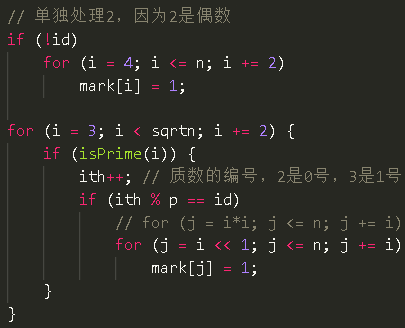
**并行计算第三次作业**

**5.9**

（1）关键代码

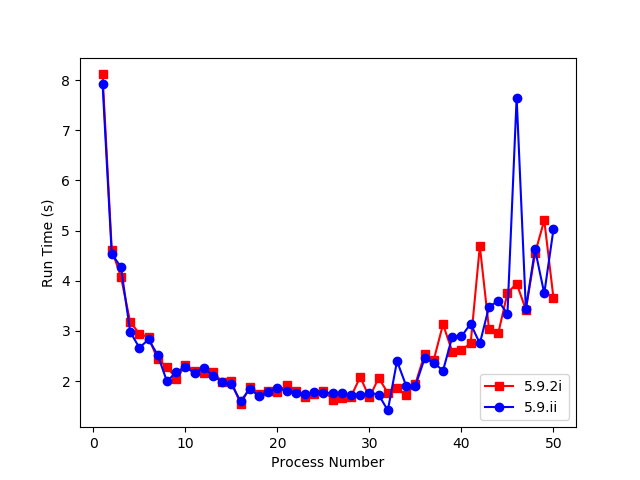


（2）思路解释

1. Mark[0]不再代表2，mark[2]代表2，mark[n]代表n，简化了实现；
2. 边找sieve边循环遍历mark，每找p个sieve就遍历一次mark数组；
3. 只检查奇数，所以2要单独处理；
4. MPI\_Reduce时需要在0号进程额外声明一个global\_mark数组用于接收结果，不能直接拿mark数组；mark数组是bool类型的，MPI\_Reduce中相应的参数改为MPI\_CHAR就好了；操作参数用MPI\_BOR和MPI\_LOR都行；
5. 第二个for循环里从i\*i开始标记更快，2i到i\*i之间的质数在之前已经被标记过了；
6. 在每个进程上计数，避免传输整个数组更快。

（3）实验结果

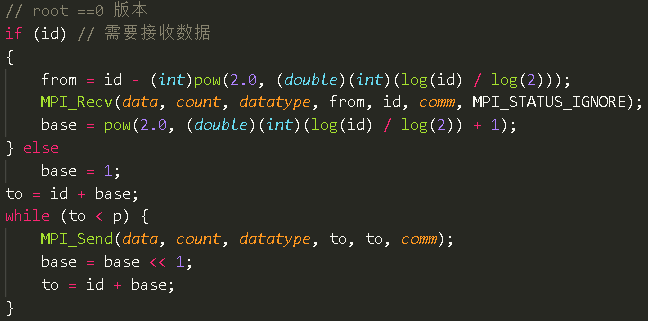
1. 计算1亿内的质数个数，分别用1-50个进程做实验，每次遍历mark分别从2i、ii开始标记，结果如下：



1. 计算的结果都是5761455个质数，2i情况下最快为16个进程时的1.535s，1-50个进程平均每次运行时间是2.649s，ii情况下最快为32个进程时的1.416s，平均运行时间是2.679s。
2. 2i和ii的运行时间基本上差不多，可能是因为ii情况需要多做一次乘法，少算的加法和标记次数的优势不大，数据规模也不是很大，实验环境的变化可能也有影响；ii情况的平均运行时间甚至还多些，主要是因为46个进程运行时突然变慢，大概是系统受影响了，是个异常点。

