并行计算第三次作业

5.9

（1）关键代码

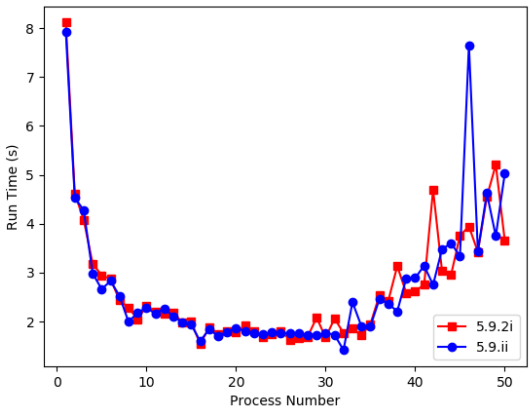
```
// 单独处理2，因为2是偶数
if (!id)
    for (i = 4; i <= n; i += 2)
        mark[i] = 1;

for (i = 3; i < sqrtn; i += 2) {
    if (isPrime(i)) {
        ith++; // 质数的编号，2是0号，3是1号
        if (ith % p == id)
            // for (j = i*i; j <= n; j += i)
            for (j = i << 1; j <= n; j += i)
                mark[j] = 1;
    }
}
```

（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：



2. 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。

3. 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）

```
// root ==0 版本
if (id) // 需要接收数据
{
    from = id - (int)pow(2.0, (double)(int)(log(id) / log(2)));
    MPI_Recv(data, count, datatype, from, id, comm, MPI_STATUS_IGNORE);
    base = pow(2.0, (double)(int)(log(id) / log(2)) + 1);
} else
    base = 1;
to = id + base;
while (to < p) {
    MPI_Send(data, count, datatype, to, to, comm);
    base = base << 1;
    to = id + base;
}
```

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。

(2) 思路解释

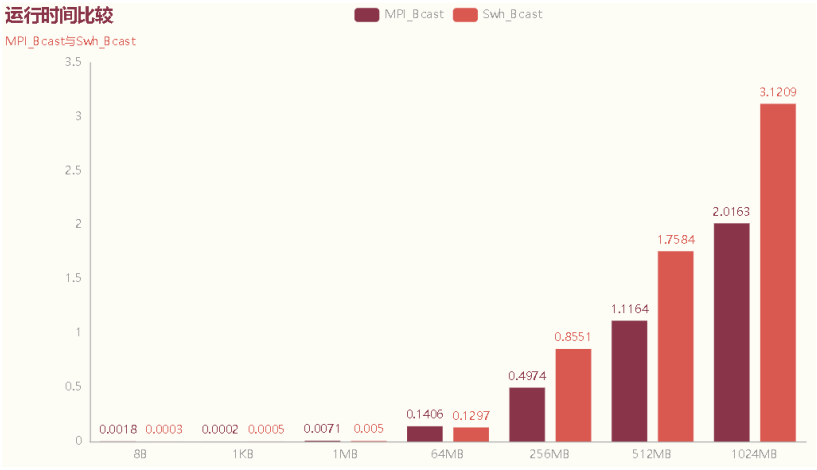
1. 利用 Binomial Tree 的思想，反向发送数据，如下图所示：1 号发数据给 2 号，然后 1、2 号发给 3、4 号，然后 1、2、3、4 号发给 5、6、7、8 号。每轮发送的数据翻一番，一共只需要 log p 次发送，向上取整。

```
// 1->2->3->5->09
//    2->4->6->10
//       3->7->11
//       4->8->12
//          5->13
//          6->14
//          7->15
//          8->16
```

2. 实现的时候每个号码需要减一，因为进程号从 0 开始。
3. 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。

