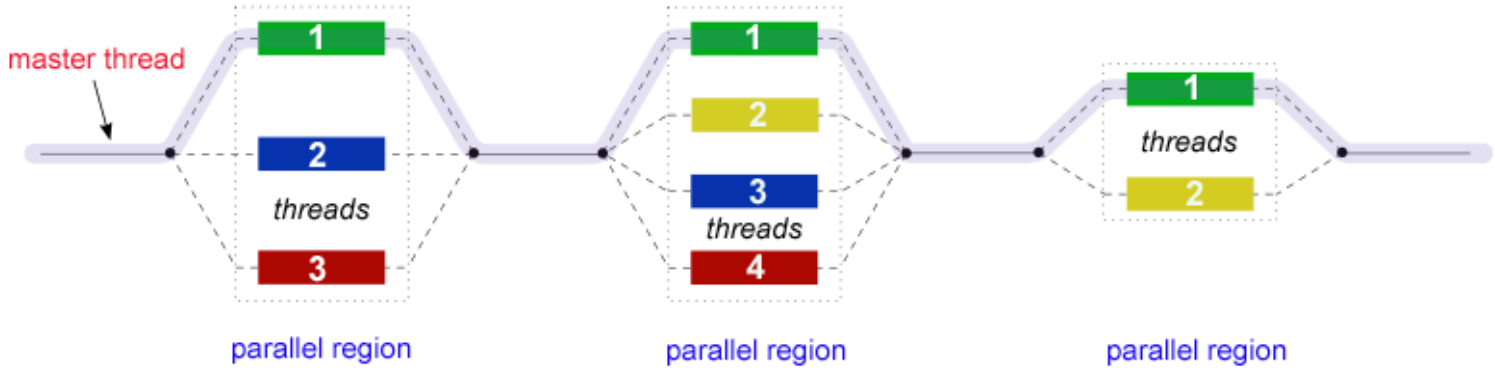
OpenMP 5.0 (Nov 2018) 规范文档有666页!

OpenMP基本定义: 基本解决方案栈



Fork - Join 模型

- OpenMP 程序以单个进程开始: 被称为主线程
  - FORK: 主线程创建的一组并行线程
  - JOIN: 主线程创建的并行线程执行完成并行区域中的语句时，会进行同步并终止运行，只留下主线程
- 一些术语
  - 原始线程和创建出的新线程被称为team
  - 原始线程被称为master
  - 其他线程被称为slaves
- 用于 C/C++ 和 Fortran 编程的高级应用程序接口
  - 预处理器（编译器）指令 (~ 80%)
    - #pragma omp directive-name [clause [clause ...]]
  - 库调用 (~ 19%)
    - #include <omp.h>
  - 环境变量 (~ 1%)
    - 全大写，添加到 srun 等。



OpenMP组件

指令(Directives)

- Parallel regions
- Work sharing
- Synchronization
- Data-sharing attributes
  - Private
  - firstprivate
  - lastprivate
  - shared
  - reduction
- Orphaning

运行时环境 (Runtime environment)

- 线程数
- 线程ID
- 动态线程调节
- 嵌套并行
- 计时器
- 用于锁定的API

环境变量 (Environment variables)

- 线程数
- 调度类型
- 动态线程调节
- 嵌套并行

编译器指令和子句

- 编译器指令的语法如下

#pragma omp directive [clause [clause] …]

- 示例

```
#pragma omp parallel if( n > threshold) \
        shared(n,x,y) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++) x[i] += y[i];
} /* End of parallel region */
```

- 创建并行区域
- 将代码块划分给不同的线程
- 在不同线程之间分配循环迭代
- 序列化代码区域
- 工作线程间的同步

- if (scalar expression)
  - 当表达式求值为真时才并行执行
  - 否则，串行执行
- private(list)
  - 与原始对象没有存储关联
  - 所有引用均指向本地对象
  - 进入和退出时的值均未定义
- shared(list)
  - 团队中的所有线程都能访问数据
  - 所有线程访问相同的地址空间
- firstprivate(list)
  - 列表中所有变量的初始化值都是原始对象在进入并行区域之前的值
- lastprivate(list)
  - 按顺序执行最后一次迭代或并行区域的线程会更新列表中对象的值

