多核并行程序设计

谭光明　博士

tgm@ict.ac.cn

中国科学院计算技术研究所

国家智能计算机研究开发中心

计算机体系结构国家重点实验室

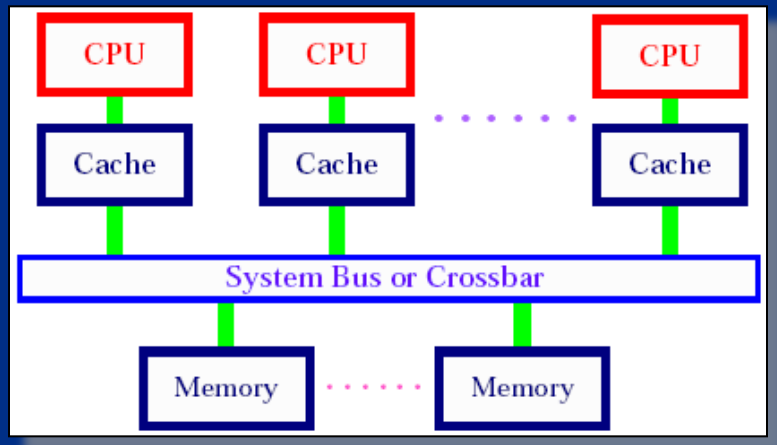
- POSIX Threads Programming
- OpenMP

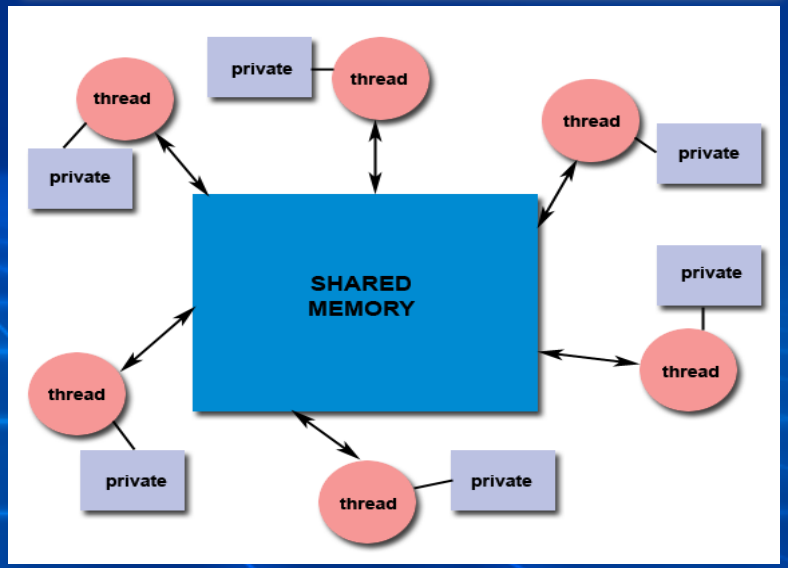
参考文献

- B. Nichols et al. *Pthreads Programming*. O'Reilly and Associates
- Pthreads: http://www.oreilly.com
- 陈文光译，《MPI＋OpenMP并行程序设计》清华大学出版社
- Michael J.Quinn,《**并行程序设计:C、MPI与OpenMP**》（影印版）清华大学出版社
- OpemMP: http://www.openmp.org

Share Memory

- 多台处理机通过互联网络共享一个统一的内存空间, 通过**单一内存地址**来实现处理机间的协调.
- 内存空间也可由多个存储器模块构成.
- 每台处理机可以执行相同或不同的指令流, **每台处理机可以直接访问到所有数据**.
- 处理机间**通信是借助于共享主存来实现的**.
- **可扩展性差**, 当处理机需要同时访问共享全局变量时, 产生内存竞争现象而严重影响效率, 比较适合中小规模应用问题的计算和事务处理.





多线程为何流行

- 多核cpu的出现
- 线程:在进程的内部执行的指令序列.
- 相对于进程,**线程开销小**:
  - 创建一个线程的时间大约是建立一个新进程的1/30.
  - 线程同步时间约是进程同步时间的1/3.
- **开发程序的并发性**,改善程序的结构.
- 容易实现**数据共享**:线程共用内存地址
- **统一的标准**:
  - 1995年的POSIX线程标准实施，各系统都支持Pthreads。

共享存储编程标准

- 共享存储器编程标准
  - Win32 API (线程标准)
  - **Pthreads**(线程标准)
  - X3H5(线程标准)
  - **OpenMP**(最常用的共享并行编程方式)
- 共享存储器编程特点
  - 显式多线程库调用.(Win32 API，Pthreads).
  - 编译制导语句，OpenMP.
- 语言
  - C,C++,Fortran77,Fortran90/95…

Pthreads线程模型

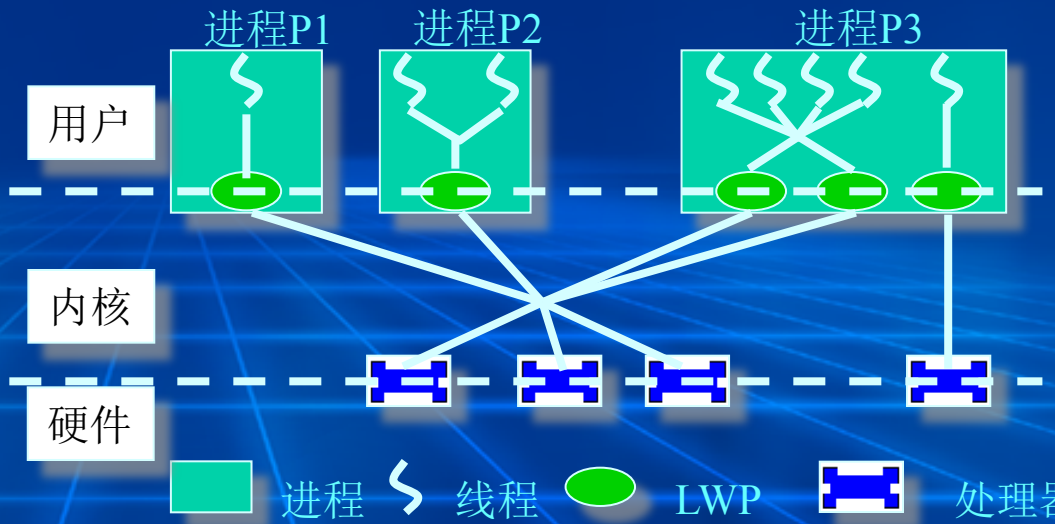
- 以前各开发商提供互不兼容的线程库, 结果导致多线程程序不能很好地移值.
- POSIX1003.4a小组研究多线程编程标准. 当标准完成后, 大多数支持多线程的系统都支持POSIX接口. 很好的改善了多线程编程的可移植性.
- IEEE Portable Operating System Interface, POSIX, 1003.1-1995标准:POSIX线程模型:*pthreads*.

Solaris 线程

---广泛使用的商品化产品

两级结构特点:用户级+内核级
用户线程:工作在用户模式/用户空间
内核线程:工作在内核模式/内核空间
用户级:动态链接的线程库实现线程应用编程接口,并管理用户线程
内核级:则管理线程池
LWP:每个进程有一个或多个线程,被分配一个或多个**LWP,**内核不能管理用户线程,只能管理**LWP;**
虚拟处理器.
进程的并发度:被分配到进程上的**LWP**数量.
线程池:线程库调度线程在它的线程池中运行
处理器池:来自进程的**LWP**将在物理处理器池中运行.
例:
P1:传统**UNIX**进程,具有单个线程
P2:两个线程,一个线程池;
P3:5个线程,分三个池;
并发度:1, 1, 3.

- Solaris线程特点:
  - –非常低的创建开销;
  - –快速同步(同步在用户空间进行);
  - –统一的实现:线程库与调试工具;



线程管理

- 创建:pthread_create
- 终止:pthread_exit
- 汇合:pthread_join
- 分离:pthread_detach
- 线程属性初始化:pthread_attr_init
- 唯一执行:pthread_once

pthread_create()函数

- **#include <pthread.h>**
- int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*start_rtn)(void), void *arg);
  1 2 3 4
- 功能：建立一个新的线程，成功返回0，失败返回非0

1. tid参数指向将创建的线程id, pthread_t: unsigned long int
2. 设置线程属性 ，一般用NULL来创建具有缺省属性的线程
3. 线程运行函数指针，start_rtn是线程开始时的执行函数
4. 运行函数的参数，函数start_rtn有一个参数arg

pthread_join()函数

- int pthread_join(pthread_t thread, void **value_ptr);
  - 1: pthread_t thread
  - 2: void **value_ptr
- 功能：等待一个线程结束，被等待线程的资源被收回。
- 调用它的函数将等待直到被等待的线程结束。
- 一个线程的结束有两种途径：
  - Pthread_join()；第二个参数为NULL
  - pthread_exit()；

