并行计算

十四、共享存储系统并行编程

OpenMP编程简介

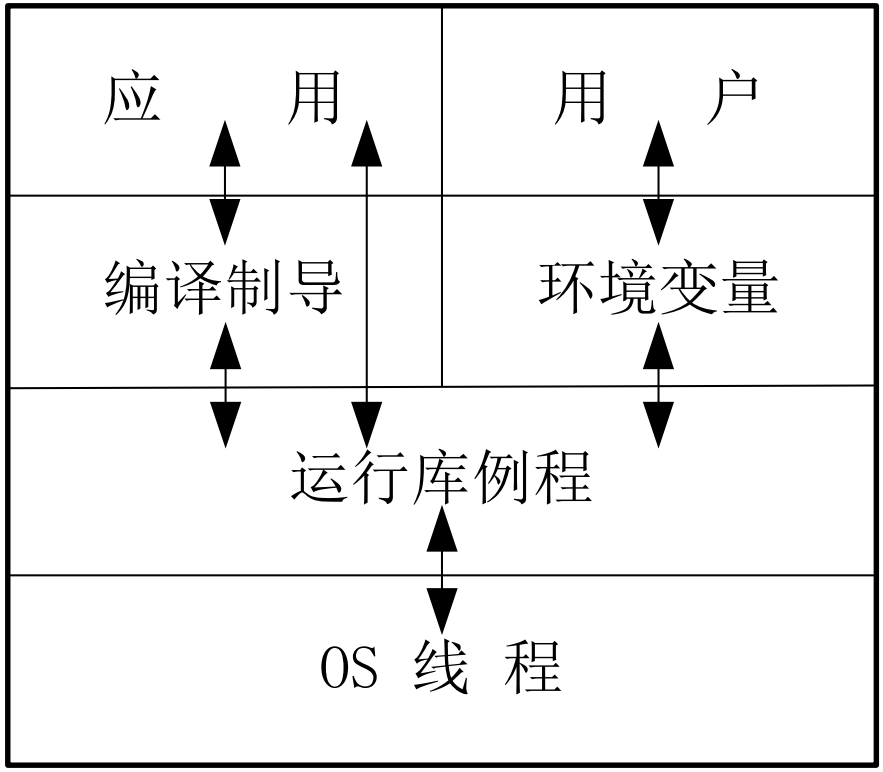
OpenMP编程简介

- OpenMP概述
- OpenMP编程风络
- OpenMP编程简介
- 运行库例程与环境变量
- OpenMP计算实例

OpenMP概述

- OpenMP应用编程接口API是在共享存储体系结构上的一个编程模型
- 包含编译制导(Compiler Directive)、运行库例程(Runtime Library)和环境变量(Environment Variables)
- 支持增量并行化(Incremental Parallelization)

OpenMP体系结构



什么是OpenMP

- 什么是OpenMP
  - 应用编程接口API（Application Programming Interface ）
  - 由三个基本API部分（编译指令、运行部分和环境变量）构成
  - 是C/C++ 和Fortan等的应用编程接口
  - 已经被大多数计算机硬件和软件厂家所标准化

- OpenMP不包含的性质
  - 不是建立在分布式存储系统上的
  - 不是在所有的环境下都是一样的
  - 不是能保证让多数共享存储器均能有效的利用

OpenMP的历史与现状

- 1994年，第一个ANSI X3H5草案提出，被否决
- 1997年，OpenMP标准规范代替原先被否决的ANSI X3H5，被人们认可
- 1997年10月公布了与Fortran语言捆绑的第一个标准规范
- 1998年11月9日公布了支持C和C++的标准规范

