Lab8 - OpenMP Lab

UNIkeEN

> 本次实验使用环境是位于 Vmware 17 下的 Ubuntu 18.04，4 核
>
> `int num_threads = omp_get_max_threads();` 的结果为 4 ，尝试增加为 8 核运行实验用代码后，部分场景出现 Vmware 致命错误，故以下实验在线程数 1~4 下运行。

Exercise 1：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?**

  线程数量越多，`method_1()` 的运行速度将明显慢于 `method_2()` 。这是由于 `method_1()` 中，并行的线程同时对相邻内存地址上的数据进行写入操作，由于缓存块可以存放不止一个字的数据，根据假共享原理，不同处理器核心的缓存之间形成竞争关系，产生频繁的缓存行无效和重新加载，缓存命中率大幅下降，内存访问次数增加，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。**

  基本达到同等性能，实现代码如下：

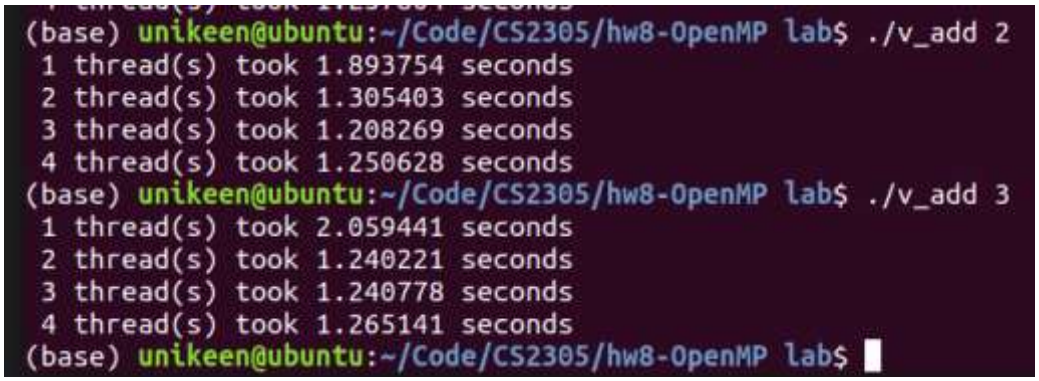
```
void method_3(double* x, double* y, double* z) {
#pragma omp parallel
    {
      // your code here:
      int t_num = omp_get_num_threads();
      int t_index = omp_get_thread_num();
      int start = (ARRAY_SIZE / t_num + 1) * t_index;
      int end = ((ARRAY_SIZE / t_num + 1) * (t_index + 1)) < ARRAY_SIZE ? ((ARRAY_SIZ
      for (int i = start; i < end; i++)
        z[i] = x[i] + y[i];
    }
}
```

  `t_num` 是总的线程数，`t_index` 是当前的线程号

- 三种方法运行结果如下（单位：秒）：

线程数	method_1	method_2	method_3
1	2.257912	1.893754	2.059441
2	2.064532	1.305403	1.240221

线程数	method_1	method_2	method_3
3	2.603888	1.208269	1.240778
4	2.709320	1.250628	1.265141



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 5.869541 seconds
  2 thread(s) took 38.488703 seconds
  3 thread(s) took 84.243149 seconds
  4 thread(s) took 109.446792 seconds
  ```

  线程的数目越多，性能越差（在本实验环境中，此现象很明显）。这是由于 `dotp_1` 中定义的 `critical` 区域是一个临界区，每次更新 `global_sum` 时，都必须只有一个线程独自执行（串行），这意味着在任何时候，所有线程都在排队完成对 `global_sum` 的更新。而 `dotp_1` 中每计算一次就更新 `global_sum` 更进一步加重了不同线程频繁进出临界区的现象，维护互斥造成大量开销，串行部分比例随着线程数量增多而增加，故随着线程增多，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。**

  实现代码如下：

  ```
  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;
}
```

修改后运行结果如下：

```
1 thread(s) took 3.405535 seconds
2 thread(s) took 1.777539 seconds
3 thread(s) took 1.363283 seconds
4 thread(s) took 1.263273 seconds
```

可以发现，将对 `global_sum` 的更新临界区移出循环体，明显提高了性能，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 OpenMP 中，`reduction` 指令用于并行地对数组进行某种形式的归约操作（例如，求和，求积等）

  在 `dotp_3` 中，`#pragma omp for reduction(+:global_sum)` 为每个线程创建一个 `global_sum` 的本地副本，每个线程会在其本地副本上进行加法操作，当所有线程都完成了他们的计算之后，OpenMP 会将对这些本地副本进行相加，并将结果存储到原始的 `global_sum` 变量中。

  运行结果如下：

  ```
  1 thread(s) took 3.299746 seconds
  2 thread(s) took 1.724061 seconds
  3 thread(s) took 1.247429 seconds
  4 thread(s) took 1.015746 seconds
  ```

  对比可知，`dotp_3` 性能略高于 `dotp_2`，远高于 `dotp_1`