1. 被等待的线程标识符
2. 用户定义的指针，用来存储被等待线程的返回值

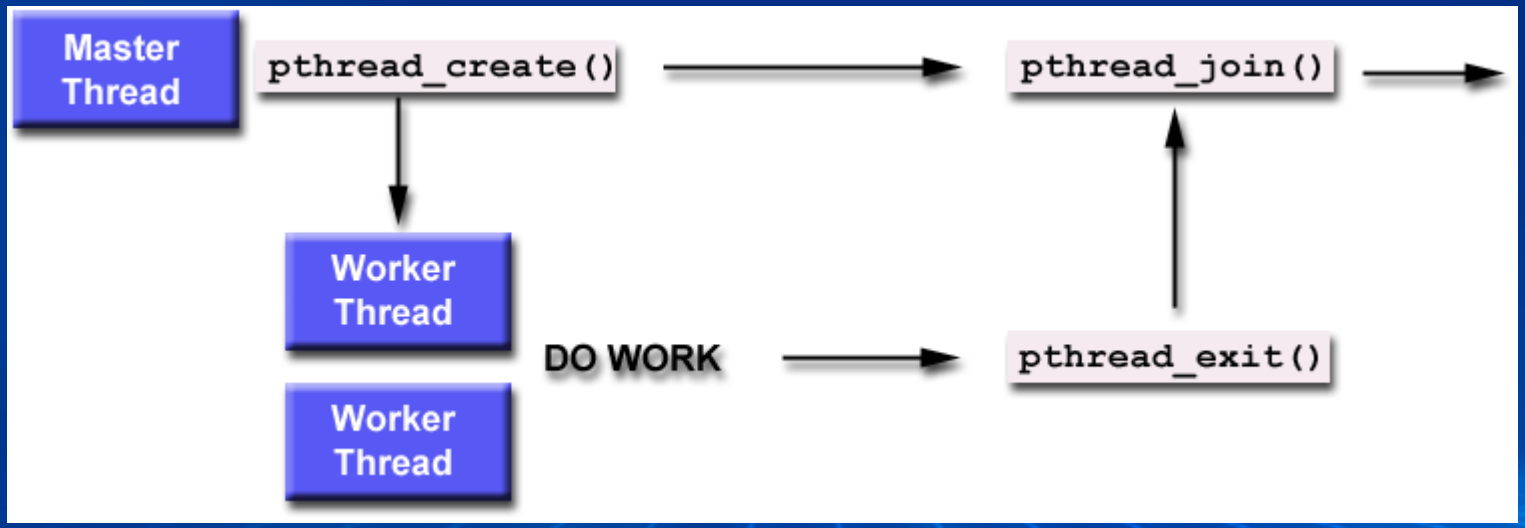
pthread_exit()函数

- void pthread_exit(void *value_ptr)
- 功能：结束线程。
- 只要pthread_join中的第二个参数value_ptr不是NULL，这个值将被传递　thread_return。

- 函数的返回代码，只要pthread_join中的第二个参数value_ptr不是NULL，这个值将被传递给thread_return

- 一个线程不能被多个线程等待

pthread_join()/pthread_exit



实例：Hello World（1）

hello1.c

```
#include <pthread.h>
#include "stdio.h"
void *worker()
{
    printf("Hello World!\n");
}
int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, worker, NULL);
    pthread_join(thread,NULL);
}
```

编译:
gcc hello.c –lpthread –o hello
运行: **./hello**
Hello World!♪

实例：Hello World（2）

hello2.c

```
#include <pthread.h>
#include "stdio.h"
void *worker()
{
    printf("Hello World!\n");
}
int main() {
    pthread_t thread;
    pthread_attr_t p_attr;
    pthread_attr_init(&p_attr);
    pthread_create(&thread, &p_attr, worker, NULL);
    pthread_exit(&p_attr);
}
```

编译:

gcc hello.c –lpthread –o hello

运行: **./hello**

Hello World!♪

pthread_attr_init()函数

- void pthread_attr_init(pthread_attr_t *attr)
- 功能：对线程属性初始化。
- 必须在pthread_create函数之前调用。

- 属性对象主要包括是否绑定、是否分离、堆栈地址、堆栈大小、优先级。默认的属性为非绑定、非分离、缺省1M的堆栈、与父进程同样级别的优先级。

实例：Hello World（3）

hello3.c

```
#include <pthread.h>
#include "stdlib.h"
#include "stdio.h"
#define N 5
typedef struct {
        int id;} parm;

void *worker(void *arg){
        parm *p = (parm *)arg;
        printf("Hello World from
                        %d!\n",p->id);
        return(0);
}

int main(){
        int i;
        pthread_t *thread;
        parm *p;
        thread = (pthread_t
            *)malloc(N*sizeof(*thread));
        p=(parm *)malloc(sizeof(parm)*N);

        for(i=0; i<N; i++) {
                p[i].id = i;
                pthread_create(&thread[i], NULL,
                    worker, (void *)(p+i));
        }

        for(i=0; i<N; i++)
            {  pthread_join(thread[i],NULL);
        }
        free(p);
        }
```

Hello World(3)运行结果

编译：

```
gcc   hello.c -o  hello  -lpthread
```

运行：

```
Hello World from thread 0!
Hello World from thread 1!
Hello World from thread 2!
Hello World from thread 3!
Hello World from thread 4!
```

同步对象

- 在共享存储多处理器并行机上, 线程通过全局变量通信, 对于全局变量的操作必须进行同步.
- pthread提供两个线程同步原语 ： 互斥和条件变量.

互斥锁函数

函数	操作
Mutex_init()	初始化一个互斥锁
Mutext_lock()	阻塞式加锁操作
Mutex_trylock()	非阻塞式加锁操作
Mutex_unlock()	解锁
Mutex_destroy()	解除互斥状态

条件变量函数

操作	函数
pthread_cond_init()	初始化条件变量
pthread_cond_wait()	阻塞直至条件为真
pthread_cond_signal()	解除等待条件的线程的阻塞
pthread_cond_timedwait()	阻塞直到等待条件为真或timeout
pthread_cond_broadcast()	解除所有等待线程的阻塞
pthread_cond _destroy()	销毁条件变量

实例：互斥变量

mutex.c

```
#include <pthread.h>
#include "stdlib.h"
#include "stdio.h"
#define N 5
typedef struct {
        int id;} parm;
pthread_t *thread;
pthread_mutex_t mutex;
int sum=0;

void *worker(void *arg){
        parm *p = (parm *)arg;
        printf("%d add to Sum in thread %d!
            \n",p->id *10, p->id);
        pthread_mutex_lock(&mutex);
        sum += p->id*10;
        pthread_mutex_unlock(&mutex);
        return(0);
}
int main(){
        int i;parm *p;
        thread = (pthread_t *)
            malloc(N*sizeof(*thread));

        p=(parm *)malloc(sizeof(parm)*N);
        pthread_mutex_init(&mutex, NULL);

        for(i=0; i<N; i++){
                p[i].id = i;
                pthread_create(&thread[i], NULL,
                    worker, (void *)(p+i));}

        for(i=0; i<N; i++){
                pthread_join(thread[i],NULL);}
        printf("The sum is %d\n",sum);
        free(p);
}
```

互斥变量运行结果

0 was added to the sum in thread 0!

10 was added to the sum in thread 1!

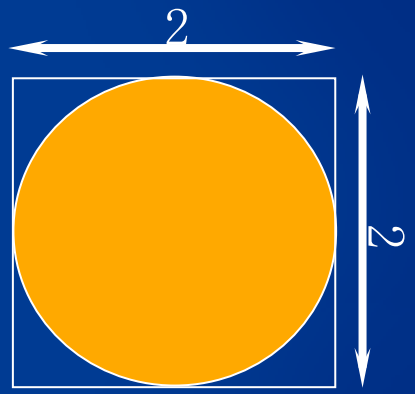
20 was added to the sum in thread 2!

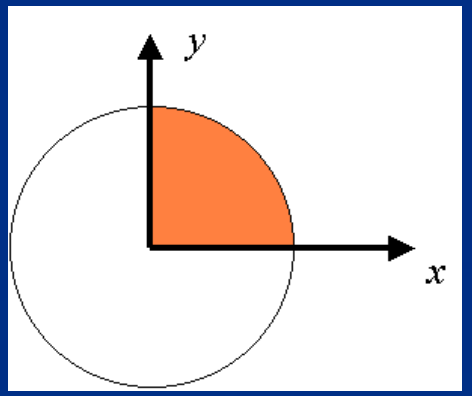
30 was added to the sum in thread 3!

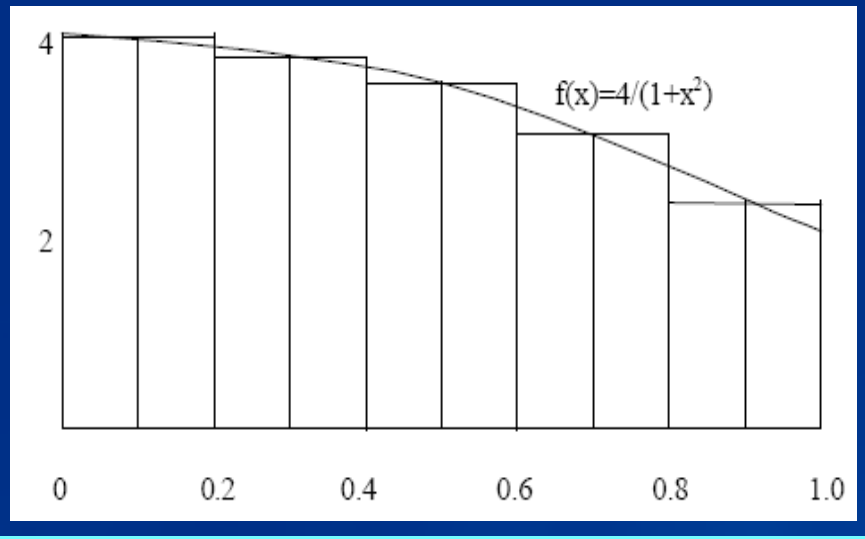
40 was added to the sum in thread 4!