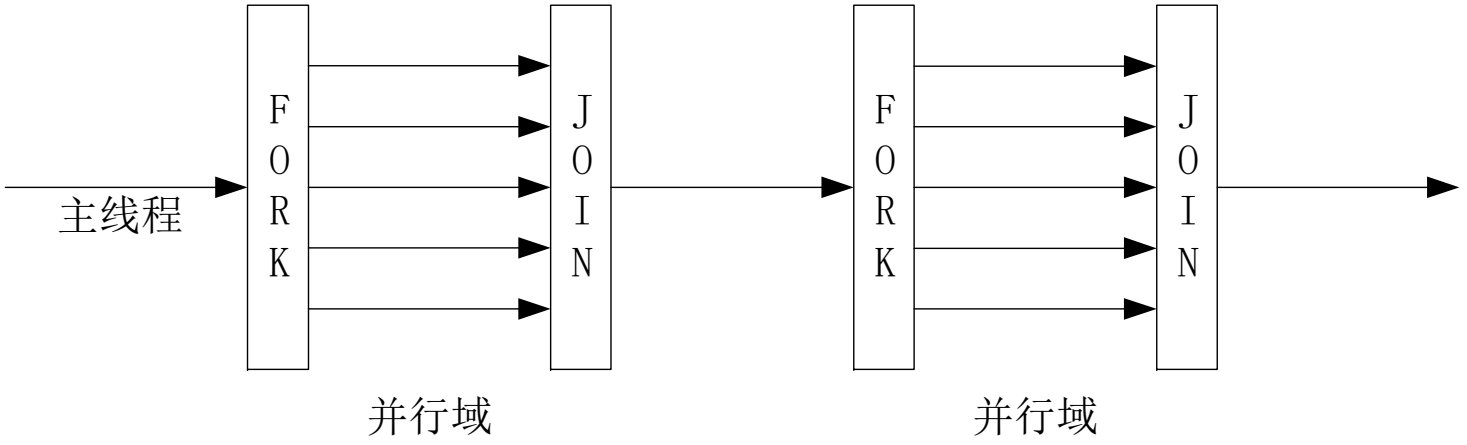
- 资源网站：http://www.openmp.org

OpenMP的目标

- 标准性
- 简洁实用
- 使用方便
- 可移植性

OpenMP并行编程模型

- 基于线程的并行编程模型(Programming Model)
- OpenMP使用Fork-Join并行执行模型



OpenMP程序结构

- 基于**Fortran**语言的**OpenMP**程序的结构

```
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3
      !Serial code
           …
      !Beginning of parallel section. Fork a team of threads.
      !Specify variable scoping
!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
      !Parallel section executed by all threads
           …
      !All threads join master thread and disband
!$OMP END PARALLEL
      !Resume serial code
           …
END
```

OpenMP程序结构

- 基于**c/c++**语言的**OpenMP**程序的结构

```
#include <omp.h>
main (){
        int var1, var2, var3;
        /*Serial code*/
          …
        /*Beginning of parallel section. Fork a team of threads*/
        /*Specify variable scoping */
        #pragma omp parallel private(var1, var2) shared(var3)
        {
                /*Parallel section executed by all threads*/
                …
                /*All threads join master thread and disband*/
        }
        /*Resume serial code */
        …
}
```

一个简单的OpenMP程序实例

- 基于**C/C++**语言的**OpenMP**程序结构的一个具体实现

```
#include "omp.h"
int main(int argc, char* argv[])
{
  int nthreads, tid;
  int nprocs;
  char buf[32];
/*    Fork a team of threads      */
  #pragma omp parallel private(nthreads, tid)
  {
      /*    Obtain and print thread id  */
       tid = omp_get_thread_num();
       printf("Hello World from OMP thread %d\n", tid);
      /*      Only master thread does this  */
       if (tid==0) {
           nthreads = omp_get_num_threads();
           printf("Number of threads %d\n", nthreads);
      }
  }
  return 0;
}
```

一个简单的OpenMP程序实例

- 运行结果 (setenv OMP_NUM_THREADS 8)

Hello World from OMP thread 0
Number of threads 8
Hello World from OMP thread 4
Hello World from OMP thread 5
Hello World from OMP thread 6
Hello World from OMP thread 7
Hello World from OMP thread 2
Hello World from OMP thread 1
Hello World from OMP thread 3

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	孤立语句 critical和sections语句出现在封闭的并行域之外

