上海大学

SHANGHAI UNIVERSITY

实验　多核环境下 OpenMP 并行编程

组　　号	5
学号姓名	20121034 胡才郁
实验序号	3
日　　期	2023 年 5 月 9 日

评分内容	评分
内容完整，有实验目的、步骤、分析、总结	
内容质量，表述清楚正确，条理清晰，分析总结到位	
格式规范，标题、段落、公式、图表、代码符合专业文献出版要求	
综合得分	

要求 1

OpenMP 介绍

OpenMP 的主要目标是使并行编程变得容易且高效，通过在代码中添加简单的编译指令来进行并行化。这些指令使用特殊的语法标记来标识哪些代码段可以并行执行，因此程序员可以针对性地对代码进行并行化，而不需要全面地重构整个应用程序。

本次实验在本人虚拟机上的 Ubuntu 操作系统上完成，具体而言，OpenMp 的安装可以通过以下步骤进行：

1. 输入以下命令以安装 OpenMP 库：

```
sudo apt-get install libomp-dev
```

2. 安装成功后，在 gcc 编译时加入`-fopenmp`参数即可使用 OpenMP 库。

要求 2

创建多线程，输出线程号和线程数。

OpenMP 提供`omp_set_num_threads()`函数可以设置程序的线程数，而`omp_get_thread_num()`函数可以获取当前线程的编号，`omp_get_num_threads()`函数可以获取当前线程的总数。通过这些函数，可以很方便地实现多线程的创建。在下面的代码中，通过 tid 来获取线程编号，并且指定 tid 为 0 的线程输出线程总数。具体代码如下：

```
#include <omp.h>
#include <stdio.h>
int main()
{
    int nthreads, tid;
    omp_set_num_threads(16);
#   pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf("Hello World from OMP thread %d\n", tid);
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads is %d\n", nthreads);
        }
    }
}
```

编译运行结果如**图 1**所示，可以观察到，在给定 16 个线程的情况下，线程编号从 0 到 15，且线程 0 输出了线程总数。并且输出的顺序并非按照线程编号的顺序，这是因为线程的创建和执行是并行的，因此输出的顺序是不确定的。

图 1: 多线程输出线程号和线程数

> 要求 3
>
> 学习 for 多线程并行

下面的代码给出了使用 for 多线程并行计算数组元素和的简单实现，利用`reduction`规约完成，更复杂的矩阵乘法实现在后面在报告后面的部分给出。

首先定义一个数组 a，包含 100 个元素，然后使用循环将数组中的元素依次初始化为 1，2，3，...，100。接下来使用 OpenMP 并行计算数组元素的总和。使用`#pragma omp parallel for`指令将 for 循环变成一个并行计算的任务，其中第二个参数`reduction(+:sum)`告诉 OpenMP 对 sum 变量进行求和操作，并使用重复加法算法保证线程安全。

其中对于线程安全的实现主要通过`reduction`完成，它可以帮助我们实现并行任务中的归约操作。在并行任务中，某些变量需要被多个线程访问和修改，这时就需要对这些变量进行归约，即将多个线程对变量的计算结果合并成一个最终结果。

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int i, n = 100, sum = 0;
    int a[n];
    // 初始化数组
    for (i = 0; i < n; i++) {
        a[i] = i + 1;
    }
    // 并行计算数组元素的总和
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++) {
        sum += a[i];
    }
    printf("The sum is: %d\n", sum);
    return 0;
}
```

结果如图 2所示，正确的求出了 1-100 的和，即 5050，说明了`reduction`规约的正确性。

图 2: for 多线程并行计算数组元素和

要求 4

学习 while 多线程并行，实现全局共享变量存取

如下，实现了使用 while 多线程并行。下面的代码中的全局共享变量是 i，该变量被所有线程共享，并且在 while 循环中被多个线程同时访问和修改。

每个线程在访问并修改 i 变量之前必须确保其他线程不同时访问和修改相同的变量 i 。

使用了 `#pragma omp atomic` 将 i 变量的递增操作转换为原子操作。这将保证 i 变量被原子地递增，从而避免多个线程同时读取和增加 i 变量的值所带来的问题。最后，在同步点处同步 i 的值，使用了 `#pragma omp flush(i)` 指令，它确保对 i 进行可见性刷新，这个过程确保任何一个线程都可以读取到前一个线程写入的新值。

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  #pragma omp critical
  printf("Thread %d is running\n",tid);

  #pragma omp barrier

  #pragma omp while
  while(i < n)
  {
      #pragma omp atomic
      i++;
      #pragma omp critical
      printf("Thread %d says i=%d\n",tid,i);

      #pragma omp flush(i)
  }
}
```

