从 OpenMP* 着手入门

摘要

您现在可能已经了解到，如果想充分利用含超线程（HT）技术的处理器的性能优势，就必须并行执行应用。可是，应用的并行执行需要线程参与，且应用的线程化也并非易事。其实，我们可以借助 OpenMP* 这样的工具更轻松地实现应用线程化。

本文作为系列白皮书三步曲之第一步，旨在教授那些具有丰富经验的 C/C++ 编程人员如何借助 OpenMP 充分利用超线程（HT）技术的优势。本文作为开篇之首，首先将为您详细介绍如何并行执行循环，即：工作共享。我们的第二篇白皮书将教您如何利用非循环并行 能力和 OpenMP 的其它特性进行编程。最后，我们将会在末篇与您深入探讨 OpenMP 的运行时库函数以及英特尔® C++ 编译器是如何在出现错误时调试您的应用。

OpenMp 简介

OpenMP 的设计人员希望能为编程人员提供一个简单方法，以使他们无需了解如何制作、同步和毁坏线程的知识，甚至无需决定创建的线程数，即可轻松地线程化其应用程 序。他们专门开发了一套独立于平台的编译器范式、指令、函数调用和环境变量，以明确地指导编译器如何将线程插入应用，并明晰地指出应用插入的准确位置。这 样，大多数循环仅在开始循环前直接插入一条编译指令即可实现线程化。此外，您还可以将细节事情留给编译器和 OpenMP 处理，从而赢取更多时间来决定哪些循环应该线程化，以及如何最佳重组算法获得最高性能。至此，在使用 OpenMP 对“热点”（您应用中最耗时的循环）进行线程化时，OpenMP 的性能将可实现最优化。

下面这个例子就为我们展示了 OpenMP 强大的功能和简易性。下列循环可将 32 位 RGB（红、绿、蓝）像素转换为 8 位灰阶像素。它只需在开始循环前直接插入一条编译指令，即可实施并行执行。

```
#pragma omp parallel for
for (i=0; i < numPixels; i++)
{
    pGrayScaleBitmap[i] = (unsigned BYTE)
                (pRGBBitmap[i].red * 0.299 +
                 pRGBBitmap[i].green * 0.587 +
                 pRGBBitmap[i].blue * 0.114);
}
```

让我们来详细了解一下这个循环。首先，上例中使用了“工作共享”。“工作共享”作为 OpenMP 所用的一个一般术语，主要用于描述线程间工作的分配情况。如上例所示，当工作共享用于 for 构造函数时，循环迭代会在多个线程间进行分配，这样就可确保每个循环迭代刚好执行一次，并与一个或以上的线程并行执行。由于 OpenMP 完全可以自行决定创建的线程数量，以及如何以最佳方式创建、同步和毁坏线程。所以，编程人员只需告知 OpenMP 应该针对哪个循环进行线程化，即可轻松完成全部工作。

OpenMP 对可以线程化的循环设置了以下五个限制条件：

- 循环变量必须为**带符号整数**型。DWORD 等无符号整数无法实现线程化。
- 必须按照 **loop_variable** <、<=、>、或 >= **loop_invariant_integer** 形式执行比较操作。

- **for** 循环的第三个表达式或增量部分必须与一个循环不变式值进行整数相加或整数相减。
- 如果比较操作的形式为 < 或 <=，则循环变量必须每个迭代增加一次；反之，如果比较操作的形式为 > 或 >=，则循环变量必须每个迭代减少一次。
- 循环必须为一个基本模块，意即不允许从内循环跳至外循环，除非使用 **exit** 语句来终止整个应用。如果使用 **goto** 或 **break** 语句，则它们必须在循环内部，而非外部进行跳跃。这同样适用于异常处理；但异常情况必须控制在循环内部。

尽管上述限制条件显得有些苛刻，但您可以依照这些限制条件轻松地对不符合条件的循环进行重新编写。

编译基本要点

如欲使用 OpenMP 范式进行编程，则需要一个与 OpenMP 相兼容的编译器和多个线程安全库。通常，最理想的选择是英特尔® C++ 编译器 7.0 版或更高版本。（英特尔® Fortran 编译器也支持 OpenMP。）而将下列命令行选项加入编译器，则可以起到提醒它注意 OpenMP 编译指令并插入线程的作用。

如果您省略命令行上的 **/Qopenmp**，则编译器将忽略 OpenMP 编译指令，为您提供一个十分简单的方法，这样无需改变任何源代码即可生成单线程版本。此外，英特尔® C++ 编译器还支持 OpenMP 2.0 规范。如欲获得最新信息，请您务必查看与英特尔® C++ 编译器一道发布的版本说明和兼容性信息。如欲获得完整的 OpenMP 规范，请访问：http://www.openmp.org* 英文。