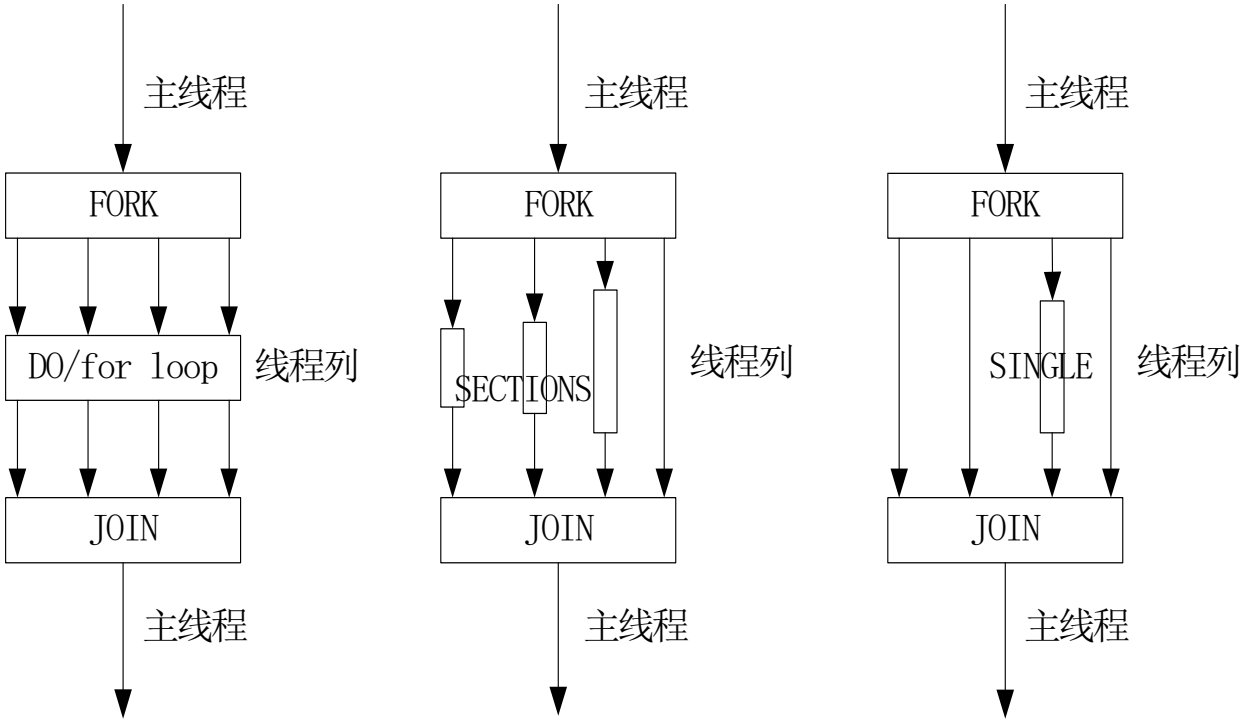
#pragma omp parallel
{
…
#pragma omp for
for(…){
…
sub1();
…
}
…
sub2();
…
}
	void sub1()
	{
	…
	#pragma omp critical
	…
	}
	void sub2()
	{
	…
	#pragma omp sections
	…
	}

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



for编译制导语句

- for语句指定紧随它的循环语句必须由线程组并行执行；
- 语句格式
  - #pragma omp for [clause[[,]clause]…] newline
  - [clause]=
    - Schedule(type [,chunk])
    - ordered
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - shared (list)
    - reduction (operator: list)
    - nowait

for编译制导语句

- schedule子句描述如何将循环的迭代划分给线程组中的线程
- 如果没有指定chunk大小，迭代会尽可能的平均分配给每个线程
- type为static，循环被分成大小为 chunk的块，静态分配给线程
- type为dynamic,循环被动态划分为大小为chunk的块，动态分配给线程

Sections编译制导语句

- sections编译制导语句指定内部的代码被划分给线程组中的各线程
- 不同的section由不同的线程执行
- Section语句格式：

```
#pragma omp sections [ clause[[,]clause]…] newline
{
[#pragma omp section newline]
        …
[#pragma omp section newline]
        …
}
```

