计算机体系结构lab6

Exercise 1: 向量加法 Vector Addition

- **运行程序，比较它们的运行时间。为什么它们的执行时间不一样？method_1() 的运行时间受了什么因素的影响？**
  随着线程数增多，method_1的运行速度明显慢于method_2。method_1中相邻的内存地址被并行的线程同时写入，导致处理器核心缓存之间形成竞争关系，缓存命中率大幅下降，内存访问耗时增加，整体运算速度下降。

- **你的 method_3 达到 method_2 同等的性能了吗？ 贴出你的实现代码。**
  两者性能基本一致，实现代码如下：

```
void method_3(double* x, double* y, double* z) {
#pragma omp parallel
{
    int tn = omp_get_num_threads();
    int n = omp_get_thread_num();
    int lower = ((ARRAY_SIZE/tn)+1)*n;
    int upper = ((ARRAY_SIZE/tn+1)*(n+1)) < ARRAY_SIZE ?
                    (((ARRAY_SIZE/tn)+1)*(n+1)) : ARRAY_SIZE;
    for (int i = lower; i < upper; ++i)
        z[i] = x[i] + y[i];
    }
}
```

- 三种方法测试结果如下（单位：秒）：

线程数	method_1	method_2	method_3
1	2.291000	2.284000	2.532000
2	1.751000	1.430000	1.372000
3	1.443000	1.225000	1.191000
4	1.337000	1.172000	1.151000
5	1.503000	1.176000	1.127000
6	1.347000	1.179000	1.136000
7	1.300000	1.145000	1.142000
8	1.340000	1.218000	1.133000
9	1.317000	1.174000	1.152000
10	1.352000	1.177000	1.161000
11	1.352000	1.259000	1.174000
12	1.542000	1.281000	1.188000

Exercise 2: Dot Product

- **编译和运行程序 (make dotp and ./dotp). 观察一下，是不是线程的数目越多，反而性能越差? 分析原因?**
  运行结果如下：

  ```
  1 thread(s) took 14.838000 seconds
  2 thread(s) took 25.158000 seconds
  3 thread(s) took 24.770000 seconds
  4 thread(s) took 24.901000 seconds
  5 thread(s) took 25.657000 seconds
  6 thread(s) took 29.337000 seconds
  7 thread(s) took 33.812000 seconds
  8 thread(s) took 43.515000 seconds
  9 thread(s) took 51.551000 seconds
  10 thread(s) took 57.894000 seconds
  11 thread(s) took 59.940000 seconds
  12 thread(s) took 61.795000 seconds
  ```

随着线程数增加，性能反而更差。 `dotp_1` 函数中for循环主体处于临界区内，整个循环几乎没有并行部分，同时由于每个线程都需要频繁出入临界区，维护临界区的互斥条件导致大量的额外开销，因此随着线程增多，性能反而下降。

- **修改程序，让各个线程在计算部分点积时，不要将结果直接写入global_sum, 而是写入各自的私有变量 local_sum, 最后再通过临界区，汇总到 global_sum。 在函数 `dotp_2(double* x, double* y)` 中给出你改写的代码，并对比修改前后的性能。**
  修改代码如下：

```
double dotp_2(double* x, double* y) {
    double global_sum = 0.0;
    #pragma omp parallel
    {
        double private_sum = 0.0;
        #pragma omp for
        for(int i=0; i<ARRAY_SIZE; i++) {
            private_sum += x[i] * y[i];
        }
        #pragma omp critical
        global_sum += private_sum;
    }
    return global_sum;
}
```

  运行结果如下：

```
1 thread(s) took 2.686000 seconds
2 thread(s) took 1.454000 seconds
3 thread(s) took 1.054000 seconds
4 thread(s) took 0.930000 seconds
5 thread(s) took 0.831000 seconds
6 thread(s) took 0.771000 seconds
7 thread(s) took 0.807000 seconds
8 thread(s) took 0.760000 seconds
9 thread(s) took 0.729000 seconds
10 thread(s) took 0.720000 seconds
11 thread(s) took 0.721000 seconds
12 thread(s) took 0.750000 seconds
```

  可见，将临界区移出循环体，可以明显提高性能，且随着线程数量增加，运行速度越快。

- **解释一下 reduction 语句的作用，并测试使用归约语句改写后的并行点积计算的性能，对比它与 dotp_1 以及 dotp_2 的性能差别。**
  在 `dotp_3` 中， `reduction` 语句用于对 `global_sum` 变量按照 + 进行规约操作，即自动

将每个并行线程中的 `global_sum` 变量加法求和。
实际运行结果如下：

```
1 thread(s) took 2.731000 seconds
2 thread(s) took 1.413000 seconds
3 thread(s) took 1.003000 seconds
4 thread(s) took 0.880000 seconds
5 thread(s) took 0.779000 seconds
6 thread(s) took 0.722000 seconds
7 thread(s) took 0.733000 seconds
8 thread(s) took 0.682000 seconds
9 thread(s) took 0.644000 seconds
10 thread(s) took 0.633000 seconds
11 thread(s) took 0.633000 seconds
12 thread(s) took 0.663000 seconds
```

可见 `dotp_3` 性能略高于 `dotp_2` ，远高于 `dotp_1` 。