对于条件编译而言，编译器会定义 _OPENMP。如有必要，此定义可接受下列测试。

```
#ifdef _OPENMP
    fn();
#endif
```

所有 OpenMP 编译指令的通用形式是：

如果命令行不以 **pragma omp** 开头，则它不是 OpenMP 编译指令。如欲获得规范中的编译指令完整列表，请访问：http://www.openmp.org 英文。

您可使用 **/MD** 或 **/MDd** （调试用）命令行编译器选项挑选 C 运行时线程安全库。通过在面向 C/C++ 项目设置的代码生成类中使用 Microsoft Visual C++* 来选择多线程 DLL 或调试多线程 DLL，即可获得这些选项。

几个简单的示例

下列示例表明了 OpenMP 使用起来十分简单。在一般的实践中，我们都会不可避免地遇到一些突发问题，而只有坚持不懈解决难题，才能在发展路上不断前行。

问题 1：下列循环将一个阵列修剪为 0 到 255 之间的范围。您需要使用 OpenMP 编译指令对其进行线程化。

解决方案：在循环开始前直接插入下列编译指令即可。

问题 2：下列循环可生成从数字 0 到 100 的平方根表。您需要使用 OpenMP 对其进行线程化。

解决方案：循环变量不是带符号整数，因此需要对其进行更改，并为其添加编译指令。

避免数据相关性和竞态条件

由于循环存在着数据相关性，因此即使循环全部满足上述五个限制条件，且编译器对循环也进行了线程化，循环仍可能无法正常进行。这是因为，只要循环进行不同迭代，或者更确切点说，当循环迭代在不同的线程读写共享内存上执行时，数据相关性就会存在。请参考下列阶乘计算范例。

```
// Do NOT do this. It will fail due to data dependencies.
// Each loop iteration writes a value that a different
iteration reads.
#pragma omp parallel for
for (i=2; i < 10; i++)
{
   factorial[i] = i * factorial[i-1];
}
```

编译器对循环进行线程化之所以失败，是因为至少有一个循环迭代与另外一个不同的迭代存在着数据相关性。这样的情形我们称之为“竞态条件”。竞态条件只在使用共享资源（如内存）和并行执行时出现。要解决这一问题，您可以重新编写循环，或者选择一个不包含竞态条件的算法。

一般来说，竞态条件很难被检测到，因为在给定的情况下，变量有可能会以程序函数正常运行的顺序“赢得竞争”。而一项程序一次正常运行恰恰不表示永远正常运 行，所以，您最好先在各种机器上测试您的程序，譬如某些支持超线程（HT）技术的机器和某些采用了多个物理处理器的机器等，才能确保程序万无一失，顺利运 行。此外，英特尔® 线程检测器等工具犹如敏锐的“眼睛”，也会对您的检测工作提供帮助。而此时，传统的调试器在检测竞态条件方面将变得毫无用处，因为一旦它们让一个线程停止 “竞争”，其它线程将继续大幅度地改变运行时行为。

管理共享数据和私人数据

几乎每个循环（至少在其有用时）都会读取和写入内存，所以编程人员只需告知编译器哪部分内存应在线程 间共享、以及哪部分应保持私有状态即可轻松完成工作任务。而一旦内存确定为共享，所有线程就将访问同一内存地址。当内容确定为私有时，每个线程将获得一份 用于私有访问的单独变量副本。当循环结束时，这些私有副本将被销毁。在默认情况下，除了私有的循环变量外，所有变量均可共享。内存将通过以下两种方式宣告 私有：

- 宣告循环内部的变量不含静态关键字，且位于并行 OpenMP 指令内部。
- 指定 OpenMP 指令上的私有子句。

由于变量 **temp** 为共享，所以下列循环无法正常运行。只有将它变为私有，循环才能正常运行。

```
// WRONG. Fails due to shared memory.
// Variable temp is shared among all threads, so while one thread
// is reading variable temp another thread might be writing to it
#pragma omp parallel for
for (i=0; i < 100; i++)
{
   temp = array[i];
   array[i] = do_something(temp);
}
```

下列两例均宣告变量 **temp** 为私有内存，这样便轻松解决了上述问题。

```
    // This works. The variable temp is now private
    #pragma omp parallel for
    for (i=0; i < 100; i++)
    {
        int temp; // variables declared within a  parallel construct
         // are, by definition, private
        temp = array[i];
        array[i] = do_something(temp);
    }    // This also works. The variable temp is declared private
    #pragma omp parallel for private(temp)
   for (i=0; i < 100; i++)
    {
        temp = array[i];
       array[i] = do_something(temp);
    }
```