The sum is 100

基于多线程PI的求解







$$\int_0^1 \frac{1}{1+x^2}dx = \arctan(x)\Big|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

令函数 $f(x) = 4/(1+x^2)$，则：$\int_0^1 f(x)dx = \pi$

实例：多线程求解PI

pi.c

```
#include <pthread.h>
#include "stdlib.h"
#include "stdio.h“
#include <math.h>

typedef struct {
      int id;
} parm;

pthread_t *thread;
pthread_mutex_t mutex;
int n, num_threads;
double pi, w;

double f (double a)
{
    return (4.0 / (1.0 + a * a));
}
```

```
void *cala_pi (void *arg)
{
    int i, myid;
    double sum=0.0,mypi, x;
    parm *p = (parm *)arg;

    for(i = p->id +1; i<= n; i+=num_threads)
    {
     x = w * ((double)i - 0.5);
     sum += f(x);
    }
    mypi = w * sum;

    pthread_mutex_lock (&mutex);
    pi +=mypi;
    pthread_mutex_unlock(&mutex);
    return(0);
}
```

实例：多线程求解PI(续)

```
int main(int argc, char* argv[])
{
    int i;
    parm        *arg;
    if (argc != 3) {
     printf("Usage: %s Num_intercals
                    Num_threads\n", argv[0]);
     exit(1);
    }
    n = atoi(argv[1]);
    num_threads = atoi(argv[2]);
    w = 1.0 / (double)n;
    pi = 0.0;
    threads = (pthread_t *) malloc(
     num_threads * sizeof(*threads));
    arg=(parm *)malloc(
     sizeof(parm)*num_threads);
    pthread_mutex_init(&mutex,NULL);
    for(i =0; i < num_threads; i++){
        arg[i].id = i;
        pthread_create(&threads[i], NULL,
                cala_pi, (void *)(arg+i));
    }
    for(i =0; i < num_threads; i++)
        pthread_join(threads[i], NULL);
    printf("coumpted pi = %.16f\n",pi);
    free(arg);
}
```

运行：pi 1000000 10
coumpted pi = 3.1415926535898704

多线程并行编程特点

- pthread_create()创建一个新线程比重新启动一个线程花费少：创建+任务结束(杀掉) vs. 维护一大堆空闲线程并且相互切换.
- 在**加锁**的前提下访问共享资源.
- **不支持数据并行**, 适合于任务级并行, 即一个线程单独执行一个任务;
- **不支持增量并行化**, 对于一个串行程序, 很难用Pthreads进行并行化;
- **Pthreads主要是面向操作系统**，而不是为高性能计算设计的，因此不是并行计算程序设计的主流平台.
- “多线程并发执行”这种思想却被广泛地应用于高性能计算.

并行编程标准

- 线程库标准(Thread Library)
  - Win32 API.
  - POSIX threads线程模型.
- 编译制导(Compiler Directives)
  - OpenMP

www.openmp.org

- *An Industry Standard API for Shared Memory Programming*
- *An API for Writing Multithreaded Applications*

- 一系列编译制导语句和库函数
- 使得**Fortran, C and C++**的多线程编程更加容易

- **Open specifications for Multi Processing**

什么是OpenMP

- 什么是**OpenMP**
  - 应用编程接口API
  - 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

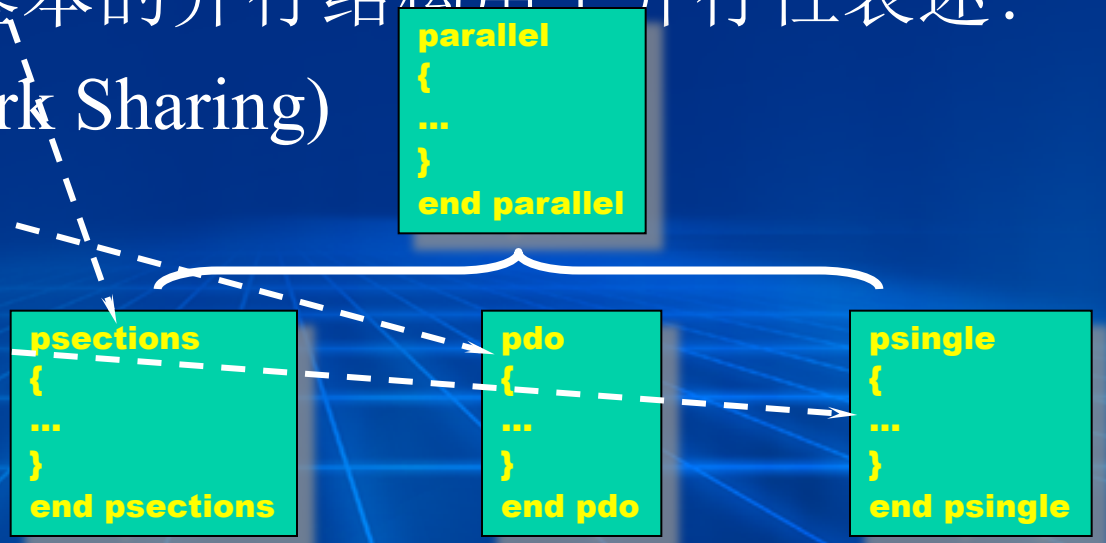
- 1994年，第一个ANSI X3H5草案提出，被否决
- 1997年，OpenMP标准规范代替被否决的ANSI X3H5，被人们认可
- 1997年10月公布了与Fortran语言捆绑的第一个标准规范 FORTRAN version 1.0
- 1998年11月9日公布了支持C和C++的标准规范C/C++ version 1.0
- 2000年11月推出FORTRAN version 2.0
- 2002年3月推出C/C++ version 2.0
- 2005年5月OpenMP2.5将原来的Fortran和C/C++ 标准规范相结合
- 相关的规范可在http://www.openmp.org/drupal/node/view/8中下载

X3H5的基础

- X3H5是ANSI/X3授权的小组委员会, 主要目的是在PCF(the Parallel Computing Forum)工作的基础上, 发展并行计算的一个ANSI标准. PCF是一非正式的工业组织, 虽在DO循环的并行化方法的标准化方面做一些工作, 但在起草拟了一个标准后就草草收场.
- OpenMP专门针对这类并行化问题, 并完成了这项工作, 同时得到工业界的广泛支持.

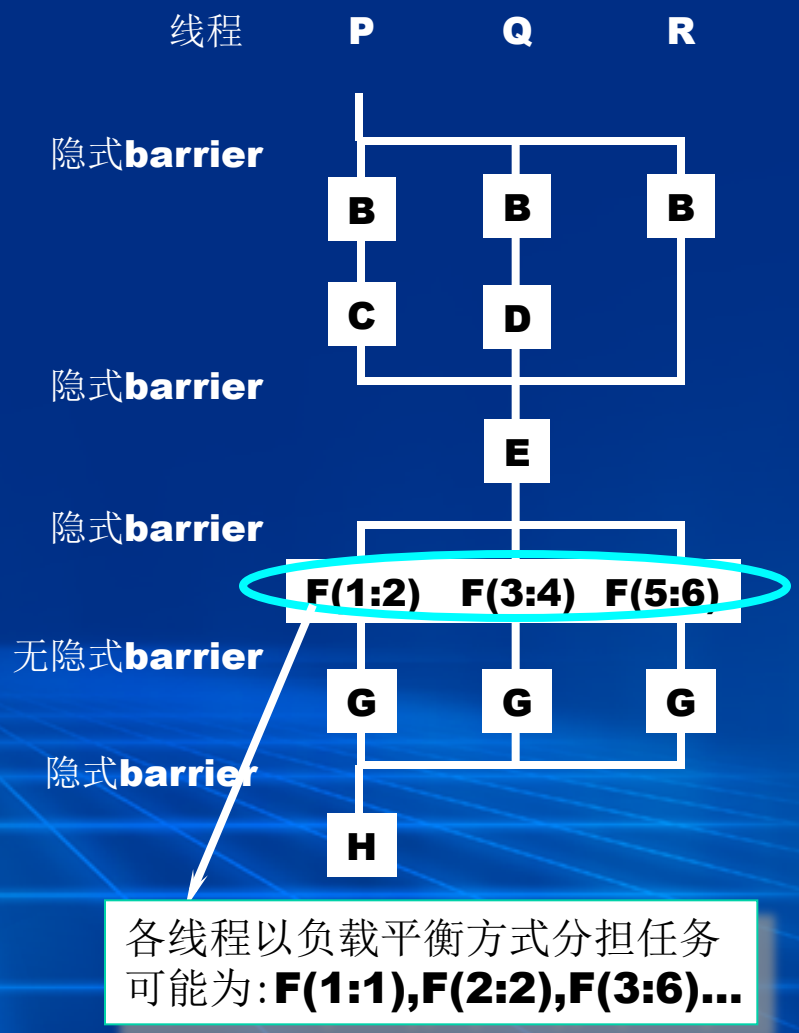
ANSI X3H5共享编程标准

- 概念性的编程模型(ANSI标准(1993))
  - 没有任何商品依附于X3H5,但其基本概念影响以后共享存储器系统的编程.(一些基本概念在OpenMP均出现!)
- X3H5规定的基本的并行结构用于并行性表述:
  - 并行块(Work Sharing)
  - 并行循环
  - 单进程



