并行计算实验 1: OpenMP

傅申 PB20000051

1. 实验目的

使用 OpenMP 编写并行程序.

- 用 4 种不同并行方式的 OpenMP 实现 π 值的计算.
- 用 OpenMP 实现 PSRS 排序.

2. 实验环境

本次实验在我自己的笔记本上运行, 硬件参数如下:

CPU Intel i5-1035G1, 4 核 8 线程.
内存 8 GB DDR4 3200 MHz × 2.

相应的软件环境如下:

OS GNU/Linux 6.1.29-1-MANJARO x86_64.
OpenMP Verison 15.0.7-1.
编译器 LLVM clang version 15.0.2.
编译选项 `-Ofast -fopenmp`

3. 实验过程与结果

3.1. OpenMP 实现 π 值的计算

使用提供的 5 份代码, 其中第 4 份代码 (使用 `private` 子句和 `critical` 部分并行化的程序) 的并行域私有变量有误, 需要添加变量 `i` 作为私有变量, 即

```
#pragma omp parallel private(x, sum)
```

修改为

```
#pragma omp parallel private(i, x, sum)
```

设置计算步数为 10000000, 编译 5 份代码, 使用 `hyperfine` 工具分析运行时间, 得到如下结果:

线程数	串行	并行域	共享任务结构	private 子句和 critical 部分	并行规约
2	11.2 ms	14.4 ms	11.7 ms	12.3 ms	13.1 ms
4	—	5.8 ms	5.5 ms	4.8 ms	4.8 ms

可以看到, 随着线程数量的增加, 并行程序相对于串行程序的加速比逐渐增加, 在线程数为 4 时, 加速比分别为 1.93, 2.03, 2.33, 2.33.

3.2. OpenMP 实现 PSRS 排序

3.2.1. PSRS 实现

排序函数的定义为: `void psrs(int array[], unsigned int size, unsigned int threads)`

其中 `size` 为数组的大小, `threads` 为线程数量.

为了尽量负载均衡, 每个线程分配到的元素个数要尽可能地相等, 因此, 每个线程至多处理 $\lceil \frac{\text{size}}{\text{threads}} \rceil$ 个元素, 如下

```
if (threads > size)
    threads = size;
// Maximize load balance
unsigned int size_per_thread = (size + threads - 1) / threads;
// In case the last thread gets no/negtive work
if (size_per_thread * (threads - 1) > size) {
    size_per_thread--;
}
```

在定义好各个线程共享的内存后, 并行域开始, 进行 PSRS.

(1) 均匀划分: 各个线程找到对应子数组的起始内存地址作为 `subarray`, 并计算其子数组大小.

```
int  id       = omp_get_thread_num();
int *subarray = array + id * size_per_thread;
int  subsize  = size_per_thread;

if (id == threads - 1) {
    subsize = size - id * size_per_thread;
}
```

(2) 局部排序: 各个线程调用 `std::sort()` 对子数组进行排序.

(3) 选取样本: 各个线程将选取的样本存储在 `samples` 中.

```
for (int i = 0; i < threads; i++) {
    samples[id * threads + i] = subarray[subsize * i / threads];
}
```

(4) 样本排序和选择主元: 设置同步障, 主线程排序 `samples` 并选择主元存储在 `pivots` 中.

```
#pragma omp barrier
#pragma omp master
{
    sort(samples, samples + threads * threads);
    for (int i = 1; i < threads; i++) {
        pivots[i - 1] = samples[i * threads];
    }
}
```

(5) 主元划分: 设置同步障, 等待主线程选择完主元. 各个线程根据 `pivots` 将子数组划分为 p 段, 存储在三维数组 `split_arrays` 的对应位置, 并将对应段的大小存储在 `split_sizes` 中.

```
#pragma omp barrier
unsigned int index = 0;
fill(split_sizes[id], split_sizes[id] + threads, 0);
for (int i = 0; i < subsize; i++) {
    while (index < threads - 1 && subarray[i] > pivots[index])
        index++;
    split_arrays[id][index][split_sizes[id][index]++] = subarray[i];
}
```

(6) 全局交换: 设置同步障, 等待所有线程划分完成. 因为交换后新的子数组大小可能发生变化, 所以要重新计算各个子数组对应的内存地址和数组大小. 由于线程可以访问所有有序段, 所以计算新的子数组的位置和大小后交换就完成了.

```
#pragma omp barrier
// Calculating new subarray and subsize
subsize_after_swap[id] = 0;
for (int i = 0; i < threads; i++) {
    subsize_after_swap[id] += split_sizes[i][id];
}
subsize = subsize_after_swap[id];
#pragma omp barrier
subarray = array;
for (int i = 0; i < id; i++) {
    subarray += subsize_after_swap[i];
}
```

(1) 归并排序: 因为各个数据段已经是有序的了, 所以直接归并各个有序段即可, 没有必要排序. 归并 p 个数组的过程不再赘述.

3.2.2. 程序其他内容

程序的用法为 `./psrs [[线程数] [数组大小]]`. 当线程数没有提供时, 将默认使用 3 个线程; 当数组大小没有提供时, 将默认使用课本上的 27 个元素的数组. 若提供了数组大小, 则程序会随机生成一个相应大小的数组进行排序. 程序会调用 `psrs()` 和 `std::sort()` 进行排序, 并相应相应的运行时间和加速比. 若两者的排序结果不一致, 还会输出错误信息: `[Error] PSRS result is wrong.`

在 Makefile 中, 提供了两个模式 `psrs` 和 `debug`. `debug` 会输出更为细致的执行信息 (例如排序后的数组).

3.2.3. 运行结果

在默认情况下运行 `debug` 模式的编译结果, 输出如下:

```
$ ./debug
[Info] Time used by psrs():      0.709095 ms
[Info] Time used by std::sort(): 0.000704 ms
[Info] Speedup: 0.000992815
[Debug] PSRS result: 6 12 14 15 20 21 27 32 33 36 39 40 46 48 53 54 58 61 69 72 72
84 89 91 93 97 97
```

可以看到, PSRS 的运行结果没有问题.

运行 `psrs` 模式的编译结果, 可能的输出如下:

```
$ ./psrs 4 100000
[Info] Time used by psrs():      3.07472 ms
[Info] Time used by std::sort(): 6.15755 ms
[Info] Speedup: 2.00264
```

程序没有输出错误信息, 可以认为运行结果没有问题.

3.2.4. 运行时间分析

分别在不同的数据规模下以 4 线程运行 `./psrs` 十次, 统计平均运行时间, 如下:

数据规模	`psrs()`	`std::sort()`	加速比
1000	0.87 ms	0.04 ms	0.046
10000	1.40 ms	0.50 ms	0.357
100000	3.43 ms	6.56 ms	1.913
1000000	27.55 ms	73.88 ms	2.682
10000000	248.96 ms	790.10 ms	3.173

可以看到, 随着数据规模的增加, 并行算法的加速比越来越大, 性能提升越来越显著.

4. 实验总结

本次实验中,

- 我分析了各种计算 π 的并行程序的性能;
- 我使用 OpenMP 实现了 PSRS 算法, 并且在各个数据规模下分析了它的性能.