每次在指示 OpenMP 对循环进行并行化处理时，您应仔细检查包括调用函数所作的引用在内的所有内存引用。并行构造函数内部声明的变量被定义为私有变量，但其采用静态声明符声明的情况除外（因为静态变量不在堆栈上分配）。

Reduction 子句

事实上，累计值的循环十分普遍，OpenMP 即是采用一个特定的子句来支持此类循环。您可以采用下列循环来计算一组整数的和。

```
 sum = 0;
    for (i=0; i < 100; i++)
    {
      sum += array[i];// this variable needs to be shared to generate the correct results, but private to
avoid race conditions from parallel execution
    }
```

前面循环中的变量 **sum** 必须与其它循环共享，才能生成正确的结果，而它只有是私有变量时，才可允许多个线程访问。为了解除变量的这一两难处境，我们可以使用 OpenMP 提供的 **reduction** 子句，高效地对循环中的一个或以上变量进行数学归约运算联合操作。下列循环就是采用 **reduction** 子句，生成了正确的结果。

```
    sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (i=0; i < 100; i++)  {
   sum += array[i];
  }
```

实际上，OpenMP 可为每个线程提供变量 **sum** 的私有副本。而当线程退出时，它还会将值加在一起，并将结果放在变量的一个全局副本中。

下表列出了可能的归约运算和临时私有变量的初始变量值（也是数学标识值）。

运算	临时私有变量初始化
+(加)	0
-(减)	0
*(乘)	1
&(位与)	~0
| (位或)	0
^(位异或)	0
&&(条件与)	1
|| (条件或)	0

在给定并行结构上指定以逗号分隔的变量和归约运算符后，即可在循环中执行多个归约运算操作。它的唯一要求如下：

1. 仅在一次归约中列出归约变量
2. 它们无法声明为常量，且
3. 它们在并行结构中无法声明为私有。

循环排程

负载均衡（即在线程间平均分配工作）是并行应用性能最为重要的属性之一。它之所以十分重要，主要在于 它能确保处理器大多数时间（如果不是所有时间）均处于运行状态。如果负载不均衡，有些线程就可能会领先其它线程，先行完成工作，这就会使处理器资源闲置， 白白流失性能提升机会。

在循环结构内部，负载不均衡通常来自循环迭代之间计算时间的差异。而通过检查资源代码，就可轻松确定循环迭代计算时间的差异。在大多数情况下，您会看到各 个循环迭代所消耗的时间量是相同的。如果事实并非如此，则也可能找到一套消耗近似时间量的迭代。例如，有时所有偶数迭代消耗的时间与所有奇数迭代所消耗的 时间大致相当。同样，一个循环上半部分消耗的时间与下半部分所消耗的时间也有可能大致一样。当然，退一步讲，我们也可能无法找到一套执行时间完全相同的循 环迭代。但是，无论应用处于上述何种情形，您都应为 OpenMP 提供额外的循环排程信息，以便它更平均地在线程之间（乃至处理器之间）分配循环迭代，从而实现最佳负载均衡。

默认情况下，OpenMP 一般都会假定所有循环迭代所消耗的时间相同。在此前提下，OpenMP 在线程间大致平均分配循环迭代，就可将因错误共享引起的内存冲突几率降至最低。这是因为循环一般按顺序读写内存，因此将循环分成几大块（正如使用两个线程 时要分成前半部分和后半部分一样）就会使内存重叠的机会降到最低，所以这种做法完全可行。但是，事物都有其两面性，尽管这种方法可能是解决内存问题的理想 之举，但它却不利于保持负载均衡。尤为遗憾的是，这两方面的矛盾性业已证实：一般来说，对负载均衡有利则可能对内存性能不利。因此，性能工程师必须通过测 量性能，以此找寻何种方法能够实现理想效果，从而在最佳内存使用和最佳负载均衡之间找到一个完美的平衡点。

采用下列句法可以将循环排程信息传送给并行结构上的 OpenMP。

```
#pragma omp parallel for schedule(kind [, chunk size])
```

如下表所示，可为 OpenMP 提供四种不同的循环排程（提示）。但是，可选参数（块）（指定的）必须是一个循环不变量正整数。

类别	说明
静态	首先将循环划分为大小相等的若干块，如果循环迭代次数不能被块大小与线程数之积相整除，则应将循环划分为大小尽可能相等的块。默认情况下，块大小等于循环数除以线程数。将块大小设置为 1，以实现交错迭代。
动态	使用内部工作队列，以按块大小为每个线程分配循环迭代块。完成一个线程后，它将从工作队列的顶部开始检索下一个循环迭代块。默认情况下，块大小为 1。本排程提示需额外开销，请谨慎使用。
引导	与动态排程相类似，但是引导排程首先获取的是较大的块，然后再通过逐步缩减块大小，来缩短线程遍历工作队列获取更多工作所需的时间。其中可选块参数用于规定使用块大小的最小值。默认情况下，块大小约等于循环数除以线程数。
运行时	通过 OMP_SCHEDULE 环境变量来指定使用上述三种循环排程中的哪一类。OMP_SCHEDULE 是一个字符串，其格式与并行结构中将显示的格式完全相同。

