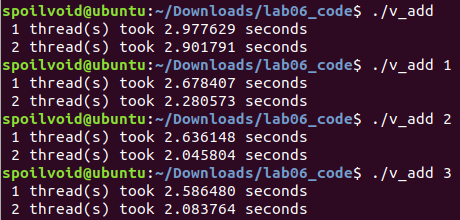
**Exercise 1: 向量加法 Vector Addition：**

（1）发现 method\_2()的平均工作时间比method\_1()短。因为method\_1()存在假共享问题（false sharing），当一个block在被载入hread0中且进行了修改，那么block整块变dirty并被告知其它线程，那么其它thread在处理同一block中的数据时由于被告知的信息需要重新载入，导致运行时间变长。

（2）mehtod\_3()与method\_2()的性能基本一致。



实现代码：

void method\_3(double\* x, double\* y, double\* z) {

#pragma omp parallel

{

// your code here:

{

int num = omp\_get\_num\_threads();

int tid = omp\_get\_thread\_num();

int length = ARRAY\_SIZE / num;

if(ARRAY\_SIZE % num == 0)

length += 1;

for (int i = tid \* length; i < ARRAY\_SIZE && i < (tid + 1) \* length; i++) {

z[i] = x[i] + y[i];

}

}

}

**Exercise 2: Dot Product**

（1）线程数目越多，性能越差。因为dotp\_1()函数每个thread进行对global\_sum的更新时，其余thread不能进入只能的等待，所以多线程时在临界区的大量切换导致了线程数目越多，反而性能越差。

（2）

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 = 0.0;

#pragma omp for

for(int i=0; i<ARRAY\_SIZE; i++) {

local\_sum += x[i] \* y[i];

}

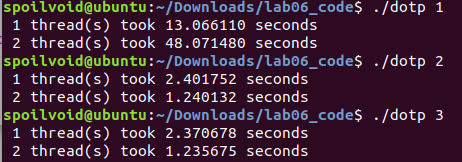
#pragma omp critical

global\_sum += local\_sum;

}

return global\_sum;

}



（p.s.修改了main函数使其读入一个额外的method参数，根据其参数选择对应计算函数）

根据结果可以看出，这样的修改使得程序获得了近10倍的性能改进，且使得多线程时，性能愈发地好。

（3）reduction语句告诉编译器：下面的for循环你要分成多个线程跑，但每个线程都要保存变量sum的拷贝，循环结束后，所有线程把自己的sum累加起来作为最后的输出。

测试结果如上图./dotp 3所示，比dotp\_1(),获得了近10倍的性能改进，且使得多线程时，性能愈发地好，且与dotp\_2()的性能相差无几。