高性能计算导论 HW3 答题文档

计 71 张程远 2017011429

Ex 3.11 (d)

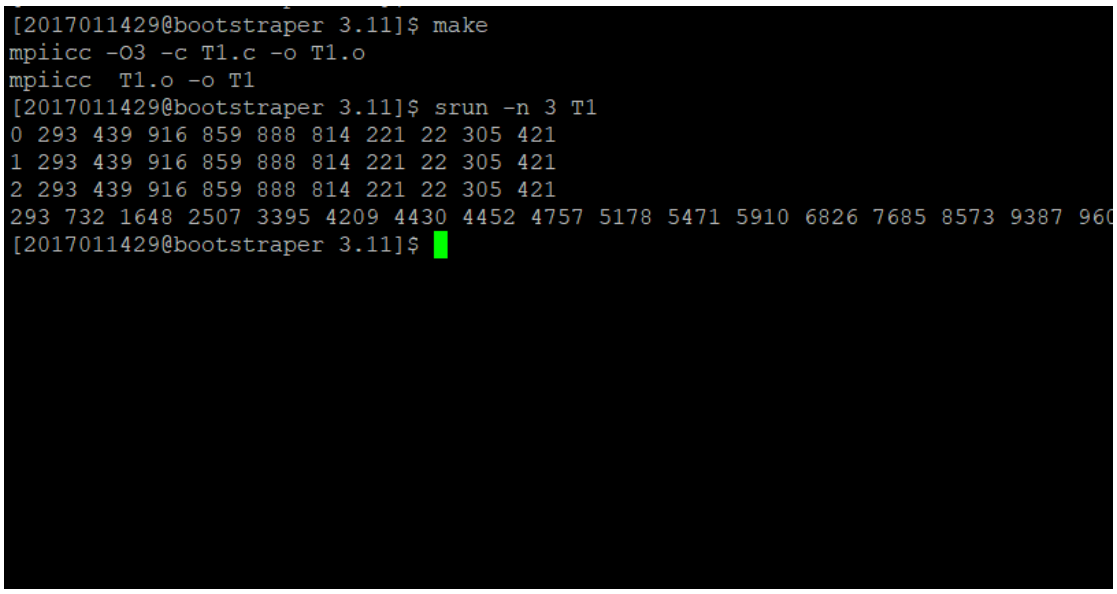
思路：首先直接在每个进程里生成数据后各自计算前缀和，得到 P 个包含了部分前缀和的向量。对于第 i 个向量的所有元素，我们要加上所有前 i-1 个向量中的最后一个元素，得到的结果才是真正的前缀和。为此，对于每一个向量，我们取向量的最后一个元素，然后对这 P 个元素调用一次 MPI_Scan 求其前缀和，并让每个进程分别接收自己的结果 sum。用 sum 减去原本向量中最后一个元素的值，得到的结果即为我们希望为原向量中每一个元素所加的值。计算结束后，调用 MPI_Gather 将所有元素聚集在 0 号向量的指定位置即可。

核心代码：

```
for(int i=0;i<local_data_count;i++){
        loc_pre_sum[i]=(i? local_data[i]+loc_pre_sum[i-1]:local_data[i]);
    }

    int part_sum;
    MPI_Scan(loc_pre_sum[local_data_count-
1],&part_sum,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
    part_sum-=loc_pre_sum[local_data_count-1];
    for(int i=0;i<local_data_count;i++){
        loc_pre_sum[i]+=part_sum;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Gather(loc_pre_sum,local_data_count,MPI_INT,pre_sum,local_data_count,MPI_INT,0
,MPI_COMM_WORLD);
```

效果图：（由于采用时间做种子，所以产生的数基本相同）



PA 3.1

思路：完成代码中缺失的部分即可。需要补充的函数为 Which_bin 和 Find_bins，前一个函数是对某个数据寻找桶，采用线性查找的方式，只需枚举桶的上下界然后看在不在里面

即可；Find_bins 借助 Which_bin 对每个进程自己的数据块进行统计，最后将统计结果用 MPI_Reduce（op 为 MPI_Sum）的方式统计到 0 号核即可。

下面为 Find_bins 的代码：

```
for(int i = 0;i < local_data_count; i++){
    int bin=Which_bin(local_data[i], bin_maxes, bin_count, min_meas);
    loc_bin_cts[bin]++;
}
MPI_Reduce(loc_bin_cts,bin_counts,bin_count,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
```

