

高性能计算与云计算

第八讲 共享存储编程

何克晶

kejinghe@gmail.com

华南理工大学

计算机科学与工程学院



华南理工大学
South China University of Technology

共享存储系统编程

- ANSI X3H5 共享存储模型
- POSIX 线程模型
- OpenMP 模型



编程标准的作用

- 规定程序的执行模型
 - SPMD, SMP 等
- 如何表达并行性
 - DOACROSS, FORALL, PARALLEL, INDEPENDENT
- 如何表达同步
 - Lock, Barrier, Semaphore, Condition Variables
- 如何获得运行时的环境变量
 - threadid, num of processes



ANSI X3H5共享存储器模型

- Started in the mid-80's with the emergence of shared memory parallel computers with proprietary directive driven programming environments
- 更早的标准化结果—PCF共享存储器并行Fortran
- 1993年制定的概念性编程模型
- Language Binding
 - C
 - Fortran 77
 - Fortran 90



X3H5共享存储器模型

■ 并行块（工作共享构造）

- ◆ 并行块(psections ... end psections)
- ◆ 并行循环(pdo ... Endo pdo)
- ◆ 单进程(psingle ... End psingle)
- ◆ 可嵌套

■ 非共享块重复执行

■ 隐式路障(**nowait**)，显式路障和阻挡操作

■ 共享/私有变量

■ 线程同步

- ◆ 门插销(latch): 临界区
- ◆ 锁: test,lock,unlock
- ◆ 事件:wait,post,clear
- ◆ 序数(ordinal):顺序



X3H5: 并行性构造

Program main

A

parallel

)

B

psections

section

C

section

D

end psections

psingle

E

end psingle

pdo i=1,6

F(i)

end pdo no wait

G

end parallel

H

...

!程序以顺序模式开始,此时只有一个

!A只由基本线程执行, 称为主线程

!转换为并行模式, 派生出多个子线程 (一个组

!B为每个组员所复制

!并行块开始

!一个组员执行C

!一个组员执行D

!等待C和D都结束

!暂时转换成顺序模式

!已由一个组员执行

!转回并行模式

!pdo构造开始

!组员共享F的六次迭代

!无隐式路障同步

!更多的复制代码

!转为顺序模式

!初始化进程单独执行H

!可能有更多的并行构造



线程

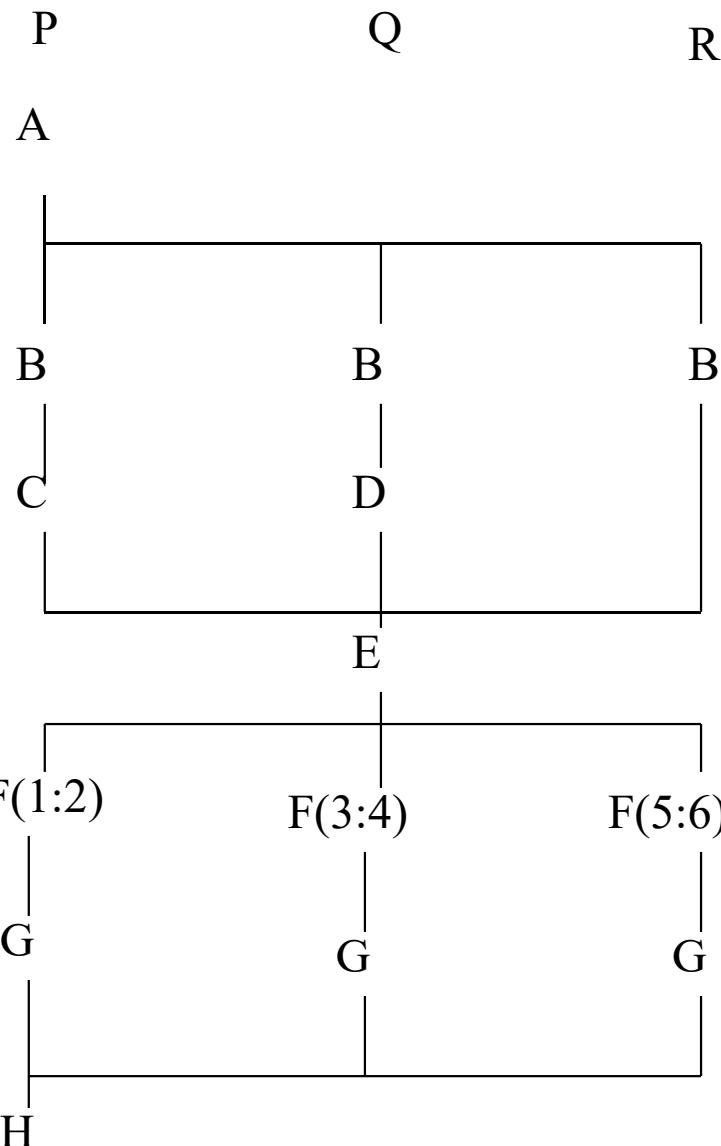
隐式路障同步

隐式路障同步

隐式路障同步

无隐式路障同步

隐式路障同步



共享存储系统编程

- ANSI X3H5 共享存储模型
- POSIX 线程模型
- OpenMP 模型

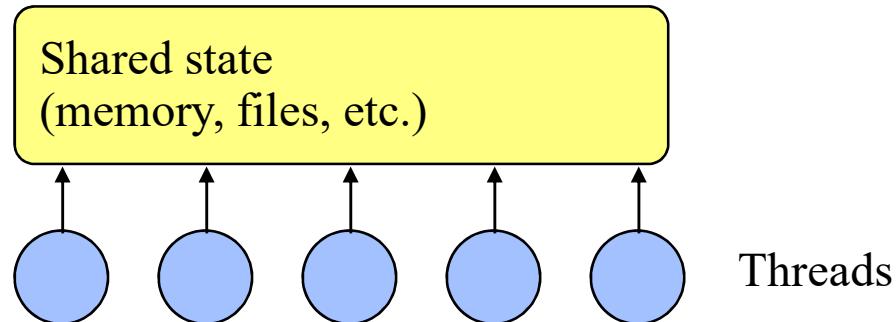


POSIX线程模型

- IEEE/ANSI标准—IEEE POSIX 1003.1c-1995线程标准—Unix/NT操作系统层上的，SMP
- Chorus, Topaz, Mach Cthreads
- Win32 Thread
 - ➔ GetThreadHandle, SetThreadPriority, SuspendThread, ResumeThread
 - ➔ TLS(线程局部存储)—TlsAlloc, TlsSetValue
- LinuxThreads: __clone and sys_clone
- 用户线程和内核线程(LWP)(一到一, 一到多, 多到多)



What Are Threads?



