5.2.1 循环并行化

循环并行化编译制导语句的格式：
循环并行化是使用 OpenMP 来并行化程序的最重要的部分，它是并行区域编程的一个特例。由于大量的科学计算程序将很大一部分时间用在处理循环计算上，对于循环进行并行化处理对这一部分应用程序非常关键，因此循环并行化是 OpenMP 应用程序中是一个相对独立且非常重要的组成部分。在 C/C++语言中，循环并行化语句的编译制导语句格式如下：

代码 5.2 循环并行化语句的编译制导语句格式

```
#pragma omp parallel for [clause[clause…]]
    for (index = first ; test_expression ; increment_expr){
        body of the loop;
    }
或者
#pragma omp parallel [clause[clause]]
{
#pragma omp for [clause[clause]]
for (index = first; test_expression; increment_expr){
            body of the loop;
}
    }
```

这两个版本的效果基本相同。只是，如果并行的线程需要在循环的开始、或结束时作些工作的话，就只能用 parallel 与 for 子句分离的版本。

这个编译制导语句中的 parallel 关键字将紧跟的程序块扩展为若干完全等同的并行区域，每个线程拥有完全相同的并行区域；而关键字 for 则将循环中的工作分配到线程组中，线程组中的每一个线程完成循环中的一部分内容。编译制导语句的功能区域一直延伸到 for 循环、或用大括号{}包围起来的程序块的结束。编译制导语句后面的字句（clause）用来控制编译制导语句的具体行为，在后面将通过例子来详细讲解子句的构成以及相关子句的语法状况。

循环并行化语句的限制
并不是所有的循环语句都能够在其前面加上#pragma omp parallel 来实现并行化，在 OpenMP 2.5 规范中，OpenMP 对于可以以多线程执行的循环有以下约束：

• 循环语句中的循环变量必须是有符号整型，如果是无符号整型，就无法使用。注意，在未来的 OpenMP 3.0 规范中，这个约束可能被取消。

• 循环语句中的比较操作必须是这样的形式：loop_variable <, <=, >或>= loop_invariant_integer。

• 循环语句中的第三个表达式（for 循环的循环步长）必须是整数加或整数减，加减的数值

必须是一个循环不变量（loop invariant value）

• 如果比较操作是<或<=，那么循环变量的值在每次迭代时都必须增加；相反地，如果比较操作是>或>=，那么循环变量的值在每次迭代时都必须减少。

•循环必须是单入口、单出口的，也就是说循环内部不允许有能够到达循环之外的跳转语句，也不允许有外部的跳转语句到达循环内部。在这里，exit 语句是一个特例，因为它将中止整个程序的执行。如果使用了 goto 或 break 语句，那么它们的跳转范围必须在循环之内，不能跳出循环。异常处理也是如此，所有的异常都必须在循环内部处理。

虽然这些约束看上去有些多，但大多数循环都能够非常容易地被重写成符合约束条件地形式。只有满足上述约束条件，编译器才能通过 OpenMP 将循环并行化。然而，虽然编译器能够完成循环的并行化，仍然需要程序员来保证循环功能的正确。

简单循环并行化
我们首先看一个简单循环的并行化过程，将两个向量相加，并将计算的结果保存到第三个向量中，向量的维数为 n。向量相加即向量的各个分量分别相加。

```
for（int i=0;i<n;i++）
    z[i]=x[i]+y[i];
```

显然向量相加的算法中，各个分量之间没有数据相关性，循环计算的过程也没有循环依赖性，即某一次循环的结果不依赖于其它次循环的结果。而如下的例子当中就存在循环依赖性。

```
for（int i=0;i<n;i++）
    z[i]=z[i-1]+x[i]+y[i]
```

可以看出，第 i 次的循环依赖于 i-1 次的循环的结果，对于这样有依赖性的循环进行并行化必须考虑循环依赖性。

关于数据相关的概念，如果语句 S2 与语句 S1 存在数据相关，那么必然存在以下两种情况之一：

• S1 在循环的一次迭代中访问存储单元 L，而 S2 在随后的一次迭代中访问同一存储单元。

• S1 和 S2 在同一循环迭代中访问同一存储单元 L，但 S1 的执行在 S2 之前。

第一种情况称为循环迭代相关（loop-carried dependence），相关的存在是因为循环有多次迭代。而第二种情况是一种非循环迭代相关（loop-independent dependence），因为该相关的存在是由循环内代码的顺序决定的。

