Lab1-report

- 实验内容

  分别用MPI和OpenMP实现以下两个求解过程。

  - 给定正整数n,编写程序计算出所有小于等于n的素数的个数。

    其中，n=1,000;10,000;100,000;500,000

  - 给定迭代次数n,编写程序计算Pi的值。

    其中，n=1,000;10,000;50,000;100,000

- 实验环境

  - 操作系统：Ubuntu 18.04 lts 64bits
  - 编译器
    - gcc 7.3.0 (for openmp)
    - mpi 3.2.1 (for mpi)
  - CPU: Inte i5-8250U CPU & 3.40Ghz * 8
  - Memory： 8GB

- 算法设计与分析

  在此次实验中，要求实现的是四个程序，分别用MPI和OpenMP实现求解素数和Pi，所以实现的过程可以分为以下四个部分。

  - **求解素数个数**

    素数是在大于1的自然数中，除了1和它本身以外不再有其他因数。所以求解小于n的所有素数时，需要一个二重循环来依次遍历实现，其中在内循环中，也就是判断某个数k<=m是否为宿舍时，可以证明只要判断到sqrt(k)即可，在遇到有个数整除k时跳出循环，此时k不是素数，否则素数的计数+1。这样求解过程的时间复杂度为$O(n^{\frac{3}{2}})$。

    具体在实现是每个线程根据其分配的起点和终点其调用求解素数的函数`primes_count(int x, int y)`，x和y分别是两个端点，函数实现思路如上述，代码见核心代码部分。

  - **模拟计算PI的值**

    模拟计算π的过程按照积分的近似求和公式

    $$\pi = \int_0^1 \frac{4}{1+x^2} \approx \sum_0^n \frac{4}{1+(\frac{i+0.5}{N})^2} \cdot \frac{1}{N}$$

    可以按照一个迭代次数为n的循环进行实现。

    具体在实现多线程并行时，计算Pi的模块函数提供一个接口，接受参数起点和终点和N，然后在for循环中实现即可。实现函数`simulate_pi(int start, int end, int num_steps)`的代码见下一部分。

  - **MPI并行实现**

在用MPI实现并行时，首先要调用MPI函数`MPI_Init(..)`进入MPI环境并完成所有的初始化工作，然后，调用`MPI_Comm_rank`函数获得当前进程在指定通信域中的编号，将自身与其他程序区分。然后要获取指定通信域的进程数，调用`MPI_Comm_size`函数获取指定通信域的进程个数，确定自身完成任务比例。如果返回的rank为0的话说明是主线程，在我的实现中，其任务是负责从命令行接收指定迭代次数N，然后通过`MPI_Send`函数将这一Int数值广播到其他线程，同时自己也会执行1/m(m是线程数)的任务， 如果rank不为0，那么该线程就要滴调用`MPI_Recv`函数等待从Id为0的线程接收参数N，接收到之后根据其rank值可以计算出相应的计算范围，计算完成后发送后rank为0的线程，同时为0的线程也会在Recv的状态等待接收所有其他线程的计算结果，累加求和就可进行汇总得到总的计算结果。最后通过`MPI_Finalize`函数从MPI环境中退出。

- **OpenMP**并行实现

  OpenMP的并行是通过for循环并行化来实现的。omp中使用parallel制导指令标识代码中的并行段，在代码段中调用`omp_get_thread_num()`获取当前线程id号，然后指定for制导语句，for制导语句是将for循环分配给各个线程执行，这里要求数据不存在依赖。最后再对各个线程的计算结果进行统计即可。

- 核心代码

  - 素数求解函数

```
void primes_count(int x, int y)
{
    int i, j, flag;
    if(x < 2)
        x = 2;
    for(i = x;i <= y;i++)
    {
        flag = 0;
        for(j = 2;j <= sqrt(i);j++)
            if(i % j == 0)
            {
                flag = 1;
                break;
            }
        if(flag == 0)
           h++;
    }
}
```

  - 模拟计算**PI**函数

```
void simulate_pi(int start, int end, long num_steps)
{
    double x;
    for (int i = start;i <= end; i++)
    {
         x = (i + 0.5) / (double)num_steps;
         sum += 4.0 / (1.0 + x*x);
    }
}
```