范例

问题：将下列循环并行化

```
for (i=0; i < NumElements; i++)
{
   array[i] = StartVal;
   StartVal++;
}
```

解决方案：注意数据相关性

如上所述，由于循环包含数据相关性，因此只有进行快速更改才可能使其并行化。下列所示的新循环就以同样的方式填充阵列，但却不包含数据相关性。此外，新循环还可采用 SIMD 指令进行编写。

```
#pragma omp parallel for
 for (i=0; i < NumElements; i++)   {
```

```
    array[i] = StartVal + i;
}
```

值得一提的是：代码并非百分之百相同，这是因为变量 **StartVal** 的值没有增加。所以，当并行循环完成时，得到的变量值会不同于串行版本生成的值。如果您在循环结束后还需要 **StartVal** 的值，则需要添加下列所示的语句。

```
// This works and is identical to the serial version.
#pragma omp parallel for
 for (i=0; i < NumElements; i++)
{
 array[i] = StartVal + i;
}
StartVal += NumElements;
```

总结

我们此次编写 OpenMP 入门知识，主要是为了让编程人员能够从线程化的细枝末节中解放出来，有更多的时间关注更重要的问题。有了 OpenMP 编译指令，大多数循环仅使用一个简单的语句即可实现线程化。这也正是 OpenMP 的威力所在，是其关键优势的完美体现。本白皮书向您介绍了 OpenMP 的相关概念和简易方法，便于读者入门实践。我们后两篇文章将在此基础上，教您如何使用 OpenMP 线程化更复杂的循环和更常见的编程结构。

借助 OpenMP* 实现更多工作共享

作者：Richard Gerber

正如您所知，OpenMP* 包含一组非常强大的编译制导语句，能够帮助实现循环的并行化。但是您可能还不知道，OpenMP 的线程化功能不仅仅局限于循环。一旦“parallel for”结构出现故障，OpenMP 便可发出其它制导语句、结构和函数调用来进行救援。

这是白皮书三部曲中的第二篇。我们的一系列白皮书将为您——经验丰富的 C/C++ 程序员详细介绍 OpenMP 的入门知识，以及如何在您的应用中简化编程的创作、同步处理以及删除工作。其中第一篇向您简要介绍了 OpenMP 最为常见的特性：循环工作共享。第二篇白皮书将教您如何利用非循环并行能力和 OpenMP 的其它一些常见特性进行编程。最后一篇将重点讨论 OpenMP 的运行库，以及如何在出现错误时调试您的应用。

并行 (Parallel) 结构

在本系列的第一篇白皮书《OpenMP* 入门》中，您已经了解了“parallel for”指令是如何将循环迭代拆分到多个线程上的。当 OpenMP 遇到“parallel for”指令时，便会创建线程，并将循环迭代分配到其中。这时，在并行区域的末尾，线程将被暂挂，并等待下一个并行段。可是，这种创建和暂挂功能会产生一些开销，而且如果两个循环相邻（如下例所示），就没必要使用此功能。

```
#pragma omp parallel for
for (i=0; i<x; i++)
fn1();
#pragma omp parallel for // adds unnecessary
                         // overhead
```

```
    for (i=0; i<y; i++)
    fn2();
```

如果一次性进入一个并行段，并直接对其进行工作拆分，便可避免上述开销。并且，由于以下代码进入并行段的流程仅执行一次，所以尽管看似与前面的代码在功能上完全相同，但是运行速度会更快。

```
    #pragma omp parallel
    {
    #pragma omp for
    for (i=0; i<x; i++)
        fn1();
    #pragma omp for
    for (i=0; i<y; i++)
          fn2();
    }
```

理想情况下，应用的所有关键性能部分（也称为热点）都将在并行段中执行。但是在现实中由于仅由循环构成的程序基本不存在，因此需要更多的结构来处理非循环代码。

区段

“section” 结构指示 OpenMP 将应用的已识别段分配到多个线程上。下面的示例中便使用了工作共享来处理循环和区段：

```
    #pragma omp parallel
    {
      #pragma omp for
          for (i=0; i<x;i++) Fn1();
       #pragma omp sections
       {
    #pragma omp section
    {
    TaskA();
    }
    #pragma omp section
    {
          TaskB();
    }
    #pragma omp section
    {
          TaskC();
    }
       }
```