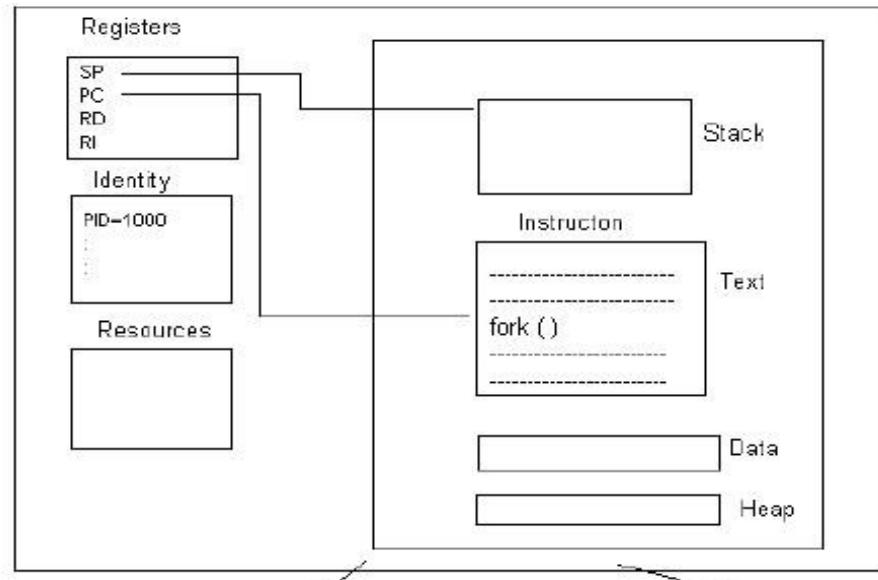
- ❑ General-purpose solution for managing concurrency.
- ❑ Multiple independent execution streams.
- ❑ Shared state.
- ❑ Preemptive scheduling.
- ❑ Synchronization (e.g. locks, conditions).



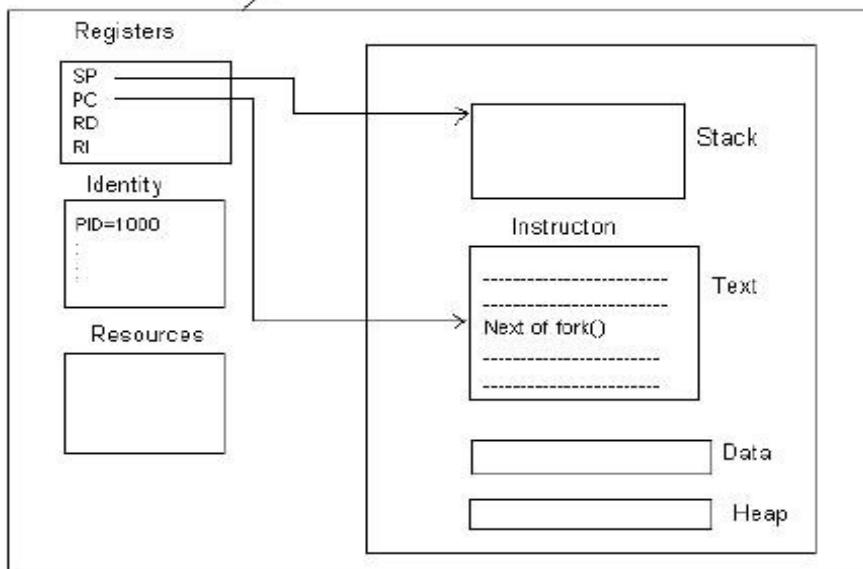
Threads

- 线程共享相同的内存空间。
- 与标准 `fork()` 相比，线程带来的开销很小。内核无需单独复制进程内存空间或文件描述符等等。这就节省了大量的 CPU 时间。
- 和进程一样，线程将利用多 CPU。如果软件是针对多处理器系统设计的，计算密集型应用。
- 支持内存共享无需使用繁琐的 IPC 和其它复杂的通信机制。
- Linux `_clone` 不可移植，`Pthread` 可移植。
- POSIX 线程标准不记录任何“家族”信息。无父无子。如果要等待一个线程终止，就必须将线程的 `tid` 传递给 `pthread_join()`。线程库无法为您断定 `tid`。





Before



POSIX Threads: Basics and Examples

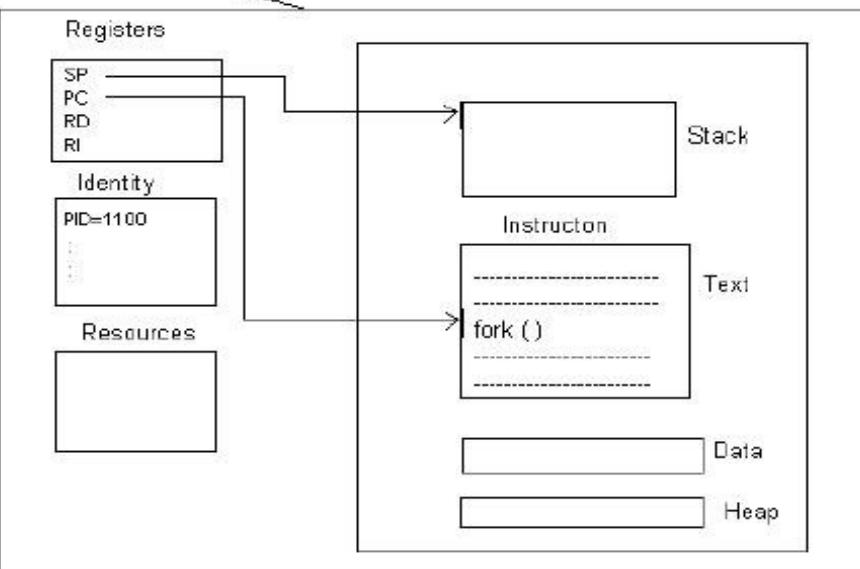
by Uday Kamath

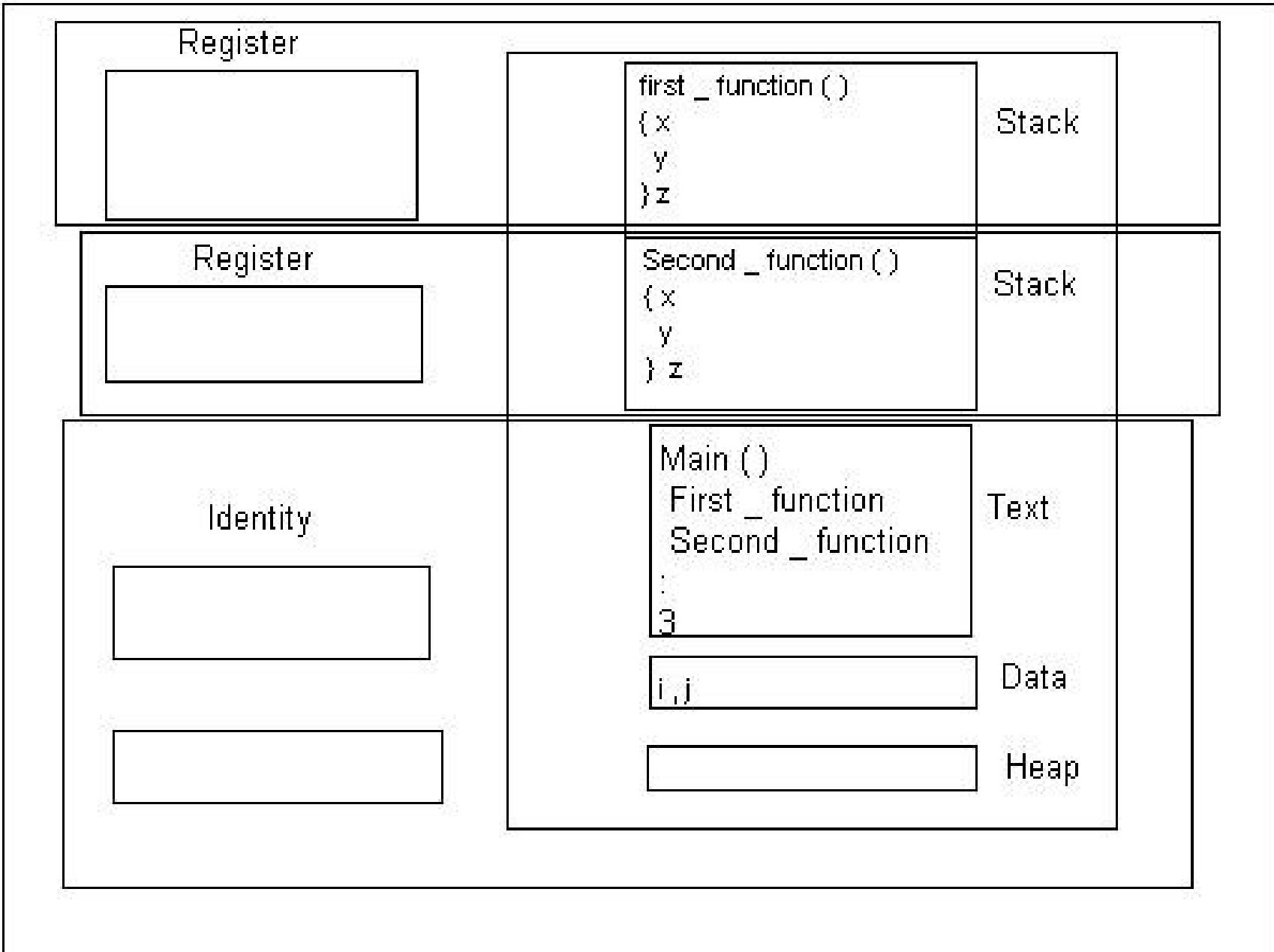
<http://www.coe.uncc.edu/~abw/parallel/pthreads/pthreads.html>

