北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

并行编程原理与实践

6. OpenMP编程

王一拙、计卫星

北京理工大学计算机学院

目录 | CONTENTS

1 OpenMP简介

2 并行域

3 工作共享

4 数据环境

5 同步

6 库函数和环境变量

1 OpenMP简介

什么是OpenMP？

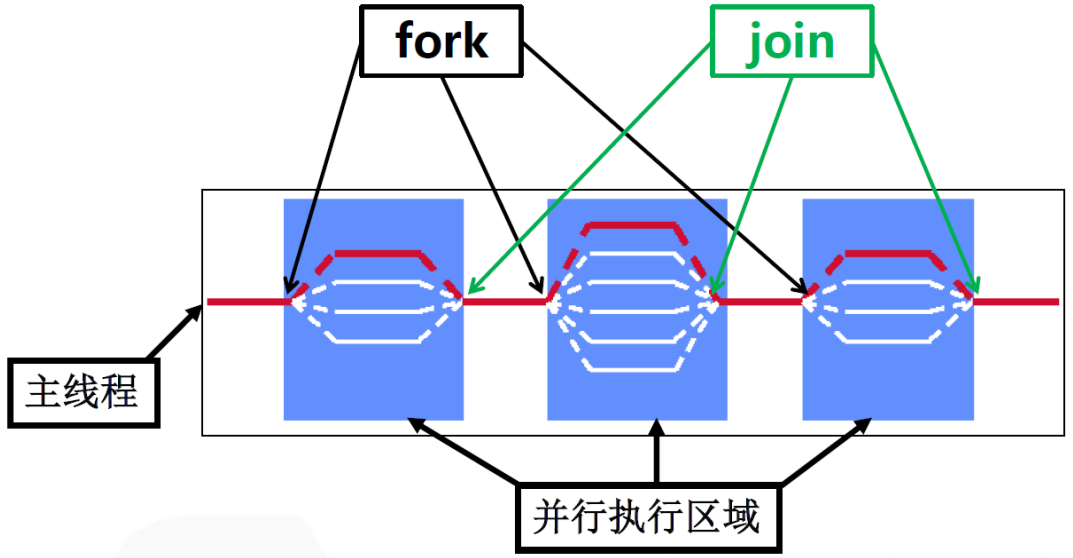
- OpenMP是面向共享存储系统的一个并行编程模型
- 已成为多线程编程的工业标准API
- 由一系列编译制导语句（Compiler Directive）、运行时库函数（Runtime Library Routines）和环境变量（Environment Variables）组成
- 使得Fortran, C and C++的多线程编程更加容易
- 编程简单，增量化并行，移植性好，可扩展性好

OpenMP™

www.openmp.org

OpenMP并行执行模式

- OpenMP是基于线程的并行编程模型
- OpenMP采用Fork-Join并行执行模式



OpenMP程序示例 — C

```c
#include <omp.h>
#include <stdio.h>
int main() {
  int nthreads,tid;
  #pragma omp parallel private(nthreads,tid) {
    tid=omp_get_thread_num();
    printf("Hello, world from OpenMP thread %d\n", tid);
    if (tid==0) {
      nthreads=omp_get_num_threads();
      printf(" Number of threads %d\n", nthreads);
    }
  }
  return 0;
}
```

- OpenMP 编译制导语句标识符 #pragma omp
- 头文件：omp.h

- 编译

```
gcc -fopenmp hello.c
icc -openmp hellp.c
```

OpenMP程序示例 — Fortran

```
program hello
use omp_lib
implicit none
integer :: tid, nthreads

!$omp parallel private(tid)
    tid = omp_get_thread_num()
    write(*,100) "Hello, world from OpenMP thread ", tid
    if (tid==0) then
        nthreads=omp_get_num_threads();
        write(*,100) "Number of threads ", nthreads
    endif
!$omp end parallel

100 format(1X,A,I1,/)
end
```

- OpenMP 编译制导语句标识符 !$omp
- 模块：omp_lib

- 编译

```
gfortran -fopenmp hello.f90
ifort -openmp hello.f90
```

OpenMP程序示例 — Fortran

```
program hello
use omp_lib
implicit none
integer :: tid, nthreads
```

- OpenMP 编译制导语句标识符 !$omp
- 模块：omp_lib

```
Hello World from OpenMP thread 2
Hello World from OpenMP thread 0
Number of threads 4
Hello World from OpenMP thread 3
Hello World from OpenMP thread 1
```

```
!$omp end parallel

100 format(1X,A,I1,/)
end
```

- 编译

```
gfortran -fopenmp hello.f90
ifort -openmp hello.f90
```

OpenMP常用于循环并行化

- 串行程序中最耗时的部分往往是循环
- 在串行程序中加上编译制导语句实现并行化

```
#define N 10000
void main()
{
  double A[N],B[N],C[N];
  ...
  for(int i=0; i<N; i++)
    C[i]=A[i]+B[i];
}
```

```
#include <omp.h>
#define N 10000
void main()
{
  double A[N],B[N],C[N];
  ...
  #pragma omp parallel for
  for(int i=0; i<N; i++)
    C[i]=A[i]+B[i];
}
```

编译制导语句

- OpenMP 通过对串行程序添加编译制导语句实现并行化
- 编译制导语句由制导指令前缀、制导指令和子句三部分构成，格式：

#pragma omp	directive-name	[clause, ...]
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。

