使用OpenMP 进行 Fortran95并行计算

原著：《Parallel Programming in Fortran 95 using OpenMP》

Fortran Coder http://www.fcode.cn

目　　录

设置计算机

1、设置

(1) 在 vs2008 中新建 fortran 控制台程序，选择项目(Project) -> 属性(property) -> Fortran -> 语言(Language)，在 Process OpenMP Directives 选项中选择 Generate Parallel Code (/Qopenmp)，点击确定以打开 OpenMP 支持。

(2) 设置环境变量：我的电脑 -> 属性 -> 高级 -> 环境变量，在系统变量栏中新建一个 OMP_NUM_THREADS 变量，值设为 2，即为程序执行的线程数（线程数一般设为处理器核数）。

运行如下代码：

```
program test
    !$OMP PARALLEL
    print*, 'ok!'
    !$OMP END PARALLEL
End program
```

运行结果：

Ok！

Ok！

第一章 OpenMP Fortran 应用程序接口界面

1.1 简介

为获得更强的计算能力，计算系统的开发人员开始考虑联合使用多个计算机，这就是并行机的起源，为程序员及研究者开启了新的研究领域。

如今，并行计算机在科研领域较为普遍，广泛应用于复杂计算，如模拟原子弹爆炸，蛋白质聚合以及水流扰动。

并行机面临的一大挑战是开发一套可用于有效硬件的代码，以便在更短的时间内解决大型问题。但由于存在各种不同的硬件构架，并行编程可不是一件容易的事情。主要有两个系列的并行机可以被识别：

内存共享结构(Shared-memory architecture)：基于一组可访问共同内存的处理器，这种构架的计算机使用 SMP(Symmetric Multi Processing) 机器指令，即对称多进程。

分布式内存结构(Distributed-memory architecture)：每个处理器拥有私有的内存，处理器之间通过消息(Messages)进行信息交换。集群(clusters)通常被用于这类计算设备。

每个系列都有自己的优缺点，并行编程标准试着针对某一构架充分利用其优点。

近年来出现了一种新的行业标准，旨在为共享内存机器上开发并行程序建立良好的基础，这就是 OpenMP。

1.1.1 历史回顾

共享内存机存在很久了。过去的每个硬件厂商都有自己的指令和库标准，允许程序使用指定的并行机。早期的标准 ANSI X3H5 未被正式采用，一方面是因为没有硬件商的大力支持；另一方面，分布式存储机拥有自己的标准消息传递库 PVM 和 MPI，能很好地代替内存共享机。

但在 1996-1997 年，出现了内存共享编程接口界面，引起了广泛的兴趣，主要因为：

(1) 硬件商恢复了对共享内存构架的兴趣

(2) 部分厂商认为，使用消息传递接口的并行程序冗长、耗时，需要更简单的编程接口

很多硬件商和编译器公司都把 OpenMP 看作行业标准：它定义的一系列编译器指令、运行库和环境变量可用于 Fortran 和 C/C++的共享内存式并行编程。

OpenMP 整合为简单的语法，不支持早期的共享内存指令集处理 coarse-grain parallelism（将目标域分解为子域，交由多个处理器进行计算）。过去，由于对 coarse-grain 的支持有限，开发者认为共享内存并行编程对 fine-grain parallelism（分解循环到多个处理器进行迭代）的支持也是有限的。

1.1.2 参与者

OpenMP 规范由 OpenMP Archi-tecture Review Board 拥有、编写和维护，很多公司积极参与共享内存编程接口界面标准的开发。2000 年，OpenMP ARB 的永久性合作者有：

- US Department of Energy, through its ASCI program (美国能源部)
- Compaq Computer Corp （康柏电脑公司）
- Fujitsu （日本富士通）
- Hewlett-Packard Company （惠普）
- Intel Corp. （英特尔）
- International Business Machines （国际商业机器公司）
- Kuck & Associates, Inc. （擅于并行软件开发，2000 年被 intel 收购）
- Silicon Graphics Incorporate （硅谷图形（美国计算机公司））
- Sun Microsystems （sun 微系统公司）

此外，还有很多公司在他们的程序或编译器里使用 OpenMP 的同时，通过提交错误报告、评论和建议等方式，对 OpenMP ARB 做出了很多的贡献。

1.1.3 关于本文档

本文档只在为有兴趣学习 OpenMP 的 Fortran95 程序员提供基本的帮助。OpenMP ARB 发布的 OpenMP 规范中未详细说明的尤为重要的部分及不同 OpenMP 指令和子句的性能比较，在本文档中都带有图示。为使文档简练，OpenMP 的部分内容未予列出，建议读者同时阅读本文档和 OpenMP 规范。

本文档只考虑 Fortran95 编程语言，尽管大部分概念和观点也适用于 Fortran77 。作者认为 Fortran95 优于 Fortran77，而且良好的编程方法非常重要，本文只提供部分与上述编程理念一致的 OpenMP 特性。因为本文提到的概念只是作者自己的观点，因此读者需要另外阅读完整 OpenMP 规范。

现有的关于 OpenMP 的文档不多，本文档可在网上免费发布，但作者保有版权。欢迎对本文内容进行评论，鼓励读者提交建设性的意见和建议。

撰写本文档期间(2001 冬-2002 春),编译器使用两个不同的 OpenMP 规范：v1.1 和 v2.0 。后者是前者的升级版，有必要区分每个版本的有效内容。本文中蓝色字体部分的内容仅适用于 v2.0

1.2 基础

OpenMP 代表一个关于编译器指令、运行库和环境变量的集合，用于在共享内存机器上进行并行程序开发，集合中的每个元素将用一章进行说明。在回顾有效编译器指令之前，有必要学习 OpenMP 的基础知识。

尽管称为“基础”，这部分内容却是 OpenMP 的基本部分，允许在程序中包含 OpenMP 命令和破坏并行运算的代码块。

1.2.1 OpenMP 指令和条件编译的标记

OpenMP 标准的目的之一，就是使相同的代码段既可以运用于支持 OpenMP 的编译器上，又可以运用于普通编译器上。要获得上述效果，只有使普通编译器忽略 OpenMP 指令和命令，而支持 OpenMP 的编译器却可以发现它们。因此引入下面两条指令标记：

```
!$OMP

!$
```

因为首字符为感叹号“!”，普通编译器将其解释为注释行，并忽略其内容。但是支持 OpenMP 的编译器可以识别所有序列，并进行如下操作：

!$OMP：OpenMP编译器获知本行的余下信息为OpenMP指令。通过在以后的代码行之前设置同样的标记和使用标准Fortran95方法限制代码行，可以将OpenMP指令扩展到多行代码：

```
!$OMP PARALLEL DEFAULT(NONE) SHARED(A, B) PRIVATE(C, D) &
!$OMP REDUCTION(+:A)
```

指令标记符!$OMP和后面的OpenMP指令之间必须有至少一个空格，否则指令标记不能被正确识别，此行将当作注释行。

!$：与条件编译相关的行，表示其后内容仅对OpenMP编译器有效。这种情况下，标记的两个字符被两个空格替代，编译器将会考虑本行内容。正如前面的例子，可以将源代码行扩展到多行：

```
!$ interval = L * OMP_get_thread_num() / &
!$ (OMP_get_num_threads() - 1)
```

再次强调，在条件编译指令!$和其后的源代码之间必须包含至少一个空格，否则条件编译指令无法正确识别，此行将当作注释行。

两个指令之前可以包含任意数量的空格，且只能为空格，否则将会被解释为注释行。

1.2.2 并行区域构造函数

OpenMP最重要的指令就是定义并行区域，这个区域是将由多线程并行执行的代码块。并行区域需要创建/打开和销毁/关闭，需要用到两个指令组成的指令对：

```
!$OMP PARALLEL
write(*,*) "Hello"
!$OMP END PARALLEL
```

指令对之间的代码会被每一个线程执行，并行区使用了多少线程，“Hello”将会在屏幕上出现多少次。并行区之外的代码只由单一线程执行，与串行程序的一般行为是一致的，因此它们被称为串行区域。

当正在执行串行区域的线程遇到并行区域时，它将创建一组新线程，自身成为其中的主线程。主线程是线程组中的一员，也参与执行计算。并行区域的每一个线程都有唯一的线程号，编号从0到Np-1，其中0为主线程，Np为线程总数。图1.1 阐明了前面示例的运行方式。

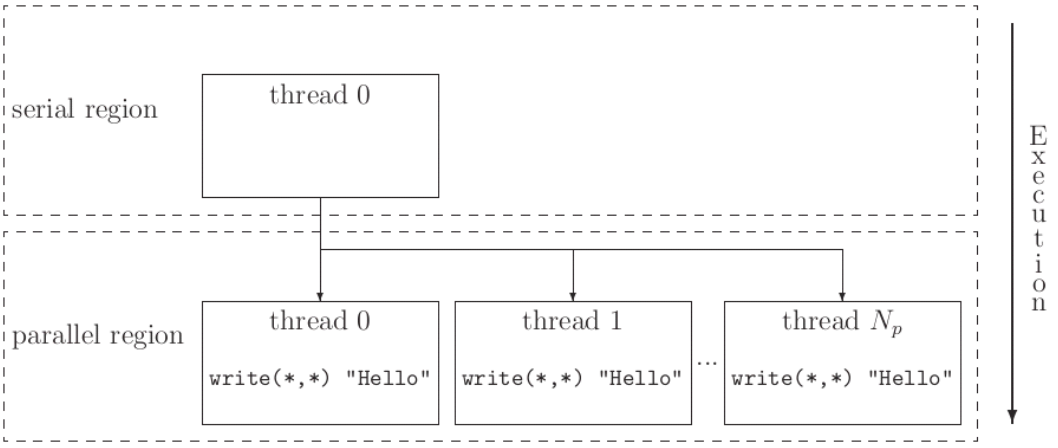

Figure 1.1: Graphical representation of the example explaining the working principle of the `!$OMP PARALLEL`/`!$OMP END PARALLEL` directive-pair.

在并行区域的开始部分，可以增加子句以限定并行区域的执行方式：如变量作用域、线程数、对某些变量的特殊处理等，形式如下：

```
!$OMP PARALLEL clause1 clause2 ...
...
!$OMP END PARALLEL
```

并不是所有在第三章详解的子句都可用于Opening指令，只有以下部分：

- PRIVATE(list): see page 37.
- SHARED(list): see page 38.
- DEFAULT(PRIVATE | SHARED | NONE) : see page 39.
- FIRSTPRIVATE(list): see page 40.
- COPYIN(list): see page 42.
- REDUCTION(operator:list): see page 43.
- IF(scalar logical expression): see page 46.
- NUM_THREADS(scalar integer expression): see page 47.

!$OMP END PARALLEL表示并行区域的结束。至此，每个线程里的局部变量（PRIVATE）都将释放，主线程继续执行并行区域后面的内容，其他线程将被删除。关闭并行区之前，主线程将等待所有其他线程执行完毕；否则数据将会丢失或者工作不能完成。这个等待在并行运行的线程之间是同步的，因此!$OMP END PARALLEL指令隐含着同步性。

当代码证包含并行区域时，为确保最终程序符合OpenMP规范，必须满足两个条件：

(1)!$OMP PARALLEL clause1 clause2 / !$OMP END PARALLEL 指令对必须出现在程序的同一文件中；

(2)并行区域内的代码必须为结构化代码，即不能跳入或跳出并行区，如使用GOTO语句。

除了以上两个规则，没有其他的限制条件。尽管如此，在使用并行区域的时候也得小心，因为即使考虑上述限制条件，也可能得到不正确的程序。

直接置于指令对!$OMP PARALLEL clause1 clause2 / !$OMP END PARALLEL 之间的代码块称为指令对的词法范围（lexical extent）。词法范围内的代码及其调用的代码均称为指令对的动态范围（dynamic extent），例如：

```
!$OMP PARALLEL
write(*,*) "Hello"
call be_friendly()
!$OMP END PARALLEL
```

此例中，子程序be_friendly包含的代码是指令对动态范围的一部分，但不属于词法范围。这两个概念非常重要，因为前面提到的子句中，有些只能用于词法范围，有些用于动态范围。

可以将平行区域嵌套到另一个平行区域。如果平行队列里的一个线程遇到另一个平行区域，它将创建新的线程组，本身称为新线程组中的主线程。第二个平行区域称为嵌套平行区域，示例如下：

```
!$OMP PARALLEL
write(*,*) "Hello"
      !$OMP PARALLEL
      write(*,*) "Hi"
      !$OMP END PARALLEL
!$OMP END PARALLEL
```

假设每个平行区域都使用Np个线程，总共将有Np^2+Np个消息出现在屏幕上。结果的树状结构如图1.2 。

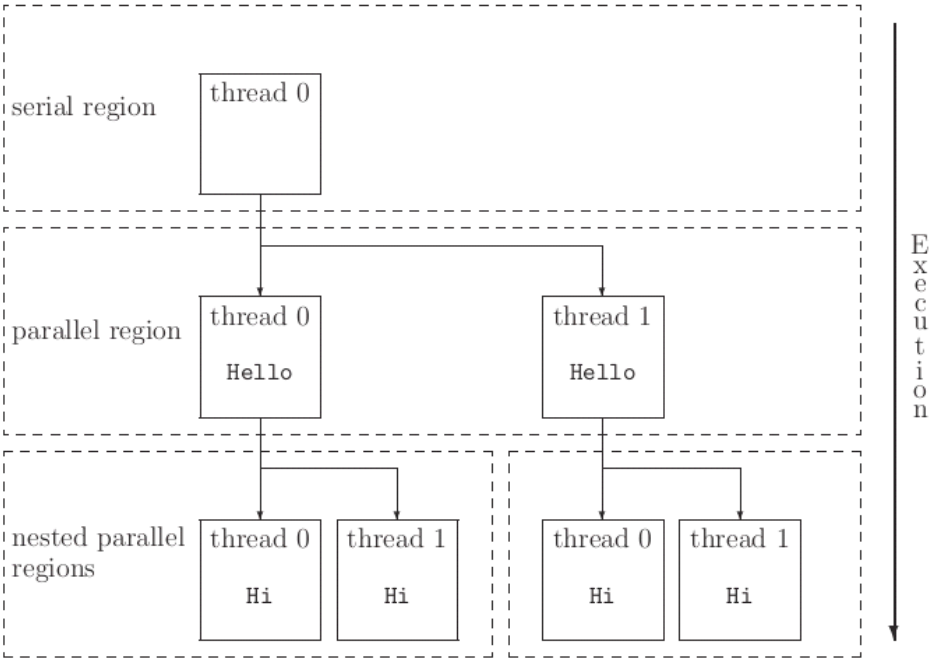

Figure 1.2: Graphical representation of the example explaining the concept of nested parallel regions.