POSIX 线程详解: 一种支持内存共享的简单和快捷的工具

by Daniel Robbins

http://www.cn.ibm.com/developerWorks/linux/thread posix_thread1/index.shtml





线程调用—线程管理

POSIX

pthread_create

pthread_exit

pthread_kill

pthread_join

pthread_self

Solaris 2

thr_create

thr_exit

thr_kill

thr_join

thr_self



线程调用—线程同步和互斥

POSIX

pthread_mutex_init
pthread_mutex_destroy
pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_unlock
pthread_cond_init
pthread_cond_destroy
pthread_cond_wait
pthread_cond_timedwait
pthread_cond_signal
pthread_cond_broadcast

Solaris 2

mutex_init
mutex_destroy
mutex_lock
mutex_trylock
mutex_unlock



POSIX线程模型

- 线程管理：线程库用于管理线程。`Pthread_create()`生成新线程；`pthread_exit()`结束调用线程；`pthread_self()`返回调用线程的ID；`pthread_join`等待其他线程结束。
- 线程调度：`pThread_yield()`是调用者将处理器让位于其他线程；`pthread_cancel()`终止指定的线程。
- 线程同步：互斥变量和条件变量。



Pthreads实现计算π的实例 1

```
void main(argc,argv)
int argc;
char *argv[];
{
    int i;
/* check command line */
    if (argc != 3) {
        printf("Usage: %s  Num-intervals  Num-threads\n", argv[0]);
        exit(0);
    }
/* get num intervals and num threads from command line */
    n = atoi(argv[1]);
    num_threads = atoi(argv[2]);
    w = 1.0 / (double) n;
    pi = 0.0;
    tid = (pthread_t *) calloc(num_threads, sizeof(pthread_t));
/* initialize lock */
    if (pthread_mutex_init(&reduction_mutex, NULL))
        fprintf(stderr, "Cannot init lock\n"), exit(1);
/* create the threads */
    for (i=0; i<num_threads; i++)
        if(pthread_create(&tid[i], NULL, PIworker, NULL))
            fprintf(stderr,"Cannot create thread %d\n",i), exit(1);
/* join threads */
    for (i=0; i<num_threads; i++)
        pthread_join(tid[i], NULL);
    printf("computed pi = %.16f\n", pi);
}
```



Pthreads实现计算π的实例 2

```
void *PIworker(void *arg)
{
    int i, myid;
    double sum, mypi, w;
    /* set individual id to start at 0 */
    myid = pthread_self()-tid[0];
    /* integrate function */
    sum = 0.0;
    for (i=myid+1; i<=n; i+=num_threads) {
        w = 4.0*((double)i - 0.5);
        sum += f(w);
    }
    mypi = w*sum;
    /* reduce value */
    pthread_mutex_lock(&reduction_mutex);
    pi += mypi;
    pthread_mutex_unlock(&reduction_mutex);
    return(0);
}
```



对生产者驱动的有界缓冲区问题 的Pthread条件变量解

```
void *producer(void *arg1){  
    int i;  
    for (i=1;i<=SUMSIZE;i++){  
        pthread_mutex_lock(&slot_lock);  
        while(nsots<=0)  
            pthread_cond_wait(&slots,&slot_lock);  
        nsots--;  
        pthread_mutex_unlock(&slot_lock);  
        put_item(i*i);  
        pthread_mutex_lock(&item_lock);  
        nitems++;  
        pthread_cond_signal(&items);  
        pthread_mutex_unlock(&item_lock);  
    }  
    pthread_mutex_lock(&item_lock);  
    producer_done=1;  
    pthread_cond_broadcast(&items);  
    pthread_mutex_unlock(&item_lock);  
    return NULL;}
```



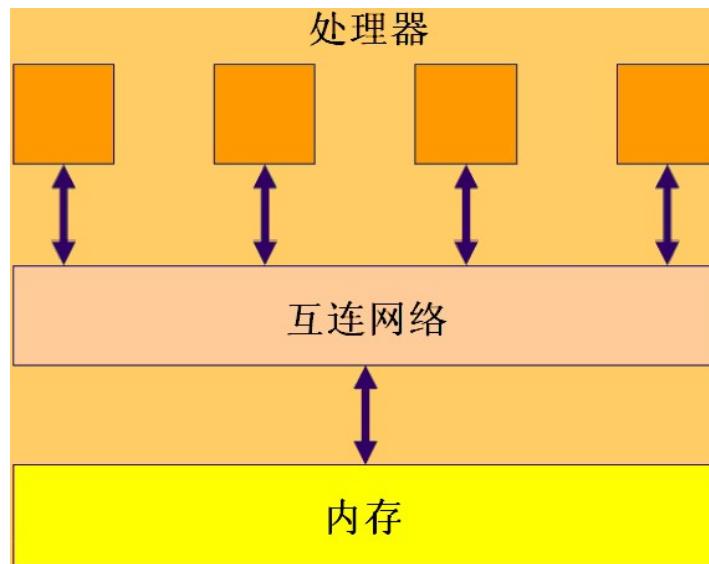
```
void *consumer(void *arg2){  
    int i,myitem;  
    for (;;){  
        pthread_mutex_lock(&item_lock);  
        while ((nitems<=0)&&!producer_done)  
            pthread_cond_wait(&items,&item_lock);  
        if ((nitems<=0)&&producer_done){  
            pthread_mutex_unlock(&item_lock);  
            break;  
        }  
        nitems--;  
        pthread_mutex_unlock(&item_lock);  
        get_item(&myitem);  
        sum+=myitem;  
        pthread_mutex_lock(&slot_lock);  
        nsots++;  
        cond_signal(&slots);  
        pthread_mutex_unlock(&slot_lock);  
    }  
    return NULL; }
```