编译制导指令大致分四类

- 并行域指令：
  - 生成并行域，即产生多个线程以并行执行任务
  - 所有并行任务必须放在并行域中才可能被并行执行
- 工作共享指令：
  - 负责任务划分，并分发给各个线程
  - 工作共享指令不能产生新线程，因此必须位于并行域中
- 同步指令：负责并行线程之间的同步
- 数据环境：负责并行域内的变量的属性（共享或私有），以及边界上（串行域与并行域）的数据传递

- **大多OpenMP编译制导语句作用于其后的一个结构块**
  - 结构块：仅有一个入口（顶端）和一个出口（底端）的一块儿语句，没有跳转到块外的分支
  - 例外：Fortran的STOP语句和C/C++的exit() 允许出现在结构块中

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

  if (conv (res[id]) goto more;
}
printf (“All done\n”);
```

A structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

Not a structured block

2 并行域

- 并行域内的代码将被多个线程并行执行
- 在并行域结尾有一个隐式同步（barrier）
- 子句（clause）用来说明并行域的附加信息，若有多个，则用空格隔开
- 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)

Fortran	!$omp parallel [clause clause ...] *structured-block* !$omp end parallel

C/C++	#pragma omp parallel [clause clause ...]
	{
	structured-block
	}

并行域可以嵌套

```
omp_set_nested(1);     //设置允许并行嵌套
#pragma omp parallel private(out_tid)      {
    out_tid = omp_get_thread_num();      // 获取外层线程id
    printf("Hello World from out_thread = %d\n", out_tid);
    if (out_tid == 0) {    // 主要线程的线程id为0，因此只有主线程执行该代码
        out_nthreads = omp_get_num_threads();
        printf("Number of out_threads = %d\n", out_nthreads);
    }
    #pragma omp parallel private(in_tid) {
        in_tid = omp_get_thread_num();     // 获取内层线程id
        printf("Hello World from in_thread = %d, out_thread = %d\n", in_tid, out_tid);
        if (in_tid == 0) {    // 主要线程的线程id为0，因此只有主线程执行该代码
            out_nthreads = omp_get_num_threads();
            printf("Number of in_threads = %d, out_thread = %d\n", in_nthreads, out_tid);
        }
    }
} // 所有线程join主线程并结束
```

3 工作共享

3 工作共享

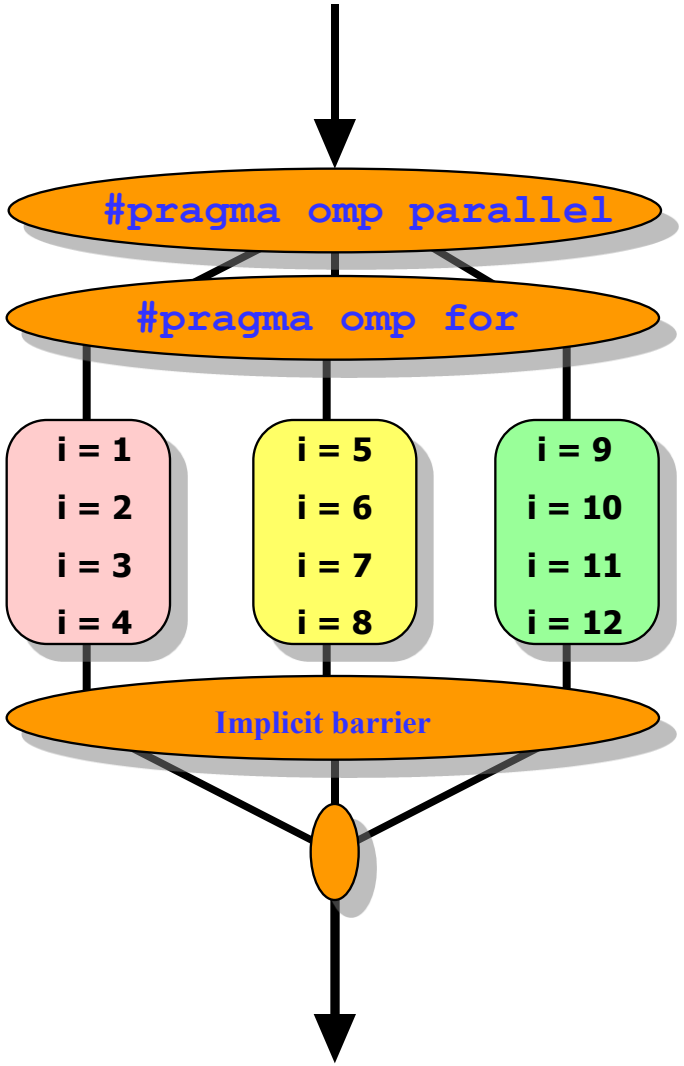
- **OpenMP通过工作共享（Work-sharing）的编译制导指令将任务划分和分配给多个线程并行执行**
- **工作共享指令主要分为三类：**
  - omp for：负责循环任务的划分和分配
  - omp sections：指定一个并行区域，其中包含多个可并行执行的结构块
  - omp task：显式定义任务，放入任务对待等待执行

- **工作共享指令使用时要注意：**
  - 工作共享指令不负责并行域的产生和管理，必须使用#pragm omp parallel
  - 工作共享指令区域如果不放在并行域内，则只会被一个线程串行执行
  - 结尾处有隐式barrier同步（也可以不同步，需用nowait子句）

工作共享 — omp for

omp for 编译制导