第二章 OpenMP 构件

如果只有并行区域构件存在，所有线程只可能执行完全相同的任务，但这并不是并行的目的。因此，OpenMP定义了更多的构件，允许将一个任务分派给不同的线程，以生成一个真正的并行计算程序。

OpenMP指令或构件存在四种不同的群，每个群有不同的目的。选择群中的哪一个指令取决于待解决问题的性质。只有了解了每一个指令的使用原则才能执行正确的选择。

2.1 Work-sharing 构件

第一个OpenMP指令集意在将特定任务分解为片段，将一个或多个片段传送给每一个并行线程。这样，原本在串行程序里由单一线程执行的任务，分配给一组线程，得到更快的执行程序（要求处理器数量大于1。如果只有一颗处理器，多线程并行计算比串行计算更慢）。

所有的work-sharing构件都必须置于平行区域的动态范围（dynamic extends）内才有效。如若不是这种情况，work-sharing构件也会执行，但线程组中只有一个线程会执行，原因是work-sharing构件不会创建新线程；它是!$OMP PARALLEL/!$OMP END PARALLEL指令对中的任务。

使用work-sharing构件有如下限制：

（1）work-sharing构件必须被所有线程执行或者没有任何线程执行

（2）work-sharing构件必须依次被线程组中的线程执行

在关闭指令处，所有work-sharing构件都是隐式同时的。一般情况下，需要确认的一点是：在work-sharing构件之后的代码所请求的所有信息是最新的。但这种线程同时性并不总是必要的，因为这是一种浪费资源的事情（等待所有线程到达后才继续执行后续代码），因此需要一种抑制机制。在关闭指令后连接一特定子句—**NOWAIT** 。关于**NOWAIT**的更多信息见第三章。

2.1.1 !$OMP DO/!$OMP END DO

指令对使最近的do循环并行执行。例如：

```
!$OMP DO
do i = 1, 1000
...
end do
!$OMP END DO
```

将do循环分散到不同的线程：每个线程仅计算部分迭代。如，有10个可用线程，一般每

个线程计算100次迭代，线程0计算1-100，线程1计算101-200 。详见图2.1 。

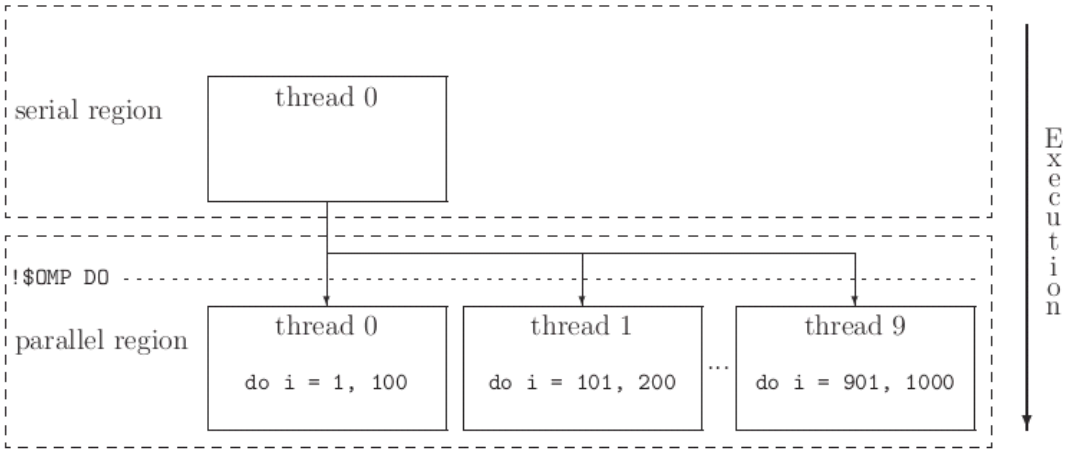

Figure 2.1: Graphical representation of the example explaining the general working principle of the !$OMP DO/!$OMP END DO directive-pair.

工作方式为分布式，可在!$OMP DO后连接子句来控制work-sharing构件的执行行为，语法如下：

```
!$OMP DO clause1 clause2 ...
...
!$OMP END DO end_clause
```

只有以下子句可用于打开指令!$OMP DO

- PRIVATE(list): see page 37.
- FIRSTPRIVATE(list): see page 40.
- LASTPRIVATE(list): see page 41.
- REDUCTION(operator:list): see page 43.
- SCHEDULE(type, chunk): see page 48.
- ORDERED : see page 52.

除了打开指令可以添加子句，关闭指令也可以添加**NOWAIT**子句已取消隐含的同时性。关闭指令还隐式更行do循环影响的共享变量。添加**NOWAIT**子句后，隐式更新亦被抑制。因此要特别注意，循环结束后是否使用了修改后的变量。如此就有必要添加对共享变量的隐式或显式更新，如使用!$OMP FLUSH指令。这种情况在其他指令中也有发生，尽管我们不再明确地提到。因此很方便阅读关于!$OMP FLUSH指令和NOWAIT子句的其他信息。第三章对概念进行详解，帮助理解其在抑制变量的隐式更新方面的影响。

由于并行do循环的工作被分配到一组并行线程，单个或多个线程插入或跳出由!$OMP DO/!$OMP END DO指令限定的代码块是没有意义的，比如GOTO命令。所以这种可能性被OpenMP规范直接禁止。

由于每个线程执行循环的一部分迭代，直到work-sharing构件的结束才能更新修改后的变量。下面的例子使用!$OMP DO/!$OMP END DO指令对进行并行计算，但得不到正确的结果：

```
real(8) :: A(1000), B(1000)
```

```
do i = 1, 1000
B(i) = 10 * i
A(i) = A(i) + B(i)
End do
```

因为矩阵B的正确值在work-sharing构件的$OMP END DO指令结束之前是不确定的。事实上，循环迭代以未知方式分布于不同的线程，并不是确定的。例如：

```
real(8) :: A(1000)
do i = 1, 999
A(i) = A(i+1)
End do
```

错误的原因是：在执行第i次迭代时，需要索引i+1处的未修改值。串行执行时毫无问题，但在并行计算时，不能获取到i+1处的未修改值。这种情况称为竞赛条件（racing condition）：结果依赖于线程执行顺序和每颗处理器的速度。修改上述循环，得到以下的并行版本代码：

```
real(8) :: A(1000), dummy(2:1000:2)
!Saves the even indices
!$OMP DO
do i = 2, 1000, 2
      dummy(i) = A(i)
End do
!$OMP END DO
!Updates even indices from odds
!$OMP DO
do i = 0, 998, 2
      A(i) = A(i+1)
End do
!$OMP END DO
!Updates odd indices with evens
!$OMP DO
do i = 1, 999, 2
      A(i) = dummy(i+1)
End do
!$OMP END DO
```

尽管以上的源代码没有明确表明，但完整的do循环需要放置在平行区域（parallel region）里面，否则将不会并行执行。示例放弃了矩阵A、B的定义，因为它们不能获得私有属性（见第三章，私有和共享变量）。

将循环分解为多个循环的技术可以解决一些问题，但也有必要依据时间和内存开销来综合评估其代价，看是否值得这样做。另一个有问题的循环如下：

```
real(8) :: A(0:1000)
```

```
do i = 1, 1000
    A(i) = A(i-1)
enddo
```

循环的每一次迭代依赖于前一次迭代，也依赖于循环的迭代顺序。但是这一次，以前提出的拆分循环的技巧是不可行的。除了使用!$OMP DO/!$OMP END DO指令对以外，还必须在do循环里面使用**ORDERED**声明强调顺序执行：

```
real(8) :: A(0:1000)
!$OMP DO ORDERED
do i = 1, 1000
    !$OMP ORDERED
    A(i) = A(i-1)
    !$OMP END ORDERED
enddo
!$OMP END DO
```

这样做就不能获得并行计算的优势，因为do循环内的所有代码都是顺序执行的。关于!$OMP ORDERED /!$OMP END ORDERED指令对的详细信息将在后面叙述。当出现几个嵌套循环时，很方便对最外层循环做并行运算，分配给每个线程的工作量达到最大，同时!$OMP ORDERED /!$OMP END ORDERED指令对的执行次数最少，这就意味着额外开销最小。看下面一个例子：

```
do i = 1, 10
    do j=1,10
    !$OMP DO
        do k = 1, 10
            A(i,j,k) =i*j*k
        enddo
    !$OMP END DO
    enddo
enddo
```

并行计算时，任务被拆分i*j=100次，每个线程获得少于10次迭代，因为只有最里层的循环是并行的。改变OpenMp指令的位置：

```
!$OMP DO
do i=1,10
    do j = 1, 10
        do k = 1, 10
        A(i,j,k) =i*j*k
        enddo
    enddo
enddo
```

```
!$OMP END DO
```

并行计算时，任务仅被拆分1次，每个线程获得至少100次迭代。因此第二总情形可获得更好的并行性能。

修改循环顺序，还可以提升结果代码的效率：

```
!$OMP DO
do k=1,10
      do j = 1, 10
            do i = 1, 10
            A(i,j,k) =i*j*k
            enddo
      enddo
enddo
!$OMP END DO
```

新代码更好地使用处理器的缓存，因为在连续内存地址中进行循环可获得更快的代码（Fortran90并没有指定数组存储的顺序，但Fortran77按列存储；建议读者仔细查看所使用的编译器的数组存储方式）。当然并不总是可以修改循环顺序：如果这样，就有必要在循环效率和并行效率之间寻找一个平衡点。

2.1.2 !$OMP SECTIONS/!$OMP END SECTIONS

指令对允许给每个线程分派完全不同的任务，生成MPMD（Multiple Programs Multiple Data）执行模块。每段代码仅被一个线程执行一次，work-sharing构件的语法如下：

```
!$OMP SECTIONS clause1 clause2 ...
!$OMP SECTION
...
!$OMP SECTION
...
...
!$OMP END SECTIONS end_clause
```

每段代码被唯一线程执行，以!$OMP SECTION指令开始，直到下一个!$OMP SECTION指令；或者以!$OMP END SECTIONS指令结束。可在当前指令对内定义任意数量的代码段，但只有已经存在的线程才可以分派代码块。这意味着如果代码段的数量大于可用线程数，部分线程将会串行执行不止一段代码。如果代码段的数量少于线程数，又将导致有效资源的低效率使用。举例来说，如果由4线程执行5个代码段，在第四个线程执行第五个代码段时，另外3个线程会被闲置。打开指令!$OMP SECTION接受以下子句：

- PRIVATE(list): see page 37.
- FIRSTPRIVATE(list): see page 40.

- LASTPRIVATE(list): see page 41.
- REDUCTION(operator:list): see page 43.

关闭指令!$OMP END SECTIONS只接受NOWAIT子句。使用!$OMP SECTIONS / !$OMP END SECTIONS指令对有以下约束：

（1）每个代码片段里面所包含的代码必须为结构化的代码块：不允许插入或跳出，如GOTO语句。

（2）所有的!$OMP SECTION指令必须位于!$OMP SECTIONS / !$OMP END SECTIONS指令对的词汇范围（lexical extend）：二者必须在相同的文件（routine）内。

应用实例如下：

```
!$OMP SECTIONS
!$OMP SECTION
write(*,*) "Hello"
!$OMP SECTION
write(*,*) "Hi"
!$OMP SECTION
write(*,*) "Bye"
!$OMP END SECTIONS
```

“Hello”、“Hi”、“Bye”这三个信息只各在屏幕上出现一次（图2.2）。OpenMP规范并没有指定以何种方式将任务分派给不同的线程，而是将这个工作留给了编译器厂商自己决定。

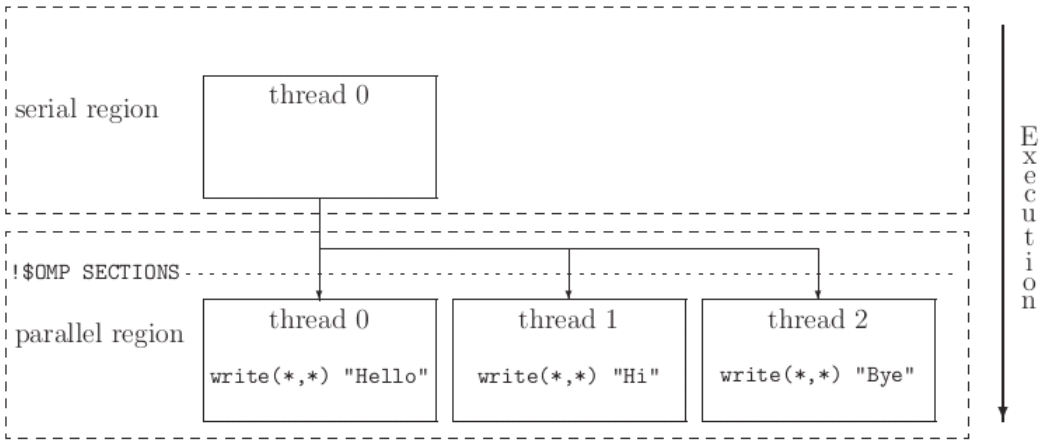

Figure 2.2: Graphical representation of the example explaining the working principle of the !$OMP SECTIONS/!$OMP END SECTIONS directive-pair.

2.1.3 !$OMP SINGLE/!$OMP END SINGLE

指令对包含的代码仅由其中一条线程执行，也就是最先到达!$OMP SINGLE指令的线程。如果没有指定NOWAIT子句，其余的线程将在!$OMP END SINGLE处等待（隐式同时性）。

指令对格式如下：

```
!$OMP SINGLE clause1 clause2 ...
...
!$OMP END SINGLE end_clause
```

结束子句可以是NOWAIT子句或者COPYPRIVATE子句，但二者不能同时存在，COPYPRIVATE子句的功能将在第三章介绍。仅有下面两个子句可用于打开指令：

- PRIVATE(list): see page 37.
- FIRSTPRIVATE(list): see page 40.