共享存储系统编程

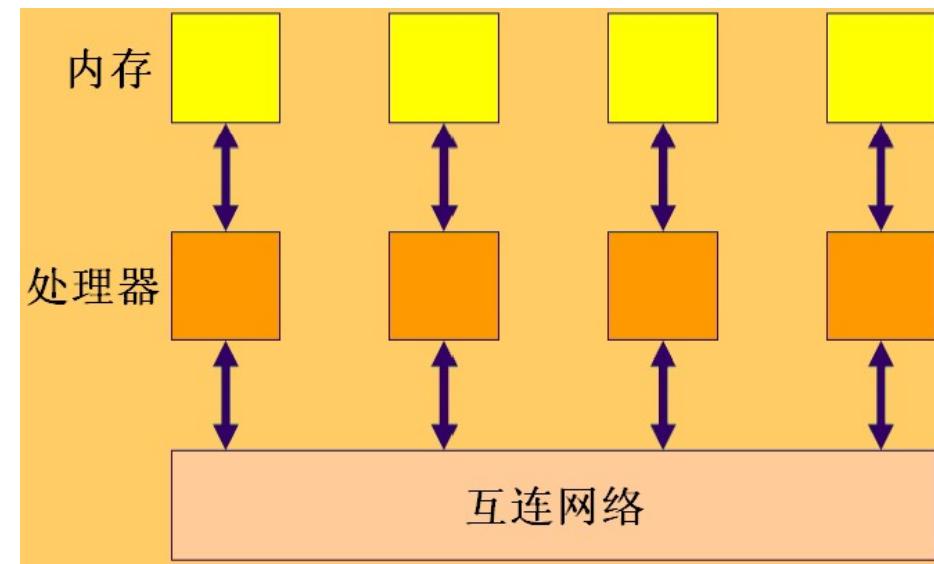
- ANSI X3H5 共享存储模型
- POSIX 线程模型
- OpenMP 模型



共享内存模式 vs 分布内存模式



共享内存模式



分布内存模式

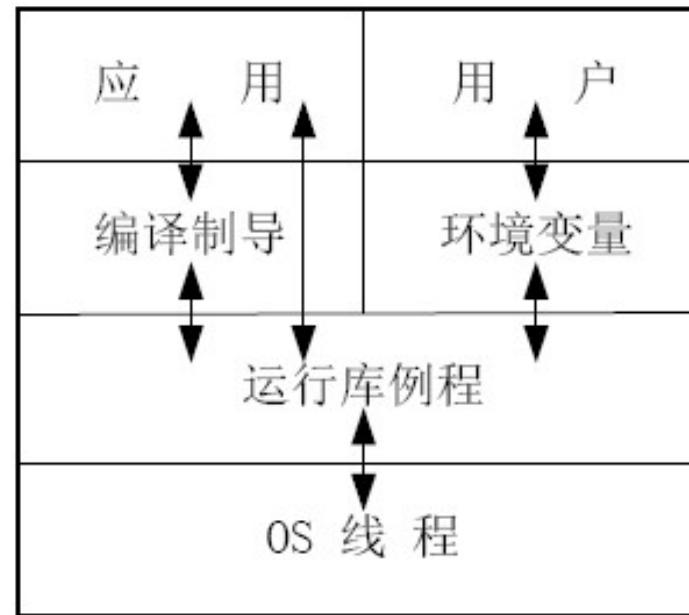


OpenMP是什么？

- Open Multi-Processing，开放多处理。
- 是一种支持多平台共享内存多处理编程的C、C++和Fortran语言API。
- 支持许多体系结构，包括Unix和Windows。
- 包含一组编译制导指令、库程序、和影响运行时行为的环境变量。
- 支持OpenMP的编译器包括Sun Compiler、GNU Compiler、Intel Compiler和Microsoft Visual C++等。
- 标准、简洁实用、使用方便、具有良好的可移植性。



OpenMP体系结构



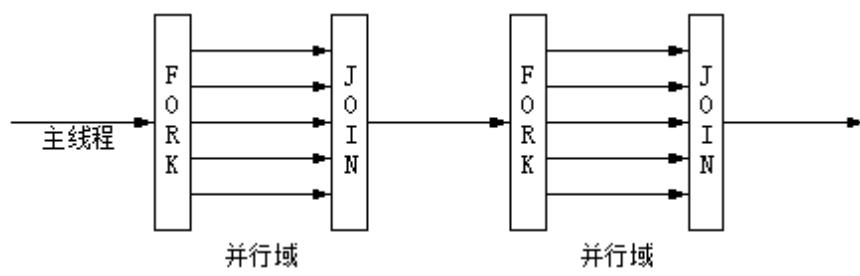
■ 三个API分量

- ◆ 编译制导(**Compiler Directives**)
- ◆ 运行库例程(**Routines Runtime Library**)
- ◆ 环境变量(**Environment Variables**)



OpenMP的编程模型

- 以线程为基础，通过编译制导语句来显示地指导并行化，为程序员提供了对并行化的完整控制。
- 采用Fork-Join（分叉-接合）的形式。



- 1) **Fork:** 主线程创建一队并行的线程，然后，并行域中的代码在不同的线程队中并行执行；
- 2) **Join:** 当主线程在并行域中执行完之后，它们或被同步或被中断，最后只有主线程在执行。

- 所有的OpenMP程序开始于一个单独的主线程(Master Thread)。
- 主线程会一直串行地执行，直到遇见第一个并行域 (Parallel Region)才开始并行执行。



编译制导语句

#pragma omp <directive> [clause[[,] clause]...]

newline

- #pragma（编译指示/注记）为编译指令，omp表示OpenMP；
- <directive>（指导/指示）部分就包含了具体的编译指导语句，包括：parallel、for、parallel for、section、sections、single、master、critical、flush、ordered和atomic；
- clause（子句）为可选的若干子句，子句间用逗号或白空符分隔；
- newline为换行符，每个OpenMP语句必须以换行符结束。



编译制导语句（2）

```
#pragma omp <directive> [clause[ [,] clause]...]
newline
```

- ◆ 整个指令语句必须写在同一行（中间不能换行）；
- ◆ 末尾必须以换行符结束（在换行符前，不能有其他语法内容）；
- ◆ 该指令只对后面的一个语句有效，若有多条语句，可以使用花括号括起来组成单个块语句{.....}；



编译制导语句（3）

OpenMP编译指导语句各部分的含义

#pragma omp	指令名directive-name	[子句clause, ...]	换行符newline
指导指令前缀。 对所有的 OpenMP语句都 需要这样的前缀。	指导指令。 在指导指令前缀和子句 之间必须有一个正确的 OpenMP指导指令。	子句。 在没有其它约束 条件下，子句可 无序，也可任选。 此部分也可没有。	换行符。 表明这条指导 语句的终止。



编译制导语句——作用域

- OpenMP编译指导语句的作用域有静态范围、孤立语句和动态扩展范围三种类型
 - ◆ 静态范围——文本代码在一个编译指导语句之后，被封装到一个结构块中。
 - ◆ 孤立语句——一个OpenMP的编译指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。



动态扩展范围

静态范围	孤立语句
for语句出现在一个封闭的并行域中	critical和sections语句出现在封闭的并行域之外
<pre>#pragma omp parallel { ... #pragma omp for for(...) { ... sub1(); ... } ... sub2(); ... }</pre>	<pre>void sub1() { ... #pragma omp critical ... } void sub2() { ... #pragma omp sections ... }</pre>