下面来看这样一个示例：d=1，n=99，迭代 k 中 S1 对存储单元 x[k]进行写操作，而迭代 k+1 中 S2 将读取该存储单元，这样就产生了一个循环迭代流相关。此外，迭代 k 中 S1 对存储单元 y[k-1]进行读操作，而迭代 k+1 中 S2 对其进行写操作，因此存在循环迭代反相关。在这

种情况下，如果插入一条 parallel for 编译制导令该循环以多线程执行，就将得到错误的结果。

```
// 这段代码是错误的。由于存在循环迭代相关，该代码无法正常执行

x[0] = 0;
y[0] = 1;
#pragma omp parallel for private(k)
for (k = 1; k < 100; k++){
        x[k] = y[k-1] + 1; //S1
        y[k] = x[k-1] + 2; //S2
}
```

因为 OpenMP 编译制导是对编译器发出的命令，所以编译器会将该循环编译成多线程代码。但由于循环迭代相关的存在，多线程代码将不能成功执行。要解决此类问题，唯一的方法就是重写该循环，或选用另一个不包含循环迭代相关的算法。对于这个示例，可以预先确定 x[49]和 y[49]的初值，然后运用循环分块技术创建无循环迭代相关的循环 m。最后，插入 parallel for 并行化循环 m。通过这样的转换，原来的循环就可以在一个双核处理器系统上使用两个线程来执行了。

代码 5.3 使用分块技术将循环转换成等效的多线程代码

```
x[0] = 0;
y[0] = 1;
x[49] = 74; //根据等式 x(k) = x(k-2) + 3 计算得出
Computed according to equation x(k) = x(k-2) + 3
y[49] = 74 ; //根据等式 y(k) = y(k-2) + 3 计算得出
Computed according to equation y(k) =y(k-2) + 3

#pragma omp parallel for private(m, k)
for (m = 0; m < 2; m++){
        for (k = m*49 + 1; k < m*50 + 50; k++){
            x[k] = y[k-1] + 1; //S1
            y[k] = x[k-1] + 2; //S2
}
}
```

对于这个示例，除了使用 parallel for 编译制导，还可以使用 parallel sections 编译制导来并行化原本具有循环迭代相关的循环，以便能够在一个双核处理器系统上运行。

代码 5.4 使用 parallel sections 编译制导将循环转换成等效的多线程代码

```
#pragma omp parallel sections private(k)
{
#pragma omp section
        {
```

```
            x[0] = 0; y[0] = 1;
            for (k = 1; k < 49; k++){
                x[k] = y[k-1] + 1; //S1
                y[k] = x[k-1] + 2; //S2
}
}
#pragma omp section
{
            x[49] = 74; y[49] = 74;
            for (k = 50; k < 100; k++){
                x[k] = y[k-1] + 1; // S3
                y[k] = x[k-1] + 2; // S4
}
}
}
```

通过这个简单的示例，我们对一个存在循环迭代相关的循环进行了并行化，从中学习了一些行之有效的方法。有时候，在编写多线程代码时，为了充分利用双核或多核处理器，除了添加 OpenMP 编译制导以外，还需要进行简单的代码重构或变换。

循环并行化编译制导语句的子句

循环并行化子句可以包含一个或者多个子句来控制循环并行化的实际执行。有多个类型的子句可以用来控制循环并行化编译。最主要的子句是数据作用域子句。由于有多线程同时执行循环语句中的功能指令，这就涉及到数据的作用域问题。注意这里所说的作用域和 C/C++的数据作用域是不同的。这里的作用域用来控制某一个变量是否是在各个线程之间共享或者是某一个线程是私有的。数据的作用域子句用 shared 来表示一个变量是各个线程之间共享的，而用 private 来表示一个变量是每一个线程私有的，用 threadprivate 表示一个线程私有的全局变量。在 OpenMP 中，如果没有指定变量的作用域，则默认的变量作用域是共享的，这与 OpenMP 对应的共享内存空间编程模型是相互符合的。除了变量的作用域子句外，还有一些编译制导子句是用来控制线程的调度（schedule 子句），动态控制是否并行化（if 子句），进行同步的子句（ordered 子句）以及控制变量在串行部分与并行部分传递的子句（copyin 子句）。这些子句将在后面逐步介绍。

循环嵌套

在一个循环体内经常会包含另外一个循环体，循环产生了嵌套。在 OpenMP 中，我们可以将嵌套循环的任意一个循环体进行并行化。循环并行化编译制导语句可以加在任意一个循环之前，则对应的最近的循环语句被并行化，其它部分保持不变。因此，实际上并行化是作用于嵌套循环中的某一个循环，其它部分由执行到的线程负责执行。