OpenMP规定：指令对里面的代码不能有插入或跳出语句，如GOTO语句。

```
!$OMP SINGLE
write(*,*) "Hello"
!$OMP END SINGLE
```

“Hello”只在屏幕出现一次。其余线程并不执行指令对内的代码，而是在关闭指令处闲置、等待。使用NOWAIT子句可消除上述问题，但为了得到正确的结果，应该确保其他线程所做工作与指令对内的任务是无关的（图2.3）。同时，应该注意一下规则：

（1）实线代表SINGLE区域以外同时进行的工作

（2）虚线代表未执行指令对内代码块的线程正在闲置状态，代码块仅由唯一线程执行

（3）信号灯代表相关线程是否包含显示或隐式同时性（红灯表示具有同时性）

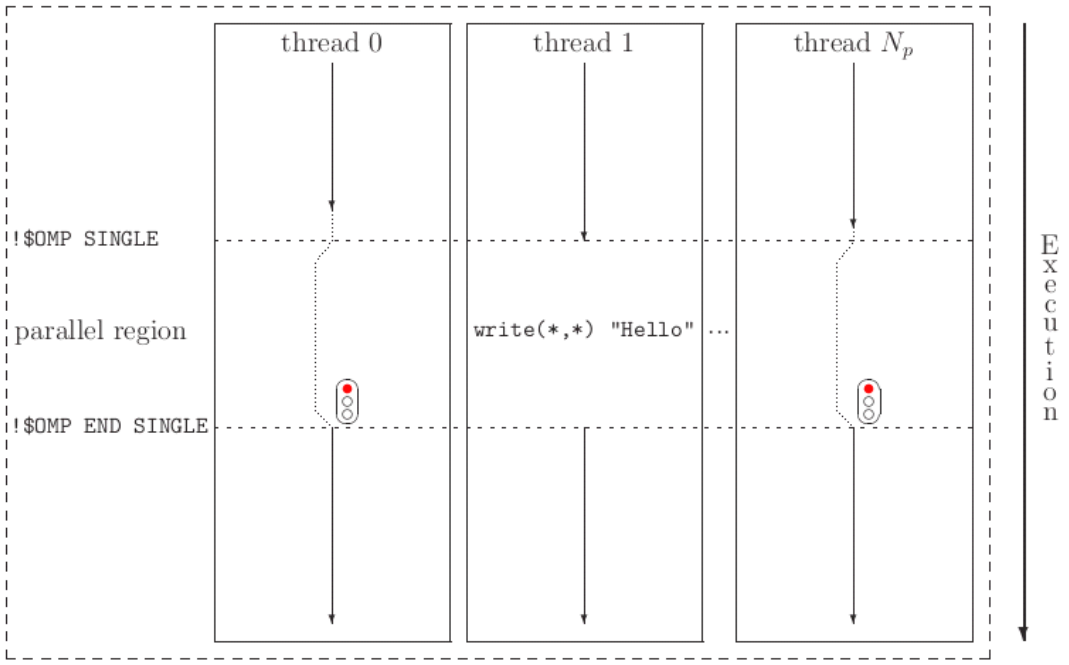

Figure 2.3: Graphical representation of the example explaining the working principle of the !$OMP SINGLE/!$OMP END SINGLE directive-pair.

2.1.4 !$OMP WORKSHARE/!$OMP END WORKSHARE

在此之前，Fortran95的部分命令，如数组符号表达式、forall 和 where 声明，由于没有明确的Do循环标志，不能被OpenMP指令识别。提出work-sharing构件的目标就是为了允许对这些声明进行并行化处理。除了以上3种情况，以下的数组转换函数也能用!$OMP WORKSHARE /!$OMP END WORKSHARE指令对处理： **matmul, dot_product , sum , product , maxval, minval, count , any , all , spread, pack, unpack,reshape , transpose , eoshift , cshift, minloc and maxloc** 。

```
!$OMP WORKSHARE
...
!$OMP END WORKSHARE end_clause
```

如果关闭指令处未指定NOWAIT子句，在**!$OMP END WORKSHARE**子句处隐含同时性。

与前面提到过的work-sharing构件相比，当前叙述的指令对中的代码块执行如下：执行完成前一个声明以后才开始执行下一个声明，好像串行执行一样。即是说在执行下一个声明之前，必须获得上一个声明的执行结果，任务右边的求值完成以后才赋值给左边的部分。

```
real(8) :: A(1000), B(1000)
!$OMP DO
do i = 1, 1000
      B(i) = 10 * i
      A(i) = A(i) + B(i)
enddo
!$OMP END DO
```

上面的例子不能正确执行。我们做一些改动：

```
real(8) :: A(1000), B(1000)
!$OMP WORKSHARE
forall(i=1:1000)
B(i) = 10 * i
end forall
A=A+B
!$OMP END WORKSHARE
```

为使Fortran语句被!$OMP WORKSHARE/!$OMP END WORKSHARE指令对接受，允许OpenMP插入许多同时性(synchronizations)，以满足硬性要求。结果因为这些额外添加的同时性不可避免地导致了冗余（overhead）。冗余量依赖于OpenMp正确解释和转化指令对中包含的语句的能力。

!$OMP WORKSHARE/!$OMP END WORKSHARE指令对的工作原理是将任务分解为工作单元，并分派给每个线程。OpenMP规范使用下列规则划分工作单元：

（1）声明中的数组表达式：数组表达式中每个元素的求值都被认为是一个工作单元

（2）对内在数组转换函数求值可分为任意多个工作单元

（3）如果WORKSHARE指令被用于数组分配声明，对每个元素的分配都可以分为一个工作单元

（4）如果WORKSHARE指令被用于具有数组参数的基本函数，函数将作用于数组的每一个元素，看作一个工作单元

（5）如果WORKSHARE指令被用于where声明，对掩码表达式求值及最终的结果（masked assignments）都是workshared

（6）如果WORKSHARE指令被用于forall声明，对掩码表达式求值、循环里面的表达式以及结果（masked assignments）都是workshared

（7）对于**ATOMIC**指令或相应的任务，对每个标量变量的更新都是一个工作单元

（8）对于**CRITICAL**构件，每个构件都是一个工作单元

（9）如果代码块内部分代码没有添加以上任何一个规则，这些代码被看做一个工作单元。

运用上述规则，可为不同线程提供大量工作单元；单元的分派方式由OpenMp决定。要正确使用**!$OMP WORKSHARE**指令，考虑如下限制条件：

（1）指令对内的代码为结构化代码

（2）代码只能包含数组赋值语句（array assignment statements）、标量赋值语句（scalar assignment statements）、forall和where语句

（3）除纯函数（pure）外，不能包含用户自定义函数，以避免副作用。函数声明为elemental

（4）指令对内修改或被引用的变量必须有shared属性，否则结果难料。

!$OMP WORKSHARE指令的范围仅限定于指令对的词汇范围。

2.2 Combined parallel work-sharing 构件

Combined parallel work-sharing 构件是指定仅包含一个work-sharing构件的并行区域的捷径，与!$OMP PARALLEL/!$OMP END PARALLEL指令中的单一work-sharing构件的行为是完全一样的。它的存在是为了在OpenMP-implementation和OpenMp指令同时出现时，降低冗余开销。

2.2.1 !$OMP PARALLEL DO/!$OMP END PARALLEL DO

这是指定一个包含单一!$OMP DO/!$OMP END DO 指令的并行区域的快捷方式：

```
!$OMP PARALLEL DO clause1 clause2 ...
...
!$OMP END PARALLEL DO
```

可使用!$OMP PARALLEL或!$OMP DO其中之一的子句。

2.2.2 !$OMP PARALLEL SECTIONS/!$OMP END PARALLEL SECTIONS

语法如下：

```
!$OMP PARALLEL SECTIONS clause1 clause2 ...
...
!$OMP END PARALLEL SECTIONS
```

说明同上。

2.2.3 !$OMP PARALLEL WORKSHARE/!$OMP END PARALLEL WORKSHARE

```
!$OMP PARALLEL WORKSHARE clause1 clause2 ...
...
!$OMP END PARALLEL WORKSHARE
```

仅使用**!$OMP PARALLEL的子句**因为!$OMP WORKSHARE没有子句。

2.3 Synchronization constructs（同步构件）

实际工作中不可能让各个线程自己运行，必须按顺序收回，一般使用线程同步。同步可以是显式的，也可以是隐式的，二者功能相同。阅读本节内容，理解是否启用隐式同步的不同之处。

2.3.1 !$OMP MASTER/!$OMP END MASTER

指令对中的代码只被主线程执行，其他线程继续执行它们的工作，没有隐含同步。

```
!$OMP MASTER
write(*,*) "Hello"
!$OMP END MASTER
```

在本质上与!$OMP SINGLE/!$OMP END SINGLE + NOWAIT子句相似，只是把执行代码的线程指定为主线程而不是最先到达的线程，执行后主线程位于其他线程后面（图2.4）。代码必须为结构化代码。

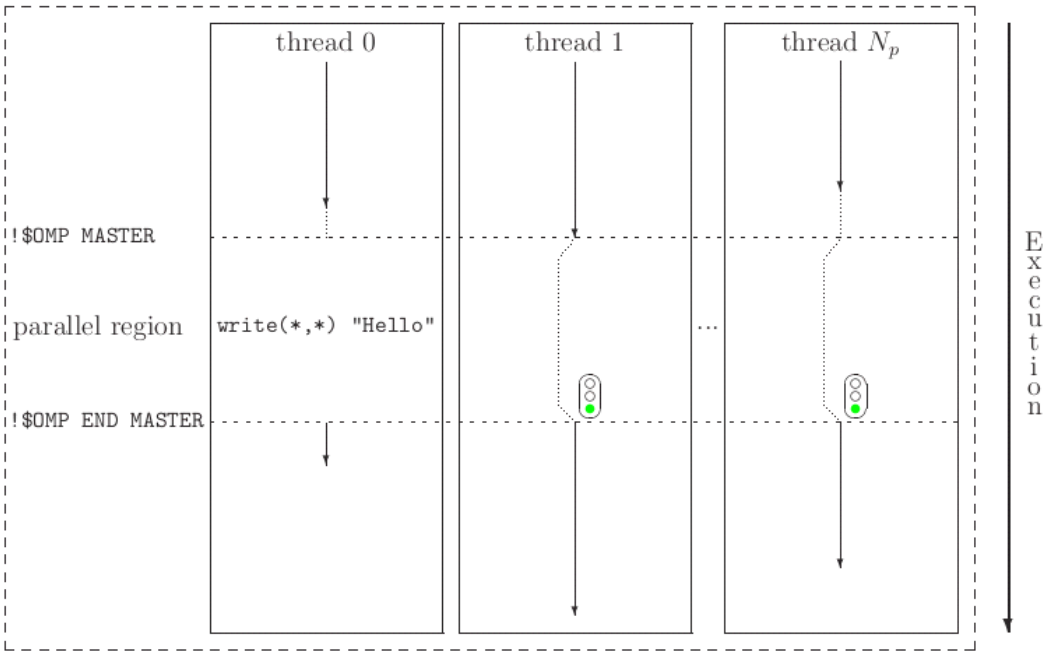

Figure 2.4: Graphical representation of the example explaining the working principle of the !$OMP MASTER/!$OMP END MASTER directive-pair.

2.3.2 !$OMP CRITICAL/!$OMP END CRITICAL

每次只有一条线程进入内部代码块，以确保内部代码正确执行。

```
!$OMP CRITICAL name
...
!$OMP END CRITICAL name
```

可选参数name用于识别临界区域。虽然不是强制的，但还是建议为每个区域添加Name 。当一条线程到达临界区域的开始部分时，它会在此等待，直到没有其他线程执行其中的代码为止。有相同名字的不同临界区域被看做共同的临界区域，一次只有一个线程在里面。此外，所有的未命名临界区都将看作同一个区域，这就是我们建议读者给它命名的原因。

下面演示从键盘或文件读取数据更新变量：

```
!$OMP CRITICAL write_file
write(1,*) data
!$OMP END CRITICAL write_file
```

如图2.5 ，线程0在等待Np退出临界区域。在Np之前，线程1执行临界区而Np则处于等

待状态。

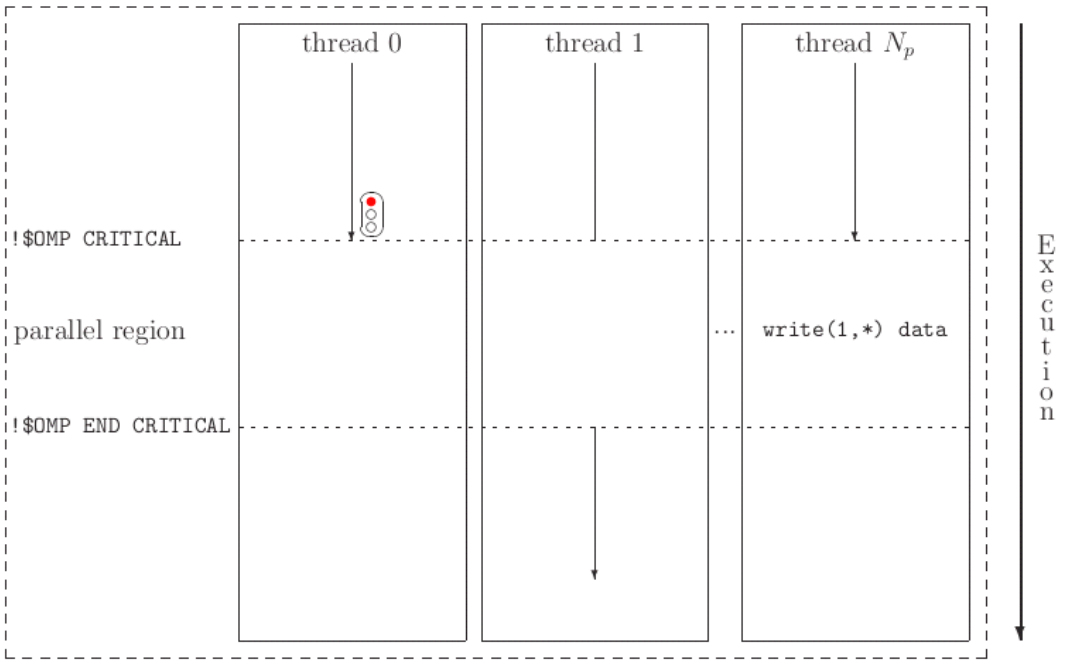

Figure 2.5: Graphical representation of the example explaining the working principle of the !$OMP CRITICAL/!$OMP END CRITICAL directive-pair.