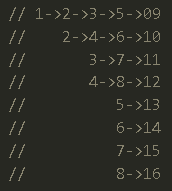
**6.10**

1. 关键代码（源数据进程号root一定等于0的版本）

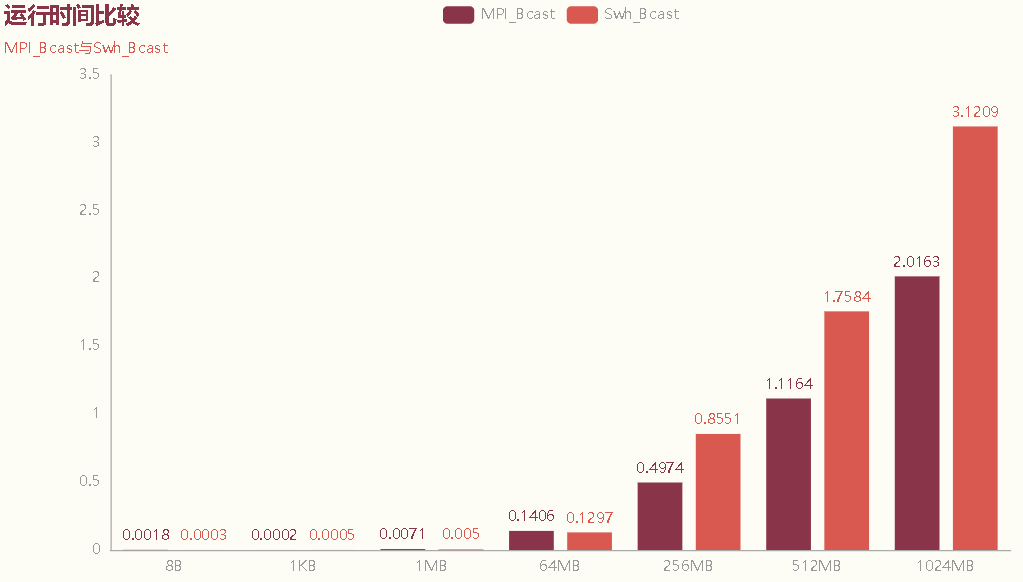


当root不一定为0时把root号进程映射到0号，把0号进程映射到root号即可，代码较长，这里不再贴出。

1. 思路解释
2. 利用Binomial Tree的思想，反向发送数据，如下图所示：1号发数据给2号，然后1、2号发给3、4号，然后1、2、3、4号发给5、6、7、8号。每轮发送的数据翻一番，一共只需要log p次发送，向上取整。

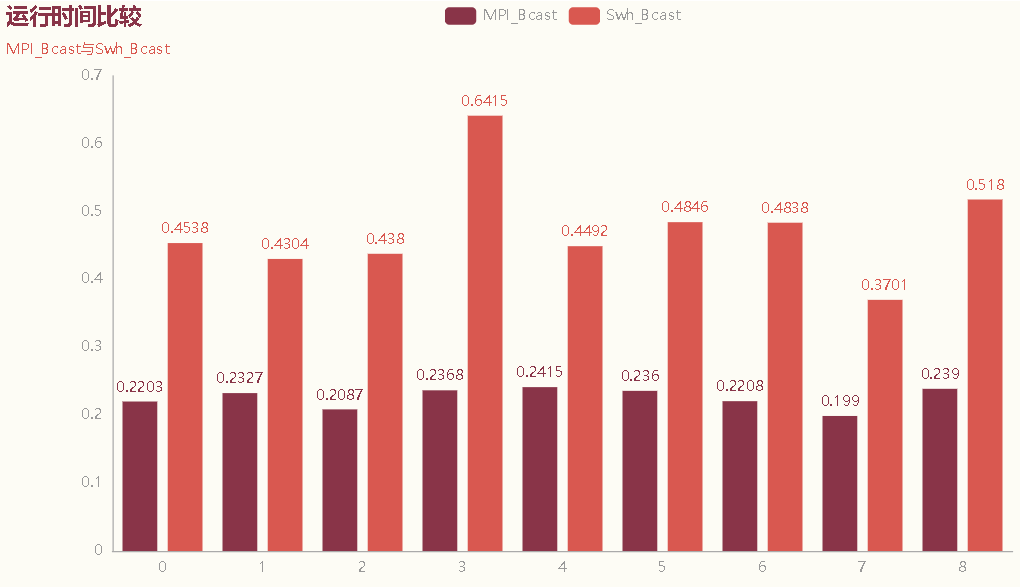


1. 实现的时候每个号码需要减一，因为进程号从0开始。
2. 上述思路是默认从0号进程开始发送数据的，所以如果要实现从任意一个进程比如root号进程发数据，则需要把root号进程看做0号进程，原来的代码中任何与0和root号进程有关的地方都需要改动，代码量多了两倍。
3. 实验
4. 用32个进程，每次都从0号进程传给其他31个进程，分别用两种方法传输8B、1KB、1MB、64MB、256MB、512MB、1GB数据，得到运行时间如下图所示，其中每种情况运行时间的单位写在横轴的括号里。