  - **MPI**执行部分

```
if (rank == 0)
   {
       n = atoi(argv[1]);
       double wtime = MPI_Wtime();
       // 广播数据n
       for(int j = 1; j < size; j++)
       {
           MPI_Send(&n, 1, MPI_INT, j, 1, MPI_COMM_WORLD);
       }
       start = n / size;
       start = start * rank;
       end = start + (n / size) - 1;
       // 调用执行函数，如果是求解PI提供乘simulate_count函数
       primes_count(start, end);

       int k;
       // 接收返回结果并汇总求和
       for(int j = 1;j < size;j++)
       {
           MPI_Recv(&k, 1, MPI_INT, j, 1, MPI_COMM_WORLD, &status);
           h += k;
       }
       wtime = MPI_Wtime() - wtime;
   }

   else
   {
       // 接收广播变量
       MPI_Recv(&n, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
       // 根据rank的值计算得到当前线程的执行任务范围
       start = n / size;
       start = start * rank;
       if(rank == size - 1)
           end = n;
       else
           end = start + (n/size)-1;
       primes_count(start, end);
       // 传回当前线程计算结果
       MPI_Send(&h, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
   }
```

- **OMP执行部分**

```
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
  double x;
  int id;
  id = omp_get_thread_num();
  sum[id] = 0;
  #pragma omp for
  for (i = 0;i < num_steps; i++)
  {
    // 如果是素数求解替换成相应代码即可。
    x = (i + 0.5) / (double)num_steps;
    sum[id] += 4.0 / (1.0 + x*x);
  }
}
```

- 实验结果

  由于有多个程序和多组参数，所以编写了脚本进行自动测试，测试时间结果 如下：

  - 计算素数-**MPI**
    - 运行时间

规模\线程数	**1**	**2**	**4**	**8**
1000	0.069ms	0.092ms	0.236ms	0.383ms
10000	1.508ms	1.164ms	0.655ms	0.503ms
100000	32.131ms	21.207ms	12.341ms	9.306ms
500000	358.575ms	212.801ms	111.913ms	80.701ms

    - 加速比

规模\线程数	**1**	**2**	**4**	**8**
1000	1	0.75	0.29	0.18
10000	1	1.30	2.30	3.00
100000	1	1.52	2.67	3.56
500000	1	1.68	3.22	4.48

  - 计算素数-**OMP**
    - 运行时间

规模\线程数	1	2	4	8
1000	0.103ms	0.225ms	0.445ms	0.722ms
10000	0.903ms	0.645ms	0.415ms	0.362ms
100000	23.725ms	14.483ms	8.451ms	6.163ms
500000	214.498ms	119.642ms	72.582ms	51.429ms

- 加速比

规模\线程数	1	2	4	8
1000	1	0.46	0.23	0.14
10000	1	1.40	2.19	2.49
100000	1	1.64	2.81	3.85
500000	1	1.79	2.96	4.17

- 计算**PI-MPI**
  - 运行时间

规模\线程数	1	2	4	8
1000	0.018ms	0.031ms	0.036ms	0.089ms
10000	0.188ms	0.107ms	0.098ms	0.089ms
50000	0.893ms	0.0473ms	0.0376ms	0.0354ms
100000	1.822ms	0.0929ms	0.0537ms	0.0499ms

  - 加速比

规模\线程数	1	2	4	8
1000	1	0.58	0.50	0.20
10000	1	1.76	1.91	2.11
50000	1	1.89	2.38	2.52
100000	1	1.96	3.39	3.65

- 计算**PI-OMP**
  - 运行时间

规模\线程数	1	2	4	8
1000	0.054ms	0.121ms	0.174ms	0.211ms
10000	0.176ms	0.160ms	0.148ms	0.133ms
50000	0.836ms	0.572ms	0.379ms	0.318ms
100000	1.759ms	1.127ms	0.775ms	0.573ms

- 加速比

规模\线程数	1	2	4	8
1000	1	0.45	0.31	0.26
10000	1	1.10	1.19	1.32
50000	1	1.46	2.21	2.63
100000	1	1.56	2.27	3.07

- 结果分析

  从实验结果来看，在n的规模较小时（比如1000），使用多线程并行并没有起到优化程序的效果，相反由于线程间通信的开销，导致加速比小于1并且线程数越多开销越大。

  在n的规模较大时，多线程的优化效果明显，随着n的增大，多线程并行的程序的加速比随之增大，并且可以推断会收敛到某一特定的值。

总结和收获

此次实验，首先是学会了在linux系统下配置mpi的并行编程环境，然后针对两个特定的迭代循环可并行的程序进行多线程优化，使用OpenMP和MPI两种并行库实现并行化，在规模不是很小时，并行优化后程序的执行时间显著减少，并且线程数越多，优化效果越好。