第十三章　共享存储系统并行编程

本章集中讨论基于共享变量的共享存储系统的并行编程。首先简单介绍一下共享存储并行编程的一些基本问题以及纯共享存储环境和虚拟共享存储环境；接着介绍早期的共享存储编程模型ANSIX3H5 和 POSIX；最后着重讨论 OpenMP 标准，主要包括 OpenMP 简介、编译制导语句、运行库例程和环境变量等。

13.1 基于共享变量的共享存储并行编程

在20世纪80年代,高性能的科学和工程计算中基于共享变量的共享存储的编程模式曾是一统天下。进入20世纪90年代后,尽管分布存储的大规模并行处理系统已夺走了峰值计算速度的桂冠,但共享存储的并行处理仍以其可编程性和系统的可用性的优势,在科学和工程计算中与分布存储系统共领风骚。

在共享存储的编程模式中,各个处理器可以对共享存储器中的数据进行存取,数据对每个处理器而言都是可访问到的,不需要在处理器之间进行传送,即数据通信是通过读/写共享存储单元来完成的。

13.1.1 共享存储并行编程的基本问题

共享存储的并行程序设计的基本问题包括:①**任务划分**:任务划分就是把一个程序划分成若干个可以分配给不同的处理器去并行执行的一组任务,划分的方法与并行程序设计风格有关:单程序多数据流(SPMD)编程风格采用按数据流划分任务的方法,即**域分解**法,它将要计算的问题的区域分解成多个子域,每个任务计算一个子域,这样实现的并行也称为**数据并行**;多程序多数据流(MPMD)编程风格则采用按控制流划分任务的方法,即**功能分解**法,它将要计算的问题分解成多个子问题,每个任务计算一个子问题,这样实现的并行也称为**控制并行**。②**任务调度**:调度就是把一个任务集合分配给一组处理器,传统的调度是操作系统管理的事,但因为由操作系统施行调度开销较大,且操作系统难于从程序中获得优化调度信息,所以现代并行机上一个程序内的调度多由用户、编译器和运行库来完成。任务调度有静态调度和动态调度之分:**静态调度**(Static Scheduling)由程序员在编程时、或者编译器在编译时将任务分配给处理器。静态调度有确定的和非确定的两种模式:**确定模式**(Deterministic Mode)任务之间的优先关系和任务所需的执行时间在调度之前是固定和已知的,常可使用 **Gantt 图**(Gantt Chart)来说明调度过程;**非确定模式**(Nondeterminstic Mode)任务的执行时间可表示为一随机变量,这就使得调度问题更为困难。**动态调度**(Dynamic Scheduling)在运行时将任务分配给处理器,分配的策略是在编程或编译时确定的,但具体的分配是在运行时才能确定的。③**任务同步**:同步对并行程序设计是非常重要的,同步常用来确定任务间正确的执行次序,或用于确保各任务对共享变量的正确的读写次序。简单高速的同步机制一般由硬件支持,复杂多功能的同步机制常由软件实现。已如第十二章所述,

同步方式有锁、路障、事件、信号灯和管程等。注意错误的同步会导致死锁。④**任务通信**:在共享存储系统中,任务间的通信借助于读写共享变量来完成,不需要专门的机制,但应注意读写的时机,即要在发送者将正确的信息送出后,接收者才可去读取,为此常要使用路障同步操作,其实同步也可视为一种特殊的通信,只不过所交换的是控制信息,而不是一般通信操作中所交换的数据信息。

13.1.2 共享存储编程环境

根据并行系统中存储器的物理分布,共享存储系统可分为纯共享存储环境和虚拟共享存储环境:前者对应于 SMP 结构,相应的访存模型为 UMA;后者对应于 DSM 结构,相应的访存模型为 NUMA。

纯共享存储环境 纯共享存储环境的主要特点是:①系统中存在着一个集中的、公共的共享存储器;②系统中集中的存储器对程序员而言是全局统一编址的;③系统不提供对非一致存储访问应用程序的任何支持。大多数早期的基于共享存储的并行程序,常使用信号灯、条件临界区和管程等进行任务(进程、线程)间的通信。近代的纯共享存储并行编程标准有 ANSI X3H5、Pthreads 和 OpenMP 等。

虚拟共享存储环境 虚拟共享存储环境的主要特点是:①系统中分布的局部存储器组成了系统的全局共享虚拟存储器;②系统中的虚拟共享存储器对程序员而言是全局可寻址的,即由操作系统负责将这些物理上分布的局部存储器向用户呈现为共享的存储视图,所共享的数据空间是连续的,且可以用通常的读、写操作访问之。近代的虚拟共享存储编程标准有 Linda 和 Jiajia 等。

13.2 早期共享存储并行编程模型

13.2.1 ANSI X3H5 共享存储模型

X3H5 **共享存储模型**(X3H5 Shared-Memory Model)是 1993 年建立的 ANSI 标准,它定义了一个概念标准编程模型,已与 C、Fortran77 和 Fortran90 三种语言相结合。下面我们只简介其主要思想,而细节可参阅[16]。

在图 13.1 中示出了 X3H5 中所使用的指明并行性的一些结构。**并行结构**(Parallel Construct)也称为并行区域(Parallel Region),是些成对的(Parallel...end Parallel)。程序以单线程(也叫基线程(Base Thread)或主线程(Master Thread))串