运行时库例程

- 调用预定义函数
  - 示例
    - #include <omp.h>
    - int omp_get_num_threads(void)
- 设置和查询线程数
- 查询线程的唯一标识符（线程 ID）、线程祖先的标识符、线程团队大小
- 设置和查询动态线程功能
- 查询是否在并行区域以及在哪个级别
- 设置和查询嵌套并行性
- 设置、初始化和终止锁及嵌套锁
- 查询挂钟时间和精度

环境变量

- OpenMP 提供了多个用于在运行时控制并行代码执行的环境变量
  - 示例

csh/tcsh	**setenv OMP_NUM_THREADS 8**
sh/bash	**export OMP_NUM_THREADS=8**

- 设置线程数
- 指定循环迭代的划分方式
- 将线程绑定到处理器
- 启用/禁用嵌套并行；设置嵌套并行的最大级别
- 启用/禁用动态线程
- 设置线程堆栈大小
- 设置线程等待策略

OpenMP公共核心功能

- 最常用的19个功能

OpenMP pragma、函数或子句	**概念**
#pragma omp parallel	并行区域、线程团队、结构块、跨线程交错执行
int omp_get_thread_num() int omp_get_num_threads()	为并行区域创建线程，并使用线程数和线程 ID 划分工作
double omp_get_wtime()	加速、阿姆达尔定律、错误共享和其他性能问题
setenv OMP_NUM_THREADS N	内部控制变量。使用环境变量设置默认线程数
#pragma omp barrier #pragma omp critical	同步和竞赛条件。重温交错执行
#pragma omp for #pragma omp parallel for	分工、并行循环、循环携带依赖性
reduction(op:list)	减少整个团队线程中对应变量的数值
schedule(dynamic [,chunk]) schedule (static [,chunk])	循环调度、循环开销和负载均衡
private(list), firstprivate(list), shared(list)	数据环境
nowait	禁用工作共享结构上的隐含障碍、障碍的高成本和刷新操作（不包括刷新指令）
#pragma omp single	单线程工作共享
#pragma omp task #pragma omp taskwait	任务，包括任务的数据环境。

OpenMP – Hello World

```
#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 */
```

Runtime header

编译器指令

运行时例程

$gcc −fopenmp −o omp_hello omp_hello.c
$./omp_hello 4

OpenMP指令

- 格式

#pragma omp	directive-name	[clause, ...]	newline
所有OpenMP C/C++指令都需要.	有效的 OpenMP 指令。必须出现在 pragma 之后和任何子句之前。	**可选**。子句可按任何顺序排列，必要时可重复，除非另有限制。	**必须填写**。位于本指令所包含的结构块之前。

- 一般规则:
  - 区分大小写
  - 指令遵循 C/C++ 编译器指令标准的约定
  - 每个指令只能指定一个指令名称
  - 每条指令仅适用于一条后继语句，后继语句必须是一个结构块。
  - 在指令行末尾用反斜杠（"\"）转义换行符，可以在后续行中 "续写 "长指令行。
- 示例
  - #pragma omp parallel default(shared) private(beta,pi)

OpenMP指令

#pragma

- 特殊的预处理指令
- 用于支持不符合基本 C/C++ 规范的行为
- 不支持 pragma 的编译器会忽略它们

OpenMP指令

#pragma omp parallel [clause[[,] clause] …]

- 最基本的并行指令。
- 运行接下来的结构化代码块的线程数量由运行时系统决定。
- 支持以下子句
  - if (scalar expression)
  - private(list)
  - shared(list)
  - default(none/shared)
  - reduction(operator:list)
  - copyin(list)
  - firstprivate(list)
  - num_threads(scalar)