2.3.3 !$OMP BARRIER

指令代表线程显式同步，某个线程遇到它时，处于等待状态，直到所有线程都到达。使用显式同步有两个限制条件：

（1）要么所有线程都到达阻塞点，要么没有任何线程到达。某些情况不满足条件，如：

```
!$OMP CRITICAL
!$OMP BARRIER
!$OMP END CRITICAL
```

因为临界区（CRITICAL）里同时只能有一条线程，其他线程不可能到达显式同步点，程序不能退出，陷入死锁（deadlock）。应避免上述情况。下面列举了一些死锁的例子：

（1）

```
!$OMP SINGLE
!$OMP BARRIER
!$OMP END SINGLE
```

（2）

```
!$OMP MASTER
!$OMP BARRIER
!$OMP END MASTER
```

（3）

```
!$OMP SECTIONS
```

```
!$OMP SECTION
!$OMP BARRIER
!$OMP SECTION
...
...
!$OMP END SECTIONS
```

这些死锁能够明显看出，但也有一些情况下，死锁可能出现却不易发现，比如循环或if语句。

```
if(my_thread_ID < 5) then
!$OMP BARRIER
endif
```

例中，如果线程数小于5，则没有问题。但如果线程数大于5，有些线程不能够到达阻塞点，程序陷入死锁。

（2）**!$OMP BARRIER**指令必须被线程依次遭遇。

请根据实际情况使用这个指令，因为它会造成资源的闲置浪费，除非必须，尽量不要使用。认真分析程序的源代码，看看是否有其他不必使用同步的方法。这条建议也适用于OpenMp中的其他显式或隐式包含同步的地方。

图2.6 表示使用了**!$OMP BARRIER**指令后的影响。可以看出，线程1必须在显式同步点等待所有线程都到达。线程0也必须等待线程Np到达。最后，当所有线程均到达阻塞点，程序才继续执行后续操作。

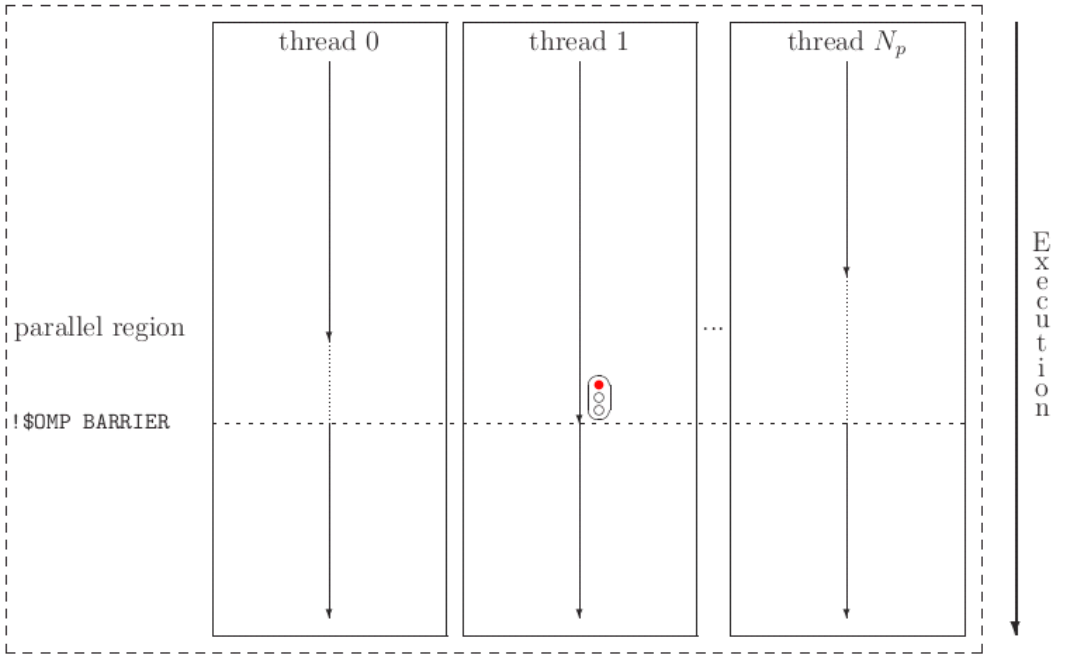

Figure 2.6: Graphical representation of the effect of the `!$OMP BARRIER` directive on a team of threads.

2.3.4 !$OMP ATOMIC

如果使用的变量能被所有线程修改，就有必要确保同时只有一个线程在修改变量的内存地址，否则会造成不可预料的结果。

```
!$OMP DO
do i = 1, 1000
a=a+i
enddo
!$OMP END DO
```

期望的结果是循环变量i的累加值，但这不一定能行。当前的目的是确保特定内存地址不被多个线程同时更新，语法很简单：

```
!$OMP ATOMIC
```

指令仅对后面的一个语句有效。不是所有的语句都可以使用这个指令，仅如下语句可用：

```
x = x operator expr
x = intrinsic_procedure(x, expr_list)
```

只有下列操作符和内部过程可用：

operator = +, *, -, /, .AND., .OR., .EQV. or .NEQV.

intrincic_procedure = MAX, MIN, IAND, IOR or IEOR

变量x必须是标量性质，且为内部数据类型，当然表达式expr也必须是标量。

必须符合所有限制条件（接下来还会介绍一些），才能确保语句被以atomic方式对待且以有效方式执行。下面将介绍一些额外的字。

加载和存储变量x的行为仅可以是atomic方式的，意味着expr的求值可在所有线程同时进行。这个特性产生更快的代码，限制expr的种类，因为不能利用变量x。使用**!$OMP CRITICAL/!$OMP END CRITICAL**指令对也可得到类似的效果：

```
xtmp = expr
!$OMP CRITICAL x
x = x operator xtmp
!$OMP END CRITICAL x
```

!$OMP ATOMIC指令的意义在于提供比**!$OMP CRITICAL/!$OMP END CRITICAL**指令对更加优化的代码，由OpenMP-implementation执行，因此所获得的优化度也与OpenMP-implementation有关。

2.3.5 !$OMP FLUSH

指令或明或暗定义了一个序列点，在此implementation被要求确保每个线程都能一致地访问特定变量内存：每个线程都能读取到相同的正确值。指令必须精确出现在代码要求数据同步（data synchronization）的地方。

乍一看来，这个指令貌似不必要，因为每次仅有一个线程可以更新共享变量。但这仅是在理论上成立，因为OpenMP-implementation模拟”one thread at a time”特性的功能并不在OpenMp规范里面。这不是错误，而是留给OpenMP-implementation的接口，以优化代码。如以下循环：

```
!$OMP DO
do i = 1, 10000
!$OMP ATOMIC
A=A+i
enddo
!$OMP END DO
```

变量A被atomic访问1000次，并不高效，因为每次只有一个线程工作。修改代码：

```
Atmp = 0
!$OMP DO
do i = 1, 1000
Atmp = Atmp + i
enddo
!$OMP END DO
!$OMP ATOMIC
A = A + Atmp
```

每个线程中Atmp都是临时变量，变量A被atomic访问Np次。后一个例子执行更高效。相同的构想很可能以不同的方式执行，因为编译器可能优化**!$OMP ATOMIC**指令。但是优化后的版本中，只有到结尾处变量A才有正确值。如果不注意，将带来错误。

指令意在更新共享变量，使后续代码可以正确执行。OpenMP-implementations必须保证编译器有额外的代码将值从寄存器移到内存，例如在硬件设备上冲写缓冲区。

不同线程对一个共享数组的不同部分进行操作时，这个指令也很重要。某些点可能需要使用收其他线程影响的部分的信息，如此就必须保证所有的写入/更新进程在读取数组之前执行，这就需要**!$OMP FLUSH**指令。

确保查看变量的一致性很大程度上取决于不同内存地址的信息交换，没有必要在指定点更新所有的共享变量：**!$OMP FLUSH**指令提供了需要刷新的变量列表：

```
!$OMP FLUSH (variable1, variable2, ...)
```

表达式代表在指定点的显式数据同步。下列OpenMP指令包含隐式数据同步：

- !$OMP BARRIER

- !$OMP CRITICAL and !$OMP END CRITICAL
- !$OMP END DO
- !$OMP END SECTIONS
- !$OMP END SINGLE
- !$OMP END WORKSHARE
- !$OMP ORDERED and !$OMP END ORDERED
- !$OMP PARALLEL and !$OMP END PARALLEL
- !$OMP PARALLEL DO and !$OMP END PARALLEL DO
- !$OMP PARALLEL SECTIONSand !$OMP END PARALLEL SECTIONS
- !$OMP PARALLEL WORKSHARE and !$OMP END PARALLEL WORKSHARE

以下指令没有隐式数据同步：
- !$OMP DO
- !$OMP MASTER and !$OMP END MASTER
- !$OMP SECTIONS
- !$OMP SINGLE
- !$OMP WORKSHARE

很重要的一点，当使用了NOWAIT子句抑制隐式线程同步后，隐式数据同步也将被抑制。

2.3.6 !$OMP ORDERED/!$OMP END ORDERED

在循环中，一些语句在每次迭代中需要依次赋值，就像在串行程序里一样。

```
do i = 1, 100
A(i) = 2 * A(i-1)
enddo
```

在这个循环中，执行第60次迭代之前必须执行第59次迭代。这种情况要并行整个循环是不可能的。更一般的例子：

```
do i = 1, 100
block1
block2
block3
enddo
```

Block2需要按正确的顺序赋值，其他block则可以任意顺序执行。一种解决方案是将之拆分为3个循环，并行第一和第三个循环；另一种方案就是使用指令向编译器声明部分代码需要顺序执行。

```
!$OMP DO ORDERED
do i = 1, 100
```

```
block1
!$OMP ORDERED
block2
!$OMP END ORDERED
block3
enddo
!$OMP END DO
```

为**!$OMP DO**的打开指令强制添加**ORDERED**子句，不同代码块的执行顺序见图2.7 。

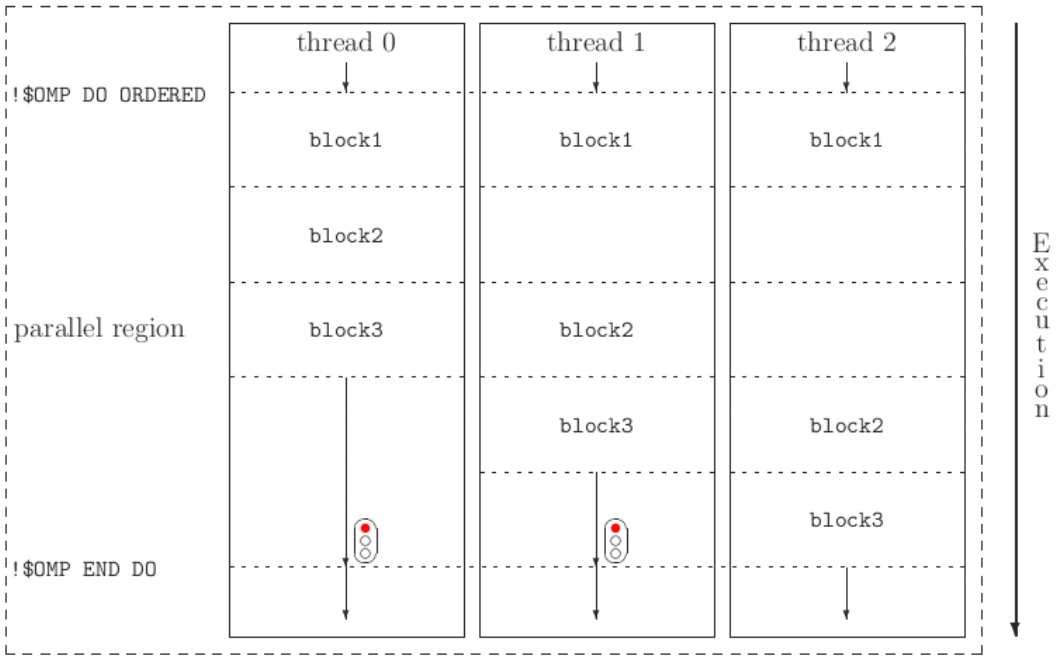

Figure 2.7: Graphical representation of the sequence of execution of the example explaining the working principle of the `!$OMP ORDERED`/`!$OMP END ORDERED` directive-pair.

!$OMP ORDERED /!$OMP END ORDERED指令对只在循环的动态区域内有效。一方面只允许一个线程在其内运行，另一方面线程入口只能遵循循环迭代的顺序：前面的迭代完成之前，没有线程可以进入**ORDERED**指令区域。

本质上指令对与**!$OMP CRITICAL/!$OMP END CRITICAL**相似，但不包含隐式同步，入口顺序由循环的顺序决定。

除了结构化代码的限制外，并行循环里面每次迭代只能有一个ORDERED块。下面的语句是不允许的：

```
!$OMP DO ORDERED
do i = 1, 1000
...
!$OMP ORDERED
...
!$OMP END ORDERED
...
```

```
!$OMP ORDERED
...
!$OMP END ORDERED
...
enddo
!$OMP END DO
```

下列情形是允许的：虽然包含多个ORDERED块，但每次循环迭代只有一个ORDERED块可见。比如在if语句里：

```
!$OMP DO ORDERED
do i = 1, 1000
...
if(i < 10) then
!$OMP ORDERED
write(4,*) i
!$OMP END ORDERED
else
!$OMP ORDERED
write(3,*) i
!$OMP END ORDERED
endif
...
enddo
!$OMP END DO
```

2.4 Data environment constructs

最后一组OpenMP指令用于控制并行程序里的数据环境。有两种不同的数据环境构件：
（1）不依赖于其他OpenMP构件
（2）与其他OpenMP构件相关、只影响某个构件和它的文本域（数据作用域子句）

本章讨论第一种，第二种以及其他非数据环境构件子句在第三章讨论。第三章将统一介绍OpenMP子句。

2.4.1 !$OMP THREADPRIVATE (list)

有时需要用到全局变量，对每个线程来说，其值是特定的。My_ID保存每个线程的线程号：对每个线程来说，它是不同的，可在线程内任意地方提取它的属性值，从一个并行区到下一个并行区，其值不会改变。

!$OMP THREADPRIVATE(a, b指令在list定义了两个变量。对每个线程来说，a和b都是局部变量，但在进程里面确实全局的。 List中的变量必须位于COMMON块（以被module取