行模式开始执行，当其遇到 parallel 时就开启到并行模式生成多个子线程；基线程及其子线程并行执行其后的代码直到遇到 end parallel，然后又返回到串行模式，由基线程继续执行之。

```
program main                    !  程序从串行模式开始
        A                       !  A 由基线程执行
        parallel                !  开启到并行模式
        B                       !  B 被每个子线程复制
        psections               !  并始执行并行块
        section
        C       section         !  一个子线程执行 C

        D                       !  另一个线程执行 D
        endpsections            !  等待 C 和 D 都完成
        psingle                 !  E 由一个子线程执行
        E
        endpsingle              !  开启到串行模式
        pdo i = 1,6             !  pdo 结构开始
        F(i)                    !  子线程分享 F 的 6 次迭代
        end pdo no wait         !  无隐路障
        G                       !  复制 G
        end parallel            !  返回至串行模式
        H                       !  H 由起始基线程执行
        ⋮
end
```

图 13.1 ANSI X3H5 标准中并行结构

在并行结构内可有一些并行块(psections...end psections)、并行循环(pdo...endpdo)或单一进程(psingle...endpsingle)等结构，其中并行块用以指明 MPMD 并行性，并行循环用于指明 SPMD 并行性，而单一进程结构指明代码仅由一个线程顺序执行。

图 13.1 中的代码执行顺序可以更清楚地示于图 13.2 中。假定有三个线程 P、Q 和 R：开始基线程 P 执行代码 A；当遇到 parallel 时就生成两个子线程 Q 和 R，所以代码 B 被复制成三份；在并行块(psections)中两段代码 C 和 D 被并行执行；单进程代码 E 仅由 Q 执行；并行循环(pdo)有 6 次迭代，由三个线程分摊执行；接着 P、Q、R 并行执行代码 G；最后由基线程 P 执行代码 H。在此执行过程中，在 parallel、endpsections、end psingle、end pdo 和 end parallel 处设有**隐路障**(Implicit Barrier)，它们也称为**栅栏操作**(Fence Operation)，强使所有存储访问统一于一点以保持一致；如果勿需等待，亦可使用勿需等待隐路障，简称为**无隐路障**(No Implicit Barrier)。

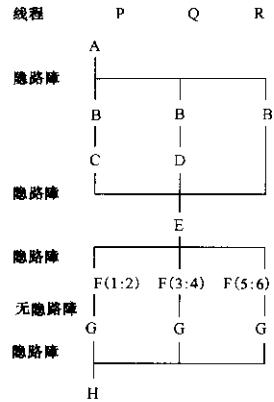

图 13.2 图 13.1 的执行过程

在 X3H5 模型中,线程间的相互作用具有很多很有兴趣的性质:①并行结构中的变量具有共享/私有属性,其中一个线程的私有变量对别的线程是不可见的;②X3H5 非常重视存储器的一致性问题,隐路障、栅栏操作和显路障都可用于此目的;③X3H5 模型引入四种形式的同步变量,即闩锁(Latch)、锁、事件和顺序(Ordinal)。每种类型的变量都伴有初始化操作和释放(Destroy)操作,任何变量在使用前均必须初始化,而采用赋值非初始化状态的办法变可使其释放。锁和事件同步类似于第 12.4.2 节所讨论的内容;而顺序变量用于按照线程的**次第**(Rank)进行线程同步,例如可以使用顺序变量指定线程 T_i 只有在 $T_{i-1},\cdots,T_1$ 在临界区均执行完毕后才能使其进入临界区;闩锁变量使用在临界区中,其语法结构如下:

Critical region[(latch_Variable)]
    Critical_section_code
end Critical region

此结构类似于普通意义的临界区结构,只是可以包含任选的闩锁变量而已。使用闩锁能减少竞争和增加并行度。

使用 X3H5 模型,如何编写计算 π 的并行程序,留给读者(习题 13.3)。

13.2.2 POSIX 线程模型

POSIX(Portable Operating System Interface)Threads，即 Pthreads，代表官方 IEEE POSIX1003.IC-1995 线程标准，系由 IEEE 标准委员会所建立，其功能和界面类似于 Solaris 线程的功能与界面。下面我们只简介其公共性质。

线程管理(Thread Management) 线程库用于管理线程，pthreads 中基本线程管理原语如表 13.1 所示。其中 pthread_create()在进程内生成新线程，新线程执行带有变元 arg 的 myroutine，如果 pthread_create()生成，则返回 0 并将新线程之 ID 置入 thread_id，否则返回指明错误类型的错误代码；pthread_exit()结束调用线程并执行清场处理；pthread_self()返回调用线程的 ID；pthread_join()等待其它线程结束。

表 13.1 Pthreads 中基本线程管理原语一览表

功 能	含 义
int pthread create (pthread_t * **thread_id** pthread_attr_t * **attr**, void * (* myroutine)lvoid *), void * **arg**)	生成线程
void *pthread_exit* (void * **status**)	退出线程
int *pthread_join* (pthread t thread, void * * **status**)	联合线程
pthread_t **pthread_self**(void)	返回调用线程 ID

线程调度 pthread_yield()的功能是使调用者将处理机让位于其它线程；pthread_cancel()的功能是终止指定的线程。

Pthread 同步 Pthread 同步原语列于表 13.2 中。重点讨论互斥 mutex (mutual-Exclusion)变量和条件 Cond(Conditional)变量。前者类似于信号灯结构；后者类似于事件结构。注意，使用同步变量之前需被初始化(生成)，用后应释放清除之。

