并行计算 Lab1：OpenMP

1. 实验简介

OpenMP 是一个共享存储并行系统上的应用程序接口。它规范了一系列的编译制导、运行库例程和环境变量。

OpenMP 使用 `FORK-JOIN` 并行执行模型。所有的 OpenMP 程序开始于一个单独的主线程（Master Thread）。主线程会一直串行地执行，直到遇到第一个并行域（Parallel Region）才开始并行执行。

① `FORK`：主线程创建一队并行的线程，然后，并行域中的代码在不同的线程队中并行执行；

② `JOIN`：当诸线程在并行域中执行完之后，它们或被同步或被中断，最后只有主线程在执行。

2. 计算 Pi

2.1 代码分析

Pi (串行)

计算圆周率 π，原理为：$\pi \approx \Delta x \sum_{i=1}^{n} \frac{4}{1+x_i^2}$，其中：

- n 是 `num_steps`，表示将区间分为多少部分
- Δx 是 `step`，表示每部分的宽度，计算为 $\frac{1}{n}$
- x_i 表示第 i 个矩形的中点，计算为 $(i-0.5)\cdot\Delta x$

所以代码核心为一个 for 循环，且循环之间没有数据依赖。

Pi (使用并行域并行化的程序)

设置了两个线程，使用并行域，将原来的 for 循环拆分到两个线程中并行执行。

Pi (使用共享任务结构并行化的程序)

将计算任务分为两个线程，并使用 OpenMP 的 `#pragma omp for` 语句自动分配循环迭代给各个线程。

Pi (使用 private 子句和 critical 部分并行化的程序)

设置了两个线程，使用并行域，将原来的 for 循环拆分到两个线程中并行执行。`#pragma omp critical` 指定了需要保证互斥的代码段，求和操作互斥进行。

Pi (使用并行规约的并行程序)

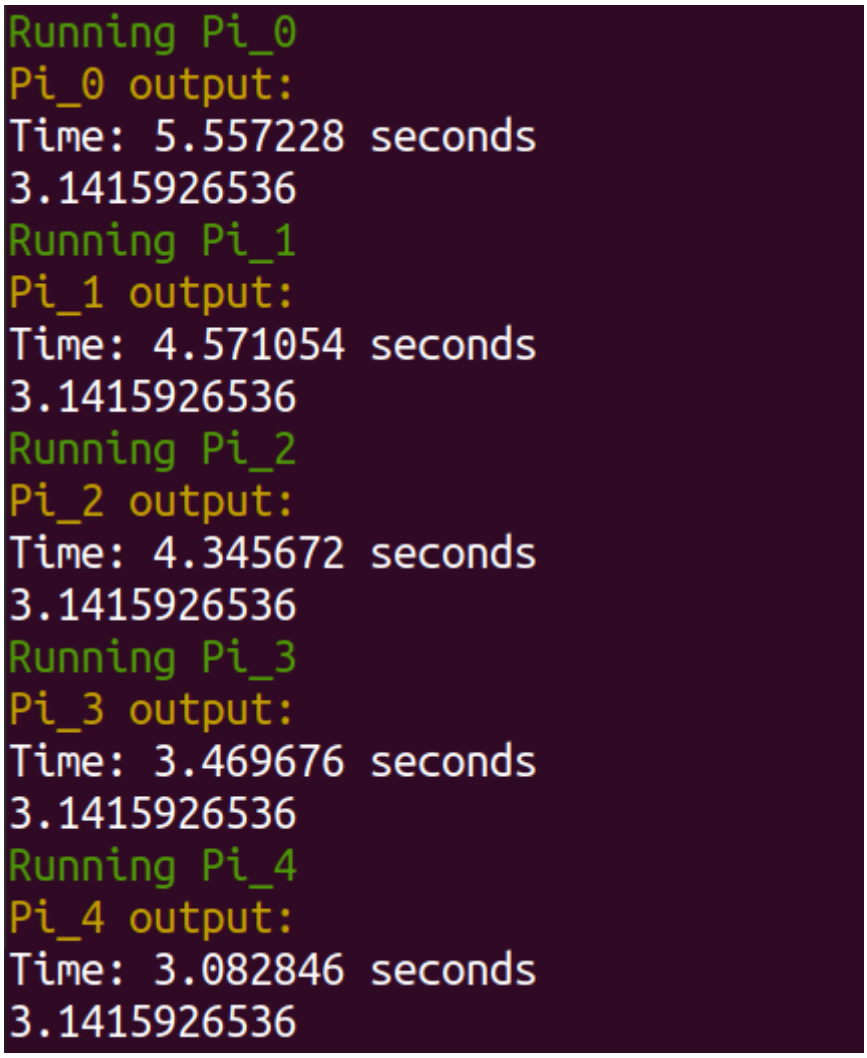
设置了两个线程，使用并行域，将原来的 for 循环拆分到两个线程中并行执行。最后对线程中所以 sum 进行+规约，并更新 sum 的全局值

2.2 修改部分

- 为所有代码引入手动输入 `num_steps`，输出运行时间
- Pi (使用并行域并行化的程序) 代码中，删除并行域内的 x 的声明，将 x 和 i 加入 private
- Pi (使用共享任务结构并行化的程序) 中，删除并行域内的 x 的声明，将 x 和 i 加入 private
- Pi (使用private子句和critical部分并行化的程序)，将 i 加入 private