```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];
```



- 给每个线程分配一组循环迭代，循环迭代之间没有依赖关系
- 循环执行结束时有隐式同步

工作共享 — omp for

- for语句指定紧随它的循环语句由线程池中的线程并行执行
- 具体格式：

#pragma omp for [clause[[,]clause]…] newline
[clause]=
Schedule(type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
nowait

Fortran	**!$omp do [clause clause ...]** ***do-loops*** **!$omp end do**
C/C++	**#pragma omp for [clause clause ...]** **{** ***for-loops*** **}**

工作共享 — omp for

- 可以将并行域和工作共享结合成一条编译制导语句

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
       res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
          res[i] = huge();
    }
```

private子句

- 指定一个或多个变量为私有变量，即在每个线程中都创建一个同名局部变量，
- 变量没有初始值，如果是C++对象会调用默认构造函数
- 循环控制变量默认是线程私有的
- 格式：private(list)

```
void* work(float* c, int N) {
  float x, y; int i;
  #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
     }
}
```

schedule子句

- schedule子句描述如何将循环的迭代划分给线程池中的线程
- 格式：schedule(type[,size])
  - type参数表示调度策略类型
  - size参数表示块大小（多少个循环迭代），必须是整数
- 调度策略类型：
  - static：静态分配，任务块的大小不变，由 size指定
  - dynamic：动态分配，任务块的大小不变，由 size指定
  - guided：动态分配，任务块的大小变化，最小块大小由size指定
  - runtime：具体调度方式到运行时才能确定

- **schedule(static, size)**
  - 省略size，循环迭代空间被划分成相同（近似）大小的区域，每个线程分配一个区域
  - 指定size，迭代空间被划分为很多size大小的块，然后这些块被轮转的分配给各个线程

例：假如线程数为 4，总任务量为 40，则

schudule(static)

T0	T1	T2	T3

schudule(static, 4)

T0	T1	T2	T3	T0	T1	T2	T3	T0	T1

3 工作共享 — omp for

- **schedule(dynamic, size)**
  - 划分循环迭代空间为size大小的块，然后基于先来先服务方式分配给各线程
  - 省略size时，其默认值为1
- **schedule(guided, size)**
  - 类似于dynamic调度，但开始分配的块比较大，之后越来越小，使用GSS（Guided Self-Scheduling）调度算法
  - size说明最小的块大小，省略size时，其默认值为1
- **schedule(runtime, size)**
  - 调度方式取决于环境变量OMP_SCHEDULE的值
  - 使用runtime时，指明size是非法的

```
export OMP_SCHEDULE=DYNAMIC, 4；
```

3 工作共享 — omp sections

```
#pragma omp sections clause1 clause2 ...
{
    #pragma omp section
    结构块
    #pragma omp section
    结构块
}
```

- 两个#pragma omp section之间的代码称为一个section
- 可以定义任意多的section，这些section被分配给多个线程同时执行
- 每个section仅被一个线程执行一次
- 线程数多于section数量时，每个线程最多执行一个section
- 线程数少于section数量时，每个线程执行一个以上section

3 工作共享 — omp sections

- omp sections必须包含在omp parallel并行域以内
- omp sections后的第一个omp section语句可以省略

#pragma omp sections 接受的子句
private(*list*)
firstprivate(*list*)
lastprivate(*list*)
reduction(*operator : list*)
nowait

- 可将并行域和sections编译制导结合成一条编译制导语句：

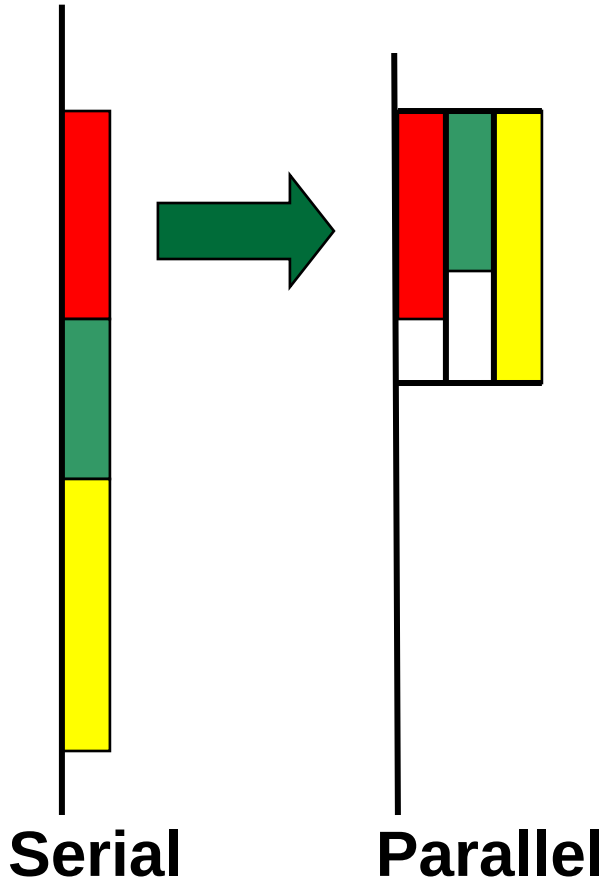
Fortran	!$omp parallel sections [clause...] … !$omp end parallel sections

C/C++	#pragma omp parallel sections [clause...]
	{
	…
	}

工作共享 — omp sections

- 示例：

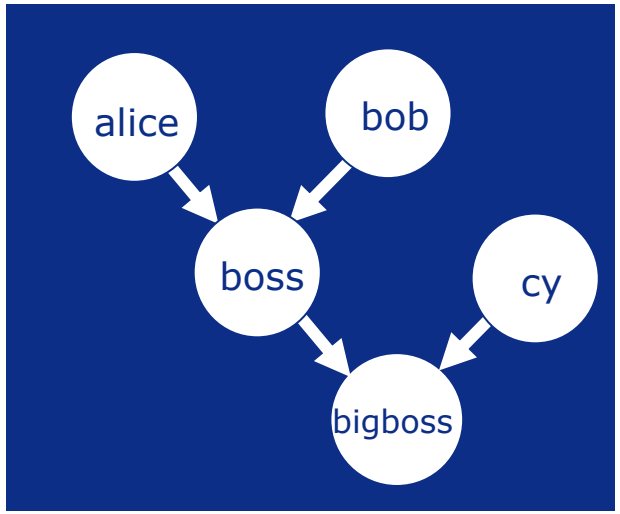
```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



工作共享 — omp sections

- **函数并行示例：**

```
#pragma omp parallel sections
{
#pragma omp section /*可删除*/
   a = alice();
#pragma omp section
   b = bob();
#pragma omp section
   c = cy();
}
s = boss(a, b);
printf ("%6.2f\n", bigboss(s,c));
```

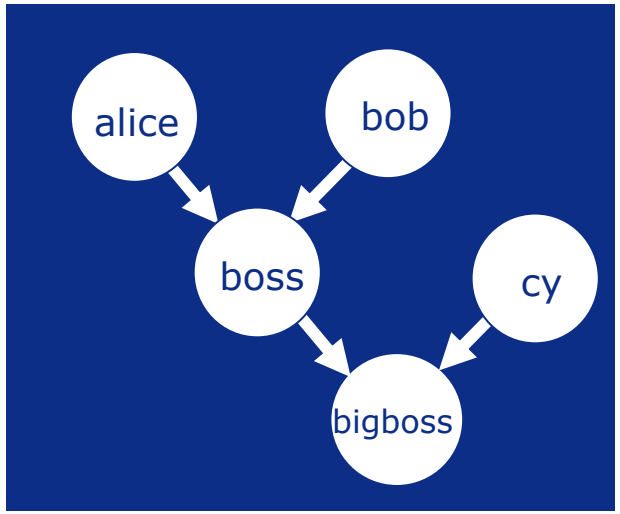



```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
     bigboss(s,c));
```

工作共享 — omp sections

- **函数并行示例：**

```
#pragma omp parallel sections
{
#pragma omp section   /*可删除*/
   a = alice();
#pragma omp section
   b = bob();
}
#pragma omp parallel sections
{
#pragma omp section   /*可删除*/
   c = cy();
#pragma omp section
   s = boss(a, b);
}
printf ("%6.2f\n", bigboss(s,c));
```

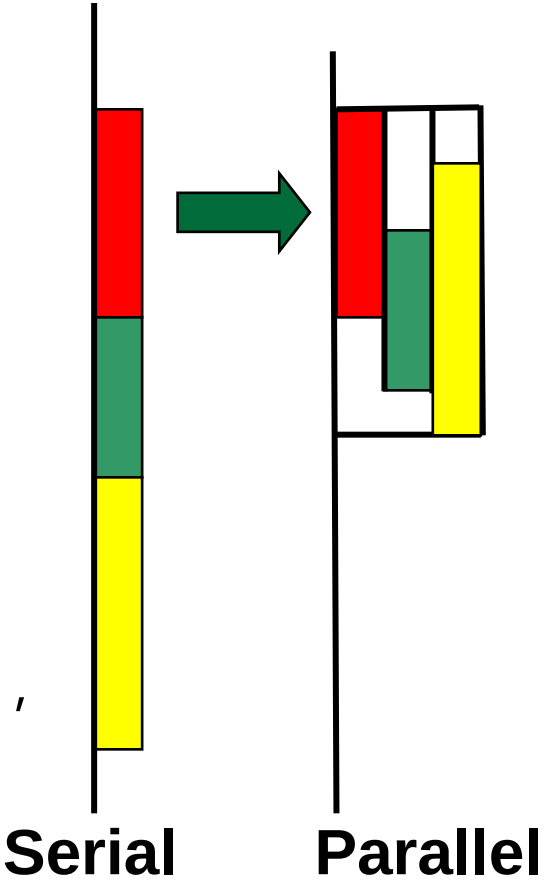



```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
     bigboss(s,c));
```

工作共享 — omp task

Tasks — OpenMP 3.0 新特性

- 适合并行不规则的程序
  - 边界不确定的循环
  - 递归算法
  - 生产者/消费者模式
- #pragma omp task显式定义一个任务
  - 任务是一个独立的工作单元，可能会被遇到的线程马上执行，也可能被延迟给线程池内其他线程来执行
  - 任务的执行，依赖于运行时OpenMP的任务调度
  - task和for、sections的区别在于：task是“动态”定义任务的



工作共享 — omp task

```
#pragma omp parallel num_threads(8)
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

8个线程的线程池被创建

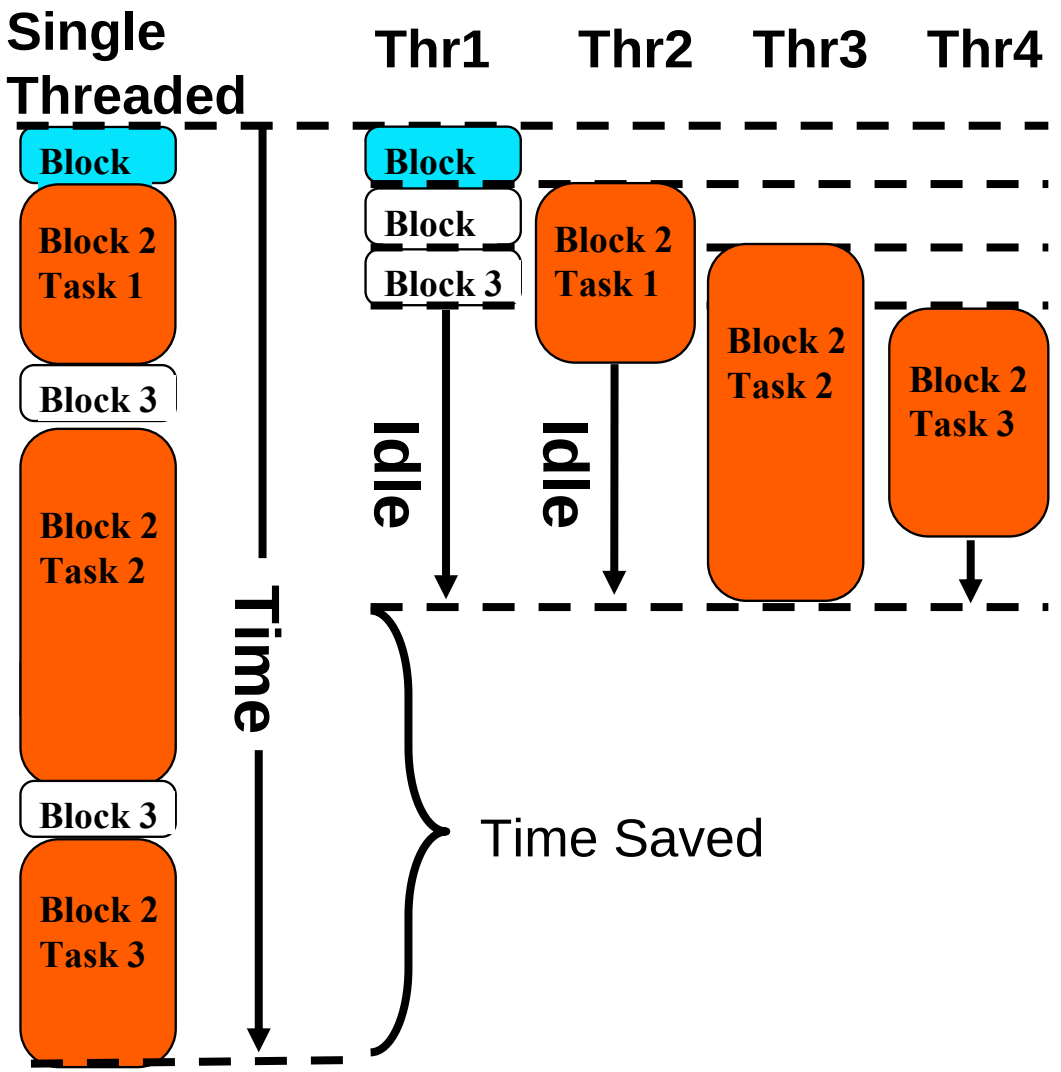
single编译制导语句指定的代码块只由首先执行到该代码的一个线程执行

执行single代码块的线程不断生成 task，新生成的task被运行时系统分配给线程池中的空闲线程

所有线程在singel代码块结束处同步

3 工作共享 — omp task

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
     node * p = head;
     while (p) {  //block 2
     #pragma omp task
        process(p);
     p = p->next;  //block 3
     }
  }
}
```



工作共享 — omp task

- **任务什么时候运行结束？**
  - 在并行域(parallel)、single、for代码块的结尾有隐式同步
  - 使用#pragma omp barrier
  - 使用#pragma omp taskwait

```
#pragma omp parallel{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single{
        #pragma omp task
        bar();
    }
}
```

N个线程则创建N个foo任务

所有foo任务在此处执行完毕

Single线程创建一个bar任务

bar任务在此处执行完毕

工作共享 — omp task

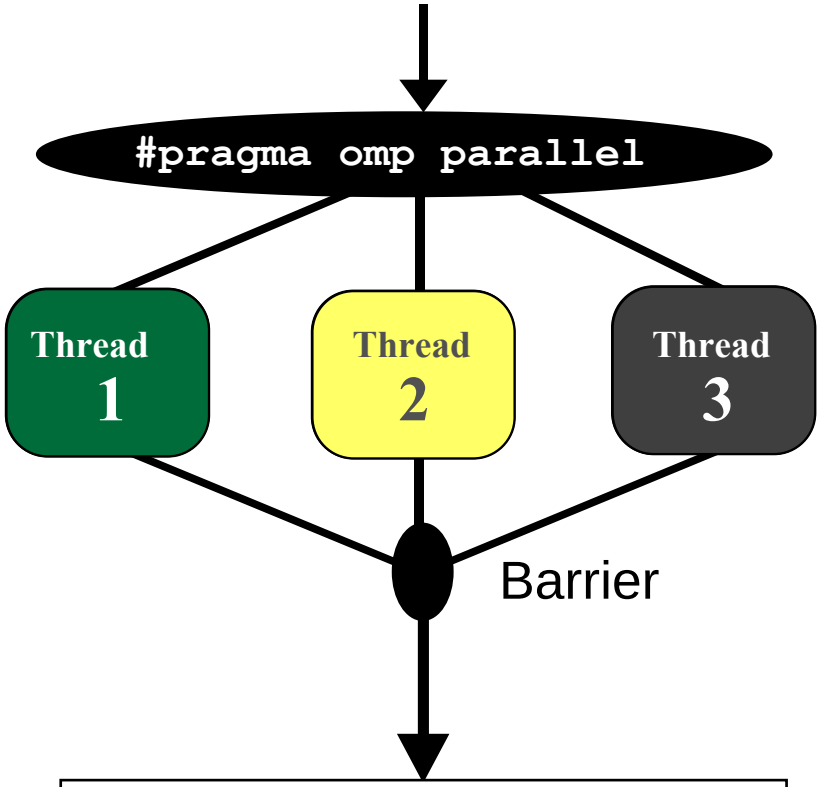
隐式任务

- 隐式任务是由隐式并行区域生成的任务
- 隐式任务为绑定（tied）任务，即隐式任务从始至终总是由最初分配给的线程执行

工作共享 — omp task

隐式任务

- 隐式任务是由隐式并行区域生成的任务
- 隐式任务为绑定（tied）任务，即隐式任务从始至终总是由最初分配给的线程执行




```
#pragma omp parallel
 {   int mydata
     code
 }
```

工作共享 — omp task

绑定（Tied）的任务

- 绑定的任务一旦分配给某个线程，则始终由该线程执行直至任务结束
- 执行绑定任务的线程可以被挂起并执行其它任务，但最终要返回来执行绑定的任务
- 任务默认是绑定的，除非显式声明为非绑定

非绑定（Untied）的任务

- 非绑定任务与任何线程都没有长期关联，任何空闲线程都可能执行未绑定的任务。只能在“任务调度点”更改执行未绑定任务的线程
- 创建非绑定任务： `#pragma omp task untied`

任务切换

- 任务切换是指线程从一个任务的执行切换到另一任务的行为
- 任务切换的目的是让负载尽量均衡，避免堆积大量未分配任务
- 对于绑定的任务，任务切换只能在以下任务调度点进行：
  - 遇到 task 指令
  - 遇到 taskwait 指令
  - 遇到 barrier 指令
  - 隐式 barrier
  - 在绑定任务的结尾
- 未绑定的任务具有与实现相关的调度点

工作共享 — omp task

任务切换示例

- 执行single代码块的线程L会在短时间内生成很多任务
- 当“任务池”填满时，线程L将不得不暂停一段时间
  - 此时会发生任务切换，线程L和其它线程一起处理任务池中的任务
  - 任务池中快没有任务的时候，线程L再切换回生成任务的循环

```
#pragma omp single
{
    for (i=0; i<LARGE_NUMBER; i++)
     #pragma omp task
         process(item[i]);
    }
}
```

4 数据环境

4 数据环境

- **数据的作用范围：哪些变量对多个线程来说是共享的？哪些是私有的？**
- **OpenMP是共享存储系统上的并行编程模型，线程间通过共享变量通信**
- **共享变量：**
  - C/C++：文件范围或namespace范围内的全局变量，静态变量，常量等
  - Fortran: COMMON 块, SAVE 变量, MODULE 变量等
- **私有变量：**
  - 并行区域内部声明的变量 — 显式的私有变量
  - 用private等子句显式声明的变量
  - 循环迭代变量
  - 并行区域中所调用函数的堆栈中的变量（函数内部定义的变量）

数据竞争问题

- 下面的循环无法正确执行：

```
#pragma omp parallel for
for(k=0;k<100;k++) {
   x=array[k];
   array[k]=do_work(x);
}
```

- 正确的方式：
  - 直接声明为私有变量

```
#pragma omp parallel for private(x)
for(k=0;k<100;k++) {
  x=array[k];
  array[k]=do_work(x);
}
```

- 在并行域中声明变量，这样的变量是私有的

```
#pragma omp parallel for
for(k=0;k<100;k++) {
   int x;
   x=array[k];
   array[k]=do_work(x);
}
```

- **两种方式控制并行执行过程中的数据环境：**
  - 独立的OpenMP编译制导指令
    - `#pragma omp threadprivate(list)`
  - OpenMP编译制导指令的数据域属性子句
    - `private`
    - `shared`
    - `default`
    - `firstprivate`
    - `lastprivate`
    - `copyin`
    - `reduction`

threadprivate指令

- 用于将某个全局变量或静态变量声明为各个线程私有的变量，例如：线程ID，在多个并行域中是同一个值
- 该指令必须紧跟着变量声明

```
int A(100), B;
double C;
#pragma omp threadprivate(A, B, C)
```

- 执行机制：当程序第一次进入并行区域时，对标记为threadprivate的变量，每个线程都创建一个副本。每个副本的初始值都不可预知（随机内存地址，未初始化）

```
int a;
#pragma omp threadprivate(a)
#pragma omp parallel
{
    a = OMP_get_thread_num();
}
#pragma omp parallel
{
    printf("a=%d\n",a);
}
```

private子句

- private子句表示它列出的变量对于该并行域的每个线程是局部的
- 语句格式：`private(list)`
- private变量在进入和退出并行区域时是“未定义”的，即：并行区域内的private变量和并行区域外的同名变量没有任何关联

```
int main(int argc, _TCHAR* argv[]){
  int A=100,B,C=0;
  #pragma omp parallel for private(A,B)
  for(int i = 0; i<10;i++){
    B = A + i; // A 未初始化，错误！
    printf("%d\n",i);
  }

  C = B; // B未初始化，错误！
  printf("A:%d\n", A);
  printf("B:%d\n", B);
  return 0;
}
```

private子句

- private子句表示它列出的变量对于该并行域的每个线程是局部的
- 语句格式：`private(list)`
- private变量在进入和退出并行区域时是“未定义”的，即：并行区域内的private变量和并行区域外的同名变量没有任何关联

```
int main(int argc, _TCHAR* argv[]){
  int A=100,B,C=0;
  #pragma omp parallel for private(A,B)
  for(int i = 0; i<10;i++){
     A = 200;
     printf("%d\n",i);
  }

  C = B; // B未初始化，错误！
  printf("A:%d\n", A);
  printf("B:%d\n", B);
  return 0;
}
```

- **firstprivate子句**
  - firstprivate私有变量在进入并行域时用并行域外的变量值进行一次初始化
  - 语句格式：`firstprivate(list)`
- **lastprivate子句**
  - 如果并行域内的私有变量经过计算后，在退出并行域时，需要将其值赋给并行域外的同名变量，可以使用lastprivate完成
  - 语句格式：`lastprivate(list)`

```
int A = 100;
#pragma omp parallel for lastprivate(A)
for(int i = 0; i<10;i++){
    A = 10 + i;
}
printf("%d\n",A);
```

shared子句

- shared子句声明变量被多个线程共享，也就是并行域内外的同名变量都指向相同内存区域，因此要注意数据竞争
- 语句格式：`shared(list)`

```
int sum = 0;
#pragma omp parallel for shared(sum)
for(int i = 0; i < COUNT; i++){
    sum = sum + i;
}
printf("%d\n",sum);
```

default子句

- 指定并行域内变量的默认属性
- 语句格式：`default(shared|none)`
  - default(shared)：表示并行区域内的变量在不确定是private的情况下都是shared属性，等同于不使用default子句
  - default(none)：表示必须显式指定所有变量的数据属性，否则会报错，除非变量有明确的属性定义（比如循环并行域的循环迭代变量只能是私有的）

copyin子句

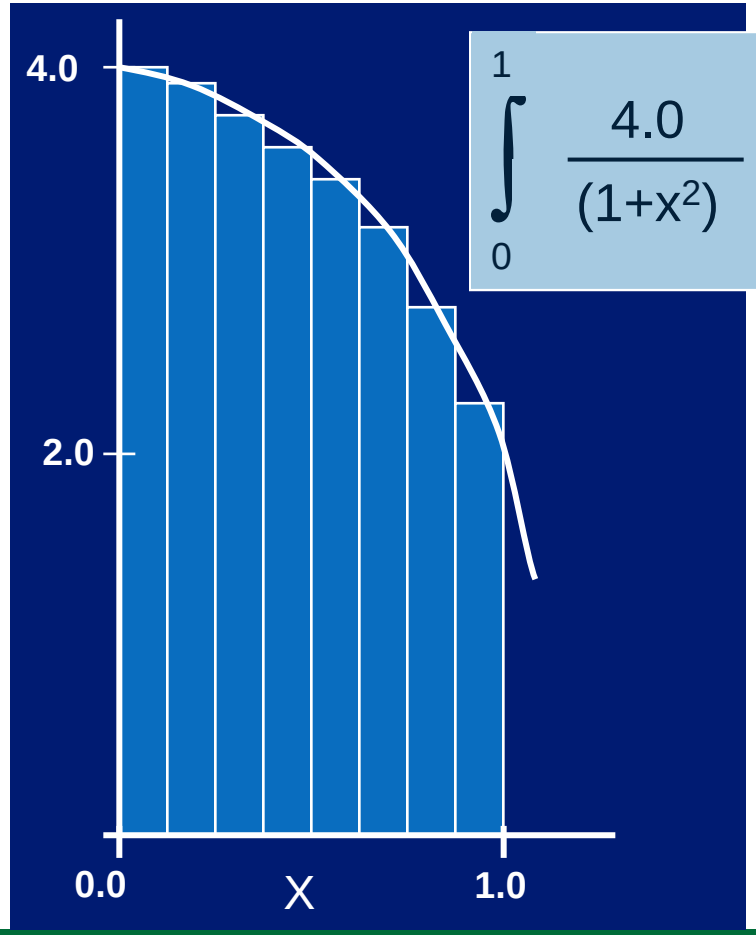
- copyin子句将主线程中threadprivate变量的值拷贝到并行区域各个线程的同名变量中
- copyin中的参数必须被声明成threadprivate的
- 语句格式：`firstprivate(list)`

reduction子句

- reduction子句为变量指定一个操作符，每个线程都会创建reduction变量的私有拷贝，在并行域结束处，将对各个线程的私有拷贝的值按照指定的操作符进行归约计算，并将结果赋值给原来的变量
- 常见reduction操作符及初始值：+(0),-(0),*(1),^(0),&(~0),|(0),&&(1),||(0)
- 语句格式：reduction(*operator*:*list*)

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < COUNT; i++){
    sum = sum + i;
}
printf("%d\n",sum);
```

数值积分法求Pi




```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
#pragma omp parallel for \
    private(i, x) reduction(+:sum)
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

5 同步

OpenMP与线程同步相关的编译制导指令

- single
- master
- critical
- barrier
- atomic
- flush
- ordered

master制导语句

- master制导语句指定结构块仅由主线程执行，其它线程跳过并继续执行，通常用于I/O
- 结尾处没有barrier同步
- 语句格式：`#pragma omp master [clauses]`

```
#pragma omp parallel {
    DoManyThings();
#pragma omp master {
//if not master skip to next stmt
      ExchangeBoundaries();
    }
    DoManyMoreThings();
}
```

```
#pragma omp parallel {
    DoManyThings();
#pragma omp single {
      ExchangeBoundaries();
    }
// threads wait here for single
    DoManyMoreThings();
}
```

critical制导语句

- critical制导语句表明结构块某一时刻仅能被一个线程执行，其它线程被阻塞在临界区外，用来保护对共享变量的修改，避免数据竞争
- 如果name被省略, 假定name为空（null）
- 使用命名临界段时，应用程序可以有多个临界段，通常都应为临界区命名
- 语句格式：

```
#pragma omp critical [(name)]
```

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (RES_lock)
    consum (B, RES);
  }
}
```

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
- 在parallel、for和single结构块后，有一个隐式barrier存在
- for和single结构块的隐式barrier可通过nowait子句去除
- 要么所有线程遇到barrier；要么没有线程遇到barrier，否则会出现死锁
- 语句格式：

  `#pragma omp barrier`

```
#pragma omp parallel shared (A, B, C)
{
        DoSomeWork(A,B);
        printf("Processed A into B\n");
#pragma omp barrier
        DoSomeWork(B,C);
        printf("Processed B into C\n");
}
```

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
        statement
```

- atomic所作用的语句格式：

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

atomic制导语句

- atomic编译指导的好处是允许并行的更新数组内的不同元素；而使用临界值时数组元素的更新是串行的
- 无论何时，当需要在更新共享存储单元的语句中避免数据竞争，应该先使用atomic，然后再使用临界段

```
#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);
   }
```

ordered制导语句

- ordered制导语句指出其所在并行循环的执行，任何时候只能有一个线程执行被ordered所限定部分，且要按照线程ID顺序执行
- 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`

```
vector<int> v;
#pragma omp parallel for ordered
for (int i = 0; i < n; ++i){
        ... ... ...
    #pragma omp ordered
    v.push_back(i);
}
```

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（手动保持共享变量的缓存一致）
- flush列表中只能放单个的变量，不能放数组中的某个值
- 语句格式：`#pragma omp flush [(list)]`
- flush将在下面几种情形下隐含运行，有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

通常OpenMP编程，对多个线程共享变量的读写是要加锁或其它同步机制的，因此一般不需要flush，除非你要尝试自己实现锁同步

6 库函数和环境变量

OpenMP运行时库函数（20多个）

- 设置/获取线程数量、ID
  - omp_[set|get]_num_threads( )
  - omp_get_thread_num( )
  - omp_get_max_threads( )
- 判断是否位于并行域中
  - omp_in_parallel( )
- 得到系统中的处理器内核个数
  - omp_get_num_procs( )
- 显式加锁和解锁
  - omp_[set|unset]_lock( )
- 程序计时
  - omp_get_wtime( )

```
double start = omp_get_wtime();
#pragma omp parallel
{
... //work to be timed
}
double end = omp_get_wtime();
//得到墙钟时间，单位秒
double time = end - start;
```

环境变量

- OMP_NUM_THREADS：设定最大线程数
  - setenv OMP_NUM_THREADS 4
- OMP_SCHEDULE：设定DO/for循环调度方式
  - setenv OMP_SCHEDULE “DYNAMIC，4”
- OMP_DYNAMIC：确定是否动态设定并行域执行的线程数，其值为FALSE或TRUE，默认为TRUE
- OMP_NESTED：确定是否可以并行嵌套，默认为FALSE


北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY
谢谢!
德以明理 学以精工

数值积分法求Pi

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_thraead_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;
}
```

数值积分法求Pi

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_thraead_num();
        sum[id] = 0;
        #pragma omp for
        for (i=id;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;
}
```

数值积分法求Pi

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
  }
}
```