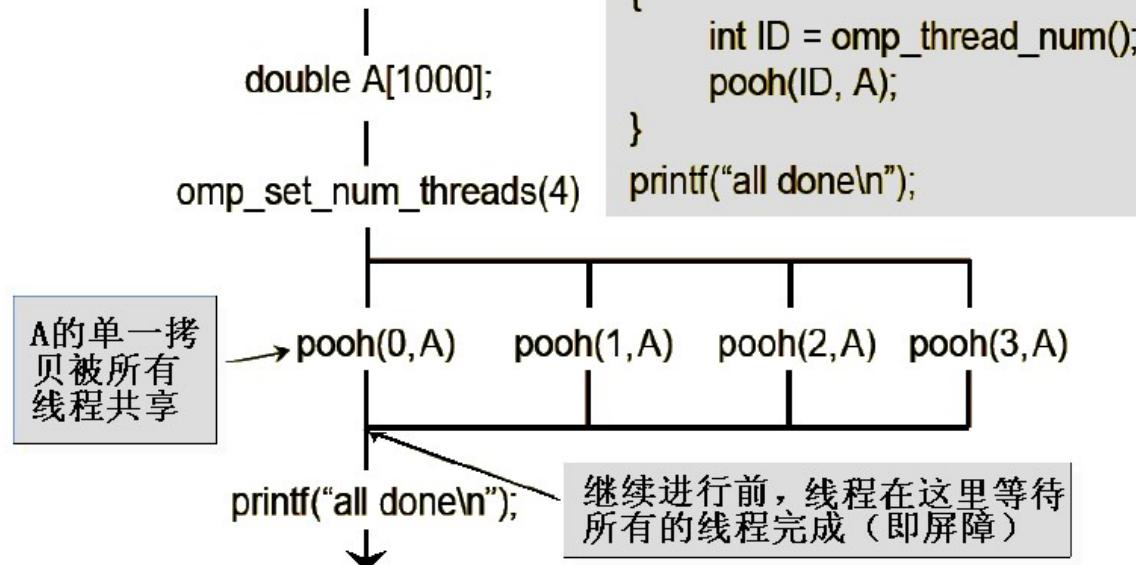
南
South

编译制导语句——并行区域

由OpenMP的parallel语句定义的并行控制语句块，并行区域中的代码被所有的线程重复执行。

OpenMP: 并行区域

- 每个线程重复地执行同样的代码



编译制导语句——并行区域

#pragma omp parallel [子句 [子句]...] 换行符
语句[块]

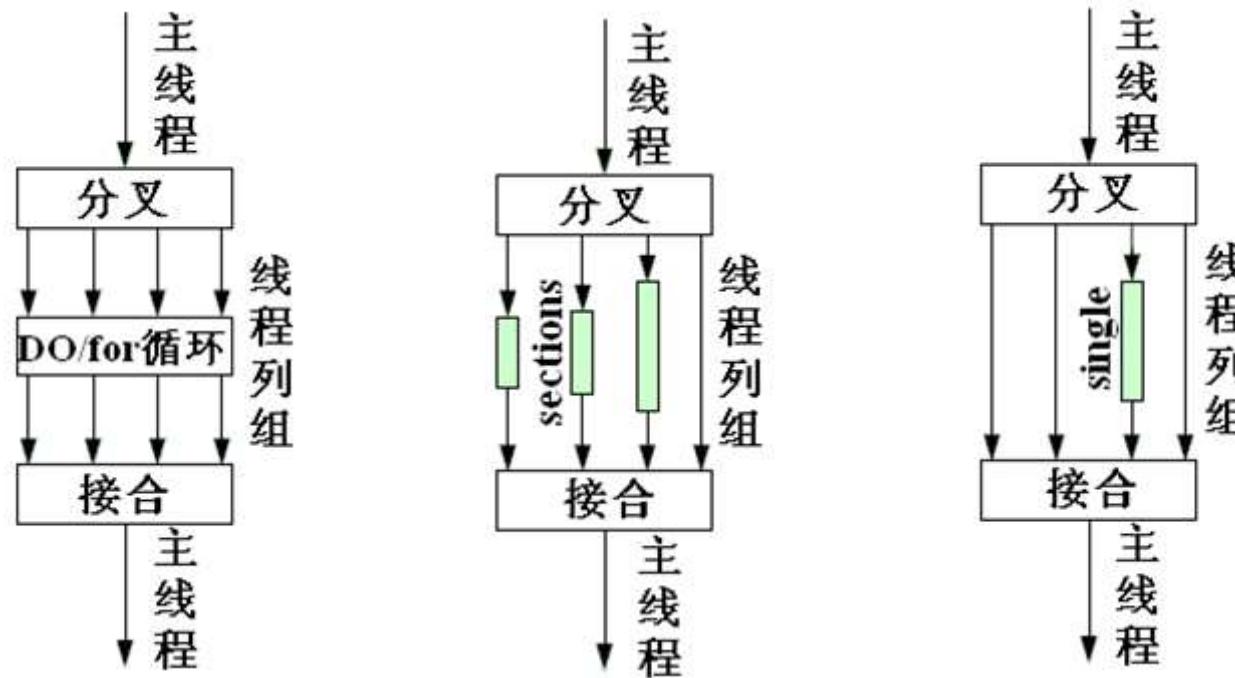
其中，子句可为：

- if(scalar-expression)
- private(list)
- firstprivate(list)
- default(shared | none)
- shared(list)
- copyin(list)
- reduction(operator: list)
- num_threads(integer-expression)



编译制导语句——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，包括：并行for循环、并行sections和串行执行。



共享任务结构将代码划分给线程组各成员执行



Parallel v.s. Parallel for

- **Parallel** 复制执行方式，将代码在所有的线程内各执行一次；
- **parallel for**则是采用工作分配执行方式，将循环需做的所有工作量，按一定的方式分配给各个执行线程，全部线程执行工作的总合等于原先串行执行所完成的工作量。



Parallel v.s. Parallel for

```
#pragma omp parallel num_threads(2)
    for ( int i = 0; i < 5; i++ )
        printf("hello world! i=%d\n",i);
```

parallel语句的每次输出都一样



```
hello world! i=0
hello world! i=0
hello world! i=1
hello world! i=1
hello world! i=2
hello world! i=2
hello world! i=3
hello world! i=3
hello world! i=4
hello world! i=4
请按任意键继续. . .
```

```
#pragma omp parallel for num_threads(2)
    for ( int i = 0; i < 5; i++ )
        printf("hello world!
i=%d\n",i);
```

parallel for的则可能不同



```
hello world! i=0
hello world! i=3
hello world! i=1
hello world! i=4
hello world! i=2
请按任意键继续. . .
```



```
hello world! i=2
hello world! i=3
hello world! i=0
hello world! i=4
hello world! i=1
请按任意键继续. . .
```



并行**sections**

- 工作分区（**sections**）是指利用OpenMP的**sections**编译指导语句；
- 将用**section**语句指定的内部代码，划分成多个工作区分配给线程组中的各个线程；
- 不同的**section**由不同的线程执行；
- 各线程工作量的总合等于原来的工作量。



并行sections (2)

```
#pragma omp sections [子句 [[,]子句]...]换行符
{
    #pragma omp section newline
    ...
    #pragma omp section newline
    ...
}
```

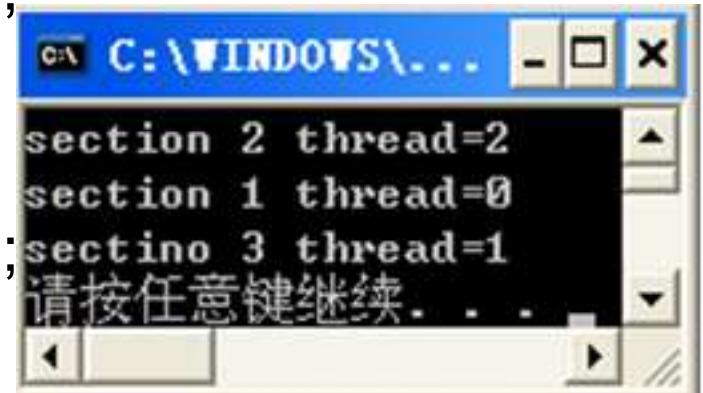
■ 其中的子句可为: private (list), firstprivate (list),
, lastprivate (list), reduction (operator: list),
nowait