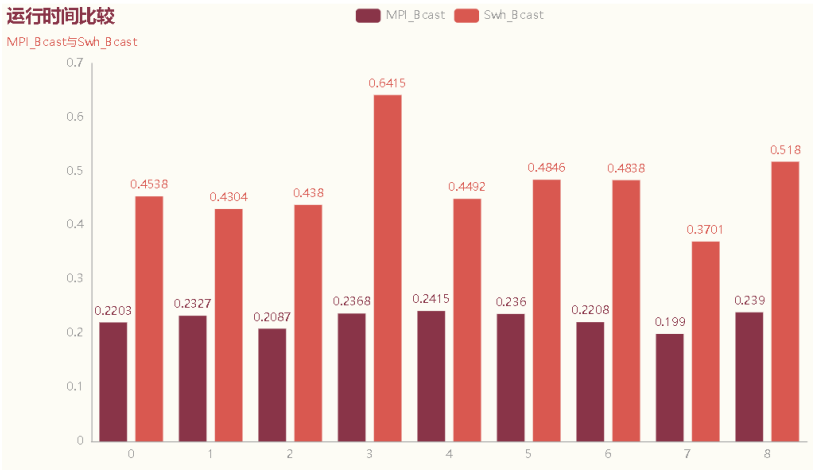
(3) 实验

1. 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。

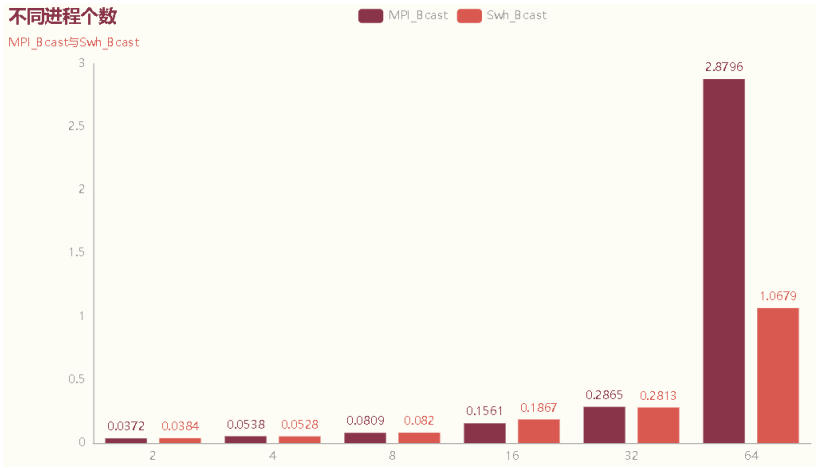


数据传输 8B、1MB、64MB 时我的实现稍快些，其他情况都慢些，有时候 1GB 时我的实现快些，所以看不出二者有什么明显差异，可能还需要加大数据传输量和测试次数。

2. 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。



3. 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

1. 初始化：

```
for (i = 0; i < p; i++) {
    begin[i] = i * m / p;         // 第i个进程从这一行开始处理
    end[i] = (i + 1) * m / p - 1; // 第i个进程处理到这一行为止
    if (end[i] >= m)
        end[i] = m - 1;
    num[i] = (end[i] - begin[i] + 1) * n; // 第i个进程每次处理的数目
    if (i)
        prefixSum[i] = prefixSum[i - 1] + num[i - 1];
}
```

2. 统一状态：

```
MPI_Allgatherv(next_state[begin[id]], num[id], MPI_INT, state[0], num,
               prefixSum, MPI_INT, MPI_COMM_WORLD); // 同步状态
```

