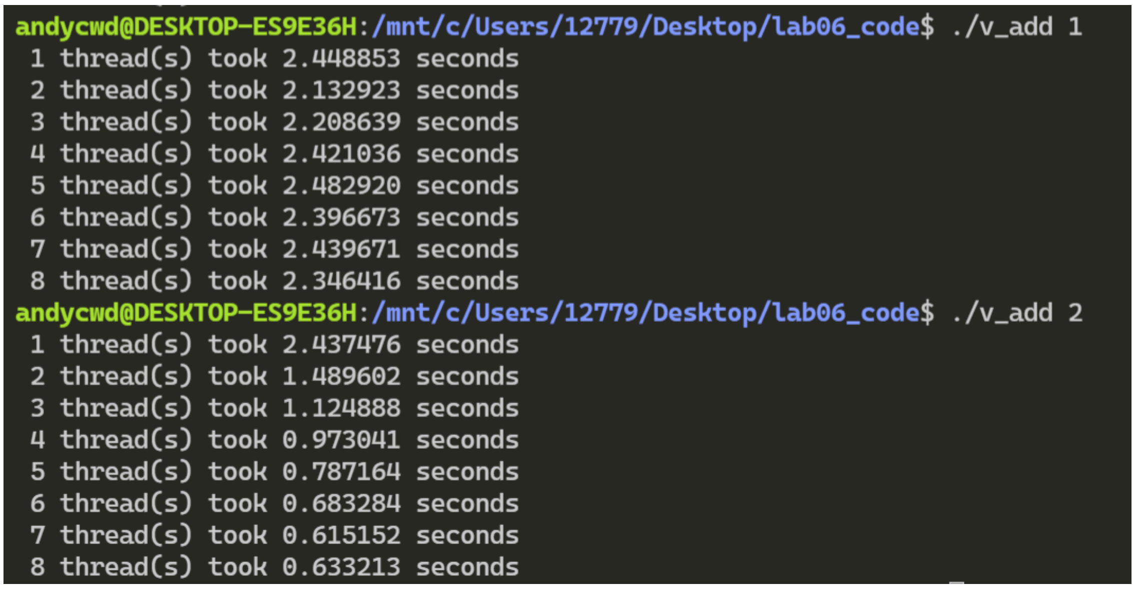
实践6 实验报告

陈文迪 519021910071

Exercise 1：向量加法 Vector Addition

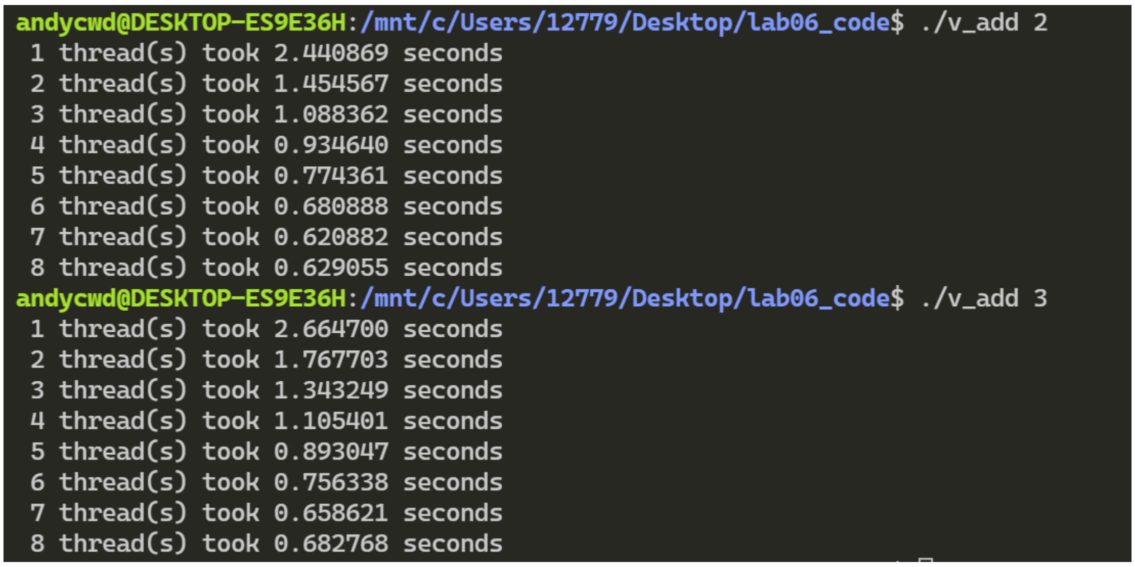
1. **运行结果：**



```
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab06_code$ ./v_add 1
 1 thread(s) took 2.448853 seconds
 2 thread(s) took 2.132923 seconds
 3 thread(s) took 2.208639 seconds
 4 thread(s) took 2.421036 seconds
 5 thread(s) took 2.482920 seconds
 6 thread(s) took 2.396673 seconds
 7 thread(s) took 2.439671 seconds
 8 thread(s) took 2.346416 seconds
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab06_code$ ./v_add 2
 1 thread(s) took 2.437476 seconds
 2 thread(s) took 1.489602 seconds
 3 thread(s) took 1.124888 seconds
 4 thread(s) took 0.973041 seconds
 5 thread(s) took 0.787164 seconds
 6 thread(s) took 0.683284 seconds
 7 thread(s) took 0.615152 seconds
 8 thread(s) took 0.633213 seconds
```

