实验：用 OpenMP 实现线程级并行

徐薪-519021910726

Exercise 1：向量加法 Vector Addition

1.

答：从运行时间可以看出，method_2()所用的时间小于 method_1()所用的时间。method_1()的运行时间受到 false sharing 的影响，使得运行时间较长。由于每个 core 都有自己独立的缓存空间（cache），所以当某个 cache block 里面的共享变量被修改时，所有 core 相应的 cache block 都需要同步这个修改，这样就造成了性能上的延迟。在 method_1()中，各个线程写入的数据是相邻的（比如线程 0 和线程 1 分别写入 z[0]和 z[1]，z[0]和 z[1]相邻，且在同一个 cache block 中），当线程 0 修改 z[0]时，z[0]所在的 cache block 就被标记为 invalid，并且线程 0 拥有 cache line 的主导权。在线程 1 要写入 z[1]之前，需要争夺 cache line 的主导权并且需要 update z[0]和 z[1]所在的 cache block，这些操作都会增加性能的延迟时间。

2.

答：我的 method_3 达到了 method_2 同等的性能，运行结果如下：

```
root@xin-VirtualBox:/mnt/shared/Computer_Organization/lab6/lab06_code# ./v_add 3
 1 thread(s) took 3.514796 seconds
 2 thread(s) took 2.202915 seconds
 3 thread(s) took 2.102223 seconds
 4 thread(s) took 2.098050 seconds
root@xin-VirtualBox:/mnt/shared/Computer_Organization/lab6/lab06_code# ./v_add 2
 1 thread(s) took 3.480778 seconds
 2 thread(s) took 2.330161 seconds
 3 thread(s) took 2.109285 seconds
 4 thread(s) took 2.071966 seconds
```

实验代码如下：

```
void method_3(double* x, double* y, double* z) {
#pragma omp parallel
{
  // your code here:
  int num = omp_get_num_threads();
  int id = omp_get_thread_num();
  int step = (ARRAY_SIZE + num - 1) / num;
  for(int i = id * step; i < (id + 1) * step && i < ARRAY_SIZE; ++i) {
    z[i] = x[i] + y[i];
  }

}

}
```

Exercise 2: Dot Product

1.

答：运行结果如下：

```
root@xin-VirtualBox:/mnt/shared/Computer_Organization/lab6/lab06_code# ./dotp
 1 thread(s) took 29.511258 seconds
 2 thread(s) took 41.311123 seconds
 3 thread(s) took 57.468966 seconds
 4 thread(s) took 72.201525 seconds
```

我们可以发现，线程的数目越多，性能越差。原因是在这个程序中，临界区包含了本应让各个线程独立计算的乘法运算。所以，在程序执行的过程中，每次只有一个线程进入临界区，其他线程只能在临界区外等待，从而每次只能执行一个元素的乘法，与单线程的运行无异。并且，线程进入临界区和离开临界区都会带来额外的开销（例如上下文切换），所以，在该程序下，线程数目越多，额外开销越大，从而使得性能越差。

2.

答：修改的代码如下：

```
double dotp_2(double* x, double* y) {
    double global_sum = 0.0;
    #pragma omp parallel
    {
      // your code here: modify dotp_1 to improve performance

        double local_sum = 0.0;
        #pragma omp for
        for(int i=0; i<ARRAY_SIZE; i++) {
            local_sum += x[i] * y[i];
        }

        #pragma omp critical
        global_sum += local_sum;

    }
    return global_sum;
}
```

运行结果如下：

```
root@xin-VirtualBox:/mnt/shared/Computer_Organization/lab6/lab06_code# ./dotp
 1 thread(s) took 3.446827 seconds
 2 thread(s) took 1.749165 seconds
 3 thread(s) took 1.255142 seconds
 4 thread(s) took 0.965175 seconds
```

由运行的结果可知，线程的数量越多，性能越好，这符合多线程运行的规律。同时，和 1.中的结果对比可知，在相同的线程数的情况下，dotp_2 的运行时间远远小于 dotp_1 的运行时间。

3.

答：reduction 语句的作用为：reduction 子句为变量指定一个操作符，并且每个线程都会创建 reduction 变量的私有拷贝，在 OpenMP 区域结束处，将各个线程的私有拷贝的值通过指定的操作符进行迭代运算，并赋值给原来的变量。相当于编译器自动完成了 dotp_2 的优化。dotp_3 的运行结果如下：

```
root@xin-VirtualBox:/mnt/shared/Computer_Organization/lab6/lab06_code# ./dotp
 1 thread(s) took 3.463048 seconds
 2 thread(s) took 1.720843 seconds
 3 thread(s) took 1.225277 seconds
 4 thread(s) took 0.968146 seconds
```

与 dotp_1、dotp_2 的运行结果对比可知，dotp_3 的运行时间与 dotp_2 的运行时间差不多，并且远低于 dotp_1 的运行时间。所以性能：dotp_1 < dotp_2 = dotp_3。