代）或者已经命名。已命名的变量如果不是在module中声明，则须有save属性。!$OMP THREADPRIVATE指令必须紧跟变量声明，且在主程序之前：

```
real(8), save :: A(100), B
integer, save :: C
!$OMP THREADPRIVATE(A, B, C)
```

当程序进入第一个并行区，由THREADPRIVATE标记的每个变量将会为各个线程创建私有拷贝。如果没有指定COPYIN子句，这些变量是没有初值的。

如果动态线程调整机制不可用，在下一个并行区的入口处，THREADPRIVATE变量的状态和值与在前一个并行区结尾处一致，除非使用COPYIN子句。

```
integer, save :: a
!$OMP THREADPRIVATE(a)
!$OMP PARALLEL
a = OMP_get_thread_num()
!$OMP END PARALLEL
!$OMP PARALLEL
...
!$OMP END PARALLEL
```

在第一个并行区，变量a获得各个线程的线程号。第二个并行区里，a保持其值不变，因为**THREADPRIVATE** 。图2.8 中虚线代表**THREADPRIVATE** 属性的影响。

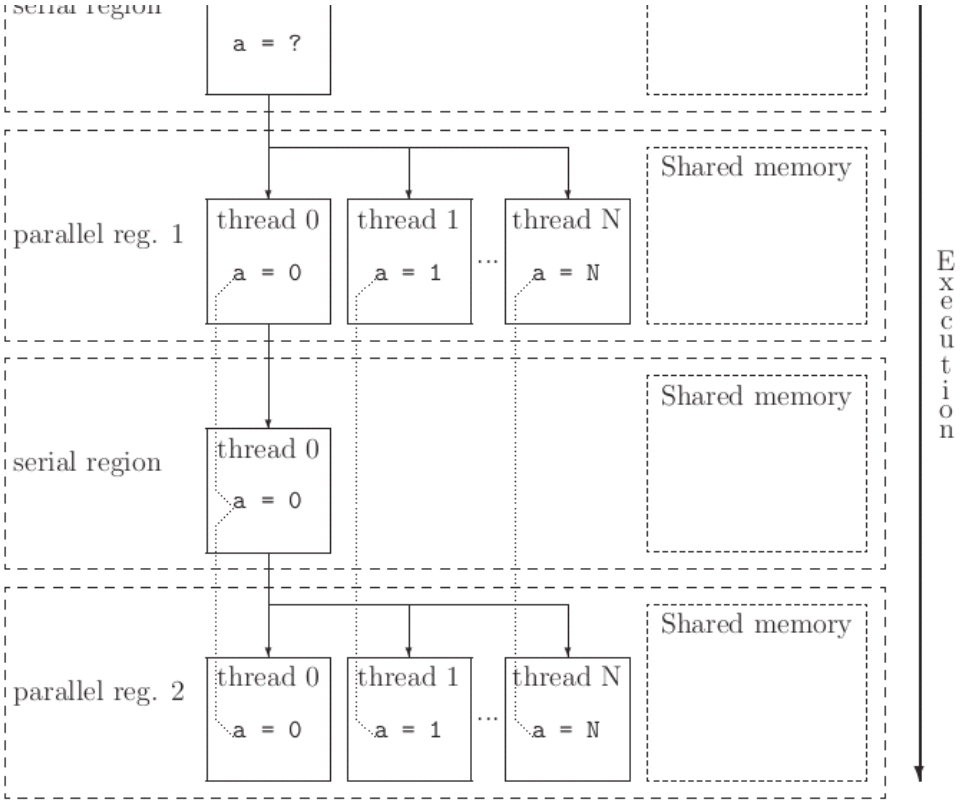

Figure 2.8: Graphical representation of the example explaining the working principle of the `!$OMP THREADPRIVATE` clause.

具有THREADPRIVATE属性的变量只能出现在COPYIN 和 COPYPRIVATE子句中。

第三章 PRIVATE , SHARED & Co.

前面介绍的许多指令都可以通过添加子句的形式改变工作方式。可以定义两种不同的子句：

（1）数据作用域属性子句。指定变量处理方式，以及说明哪些代码可以访问和修改变量。

（2）除了第一中的所有子句。

将要介绍的数据作用域属性子句中，并非所有都可用于全部指令。在前面的指令介绍中已经对可用子句进行了说明。尽管接下来的大多数例子使用**!$OMP PARALLEL/!$OMP END PARALLEL**指令对，并不意味着用法局限于这个指令对，只是因为这样使大多数概念比较易懂。

3.1 Data scope attribute clauses

3.1.1 PRIVATE(list)

只有当每个线程都有各自对变量的私有备份时，变量对各个线程才能有不同值。这个子句指定变量为各个线程的局部变量。

```
!$OMP PARALLEL PRIVATE(a, b)
```

变量a、b在不同线程中拥有不同的值，对于线程，它是局部变量。

变量被指定为私有属性后，会在每个线程里定义新的同类型对象，他将会取代原始变量（图3.1）。

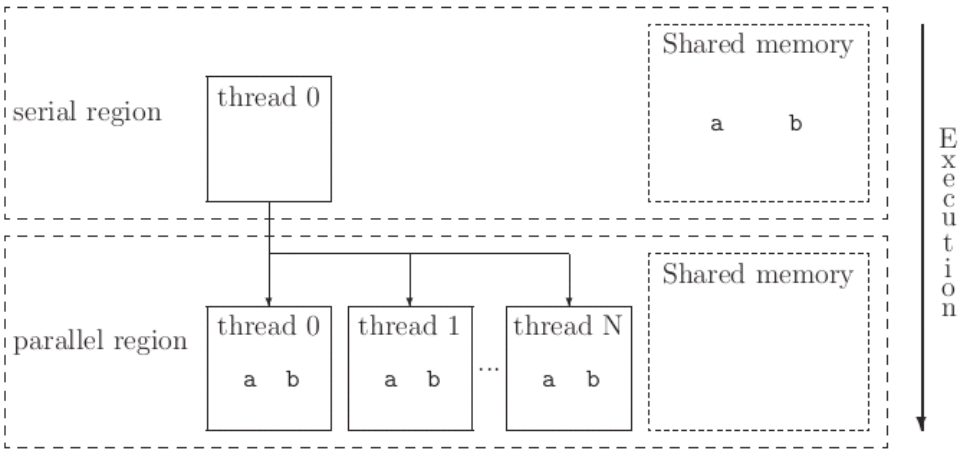

Figure 3.1: Graphical representation of the effect of the `PRIVATE` clause on the variables `a` and `b` of the presented example.

在指令对的开始部分，私有变量刚刚创建，其值未定义。指令对结束后，源变量的值也是不确定的（不知道应该用哪个线程的值）。

事实上为每个线程都创建对象是一件很耗费资源的事情：例如为每个并行域声明5G的私有数组，则需要总共55G的内存开销，这个数量是所有SMP机器都无法承受的。

尽管在指令对开始部分没有显式说明，循环计数器、 forall命令或THREADPRIVATE变量，自动认为是线程私有的。

3.1.2 SHARED(list)

有时候需要定义对所有线程都有效的变量，共其使用或修改：

!$OMP PARALLEL SHARED(c, d)

在!$OMP PARALLEL/!$OMP END PARALLEL 指令对内，所有线程都可以使用变量c、d（图3.2）。

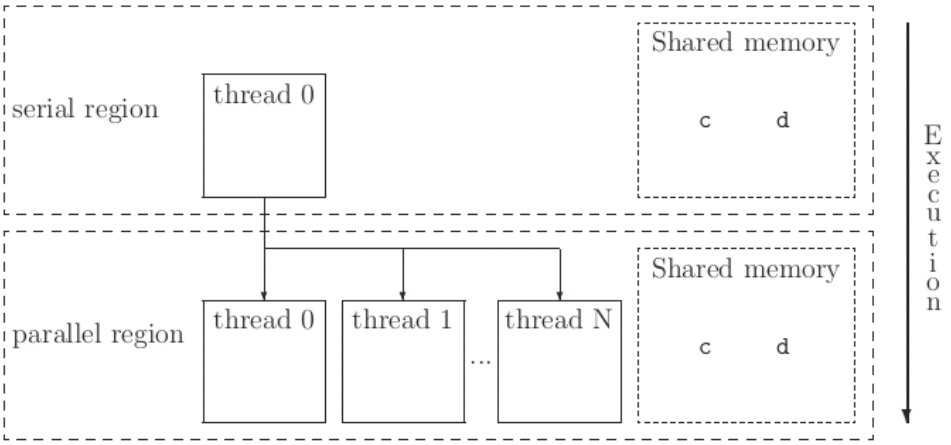

Figure 3.2: Graphical representation of the effect of the SHARED clause on the variables c and d of the presented example.

如果变量具有共享属性，则不需要开辟新的空间。线程在相同的地址读取或改变变量，因此不需要额外的开销。但这并不保证一个线程立即可以感知到另一个线程对变量的改变。OpenMP implementation将共享变量的值存储在临时变量里，后续才执行更新任务。可用!$OMP FLUSH执行共享变量的强制更新。

由于同一时间可能要多个线程刷新变量，其值是不确定的。这就是racing condition，程序员应该尽量避免这种情况。一种解决方案就是使用!$OMP ATOMIC强制writing过程是atomically的。

3.1.3 DEFAULT(PRIVATE | SHARED | NONE)

当指令对中多数变量属性为共享或私有，将他们全部都写入子句中显得很笨重。针对这种情况，可以指定default设置：

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(a)

除了变量a为共享属性，其余均为私有属性。如果没有指定 DEFAULT子句，默认为共享属性（图3.3）。

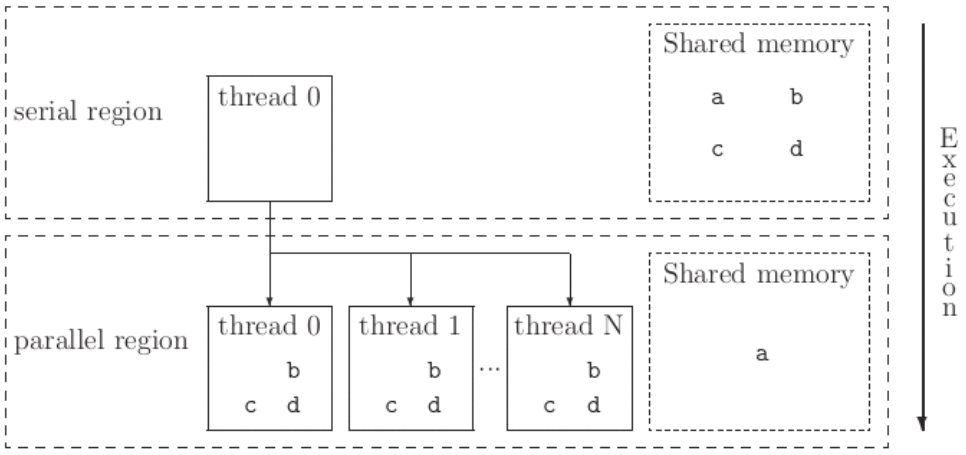

Figure 3.3: Graphical representation of the effect of the DEFAULT clause on the variables of the presented example.

除了PRIVATE 和 SHARED。还有NONe。指定DEFAULT(NONE)要求：指令作用域中每个变量必须在数据作用域属性中明确列出，除非变量是THREADPRIVATE、循环计数器、forall命令或者隐式循环。

DEFAULT子句只作用域文本域，定义在过程中的变量不受其影响，具有私有属性。

3.1.4 FIRSTPRIVATE(list)

正如之前所说，私有变量在指令对开头具有不确定的值，但有时候这些局部变量在指令对开始部分可以拥有初始值。将变量包含进FIRSTPRIVATE子句：

a=2
b=1
!$OMP PARALLEL PRIVATE(a) FIRSTPRIVATE(b)

此例中a具有不确定值，而b的初值为1（图3.4）。

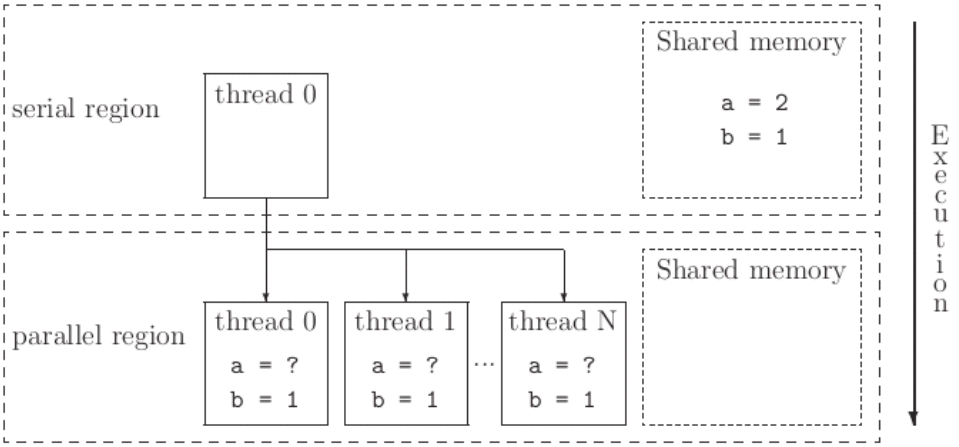

Figure 3.4: Graphical representation of the example given in the description of the FIRSTPRIVATE clause.