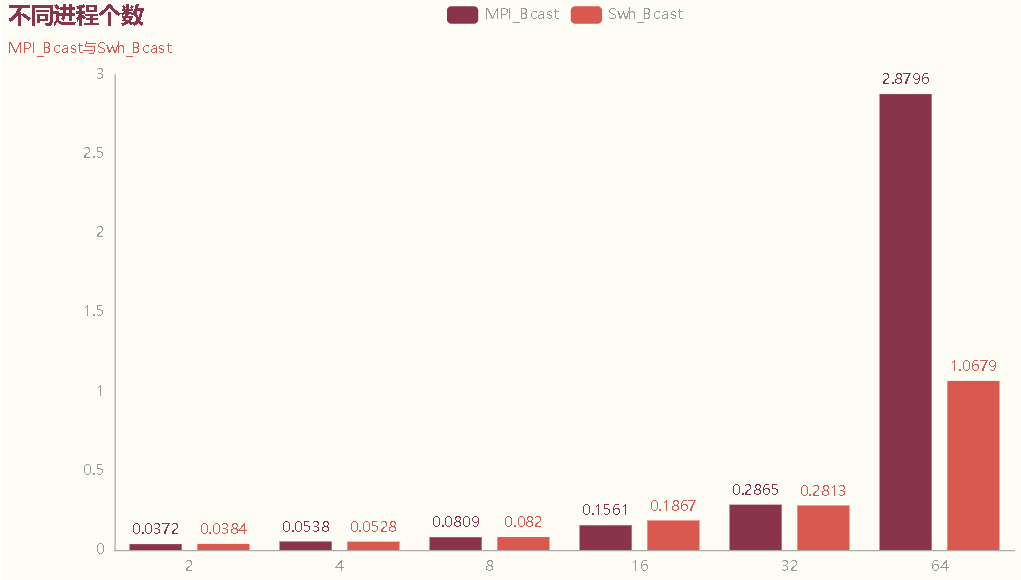


数据传输8B、1MB、64MB时我的实现稍快些，其他情况都慢些，有时候1GB时我的实现快些，所以看不出二者有什么明显差异，可能还需要加大数据传输量和测试次数。

1. 用32个进程，每次传输128MB数据，分别以0、1、…、8号进程为数据源，得到运行时间如下，单位为秒。可见我的实现的波动幅度更大，没MPI的稳定。



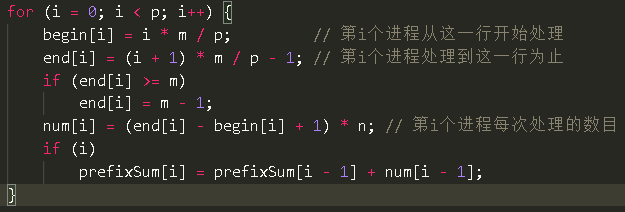
1. 以0号进程为数据源传128MB的数据，分别用2、4、8、16、32、64个进程运行程序，得到结果如下，单位为秒：