表 13.2 Pthreads 中线程相互作用原语一览表

功 能	含 义
pthread_mutex_init(…)	生成新的互斥变量
pthread_mutex_destroy(…)	释放互斥变量
pthread_mutex_lock(…)	锁住互斥变量
pthread_mutex_trylock(…)	试探锁住互斥变量
pthread_mutex_unlock(…)	开锁互斥变量
pthread_cond_init(…)	生成新的条件变量
pthread_cond_destroy(…)	释放条件变量
pthread_cond_wait(…)	等待(阻塞)条件变量
pthread_cond_timedwait(…)	等待条件变量直至到达时限
pthread_cond_sinnal(…)	投寄一个事件,开锁一个等待进程
pthread_cond_broadcast(…)	投寄一个事件,开锁所有等待进程

pthread_mutex_lock()锁住互斥(mutex)变量,如果它未被加锁;如果 mutex 已被加锁,调用线程一直被阻塞到 mutex 变成有效。pthread_mutex_trylock()类似于 test-and-set,它一旦锁住 mutex 即立刻返回。pthread_mutex_unlock()释放先前所获得的 mutex,当 mutex 被释,它就能由别的线程获取。

pthread_cond_wait()自动阻塞等待条件满足的现行线程,并开锁 mutex 变量。pthread_cond_timedwait()与 pthread_cond_wait()类似,除了当等待时间达到时限它将解除阻塞外。pthread_cond_signal()解除等待条件满足的已被阻塞的一个线程的阻塞。pthread_cond_broadcast()将所有等待条件满足的已被阻塞的线程解除阻塞。

使用 pthreads 编写计算 π 的程序比较复杂,读者可参见习题 13.4。

13.3 OpenMP 编程简介

OpenMP 标准(OpenMP Standard)是共享存储体系结构上的一个编程模型,已经应用到 UNIX、Windows NT 等多种平台上。本节讲述基本 OpenMP 并行程序设计所需要的知识,包括 OpenMP 简单介绍、编译制导语句、运行库例程和环境变量等。通过本节的学习,可以掌握 OpenMP 最基本和最常见的程序设计方法,编

写出能满足一般需求的OpenMP并行程序。

13.3.1 OpenMP概述

OpenMP起源于ANSI X3H5标准,它具有简单、移植性好和可扩展等优点,是共享存储系统编程的一个工业标准。实际上OpenMP并不是一门新的语言,它是对基本语言(如Fortan 77、Fortan 90、C、C++等)的扩展。OpenMP规范中定义的制导指令、运行库和环境变量,能够使用户在保证程序的可移植性的前提下,按照标准将已有的串行程序逐步并行化。制导指令是对程序设计语言的扩展,进一步提供了对并行区域、工作共享、同步构造的支持,并且支持数据的共享和私有化。这样,用户对串行程序添加制导指令的过程,就类似于进行显式并行程序设计。运行库和环境变量,使得用户可以调整并行程序的执行环境。OpenMP的提出,是希望遵循该并行编程模型的并行程序,可以在不同的厂商提供的共享存储体系结构间比较容易地移植。实际上,已经有许多硬件和软件供应商提供支持OpenMP的编译器,如DEC、Intel、IBM、HP、Sun、SGI及U.S.DOEASCI program等,并且包括了UNIX和NT两种操作系统平台。目前,Fortran 77、Fortran 90、C、C++语言的实现规范已经完成,详细说明可参看http://www.openmp.org。

13.3.2 OpenMP编程风格

OpenMP并行编程模型 首先,OpenMP是基于线程的并行**编程模型**(Parallel Programming Model),一个共享存储的进程由多个线程组成,OpenMP就是基于已有线程的共享编程模**型**;其次,OpenMP是一个外部的编程**模型**,而不是自动编程模型,它能够使程序员完全控制并行化。

OpenMP使用Fork-Join并行执行模型。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。接下来的过程如下:①Fork: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;②Join: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。实际上,所有OpenMP的并行化,都是通过使用嵌入到C/C++或Fortran源代码中的编译制导语句来达到的。并且,一个OpenMP应用编程接口(API)的并行结构可以嵌入到别的并行结构中去。应用编程接口还可以随着不同并行域的需要动态地改变线程数。有些应用也可能不支持上述性质。

OpenMP程序结构 让我们先了解一下OpenMP程序具体的结构。下面分别

是基于 Fortran 语言的 OpenMP 程序的结构和基于 C/C++ 语言的 OpenMP 程序的结构。

(1) 基于 Fortran 语言的 OpenMP 程序的结构

```
PROGRAM HELLO
INTEGER VAR1,VAR2,VAR3
...
! $OMP PARALLEL PRIVATE(VAR1,VAR2)SHARED(VAR3)
...
! $OMP END PARALLEL
...
END
```

(2) 基于 C/C++ 语言的 OpenMP 程序的结构

```
#include<omp.h>
main(){
int var1,var2,var3;
...
#pragma omp parallel private(var1,var2)shared(var3)
{
...
}
...
}
```

13.3.3 OpenMP 编程要素

有了前面的基础知识,接下来就可以学习 OpenMP 编程了。由于 OpenMP 有着一套自己的编译制导语句,所以,要逐个讲解这些语句。需要注意的是,OpenMP 是基于共享存储的编程模型,因此,它本身必然有着符合自己体系结构特点的语句。在学习中,我们要体会这些语句的特点和学会使用它们。另外,由于 OpenMP 可以嵌入到 C/C++ 或 Fortran 等语言中去,所以具体的 OpenMP 程序在不同的环境下会有一些不同。下面主要介绍基于 C/C++ 语言的 OpenMP 的编程。