■ 在sections语句结束处有一个隐含的屏障, 使用
了nowait子句除外。



并行sections示例1

```
#pragma omp parallel sections
{
    #pragma omp section
    printf("section 1
thread=%d\n",omp_get_thread_num());
    #pragma omp section
    printf("section 2
thread=%d\n",omp_get_thread_num());
    #pragma omp section
    printf("sectino 3
thread=%d\n",omp_get_thread_num());
}
```



并行sections示例2

```
#pragma omp parallel sections
```

```
{
```

```
    #pragma omp section /* 可选 */
```

```
    v = alpha();
```

```
    #pragma omp section
```

```
    w = beta();
```

```
}
```

```
#pragma omp section
```

```
y = delta();
```

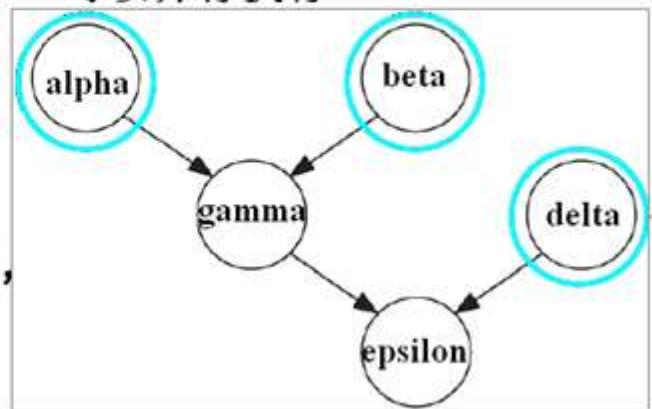
```
x = gamma(v, w);
```

```
printf ("%6.2f\n", epsilon(x,y));
```

串行工作流程

- v = alpha();
- w = beta();
- x = gamma(v, w);
- y = delta();
- printf ("%6.2f\n", epsilon(x, y));

■ alpha、beta和delta可以并行执行



Single

```
#pragma omp single [clause[,]clause]...] newline
```

其中子句可为：

`private(list)`

`firstprivate(list)`

❑ **single**编译指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。



Single示例

```
void work1() { printf("work1在运行。\\n");}  
void work2() { printf("work2在运行。\\n");}  
int _tmain(int argc, _TCHAR* argv[])  
{  
    #pragma omp parallel num_threads(2)  
    {  
        #pragma omp single  
        printf("开始wok1。\\n");  
        work1();  
        #pragma omp single  
        printf("结束work1。\\n");  
        #pragma omp single nowait  
        printf("结束work1并且开始work2。\\n");  
        work2();  
    }  
    return 0;  
}
```



OpenMP的线程同步机制

- 在多线程执行的程序中，出现数据竞争时如何能够得到正确结果。

?



数据竞争

```
int max_num = -1;  
#pragma omp parallel for  
for ( int i = 0; i < N; i++ )  
    if ( a[i] > max_num )  
        max_num = a[i];
```

- 由于max_num是共享变量，多线程同时执行循环时，可能会出现错误的结果。
 - 例如N=2, a[0]=5、a[1]=2，由两个线程各执行一个if语句；
 - 如果执行i=1的线程在完成比较操作后被挂起，此时执行i=0的线程完成比较和赋值（max_num = 5），接着执行i=1的线程被唤醒，因为比较操作已经完成，所以继续进行赋值操作（max_num = a[i];）；
 - 最后的结果是max_num = 2。



线程同步机制——互斥锁机制

■ 三种不同的互斥锁机制

- ◆ 临界区（critical）；
- ◆ 原子操作（atomic）；
- ◆ 由库函数来提供的若干同步操作函数omp_*_lock

■ 线程同步语句

- ◆ master、critical、barrier、atomic、flush和ordered等指导语句。



Master 指导语句

- 指定代码段只有主线程执行

- `#pragma omp master newline`



Critical指导语句

- 表明域中的代码一次只能由一个线程执行，而其他线程被阻塞在临界区。
- `# pragma omp critical [name] newline`
 - name为需加锁的变量名
- 在程序需要访问可能产生竞争的内存数据时，都需要插入相应的临界区代码。



Critical示例

```
int max_num = -1;  
#pragma omp parallel for  
for ( int i = 0; i < N; i++ )  
    #pragma omp critical (max_num)  
    if ( a[i] > max_num )  
        max_num = a[i];
```



Barrier 指导语句

- 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，而不能只是一个单行的指导语句。
- `#pragma omp barrier newline`



Barrier示例（1）

```
#pragma omp parallel for share (A, B, C) // 设置共享变量
{
    f(A, B);
    printf("处理过的A进入B\n");
    #pragma omp barrier // 等待修改过的B
    f(B, C);
    printf("处理过的B进入C\n");
}
```



Barrier 示例 (2)

错误

```
if (x != 0)  
    # pragma omp barrier  
    f(x);
```

正确

```
if (x != 0)  
{  
    #pragma omp barrier  
    f(x);  
}
```



Atomic 指导语句

- 指定特定的存储单元将被原子更新。
- 原子操作是OpenMP编程方式给同步编程带来的特殊的编程功能，通过编译指导语句的方式直接获取了现在多处理器计算机体系结构的功能。
- `#pragma omp barrier newline`



Atomic 示例

- 只能作用在语言内建的基本数据结构。

`#pragma omp atomic
x <binop>= expr`

OR

`#pragma omp atomic
x++ // 或 x--, --x, ++x`

- 其中：

- 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；
- $binop$ （二进制操作）是 $+, *, -, /, \&, ^, |, >>, or <<$ 之一，且不被重载。



Atomic 示例 (2)

```
int counter=0;  
#pragma omp parallel  
{  
    for(int i=0;i<10000;i++)  
        #pragma omp atomic // 原子操作  
        counter++;  
}  
printf("counter = %d\n", counter);
```



flush 指导语句

■ 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。

■ `#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`

`for`:退出部分

`critical`:进入与退出部分

`sections`:退出部分

`ordered`:进入与退出部分

`single`:退出部分

`parallel`:退出部分



flush 示例

```
int iam, neighbor;
int sync[NUNBER_OF_THREADS];
float work[NUNBER_OF_THREADS];
#pragma omp parallel private(iam, neighbor)
shared(work, sync)
{
    iam = omp_get_thread_num();
    sync[iam] = [0];
    #pragma omp barrier
    // 计算我的work数组部分
    work[iam] = .....;
    // 宣告我 (I am) 在干我的活work, 第一个flush确保我的工作
    // 在sync之前成为可见, 第二个flush确保sync成为可见。
    #pragma omp flush(work)
    sync[iam] = 1;
    #pragma omp flush(sync)
    // 等待neighbor
    neighbor = (iam > 0 ? iam : omp_get_num_threads()) - 1;
    while (sync[neighbor] == 0) {
        #pragma omp flush(sync)
    }
    // 读入neighbor的work数组值
    ..... = work[neighbor];
}
```



ordered 指导语句

- 指出其所包含循环的执行在任何时候只能有一个线程执行被**ordered**所限定的部分。
 - 只能出现在**for** 或者**parallel for**语句的动态范围内。
-
- `#pragma omp ordered newline`



ordered 示例

```
void work(int k)
{
    #pragma omp ordered
    printf(" %d", k);
}

.....
#pragma omp for ordered schedule(dynamic)
for (i = lb; i < ub; i += st)
    work(i);
```



threadprivate 指导语句

- 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。
- `#pragma omp threadprivate (list) newline`



threadprivate 示例

```
int alpha[10], beta[10], i;  
#pragma omp threadprivate(alpha)  
int main ()  
{  
    // 第一个并行区域  
    #pragma omp parallel private(i,beta)  
    for (i=0; i < 10; i++)  
        alpha[i] = beta[i] = i;  
    // 第二个并行区域  
    #pragma omp parallel  
    printf("alpha[3]= %d and beta[3]=%d\n",  
          alpha[3], beta[3]);
```