Sections编译制导语句

- clause=
  - private (list)
  - firstprivate (list)
  - lastprivate (list)
  - reduction (operator: list)
  - nowait

- 在sections语句结束处有一个隐含的路障，使用了nowait子句除外

Sections编译制导语句

```
#include <omp.h>
#define N     1000
int main (){
        int i;
        float a[N], b[N], c[N];
        /* Some initializations */
        for (i=0; i < N; i++)
                a[i] = b[i] = i * 1.0;
        #pragma omp parallel shared(a,b,c) private(i)
           {
           #pragma omp sections nowait
                {
                #pragma omp section
                for (i=0; i < N/2; i++)
                                c[i] = a[i] + b[i];
                #pragma omp section
                for (i=N/2; i < N; i++)
                                c[i] = a[i] + b[i];
                }  /* end of sections */
           } /* end of parallel section */
}
```

single编译制导语句

- single编译制导语句指定内部代码只有线程组中的一个线程执行。
- 线程组中没有执行single语句的线程会一直等待代码块的结束，使用nowait子句除外
- 语句格式：
  - #pragma omp single [clause[[,]clause]…] newline
  - clause=
    - private(list)
    - firstprivate(list)
    - nowait

组合的并行共享任务结构

- parallel for编译制导语句
- parallel sections编译制导语句

parallel for编译制导语句

- Parallel for编译制导语句表明一个并行域包含一个独立的for语句
- 语句格式
  - #pragma omp parallel for [clause…] newline
  - clause=
    - if (scalar_logical_expression)
    - default (shared | none)
    - schedule (type [,chunk])
    - shared (list)
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - reduction (operator: list)
    - copyin (list)

parallel for编译制导语句

```
#include <omp.h>
#define N       1000
#define CHUNKSIZE   100
int main () {
      int i, chunk;
      float a[N], b[N], c[N];
      /* Some initializations */
      for (i=0; i < N; i++)
            a[i] = b[i] = i * 1.0;
      chunk = CHUNKSIZE;
      #pragma omp parallel for \
            shared(a,b,c,chunk) private(i) \
            schedule(static,chunk)
            for (i=0; i < n; i++)
                  c[i] = a[i] + b[i];
}
```

parallel sections编译制导语句

- parallel sections编译制导语句表明一个并行域包含单独的一个sections语句
- 语句格式
  - #pragma omp parallel sections [clause…] newline
  - clause=
    - default (shared | none)
    - shared (list)
    - private (list)
    - firstprivate (list)
    - lastprivate (list)
    - reduction (operator: list)
    - copyin (list)
    - ordered

同步结构

- master 制导语句
- critical制导语句
- barrier制导语句
- atomic制导语句
- flush制导语句
- ordered制导语句

master 制导语句

- master制导语句指定代码段只有主线程执行
- 语句格式
  - #pragma omp master newline

critical制导语句

- critical制导语句表明域中的代码一次只能执行一个线程
- 其他线程被阻塞在临界区
- 语句格式：
  - #pragma omp critical [name] newline

critical制导语句

```
#include <omp.h>
main()
{
    int x;
    x = 0;
    #pragma omp parallel shared(x)
        {
        #pragma omp critical
                    x = x + 1;
          }  /* end of parallel section */
}
```

barrier制导语句

- barrier制导语句用来同步一个线程组中所有的线程
- 先到达的线程在此阻塞，等待其他线程
- barrier语句最小代码必须是一个结构化的块
- 语句格式
  - #pragma omp barrier newline

barrier制导语句

- barrier正确与错误使用比较

错误	正确

if (x == 0)
#pragma omp barrier
	if (x == 0)
	{
	#pragma omp barrier
	}

atomic制导语句

- atomic制导语句指定特定的存储单元将被原子更新
- 语句格式
  - #pragma omp atomic newline
- atomic使用的格式

```
x binop = expr
x++
++x
x--
--x
```

x是一个标量
expr是一个不含对x引用的标量表达式，且不被重载
binop是+,*,-,/,&,^,|,>>,or<<之一，且不被重载

flush制导语句

- flush制导语句用以标识一个同步点，用以确保所有的线程看到一致的存储器视图
- 语句格式
  - #pragma omp flush (list) newline
- flush将在下面几种情形下隐含运行，nowait子句除外