可以看到，`method_1()`并没有很好地利用多线程的优势，增加线程后运行速度并没有加快，与`method_2()`相比速度较慢。`method_1()`运行较慢的主要原因是因为其采用的任务分解模式让不同线程所访问的数据是相邻的，这就会导致假共享的问题：缓存是按块分割的，为了保证各线程的缓存一致，当一个线程修改了某一缓存块中的数据后，其他线程由于需要访问相邻元素，需要额外花费不必要的时间进行缓存更新，导致开销，线程数越多、单次写导致的开销越大。

2. **测试结果：**



```
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab06_code$ ./v_add 2
 1 thread(s) took 2.440869 seconds
 2 thread(s) took 1.454567 seconds
 3 thread(s) took 1.088362 seconds
 4 thread(s) took 0.934640 seconds
 5 thread(s) took 0.774361 seconds
 6 thread(s) took 0.680888 seconds
 7 thread(s) took 0.620882 seconds
 8 thread(s) took 0.629055 seconds
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab06_code$ ./v_add 3
 1 thread(s) took 2.664700 seconds
 2 thread(s) took 1.767703 seconds
 3 thread(s) took 1.343249 seconds
 4 thread(s) took 1.105401 seconds
 5 thread(s) took 0.893047 seconds
 6 thread(s) took 0.756338 seconds
 7 thread(s) took 0.658621 seconds
 8 thread(s) took 0.682768 seconds
```

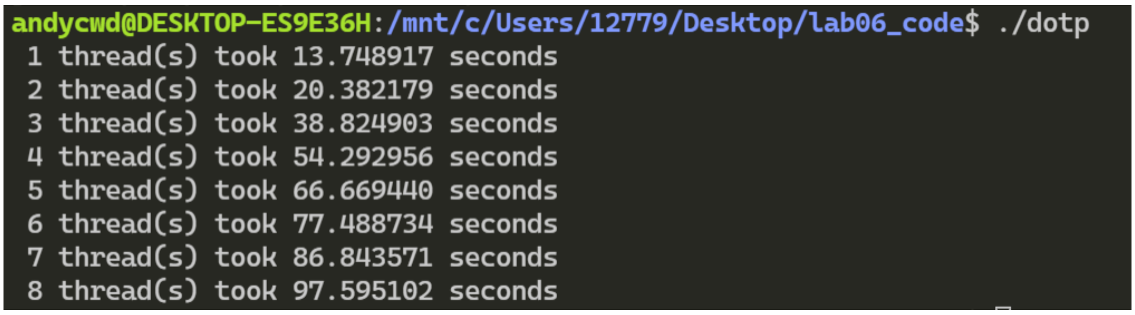
从结果中可以看出`method_3()`的性能相较于`method_1()`有很大提升，虽然与`method_2()`还有一点差距，但线程数比较多的情况下差距已经非常小。