如果在指令对的开始部分将变量包含进FIRSTPRIVATE子句，则变量自动具有私有属性，而不需要再次声明其私有属性。

原变量中的信息需要拷贝到新的局部变量中，如将其声明为FIRSTPRIVATE属性，则非常耗时。同样的例子，如果有5G的数组和10个进程，则将转移50G的数据，非常耗时。

3.1.5 LASTPRIVATE(list)

私有变量在指令对结尾处具有不确定值，很多时候非常不方便。如将其置于LASTPRIVATE子句中，原变量将会被最后的值更新，就如同在串行程序里一样。

```
!$OMP DO PRIVATE(i) LASTPRIVATE(a)
do i = 1, 1000
a=i
enddo
!$OMP END DO
```

指令对完成后，a的值为1000 。另一个例子:

```
!$OMP SECTIONS LASTPRIVATE(a)
!$OMP SECTION
a=1
!$OMP SECTION
a=2
!$OMP END SECTIONS
```

执行最后一个SECTION的线程将会更新原变量a的值，a=2

如果循环的最后一次迭代或者最后一个SECTION语句中没能设置LASTPRIVATE中变量的值，则其最终的值不确定。

```
!$OMP SECTIONS LASTPRIVATE(a)
!$OMP SECTION
a=1
!$OMP SECTION
a=2
!$OMP SECTION
b=3
!$OMP END SECTIONS
```

通过last为原变量赋值的过程在所有线程中是同时进行的。如果这个隐含的同时不存在（例如 NOWAIT），那么在出现隐式或显式同时之前，变量值是未知的。它确保执行之后一次迭代的线程得出变量的终值。

信息在不同内存地址间传递，如果位于LASTPRIVATE中的变量size很大，则很耗时。与FIRSTPRIVATE子句相反，只有与变量相等尺度的信息才被移动，与线程数无关。

3.1.6 COPYIN(list)

具有THREADPRIVATE属性的变量，可用COPYIN子句将每个线程中的值设置为与主线程中的值相等：

```
!$OMP THREADPRIVATE(a)
!$OMP PARALLEL
a = OMP_get_thread_num()
!$OMP END PARALLEL
!$OMP PARALLEL
...
!$OMP END PARALLEL
!$OMP PARALLEL COPYIN(a)
...
!$OMP END PARALLEL
```

上例中，变量a在第一个并行区被赋值为线程号。第二个并行区中，因为具有THREADPRIVATE属性，其值不变。第三个并行区中，将所有线程中的a的值都设为0，即主线程号（图3.5）。这跟FIRSTPRIVATE子句相似，不同的是：它应用于THREADPRIVATE变量中，而不是PRIVATE变量。

使用COPYIN子句的计算开销与FIRSTPRIVATE接近，因为变量值存储的值需要拷贝到所有线程中去。

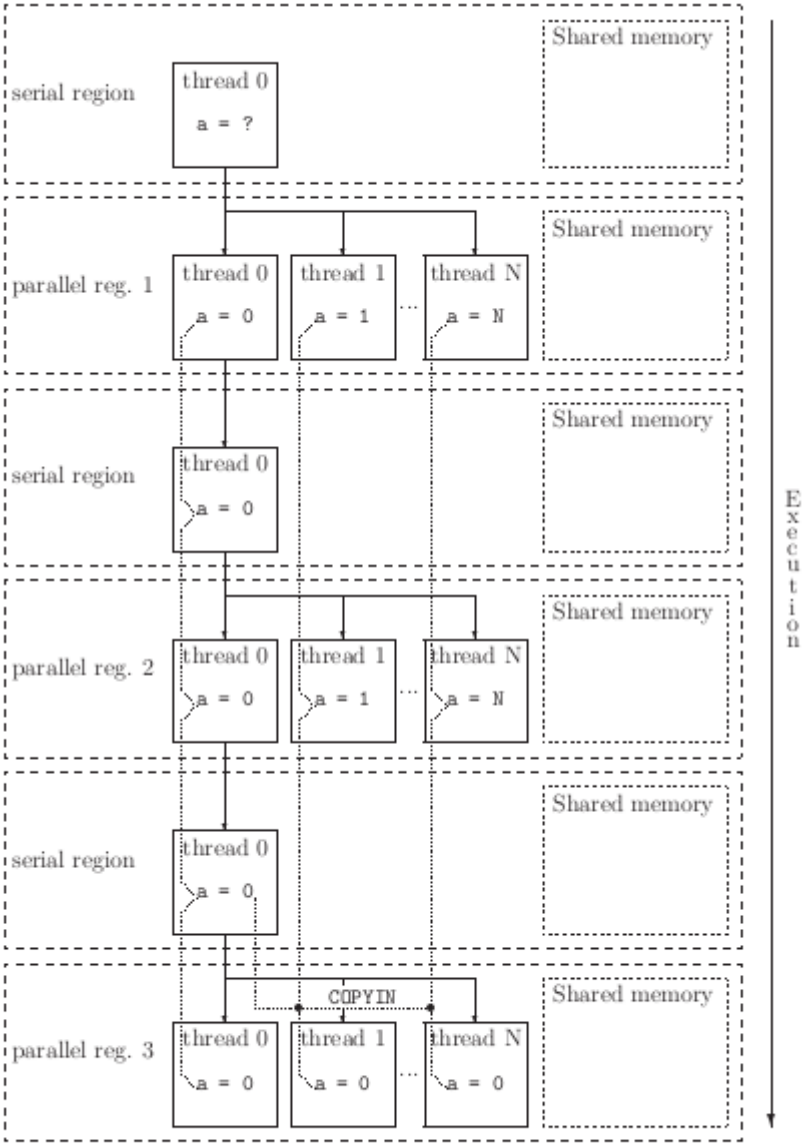

Figure 3.5: Graphical representation of the example given in the description of the COPYIN clause.

3.1.7 COPYPRIVATE(list)

并行区里的单一线程执行了位于!$OMP SINGLE/!$OMP END SINGLE指令对中的一系列指令后，可将一个私有变量的值传递给其他进程。

```
!$OMP SINGLE
read(1,*) a
!$OMP END SINGLE COPYPRIVATE(a)
```

进程获取a的值后，使用COPYPRIVATE 子句更新所有其他进程里对应变量的备份。COPYPRIVATE对列表中变量的影响，出现在执行!$OMP SINGLE/!$OMP END SINGLE指令

对中内容之后，任意线程离开隐含阻塞之前。

OpenMP规范不允许在同一!$OMP END SINGLE指令中同时使用COPYPRIVATE 和NOWAIT ，对COPYPRIVATE列表中变量的更新总是在关闭指令以后。COPYPRIVATE子句只能与!$OMP END SINGLE 指令对一起使用。

3.1.8 REDUCTION(operator:list)

如果存在共享变量，需要确保同时仅有一个线程访问或者更新其值，否者将出现不可预料的结果：

```
!$OMP DO
do i = 1, 1000
a=a+i
enddo
!$OMP END DO
```

期望值为循环变量i的总和，但实际可能并不如此。子句REDUCTION可解决这个问题，因为同时只有一个线程更新a的值，确保了结果的正确性。一定程度上，它具有!$OMP ATOMIC / !$OMP CRITICAL指令对的功能，但执行方式不同。上例可改为：

```
!$OMP DO REDUCTION(+:a)
do i = 1, 1000
a=a+i
enddo
!$OMP END DO
```

REDUCTION子句的语法——在列表中变量前有一操作符，操作符与变量同时出现，使用了受保护的写进程机制。

受保护的写进程机制定义如下：为每个线程创建列表中所有变量的私有备份，如同使用PRIVATE子句；私有备份的初始化见表3.1 。在REDUCTION子句的结尾，用指定的操作符连接初始值和各个线程的终值，然后更新共享变量。

Operator/Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	smallest representable number
MIN	largest representable number
IAND	all bits on
IOR	0
IEOR	0

Table 3.1: Initialization rules for variables included in `REDUCTION` clauses.

通过定义保护机制，使原变量的写进程执行次数最少，达到保护的目的。执行次数等于线程数，与具体的计算无关。由于需要创建备份和初始化，这将产生一定的计算时间冗余。

原始的共享变量只在REDUCTION的结尾处更新，在与各个线程的隐含同步相遇之前，其值是不确定的。如不存在这种隐式同步，REDUCTION类变量的值在遇到一个同步之后才能确定。

OpenMP implementation 不保证REDUCTION类变量从一个区域传递到另一个区域后完全不变，因为REDUCTION类变量的中间值的顺序是随机的。REDUCTION 子句只用于下列含有变量和操作符的情况：

x = x operator expr

x = intrinsic_procedure(x, expr_list)

仅下列操作符和内部过程有效：

operator = +, *, -, .AND., .OR., .EQV. or .NEQV.

intrincic_procedure = MAX, MIN, IAND, IOR or IEOR

变量x必须为标量，同时为内部数据类型。X也可以是标量数组的入口，数组不能是延迟形状或假定大小。如果x是很大的数组，创建和初始化的冗余度就很高。

显然，表达式expr必须为标量表达式，且不能用到x的值，因为只有到REDUCTION process的结尾才能知道量级大小。尽管内部过程看似不满足条件，但因具associative nature的过程也是被允许的，所以内部过程也是可以的。

在开始指令处可以指定任意数量的REDUCTION子句，但同一变量只能出现一次。

3.2 Other clauses

除了上述数据作用域属性子句，下面将介绍另一些可用子句

3.2.1 IF(scalar logical expression)

并行区的部分代码可能并不需要执行，例如当创建和关闭并行区的时间超过并行计算的收益时。下面的代码：

```
!$OMP PARALLEL IF(N > 1000)
!$OMP DO
doi=1,N
...
enddo
!$OMP END DO
!$OMP END PARALLEL
```

只有当迭代次数大余1000时才执行并行，否则串行运行。

3.2.2 NUM_THREADS(scalar integer expression)

特定并行区需要使用固定数量的线程，如!$OMP SECTIONS指令中的任务。下例使用了4个线程：

```
!$OMP PARALLEL NUM_THREADS(4)
```

在并行区开头使用NUM THREADS子句，将重写由环境变量OMP NUM THREADS或运行时库subroutine OMP set num threads设置的值。指定的线程数仅影响当前并行区。

3.2.3 NOWAIT

一定情况下，并不需要所有线程在同时结束工作，因为接下来的操作不依赖于之前的结果。这种情况下，应该在指令对结尾处添加NOWAIT子句，以避免隐式同步。下例：

```
!$OMP PARALLEL
!$OMP DO
do i = 1, 1000
a=i
enddo
!$OMP END DO NOWAIT
!$OMP DO
do i = 1, 1000
b=i
enddo
```

```
!$OMP END DO
!$OMP END PARALLEL
```

如果第二个循环与前一个无关，在第一个并行循环结束的地方，并不需要等待所有线程结束。因此在!$OMP END DO的关闭指令添加NOWAIT子句，禁用最近的同步，提高计算速度。

但使用NOWAIT子句需要小心，错误的运用会导致不希望的结果，在开发阶段是发现不了的。同时需要知道，使用NOWAIT子句会抑制隐式数据同步。

3.2.4 SCHEDULE(t ype, chunk)

并行执行循环语句会将迭代分派到各个线程，简单的做法就是每个线程执行相同数量的迭代。但这并不总是最好的方法，因为有时候迭代的计算用时并不等于所有迭代用时之和因此有了不同的分派迭代的方法。这里的子句允许程序员为每个循环指定scheduling：

```
!$OMP DO SCHEDULE(type, chunk )
```

任务安排子句SCHEDULE包含两个参数：1、指定分派方式；2、可选，指定分派给每个线程的工作量，与分派方法相关。存在4个不同的选项：

1、STATIC： 按照线程号顺序，将由循环迭代空间创建的工作片段分派给线程。分派工作在循环的开始处进行，整个执行过程中保持不变。

默认情况下，工作片段数量等于线程数，每个片段的工作量大致相等。工作片段的总和等于总的工作量。

如果指定了可选参数，工作片段的工作量将由参数chunk指定。为了正确地分派工作，其中一个工作片段可以不等于chunk值。分配好的工作片段将派给每个线程。下面的例子具有3个线程：

```
!$OMP DO SCHEDULE(STATIC, chunk )
do i = 1, 600
...
enddo
!$OMP END DO
```

改变chunk的值，600次迭代将以不同方式分派给每个线程（图3.6）。图中方块代表工作片段，里面的数字代表线程号。

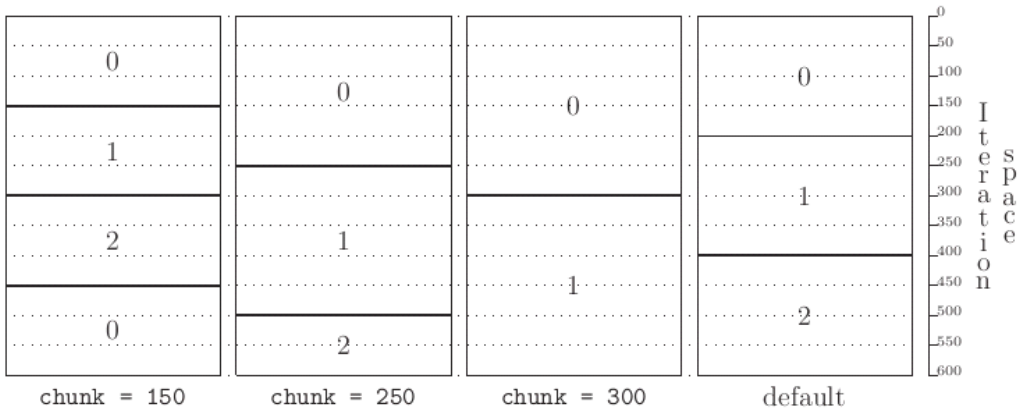

Figure 3.6: Graphical representation of the example explaining the general working principle of the SCHEDULE(STATIC,*chunk*) clause for different values of the optional parameter *chunk*.