---

barrier
critical:进入与退出部分
ordered:进入与退出部分
parallel:退出部分
for:退出部分
sections:退出部分
single:退出部分

---

ordered制导语句

- ordered制导语句指出其所包含循环的执行
- 任何时候只能有一个线程执行被ordered所限定部分
- 只能出现在for或者parallel for语句的动态范围中
- 语句格式：
  - #pragma omp ordered newline

threadprivate编译制导语句

- threadprivate语句使一个全局文件作用域的变量在并行域内变成每个线程私有
- 每个线程对该变量复制一份私有拷贝
- 语句格式:
  - #pragma omp threadprivate (list) newline

threadprivate编译制导语句

```
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
int main ()
{
    /* First parallel region */
    #pragma omp parallel private(i,beta)
    for (i=0; i < 10; i++)
        alpha[i] = beta[i] = i;
    /* Second parallel region */
    #pragma omp parallel
        printf("alpha[3]= %d and beta[3]=%d\n",alpha[3],beta[3]);
}
```

数据域属性子句

- 变量作用域范围
- 数据域属性子句
  - private子句
  - shared子句
  - default子句
  - firstprivate子句
  - lastprivate子句
  - copyin子句
  - reduction子句

private子句

- private子句表示它列出的变量对于每个线程是局部的 。
- 语句格式
  - private(list)
- private和threadprivate区别

	PRIVATE	THREADPRIVATE
数据类型	变量	变量
位置	在域的开始或共享任务单元	在块或整个文件区域的例程的定义上
持久么	否	是
扩充性	只是词法的- 除非作为子程序的参数而传递	动态的
初始化	使用 FIRSTPRIVATE	使用 COPYIN

shared子句

- shared子句表示它所列出的变量被线程组中所有的线程共享
- 所有线程都能对它进行读写访问
- 语句格式
  - shared (list)

default子句

- default子句让用户自行规定在一个并行域的静态范围中所定义的变量的缺省作用范围
- 语句格式
  - default (shared | none)

firstprivate子句

- firstprivate子句是private子句的超集
- 对变量做原子初始化
- 语句格式：
  - firstprivate (list)

lastprivate子句

- lastprivate子句是private子句的超集
- 将变量从最后的循环迭代或段复制给原始的变量
- 语句格式
  - lastprivate (list)

copyin子句

- copyin子句用来为线程组中所有线程的threadprivate变量赋相同的值
- 主线程该变量的值作为初始值
- 语句格式
  - copyin(list)

reduction子句

- reduction子句使用指定的操作对其列表中出现的变量进行规约
- 初始时，每个线程都保留一份私有拷贝
- 在结构尾部根据指定的操作对线程中的相应变量进行规约，并更新该变量的全局值
- 语句格式
  - reduction (operator: list)

reduction子句

```
#include <omp.h>
int main ()
{
      int   i, n, chunk;
      float a[100], b[100], result;
      /* Some initializations */
      n = 100;
      chunk = 10;
      result = 0.0;
      for (i=0; i < n; i++)
      {
                a[i] = i * 1.0;
                b[i] = i * 2.0;
      }
      #pragma omp parallel for default(shared) private(i)\
                schedule(static,chunk) reduction(+:result)
      for (i=0; i < n; i++)
                result = result + (a[i] * b[i]);
      printf("Final result= %f\n",result);
}
```

2022/4/26

reduction子句

- Reduction子句的格式

---

x=x op expr
x = expr op x (except subtraction)
x binop = expr
x++
++x
x--
--x

---

x是一个标量
expr是一个不含对x引用的标量表达式，且不被重载
binop是+,*,-,/,&,^,|之一，且不被重载
op是+,*,-,/,&,^,|,&&,or||之一，且不被重载

子句/编译制导语句总结

子句	编译制导					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	√				√	√
PRIVATE	√	√	√	√	√	√
SHARED	√	√			√	√
DEFAULT	√				√	√
FIRSTPRIVATE	√	√	√	√	√	√
LASTPRIVATE		√	√		√	√
REDUCTION	√	√	√		√	√
COPYIN	√				√	√
SCHEDULE		√			√	
ORDERED		√			√	
NOWAIT		√	√	√		

语句绑定和嵌套规则

- 语句绑定
  - 语句DO/for、SECTIONS、SINGLE、MASTER和BARRIER绑定到动态的封装PARALLEL中，如果没有并行域执行，这些语句是无效的；
  - 语句ORDERED指令绑定到动态DO/for封装中；
  - 语句ATOMIC使得ATOMIC语句在所有的线程中独立存取，而并不只是当前的线程；
  - 语句CRITICAL在所有线程有关CRITICAL指令中独立存取，而不是只对当前的线程；
  - 在PARALLEL封装外，一个语句并不绑定到其它的语句中。

语句绑定和嵌套规则

- 语句嵌套
  - PARALALL 语句动态地嵌套到其它地语句中，从而逻辑地建立了一个新队列，但这个队列若没有嵌套地并行域执行，则只包含当前的线程；
  - DO/for、SECTION和SINGLE语句绑定到同一个PARALLEL 中，则它们是不允许互相嵌套的；
  - DO/for、SECTION和SINGLE语句不允许在动态的扩展CRITICAL、ORDERED和MASTER域中；
  - CRITICAL语句不允许互相嵌套；
  - BARRIER语句不允许在动态的扩展DO/for、ORDERED、SECTIONS、SINGLE、MASTER和CRITICAL域中；
  - MASTER语句不允许在动态的扩展DO/for、SECTIONS和SINGLE语句中；
  - ORDERED语句不允许在动态的扩展CRITICAL域中；
  - 任何能允许执行到PARALLEL 域中的指令，在并行域外执行也是合法的。当执行到用户指定的并行域外时，语句执行只与主线程有关。

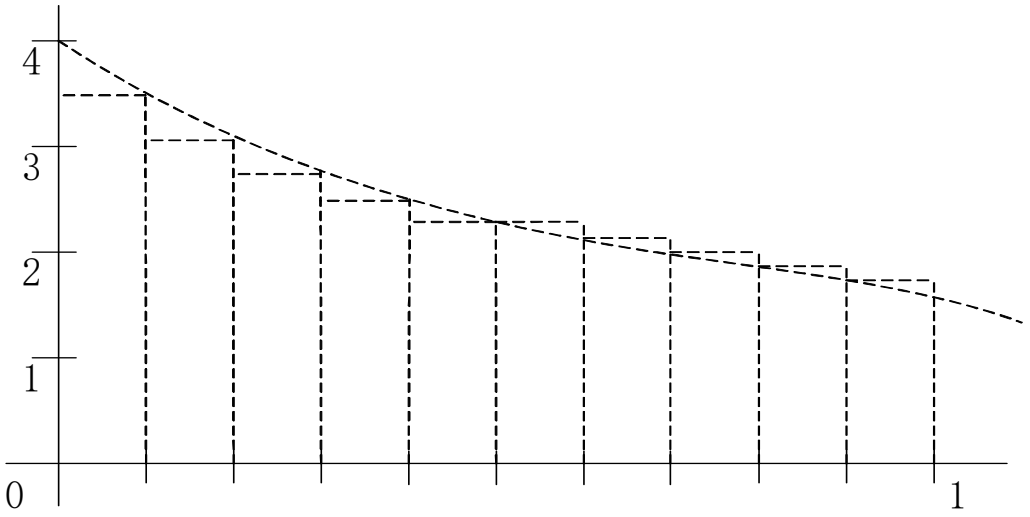
运行库例程与环境变量

- 运行库例程
  - OpenMP标准定义了一个应用编程接口来调用库中的多种函数
  - 对于C/C++，在程序开头需要引用文件“omp.h”

- 环境变量
  - OMP_SCHEDULE：只能用到for,parallel for中。它的值就是处理器中循环的次数
  - OMP_NUM_THREADS：定义执行中最大的线程数
  - OMP_DYNAMIC：通过设定变量值TRUE或FALSE,来确定是否动态设定并行域执行的线程数
  - OMP_NESTED：确定是否可以并行嵌套

计算实例：π的计算

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \approx \sum_{0 \le i \le N} \frac{4}{1+\left(\frac{i+0.5}{N}\right)^2} \cdot \frac{1}{N}$$



OpenMP计算实例

- C语言写的串行程序

```
/* Seriel Code */
static long num_steps = 100000;
double step;
void main ()
{      int i;
       double x, pi, sum = 0.0;
       step = 1.0/(double) num_steps;
       for (i=0;i< num_steps; i++){
              x = (i+0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
       }
       pi = step * sum;
}
```

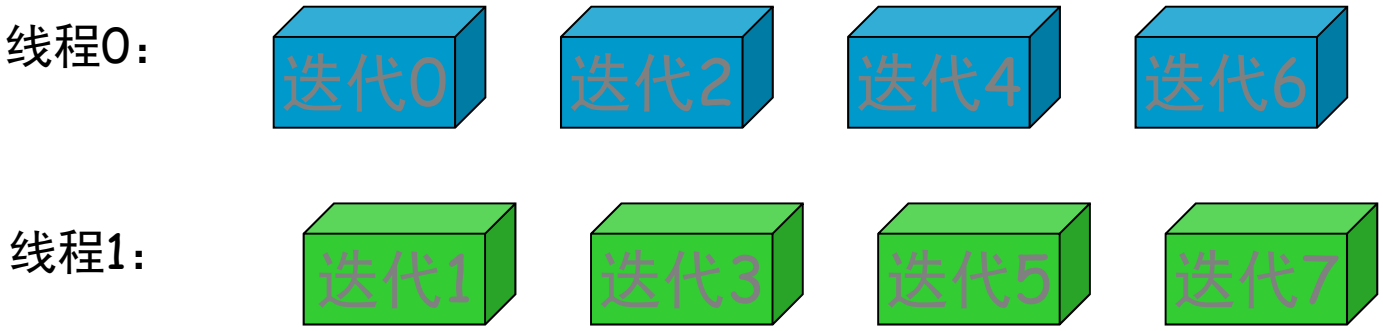
OpenMP计算实例

- 使用并行域并行化的程序

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{       int i;
        double x, pi, sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
       #pragma omp parallel
       {
                double x;
                int id;
                id = omp_get_thread_num();
                for (i=id, sum[id]=0.0;i< num_steps; i=i+NUM_THREADS){
                                x = (i+0.5)*step;
                                sum[id] += 4.0/(1.0+x*x);
                }
       }
        for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

- 使用并行域并行化的程序

  假设有2个线程参加计算：



OpenMP计算实例

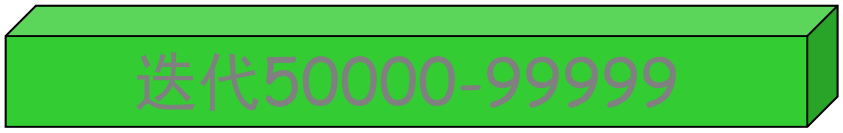
- 使用共享任务结构并行化的程序

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
        int i;
        double x, pi, sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel
        {
                double x;
                int id;
                id = omp_get_thread_num();
                sum[id] = 0;
                #pragma omp for
                for (i=0;i< num_steps; i++){
                                x = (i+0.5)*step;
                               sum[id] += 4.0/(1.0+x*x);
                }
        }
for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

- 使用共享任务结构并行化的程序
  假设2个线程参加并行计算：

线程0：

线程1：



OpenMP计算实例

- 使用private子句和critical部分并行化的程序

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
        int i;
        double x, sum, pi=0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel private (x, sum)
        {
                id = omp_get_thread_num();
                for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
                }
                #pragma omp critical
                        pi += sum*step
        }
}
```

OpenMP计算实例

- 使用并行归约得出的并行程序

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{       int i;
        double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel for reduction(+:sum) private(x)
        for (i=0;i<num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```