实验 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。实验题目包括给定范围内素数个数的计算和使用积分法求 Pi 的值。

实验环境：实验所使用电脑操作系统为 windows10 操作系统，IDE 为 Visual Studio2017，OpenMP 实验环境由 VS2017 提供，MPI 使用 MS-MPI，代码编写和测试均由 VS2017 实现。

实验所需基本语法知识：
MPI 部分：
MPI_Init：告知 MPI 系统进行所有必要的初始化设置。它是写在启动 MPI 并行计算的最前面的。使用方式：MPI_Init(int* argc_p,char*** argv_p);

通讯子（communicator）：MPI_COMM_WORLD 表示一组可以互相发送消息的进程集合。

MPI_Comm_rank:用来获取正在调用进程的通信子中的进程号的函数。

MPI_Comm_size:用来得到通信子的进程数的函数。

MPI_Send：阻塞型消息发送。其结构为：
int MPI_Send (void *buf, int count, MPI_Datatype datatype,int dest, int tag,MPI_Comm comm)
参数 buf 为发送缓冲区；count 为发送的数据个数；datatype 为发送的数据类型；dest 为消息的目的地址(进程号)，其取值范围为 0 到 np－1 间的整数(np 代表通信器 comm 中的进程数) 或 MPI_PROC_NULL; tag 为消息标签，其取值范围为 0 到 MPI_TAG_UB 间的整数; comm 为通信器。
MPI_Recv：阻塞型消息接收。结构为：
int MPI_Recv (void *buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm,MPI_Status *status)
参数 buf 为接收缓冲区；count 为数据个数，它是接收数据长度的上限，具体接收到的数据长度可通过调用 MPI_Get_count 函数得到；datatype 为接收的数据类型；source 为消息源地址(进程号)，其取值范围为 0 到 np－1 间的整数(np 代表通信器 comm 中的进程数)，或 MPI_ANY_SOURCE，或 MPI_PROC_NULL；tag 为消息标签，其取值范围为 0 到 MPI_TAG_UB 间的整数或 MPI_ANY_TAG；comm 为通信器；status 返回接收状态。

int MPI_Finalize (void)
退出 MPI 系统， 所有进程正常退出都必须调用。 表明并行代码的结束,结束除主进程外其它进程。串行代码仍可在主进程(rank = 0)上运行， 但不能再有 MPI 函数（包括 MPI_Init()）。

规约函数 MPI_Reduce()，将通信子内各进程的同一个变量参与规约计算，并向指定的进程输出计算结果
MPI_METHOD MPI_Reduce(
    _In_range_(!= , recvbuf) _In_opt_ const void* sendbuf, // 指向输入数据的指针

```
    _When_(root != MPI_PROC_NULL, _Out_opt_) void* recvbuf, //  指向输出数据的指针
    _In_range_(>= , 0) int count,                             //  数据尺寸
    _In_ MPI_Datatype datatype,                               //  数据类型
    _In_ MPI_Op op,                                           //  规约操作类型
    _mpi_coll_rank_(root) int root,                           //  目标进程号
    _In_ MPI_Comm comm                                        //  通信子
);
```

OpenMP 部分：
OpenMP 采用 fork-join 的执行模式。开始的时候只存在一个主线程，当需要进行并行计算的时候，派生出若干个分支线程来执行并行任务。当并行代码执行完成之后，分支线程会合，并把控制流程交给单独的主线程。
Compiler Directive 的基本格式：#pragma omp directive-name [clause[[,] clause]…]
directive-name 可以为：parallel, for, sections, single, atomic, barrier, critical, flush, master, ordered, threadprivate（共 11 个，只有前 4 个有可选的 clause）。