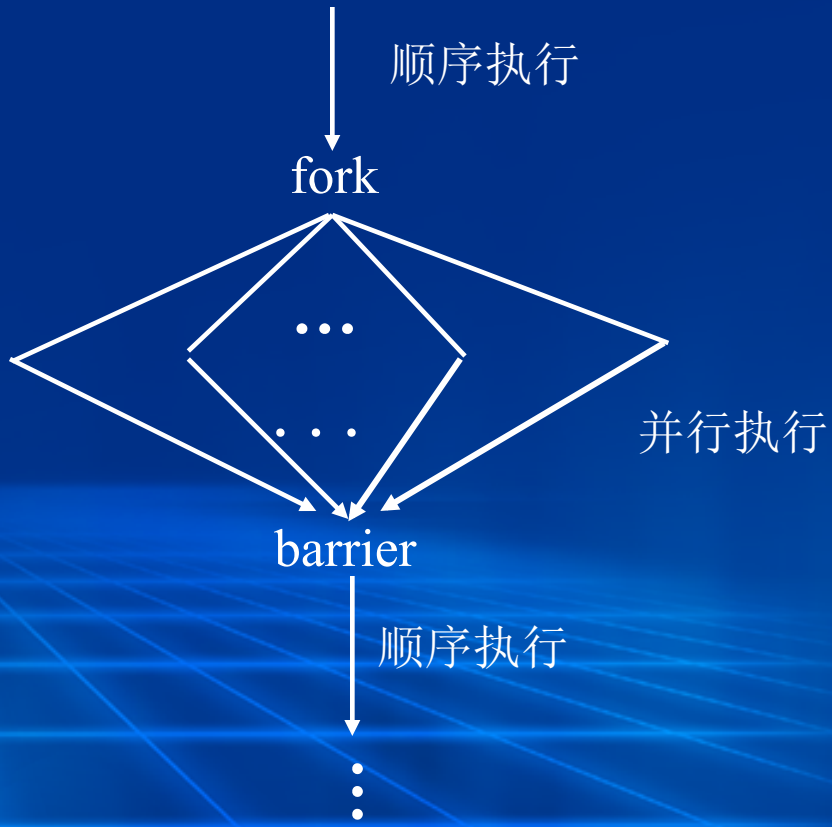
X3H5编程实例

```
program main                !程序以顺序模式执行
      A                     !A只由基本线程执行
      parallel              !转换成并行模式
            B               !B为每个组员所复制
            psections       !并行块开始
            section
                  C         !一个组员执行C
            section
                  D         !另一个组员执行D
            end psections   !等待C和D都结束
            psingle         暂时转换成顺序模式
            E               !E只能被一个组员执行
            end psingle     !转回并行模式
            pdo I=1,6       !并行do循环开始
                  F(i)      !各线程分担循环任务
            end pdo no wait !无隐式路障同步
            G               !更多的复制代码
      end parallel          !结束并行模式
      H                     !根进程执行H
      …                     !更多的并行构造
end
```



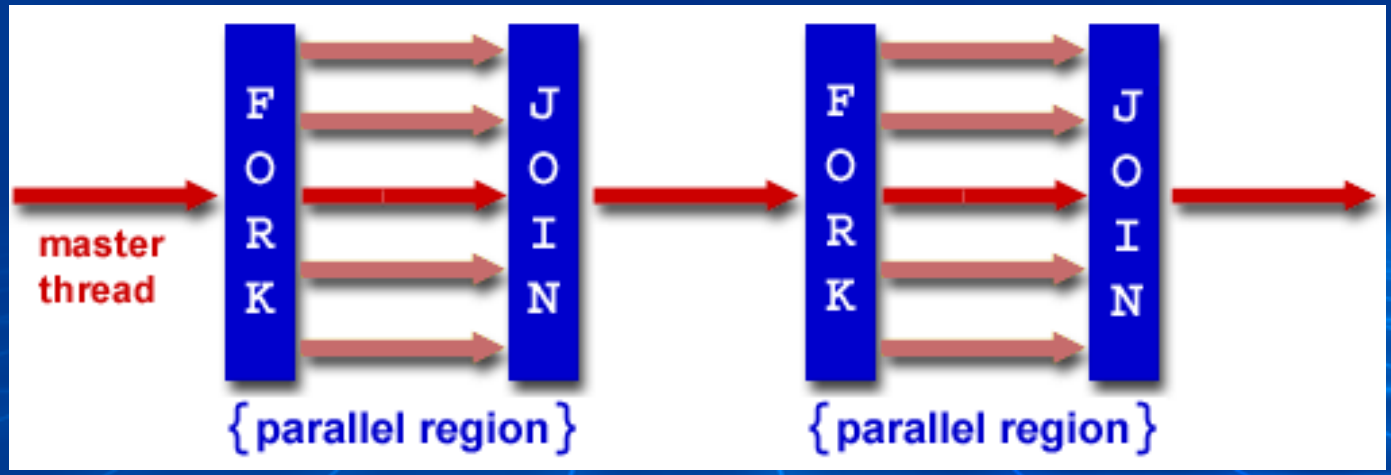
X3H5例程执行过程描述

- 程序以顺序方式启动, 此时只有一个初始化线程, 称为基本线程或主线程. 当程序遇到parallel时, 通过派生多个子线程转换为并行执行模式(线程数隐式决定). 基本线程与它的子线程形成一个组. 所有组员并行处理后继并行代码, 直至end parallel.然后程序转为顺序模式, 只有基本线程继续执行.
- 子线程遇到内部并行或任务分担构造时, 可以继续派生其子线程, 从而成为一个新组的基本线程.
- 线程间同步, 通信与交互
  - 隐式路障:parallel, end parallel, end pdo或end psingle处隐式barrier.如果不需, 则加no wait;
  - 各处理机通过全局变量通信, 通过私有变量封装数据



OpenMP: 并行模型

- Fork-Join 并行模式:
  - 主线程根据需要创建一组子线程进行工作分担.
  - 可对串行程序进行逐步并行化.



如何应用OpenMP?

- **OpenMP常用于循环并行化:**
  - 找出最耗时的循环.
  - 将循环由多线程完成.
  - 在串行程序上加上编译制导语句, 完成并行化, 因此可先完成串行程序, 然后再进行OpenMP并行化.

用OpenMP将该循环通过多线程进行任务分割

串行程序

```
void main()
{
  double Res[1000];
  for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
  }
}
```

并行程序

```
void main()
{
  double Res[1000];
  #pragma omp parallel for
  for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
}
```

线程间如何交互?

- OpenMP 是基于共享内存模型.
  - 线程通过共享变量通信.
- 访问共享变量会导致竞态状态
  - 两个实体(例如两个处理过程)对同一资源进行竞争，而系统没有一种机制来测定首先要执行的是哪一个。因此，由于系统不能保证数据的正确处理，其结果是不可预测的。
- 通过同步对象避免线程进入竞态状态

OpenMP术语

- 大多**OpenMP**构造是制导语句
  - C和C++形式:
    - **#pragma omp *construct [clause [clause]…]***
  - Fortran有以下几种形式:
    - **C$OMP *CONSTRUCT [clause [clause]…]***
    - **!$OMP *CONSTRUCT [clause [clause]…]***
    - ***$OMP *CONSTRUCT [clause [clause]…]***
- 由于OpenMP构造为注释性语句, 因此一个OpenMP程序在用不支持OpenMP的编译器编译后, 仍为串行程序.

OpenMP 程序结构

- 结构化块:
  - 仅在块顶有一个入口和块底有一个出口;
  - 块功能可通过构造的语法清晰地识别;
  - 块内除Fortran中的STOP语句和c/c++中的exit()语句外, 不能有其它分支.

- Fortran,C/C++中, OpenMP基本上一样.