- Pi (使用private子句和critical部分并行化的程序)，将 i 加入 private

2.3 运行结果



可以看到，并行程序虽然将主要的 for 循环拆分到两个线程并行执行，但是运行时间并没有减到串行程序的一半。

原因主要在于并行计算需要额外的线程创建、线程同步等操作，引入了额外的开销。

此外可以看出，四种并行方法中，使用并行规约的并行程序效率最高。

3. PSRS 排序

3.1 思想

(1) 均匀划分：将 n 个元素 A[1..n] 均匀划分成 p 段，每个 pi 处理 A[(i-1)n/p+1..in/p]

(2) 局部排序：pi 调用串行排序算法对 A[(i-1)n/p+1..in/p] 排序

(3) 选取样本：pi 从其有序子序列 A[(i-1)n/p+1..in/p] 中选取 p 个样本元素

(4) 样本排序：用一台处理器对 p^2 个样本元素进行串行排序

(5) 选择主元：用一台处理器从排好序的样本序列中选取 p-1 个主元，并播送给其他 pi

(6) 主元划分：pi 按主元将有序段 A[(i-1)n/p+1..in/p] 划分成 p 段

(7) 全局交换：各处理器将其有序段按段号交换到对应的处理器中

(8) 归并排序：各处理器对接收到的元素进行归并排序

3.2 核心代码

八个步骤中，最后三步较为复杂。

step 6 & step 7

首先，设置了划分后存储的数组：

```
int parts[NUM_THREADS][NUM_THREADS][N]; // 第i个线程，发往来自各个线程的有序数组
int count[NUM_THREADS][NUM_THREADS]; // 第i个线程，发往各个线程的数据数目
// 以上两个数组，除去第一维，即为某个线程局部数据划分出的数组，一个存数据，一个存个数
```

然后根据选出的主元进行划分：

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int pstart = id * N / NUM_THREADS;
    int pend = (id + 1) * N / NUM_THREADS;
    int index = 0; // 主元下标
    int j = pstart;
    while (j < pend && index < NUM_THREADS - 1) {
        if (A[j] < pivots[index]) {
            parts[id][index][count[id][index]] = A[j];
            count[id][index]++; // <= pivot[index] 的有 count[i][index] 个
            j++;
        }
        else {
            index++;
        }
    }
    // 大于最后一个主元的部分：此时 index = NUM_THREADS - 1
    while (j < pend) {
        parts[id][index][count[id][index]] = A[j];
        count[id][index]++;
        j++;
    }
}
// 至此，每个线程 pi 只需要处理 parts[][i][]
```

之后每个线程获得了 `NUM_THREADS` 个有序数组。

step 8

第八步则是对这些数组进行 Merge，因为是已经各自有序，所以不需要递归。

```
int result[NUM_THREADS][N];
int result_count[NUM_THREADS];
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int total_count = 0;
```

```
    int* tmp = (int*)malloc(N * sizeof(int));
    int tmp_count = 0;
    for (int index = 0; index < NUM_THREADS; index++) {
        // Merge: result[] 和 parts[index][id][]
        int i = 0, j = 0, k = 0;
        int A_count = total_count;
        int B_count = count[index][id];
        tmp_count = A_count + B_count;
        while (i < A_count && j < B_count) {
            if (result[id][i] <= parts[index][id][j]) tmp[k++] = result[id]
[i++];
            else tmp[k++] = parts[index][id][j++];
        }
        while (i < A_count) tmp[k++] = result[id][i++];
        while (j < B_count) tmp[k++] = parts[index][id][j++];
        total_count = tmp_count;
        for (int l = 0; l < tmp_count; l++) {
            result[id][l] = tmp[l];
        }
    }
    result_count[id] = total_count;
}
```

`for` 循环里面就是每次对两个数组进行 Merge.

3.3 结果

本程序需要手动输入待排序的元素个数以及线程数。

运行结果：

```
xxa@ubuntu:~/Desktop/Parallel-Computing-Labs/Lab1$ ./PSRS
please input the number of numbers:
27
please input the number of threads:
3
number of threads: 3
before sorting:
67 93 59 90 36 22 42 75 11 11 31 35 68 99 4 7 89 17 3 98 71 9 26 31 35 72 57
after sorting:
3 4 7 9 11 11 17 22 26 31 31 35 35 36 42 57 59 67 68 71 72 75 89 90 93 98 99
```

运行时间，对比串行 STL 库中的 sort

以下是 30000 个随机数字排序，并行采用 3 个线程：

```
sequential sorting:
Time: 0.003588 seconds
parallel sorting:
Time: 0.002282 seconds
```

可以看到，虽然并行方法采用了 3 个线程，但是时间并没有压缩到串行版本的 1/3，原因有如下几条：

- PSRS 虽然采用了并行技巧，但并非将所有步骤均并行化，仍存在串行操作。
- PSRS 为了将一些步骤并行化，做了一些额外的处理，尤其是做了很多数据移动，这是相对于串行版本多出来的开销

4. 附录

- `PB20061343_徐奥_实验一.pdf`：实验报告
- `Pi_0.cpp`：Pi (串行)
- `Pi_1.cpp`：Pi (使用并行域并行化的程序)
- `Pi_2.cpp`：Pi (使用共享任务结构并行化的程序)
- `Pi_3.cpp`：Pi (使用 private 子句和 critical 部分并行化的程序)
- `Pi_4.cpp`：Pi (使用并行规约的并行程序)
- `PSRS.cpp`：PSRS 排序
- `run.sh`：自动运行测试脚本