算法设计与分析：
素数计数：
首先判断一个数 n 是否是素数，可以通过测试从 2 到 sqrt(n)的所有整数是否能整除 n，若均不能整除，则 n 是素数。要把素数计数这个过程并行化，可以把整个数据按照并行度划分为多个数据块，然后在每个处理器并行对自己所负责的数据块进行素数计数。最后把计数结果求和就得到所有数据中素数的个数。对于数据的分块，在使用 MPI 的情形下，可以按照进程名设定步长来进行数据划分，在所有处理都完成后，使用 MPI_Reduce()函数来进行规约获取结果。在 OpenMP 的情形下，设定好线程数后，使用 omp_set_num_threads(NUM_THREADS);设定并行数，然后使用#pragma omp parallel for reduction(+:num)语句来把 for 循环中对每个单个数据的素数判断 for 循环进行并行化执行即可，要注意，语句中(+:num)意味着在并行处理完成后，要规约的变量是 num，且使用加法规约合并获取最后结果。

Pi 的计算：
首先，Pi 的计算可以使用以下公式积分获得：

$$\int_0^1 \frac{1}{1+x^2}dx = \arctan x\Big|_0^1 = \frac{\pi}{4} \Rightarrow \pi = \int_0^1 \frac{4}{1+x^2}dx$$

基于上述方法，在编程实现时，可以使用微分求和的方法逼近计算 Pi 的值，通过约定[0,1]区间上分块个数，在每个区间上选定代表值并计算 1/(1+x^2)乘上块的宽度并求和即可计算。MPI 情形下，要并行化上述过程，思路也是把计算小块面积并求和的过程并行化，具体来说，把求和区间按照并行度进行划分，每个处理器负责一部分数据的计算，。对于数据的分块，在使用 MPI 的情形下，可以按照进程名设定步长来进行数据划分，最后使用 MPI_Reduce()函数进行规约获取最终结果。这里我尝试把从命令行获取分块个数的操作推迟到并行化开始，使用一个处理器获取分块个数，再把这个数据广播给其他处理器。在使用 OpenMP 的情形下，我通过使用语句#pragma omp parallel sections reduction(+:sum) private(x,i)和语句

#pragma omp section 来开始进程的并行化，其中，每一句#pragma omp section 都对应了一个并行化线程的执行。注意第一个语句中，private 子句将一个或多个变量声明为线程的私有变量。每个线程都有它自己的变量私有副本，其他线程无法访问。即使在并行区域外有同名的共享变量，共享变量在并行区域内不起任何作用，并且并行区域内不会操作到外面的共享变量。

核心代码：
素数计数-MPI:

```
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    if (myid == 0)
    {
        printf("输入一个数字:");
        fflush(stdout);
        scanf_s("%d", &n);
        startwtime = MPI_Wtime();
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);   //将n值广播出去
    sum = 0;
    for (i = myid * 2 + 1; i <= n; i += numprocs * 2)
        sum += isPrime(i);
    mypi = sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);  //规约
    if (myid == 0)
    {
        printf("结果=%d\n", pi);
        endwtime = MPI_Wtime();
        printf("并行时间=%f\n", endwtime - startwtime);
    }
```

素数计数_OpenMP:

```
#pragma omp parallel for reduction(+:num)
    for (i = 2; i <= N; i++)
    {
        num += isPrime(i);
    }
    t2 = clock();
    printf("素数共有%d个\n", num);
    t5 = t2 - t1;
    printf("并行时间是%.9f\n", double(t5)/CLOCKS_PER_SEC);
```

Pi_MPI:

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Get_processor_name(processor_name, &proc_len);
//printf("Process %d of %d\n", my_rank, num_procs);
if (my_rank == 0) {
    scanf_s("%d", &n);
    printf("\n");
    start = MPI_Wtime();
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
sum = 0.0;
width = 1.0 / n;
//每个进程my_rank，计算4.0/(1.0+local*local)放入sum
for (i = my_rank; i<n; i += num_procs) {
    local = width * ((double)i + 0.5);
    sum += 4.0 / (1.0 + local * local);
}
mypi = width * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Pi_OpenMP:

```
#pragma omp parallel sections reduction(+:sum) private(x,i)
    {
#pragma omp section
        {
            for (i = omp_get_thread_num(); i<num_steps; i = i + NUM_THREADS)
            {
                x = (i + 0.5)*step;
                sum = sum + 4.0 / (1.0 + x * x);

            }
        }
```

```
#pragma omp section
        {
            for (i = omp_get_thread_num(); i<num_steps; i = i + NUM_THREADS)
            {
                x = (i + 0.5)*step;
                sum = sum + 4.0 / (1.0 + x * x);

            }
        }
#pragma omp section
        {
            for (i = omp_get_thread_num(); i<num_steps; i = i + NUM_THREADS)
            {
                x = (i + 0.5)*step;
                sum = sum + 4.0 / (1.0 + x * x);

            }
        }
#pragma omp section
        {
            for (i = omp_get_thread_num(); i<num_steps; i = i + NUM_THREADS)
            {
                x = (i + 0.5)*step;
                sum = sum + 4.0 / (1.0 + x * x);

            }
        }

    }
    pi = step * sum;
    end_time = clock();
    tol = end_time - start_time;
```