效果：

```
[2017011429@bootstraper ~]$ cd HW3
[2017011429@bootstraper HW3]$ cd 3.1
[2017011429@bootstraper 3.1]$ srun -n 4 prog3.1
Enter the number of bins
8
Enter the minimum measurement
0
Enter the maximum measurement
100
Enter the number of data
80
0.000-12.500:   XXXXXXXX
12.500-25.000:  XXXXXXXXX
25.000-37.500:  XXXXXXXXXXX
37.500-50.000:  XXXXXXXX
50.000-62.500:  XXXXXXXXXXX
62.500-75.000:  XXXXXXXXX
75.000-87.500:  XXXXXXXXXXX
87.500-100.000: XXXXXXXXXXXXX
[2017011429@bootstraper 3.1]$ cd
[2017011429@bootstraper ~]$
```

PA3.5&3.6

思路：将矩阵分块，分到各子进程中分别计算，然后将各自结果发给 0 号进程，并由 0 号进程整合输出。不计所有的数据生成时间；Pararrel Time 是并行程序中 0 号进程需要的时间，因 0 号进程需要额外的判断操作，故理论用时最长；Distribute Time 是数据分发处理所需的时间，包括 Bcast 和 Scatter,Scatterv 以及 3.6 中把二维数据处理成一维的时间。Serial Time 是调用 Serial 得到结果所需要的时间，Delta 是 2-范数形式的误差。

运行结果：

```
[2017011429@bootstraper 3.5]$ make
mpiicc -O3 -c prog3_5.cpp -o prog3_5.o
mpiicc  prog3_5.o -o prog3_5
[2017011429@bootstraper 3.5]$ srun -n 1 prog3_5 2000
Parallel Time-- 0.020205 s
Distribute Time-- 0.013511 s
Serial Time-- 0.002607 s
delta: 0.000000
[2017011429@bootstraper 3.5]$ srun -n 2 prog3_5 2000
Parallel Time-- 0.014821 s
Distribute Time-- 0.009435 s
Serial Time-- 0.002567 s
delta: 0.000000
[2017011429@bootstraper 3.5]$ srun -n 4 prog3_5 2000
Parallel Time-- 0.012340 s
Distribute Time-- 0.007718 s
Serial Time-- 0.002577 s
delta: 0.000000
[2017011429@bootstraper 3.5]$ srun -n 8 prog3_5 2000
Parallel Time-- 0.010936 s
Distribute Time-- 0.008246 s
Serial Time-- 0.002559 s
delta: 0.000000
[2017011429@bootstraper 3.5]$ srun -n 1 prog3_5 20000
Parallel Time-- 2.097521 s
Distribute Time-- 1.719320 s
Serial Time-- 0.263325 s
delta: 0.000000
[2017011429@bootstraper 3.5]$ srun -n 2 prog3_5 20000
Parallel Time-- 1.956764 s
Distribute Time-- 1.695936 s
Serial Time-- 0.260526 s
delta: 0.000000
[2017011429@bootstraper 3.5]$ srun -n 4 prog3_5 20000
Parallel Time-- 1.418696 s
Distribute Time-- 1.300794 s
Serial Time-- 0.302636 s
delta: 0.000000
[2017011429@bootstraper 3.5]$ srun -n 8 prog3_5 20000
Parallel Time-- 1.005358 s
Distribute Time-- 0.920509 s
Serial Time-- 0.309892 s
delta: 0.000000
```