当进程个数达到64后我实现的Bcast的性能才体现出来。

**6.13**

1. 关键代码
2. 初始化：



1. 统一状态：

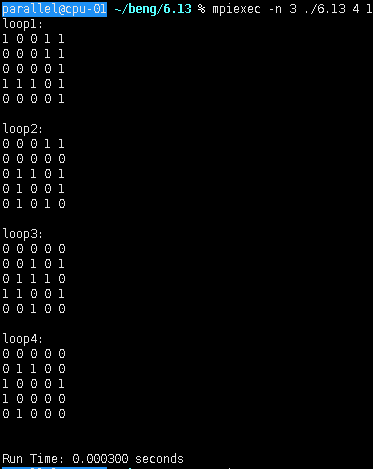


（2）思路解释

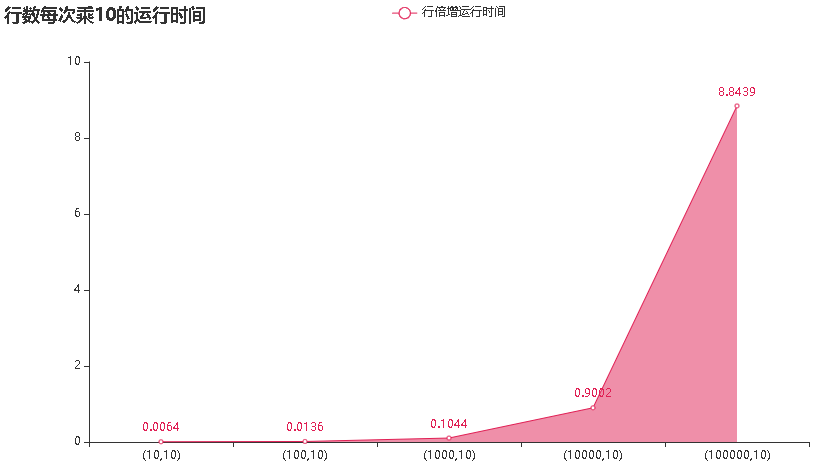
1. 每个进程负责连续的m/p行状态，记录所有进程的起始行、结束行、处理格子的个数、起始点所在的个数数组等信息；
2. 每次循环结束后用MPI\_Allgatherv函数统一状态，即每个进程把自己负责的行的下一时间的状态发给其他所有进程；
3. 本题主要难点在读取文件到二维矩阵，其实用一维矩阵就可以代替二维的，但我不想这么做，用了二维vector、c的malloc一次性分配给0号指针、c++的new一次性分配给0号指针，都没做到连续，后两种连矩阵都读不进来，读到最后一行就报错，不知缘由；用c++的for循环挨个分配给一级指针也能读矩阵，但也不连续；空间不连续则统一状态时一次不能传多行数据，只能一行一行传。最后找到两种方法，一种是先生成m\*n个元素的一维数组，然后把m个一级指针指向每行开始的位置，m个一级指针构成一个二维数组；另一种是直接声明“int state[m][n];”即可。
4. 应该也可以用MPI\_Allgather、MPI\_Gather、MPI\_Send/Recv三种方法实现状态同步，但实现代码可能多些。

（3）实验分析

1. 验证题中demo：用3个进程运行，每次循环都输出，迭代4次（算上初始状态）

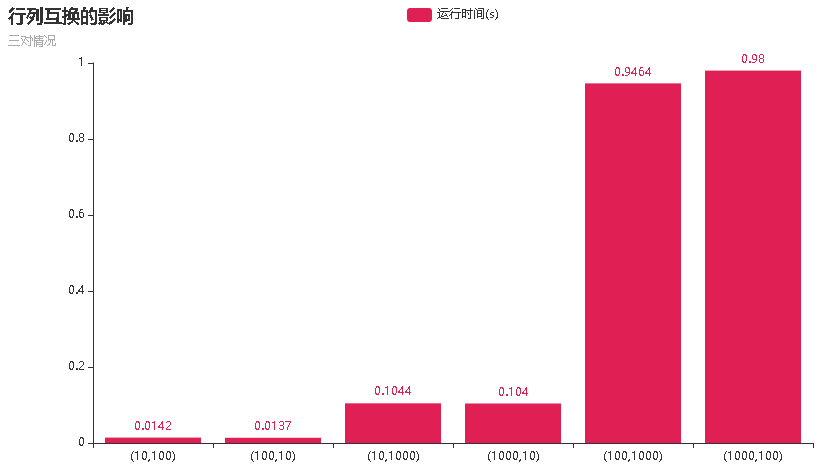


1. 以下实验都只输出一次最终状态、迭代次数据不算初始状态、随机生成m行n列的数据。每次将问题规模按行扩大到第10倍，分别传入（10,10）、（100,10）、（1000,10）、（10000,10）、（100000,10），每次用10个进程运行，迭代1000次，输出最后一次的结果，得到运行时间如下，单位为秒：