实验结果：

素数_MPI:

运行时间：

规模/进程数	1	2	3	4
1000	0.000046	0.001242	0.001402	0.002186
10000	0.000640	0.001846	0.001820	0.001909
100000	0.011430	0.010828	0.008157	0.012336
500000	0.100308	0.068094	0.060268	0.049799

加速比：

规模/进程数	1	2	3	4
1000	1	0.0370	0.0328	0.0211
10000	1	0.3467	0.3516	0.3368
100000	1	1.0556	1.4012	0.9266
500000	1	1.4731	1.6644	2.0183

素数计数_OpenMP:

运行时间：

规模/进程数	1	2	3	4
1000	0.00001	0.00100	0.00200	0.00300
10000	0.00300	0.00200	0.00400	0.00300
100000	0.04900	0.02900	0.002500	0.02200
500000	0.29700	0.02800	0.15800	0.16100

加速比：

规模/进程数	1	2	3	4
1000	1	0.01	0.005	0.003
10000	1	1.50	0.75	1
100000	1	1.69	24.5	2.227
500000	1	10.607	1.880	1.845

Pi_MPI:

运行时间：

规模/进程数	1	2	3	4
1000	0.000026	0.001366	0.001039	0.002268
10000	0.000104	0.001367	0.001403	0.002271
50000	0.000555	0.001709	0.001677	0.002311
100000	0.001071	0.001048	0.001684	0.002345

加速比：

规模/进程数	1	2	3	4
1000	1	0.0190	0.0250	0.0118
10000	1	0.0761	0.0743	0.0458
50000	1	0.3248	0.3309	0.2403
100000	1	1.0710	0.6360	0.5355

Pi_OpenMP:

运行时间：

规模/进程数	1	2	3	4
1000	0.00001	0.00100	0.00200	0.00200
10000	0.00001	0.00100	0.00200	0.00200
50000	0.00100	0.00100	0.00200	0.00300
100000	0.00200	0.00100	0.00300	0.00300

加速比：

规模/进程数	1	2	3	4
1000	1	0.010	0.005	0.005
10000	1	0.010	0.005	0.005
50000	1	1	0.5	0.333
100000	1	2	0.667	0.667

实验分析与总结：

MPI 是多主机联网协作进行并行计算的工具，当然也可以用于单主机上多核/多 CPU 的并行计算，不过效率低。它能协调多台主机间的并行计算，因此并行规模上的可伸缩性很强。缺点是使用进程间通信的方式协调并行计算，这导致并行效率较低、内存开销大、不直观、编程麻烦。

OpenMP 是针对单主机上多核/多 CPU 并行计算而设计的工具，换句话说，OpenMP 更适合单台计算机共享内存结构上的并行计算。由于使用线程间共享内存的方式协调并行计算，它在多核/多 CPU 结构上的效率很高、内存开销小、编程语句简洁直观，因此编程容易、编译器实现也容易。OpenMP 最大的缺点是只能在单台主机上工作，不能用于多台主机间的并行计算。

因此在数据量不大且单台计算机编程运行的情形下，使用 OpenMP 可能会更有效，在数据量较大且多台计算机合作并行运行时，就需要使用 MPI 或是 MPI+OpenMP。