OpenMP 程序结构 (Fortran)

```
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3
Serial code
      .
      .
!$OMP PARALLEL PRIVATE(VAR1) SHARED(VAR2)
    Parallel section executed by all threads
          .
          .
      All threads join master thread and disband
!$OMP END PARALLEL
Resume serial code
      .
      .
END
```

OpenMP程序结构(C/C++)

```
#include <omp.h>
main () {
    int var1, var2, var3;
    Serial code
        .
        .
        .
#pragma omp parallel private(var1, var2) shared(var3) {
    Parallel section executed by all threads
        .
        .
        .
      All threads join master thread and disband
}
    Resume serial code
        .
        .
        .
}
```

OpenMP结构化块类型

- OpenMP主要有五类结构化块:
  - 并行区 **Parallel Regions**
  - 任务分割 Work-sharing
  - 同步 Synchronization
  - 数据环境 Data Environment
  - 运行库函数

Parallel Regions(并行区)

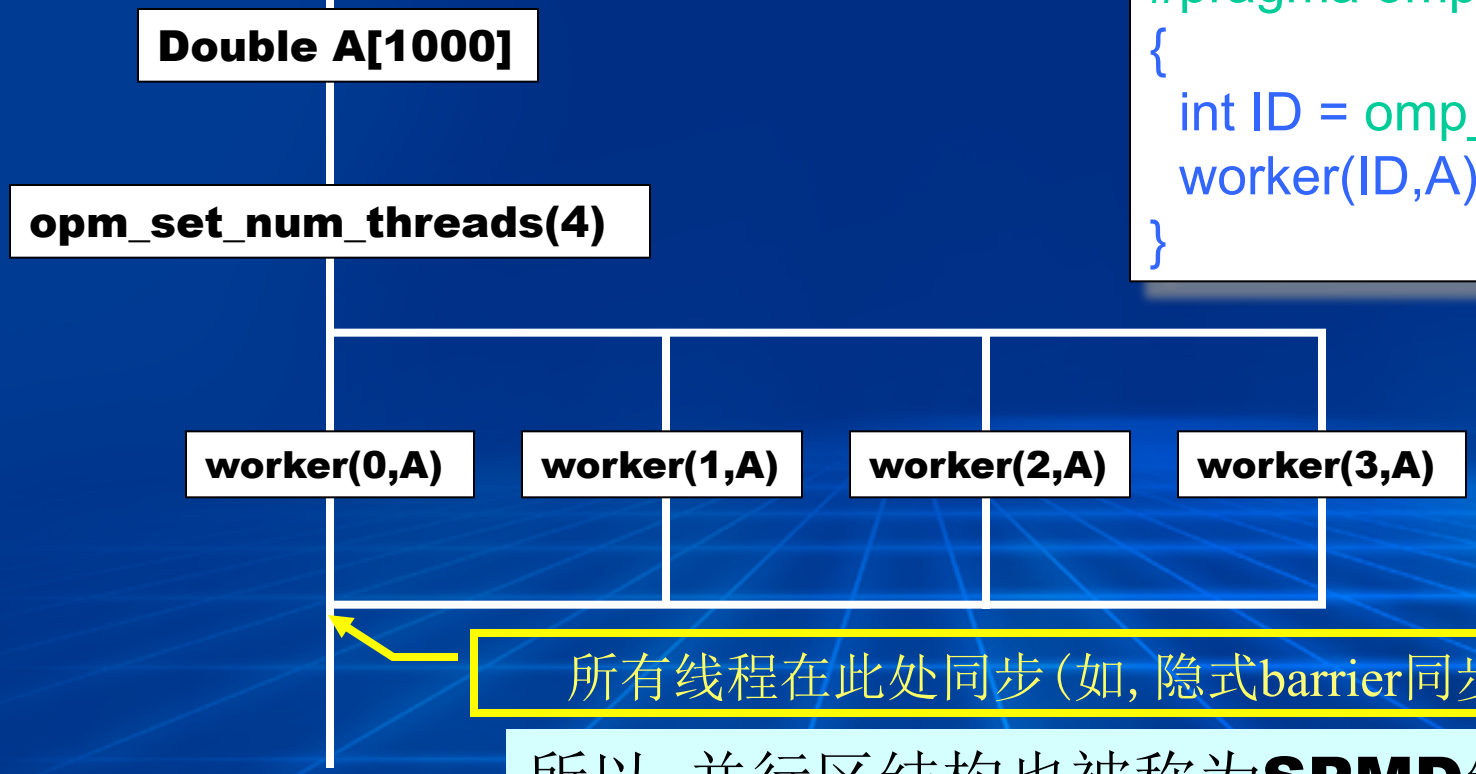
- 当一个线程执行“omp parallel”后, 建立一组线程, 该线程成为新建立的线程组的主线程. 所有线程构成一线程组, 各线程以线程ID区分, 主线程ID为0.
- 并行区内的代码由各线程同时执行.
- 如:建立一个4线程的并行区:

每一线程以不同的线程ID和相同的参数A执行并行区内代码的一拷贝.
ID(=0,1,2,3).

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int ID = omp_thread_num();
  worker(ID,A);
}
```

并行区代码流程

每一线程执行相同代码, 不同数据.




```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int ID = omp_thread_num();
  worker(ID,A);
}
```

所有线程在此处同步(如, 隐式barrier同步)

所以, 并行区结构也被称为**SPMD**结构.

实例：Hello World (C)

hello.c

```
#include <omp.h>

int main{
    int myid, numthreads;
     omp_set_num_threads(4);

     #pragma omp parallel
      {
            myid = omp_get_thread_num();
      numthreads = omp_get_num_threads();
      printf("Hello World from thread %d of %d!\n",myid,numthreads);
       }
}
```

实例：Hello World(Fortran)

```
PROGRAM HELLO
integer myid, numthreads
integer omp_get_num_threads, omp_get_thread_num
!$omp   parallel private(numthreads,myid)
              numthreads = omp_get_num_threads()
              myid = omp_get_thread_num()
              print *, 'Hello World from
thread ,myid,'of',numthreads
!$omp   end parallel
              stop
              end
```

OpenMP并行程序编译

- 支持OpenMP的编译器会提供编译器命令选项, 以解释OpenMP编译制导语句.
- pgi: pgcc -mp hello.c / pgf77 -mp hello.f
- Intel: icc -openmp hello.c
- gcc 4.2以后版本支持OpenMP
  - gcc –fopenmp hello.c –o hello