（1）chunk = 150 ，需要4个工作块来完成迭代。因为总的迭代次数为chunk的4倍，因此各个工作块的大小精确相等。用循环将所有工作块分派给已存在的线程。线程0工作时，其他线程处于闲置状态，因此这种目标代码的效率不高。

（2）chunk = 250 ，需要3个工作块来完成迭代，工作块数量等于线程数，这样比较高效。因为总的迭代次数不是chunk的倍数，因此各个工作块的大小不都相等。代码的执行效率高于第一种情况，因为闲置时间更少。

（3）chunk=300 ，工作块少于线程数，意味着线程2在这个循环中并不工作。执行效率与（1）相等，比（2）差。

（4）chunk=default ，不指定chunk值，OpenMP创建等于线程数的工作块数量，每个工作块的大小相等。通常，如果每次迭代的计算时间一样，这种情况的执行速度最快。前面的3个例子也是基于这个假设。

2、DYNAMIC：前面的安排方法存在分配不均的问题，很难估计最佳的分配方式。解决方法就是动态地为每个工作块进行分配：当线程完成其工作后，为其分配新的工作。这种分配方式非常精确，也就是使用它的原因。

如果指定了SCHEDULE(DYNAMIC, chunk) ，迭代将按照chunk大小拆分为工作块。如果不指定chunk值，默认chunk=1 。每个线程执行一次迭代，完成后在分配一次迭代，知道所有迭代均被执行。

举例说明其工作原理。图3.7的顶部有12个工作块，长度代表需要的计算时间。将所有工作分给3个线程，最终的执行模式见图3.7底部。显而易见，每个线程执行完上一个任务后会得到一个新的任务。

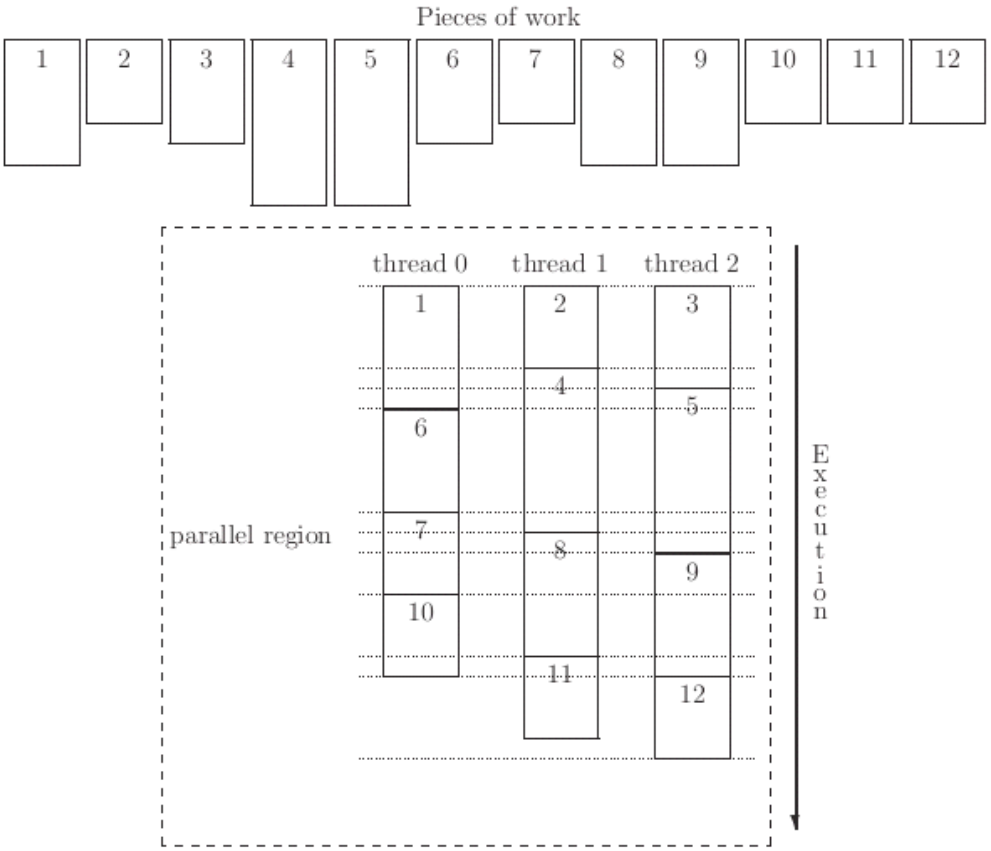

Figure 3.7: Graphical representation of the example explaining the working principle of the SCHEDULE(DYNAMIC,*chunk*) clause.

3、GUIDED：与静态方法相比，动态方法提高了执行能力和效率，但在处理和分配工作块的时候产生了计算时间的冗余，每个工作块越大，冗余越少。

另一种动态方法：每个块的工作量逐步减少，线程获得的相关工作量也逐渐减少。减少的规则是按指数递减，后一个工作块只有前一个的一半。

可选参数chunk指定了工作块的最小迭代次数，但为了包含所有迭代，最后一个工作块的迭代次数可能少于chunk 。工作块的最大尺寸和具有相等尺寸的工作块数量由OpenMP implementation决定。

图3.8列出了最终的工作块，相应的循环如下：

```
!$OMP DO SCHEDULE(GUIDED,256)
do i = 1, 10230
...
enddo
!$OMP END DO
```

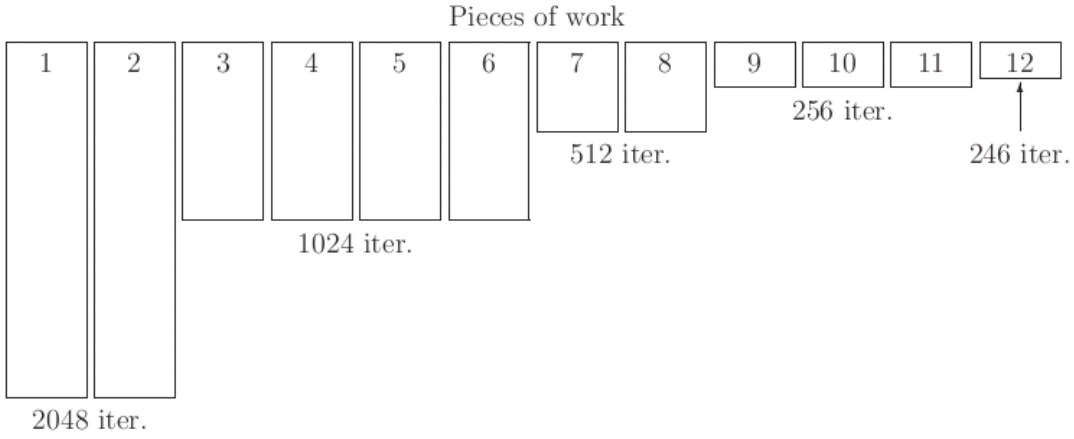

Figure 3.8: Graphical representation of the effect of the SCHEDULE(GUIDED,*chunk*) clause on a do-loop with 10230 iterations and with *chunk = 256*.

可以看到，最后一个工作块小于chunk值（256）。为了包含所有的迭代（10230），这是必要的。工作块一旦建立，分配方式与DYNAMIC .一直。

3、RUNTIME：前面的3种方法在编译源代码时就固定了。有时需要在运行时修改分配方式，则可指定为RUNTIME方式：

!$OMP DO SCHEDULE(RUNTIME)

将用到环境变量OMP SCHEDULE来指定分配方式，后面将讲述环境变量。

3.2.5 ORDERED

当循环中有需要顺序执行的语句时，如下：

```
do i = 1, 1000
A(i) = 2 * A(i-1)
End do
```

!$OMP ORDERED /!$OMP END ORDERED指令对用于指定顺序区。在!$OMP DO打开指令后强制添加ORDERED，告诉编译器存在顺序区：

```
!$OMP DO ORDERED
do i = 1, 1000
!$OMP ORDERED
A(i) = 2 * A(i-1)
!$OMP END ORDERED
enddo
!$OMP END DO
```

第四章 The OpenMP run-time library

前面章节的一些示例中提到了运行时库，本章的目的就是介绍OpenMP Fortran API run-time library，解释说明其中的子程序和函数，以及其功能和限制条件。

OpenMP运行时库可看做并行执行环境下的控制和查询工具，程序员可将其运用于自己的程序中。运行时库是一系列具有特定接口界面的外部过程。接口界面在Fortran95的模块omp_lib中说明。接下来将介绍不同的程序和接口界面：介绍不同OpenMP-implementation中变量的精度和size（kind值），其本质是一样的。

4.1 Execution environment routines

OpenMP Fortran API run-time library中的以下函数和子程序允许修改运行时环境的约束条件，控制并行运算程序。

4.1.1 OMP_set_num_threads

设置紧接其后的并行区的线程数，因此只能在并行区之外调用。接口界面声明如下：

```
subroutine OMP_set_num_threads(number_of_threads)
integer(kind = OMP_integer_kind), intent(in) :: number_of_threads
end subroutine OMP_set_num_threads
```

Number_of_threads为需要设置的线程数量。如果动态调节不可用，在下一个OMP_set_num_threads子程序之前的所有并行区的线程数都将设为Number_of_threads ；如果动态调节可用，则为允许的最大线程数。这种方法设置的线程数优先级高于环境变量 OMP NUM THREADS 。

4.1.2 OMP_get_num_threads

函数返回正在执行调用函数的并行区的线程的线程数，因此用于并行区之内。即使是在串行区域或者串行的嵌套并行区，也将返回正确的结果（1）。接口界面如下：

```
function OMP_get_num_threads()
integer(kind = OMP_integer_kind) :: OMP_get_num_threads
end function OMP_get_num_threads
```

4.1.3 OMP_get_max_threads

函数返回程序可使用的最大线程数。如果线程动态调整可用，它的返回值可能与 OMP get num threads 不同，否则一致。接口界面信息与OMP get num threads一样：

```
function OMP_get_max_threads()
integer(kind = OMP_integer_kind) :: OMP_get_max_threads
end function OMP_get_max_threads
```

返回值为允许的最大线程数。返回值与当前并行区使用的实际线程数无关，在并行区或者串行区调用的结果是一样的。

4.1.4 OMP_get_thread_num

返回当前线程的线程号，取值范围0到OMP_get_num_threads()-1 。接口如下：

```
function OMP_get_thread_num()
integer(kind = OMP_integer_kind) :: OMP_get_thread_num
end function OMP_get_thread_num
```

如果在在串行区域或者串行的嵌套并行区内调用此函数，则返回值为0 ，即为主线程号。

4.1.5 OMP_get_num_procs

返回可用处理器核数：

```
function OMP_get_num_procs()
integer(kind = OMP_integer_kind) :: OMP_get_num_procs
end function OMP_get_num_procs
```

4.1.6 OMP_in_parallel

标志特定区域是否被并行执行。为达到这个目的，它将监视所有包含OMP_in_parallel的并行区，如果至少有一个被并行执行，则返回.true. ； 否则返回false 。接口声明如下：

```
function OMP_in_parallel()
logical(kind = OMP_logical_kind) :: OMP_in_parallel
end function OMP_in_parallel
```

如果返回值为true ，将从并行区的动态范围内的任意点返回，即使是nested serialized parallel regions（被认为是串行的）。

4.1.7 OMP_set_dynamic

启用或禁用并行区线程数动态调节。

```
subroutine OMP_set_dynamic(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_dynamic
```

enable=.true. ，表示用于随后并行区的线程数由运行时环境自动调整，最大程度使用SMP machine ；反之，禁用动态调整。

由环境变量OMP NUM THREADS或者运行时库子程序subroutine OMP set num threads设置的最大线程数，可通过运行时环境赋值给程序。

由OMP set dynamic和OMP DYNAMIC设置的环境变量，前者优先级较高。

动态线程调整默认是否启用取决于OpenMP-implementation，因此如果程序需要指定的线程来运行，应该明确禁用动态线程调整。当然implementation不必要支持动态调整，但出于可移植性的目的，它至少应该支持这个接口界面。

4.1.8 OMP_get_dynamic

返回动态线程调整的状态：可用（true）或不可用（false）。如果OpenMP-implementation不支持动态调整，则返回值总为false 。

```
function OMP_get_dynamic()
logical(kind = OMP_logical_kind) :: OMP_get_dynamic
end function OMP_get_dynamic
```

4.1.9 OMP_set_nested

启用(true)或禁用(false)嵌套并行

```
subroutine OMP_set_nested(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_nested
```

默认嵌套并行区为串行执行。用于执行嵌套并行区的线程数由 OpenMP-implementation

决定。当然，开启嵌套并行后，支持OpenMP的设备也可以串行执行嵌套并行区。

调用这个子程序可重写由环境变量OMP NESTED为紧接着的嵌套并行区指定的设置。

4.1.10 OMP_get_nested

函数返回嵌套并行机制的状态：可用(true)或不可用(false) 。如果OpenMP-implementation不支持嵌套并行，返回值总为false 。

```
function OMP_get_nested()
logical(kind = OMP_logical_kind) :: OMP_get_nested
end function OMP_get_nested
```

4.2 Lock routines

第二组函数/子程序用于处理LOCKS。LOCKS代表与前面的!$OMP ATOMIC 或 !$OMP CRITICAL不同的另一种同步机制。

锁可被看作一个能够设置和取消的标志。每个线程观察标志的状态，根据不同的状态按不同方法执行。当一个线程发现一个为设置的标志时，将设置这个标志，以获取一定的特权，并警告后续的线程。设置锁的线程为ownership of the lock。一旦线程不需要所获取的特权，则取消这个锁：releases the ownership 。

以前介绍的同步指令和lock的主要不同在于后者不是必须要遵守lock，即给定线程如果不关心锁的状态就不会被其影响。尽管这看似是一个无用的特性，但实际非常有用和灵活。

例如，存在一个共享矩阵A(1:100,1:100)和具有4个线程的并行区域，每个线程都要修改矩阵的所有元素，但是顺序无所谓。为避免race conditions，可用下面的例子：

```
!$OMP PARALLEL SHARED(A)
!$OMP CRITICAL
... !works on the matrix A
!$OMP END CRITICAL
!$OMP END PARALLEL
```

此例中矩阵A所有元素被传给一个线程，其余的线程处于等待状态。另一种方法，将矩阵分为4个工作块A1 = A(1:50,1:50) , A2 = A(51:100,1:50) , A3 = A(1:50,51:100) and A4= A(51:100,51:100)，每个线程都将修改4个块中的值。每个线程获得一个工作块，执行完毕后交换另一个工作块。这样既可以避免race conditions，又能让4个线程都工作。

可以通过使用锁来达到这种拆分的目的，而不需要使用OpenMP指令。使用锁和运行时库程序的一般顺序为：

（1）初始化锁和相应的锁变量（在程序内使用识别机制）

（2）一个线程获得某个锁的所有权

（3）锁的状态影响所有线程

（4）一旦完成工作将释放所有权，其他线程可捕获锁

（5）锁不再被需要，释放锁和锁变量。

简单锁：如果被锁住就不会其他线程锁住

嵌套锁：解锁之前可被同一线程多次锁住。包含一个嵌套计数器（nesting counter），记录嵌套锁被锁住和释放的次数。

每个与锁一起工作的运行时库程序都有两个版本：一个用于简单锁，另一个用于嵌套锁。两者不得交叉使用。

下一节将介绍执行锁的运行时程序。

4.2.1 OMP_init_lock and OMP_init_nest_lock

子程序初始化一个锁：分别为简单锁和嵌套锁。在使用下面任何一个子程序之前，必须调用其中之一初始化一个锁。接口如下：

```
subroutine OMP_init_lock(svar)
integer(kind = OMP_lock_kind), intent(out) :: svar
end subroutine OMP_init_lock
subroutine OMP_init_nest_lock(nvar)
integer(kind = OMP_nest_lock_kind), intent(out) :: nvar
end subroutine OMP_init_nest_lock
```

