实验 1

PB16001715 陈思源

2020 年 7 月 14 日

1 实验题目

利用 MPI，OpenMP 编写简单的程序，测试并行计算系统性能。

2 实验环境

运行在服务器节点上，操作系统内核 Linux 3.10.0-862.el7.x86_64，使用 gcc 编译器，版本 4.8.5。处理器为 Intel 至强 E5-2620，基准频率 2.00GHz。

3 算法设计与分析

3.1 求素数个数

输入：正整数 n
输出：小于等于 n 的素数的个数
资源：p 个进程

解题思路

最简单的方法是对 1 到 n 的每个数，使用取模运算分别判断它们是否为素数，再统计素数的个数。有两处简单优化，第一，对于每个数 i，只用对 $2, 3, \ldots, \sqrt{i}$ 取模即可；第二，除了 2 以外，所有素数都是奇数。

实现步骤

1. 将从 3 到 n 的所有奇数进行域分解，将连续的 p 个数依次分配给每个进程。
2. 对于每个数 i，将 i 对 $2, 3, \ldots, \sqrt{i}$ 取模，若存在结果为 0，则 i 不是素数，否则 i 是素数。
3. 每个进程对自己分配到的数进行统计，保存在局部变量中。

4. 最后通过归约或临界区互斥同步将所有进程的局部变量求和，保存到主进程的 sum 变量中，再加上 2 这个唯一的偶素数。
5. 主进程的 sum 中的值即为小于等于 n 的素数的个数。

3.2 求 Pi 值

输入：正整数 n
输出：以 $\frac{1}{n}$ 为步长对 $\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$ 进行数值积分估算的 π 值。
资源：p 个进程

解题思路

用求和代替积分，可将求和任务均匀分配到每个进程，最后汇总。

实现步骤

1. 将 0 到 $n-1$ 进行域分解，将连续的 p 个数依次分配给每个进程。
2. 对分配到的数，每个进程对 $\frac{4}{1+\left(\frac{i+0.5}{n}\right))^2}$ 求和，保存在局部变量 $local$ 中。
3. 最后通过归约或临界区互斥同步将所有进程的局部变量求和，保存到主进程或共享的 pi 变量中。
4. 将 pi 再除以 n 即为 π 的估计值。

4 核心代码

4.1 求素数个数

```
for (i = rank*2+3; i <= n; i += size*2)
{
    if (isPrime(i))
    {
        local++;
    }
}
MPI_Reduce(&local, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0 && n >= 2)
{
    sum++;        //加上素数2
}
```

Listing 1: 求素数个数的 MPI 核心代码

```
# pragma omp parallel private(i, count)
{
    int id = omp_get_thread_num();
    for (i = id*2+3, count = 0; i <= n; i += NUM_THREADS*2)
    {
        if (isPrime(i))
        {
        count++;
        }
    }
    # pragma omp critical
    sum += count;
}
if (n >= 2) sum++;        //加上素数2
```

Listing 2: 求素数个数的 OpenMP 核心代码

4.2 求 Pi 值

```
for (i = rank; i < n; i += size)
{
    temp = (i + 0.5) * w;
    local = 4.0 / (1.0 + temp * temp) + local;
}
MPI_Reduce(&local, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Listing 3: 求 Pi 值的 MPI 核心代码

```
#pragma omp parallel private(i, x, sum)
{
    int id = omp_get_thread_num();
    for (i = id, sum = 0; i < num_steps; i += threads)
    {
        x = (i + 0.5) *step;
        sum += 4.0 / (1.0 + x * x);
    }
    #pragma omp critical
    pi += sum * step;
}
```

Listing 4: 求 Pi 值的 OpenMP 核心代码

5 实验结果

5.1 求素数个数

MPI 实现的结果

表 1: 求素数个数 MPI 实现的实验结果

(a) 运行时间 (s)

规模 \ 进程数	1	2	4	8
1,000	0.000043	0.000043	0.00008	0.000285
10,000	0.000689	0.000354	0.000291	0.00032
100,000	0.014351	0.006635	0.00515	0.00273
500,000	0.104079	0.05747	0.041945	0.02257

(b) 加速比

规模 \ 进程数	1	2	4	8
1,000	1	1	0.5375	0.1509
10,000	1	1.9463	2.3677	2.1531
100,000	1	2.16292	2.7866	5.2568
500,000	1	1.8110	2.4813	4.6114

OpenMP 实现的结果

表 2: 求素数个数 OpenMP 实现的实验结果

(a) 运行时间 (s)

规模 \ 进程数	1	2	4	8
1,000	0.000044	0.013683	0.013688	0.013673
10,000	0.000501	0.004293	0.023782	0.013792
100,000	0.009989	0.017679	0.016292	0.014938
500,000	0.173053	0.098524	0.048909	0.024816

(b) 加速比

规模 \ 进程数	1	2	4	8
1,000	1	0.0032	0.0032	0.0032
10,000	1	0.1167	0.0211	0.0365
100,000	1	0.5650	0.6131	0.6687
500,000	1	1.7565	3.5383	6.9734

5.2 求 Pi 值

MPI 实现的结果

表 3: 求 Pi 值 MPI 实现的实验结果

(a) 运行时间 (s)

规模 \ 进程数	1	2	4	8
1,000	0.000019	0.00003	0.000054	0.000331
10,000	0.000055	0.000044	0.00007	0.000305
50,000	0.000178	0.000119	0.000184	0.000347
100,000	0.000404	0.000196	0.000178	0.000268

(b) 加速比

规模 \ 进程数	1	2	4	8
1,000	1	0.6333	0.3519	0.0574
10,000	1	1.25	0.7857	0.1803
50,000	1	1.4958	0.9674	0.5130
100,000	1	2.0612	2.2697	1.5075

OpenMP 实现的结果

表 4: 求 Pi 值 OpenMP 实现的实验结果

(a) 运行时间 (s)

规模 \ 进程数	1	2	4	8
1,000	0.000028	0.013799	0.004077	0.019801
10,000	0.000185	0.013888	0.013894	0.013844
50,000	0.000875	0.012378	0.014015	0.023899
100,000	0.001738	0.01458	0.018274	0.022946

(b) 加速比

规模 \ 进程数	1	2	4	8
1,000	1	0.0020	0.0069	0.0014
10,000	1	0.0133	0.0133	0.0134
50,000	1	0.0707	0.0624	0.0366
100,000	1	0.1192	0.0951	0.0757

6 分析与总结

求素数个数和求 Pi 值的并行算法都需要在最后对得到的结果求和，串行部分开销为 $O(p)$，p 为线程数。只考虑计算时间，求素数个数的并行算法时间复杂度为 $O(n^{\frac{3}{2}}/p)$，而求 Pi 值的并行算法时间复杂度为 $O(n/p)$，因此理论上在随着规模增长，求素数个数算法的加速比要优于求 Pi 值算法的加速比。本次实验中，求素数个数的 MPI 实现取得了较好的结果，规模越大，多线程的加速效果越明显，当求素数个数的规模达到 $100,000$ 以上时 8 进程可以取得较好的加速比，但 OpenMP 实现使用临界区求和的开销很大，使得当规模较小时，多进程相比单进程的加速比反而远小于 1。求 Pi 值实验的规模较小，使得 MPI 实现和 OpenMP 实现的加速比都不理想，特别是 OpenMP 实现。

本次实验通过简单的程序让我熟悉了 MPI 和 OpenMP 的使用，了解了库的安装，程序的编译方法等，为接下来的实验打下基础。另外，OpenMP 使用临界区保证多进程操作的原子性开销很大，应尽量避免使用。