（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。

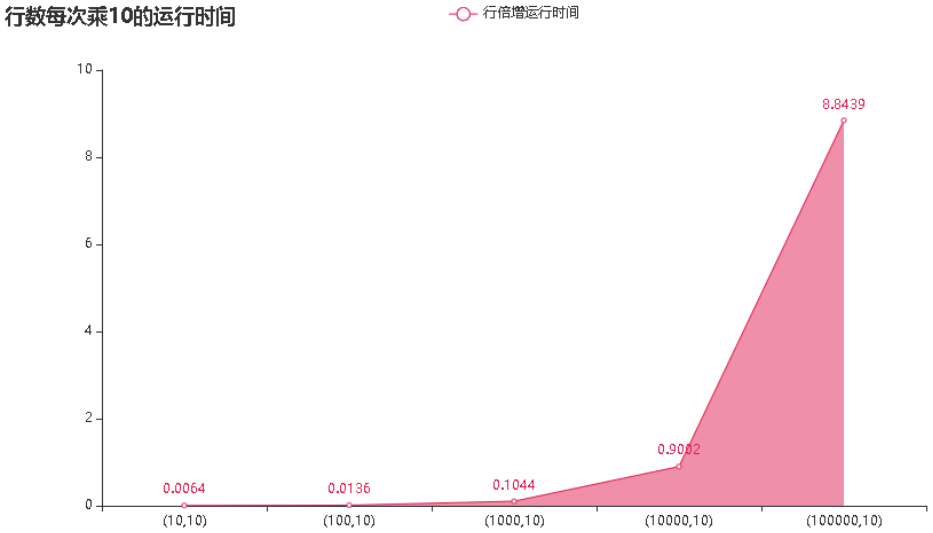
（3）实验分析

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）

```
parallel@cpu-01 ~/beng/6.13 % mpiexec -n 3 ./6.13 4 1
loop1:
1 0 0 1 1
0 0 0 1 1
0 0 0 0 1
1 1 1 0 1
0 0 0 0 1

loop2:
0 0 0 1 1
0 0 0 0 0
0 1 1 0 1
0 1 0 0 1
0 1 0 1 0

loop3:
0 0 0 0 0
0 0 1 0 1
0 1 1 1 0
1 1 0 0 1
0 0 1 0 0

loop4:
0 0 0 0 0
0 1 1 0 0
1 0 0 0 1
1 0 0 0 0
0 1 0 0 0


Run Time: 0.000300 seconds
```

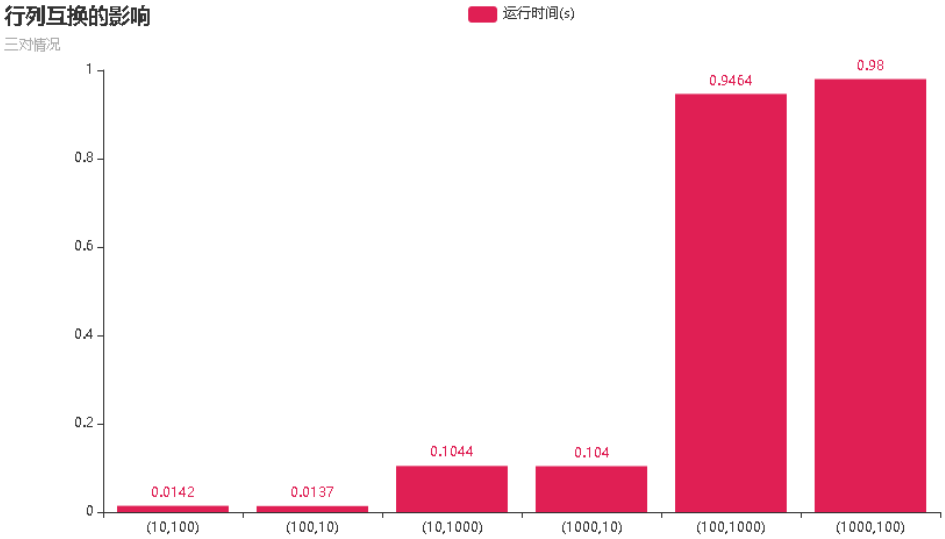
2. 以下实验都只输出一次最终状态、迭代次数据不算初始状态、随机生成 m 行 n 列的数据。每次将问题规模按行扩大到第 10 倍，分别传入（10, 10）、（100, 10）、（1000, 10）、（10000, 10）、

（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：



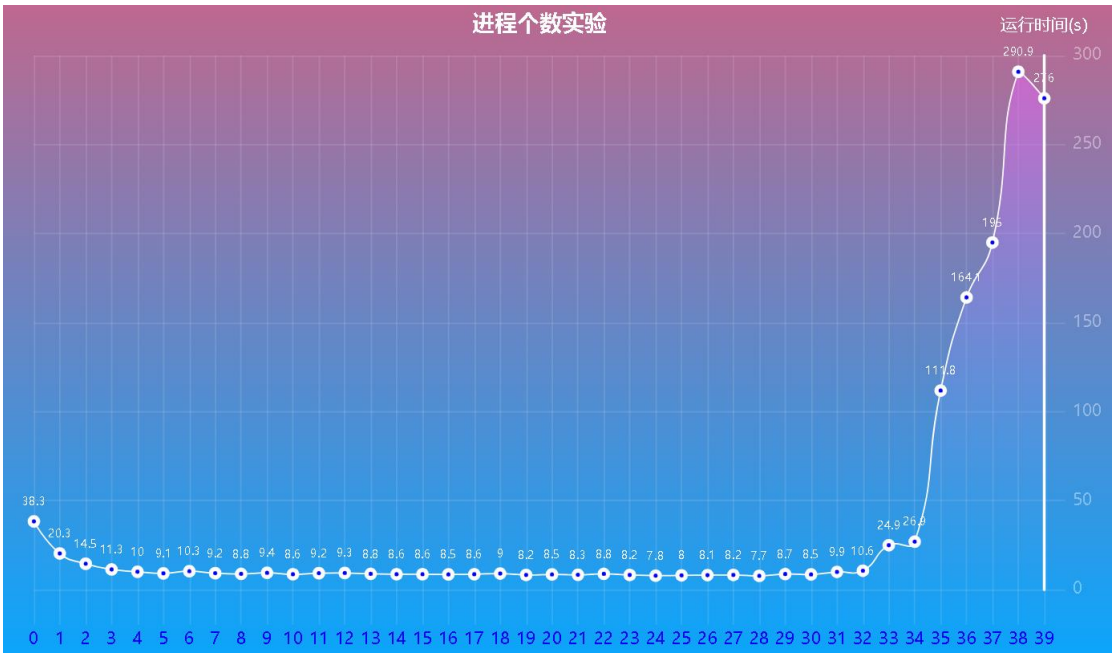
行数每乘以 10，运行时间接近乘以了 10 倍，进程间同步的开销并没有成为瓶颈。

3. 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：



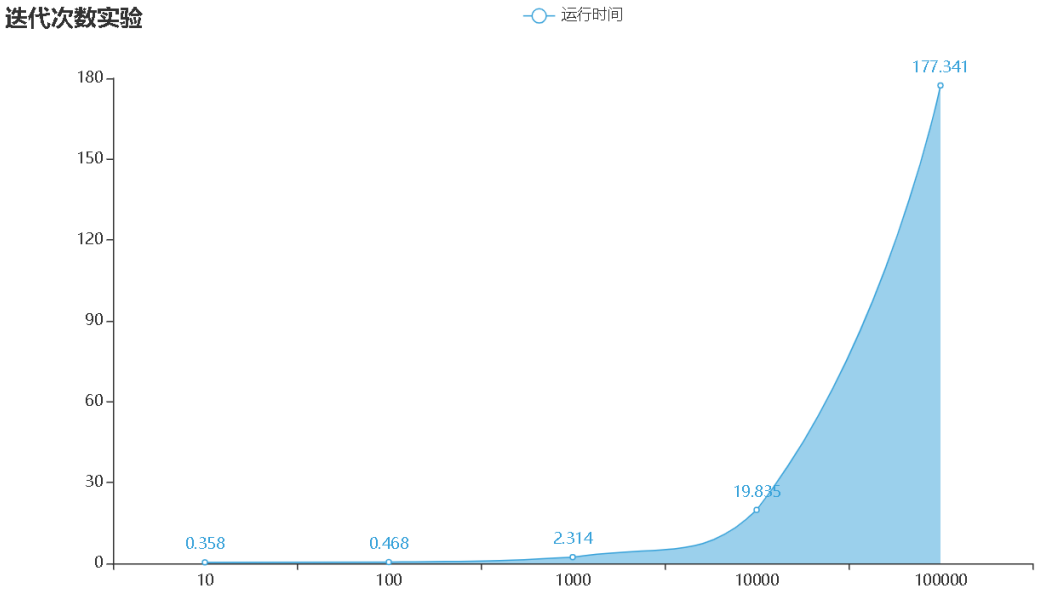
当行数为 10 时多进程的优势体现不出来，不过行列互换的运行时间都差不多。

4. 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。

5. 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

7.11 Both Amdahl's Law and Gustafson-Barsis's Law are derived from the same general speedup formula. However, when increasing the number of processors p, the maximum speedup predicted by Amdahl's Law converges on $1/f$, while the speedup predicted by Gustafson-Barsis's Law increases without bound. Explain why this is so.

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，$\frac{\frac{n}{1}}{\frac{n}{p-(p-1)s}} = p + (1-p)s$，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。