# 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：

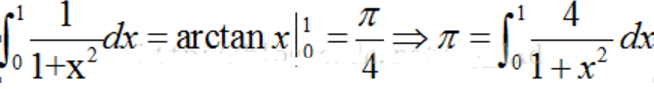
素数计数：

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：

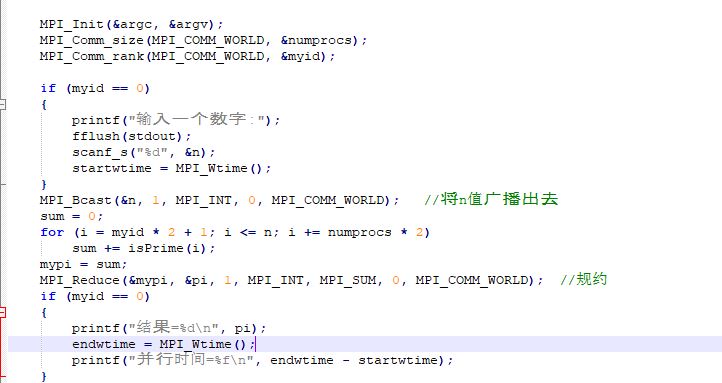


基于上述方法，在编程实现时，可以使用微分求和的方法逼近计算Pi的值，通过约定[0,1]区间上分块个数，在每个区间上选定代表值并计算1/(1+x^2)乘上块的宽度并求和即可计算。

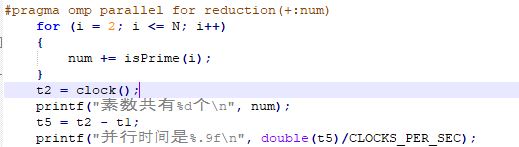
MPI情形下，要并行化上述过程，思路也是把计算小块面积并求和的过程并行化，具体来说，把求和区间按照并行度进行划分，每个处理器负责一部分数据的计算，。对于数据的分块，在使用MPI的情形下，可以按照进程名设定步长来进行数据划分，最后使用MPI\_Reduce()函数进行规约获取最终结果。这里我尝试把从命令行获取分块个数的操作推迟到并行化开始，使用一个处理器获取分块个数，再把这个数据广播给其他处理器。在使用OpenMP的情形下，我通过使用语句#pragma omp parallel sections reduction(+:sum) private(x,i)和语句#pragma omp section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:



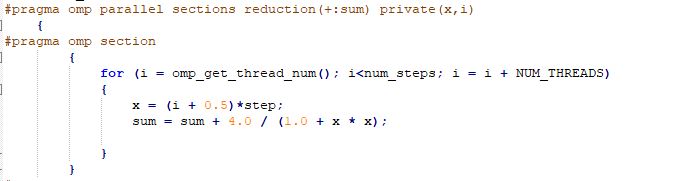
素数计数\_OpenMP:

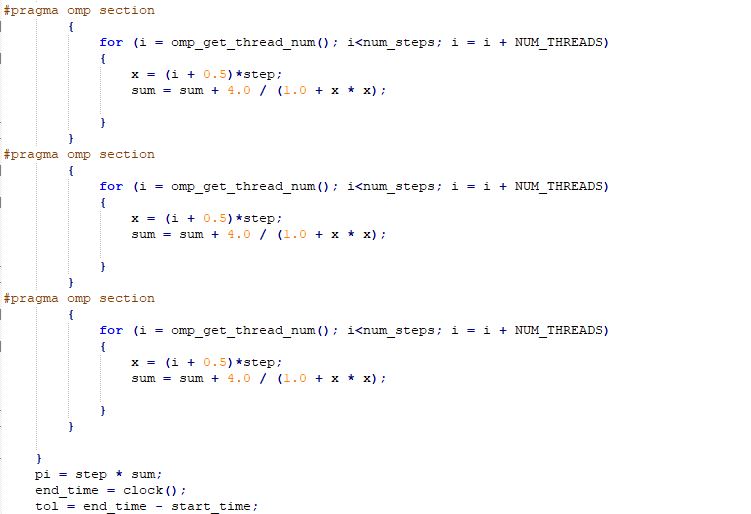


Pi\_MPI:



Pi\_OpenMP:





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。