在这里，OpenMP 首先创建了一组线程，然后将循环迭代分配到其中。循环结束后，区段被分配到线程中，从而可使每条线程与其它线程精确地并行执行一次。如果程序包括的区段数 量多于线程数，则剩余的区段将在线程处理完前面的区段之后再行调度。与循环迭代不同，OpenMP 将完全控制线程执行区段

的方式、时间和顺序。但是，您仍然可以通过与循环结构相同的方式使用“私有”和“减少”子句，来控制变量的分享或私有属性。

Barrier 与 Nowait

Barrier 是 OpenMP 用于同步处理线程的一种同步化形式。所有线程执行到“barrier” 时要停止，直到并行段中的所有线程都执行到该“barrier”时才继续往下执行。您一直都在使用隐式“barrier”，只不过没有意识到它在针对结构 的工作共享中的作用。该隐式“barrier”在循环结束时，会等待所有线程完成循环后，再让程序继续执行更多工作。该“barrier”可使用**“nowait”** 子句移除，如以下代码范例所示。

```
    #pragma omp parallel
    {
    #pragma omp for nowait
        for (i=0; i<100; i++)
   compute_something();
    #pragma omp for
       for (j=0; j<500; j++)
    more_computations();
    }
```

在以上范例中，程序不等所有线程处理完第一个循环便会立即继续处理第二个循环。这种“因情而定”的处理方式将使您受益匪浅，可显著缩短线程的空闲时间。此外，**“nowait”**子句可与区段结构共同使用，移除隐式“barrier”。

反之，它也支持“barrier”的添加，如下例所示。

```
    #pragma omp parallel
    {
    A bunch of work...
    #pragma omp barrier
    Additional work...
    }
```

当所有线程需要在完成更多工作之前结束一项任务时，这一结构就会非常实用，例如在显示帧缓冲前对其进行更新。

主线程与单线程

不幸的是，应用却不仅仅只由循环和并行段两部分构成。并行区域内无疑更需要仅由一条线程执行一次的情况，尤其是当您为了减少开销而尽可能的增大并行 段的大小的时候。为了满足这一需求，OpenMP 内置了一种方法，用以指定并行段中的代码顺序应仅由一条线程执行一次。并且，OpenMP 还指派了特定的单线程来执行。但是如果需要，您也可指定仅由主线程来执行代码，如下例所示。

```
#pragma omp parallel
   {
       do_multiple_times();
       // every thread calls
       //this function
       #pragma omp for
          for (i=0; i<100;i++)// this loop is divided among the threads
```

```
      fn1();
   // implicit barrier at the end of the above loop
   // will cause all thread to synchronize here
       #pragma omp master
   fn_for_master_only();  // only the master thread calls this function
       #pragma omp for nowait
      for (i=0; i<100;i++)  // again, loop is divided among threads
          fn2();
// The above loop does not have a barrier, and
// threads will not wait for each other. One thread,
// hopefully the first one done with the above loop,
// will continue and execute the code below.
   #pragma omp single
      one_time_any_thread();
   // any thread will execute this
 }
```

“Atomic”指令操作

并行执行代码时，很有可能需要同步共享内存。根据定义，“atomic”指令操作保证不会被打断，并且可用于更新共享内存位置的语句，以避免一些竞态条件。在以下代码行中，程序员已经明确了变量在语句当中保持稳定的重要性。

```
a[i] += x; // may be interrupted half-complete
```

虽然单条汇编指令的执行从未被打断过，但是高级语言（如 C/C++）中的语句却会被转化为多条汇编指令，因而有可能被打断。在上例中，**a[i]** 的值可能会在读值、添加变量 **x**，以及将值写回内存等的汇编语句中变化。但是，以下 OpenMP 结构却能确保语句自动执行代码，不存在被打断的可能性。

```
#pragma omp atomic
a[i] += x; // never interrupted because defined
          //atomic
```

“Atomic”指令操作是下列非过载操作中的基本形式之一。

Expr1++	++expr1
Expr1--	--Expr1
Expr1 += expr2	Expr1 -= expr2
Expr1 *= expr2	Expr1 /= expr2
Expr1 <<= expr2	Expr1 >>= expr2
Expr1 &= expr2	Expr1 ^=expr2
Expr1 |= expr2	

如果操作系统具备相关特性和硬件能力，那么 OpenMP 可选择最有效的方法来执行语句。

“Critical Section”

“critical section”可以保护代码块免受多次访问。当一条线程遇到关键段时，只有在其它线程都没有处于任何关键段时，才能进入其中一个或几个区段。以下范例使用了一个未命名的关键段。

```
    #pragma omp critical
    {
        if (max < new_value)
   max = new_value
    }
```