代码 5.5 循环嵌套，并行化作用于外层循环

```
            int i;int j
            #pragma omp parallel for private（j）
            for（i=0;i<2;i++）
```

```
                for（j=6;j<10;j++）
                 printf（“i=%d j=%d\n”,i,j）;
```

执行结果：

```
i=0 j=6
i=1 j=6
i=0 j=7
i=1 j=7
i=0 j=8
i=1 j=8
i=1 j=9
i=0 j=9
```

代码5.6 循环嵌套，并行化作用于内层循环

```
            int i;int j;
            for（i=0;i<2;i++）
            #pragma omp parallel for
                for（j=6;j<10;j++）
                    printf（"i=%d j=%d\n",i,j）;
```

执行结果：：

```
i=0 j=6
i=0 j=8
i=0 j=9
i=0 j=7
i=1 j=6
i=1 j=8
i=1 j=7
i=1 j=9
```

在上述的程序中，程序 5.5 并行化作用于外层循环，程序 5.6 并行化作用于内层循环。在执行的过程中，并行执行的效果只与有作用的循环相关，在每一个并行执行线程的内部，程序继续按照串行执行。

控制数据的共享属性

OpenMP 程序在同一个共享内存空间上执行，由于可以任意使用这个共享内存空间上的变量进行线程间的数据传递，使得线程通信非常容易。一个线程可以写入一个变量而另外一个线程则可以读取这个变量来完成线程间的通信。

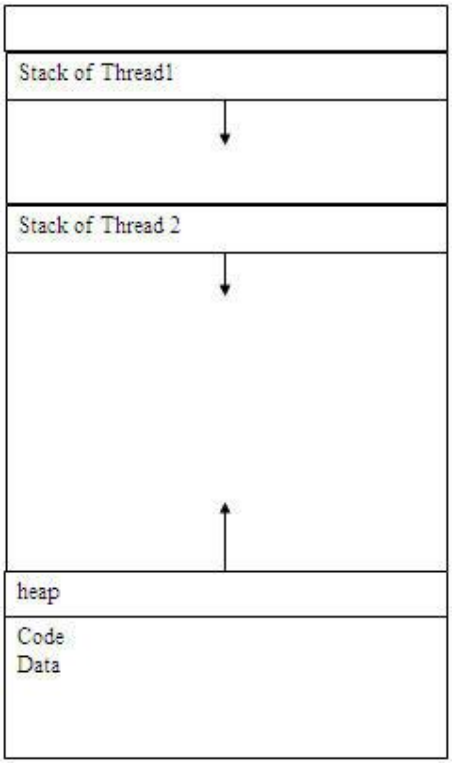

图 5．6 多线程应用程序的内存分布结构

上述图 5.6 给出了多线程应用程序的内存分布结构。可以看出，每一个线程的栈空间都是私有的，因此分配在栈上的数据都是线程私有的，例如函数调用时分配的自动变量等。全局变量以及程序代码都是全局共享的；而动态分配的堆空间也是共享的。另外，在 OpenMP 中通过 threadprivate 来明确指出的某一个数据结构属于线程范围的全局变量。除了共享变量之外，OpenMP 还允许线程保留自己的私有变量不能让其它线程访问到。每一个线程会建立变量的私有拷贝，虽然变量名是相同的，实际上在共享内存空间内部的位置是不同的。因此，控制变量的作用域就非常重要。为了清楚地讨论两种不同的作用域，将 C/C++语言层面的作用域（即变量的是否 static，global，file level 等）称为“语法作用域”。数据作用域子句用来确定数据的共享属性，有下面四个子句。

shared 用来指示一个变量的作用域是共享的。private 用来指示一个变量作用域是私有的。firstprivate 和 lastprivate 分别对私有的变量进行初始化的操作和最后终结的操作。firstprivate 将串行的变量值拷贝到同名的私有变量中，在每一个线程开始执行的时候初始化一次。而 lastprivate 则将并行执行中的最后一次循环的私有变量值拷贝的同名的串行变量中。default 语句用来改变变量的默认私有属性。