传递的变量称为与锁相关的锁变量。初始化锁变量后，可用其在后续的调用中代替锁。如果锁是嵌套锁，嵌套计数器置零。

不允许使用一个关联了锁的锁变量来调用上述子程序，因为没有指明OpenMP implementation如何做出反应。

4.2.2 OMP_set_lock and OMP_set_nest_lock

如果线程需要一个锁，可调用这子程序。如果锁未被其他线程使用，将建立二者的所有关系；如果锁被其他线程所有，将等待其他线程释放这个锁。子程序接口如下：

```
subroutine OMP_set_lock(svar)
integer(kind = OMP_lock_kind), intent(inout) :: svar
end subroutine OMP_set_lock
subroutine OMP_set_nest_lock(nvar)
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end subroutine OMP_set_nest_lock
```

未被锁住的简单锁是有效的，未被锁住或被当前线程锁住的嵌套锁也是有效的。后一种情况，嵌套锁变量的计数器将加一。

4.2.3 OMP_unset_lock and OMP_unset_nest_lock

子程序将释放一个所有关系。接口如下：

```
subroutine OMP_unset_lock(svar)
integer(kind = OMP_lock_kind), intent(inout) :: svar
end subroutine OMP_unset_lock
subroutine OMP_unset_nest_lock(nvar)
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end subroutine OMP_unset_nest_lock
```

前一种情况，通过锁变量释放调用线程和简单锁的所有关系；后一种情况，使嵌套计数器减一，仅当计数器等于零时释放锁。

调用这个程序之前，应该确定调用程序是否拥有一个锁；如果没有，则后续行为未指明。

4.2.4 OMP_test_lock and OMP_test_nest_lock

函数试图以与OMP_set_lock和OMP_set_nest_lock一样的方式建立一个锁，但并不强制调用线程等待直到指定锁可用。

```
function OMP_test_lock(svar)
logical(kind = OMP_logical_kind) :: OMP_test_lock
integer(kind = OMP_lock_kind), intent(inout) :: svar
end function OMP_test_lock
function OMP_test_nest_lock(nvar)
integer(kind = OMP_integer_kind) :: OMP_test_nest_lock
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end function OMP_test_nest_lock
```

设置成功后，函数OMP_test_lock返回true，否则返回false。函数OMP_test_nest_lock设置成功后返回新的计数器值，否则返回零。

4.2.5 OMP_destroy_lock and OMP_destroy_nest_lock

如果不再需要一个锁，则释放相关的锁变量，这就是当前子程序的作用。

```
subroutine OMP_destroy_lock(svar)
integer(kind = OMP_lock_kind), intent(inout) :: svar
end subroutine OMP_destroy_lock
subroutine OMP_destroy_nest_lock(nvar)
```

```
integer(kind = OMP_nest_lock_kind), intent(inout) :: nvar
end subroutine OMP_destroy_nest_lock
```

4.2.6 示例

以下示例将演示子程序和函数的作用

```
program Main
use omp_lib
implicit none
integer(kind = OMP_lock_kind) :: lck
integer(kind = OMP_integer_kind) :: ID
call OMP_init_lock(lck)
!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
ID = OMP_get_thread_num()
call OMP_set_lock(lck)
write(*,*) "My thread is ", ID
call OMP_unset_lock(lck)
!$OMP END PARALLEL
call OMP_destroy_lock(lck)
end program Main
```

第一件事就是用OMP_init_lock将一个简单锁与锁变量lck绑定。在并行区，线程调用OMP_set_lock 建立与锁lck的所有关系。Write语句只被拥有锁的线程执行。工作完成后，使用OMP_unset_lock 释放锁，使其他线程可以执行write语句。并行区完成后，不再需要锁，调用OMP_destroy_lock 销毁锁。

也可用OpenMP指令!$OMP CRITICAL执行上述操作：

```
program Main
use omp_lib
implicit none
integer(kind = OMP_integer_kind) :: ID
!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
ID = OMP_get_thread_num()
!$OMP CRITICAL
write(*,*) "My thread is ", ID
!$OMP END CRITICAL
!$OMP END PARALLEL
end program Main
```

如果指令可以做到相同的功能，还需要运行时锁子程序吗？前面提到的关于矩阵A的例子，也有另外的写法，但OpenMP指令不能模仿锁程序。

```
program Main
use omp_lib
implicit none
integer(kind = OMP_lock_kind) :: lck
integer(kind = OMP_integer_kind) :: ID
call OMP_init_lock(lck)
!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
ID = OMP_get_thread_num()
do while(.not.OMP_test_lock(lck))
... !work to do while the thread is waiting to get owner of the lock
enddo
... !work to do as the owner of the lock
!$OMP END PARALLEL
call OMP_destroy_lock(lck)
end program Main
```

本例中，其他不拥有锁的线程也未闲置，而是在执行其他任务，这是OpenMP指令无法实现的。对于特定问题，比如同步所需的时间花费很大，这一点很有用。

上面两个例子使用了简单锁。当一个使用锁的子程序在主程序的不同位置被调用时，嵌套锁很有用。此时不需要考虑是谁在调用它。

```
module data_types
use omp_lib, only: OMP_nest_lock_kind
implicit none
type number
integer :: n
integer(OMP_nest_lock_kind) :: lck
end type number
end module data_types
!---------------------------------------------------------!
program Main
use omp_lib
use data_types
implicit none
type(number) :: x
x%n = 0
call OMP_init_lock(x%lck)
!$OMP PARALLEL SECTIONS SHARED(x)
!$OMP SECTION
call add(x,20)
!$OMP SECTION
call substract(x,10)
!$OMP END PARALLEL
call OMP_destroy_lock(lck)
```

```
end program Main
!-------------------------------------------------------!
subroutine add(x,d)
use omp_lib
use data_types
implicit none
type(number) :: x
integer :: d
call OMP_set_nest_lock(x%lck)
x%n = x%n + d
call OMP_unset_nest_lock(x%lck)
end subroutine add
!-------------------------------------------------------!
subroutine substract(x,d)
use omp_lib
use data_types
implicit none
type(number) :: x
integer :: d
call OMP_set_nest_lock(x%lck)
call add(x,-d)
call OMP_unset_nest_lock(x%lck)
end subroutine substract
```

子程序add被每个线程调用一次，每次调用，变量x都被正确更新，因为加锁之后同时只有一个线程访问变量x。Substract不必要知道add的内部是如何执行的，所以substract可以影响锁。

4.3 Timing routines

OpenMP Fortran API run-time library中的最后一组程序用作评价工具，测量程序执行的绝对耗时。

4.3.1 OMP_get_wtime

函数返回以秒为单位的绝对时间。调用两次函数就可计算出它们之间的代码的执行时间：

```
start = OMP_get_wtime()
... !work to be timed
end = OMP_get_wtime()
```

```
time = end - start
```

接口如下：

```
function OMP_get_wtime()
real(8) :: OMP_get_wtime
end function OMP_get_wtime
```

返回的时间是”per-thread times”，并不需要保持所有线程的全局一致性。

4.3.2 OMP get wtick

函数返回连续计时的秒数。

```
function OMP_get_wtick()
real(8) :: OMP_get_wtick
end function OMP_get_wtick
```

4.4 The Fortran 90 module omp_lib

为了正确调用前面提到的函数和子程序，它们都有已定义的接口界面。通过被OpenMP implementation支持的Fortran 90模块 omp_lib来实现这一功能。尽管如此，至少还应包含一下代码（很多Fortran 90编译器并不支持这些代码，因为不能在接口界面里声明parameter。故需要手动将其改为常数）：

```
module omp_lib
implicit none
!Defines standard variable precisions of the OpenMP-implementation
integer, parameter :: OMP_integer_kind = 4
integer, parameter :: OMP_logical_kind = 4
integer, parameter :: OMP_lock_kind = 8
integer, parameter :: OMP_nest_lock_kind = 8
!Defines the OpenMP version: OpenMP Fortran API v2.0
integer, parameter :: openmp_version = 200011
!Gives the explicit interface for each routine of the run-time library
interface
subroutine OMP_set_num_threads(number_of_threads)
integer(kind = OMP_integer_kind), intent(in) :: number_of_threads
end subroutine OMP_set_num_threads
function OMP_get_num_threads()
integer(kind = OMP_integer_kind) :: OMP_get_num_threads
end function OMP_get_num_threads
```

```
function OMP_get_max_threads()
integer(kind = OMP_integer_kind) :: OMP_get_max_threads
end function OMP_get_max_threads
function OMP_get_thread_num()
integer(kind = OMP_integer_kind) :: OMP_get_thread_num
end function OMP_get_thread_num
function OMP_get_num_procs()
integer(kind = OMP_integer_kind) :: OMP_get_num_procs
end function OMP_get_num_procs
function OMP_in_parallel()
logical(kind = OMP_logical_kind) :: OMP_in_parallel
end function OMP_in_parallel
subroutine OMP_set_dynamic(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_dynamic
function OMP_get_dynamic()
logical(kind = OMP_logical_kind) :: OMP_get_dynamic
end function OMP_get_dynamic
subroutine OMP_set_nested(enable)
logical(kind = OMP_logical_kind), intent(in) :: enable
end subroutine OMP_set_nested
function OMP_get_nested()
logical(kind = OMP_logical_kind) :: OMP_get_nested
end function OMP_get_nested
subroutine OMP_init_lock(var)
integer(kind = OMP_lock_kind), intent(out) :: var
end subroutine OMP_init_lock
subroutine OMP_init_nest_lock(var)
integer(kind = OMP_nest_lock_kind), intent(out) :: var
end subroutine OMP_init_nest_lock
subroutine OMP_destroy_lock(var)
integer(kind = OMP_lock_kind), intent(inout) :: var
end subroutine OMP_destroy_lock
subroutine OMP_destroy_nest_lock(var)
integer(kind = OMP_nest_lock_kind), intent(inout) :: var
end subroutine OMP_destroy_nest_lock
subroutine OMP_set_lock(var)
integer(kind = OMP_lock_kind), intent(inout) :: var
end subroutine OMP_set_lock
subroutine OMP_set_nest_lock(var)
integer(kind = OMP_nest_lock_kind), intent(inout) :: var
end subroutine OMP_set_nest_lock
subroutine OMP_unset_lock(var)
integer(kind = OMP_lock_kind), intent(inout) :: var
```

```
end subroutine OMP_unset_lock
subroutine OMP_unset_nest_lock(var)
integer(kind = OMP_nest_lock_kind), intent(inout) :: var
end subroutine OMP_unset_nest_lock
function OMP_test_lock(var)
logical(kind = OMP_logical_kind) :: OMP_test_lock
integer(kind = OMP_lock_kind), intent(inout) :: var
end function OMP_test_lock
function OMP_test_nest_lock(var)
integer(kind = OMP_integer_kind) :: OMP_test_nest_lock
integer(kind = OMP_nest_lock_kind), intent(inout) :: var
end function OMP_test_nest_lock
function OMP_get_wtime()
real(8) :: OMP_get_wtime
end function OMP_get_wtime
function OMP_get_wtick()
real(8) :: OMP_get_wtick
end function OMP_get_wtick
end interface
end module omp_lib
```

第五章 环境变量

OpenMP并行程序使用的并行环境被环境变量控制，或由平台指定。这些变量可由操作系统的命令行或调用OpenMP Fortran API run-time library中的子程序来设置。由命令行设置环境变量的方式由操作系统决定，例如在Linux/Unix系统中使用csh壳（shell），setenv命令来执行：

```
> setenv OMP_NUM_THREADS 4
```

如果使用了sh、ksh或bash壳，上例改为：

```
> OMP_NUM_THREADS=4
> export OMP_NUM_THREADS
```

以上两种情况都可以用echo命令查看环境变量

```
> echo $OMP_NUM_THREADS
4
>
```

在微软的NT/2000/XP系统中，可在控制面板（用户环境变量）中指定或通过在AUTOEXEC.BAT 文件(AUTOEXEC.BAT 环境变量)中添加相应行实现。两种方法的语法都很简单：

```
OMP_NUM_THREADS = 4
```

环境变量的标识符必须全为大写，否则其值不被接受并可能产生空格。
本章将介绍环境变量的可能值及其在并行程序里的影响。

5.1 OMP_NUM_THREADS

指定OpenMP并行程序中并行区的线程数。理论上可以是任何大于零的整数，即使多于处理器个数，但这种情况下并行计算很慢。如果动态线程调整可用，则代表允许的最大线程数。

在Linux/Unix系统下，使用csh和bash壳的例子分别为：

```
> setenv OMP_NUM_THREADS 16
***************
> OMP_NUM_THREADS=16
> export OMP_NUM_THREADS
```

5.2 OMP_SCHEDULE

影响指令!$OMP DO和!$OMP PARALLEL DO的工作方式，如果设置为RUNTIME：

!$OMP DO SCHEDULE(RUNTIME)

通过设置环境变量为任意表类型和可选参数chunk值，可在程序运行中设置表类型和chunk值。可能的表类型为STATIC, DYNAMIC、GUIDED：相关说明见!$OMP DO指令。

默认值由OpenMP implementation 指定，请详细阅读相关文档。

如果为设置可选参数chunk，则默认为1 （STATIC除外，为迭代总数除以线程数）。

在Linux/Unix系统下，使用csh和bash壳的例子分别为：

```
> setenv OMP_SCHEDULE "GUIDED,4"
**************************
> OMP_SCHEDULE=dynamic
> export OMP_SCHEDULE
```

5.3 OMP_DYNAMIC

大型SMP机上，多个程序共享可用机器资源的情况很常见。这种情况，在并行区使用固定线程的程序可能导致资源使用效率低下。为最大化使用资源，SMP机的运行时环境能够动态调整线程数。设置环境变量OMP_DYNAMIC为true，就能达到这个目的。

环境变量OMP_DYNAMIC的默认值由OpenMP implementation确定。

在Linux/Unix系统下，使用csh和bash壳的例子分别为：

```
> setenv OMP_DYNAMIC TRUE
**********************
> OMP_DYNAMIC=FALSE
> export OMP_DYNAMIC
```

5.4 OMP_NESTED

当存在嵌套OpenMP指令时，指定运行时环境必须做的事。如果为true ，则嵌套并行可用，并创建一组新的线程执行嵌套的指令，结果如图1.2的树状结构，每个嵌套使用两个线程。

如果值为false ，嵌套指令为串行，即只有一个线程执行其中的代码。

在Linux/Unix系统下，使用csh和bash壳的例子分别为：

> setenv OMP_NESTED TRUE

> OMP_NESTED=FALSE
> export OMP_NESTED

End