实现代码：

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

Exercise 2: Dot Product

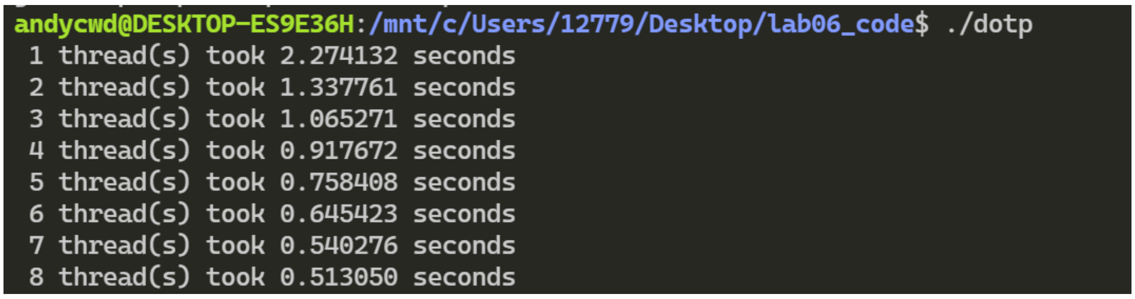
1. **测试结果：**



```
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab06_code$ ./dotp
 1 thread(s) took 13.748917 seconds
 2 thread(s) took 20.382179 seconds
 3 thread(s) took 38.824903 seconds
 4 thread(s) took 54.292956 seconds
 5 thread(s) took 66.669440 seconds
 6 thread(s) took 77.488734 seconds
 7 thread(s) took 86.843571 seconds
 8 thread(s) took 97.595102 seconds
```

从结果中确实可以观察到线程数越多，性能反而越差。其原因是因为每做一次乘法都需要进入临界区，而多个线程竞争临界区时只有单个线程可以进入，其他线程需要等待，这样反复的竞争会产生开销。

2. **测试结果：**



```
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab06_code$ ./dotp
 1 thread(s) took 2.274132 seconds
 2 thread(s) took 1.337761 seconds
 3 thread(s) took 1.065271 seconds
 4 thread(s) took 0.917672 seconds
 5 thread(s) took 0.758408 seconds
 6 thread(s) took 0.645423 seconds
 7 thread(s) took 0.540276 seconds
 8 thread(s) took 0.513050 seconds
```

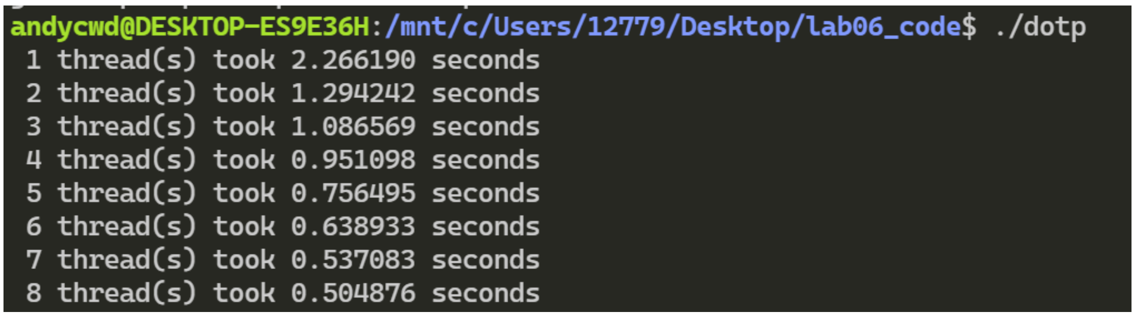
可以看到相比于`dotp_1()`，`dotp_2()`的性能得到了极大提高，并且随着线程数量的增多，性能逐步提高。

实现代码：

```
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;
        #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;
}
```

3. **测试结果：**

   `reduction`语句的作用是指明了对于该列表中的变量，在每个线程创建、初始化的同时需要创建该变量的私有副本，并在循环执行完成后，使用规约运算符将所有的副本变量归约到全局共享变量中。例如在本例中，将为每个线程创建一个global_sum的副本，并在循环结束之后将他们累加起来作为global_sum的值。



```
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab06_code$ ./dotp
 1 thread(s) took 2.266190 seconds
 2 thread(s) took 1.294242 seconds
 3 thread(s) took 1.086569 seconds
 4 thread(s) took 0.951098 seconds
 5 thread(s) took 0.756495 seconds
 6 thread(s) took 0.638933 seconds
 7 thread(s) took 0.537083 seconds
 8 thread(s) took 0.504876 seconds
```

   `dotp_3()`的性能与`dotp_2()`相近，与`dotp_1()`相比得到了极大的提升。