```
[zhangfa@zhangfa ~]$ icc -openmp  hello.c -o hello
[zhangfa@zhangfa ~]$ ./a.out
 Hello World from thread  2 of 4
 Hello World from thread  3 of 4
 Hello World from thread  1 of 4
 Hello World from thread  0 of 4
```

OpenMP结构化块格式(C)

#pragma omp parallel *[clause ...] newline*

- if *(scalar_expression)* 判别
- private *(list)* 变量私有化
- shared *(list)* 并行区间中的共享变量列表
- default (shared | none)
- firstprivate *(list)*
- reduction *(operator: list)* 归约操作
- copyin *(list)*
- num_threads *(integer-expression)*

structured_block

firstprivate：指定每个线程都有自己的变量私有副本，并且变量要被继承主线程中的初值。

OpenMP结构化块格式(F)

!$OMP PARALLEL *[clause ...]*
IF *(scalar_logical_expression)*
PRIVATE *(list)*
SHARED *(list)*
DEFAULT (PRIVATE | SHARED | NONE)
FIRSTPRIVATE *(list)*
REDUCTION *(operator: list)*
COPYIN *(list)*
NUM_THREADS *(scalar-integer-expression)*
block
!$OMP END PARALLEL

OpenMP结构化块类型

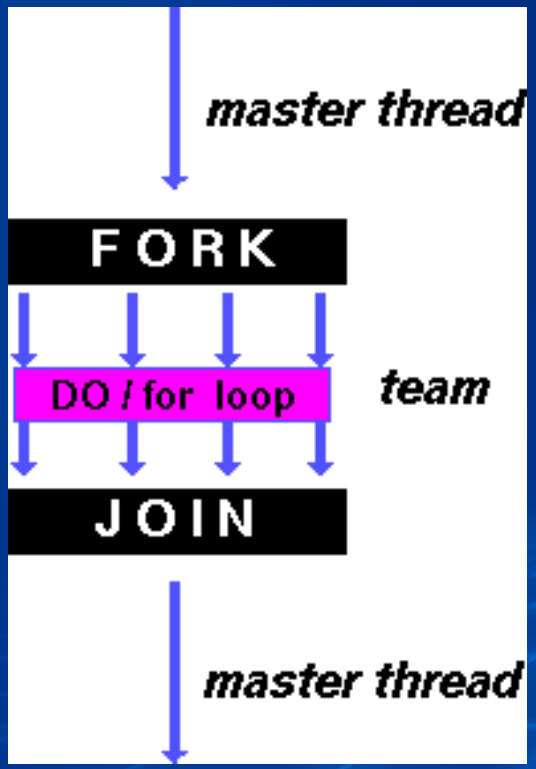
- OpenMP主要有五类结构化块:
  - 并行区 Parallel Regions
  - → **任务分割 Work-sharing**
  - 同步 Synchronization
  - 数据环境 Data Environment
  - 运行库函数

任务分割 Work-sharing

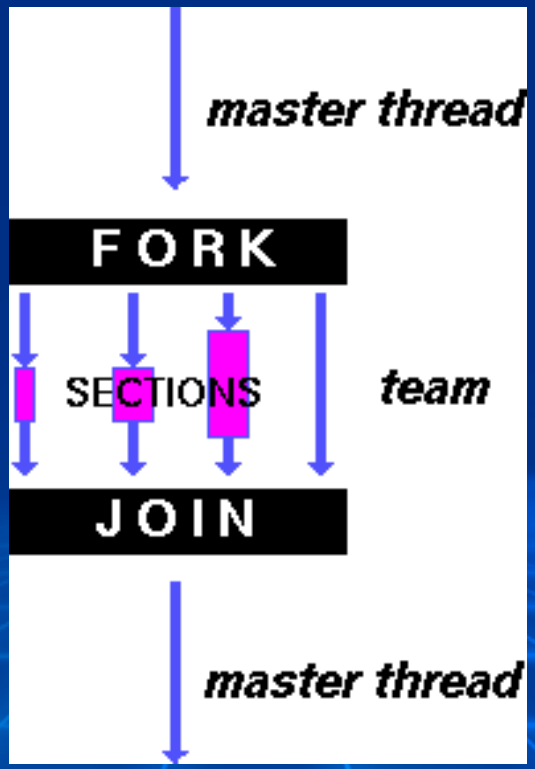
- 任务分割 (OpenMP最重要部分)
  - **for(C)/DO(Fortran)结构**:针对循环的并行化结构
  - **Sections**:代码段间的并行
  - **Single**:强制并行区中代码以串行方式执行(如:I/O)

任务分割 Work-sharing

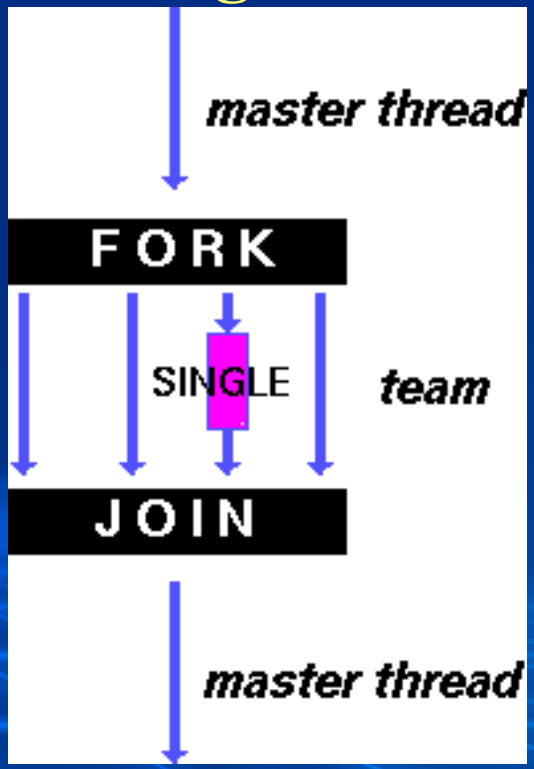
DO/for



section



single



循环分割：for结构(C/C++)

#pragma omp for *[clause ...] newline*

schedule (type [, chunk])	判别
ordered	for循环按顺序执行
private *(list)*	变量私有化
firstprivate *(list)*	
lastprivate *(list)*	
shared *(list)*	共享变量列表
reduction *(operator: list)*	归约操作
nowait	

for_loop

nowait： 在DO/for结构之后有一隐式barrier同步操作,用no wait可以禁止。

循环分割：DO结构(Fortran)

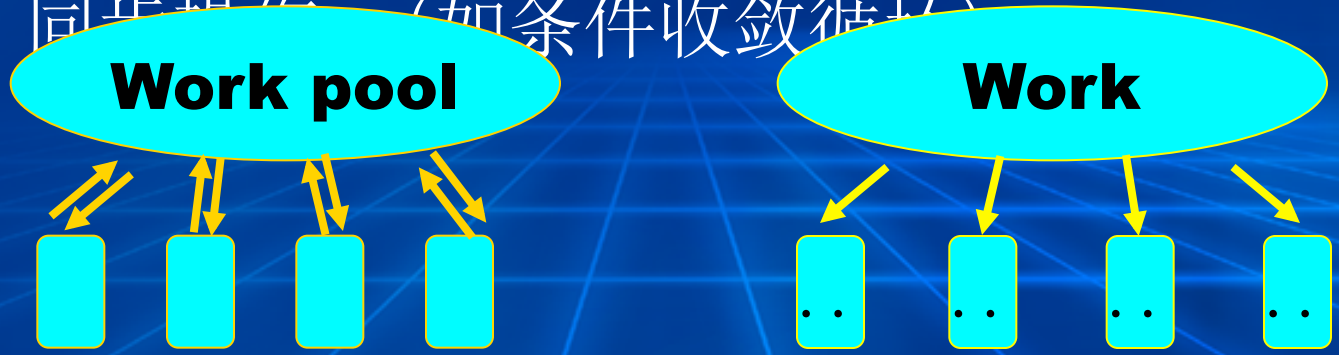
```
!$OMP DO [clause ...]
        SCHEDULE (type [, chunk])
        ORDERED
        PRIVATE (list)
        FIRSTPRIVATE (list)
        LASTPRIVATE (list)
        SHARED (list)
        REDUCTION (operator: list)

    do_loop
!$OMP END DO  [NOWAIT]
```

Schedule (type, [,chunk])

- 决定循环如何在各线程中分配:
  - schedule(dynamic[,chunk])
    - 各线程每次得到chunk_size大小的任务, 执行完后继续取得任务, 以此反复, 直至任务完成
    - 当chunk_size未被指定时, 默认为1.
  - schedule(static [,chunk])
    - 如果chunk_size被指定， 则各线程按线程号顺序得到chunk次任务.
    - 如果chunk_size未被指定, 则各线程任务数为循环数除以线程数的结果.

Schedule (type, [,chunk])

- 静态:适用于大部分情形.
  - 各线程任务明确, 在任务分配时无需同步操作.
- 动态:适用于任务数量可变或不确定的情形
  - 各线程将要执行的任务不可预见, 任务分配需同步操作(如条件收敛循环)



比较并行区构造与for构造

串行代码

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

用并行区实现并行化

```
#pragma omp parallel {
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

用任务分割实现并行化

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

DO/for使用注意事项

- 循环体必须紧接在DO或for之后.
- For循环必须为一结构化块, 且其执行不被break语句中断.
- 在Fortran中, 如果写上END DO制导语句, 其必须要紧跟在DO循环的结束之后.
- 循环变量必须为整形.
- Schedule, ordered, nowait只能出现一次.

循环分割for实例

```
#include <omp.h>
int main (){
    int i, chunk;
    float a[1000], b[1000], c[1000];
    for (i=0; i < 1000; i++) /* 初始化  */
          a[i] = b[i] = i * 1.0;
    chunk =100;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
     #pragma omp for schedule(dynamic,chunk) nowait
          for (i=0; i <1000; i++)
               c[i] = a[i] + b[i];
  }  /* end */
}
```

Sections构造(C/C++)

#pragma omp sections *[clause ...] newline*

private *(list)*
firstprivate *(list)*
lastprivate *(list)*
*r*eduction *(operator: list)*
nowait

{

#pargma omp stction *newline*
structured_block

#pargma omp stction *newline*
structured_block

}

Sections构造 (Fortran)

!$OMP SECTIONS *[clause ...]*
PRIVATE *(list)*
FIRSTPRIVATE *(list)*
LASTPRIVATE *(list)*
REDUCTION *(operator: list)*

!$OMP SETION

bolck

!$OMP SETION

bolck

!$OMP END STCTIONS [NOWAIT]

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
- 在sections语句结束处有一个隐含的同步，可用nowait禁止

非循环Sections并行

- Sections用于程序中大范围的非迭代执行代码段间的并行化.（如前10行和后10行间代码间无依赖关系）

```
#pragma omp sections [no wait]
{
    x_calculation();
    #pragma omp section
    y_calculation();
    #pragma omp section
    z_calculation();
}
```

x_calculation()，y_calculation()和z_calculation()代表三部分之间无依赖关系的非循环代码段.实质上它们各代表很多行代码.

sections实例

section.c

```
#include <omp.h>
int main (){
    int i;
    float a[1000], b[1000], c[1000];
    for (i=0; i < 1000; i++) /* 初始化 */
        a[i] = b[i] = i * 1.0;
    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
    {
            #pragma omp section
            for (i=0; i < 500; i++)    c[i] = a[i] + b[i];
            #pragma omp section
            for (i=500; i < 1000; i++)     c[i] = a[i] * b[i];
        }  /* end of sections */
    }/* end of parallel section */
```

Single构造(C/C++)

#pragma omp sections *[clause ...] newline*

private *(list)*

firstprivate *(list)*

nowait

structured_block

Single构造 (Fortran)

!$OMP SECTIONS *[clause ...]*
PRIVATE *(list)*
FIRSTPRIVATE *(list)*

bolck

!$OMP END SINGLE [NOWAIT]

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禁止.

parallel for编译制导语句

- 一个并行域包含一个独立的for语句

#pragma omp parallel for *[clause ...] newline*

schedule (type [, chunk])
ordered
private *(list)*
firstprivate *(list)*
lastprivate *(list)*
shared *(list)*
reduction *(operator: list)*
nowait

for_loop

Parallel for实例

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

parallel sections编译制导语句

- 一个并行域包含一单独的sections语句

#pragma omp parallel sections *[clause ...] newline*

private *(list)*
firstprivate *(list)*
lastprivate *(list)*
reduction *(operator: list)*
nowait

#pargma omp stction *newline*

structured_block

并行结构的联合使用

for, sections, single, 和 barrier 如果不位于并行区内或与并行区联合使用, 便不起任何作用.