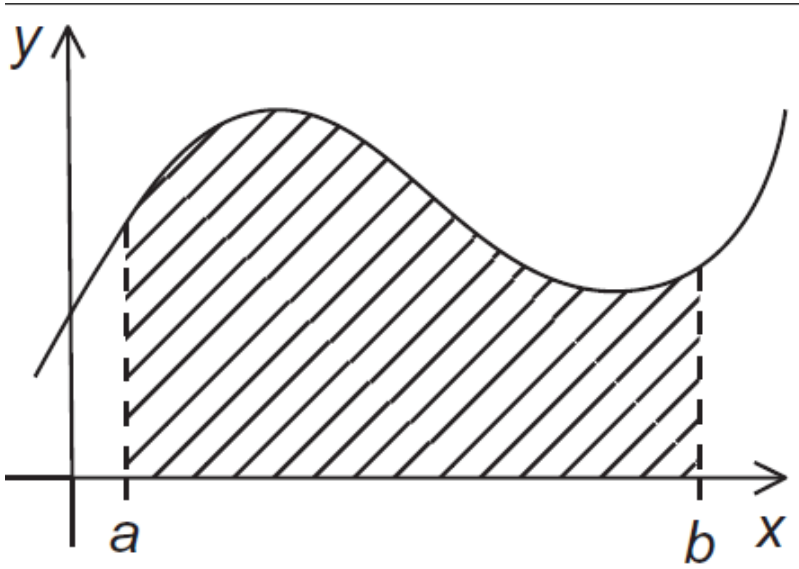
OpenMP指令

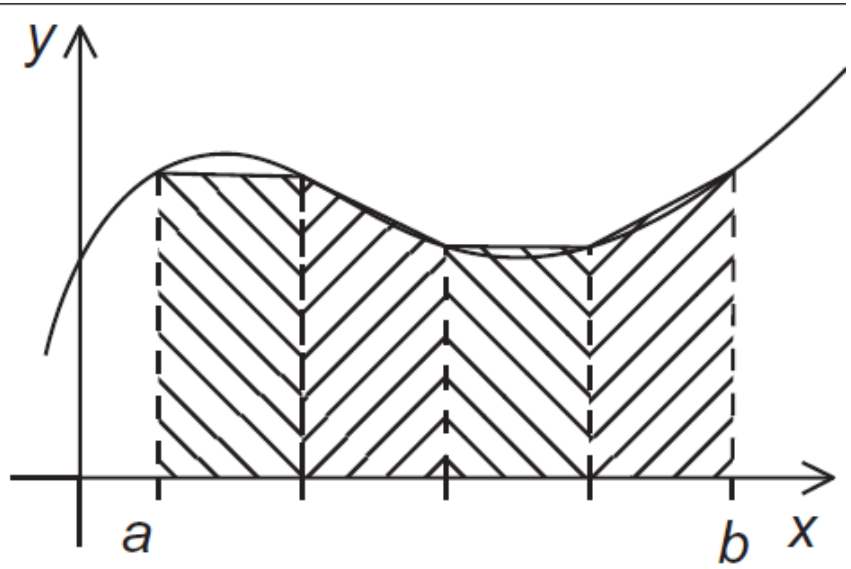
#pragma omp parallel num_threads (thread_count)

- 子句
  - 修改指令的文本。
  - 可以在一个并行指令中添加 num_threads子句。
  - 它允许程序员指定执行接下来代码块的线程数。

示例 - 梯形法则(The Trapezoidal Rule)