由于每个线程都会有力竞争相同的全局关键段，因而全局或未命名的关键段未必会影响性能。为此，OpenMP 对关键段进行了命名。这些命名的关键段可支持更细化的同步，所以只有那些需要在特殊区段上隔离的线程才会被隔离。下面的范例即依此对以上范例的编码进行了改进。

```
    #pragma omp critical(maxvalue)
    {
          if (max <new_value) max = new_value;
    }
```

借助命名的关键段，应用可以拥有多个关键段，线程也可一次性进入一个以上的关键段。值得注意的是，进入嵌套的关键段有可能导致死锁，这一点 OpenMP 无法探测到。因此，在使用多个关键段时，要格外细心检查那些可能“隐藏”在子程序中的关键段。

First Private, Last Private

添加到您工具条中的 OpenMP 制导语句和功能是**“firstprivate”** 和 **“lastprivate”** 子句。这些子句可在变量的全局“主”副本与私有临时副本之间复制数值，反之亦然。

“firstprivate”子句用于指定 OpenMP 采用主变量的值对私有变量的值进行初始化。通常，临时私有变量并未定义初始值，因而可节省副本的性能开销。**“lastprivate”**子句可以在破坏发生前，将变量中最后一个迭代（按照顺序）计算出的值复制到主变量中。因此，变量可以同时声明为**“firstprivate”** 和 **“lastprivate”**。

通常，我们可通过对代码进行细小改动来避免使用这些子句。但是在特定环境中，它们却十分有用。例如，以下代码可将彩色图片转换为黑白图片。

```
    for (row=0; row<height; row++)
    {
       for (col=0; col<width; col++)
       {
          pGray[col] = (BYTE)(pRGB[row].red * 0.299 + pRGB[row].green * 0.587 +
pRGB[row].blue * 0.114);
       }
       pGray += GrayStride;
       pRGB += RGBStride;
    }
```

但是，如何在位图内将指示器移动到正确的位置呢？我们可以通过以下代码执行每一迭代的地址计算：

```
    pDestLoc = pGray + col + row * GrayStride;
    pSrcLoc = pRGB + col + row * RGBStride;
```

不仅如此，该代码还能够在每个像素上执行许多额外的计算。同样，如下例所示，**“firstprivate”** 子句可用作一种初始化指令：

```
BOOL FirstTime=TRUE;

#pragma omp parallel for private (col)
firstprivate(FirstTime, pGray, pRGB)
   for (row=0; row<height; row++)
   {
      if (FirstTime == TRUE)
      {
         FirstTime = FALSE;
         pRGB += (row * RGBStride);
         pGray += (row * GrayStride);
      }
      for (col=0; col<width; col++)
      {
         pGray[col] = (BYTE) (pRGB[row].red * 0.299 +
                  pRGB[row].green * 0.587 +
                  pRGB[row].blue * 0.114);
      }
      pGray += GrayStride;
      pRGB += RGBStride;
   }
}
```

技巧范例

问：请描述以下两个循环在执行上的不同之处。

循环 1：(parallel for)

```
    int i;
    #pragma omp parallel for
    for (i='a'; i<='z'; i++)
    printf ("%c", i);
```

循环 2：（仅 parallel）

```
    int i;
    #pragma omp parallel
    for (i='a'; i<='z'; i++)
    printf ("%c", i);
```

答：第一个循环仅打印一次字母表，第二个循环可将字母表打印多次。如果变量 **i** 被声明为私有，那么它将在每个线程上打印一次字母表。

高级 OpenMP* 编程

作为白皮书三部曲中的最后一篇，本文将为您介绍经验丰富的 C/C++ 程序员如何开始使用 OpenMP*，以及如何在应用中简化线程的创建、同步以及删除工作。我们的一系列白皮书将为您全面揭秘 OpenMP，其中第一篇向您简要介绍了 OpenMP 最常见的特性：循环工作共享。第二篇告诉您如何充分利用非循环并行能力及如何使用同步指令。最后一篇则讨论了库函数、环境变量、如何在发生错误时调试应 用，以及最大限度地发挥性能的一些技巧。

运行时库函数

您可能还记得，OpenMP 由一套编译指令、函数调用和环境变量组成。前两篇文章只讨论了编译指令，本文将重点探讨函数调用和环境变量。这样安排的理由很简单：编译指令是 OpenMP 的“原因”，它们提供最大程度的简易性，不需要改变源代码，并且您可忽略它们来生成代码的系列版本。另一方面，使用函数调用需要改变程序，这将为执行系列 版本（如需）带来困难。如果遇到疑问，请您在计划使用函数调用（包括 标头文件）时，尝试使用编译指令并保留函数调用。当然，您还应继续使用英特尔® C++ 编译器命令行切换 **/Qopenmp**。进行链接不需要其它库。

