**实验：用OpenMP实现线程级并行**

**徐薪-519021910726**

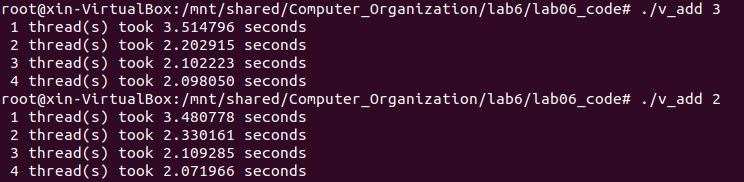
**Exercise 1：向量加法 Vector Addition**

**1.**

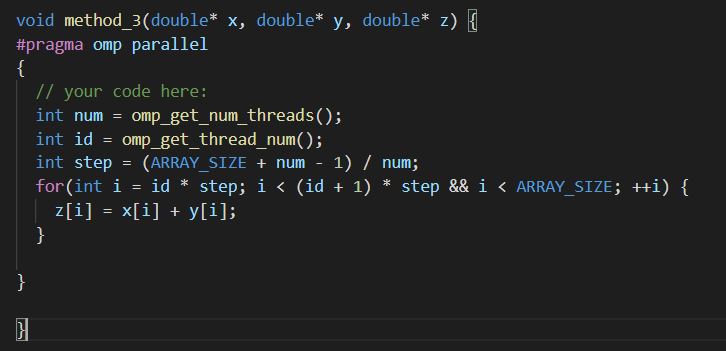
答：从运行时间可以看出，method\_2()所用的时间小于method\_1()所用的时间。method\_1()的运行时间受到false sharing的影响，使得运行时间较长。由于每个core都有自己独立的缓存空间（cache），所以当某个cache block里面的共享变量被修改时，所有core相应的cache block都需要同步这个修改，这样就造成了性能上的延迟。在method\_1()中，各个线程写入的数据是相邻的（比如线程0和线程1分别写入z[0]和z[1]，z[0]和z[1]相邻，且在同一个cache block中），当线程0修改z[0]时，z[0]所在的cache block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：



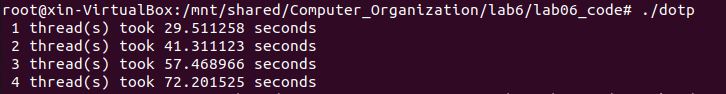
实验代码如下：



**Exercise 2: Dot Product**

**1.**

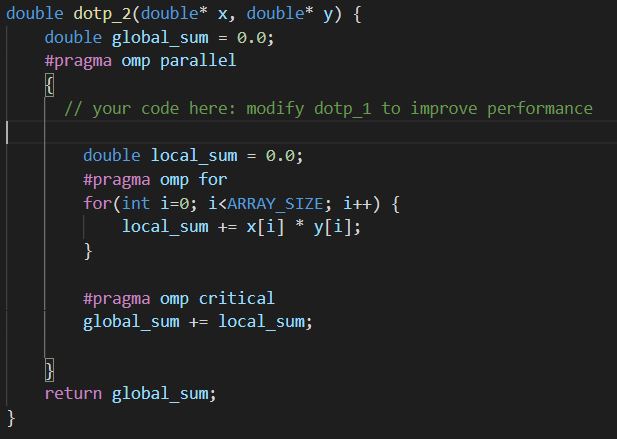
答：运行结果如下：



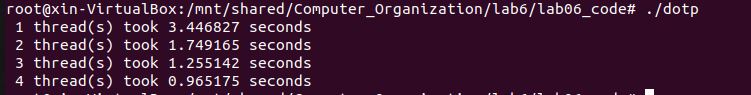
我们可以发现，线程的数目越多，性能越差。原因是在这个程序中，临界区包含了本应让各个线程独立计算的乘法运算。所以，在程序执行的过程中，每次只有一个线程进入临界区，其他线程只能在临界区外等待，从而每次只能执行一个元素的乘法，与单线程的运行无异。并且，线程进入临界区和离开临界区都会带来额外的开销（例如上下文切换），所以，在该程序下，线程数目越多，额外开销越大，从而使得性能越差。

**2.**

答：修改的代码如下：



运行结果如下：

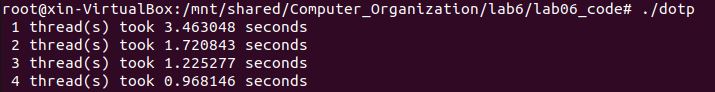


由运行的结果可知，线程的数量越多，性能越好，这符合多线程运行的规律。同时，和1.中的结果对比可知，在相同的线程数的情况下，dotp\_2的运行时间远远小于dotp\_1的运行时间。

**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：



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。