- 将总面积分成较小的梯形，而不是使用矩形来计算曲线与座标轴之间的面积





$$Area = \int_a^b f(x)dx$$

$$\Delta x = \frac{b-a}{n}$$

$$a = x_0 < x_1 < x_2 < x_3 < \cdots < x_n = b$$

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n))$$

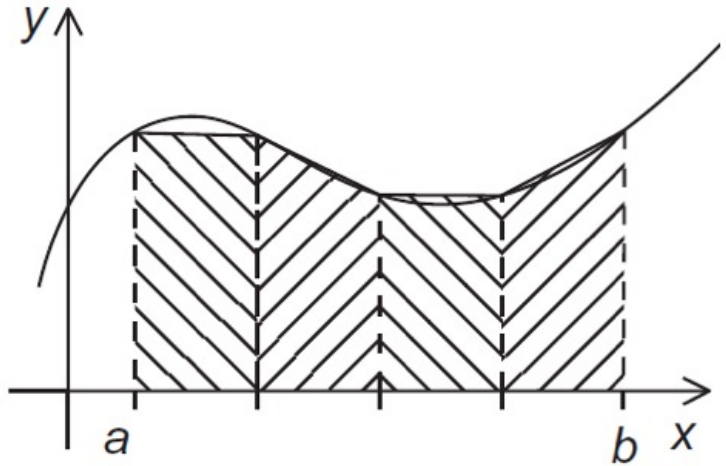
$$x_i = a + i\Delta x$$

梯形法则 – 串行算法

```
/* 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;
```

第一个基于OpenMP实现的版本

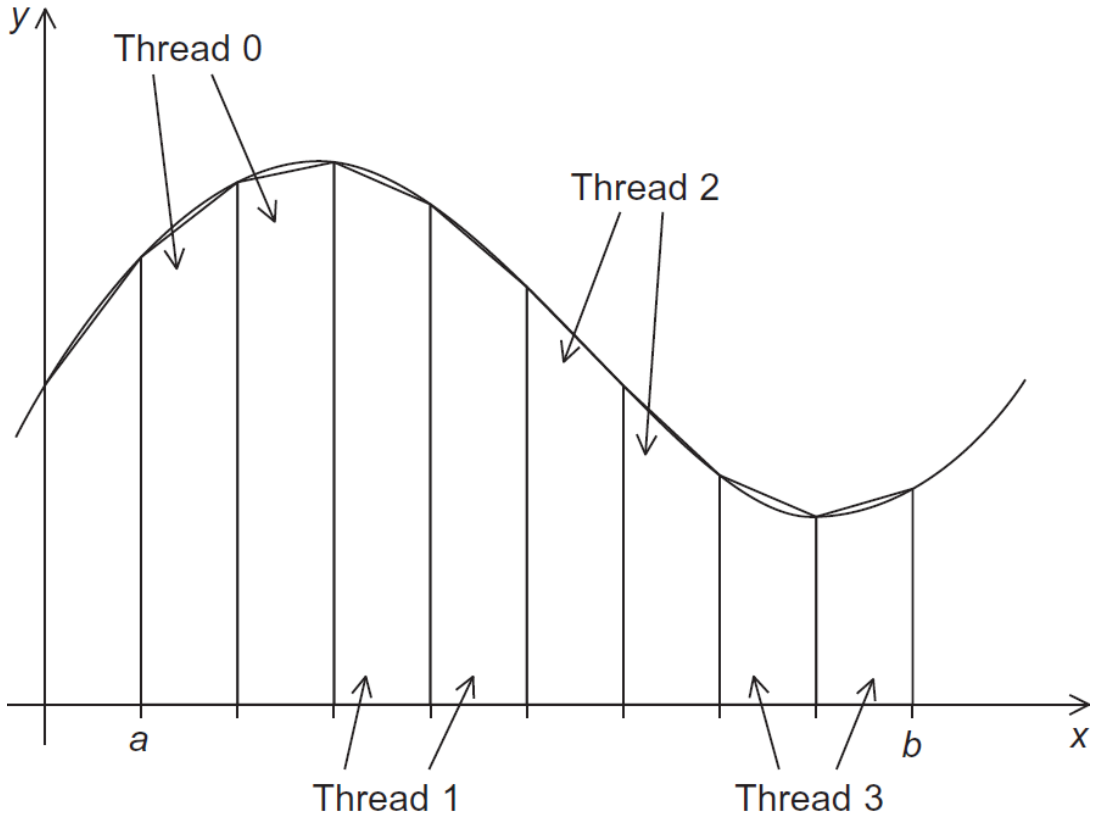
- 1) 我们确定了以下两类任务:
  - a) 计算各个梯形的面积
  - b) 累加所有梯形的面积
- 2) 第一类任务之间没有通信，但第一类任务的每个实例都与任务 1b 通信。




```
/* 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;
```

将梯形面积计算任务分配给线程

- 1.a) 计算各个梯形的面积
- 1.b) 累加所有梯形的面积



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`

- 两个（或更多）线程试图同时执行时会出现无法预测的结果：：

```
global_result += my_result ;
```

OpenMP指令- 相互排斥

- 临界区域
  - 一次只能有一个线程执行接下来的结构块

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

基于OpenMP的梯形法则程序

```c
#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 */
```

```c
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 */
```

变量作用域

- 作用域
  - 在串行编程中，变量的作用域为程序中可以使用该变量的部分
  - OpenMP中，变量的作用域指的是在并行块中可以访问该变量的线程集。
    - 团队中所有线程都能访问的变量具有共享作用域
    - 只能由单个线程访问的变量具有私有作用域
    - 在并行代码块之前声明的变量的默认作用域是共享的

变量作用域

我们需要一个更复杂的方法来累加每个线程的本地计算结果，从而得到 global_result 。

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

尽管我们更希望像下面这样。

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

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

变量作用域

如果按下面这种方式使用，就没有临界区了!