数据域属性子句

用于指定变量的作用域范围。

- ◆ **private**子句；
- ◆ **shared**子句；
- ◆ **default**子句；
- ◆ **firstprivate**子句；
- ◆ **lastprivate**子句；
- ◆ **copyin**子句；
- ◆ **reduction**子句；



private子句

■ 表示它列出的变量（可能局部）对于每个线程是局部的。

- 语句格式为： 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。
 - ◆ 语句格式为: **shared (list)**。



default子句

■ 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。

- ◆ default (shared | none)。

```
int x, y, z[1000];
#pragma omp threadprivate(x)
void fun(int a) {
    const int c = 1;
    int i = 0;
    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_thread(); // 正确！因为j声明在并行区域内
        a = z[j]; // 正确！因为a和z分别被列在private和shared子句中
        x = c; // 正确！因为x是threadprivate, c是常量限定类型
        z[i] = y; // 错误！不能在这里引用i或y
    }
}
```



firstprivate子句

- 是private子句的超集，用于对变量做原子初始化。
 - firstprivate (list)。

```
incr = 0;  
#pragma omp parallel for firstprivate(incr)  
for ( i = 0; i < MAX; i++) {  
    if ( (i%2) == 0) incr++;  
    a[i] = incr;  
}
```



lastprivate子句

- 也是private子句的超集，用于将变量从最后的循环迭代或段复制给原始的变量。
 - lastprivate (list)。

```
void sq2(int n, double *last)
{
    double x;
    #pragma omp parallel
    #pragma omp for lastprivate(x)
    for (int i = 0; i < n; i++) {
        x = a[i] * a[i] + b[i] * b[i];
        b[i] = sqrt(x);
    }
    last = x;
}
```



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。
 - ◆ 语句格式为： copyin(list)。



reduction子句

- 使用指定的操作对其列表中出现的变量进行归;
- 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归约，并更新该变量的全局值。
- 语句格式为： reduction (operator: list)
- **#pragma omp parallel for reduction (op: x)**
 - ◆ $x = x \text{ op } expr$
 - ◆ $x = expr \text{ op } x$ (except subtraction)
 - ◆ $x \text{ binop} = expr$
 - ◆ $x++$ 、 $++x$ 、 $x--$ 、 $--x$
 - ◆ 其中： 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 $+$, $*$, $-$, $/$, $\&$, $^$, $|$, $\&\&$, 或 $\|$ 之一，且不被重载。



Reduction示例

```
#include <omp.h>
int main ()
{
    int i, n, chunk;
    float a[100], b[100], result;
    /* 初始化 */
    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 reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
```



子句与编译指导语句小结

编译指导 子句	parallel	DO/for	sections	single	parallel DO/for	parallel sections
if	√				√	√
private	√	√	√	√	√	√
shared	√	√			√	√
default	√				√	√
firstprivate	√	√	√	√	√	√
lastprivate		√	√		√	√
reduction	√	√	√		√	√
copyin	√				√	√
schedule		√			√	
ordered		√			√	
nowait		√	√	√		



语句绑定规则

- 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。



语句嵌套规则

- **PARALLEL** 语句动态地嵌套到其它语句中，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

- OpenMP标准定义了一个应用编程接口（API）来调用库中的多种函数。

```
omp_get_num_threads() // 获取线程数  
omp_get_max_threads() // 获取最大线程数  
omp_get_thread_num() // 获取当前线程组中的线程数  
omp_get_num_procs() // 获取处理器（核）数  
omp_in_parallel() // 判断是否在并行区域中  
omp_set_dynamic(dynamic_threads) // 设置并行域可动态执行的线程数  
omp_get_dynamic() // 获取并行域可动态执行的线程数  
omp_set_nested(nested) // 设置是否允许并行嵌套  
omp_get_nested() // 获取是否允许并行嵌套  
omp_init_lock(omp_lock_t *lock) // 初始化互斥锁  
omp_init_nest_lock(omp_nest_lock_t *lock) // 初始化嵌套互斥锁  
omp_destroy_lock(omp_lock_t *lock) // 销毁互斥锁  
omp_destroy_nest_lock(omp_nest_lock_t *lock) // 销毁嵌套互斥锁  
omp_set_lock(omp_lock_t *lock) // 设置互斥锁  
omp_set_nest_lock(omp_nest_lock_t *lock) // 设置嵌套互斥锁  
omp_unset_lock(omp_lock_t *lock) // 释放互斥锁  
omp_unset_nest_lock(omp_nest_lock_t *lock) // 释放嵌套互斥锁  
omp_test_lock(omp_lock_t *lock) // 测试互斥锁  
omp_test_nest_lock(omp_nest_lock_t *lock) // 测试嵌套互斥锁  
omp_get_wtime() // 获取等待时间的秒数  
omp_get_wtick() // 获取处理器（核）时钟的秒数
```



OpenMP的互斥锁操作库函数

函数名称	描述
void omp_init_lock(omp_lock_t *)	初始化一个互斥锁
void omp_destroy_lock(omp_lock_t*)	结束一个互斥锁的使用并释放内存
void omp_set_lock(omp_lock_t *)	获得一个互斥锁
void omp_unset_lock(omp_lock_t *)	释放一个互斥锁
int omp_test_lock(omp_lock_t *)	试图获得一个互斥锁，并在成功是返回真 (true) , 失败是返回假 (false)



互斥锁操作示例

```
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);
    // 该printf语句每次只能被一个线程执行
    printf("我的线程ID为%d。 ", id);
    omp_unset_lock(&lck);
    while(!omp_test_lock(&lck)) {
        skip(id); // 我们不拥有互斥锁， 所以得干点其他事
    }
    work(id); // 我们拥有了互斥锁， 所以可以干活
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```



环境变量

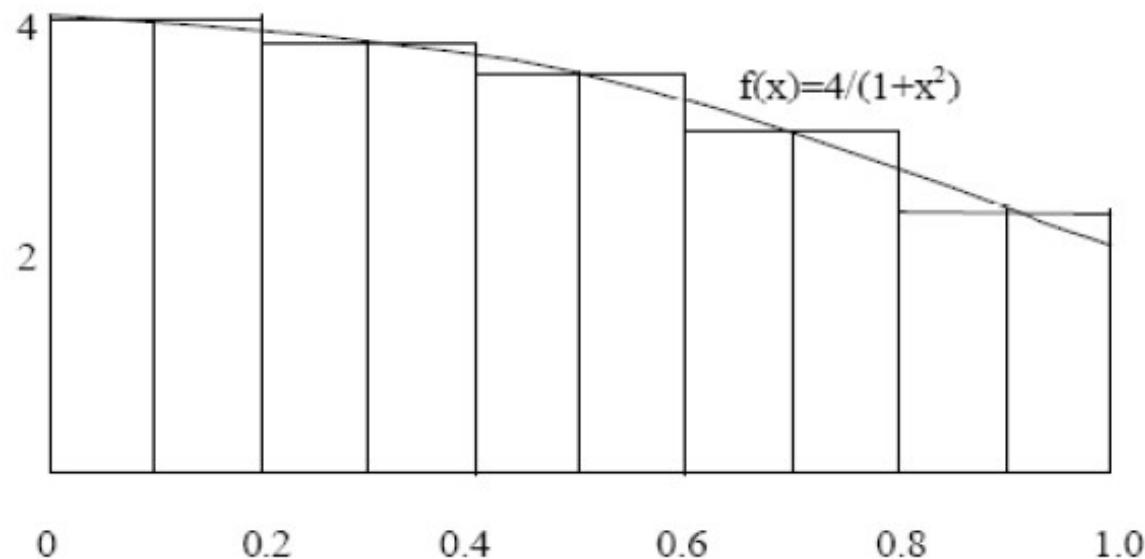
- **OMP_SCHEDULE**: 只能用到for和parallel for中。它的值就是处理器中循环的次数；
- **OMP_NUM_THREADS**: 定义执行中最大的线程数；
- **OMP_DYNAMIC**: 通过设定变量值**TRUE**或**FALSE**,来确定是否动态设定并行域执行的线程数；
- **OMP_NESTED**: 确定是否可以并行嵌套。



计算 π 实例

下面我们用矩形法则的数值积分方法来估算 π 的值

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



计算π实例——串行程序