下表列出了四个最常使用的库函数，分别用于检索线程总数，设置线程数，返回当前线程数，及返回可用逻辑处理器的数量。如欲获得 OpenMP 库函数的全部列表，请务必访问 OpenMP 网站

int omp_get_num_threads（无效）；	返回当前使用的线程数。如果在并行区域外被调用，则此函数的返回值为 1。
int omp_set_num_threads(int NumThreads)	此函数用于设置进入并行区域将使用的线程数。它可撤销 OMP_NUM_THREADS 环境变量。
int omp_get_thread_num（无效）；	返回数值在 0（主线程）和线程总数 -1 之间的当前线程数。
int omp_get_num_procs（无效）；	返回可用处理器数量。含超线程技术的处理器将被算作两颗处理器。

以下为使用上述函数来打印字母表的例子。

```
omp_set_num_threads(4);
#pragma omp parallel private(i)
{   // This code has a bug. Can you find it?
     int LettersPerThread = 26 / omp_get_num_threads();
     int ThisThreadNum = omp_get_thread_num();
     int StartLetter = 'a'+ThisThreadNum*LettersPerThread;
     int EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread;
for (i=StartLetter; i<EndLetter; i++)
printf ("%c", i);
```

```
}
```

该例子阐述了使用函数调用（而非编译指令）的几个重要概念。首先，您必须重新编写代码。重新编写就意 味着额外的记录、调试、测试和维护工作。其次，没有 OpenMP 的帮助很难甚或不可能进行编译。再次，极易产生缺陷；例如在上述循环中，如果线程数不是 26 的倍数，则循环就无法打印出字母表上的所有字母。如果您没有额外费心去创建自己的工作队列算法，那么最终您将会失去调整循环调度的能力。您将受制于自己的 调度——极可能像上述例子中的静态调度。

环境变量

OpenMP 规范规定了两个常用环境变量（如下表所示）。

环境变量	说明	实例
OMP_SCHEDULE	控制 for 循环工作共享结构的调度。	**设置 OMP_SCHEDULE= "guided, 2"**
OMP_NUM_THREADS	设置默认线程数。使用 omp_set_num_threads() 函数调用可撤销该值。	**设置 OMP_NUM_THREADS =4**

通常可以使用其它特定编译器环境变量。请务必检查您的编译器文件，了解其它变量信息。

调试

调试线程化应用十分棘手，因为调试器会改变运行时性能，掩盖竞态条件。甚至打印声明都可以掩盖问题， 因为它们使用同步和操作系统函数。而 OpenMP 进一步加剧了情况的复杂性。OpenMP 会插入私有变量、共享变量和其它代码，该代码如果不借助专用的 OpenMP 感知调试器就无法查出和越过。因此，关键的调试手段是消除流程。

首先，请您务必意识到大多数错误都是竞态条件。大多数竞态条件都是由实际应被声明为私有变量的共享变量引起的。首先应查看并行区域内的变量，确保这些变量 在需要时被声明为私有。同时，应检查并行结构内部被调用的函数。在默认情况下，堆栈上声明的变量为私有变量，但 C/C++ 关键词 **static** 会改变将放在全局堆上的变量，从而使其被 OpenMP 循环共享。下表所示的 **default(none)** 子句可用于协助找到那些难于发现的变量。如果您指定了 **default(none)，**则每个变量都必须采用数据共享属性子句予以声明。

```
#pragma omp parallel for default(none) private(x,y)
 shared(a,b)
```

另一个常见错误是使用未经初始化的变量。请谨记私有变量在进入并行结构时没有初始值。请仅在需要时使用 **firstprivate** 和 **lastprivate** 子句来对它们进行初始化，否则会增加大量额外的开销。

如果您仍然找不到缺陷的所在，原因可能是您使用了过多的代码。请尝试寻找二进制。通过在并行结构上使用 **if(0)** 或完全注释掉编译指令来强制并行部分再次连续。另一个方法是强制大部分并行区域成为临界区。挑选一个您认为包含缺陷的代码区域，将其置于临界区内。试着找 出在临界区内突然工作、在临界区外无法工作的代码段。然后查看变量，检查缺陷是否明显。如果这样做仍然不能凑效，则设置英特尔® C++ 编译器特定环境变量 **KMP_LIBRARY=serial，**尝试对整个程序进行设置使之按顺序运行。如果代码仍然不能工作，则不使用 **/Qopenmp** 对其进行编译，以确保系列版本能够正常工作。