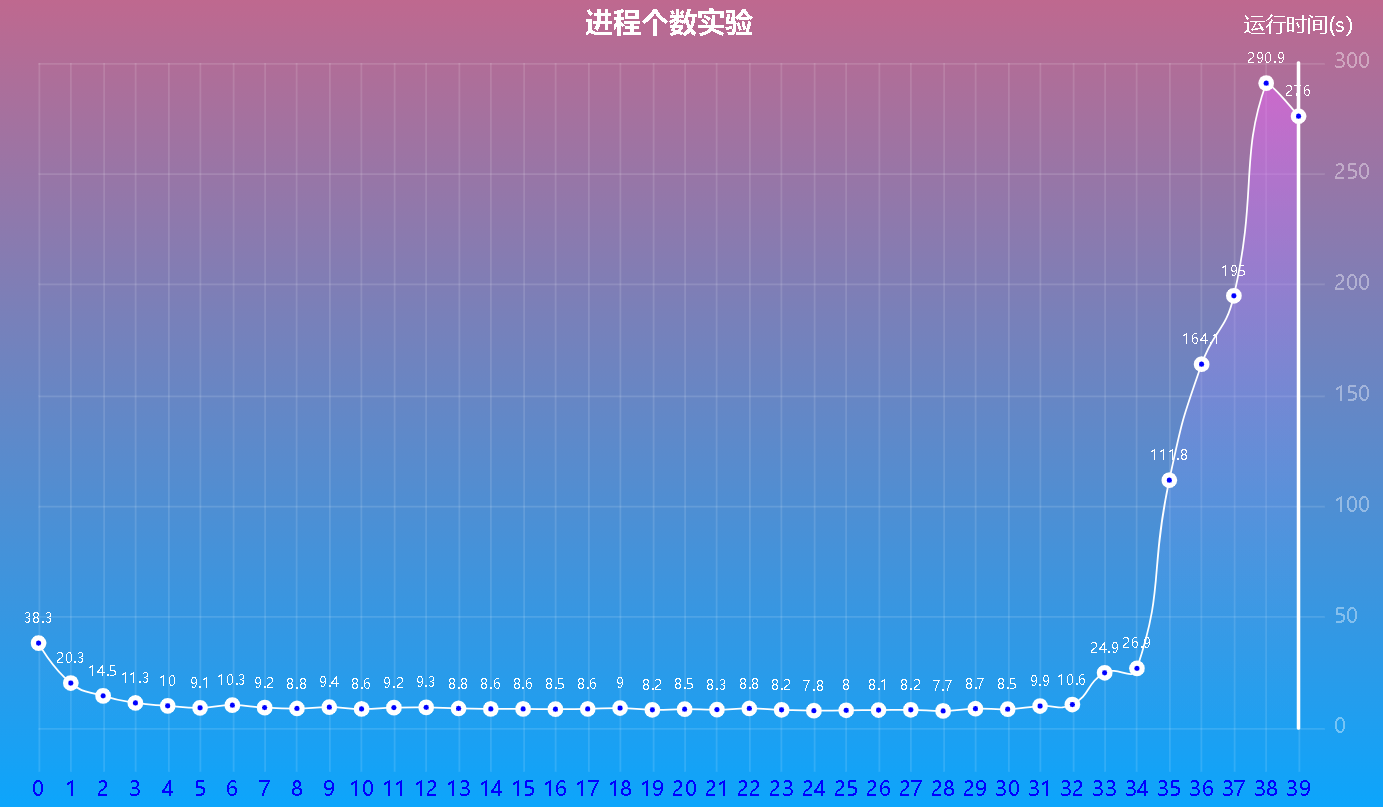
行数每乘以10，运行时间接近乘以了10倍，进程间同步的开销并没有成为瓶颈。

1. 测试行列相对大小的影响。用10个进程运行，迭代1000次，问题规模分别为（10，100）和（100，10）、（10,1000）和（1000,10）、（100,1000）和（1000,100）。运行时间如下：