```
static long num_steps = 100000;  
double x, pi, sum = 0.0, step = 1.0/(double) num_steps;  
for (long i = 1; i <= num_steps; i++){  
    x = (i + 0.5)*step;  
    sum += 4.0/(1.0 + x*x);  
}  
pi = step * sum;  
printf("Pi = %f\n", pi);
```



计算π实例——并行区域

```
#define NUM_THREADS 2
static long num_steps = 100000;
double x, pi = 0.0, sum[NUM_THREADS],
        step = 1.0/(double)num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    sum[id] = 0.0;
    for (long i = id; i < num_steps; i += NUM_THREADS) {
        x = (i + 0.5)*step;
        sum[id] += 4.0/(1.0 + x*x); // 根据ID号分配任务
    }
}
for(int i = 0; i < NUM_THREADS; i++) pi += sum[i] * step;
printf("Pi= %f\n", pi);
```



计算π实例——循环并行化

```
#define NUM_THREADS 2
static long num_steps = 100000;
double x, pi = 0.0, sum[NUM_THREADS],
        step = 1.0/(double)num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    sum[id] = 0.0;
    #pragma omp for
    for (long i = 0; i < num_steps; i++)
    {
        x = (i + 0.5)*step;
        sum[id] += 4.0/(1.0 + x*x); // 存储每次的结果
    }
}
for(int i = 0; i < NUM_THREADS; i++) pi += sum[i] * step;
printf("Pi = %f\n", pi);
```



计算π实例——私有化和临界区

```
#define NUM_THREADS 2
static long num_steps = 100000;
double x, pi = 0.0, sum, step = 1.0/(double)num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, sum)
{
    int id = omp_get_thread_num();
    sum = 0.0;
    for (long i = id; i < num_steps; i += NUM_THREADS)
    {
        x = (i + 0.5)*step;
        sum += 4.0/(1.0 + x*x);
    }
    #pragma omp critical
    pi += sum*step;
}
printf("Pi= %f\n", pi);
```



OpenMP的作用

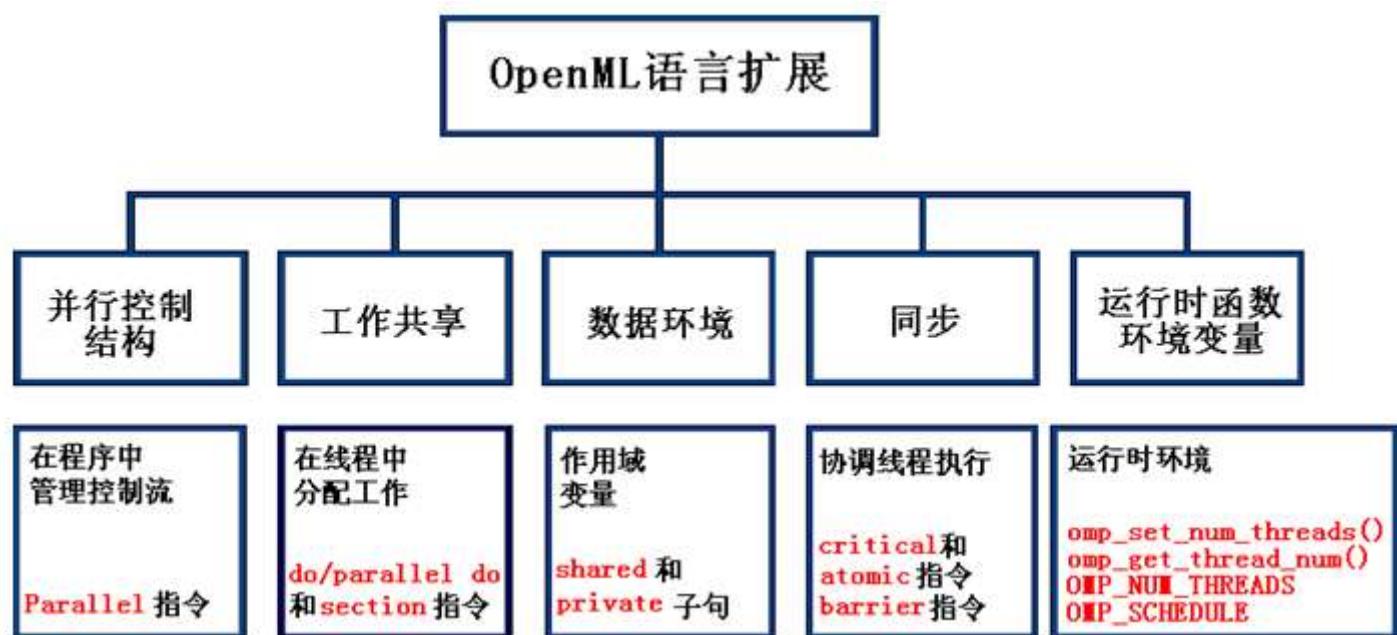
■ 提供了对并行算法的高层的抽象描述

- ◆ 程序员通过在源代码中加入各种专用的**pragma**指令（**directive**, 指示/命令）来指明自己的意图；
- ◆ 由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。
- ◆ 当选择忽略这些**pragma**，或者编译器不支持OpenMP时，程序又可退化为通常的（串行）程序，代码仍然可以正常运作，只是不能利用多线程来加速程序执行。



运行时库函数

原本用以设置和获取执行环境相关的信息，它们当中也包含一系列用以同步的API，支持运行时对并行环境的改变和优化，给编程人员足够的灵活性来控制运行时的程序运行状况。



OpenMP的好处

- 提供的这种对于并行描述的高层抽象降低了并行编程的难度和复杂度。
- 这样程序员可以把更多的精力投入到并行算法本身，而非其具体实现细节。
- 对基于数据分集的多线程程序设计，OpenMP是一个很好的选择。
- 同时，使用OpenMP也提供了更强的灵活性，可以较容易的适应不同的并行系统配置。
- 线程粒度和负载平衡等是传统多线程程序设计中的难题，但在OpenMP中，OpenMP库从程序员手中接管了部分这两方面的工作。



OpenMP的坏处

- 作为高层抽象，OpenMP并不适合需要复杂的线程间同步和互斥的场合。
- 另一个缺点是不能在非共享内存系统（如计算机集群，MPI使用较多）上使用。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。



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

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

串行程序

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

并行程序