程序运行结果如**图 3**所示。可以看出，完成了 while 在多线程并发的使用，输出时，线程的顺序是不确定的，但是输出的 i 的值是递增的，这说明了 while 的多线程并行是成功的。

```
❯ cd "/root/Desktop/EXP
 ./while
Thread 4 is running
Thread 5 is running
Thread 3 is running
Thread 2 is running
Thread 1 is running
Thread 0 is running
Thread 0 says i=1
Thread 0 says i=2
Thread 1 says i=3
Thread 1 says i=4
Thread 2 says i=5
Thread 2 says i=6
Thread 3 says i=7
Thread 3 says i=8
Thread 4 says i=9
Thread 5 says i=10
```

图 3: while 多线程并行

> 要求 5
>
> 编程实现大规模向量的并行计算

以下代码为使用 OpenMP 实现矩阵之间乘法的核心代码块。调用了`omp_get_wtime()`包围了核心代码块，以便计算程序运行时间。在核心代码块中，使用了`omp_parallel`和`omp_for`来实现多线程并行，其中`omp_parallel`用于创建线程，`omp_for`用于指定需要并行的代码块。

在`omp_for`中，通过`private`来限制变量的作用域，从而避免了多线程对同一变量的读写冲突。具体而言，`i`、`j`和`k`是循环变量，因此需要在每个线程中单独创建。在计算矩阵乘法时，将矩阵的每一行分配给不同的线程进行计算。对于每一个进程而言，其任务是将矩阵 A 的第 i 行与矩阵 B 的第 j 列相乘，然后将结果累加到矩阵 C 的第 i 行第 j 列。因此，在每一个线程的局部计算结束后，便得到了矩阵 C 的一部分，最后将这些部分合并即可得到最终结果。下述代码以 8 线程为例，具体代码如下：

```
1  // 8线程并行
2  start = omp_get_wtime();
3
4  #pragma omp parallel num_threads(8)
5  {
6  #pragma omp for private(i, j, k)
7      for (i = 0; i < N; i++)
8      {
9          for (j = 0; j < N; j++)
10         {
11             *(c + i * N + j) = 0;
12
13             for (k = 0; k < N; k++)
14             {
15                 *(c + i * N + j) += *(a + i * N + k) * *(b + k * N + j);
16             }
17         }
```

```
    }
}

end = omp_get_wtime();
cpu_time_used = end - start;
ratio = serial_time_used / cpu_time_used;

printf("8线程并行计算时间:%f, 加速比:%f\n", cpu_time_used, ratio);
```

实验结果如图 4所示。观察结果可以发现，随着线程数的增加，在线程数量较少时，加速比可以看成线性增长，而档线程数为 6、8、16 时，加速比接近 4.5，不再增长。出现这样的实验结果，是由于我分配给这台实验机器的虚拟 CPU 核心数为 6，因此当线程数接近或者超过 6 时，在计算任务较长，CPU 核心满载时，线程数的增加并不能带来更多的加速比，只会趋近于理论最大加速比 6。

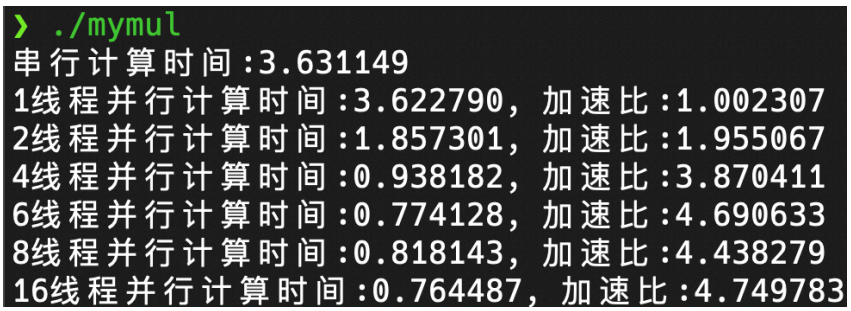

图 4: 矩阵乘法计算实验结果