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

按下面的代码进行修改...

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

... 可强制线程按顺序执行

变量作用域

可以在并行代码块中声明一个私有变量，并将关键部分移到函数调用之后，从而规避临界区运行时间过长的问题。

```
   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;
   }
```

归约操作

- 归约操作是一种二元操作（如加法或乘法）。
- 归约运算是对操作数序列重复应用同一归约操作以得到单一结果的计算。
- 操作的所有中间结果都应存储在同一个变量中：归约变量。

OpenMP指令- Reduction Clause

- 并行指令支持归约子句

```
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);
```

OpenMP指令- Parallel For

- 让一组线程执行接下来的结构化代码块。
- parallel for 指令之后的结构块必须是 for 循环。
- 利用并行 for 指令后，系统会将 for 循环的迭代步分配给不风吹来线程，从而实现 for 循环的并行化。

OpenMP指令- Parallel For

```
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 for 语句的合法形式

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

- 变量索引必须是整数或指针类型（例如，不能是浮点型）。
- 表达式 start、end 和 incr 的类型必须能够兼容的类型。例如，如果 index 是指针，那么 incr 必须是整数类型。
- 表达式 start、end 和 incr在循环执行过程中不能更改。
- 在循环执行期间，只能通过 for 语句中的 "增量表达式 "修改变量索引。

数据依赖

- “parallel for” 指令不检查数据依赖
- 一般来说，一个或多个迭代结果依赖于其他迭代结果的循环无法通过 OpenMP 正确并行化

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

```
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 ];
```

注意有2个线程

有时会得到不正确的结果

1 1 2 3 5 8 13 21 34 55

正确的结果

1 1 2 3 5 8 0 0 0 0

估算 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;
```

基于OpenMP方案#1

```
        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;
```

循环依赖

基于OpenMP方案 #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);
        }
```

确保factor具有私有作用域

默认子句

- 要求程序员指定程序块中每个变量的作用域。

default(none)

- 使用该子句后，编译器会要求我们指定每个在代码块中使用的、在代码块外声明的变量的作用域。

```
        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);
        }
```

OpenMP指令- 循环调度

- 并行处理循环
  - 利用循环分区分配迭代步

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

Thread	Iterations
0	$0, n/t, 2n/t, \ldots$
1	$1, n/t+1, 2n/t+1, \ldots$
⋮	⋮
$t-1$	$t-1, n/t+t-1, 2n/t+t-1, \ldots$

```
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 */
```

- f(i) 调用 sin 函数 i 次
  - 假设执行 f(2i) 所需的时间约为执行 f(i) 所需的时间的两倍
- n=10,000
  - 单线程耗时约 3.67 秒

OpenMP 指令 - Schedule子句

- 默认方式

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

  - n = 10,000
    - 2个线程
    - 默认分配方式
    - 运行时间 = 2.76 秒
    - 加速比 = 1.33x

- 循环方式

```
        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);
```

  - n = 10,000
    - 2个线程
    - 循环分配
    - 运行时间 = 1.84秒
    - 加速比 = 1.99x

OpenMP指令- Schedule子句

- schedule (type , chunksize)
  - type取值:
    - static: 在执行循环之前将迭代步分配给线程。
    - dynamic 或 guided: 在循环执行过程中将迭代步分配给线程。
    - auto: 由编译器和/或运行时系统自动选择。
    - runtime: 调度在运行时确定。
  - chunksize: 表示块大小的正整数

Schedule子句 - Type