英特尔® 线程检测器

英特尔® 线程检测器既不是纯粹的调试器，也不是彻底的 lint 工具，它可提供极具价值的并行执行信息和调试提示。英特尔® 线程检测器利用源代码或二进制替换和测试覆盖插入，对 OpenMP 编译指令、Win32 线程化应用编程接口和所有内存访问进行监控，尝试识别代码错误。它可以找到在测试中不常发生，但在客户现场时常出现的错误。请务必谨记在访问尽可能少的内 存时，使用该工具测试所有代码路径，以便加快数据收集进程。在通常情况下，您需要对源代码或数据集稍作修改，以减少应用处理的数据量。

英特尔® 线程检测器是为英特尔® VTune™ 性能分析器提供的一个插件，请访问 http://www.intel.com/cd/software/products/apac/zho/index.htm 获取相关信息。

性能
OpenMP 线程化应用的性能主要取决于以下几点：

- 单线程代码的潜在性能。
- CPU 占用率、闲置线程和负载平衡不足。
- 并行执行的应用的比例。
- 线程之间的同步和通信数量。
- 创建、管理、销毁和同步化线程所需的开销因 fork-join 转换（从单线程到并行线程或从并行线程到单线程的转换）的次数的增加而增加。
- 内存、总线带宽和 CPU 执行单元等共享资源的性能限制。
- 由共享内存或错误共享内存引起的内存冲突。

线程化代码性能主要可归结为以下两点：1) 单线程版本如何运行；2) 您如何使用最少的开销在多个处理器间分配工作。架构良好的并行算法或应用是确保出色性能的良好开端。显然，不宜从并行化冒泡排序法（即使是采用人工优化的 汇编语言编写而成）开始。同时您还应考虑可扩充性；创建一个可在两颗 CPU 上运行良好的程序不如创建一个可在 n 颗 CPU 上都能运行良好的程序那样高效。请记住，一旦采用了 OpenMP，线程数量就将由编译器来决定（而不是您），因此无论线程数量如何，重要的是使程序能够正常运行。由于专为两条线程而创建，因此生产者/消费 者架构效率不高。

一旦算法就位，则应确保代码在英特尔® 架构上高效运行，而单线程版本将会助益良多。关闭 OpenMP 编译器选项，您就能生成一个单线程版本，并可通过常用优化集运行该版本。Richard Gerber 所著的**软件优化指南**（在技术书籍销售处有售）为单线程优化提供了有益的参考。一旦您获得了单线程性能，您就可以开始生成多线程版本，并进行一些分析了。

首先，请查看操作系统的闲置循环所花费的时间。英特尔® VTune 性能分析器是协助进行此项调查的一个得力工具。闲置时间可以指示不平衡的负载、大量受阻同步和连续区域。解决这些问题后，返回 VTune 性能分析器，寻找过量的高速缓存未命中和错误共享等内存问题。解决这些基本问题后，您将拥有一个优化的并行程序，它将在超线程技术以及多个物理 CPU 上良好运行。

听起来简单，实际上却很神奇，不是吗？优化的确需要耐心、不断的试错和实践。制作一些小的测试程序来模拟您的应用使用电脑资源的方式，可帮助您了解哪些任 务处理起来更快。请务必在并行部分尝试不同的调度子句。与计算时间相比，如果并行区域的开销很大，您可使用下述例子中的 **if** 子句来按顺序执行该部分。

```
#pragma omp parallel for if(NumBytes > 50)
```

由于本系列白皮书均关于 OpenMP，因此本文暂且不能为您提供一个优化性能的完整且详细的方法。如

果您想了解更多性能优化方面的知识，请访问我们英特尔® 软件网络网站上的其它文章。我已经将自己喜欢的几篇文章附在本文末尾的参考资料里，希望能够对您有所帮助。

英特尔® 线程分析工具

最后，我要提一下英特尔® 线程分析工具。与英特尔® 线程检测器一样，它是英特尔® VTune 性能分析器的一个插件，可以通过图形直观显示出某项应用的线程性能状况。它可显示出花费在并行部分、连续部分、开销、同步化及其它多个方面的时间。它能够 用性能测量和监控指令替代 OpenMP 编译指令来记录数据。如欲下载英特尔® 线程分析工具和英特尔® VTune 性能分析器，请访问 http://www.intel.com/cd/software/products/apac/zho/index.htm。

总结

OpenMP 由一系列灵活、简单的编译指令、函数调用和环境变量组成，用来明确指导编译器对您的应用进行线程化处理的方式和位置。只要充分利用 OpenMP 的功能，线程化编程将和单线程编程一样轻松。最后，祝您的线程化编程之路一帆风顺！