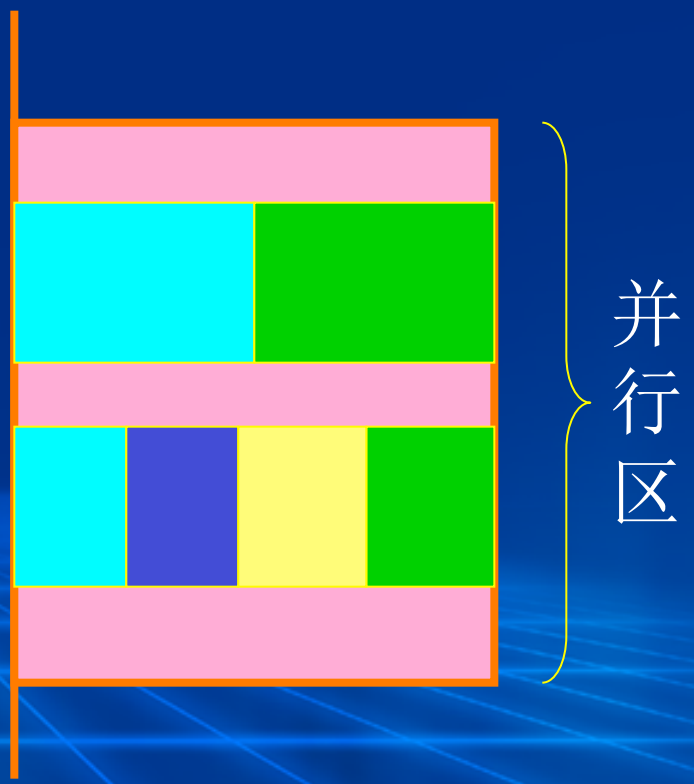
```
#pragma omp parallel for
for (i=0;i<N;i++){
    function(i);
}
```

=

```
#pragma omp parallel
#pragma omp for
for (i=0;i<N;i++){
    function(i);
}
```

并行区与任务分割间的关系

- 并行区和任务分割是OMP两类基本的并行性构造;
- 并行区中的代码对组内的各线程是可见的, 即并行区内的代码由各线程同时执行;
- 任务分割与并行区不同, 它将一个整体任务按负载平衡的方式分配给各线程来互相配合完成.
- 并行区是并行的先决条件, 任务分割必须要与并行区一起使用才能生效;
- 并行区构造为omp parallel;
- 任务分割构造有: DO/for, section 和 single三种



OpenMP结构化块类型

- OpenMP主要有五类结构化块:
  - 并行区 Parallel Regions
  - 任务分割 Work-sharing
  - **同步 Synchronization**
  - 数据环境 Data Environment
  - 运行库函数

OpenMP同步结构

- 共享变量读写，　单一文件I/O
- OpenMP提共下列同步结构:
  - **master**
  - **single**
  - **critical**
  - **atomic**
  - **barrier**
  - **flush**
  - **order**

这两个结构实质上不属于同步结构, 是并行区结构和任务分割结构中的内容

master制导语句

- 语句格式　　#pragma omp master *newline*

master制导语句指定代码段只有主线程执行，其它线程跨越该块。

该结构无隐式barrier同步

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        do_something();
    }
    #pragma barrier
    do_other_things();
}
```

single 制导语句

- 语句格式　　#pragma omp single *newline*

Single结构用于标志一个块内的代码仅由一个线程执行(最早遇到single块者,不一定是主线程).

在single块后有一隐式的barrier同步

```
#pragma omp parallel {
    #pragma omp single{
        printf("Beginning work1.\n");
        work1();
    }
    #pragma omp single nowait{
        printf("Beginning work2.\n");
        work2();
    }
}
```

同步

非同步

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*

structured_bolck

Critical实例

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

atomic制导语句

- atomic指定特定的存储单元将被原子更新
- 语句格式：

  #pragma omp atomic *newline*

  statement_expression

- atomic使用的格式：
  - ***x* binop = expr**
  - ***x*++**
  - **++*x***
  - ***x*--**
  - **--*x***

atomic vs. critical

```
#pragma omp parallel for \
    shared(x, y, index, n)
for (i=0; i<n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```

原子操作

```
#pragma omp parallel for \
    shared(x, y, index, n)
for (i=0; i<n; i++) {
    #pragma omp critical
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```

临界区

atomic vs. critical

- 原子性是指操作的不可再分性, OpenMP利用原子结构主要是用于防止多线程对内存的同一地址的并发写.
- 临界区可以完成所有的原子操作.
- 原子结构可更好被编译优化:
  - 有硬件指令可用于实现原子操作, 且系统开销也很小.
- 原子操作与并发并不矛盾, 但临界区一定是串行执行的.

barrier制导语句

- barrier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实例

```
#pragma omp parallel shared (A, B, C) private(id)
{
        id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++)
            C[i]=big_calc3(I,A);
#pragma omp for nowait
        for(i=0;i<N;i++)
            B[i]=big_calc2(C, i);
        A[id] = big_calc3(id);
}
```

for结构有隐式barrier同步

for结构无隐式barrier同步

parallel结构隐式barrier同步

ordered制导语句

- ordered语句指出其所包含循环的执行顺序
- 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语句中
- 语句格式：

  #pragma omp ordered *newline*

Ordered实例

ordered.c

```
void worker(int k){
    #pragma omp ordered
    printf("%d  ",k);
}
int main(){
    int i;
    #pragma omp parallel for schedule (dynamic)
    for (i=0;i<5;i++)
        worker(i);
}
```

运行结果:

0 1 2 3 4

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	

隐式同步

- 下列OpenMP结构之后有隐式Barrier:
  - end parallel
  - end do/for (用nowait禁止)
  - end sections (用nowait禁止)
  - end critical
  - end single (用nowait禁止)

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实例

threadprivate.c

```
#include <omp.h>
int  a, b, i, tid; float x;
#pragma omp threadprivate(a, x)
int main ()  {
    printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid) {
    tid = omp_get_thread_num();
    a = b = tid;   x = 1.1 * tid +1.0;
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
    }  /* end of parallel section */
    printf("2st Parallel Region:\n");
#pragma omp parallel private(tid) {
    tid = omp_get_thread_num();
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
    }  /* end of parallel section */
}
```

threadprivate运行结果

Output:

1st Parallel Region:
Thread 0: a,b,x= 0 0 1.000000
Thread 2: a,b,x= 2 2 3.200000
Thread 3: a,b,x= 3 3 4.300000
Thread 1: a,b,x= 1 1 2.100000

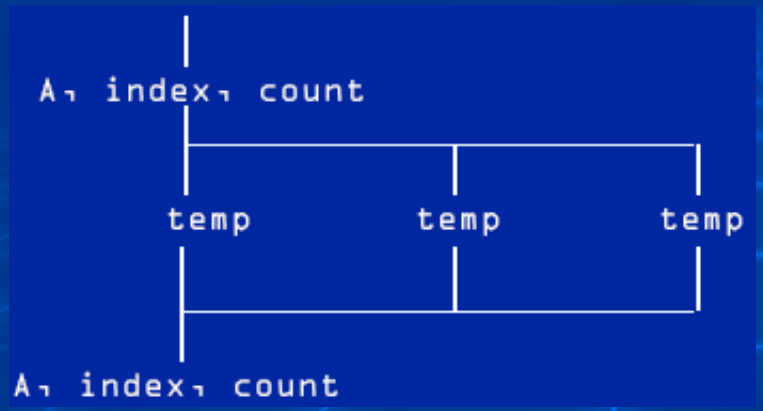
2nd Parallel Region:
Thread 0: a,b,x= 0 0 1.000000
Thread 3: a,b,x= 3 0 4.300000
Thread 1: a,b,x= 1 0 2.100000
Thread 2: a,b,x= 2 0 3.200000

OpenMP结构化块类型

- OpenMP主要有五类结构化块:
  - 并行区 Parallel Regions
  - 任务分割 Work-sharing
  - 同步 Synchronization
  - **数据环境 Data Environment**
  - 运行库函数

默认数据类型

- 共享变量编程模型:
  - 大部分变量默认为共享类型
- 各线程共享全局变量
- 但并不是所有变量全部共享...
  - 在并行区内调用的子程序中的栈变量是私有
  - 在语句块中的**Auto**变量为私有变量.



变量A, index和count为所有线程共享.
temp为线程私有变量,各线程间到不可见.

变量类型

- private 类型
- shared 类型
- default 类型
- firstprivate 类型
- lastprivate 类型
- copyin 类型
- reduction

private

- private表示它列出的变量对于每个线程是局部的.
- 语句格式　　private *(list)*

```
int main(){
    int i = 10000;
    #pragma omp parallel
    {
        i = -10000;
    }
    printf("%d\n",i);
}
运行:
-10000
```

```
int main(){
    int i = 10000;
    #pragma omp parallel private(i)
    {
        i = -10000;
    }
    printf("%d\n",i);
}
运行:
10000
```

private

- private变量将在组内每一线程中进行创建，如果变量有构造函数，则调用构造函数进行初始化，否则变量的值不确定.

- 在并行结构中，私有变量引用的原变量值是不确定的；不能在并行构造对引用的变量进行修改；在结构中对这些对私有变量的修改不影响退出结构后的同名变量值.

private实例

private.c

for结构中：

```
main(){
    int i,sum=0,myid;
    #pragma omp parallel for private(i,sum)
    for(i=0;i<10;i++){
        myid = omp_get_thread_num();
        printf("%d\n",myid);
        sum = sum + 1;
        printf("%d\n",sum);
    }
    printf("sum = %d\n",sum);
}
```

变量sum
未初始化.