在使用作用域子句的时候要遵循如下的一些规则。1）作用域子句作用的变量是已经申明的有名变量；2）作用域子句在作用到类或者结构的时候，必须作用到类或者结构的整体，而不能只作用于类或者结构的一个部分；3）一个编译制导语句能够包含多个数据作用域子句，但是变量只能出现在一个作用域子句中，即变量不能既是共享的，又是私有的。4）在语法结构上，作用域子句只能作用在出现在编译制导语句起作用的语句变量部分。另外，可以将作用域子句作用在类的静态变量上，例如 #pragma omp parallel for shared

（ClassName::_staticVariable）。由于整个类别可能被标记为私有的，则对类的构造有一定的特殊要求。例如，私有变量在分配空间的时候会被默认调用构造函数，则相应的类要求有默认的构造来初始化类变量；类变量在串行部分和并行部分需要进行数据传递的时候，则需要类提供相应的拷贝构造函数等。

除了上述显式申明变量作用域外，OpenMP 对不同的语法作用域规定了相应的默认数据作用域。默认情况下，并行区中所有的变量都是共享的，但有三种例外情况。首先是在 parallel for 循环中，循环索引变量是私有的。其次，那些并行区中的局部变量是私有的。第三，所有在 private，firstprivate，lastprivate 或 reduction 子句中列出的变量都是私有的。私有化是通过为每个线程创建各个变量的独立副本来完成的。

代码 5.7 数据的共享

```
int gval=8;
void funcb（int * x, int *y, int z）
{
        static int sv;
        int u;
        u=（*y）*gval;
        *x=u+z;
}

void funca（int * a, int n）
{
        int i;
        int cc=9;
        #pragma omp parallel for
        for（i=0;i<n;i++）{
            int temp=cc;
            funcb（&a[i],&temp,i）;
        }
}
```

在上述的例子中，函数 funca 调用了函数 funcb，并且在函数 funca 中使用了 OpenMP 进行并行化。全局变量 gval 是共享的。在 funca 函数的内部，变量 i 由于是循环控制变量，因此是线程私有的；cc 在并行化语句之外声明，是共享的；temp 在循环并行化语句内部的自动变量，是线程私有的；输入的指针变量 a 以及 n 是共享的，都在循环并行化语句之外声明。在函数 funcb 内部，静态变量 sv 是共享的，在程序内存空间中只有一份，因此，在这种使用方式下会引起数据冲突；变量 u 是自动变量，由于被并行线程调用，是线程私有的；参数 x 的本身是私有的指针变量，但是*x 指向的内存空间是共享的，其实际参数即函数 funca 中的 a 数组；参数 y 的本身是私有的指针变量，指向的*y 也是私有的，其实际内存空间即私有的 temp 占用的空间；数值参数 z 是线程私有的。

为了编程序方便，OpenMP 也提供了改变默认变量作用域的方法，即通过 default（shared）

将变量作用域变成默认共享的，或者使用 default（none）将默认作用域去掉，则需要显式的指定变量的作用域，否则会被认为错误。在 C/C++里面没有 default（shared）的作用域子句（在 Fortran 里面有），这是由于很多库函数为了效率使用了宏，对于 OpenMP 不能提供很好的支持，会造成程序的错误。

规约操作的并行化

一类经常需要并行化的计算是规约操作。在规约操作中，会反复将一个二元运算符应用在一个变量和另外一个值上，并把结果保存在原变量中。一个常见的规约操作就是数组求和，使用一个变量保存部分和，并把数组中的每一个值加到这个变量中，就可以得出最后所有数组的总和。OpenMP 对于这一类规约操作提供了特殊的支持，在使用规约操作时，只需在变量前指明规约操作的类型以及规约的变量。下面是一个规约操作的实例，分别计算两个数组中数值的总和。

代码 5.8 规约操作的并行化

```
# pragma omp parallel for private（arx,ary,n） reduction（+:a,b）
for（i=0;i<n;i++）{
        a=a+arx[i];
        b=b+ary[i];
}
```

并不是所有的操作都能够使用规约操作。下面的表格列出了所有能够在 OpenMP 的 C/C++语言中出现的规约操作。

运算符	数据类型	默认初始值
+	Integer, floating point	0
*	Integer, floating point	1
-	Integer, floating point	0
&	Integer	所有位都开启，~0
|	Integer	0
^	Integer	0
&&	Integer	1
||	Integer	0