一个简单的 OpenMP 程序实例 在前文中,已经了解了 OpenMP 程序的简单结构。下面给出一个具体的 OpenMP 程序。虽然程序比较简单,但是可以使读者对 OpenMP 有一个感性的认识,消除对编写 OpenMP 并行程序的疑虑。之后,我

们由简单到复杂对 OpenMP 进一步讲解。

C 语言的程序设计者对“Hello World”这一例子也许还记忆犹新，因为这一例子非常简单，且又具有一定的代表性，因此，我们首先向读者介绍这一例子。下面给出的 OpenMP 并行程序是基于 C/C++ 语言的 OpenMP 程序结构的一个具体实现。

```
//* 用 OpenMP(C)编写 Hello World 代码段 *//
#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;
}
```

经过 OpenMP 的编译，该程序的可能运行结果为：

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

在下文中，我们将以它们为例子，具体剖析 OpenMP 的使用。

编译制导语句 下面分别介绍编译制导语句格式和作用域。

(1) 语句格式：参看基于C/C++语言的OpenMP程序结构，我们可以看到，在并行开始的部分，需要一条语句"# pragma omp parallel private(varl, var2)shared (var3)"。在"Hello world"例子中，并行部分开始时有条语句"# pragma omp parallel private(nthreads,tid)"进行Fork，这条语句就是OpenMP编译制导语句，具体的编译制导语句格式解释如表13.3。

表13.3 编译制导语句格式的解释一览表

# 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。在没有其他约束条件下，子句可以无序，也可以任意地选择。这一部分也可以没有	换行符。表明这条制导语句的终止

(2) 作用域：**编译制导语句的作用域**(Scoping)可分为**静态范围**、**孤幼制导**和**动态范围**。**静态范围**(Static Extent)，又称为**词法范围**(Lexical Extent)，指的是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。一个语句的静态范围并不能用到多个例程或代码文件中。**孤幼制导**(Orphaned Directive)指的是，一个OpenMP的编译制导语句并不依赖于其他的语句。它存在于其他的静态范围语句之外，可以作用于所有的例程和可能的文件代码。一个语句的**动态范围**(Dynamic Extent)包括它的静态(词法)范围和孤幼制导范围。

并行域结构 一个**并行域**(Parallel Region)就是一个能被多个线程执行的程序块，它是最基本的OpenMP并行结构，也就是前面"Hello World"例子中的如下程序段：

```
# pragma omp parallel private(nthreads,tid)
{
...
}
```

该段被并行域封装起来，其中并行域的具体格式如下：

pragma omp parallel [if(scalar _ expression) | private(list) | shared(list) | default(shared | none) | firstprivate(list) | reduction(operator: list) | copyin(list)] newline

注意，当一个线程运行到parallel这个指令时，线程会生成一个线程列，而其自己会成为主线程。主线程也是这个线程列的一员，并且线程号为0。当并行域开

始时，程序代码就会被复制，每个线程都会执行该代码。这就意味着，到了并行部分结束会有一个路障(Barrier)，且只有主线程能通过这个路障。并行域的线程数是由下面的因素决定，且优先级逐步递减，而线程号依次为 0(主线程)到 n－1：①所使用的库函数，即 omp _ set _ num _ threads()；②所设置的环境变量，即 OMP _ NUM _ THREADS；③所使用的默认值。一般地，有多个并行域的程序可以有相同的线程数。系统可以动态地改变一个特定并行部分的线程数。使用动态线程的方法有两个：①使用库函数 omp _ set _ dynamic()；②设置环境变量 OMP _ DYNAMIC。此外，一个并行域可以嵌入到另一并行域中。若有 IF 子句，其值必须为 TRUE(Fortran)或非零(C /C++)，这样才能保证产生一个线程列。否则，该域就会只有主线程串行执行。特别指出，不能跳入与跳出并行域，且只能允许一个 IF 存在。

回头看一下“Hello World”的例子，可以看到，每个线程执行并行部分的所有代码。可使用 OpenMP 的运行库例程获得线程 ID 和线程数。

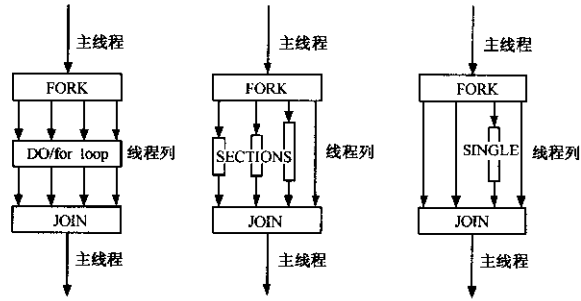

图 13.3 共享任务类型

共享任务结构 共享任务结构把其内封闭的代码段划分给线程队列中的各线程执行。它不产生新的线程，也不存在着进入共享任务结构时有个路障，但在共享任务结构结束时有一个隐含的路障。图 13.3 中示出了三种典型的共享任务结构，其中①DO/for：共享队列中循环代表一种数据并行的类型；②SECTIONS：把任务分割成各部分，每个部分由一个线程执行，可以看成过程并行类型；③SINGLE：由线程序列中的一个线程串行执行。

注意，一个共享任务结构必须在并行域中动态地封装，这是为了能够指示并行执行。共享任务结构必须出现在所有的队列中或一个队列也不出现。队列的所有成员中连续的共享任务结构按同样的次序出现。

(1) DO/for 编译制导语句： DO/for 语句，即 Fortran 中 DO 语句和 C/C++ 中

for 语句，它表明若并行域已经初始化了，之后的循环必须由线程队列并行地执行；否则就会顺序执行。语句格式如下：

pragma omp for [schedule(type[, chunk]) | ordered | private(list) | firstprivate(list) | lastprivate(list) | shared(list) | reduction(operator:list) |]newline

其中，schedule 子句用来描述如何划分线程队中的循环。type 为 static 时，循环被划分成块，静态地分配给线程执行。若对 chunk 没有特别声明，循环会在线程列中尽量平分。type 为 dynamic 时，循环分成的块被动态安排到线程列中处理。若一个线程完成了一块，它就会被动态地安排执行别的块。默认的块的长度为 1。下面介绍向量加法的例子，体会一下 DO/for 语句的具体用法。

```
# include <omp.h>
# define CHUNKSIZE 100
# define N 1000
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 shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  } /* end of parallel section */
}
```

(2) SECTIONS 编译制导语句：SECTIONS 编译制导语句是非循环的共享任务结构，它表明内部的代码是被线程队列分割。不关联的 SECTIONS 编译制导语句可以相互嵌套。SECTIONS 语句的格式为：

#pragma omp sections [private (list) | firstprivate (list) | lastprivate (list) | reduction(operator: list) | nowait] newline

```
{
  # pragma omp section newline
    structured _ block
  # pragma omp section newline
```

```
    structured_block
}
```

注意,在没有 nowait 语句时,SECTIONS 语句之后有一路障。

为了比较,下面我们用 SECTIONS 语句来实现向量加法运算。在开始的 N/2 个 DO 循环由第一个线程执行,后面的由第二个线程执行。当任一个线程执行完时,它都会进行下面的操作(NOWAIT)。下面是具体的程序:

```
#include <omp.h>
#define N    1000
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 */
}
```

(3) SINGLE 编译制导语句: SINGLE 编译制导语句表明内部的代码只是由一个线程执行。这个语句是非常有用的。具体的格式如下:

#pragma omp single [proivate(list)|firstprivate(list)|nowait] newline

若没有 *nowait* 语句,队列中没有执行 SINGLE 语句的线程,则会一直等到代码路障同步才会继续执行下面的代码。

组合的并行共享任务结构 下面分别介绍两种编译制导语句。

(1) PARALLEL DO/parallel for 编译制导语句: parallel for 编译制导语句(即 Fortran 中 PARALLEL DO 语句)表明一个并行域包含一个单独的 DO/for 语句,其

格式如下：

#pragma omp parallel for [if(scalar _ logical _ expression) | default(shared | none) | schedule(type[, chunk]) shared(list) | private(list) | firstprivate(list) | lastprivate(list) | reduction(operator:list) | copyin(list)]newline

该语句的格式必须符合 PARALLEL 和 DO/for 语句的格式。下面给出一个 parallel for 的例子，为了比较，仍以向量加法作为例子，看看 paraller for 的使用方法，由此可以看出每个循环被平均分到各个线程上：

```
#include<omp.h>
#define N       1000
#define CHUNKSIZE     100
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];
}
```

(2) 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，其具体格式如下：

#pragma omp parallel sections [default(shared|none)|shared(list)|private(list)|firstprivate(list)|lastprivate(list)|reduction(operator:list)|copyin(list)|ordered]newline

同样，相应的格式必须符合 PARALLEL 语句和 SECTIONS 语句格式。其使用方法类似 parallel for 语句。

同步结构 OpenMP 提供了多种同步结构来控制与其他线程相关的线程的执行。

(1) MASTER 编译制导语句：MASTER 编译制导语句表明一个只能被主线程执行的域。队列中所有其他的线程必须跳过这部分的代码，语句中并没有路障。其格式如下：

#pragma omp master newline

(2) CRITICAL 编译制导语句：CRITICAL 编译制导语句表明域中的代码一次只能一个线程执行。其具体的格式如下：

#pragma omp critical[name] newline

注意,当一个线程正在执行一个 CRITICAL 域,而另一个到达 CRITICAL 域准备执行时,则第一个线程执行,后一个等待直到第一个退出域为止。

(3) BARRIER 编译制导语句:BARRIER 语句同步队列中的所有线程。当有一个 BARRIER 语句时,线程必须等到所有的其他线程也到达这个路障时才可继续。然后,所有的线程都并行执行路障之后的代码。其具体的格式如下:

```
# pragma omp barrier newline
```

所有的线程要么都遇到路障,要么都不遇到路障。

(4) ATOMIC 语句:ATOMIC 语句表明一个特别的存储单元只能原子地更新,而不允许让多个线程同时去写。最重要的是,这个语句提供了一个 mini - CRITICAL 部分。其格式如下:

```
# pragma omp atomic newline
```

(5) FLUSH 语句:FLUSH 语句是用来确保执行中存储器中数据一致性,线程可见的变量在此写回存储器。其格式如下:

```
# pragma omp flush(list) newline
```

(6) ORDERED 语句:ORDERED 语句指出被包含的循环如同其在一个串行处理器上的执行一样。其格式如下:

```
# pragma omp ordered newline
structured _ block
```

注意,ORDERED 语句只能出现在 DO 或 PARALLEL DO(Fortran)和 for 或 parallel for(C/C++)语句的动态范围中。

在任何时候,一个命令只能由一个线程执行,有分支从一个 ORDERED 块中跳进或跳出都是不合法的。循环中不能执行 ORDERED 指令多于一次,也不能执行多个 ORDERED 指令。包含 ORDERED 指令的循环,必须含有一个 ORDERED 子句。

THREADPRIVATE 编译制导语句 它表明整个文件**局部变量**(Scope Variable)在多个并行线程执行时,变成每个线程私有。其格式如下:

```
# pragma omp tbreadprivate(list)
```

这条语句必须出现在变量序列定义之后。每个线程都复制这个变量块,所以一个线程的写数据对于别的线程是不可见的。

数据域属性子句 学习 OpenMP 编程重要的一点是理解和会使用**数据域**(Data Scope)。由于 OpenMP 是建立在共享存储编程模型之上的,所以大部分变量默认为共享。全程变量有**文件域变量**(File Scope Variable)和静态变量;局部变量有循环内的变量和并行域调用子程序的堆栈变量。

OpenMP 的数据域属性子句用来定义变量的范围,它包括 PRIVATE、FIRSTPRIVATE、LASTPRIVATE、SHARED、DEFAULT、REDUCTION 和

COPYIN 等。数据域变量与编译制导语句 PARALLEL、DO/for 和 SECTIONS 相结合可用来控制变量的范围。它们在并行结构执行过程中控制数据环境，比如，哪些串行部分中的数据变量被传到程序的并行部分以及如何传送；哪些变量对所有的并行部分的线程是可见的；哪些变量对所有的线程是局部的等。

(1) PRIVATE 子句：PRIVATE 子句表示它列出的变量对子每个线程是局部的。其具体格式如下：

```
private(list)
```

(2) SHARED 子句：SHARED 子句表示它列出的变量被线程列中所有的线程共享。其具体格式如下：

```
shared(list)
```

一个共享的变量只存在存储器的一个地方，所有的线程都能读或写这个地址。程序员能够使多个线程存取 SHARED 变量(例如通过 CRITICAL 语句)。

(3) DEFAULT 子句：DEFAULT 子句让用户规定在并行域的词法范围中所有变量的一个默认的范围(如可以是 PRIVATE 、SHARED 或是 NONE)。其格式如下：

```
default(shared|none)
```

(4) FIRSTPRIVATE 子句：FIRSTPRIVATE 子句包含 PRIVATE 子句的操作，并初始化其列出的变量。其格式如下：

```
firstprivate(list)
```

(5) LASTPRIVATE 子句：LASTPRIVATE 子句包含 PRIVATE 的操作，并将变量从最后一个循环或段中复制给原始的变量。其格式如下：

```
lastprivate(list)
```

(6) COPYIN 子句：COPYIN 子句用来赋值所有线程中的同样变量与 THREADPRIVATE 中的变量一样的值。其格式如下：

```
copyin(list)
```

(7) REDUCTION 子句：REDUCTION 子句用来归约其列表中出现的变量。每个线程复制一个私有的变量列表。在归约的最后，归约的变量可以应用子所有的共享变量的私有变量列表中，最终的结果写到全局共享变量。其格式如下：

```
reduction(operator:list)
```

语句的绑定和嵌套规则 下面主要介绍 OpenMP 语句的绑定和嵌套规则。

(1) 语句绑定：对于语句的绑定，读者需要注意以下几点：①语句 DO/for、SECTIONS、SINGLE、MASTER 和 BARRIER 绑定到动态封装的 PARALLEL 中，如果没有并行域执行，这些语句是无效的。②语句 ORDERED 指令绑定到动态封装的 DO/for 中。③语句 ATOMIC 使得 ATOMIC 语句在所有的线程中互斥存取，而并不只是当前的线程列。④语句 CRITICAL 在所有线程有关 CRITICAL 指令

中互斥存取，而不是只对当前的线程列。⑤在 PARALLEL 封装外，一个语句并不绑定到其他的语句中。

(2) 语句嵌套：对于语句的嵌套，读者应注意以下几点：①PARALLEL 语句动态地嵌套到其他的语句中，从而逻辑地建立了一个新队列，但这个队列若没有嵌套的并行域执行，则只包含当前的线程。②DO/for、SECTIONS 和 SINGLE 语句绑定到同一个 PARALLEL 中，则它们是不允许互相嵌套的。③DO/for、SECTIONS 和 SINGLE 语句不允许在动态范围的 CRITICAL、ORDERED 和 MASTER 域中。④CRITICAL 语句不允许互相嵌套。⑤BARRIER 语句不允许在动态范围的 DO/for、ORDERED、SECTIONS、SINGLE、MASTER 和 CRITICAL 域中。⑥MASTER 语句不允许在动态范围的 DO/for、SECTIONS 和 SINGLE 语句中。⑦ORDERED 语句不允许在动态范围的 CRITICAL 域中。⑧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。

13.3.4 OpenMP 计算实例

下面分别给出基于 C/C++ 语言的使用并行域并行化的程序和使用共享任务结构并行化的程序以及基于 Fortran 语言描述的并行化程序。

```
//* 用并行域并行化的 OpenMP 计算 π 的代码段 *//
#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_ser_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;
}
```

```
//* 用共享任务结构并行化的 OpenMP 计算 π 的代码段 *//
#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=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;
}
```

另外,我们给出一个基于 Fortran 语言描述的 OpenMP 计算 π 的示范程序。

```
//* 用 Fortran90 语言描述的 OpenMP 计算 π 的代码段 *//
      program compute_pi
      integer n,i
      real w,x,sum,pi,f,a
! function to integrate
          f(a) = 4.d0/(1.d0+a*a)
          print *,'Enter number of intervals:'
          read *,n
! calculate the interval size
          w=1.d0/n
          sum=0.0d0
! $OMP PARALLEL DO PRIVATE(x), SHARED(w), REDUCTION(+:sum)
          do i=1, n
             x=w*(i+0.5d0)
             sum=sum+f(x)
          enddo
```

```
! $OMP END PARALLEL DO
        pi = w * sum
        print *, 'compute pi = ', pi
        stop
    end
```

13.3.5 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。比如获取线程或处理器数、设置使用的线程数、加锁例程、设置执行的环境变量、嵌入的并行化、动态调整线程等。对于 C/C++，在程序开头需要引用文件"omp.h"。对于加锁的例程，加锁的变量只能被加锁的例程访问，加锁的变量必须定义为"omp_lock_t"或"omp_nest_lock_t"。下面具体看几个例程：

(1) OMP_SET_NUM_THREADS：这个例程设定在下一个域使用的线程数。其格式如下：

```
void omp_set_num_threads(int num_threads)
```

(2) 其他的例程参见章末附录。

环境变量 OpenMP 提供了 4 个环境变量来控制并行代码的执行，即 OMP_SCHEDULE，OMP_NUM_THREADS，OMP_DYNAMIC 和 OMP_NESTED。所有环境变量的名字都是大写字母。下面进行具体的解释：

(1) **OMP_SCHEDULE**：只能用到 DO、PARALLEL DO(Fortran)和 for、parallel for(C/C++)中。它的值就是处理器中循环的次数。例如：

```
setenv OMP_SCHEDULE "guided, 4"
```

(2) **OMP_NUM_THREADS**：定义执行中最大的线程数，例如：

```
setenv OMP_NUM_THREADS 8
```

(3) **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，例如：

```
setenv OMP_DYNAMIC TRUE
```

(4) **OMP_NESTED**：确定是否可以并行嵌套，例如：

```
setenv OMP_NESTED TRUE
```

13.4 小结和导读

小结 本章首先谈到了共享存储的并行编程所涉及的一些基本问题，诸如任

务划分、任务调度、任务同步和任务通信等，并介绍了纯共享存储和虚拟共享存储两种编程环境的概念；然后对早期的共享存储的编程标准 ANSI X3H5 和 POSIX 做了简单介绍；最后着重介绍目前普遍采用的共享存储体系结构的 OpenMP 编程标准，包括 OpenMP 编程风格、编程要素（编译制导、共享任务结构、同步结构、数据域以及语言的绑定和嵌套规则等）以及运行库例程与环境变量等。希望通过对 OpenMP 的学习，读者可以掌握 OpenMP 最基本和最常用的程序设计方法，能编写出满足一般要求的 OpenMP 并行程序。

对于共享存储编程方法，除了本章简介的 X3H5、Pthreads 和 OpenMP 三个标准外，另外还有像 SGI POWER C 和新型并行 C 语言 C//等，前者是串行 C 语言的推广，具有编译制导和库函数；后者是基于标准 C 语言，具有少量的为并行和进程相互作用的一组扩展结构。

导读 X3H5 标准在[16]中给予了描述，POSIX 在[141]中进行了讨论，有关 OpenMP 标准描述在[133,134,194]中，SGI Power C 编程模型读者可参阅[159]，新型并行语言 C//读者可参阅[186]，函数式程序设计语言讨论在[9]中，面向对象方法讨论在[6]中，应用 C++ 进行并行编程读者可参阅[182]，分布计算系统程序设计语言读者可参阅[23]。

习 题

13.1 试分析下列循环嵌套中各语句间的相关关系：

① DO I=1,N
　　DO J=2,N
　S_1：　A(I,J)=A(I,J-1)+B(I,J)
　S_2：　C(I,J)=A(I,J)+D(I+1,J)
　S_3：　D(I,J)=0.1
　　Enddo
　Enddo

② DO I=1,N
　S_1：　A(I)=B(I)
　S_2：　C(I)=A(I)+B(I)
　S_3：　D(I)=C(I+1)
　Enddo

③ DO I=1,N
　　DO J=2,N

S_1: A(I,J) = B(I,J) + C(I,J)

S_2: C(I,J) = D(I,J)/2

S_3: E(I,J) = A(I,J-1) * *2 + E(I,J-1)

Enddo

Enddo

13.2 令 $N=10^5$ 和 $N=10^8$,试编写计算 π 的 SPMD 并行程序,并在您现有的共享存储的平台上调试之;同时应执行在 1,2,3,4,5,6,7 和 8 个处理器上。

13.3 试用 X3H5 模型,写出计算 π 的并行程序。

13.4 下面是使用 Pthreads 方法计算 π 的一种并行代码段:

```
//* 计算 π 的 Pthreads 编程代码段 *//
#include <stdio.h>
#include <pthread.h>      /* 线程控制所需的头文件 */
#include <synch.h>        /* 同步操作所需的头文件 */
extern unsigned * micro-timer;      /* 系统计时变量 */
unsigned int   fin,start;
semaphore_t semaphore;
barrier_spin_t barrier;      /* 同步信号量说明 */
double delta,pi;
typedef struct{      /* 定义参数结构 */
    int low;
    int high;
}Arg;
/* 定义线程执行的函数 */
void child(arg)
Arg * arg
{
    int low = arg→low;
    int high = arg→high;
    int i;
    double x ,part = 0.0;
    for(i = low;i <= high;i++){
        x = (i + 0.5) * delta;
        part + = 1.0/(1.0 + x * x);
    }
    psema(&semaphore);/* 利用信号量进行互斥累加 */
        pi + = 4.0 * delta * part;
    vsema(&semaphore);
```

```
    -barrier_spin(&barrier);/* 线程在完成计算后需 barrier 同步 */
    pthread_exit();/* 线程终止 */
}
main(argc,argv)
int argc;
char * argv[];
{
    int no_of_threads,segments,i;
    pthread_t thread;
    Arg * arg;
    if (argc != 3){
        printf("usage:pi<no-of-threads><no-of-strips>\n");
        exit(1);
    }
    no-of-threads = atoi(argv[1]);
    segments = atoi(argv[2];
    delta = 1.0/segments;
    pi = 0.0;
    sema_init(&semaphore,1);      /* 初始化同步变量 */
    -barrier_spin_init(&barrier,no_of_threads + 1);
    start = * micro_timer;      /* 线程开始计时 */
    for(i = 0;i<no_of_threads;i++){      /* 启动线程 */
        arg = (Arg * )malloc(size of (Arg));
        arg→low = i * segments/no_of_threads;
        arg→high = (i + 1) * segments/no_of_threads - 1;
        pthread_create(&thread,pthread_attr_default,child,arg);
    }
    -barrier_spin(&barrier);      /* 主进程等待所有子线程结束 */
    fin = * micro_timer;      /* 线程结束记时 */
    printf("%u\n",fin - start);
    printf("\npit\t%15.14f\n",pi);
}
```

①试解释上述代码段的工作流程。

②通过三种模型 X3H5、Pthreads 和 OpenMP 上计算 π 的代码段，比较它们的编程风格的异同和优缺点。

13.5 下面是使用经理员工(Manager-Worker)模型(即主－从模型)求解 N-皇后问题的并行代码段：

```
//* 求解 N-皇后经理-员工编程代码段 *//
/* Manager 程序段 */
if (!iam){/* 如果我是节点 0 */
    printf("\n STARTING…\n");
    while(get_board(board)-!DONE){
        crecv(READY,NULL,0);
        nodenbr=infonode();
        msgcount++;/* 计数多少节点准备好 */
        csend(TASK,board,sizeof(two ,two D),nodenbr,0);
        msgcount--;
    }
/* 等待所有员工空闲 */
    while(msgcount!=nodes-1){
        crecv(READY,NULL,0);
        msgcount++;
    }
    /* 发送 FINISHED 消息给所有节点,并退出 */
    board[0][0]=FINISHED;
    csend(TASK,board,sizeof(two D),-1,0);
    goodbye();
}
else{/* 员工程序段 */
    for(;;){
        csend(READY,0,0,0,0);
        crecv(TASK,board,sizeof(board));
        if(board[0][0]==FINISHED
            goodbye();
        if(chk-board(board))
            move_to_right(board,0,MCOLS);
    }
}
```

试解释上述代码段的计算过程。

附录 OpenMP 运行库例程

OMP_SET_NUM_THREADS:

这个子例程设定在下一个域使用的线程数。其格式如下：

```
void omp_set_num_threads(int num_threads)
```

OMP_GET_NUM_THREADS

这个例程返回目前调用的执行域队列的线程数。格式如下：

```
int omp_get_num_threads(void)
```

OMP_GET_MAX_THREADS

这个例程返回调用 OMP_GET_NUM_THREADS 例程的最大值。格式如下：

```
int omp_get_max_threads(void)
```

OMP_GET_THREAD_NUM

这个例程返回队列中的线程的序号。这个序号可在 0 到 OMP_GET_NUM_THREADS −1 之间。队列中主线程的序号是 0。格式如下：

```
int omp_get_thread_num(void)
```

OMP_GET_NUM_PROCS

这个例程返回程序中使用的处理器数。格式如下：

```
int omp_get_num_procs(void)
```

OMP_IN_PARALLEL

这个例程决定执行的代码部分是并行的还是不是。格式如下：

```
int omp_in_parallel(void)
```

OMP_SET_DYNAMIC

这个子例程可以设定能或不能动态调整并行域执行的线程数。格式如下：

```
void omp_set_dynamic(int dynamic_threads)
```

OMP_GET_DYNAMIC

这个函数/例程用来决定可以或不可以调整动态线程。其格式如下：

```
int omp_get_dynamic(void)
```

OMP_SET_NESTED

这个子例程用来设定能够或不能够并行嵌套。其格式如下：

```
void omp_set_nested(int nested)
```

OMP_GET_NESTED

这个函数/例程用来决定并行嵌套是可以还是不可以，格式如下：

```
void omp_get_nested
```

OMP_INIT_LOCK

这个子例程用来初始化一个锁。其格式如下：

```
void omp_init_lock(omp_lock_t * lock)
void omp_nest_init_lock(omp_nest_lock_t * lock)
```

注意初始状态没有上锁。

OMP_DESTROY_LOCK

这个子例程用来删除一个锁。格式如下：

```
void omp_destroy_lock(omp_lock_t * lock)
```

```
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
```

注意,若锁变量没有初始化,这个调用是非法的。

OMP_SET_LOCK

这个子例程使得执行线程等待,直到指定的锁可用。格式如下:

```
void omp_set_lock(omp_lock_t *lock)
void omp_set_nest_lock(omp_nest_lock_t *lock)
```

OMP_UNSET_LOCK

这个子例程表示从执行子例程中解锁。格式如下:

```
void omp_unset_lock(omp_lock_t *lock)
void omp_unset_nest_lock(omp_nest_lock_t *lock)
```

同样,若锁变量没有初始化,调用这个例程是非法的。

OMP_TEST_LOCK

这个子例程用来测试一个锁,若没有这个锁,则没有阻塞。格式如下:

```
void omp_test_lock(omp_lock_t *lock)
void omp_test_nest_lock(omp_nest_lock_t *lock)
```