```
3
804398705
2
804398705
2
804398706
2
804398707
0
804398705
0
804398706
0
804398707
1
804398705
1
804398706
1
804398707
sum = 0
```

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
持久性	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

- 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

- 用户自行规定在一个并行域中所定义变量的缺省作用范围
- 默认:default(shared)(因此不显示使用)
- 语句格式

  default (shared | none)

firstprivate

firstprivate.c

- firstprivate是private的特例
- 在主线程中初始化每一个线程的私有拷贝变量

```
main(){
    int i,sum=0,myid;
    #pragma omp parallel for firstprivate(i,sum)
    for(i=0;i<10;i++){
          sum = sum + 1;
          printf("%d\n",sum);
    }
    printf("sum = %d\n",sum);
}
```

```
1
2
3
4
1
2
3
1
2
3
1
sum=0
```

lastprivate

- 将变量从最后的循环迭代复制给原始的变量
- 语句格式

  lastprivate *(list)*

reduction

- reduction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

reduction

- reduction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

reduction格式

- reduction *(operator: list)*
  - List中所列变量必须为并行区内的共享变量
  - 仅支持标量的归约操作
  - op是+,*,-,/,&,^,|,&&, ||之一

reduction实例

redution.c

```
#include <omp.h>

int main () {
    int   i, n = 100 , chunk = 10;
    float a[100], b[100], result = 0.0;

    for (i=0; i < n; i++) {/* Some initializations */
          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);
}
```

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
  - Parallel语句动态嵌套到其它语句中，可建立了一新队列，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。

OpenMP结构化块类型

- OpenMP主要有五类结构化块:
  - 并行区　　Parallel Regions
  - 任务分割　　Work-sharing
  - 同步　　Synchronization
  - 数据环境　　Data Environment
  - → 运行库函数

OpenMP函数

- 锁函数
  - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- 运行时环境函数:
  - 改变或检查线程数
  - **omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()**
  - 开/关嵌套和动态代码
  - omp_set_nested(), omp_set_dynamic(), omp_get_nested(), omp_get_dynamic()
  - 当前代码位于并行区内吗?
  - omp_in_parallel()
  - 机器有多少个处理器?
  - omp_get_num_procs()

OpenMP函数

- **omp_set_num_threads**, 设置并行执行代码时的线程个数
- **omp_get_num_procs**, 返回运行本线程的处理器个数。
- **omp_get_num_threads**, 返回当前并行区中的线程个数。
- **omp_get_thread_num**, 返回线程号

- **omp_init_lock**, 初始化一个简单锁
- **omp_set_lock**, 上锁操作
- **omp_unset_lock**, 解锁操作，要和omp_set_lock函数配对使用。
- **omp_destroy_lock**, omp_init_lock函数的配对操作函数，关闭一个锁

锁的使用

- 下面的例子是用来显示锁的使用：
  - 锁对象具有omp_lock_t,在锁使用前需对锁进行初始化：omp_init_lock().
  - 当一个线程试图获得锁以进入第一个临界区时，处理于空闲状态，以等待进入临界区.
  - 一旦得到锁，则对临界区加锁：omp_set_lock(),退出完临界区时解锁omp_unset_lock().
  - 在进入第二个临界区时，则通过非阻塞函数omp_test_lock()来测试可能性.
  - 在锁不再使用时，用omp_destroy_lock()销毁.

锁的使用实例

```
#include <omp.h>
int main()
{
    omp_lock_t lck;
    int id;
    omp_init_lock(&lck);
    #pragma omp parallel shared(lck) private(id) {
        id = omp_get_thread_num();
        omp_set_lock(&lck);
        printf("My thread id is %d.\n", id);
        omp_unset_lock(&lck);
        while (! omp_test_lock(&lck)) {
            skip(id); /* do not yet have the lock, do something else */
        }
        work(id); /* we now have the lock and can do the work */
        omp_unset_lock(&lck);
    }
    omp_destroy_lock(&lck);
}
```

OpenMP函数汇总

- 并行区
  - parallel
- 任务分割
  - for
  - sections
  - single
- parallel for / parallel sections
- 同步
  - master
  - critical
  - barrier
  - atomic
  - flush
  - ordered

OpenMP函数汇总

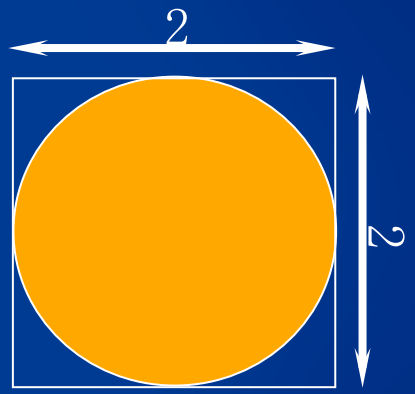
运行环境函数汇总

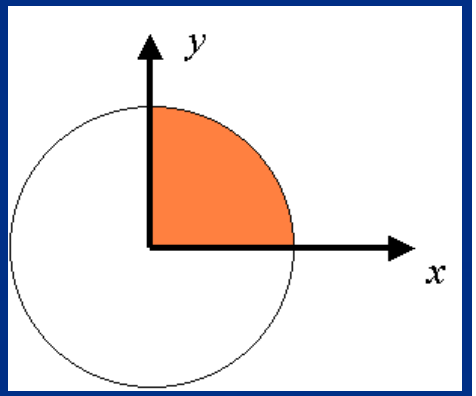
omp_set_num_threads
omp_get_num_threads
omp_get_max_threads
omp_get_thread_num
omp_get_num_procs
omp_in_parallel
omp_set_dynamic
omp_get_dynamic
omp_set_nested
omp_get_nested

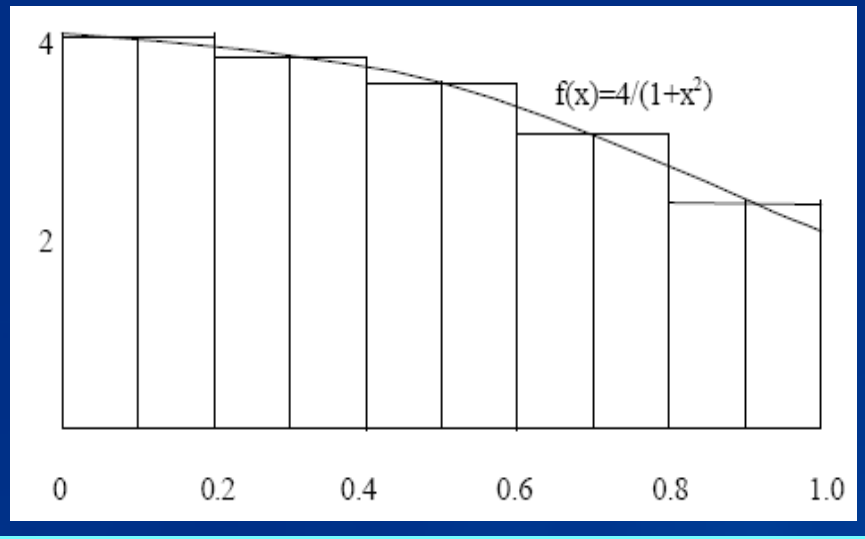
锁函数汇总

omp_init_lock
omp_init_nest_lock
omp_destroy_lock
omp_destroy_nest_lock
omp_set_lock
omp_set_nest_lock
omp_unset_lock
omp_unset_nest_lock
omp_test_lock
omp_test_nest_lock

基于多线程PI的求解







$$\int_0^1 \frac{1}{1+x^2}dx = \arctan(x)\Big|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

令函数 $f(x) = 4/(1+x^2)$，则：$\int_0^1 f(x)dx = \pi$

计算Pi的串行程序

pi.c

```
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;
}
```

下面的程序有错误！

OpenMP计算Pi(并行区)

pi_p_region.c

```
#include <omp.h>
static long num_steps = 100000;
#define NUM_THREADS 4
int main () {
      int i, id;
      double x, pi, step, sum[NUM_THREADS];
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel  {
            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;
}
```

OpenMP计算Pi(work sharing)

pi_work_sharing.c

```
#include <omp.h>
static long num_steps = 100000;
#define NUM_THREADS 4
int main () {
      int i, id;
      double x, pi, step, sum[NUM_THREADS];
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS);
      #pragma omp parallel     {
            id = omp_get_thread_num();
            sum[id] = 0.0;
            #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;
}
```

OpenMP计算Pi(private)

pi_private.c

```
#include <omp.h>
static long num_steps = 100000;
#define NUM_THREADS 4
int main () {
      int i, id;
       double x, pi = 0.0, step, sum;
       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Pi(reduction)

pi_reduction.c

```
#include <omp.h>
static long num_steps = 100000;
#define NUM_THREADS 4
int main () {
      int i, id;
       double x, pi, step, sum=0.0;
       step = 1.0/(double) num_steps;
       omp_set_num_threads(NUM_THREADS);
      #pragma omp parallel reduction (+ : sum) private (x, sum)

      for (i=0; i< num_steps; i++){
            x = (i+0.5)*step;
           sum[id] += 4.0/(1.0+x*x);
      }
     pi = sum * step;
}
```

THANKS

www.ncic.ac.cn/~tgm

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES