并行计算第四次作业

8.6

（1）关键代码

```
for (r = 0; r < m; ++r) {
    if (!id)
        for (c = 0; c < n; ++c)
            fscanf(f, "%lf", &a_row[c]);
    MPI_Scatterv(a_row, num, begin, MPI_DOUBLE,
                    a[r], num[id], MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

if (!id)
    for (c = 0; c < n; ++c)
        fscanf(f, "%lf", &b_vector[c]);
MPI_Scatterv(b_vector, num, begin, MPI_DOUBLE, b, num[id],
                                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
for (i = 0; i < m; i++) {
    ab[i] = 0.0;
    for (j = 0; j < k; j++)
        ab[i] += a[i][j] * b[j];
}

MPI_Reduce(ab, result, m, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

（2）思路解释

1. 0 号进程读取矩阵和向量，每读取一行就把相应数据 Scatter 到每个进程（包括自己），每个进程就只需要申请自己所需大小的空间。
2. 每个进程把自己负责的列的结果相加后放到结果向量里，最后通过 reduce 累加到 0 号进程的 result 数组中。

（3）实验结果

1. 本地验证：

   原始数据：（最后一行是向量 b）

```
m=5, n=4
1 2 3 4
4 3 2 1
9 8 7 6
1 1 0 1
6 6 5 5
10 10 1 1
```

结果：

```
D:\codes\c++\ParallelProgramming\x64\Debug>mpiexec -n 1 ParallelProgramming.exe

37.000000 73.000000 183.000000 21.000000 130.000000
Run Time: 0.000322 seconds
```

2. 在华为服务器上运行，一共 3 台机器，配置如下所示：

规格/镜像
1vCPUs | 1GB | kc1.small.1 Ubuntu 18.04 server 64bit with ARM
1vCPUs | 1GB | kc1.small.1 Ubuntu 18.04 server 64bit with ARM
2vCPUs | 4GB | kc1.large.2 Ubuntu 18.04 server 64bit with ARM

分别用 1-18 个进程运行，每次都随机生成 1000*1000 的矩阵和 1000*1 的向量，每个数的取值范围是-1.0 到 1.0。运行时间如下：

```
0.150960
0.302616
0.459052
0.769012
1.223988
1.827994
2.443992
3.071988
3.996809
4.949931
6.008006
7.075985
7.816391
9.751992
11.151994
11.163147
13.952006
14.379067
```

单位是秒，其中包含了生成 double 类型数据的时间。进程数越多，运

行时间反而越长，可能的原因是网络通信耗时长，进程越多就需要越多的通信开销，而且机器性能低，机器之间也有性能差异，有时需要等待结果。

8.12

(1) 关键代码

```
for (l = 0; l <= n; l++) { // 计算长度为l的连续子序列，l最多是n
    for (low = id; low + l <= n; low += p) {
        high = low + l;
        if (low == high) { // l为0即只有1个结点
            cost[low][low] = 0.0;
            root[low][low] = low;
        } else {
            bestcost = MPI_FLOAT;
            for (r = low; r < high; r++) { // 遍历 l 种根结点
                rcost = cost[low][r] + cost[r + 1][high] + pro_sum[high] -
                        pro_sum[low];
                if (rcost < bestcost) {
                    bestcost = rcost;
                    bestroot = r;
                }
            }
            cost[low][high] = bestcost;
            root[low][high] = bestroot;
        }
    }
    for (low = 0; low + l <= n; low++) { // 共享长为 l 的最优解
        high = low + l;
        MPI_Bcast(&cost[low][high], 1, MPI_FLOAT, low % p, MPI_COMM_WORLD);
        MPI_Bcast(&root[low][high], 1, MPI_INT, low % p, MPI_COMM_WORLD);
    }
}
```

(2) 思路解释

1. 这个动态规划中每次处理长为 l 的有序子序列时只需要长为 l-1 时的最优解，因此可以让所有进程同时处理同一长度的子序列，求出所有长为 l 的子序列的最优解后广播共享一下，再继续处理 l+1 长的序列。

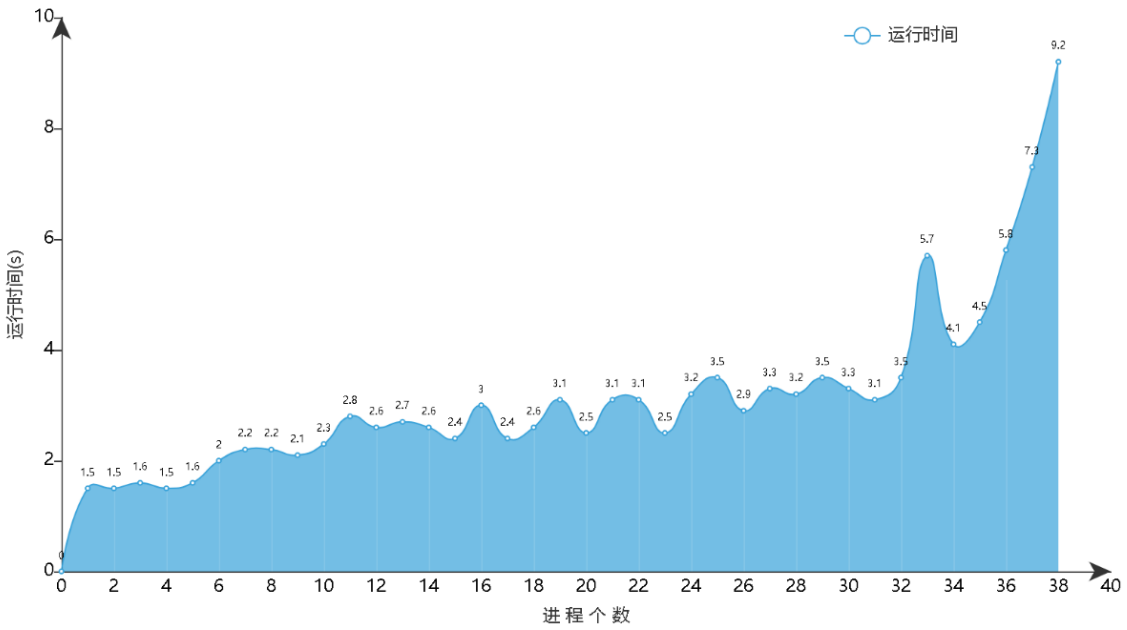
(3) 实验

1. 验证，分别用 1、2、4 个进程运行，结果正确。

```
parallel@cpu-01 ~/beng/8.12 % mpiexec -n 1 -f config ./8.12
Root spanning 0-7 is 5
Root spanning 0-4 is 1
Root spanning 2-4 is 3
Root spanning 6-7 is 7
parallel@cpu-01 ~/beng/8.12 % mpiexec -n 2 -f config ./8.12
Root spanning 0-7 is 5
Root spanning 0-4 is 1
Root spanning 2-4 is 3
Root spanning 6-7 is 7
parallel@cpu-01 ~/beng/8.12 % mpiexec -n 4 -f config ./8.12
Root spanning 0-7 is 5
Root spanning 0-4 is 1
Root spanning 2-4 is 3
Root spanning 6-7 is 7
```

2. 随机产生 1000 个概率，分别用 1-38 个进程运行，结果如下：

进程个数实验



2 个进程运行时最快，为 1.477 秒；1-5 个进程运行时时间基本一样。

每次大循环最后都要广播一下，比较耗时，数据量更大时才能体现并行优势。

9.7

(1) 关键代码: 略

(2) 思路解释

1. 根据题意，worker 进程间无需通信，只需要 manager 去读数据再发给其他进程，因此不需要建新的通信域;
2. Manager 进程广播列数 n 和向量 b 给其他进程时可以用 Ibcast 而且不需要 wait，读取 n 就立刻广播，读取完 b 就立即广播，但 worker 进程里也要跟着用 Ibcast，由于 n 是立刻分配向量 b 要用到的，所以需要立刻 wait，对于 b 可以在分配完 a_row 后再进行 wait;
3. 在 manager 的主体部分，先用 Irecv 接收信息，如果还有行数据未读出来就读一行或者 k 行，然后调用 wait 读取收到的信息，如果是一个答案就保存到 c 向量中，如果刚刚读了行数据就把数据通过 Send 发给这个信息的发送者并更新 assign_pid_rowid 和 assign_cnt，记录这个 worker 在处理哪行数据，因为需要立即更新所以这里不能用 Isend；如果没有行数据了就用 Isend 发送一个空消息给这个 worker 告知它停止运行;
4. 在 worker 进程的主体部分，先用 MPI_Probe 函数尝试获取数据，如果这个数据长度为 0 就说明可以停止运行了，否则就用 Recv 接收行数据，计算两个向量的点乘，最后通过 Send 发送数据。

(3) 实验分析

1. 验证:

   构造向量和矩阵如下:

第四次作业 > 9.7 > matrix_vector.txt

```
1   n=4
2   10 10 1 1
3   m=5
4   1 2 3 4
5   4 3 2 1
6   9 8 7 6
7   1 1 0 1
8   6 6 5 5
```

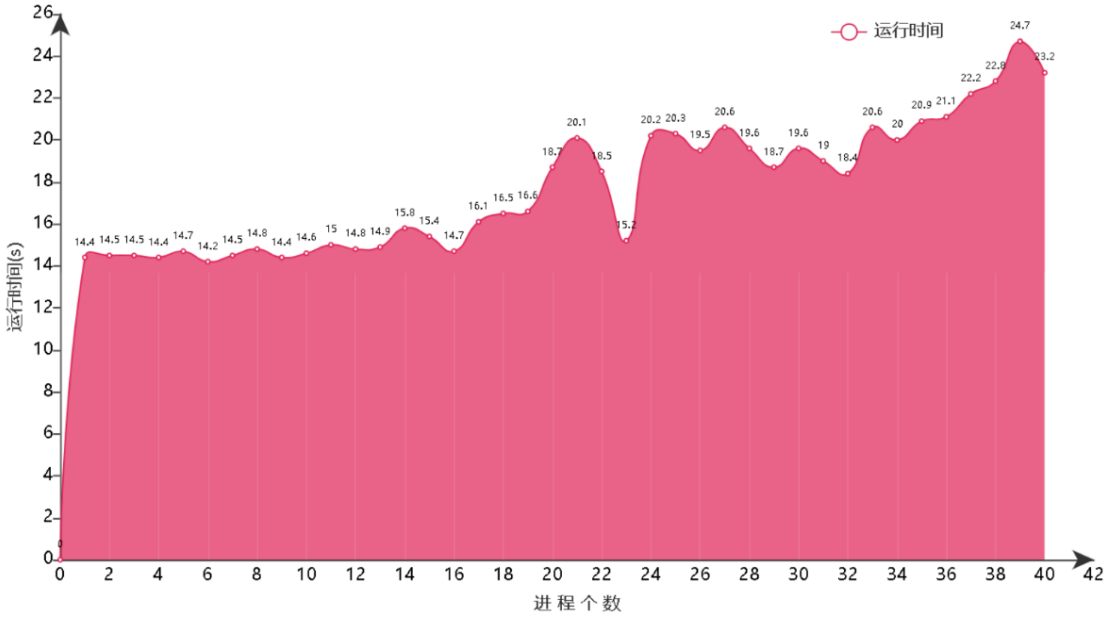
分别用 2 和 6 个进程运行，结果如下：

```
parallel@cpu-01 ~/beng/9.7 % mpiexec -n 2 -f ./config ./9.7
发第0行数据给1号进程
发第1行数据给1号进程
发第2行数据给1号进程
发第3行数据给1号进程
发第4行数据给1号进程

Result c=ab={37.000000, 73.000000, 183.000000, 21.000000, 130.000000}

Run Time: 0.000119 seconds
parallel@cpu-01 ~/beng/9.7 % mpiexec -n 6 -f ./config ./9.7
发第0行数据给2号进程
发第1行数据给3号进程
发第2行数据给4号进程
发第3行数据给5号进程
发第4行数据给1号进程

Result c=ab={37.000000, 73.000000, 183.000000, 21.000000, 130.000000}

Run Time: 0.001056 seconds
```

易知答案是对的。

2. 分别用 1-40 个 worker 进程运行，随机构造-1000 到 1000 的 double 数据，设置矩阵大小为 100000*2000，得到运行时间如下：

进程个数实验



最快时是 6 个 worker 进程的时候（14.23 秒），1-13 个 worker 进程运行的时间都是 14 秒多，没有明显变化，体现不出进程增多的性能优势。输出每个进程处理的行数后，发现很多进程一直没拿到行数据，比如用 15 个 worker 进程时：

```
193    p0: 0
194    p1: 1
195    p2: 1
196    p3: 11140
197    p4: 1
198    p5: 39937
199    p6: 1
200    p7: 48919
201    p8: 0
202    p9: 0
203    p10: 0
204    p11: 0
205    p12: 0
206    p13: 0
207    p14: 0
208    p15: 0
209
210    Run Time: 15.407355 seconds
```

只有 3、5、7 号进程拿到了大量数据，其他进程要么是 0 行要么是 1 行。尝试了很多种不同矩阵大小的实验，都是这样。可能是代码实现不完善，资源调度有缺陷，拿到数据的进程持久地占着获取数据的特权；也可能是因为每行的计算量是一样的，拿到数据的进程总是优先计算完然后立即 probe。如果 manager 准备好所有数据再开始和 worker 交互可能速度有所上升，但存储空间就变大了近 m 倍，m 是矩阵行数。

9.10

（1）关键代码

Manager 进程：