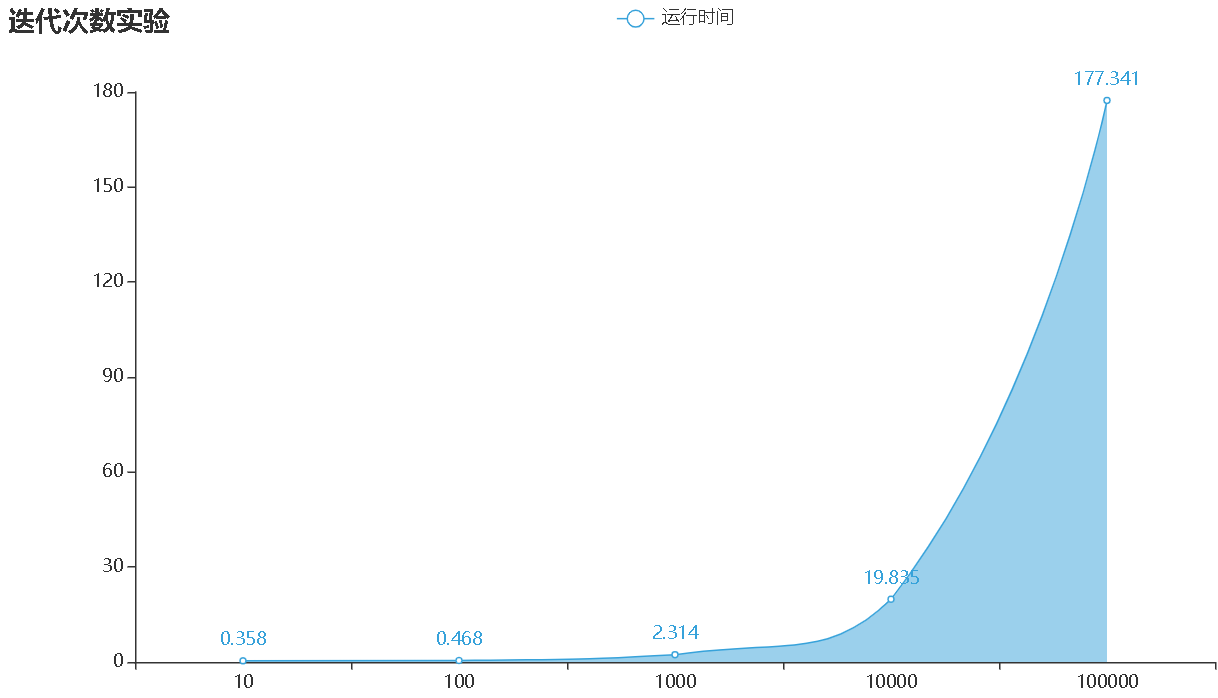
当行数为10时多进程的优势体现不出来，不过行列互换的运行时间都差不多。

1. 固定问题规模为（1000,1000），迭代1000次，分别用1、2、…、40个进程运行，得到结果如下：

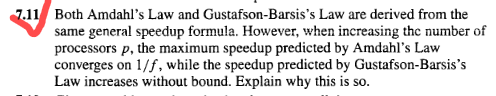


刚开始运行时间会减少，进程数达到8个后运行时间基本平稳了，达到34后开始突增，最快是29个进程时跑了7.68s，不需要进程数是行数的因子。此外，最终状态的输出占了较多时间，是瓶颈。

1. 固定问题规模为（1000,1000），进程数为10，迭代次数分别为10、100、1000、10000，得到运行结果如下：



和第二点的实验一样，迭代次数乘以10，运行时间增长小于10倍。

**7.11**

Amdahl’s Law把问题规模定成了分子1，串行执行时需要1的时间，然后尽可能的减小分母也就是并行运行的时间，而并行时有f比例的操作是必须串行的，其他（1-f）比例的操作可以分摊到p个进程去执行，所以无论p怎么增大，都只能使可并行操作的执行时间趋向0、把分母减小到f，所以加速比有上限。

Gustafson-Barsis’s Law中相当于把分母定成了1，问题规模被消去了也就是可以无限增大。设一共要做n个操作，一台机器串行执行的速度是1个单位时间做1个操作，并行执行时不考虑串行操作的话p台机器可以做p个操作，考虑串行操作的话，一台机器串行，它的所有操作都有效，其他（p-1）台机器的（p-1）个操作中有s比例是需要串行的也就是无效的不算数的，需要减去，所以并行时的速度是1个单位做p-(p-1)s个操作，，问题规模n被消去了，p可以任意大，每p个操作都有（1-s）比例的操作都算数。