```
        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);
```

- 例如, thread_count=3
  - schedule(static, 1)
    - 线程0: 0, 3, 6, 9
    - 线程1: 1, 4, 7, 10
    - 线程2: 2, 5, 8, 11
  - schedule(static, 2)
    - 线程0: 0, 1, 6, 7
    - 线程1: 2, 3, 8, 9
    - 线程2: 4, 5, 10, 11

Schedule子句- Dynamic调度类型

- 整个迭代会被划分成包含chunksize个连续迭代步的块。
- 每个线程执行一个分块，当一个线程完成一个分块时，它会向运行时系统请求另一个分块。
- 这一过程一直持续到所有块(即迭代)完成。
- chunksize可省略。如果省略，则默认为1 。

Schedule子句- Guided调度类型

- 每个线程执行一个分块，当一个线程完成一个分块时，它会请求另一个分块。
- 然而，在guided调度中，随着分块的完成，新分块的大小会减小。
- 如果未指定chunksize，分块大小会减小到 1。
- 如果指定了chunksize，则分块大小会向下递减到 chunksize，但最后一个块可以小于chunksize。

Schedule子句- 运行时调度

- 系统使用环境变量 OMP_SCHEDULE 来决定运行时如何对循环进行调度。
- OMP_SCHEDULE 环境变量的值可用于static、dynamic或guided类型的调度。
  - $setenv OMP_SCHEDULE "guided,4"
  - $setenv OMP_SCHEDULE "dynamic"

OpenMP指令- Atomic

```
# pragma omp atomic
```

- 与临界指令不同，它只能保护由单个赋值语句组成的临界部分
- 声明必须采用以下形式之一

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

  - <op> 可以是以下二元运算符之一

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

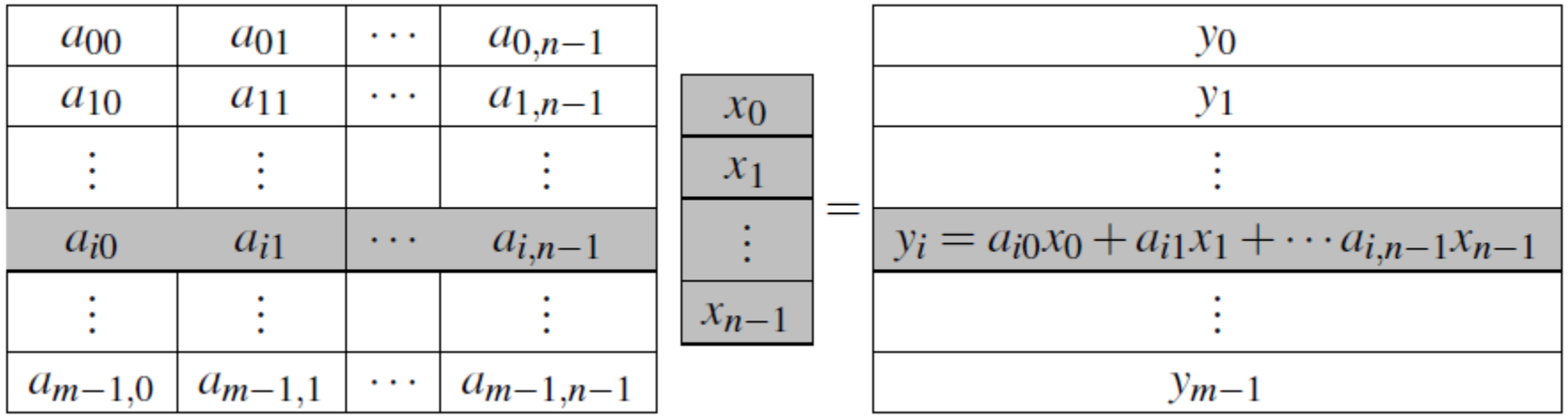
  - 许多处理器提供特殊的加载-修改-存储指令。
  - 相对于保护更一般临界, 使用Atomic指令可以更有效地保护只进行加载-修改-存储的临界区段

同步子句

- 高级别同步
  - critical
  - atomic
  - barrier
  - ordered
- 低级别同步
  - flush
  - locks

矩阵-向量乘法

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




```
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];
   }
```

阅读清单

- https://hpc-tutorials.llnl.gov/openmp/
- https://www.nersc.gov/users/training/events/openmp-common-core-february-2018/

总结

- OpenMP 是一种面向共享内存系统的编程标准
- OpenMP 既使用特殊函数，也使用称为注解的预处理器指令
  - 创建并行区域
  - 并行循环
  - 同步
  - 数据共享