```
ll *perfect = (ll *)malloc(N * sizeof(ll));
do {
    MPI_Recv(&buffer, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
    source = status.MPI_SOURCE;
    msgTag = status.MPI_TAG;
    if (msgTag == Perfect_MSG)
        perfect[get++] = buffer;
    if (get < N && n < 32) {
        MPI_Send(&n, 1, MPI_LONG_LONG, source, n_MSG, MPI_COMM_WORLD);
        numP[source]++;
        n++;
    } else {
        MPI_Send(NULL, 0, MPI_LONG_LONG, source, EMPTY_MSG, MPI_COMM_WORLD);
        numTerminated++;
    }
} while (numTerminated < p - 1);
```

Worker 进程：

```
while (true) {
    MPI_Probe(MANAGER_ID, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_LONG_LONG, &msgLen);
    if (!msgLen)
        break;
    MPI_Recv(&n, 1, MPI_LONG_LONG, MANAGER_ID, n_MSG, MPI_COMM_WORLD,
             &status);
    if (isPrime((ll)(1 << n) - 1)) {
        n = (ll)((1 << n) - 1) * (ll)(1 << (n - 1));
        MPI_Send(&n, 1, MPI_LONG_LONG, MANAGER_ID, Perfect_MSG,
                 MPI_COMM_WORLD);
        // printf("p%d find a perfect number: %lld\n", id, n);
        // fflush(stdout);
    } else
        MPI_Send(NULL, 0, MPI_LONG_LONG, MANAGER_ID, EMPTY_MSG,
                 MPI_COMM_WORLD);
}
```

（2）思路解释

1. 思路和上一题差不多，逻辑也比上一题简单，只是要注意数据的范围，及时做数据类型转换，为了防止越界，n 不能超过 32；

2. manager 进程先用 Recv 接收 worker 的信息，如果是 perfect 数字就保存下来，如果收到的 perfect 数字还小于 8，就向该 worker 发数字并且更新状态，否则就发空消息；

3. worker 进程收到数字后判断是否为质数，是质数则把相应的 perfect 数字发给 manager，否则发一共空消息通知 manager 继续发数字过来。

（3）实验

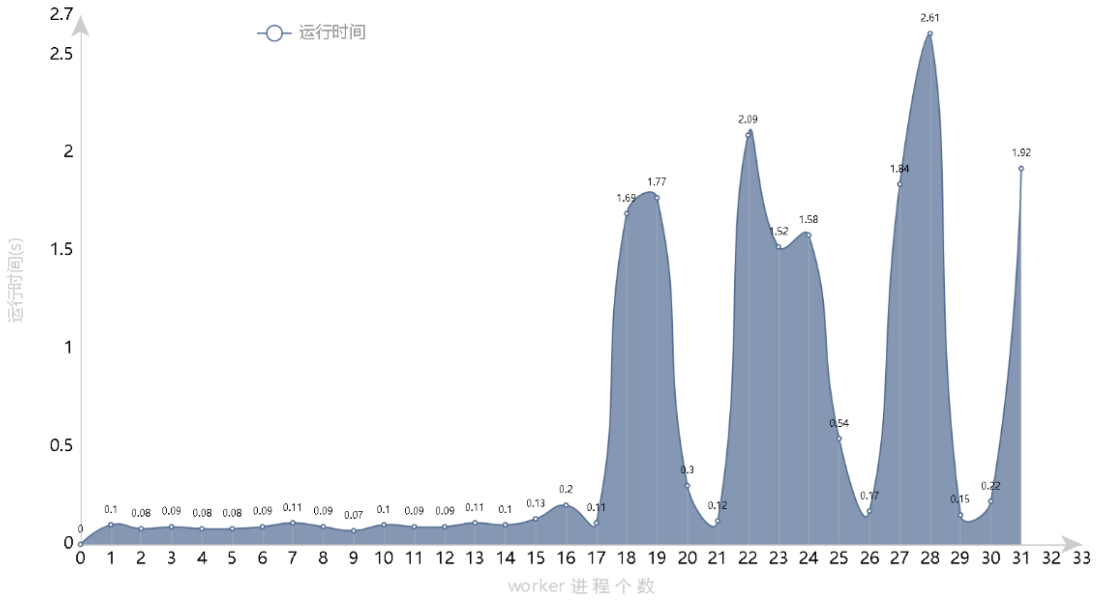
分别用 1-31 个 worker 进程运行程序，得到 8 个 perfect 数和运行时间如下所示：

9.10.log

第四次作业 > 9.10 > 9.10.log

```
8 个Perfect Number: 6, 28, 496, 8128, 33550336, 8589869056, 2305843008139952128, 137438691328.

p0: 0
p1: 8
p2: 2
p3: 8
p4: 8
p5: 4

Run Time: 0.000080 seconds
```

进程个数实验



最快的情况是 5 个进程时，花了 80 微秒。本题的每个 worker 进程不再抢占数据了，都分到了数据，与上题的不同是本题每次处理数据耗费的时间差异较大。