表 5.1 OpenMP 规约操作符及规约变量的初值

可以看到，规约操作只对语法内建的数值数据类型有效，对其他类型则无效。如果对其它类型或者用户自定义的类型，则必须使用同步语句来对共享变量进行保护。

对于在 reduction 子句中所指定的每一个变量，都会为每个线程创建一个私有副本，就象使用了 private 子句一样。此后私有副本的初值被设置为表 5.1 所列出的操作初值。在指定了 reduction 子句的区域或循环后，规约变量的原始值与各个私有副本的最终值进行指定的操作，然后用操作结果更新该规约变量的值。在尝试使用 reduction 子句进行多线程程序设计时，要记住以下三个要点：

•在第一个线程到达指定了 reduction 子句的共享区域或循环末尾时，原来的规约变量的值变为不确定，并保持此不确定状态直至规约计算完成。

• 如果在一个循环中使用到了 reduction 子句，同时又使用了 nowait 子句，那么在确保所有线程完成规约计算的栅栏同步操作前，原来的规约变量的值将一直保持不确定的状态。

•各个线程的私有副本值被规约的顺序是未指定的。因此，对于同一段程序的一次串行执行和一次并行执行，甚至两次并行执行来说，都无法保证得到完全相同的结果（这主要针对浮点计算而言），也无法保证计算过程中诸如浮点计算异常这样的行为会完全相同。

私有变量的初始化和终结操作

对于线程私有的变量，在每一个线程开始创建执行的时候其值是未确定的。当然，也可能通过 C++的构造函数来初始化类对象，但一般的内建类型的初始值都未定。对于把私有变量作为临时变量，并在线程内部的使用过程中初始化当然没有问题。然而某些情况下面，我们可能需要在循环并行化开始的时候访问到私有变量在主线程中的同名变量的值，也有可能需要将循环并行化最后一次循环的变量结果返回给主线程中的同名的变量，就好像我们正常地通过串行执行的效果一样。OpenMP 编译制导语句也对于这种需求给予支持，即使用 firstprivate 和 lastprivate 对这两种需求进行支持，使得循环并行开始执行的时候私有变量通过主线程中的变量初始化，同时循环并行结束的时候，将最后一次循环的相应变量赋值给主线程的变量。

代码 5.9 私有变量的初始化和终结操作

```
int val=8;
#pragma omp parallel for firstprivate（val） lastprivate（val）
for（int i=0;i<2;i++）{
    printf（"i=%d val=%d\n",i,val）;
    if（i==1）
        val=10000;
    printf（"i=%d val=%d\n",i,val）;
}
printf（"val=%d\n",val）;
```

下面是程序的执行结果

```
i=0 val=8
i=1 val=8
i=0 val=8
i=1 val=10000
val=10000
```

可以看到，在每一个线程的内部，私有变量 val 被初始化为主线程原有的同名变量的值，并且在循环并行化退出的时候，相应的变量被原有串行执行的最后一次执行对应的值所赋值。

数据相关性与并行化操作

并不是所有的循环都能够使用#pragma omp parallel for 来进行并行化。为了对一个循环进行并行化操作，我们必须要保证数据两次循环之间不存在数据相关性。数据相关性又被称为数据竞争（Data Race）。当两个线程对同一个变量进行操作，并且有一个操作为写操作的时候，就说明这两个线程存在数据竞争。此时，读出的数据不一定就是前一次写操作的数据，而写入的数据也可能并不是程序所需要的。下面是一个典型的具有数据相关性的循环，不能通过直接采用#pragma omp parallel for 的方式将其并行化，因为两个相邻的循环会使用到相同的内存空间中的数据，并且有一个操作为写操作，无法保持数据的一致性。

```
for（int i=0;i<99;i++）
    a[i]=a[i]+a[i+1];
```

为了将一个循环并行化，而不影响程序的正确性，需要仔细检查程序使得程序在并行化之后，两个线程之间不能够出现数据竞争。当然，有时候根据应用的特殊需求，在能够保证得出正确结果的情况下，可以允许存在数据竞争，并将循环并行化。在出现数据竞争的时候，我们可以通过增加适当的同步操作，或者通过程序改写来消除竞争。

下面是一个具有循环之间数据相关性的串行程序

```
for（int j=1;j<N;j++）
    for（int i=0;i<N;i++）
        a[i,j]=a[i,j]+a[i,j-1];
```

但是，由于内部循环的过程中，j 的值是固定的，对于内部循环来说，循环之间没有数据的相关性，可以将内部循环进行并行化，得到我们的并行化版本。

代码 5.10 具有循环之间数据相关性的并行程序

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

可以看到，在进行并行化的过程中，我们必须要仔细分析循环之间的数据相关性，在某些时候，可以通过程序改写，消除产生数据相关性的原因，才能获得并行应用程序。