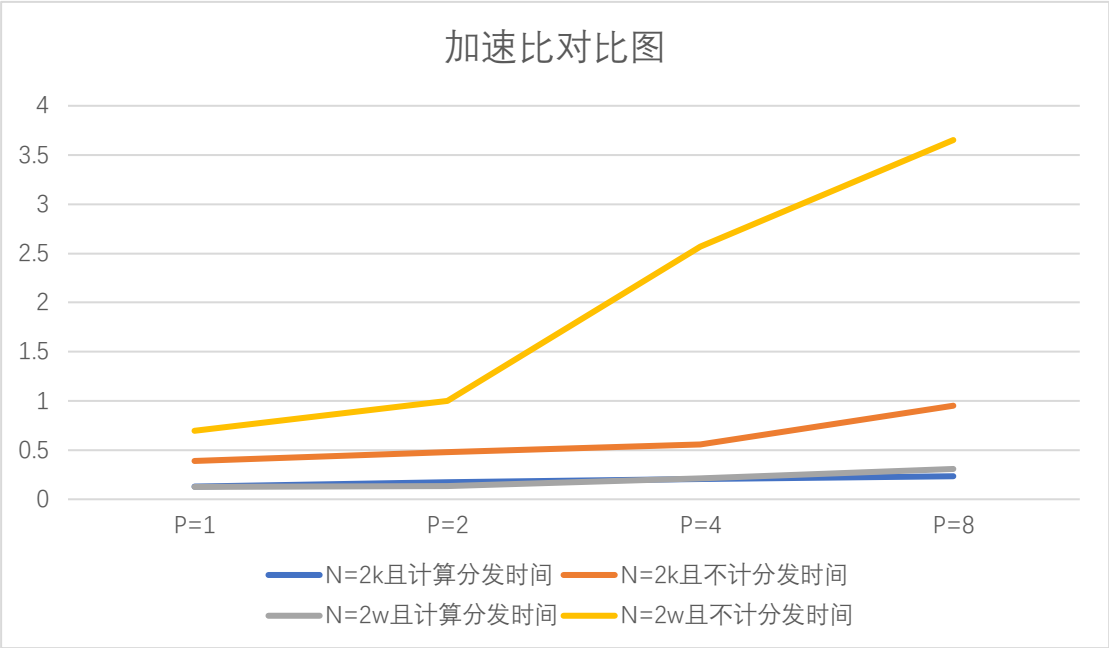
```
[2017011429@bootstraper 3.6]$ srun -n 1 prog3_6 2000
Parallel Time-- 0.023613 s
Distribute Time-- 0.021149 s
Serial Time-- 0.002561 s
delta: 0.000000
[2017011429@bootstraper 3.6]$ srun -n 4 prog3_6 2000
Parallel Time-- 0.019022 s
Distribute Time-- 0.018210 s
Serial Time-- 0.002549 s
delta: 0.000000
[2017011429@bootstraper 3.6]$ srun -n 9 prog3_6 2001
Parallel Time-- 0.017134 s
Distribute Time-- 0.016716 s
Serial Time-- 0.002544 s
delta: 0.000000
[2017011429@bootstraper 3.6]$ srun -n 16 prog3_6 2000
Parallel Time-- 0.017008 s
Distribute Time-- 0.016712 s
Serial Time-- 0.002543 s
delta: 0.000000
[2017011429@bootstraper 3.6]$ srun -n 1 prog3_6 20000
Parallel Time-- 2.382699 s
Distribute Time-- 2.126727 s
Serial Time-- 0.263911 s
delta: 0.000000
[2017011429@bootstraper 3.6]$ srun -n 4 prog3_6 20000
Parallel Time-- 1.792122 s
Distribute Time-- 1.723522 s
Serial Time-- 0.269721 s
delta: 0.000000
[2017011429@bootstraper 3.6]$ srun -n 9 prog3_6 20001
Parallel Time-- 1.554872 s
Distribute Time-- 1.504013 s
Serial Time-- 0.276859 s
delta: 0.000000
[2017011429@bootstraper 3.6]$ srun -n 16 prog3_6 20000
Parallel Time-- 1.434723 s
Distribute Time-- 1.397400 s
Serial Time-- 0.277734 s
delta: 0.000000
```

按列分块时的表格:

N=2000	分发时间/s	计算时间/s	总时间/s	计算分发时间的加速比	不计算分发时间的加速比
P=1	0.013511	0.006694	0.202050	0.129027	0.389453
P=2	0.009435	0.005386	0.014821	0.173200	0.476606
P=4	0.007718	0.004622	0.012340	0.208833	0.557551
P=8	0.008246	0.002690	0.010936	0.233998	0.951301

N=20000	分发时间/s	计算时间/s	总时间/s	计算分发时间的加速比	不计算分发时间的加速比
P=1	1.719320	0.378201	2.097521	0.125541	0.696257
P=2	1.695936	0.260828	1.956764	0.133141	0.998842
P=4	1.300794	0.117902	1.418696	0.213320	2.566844
P=8	0.920509	0.084849	1.005358	0.308240	3.652276

折线图:

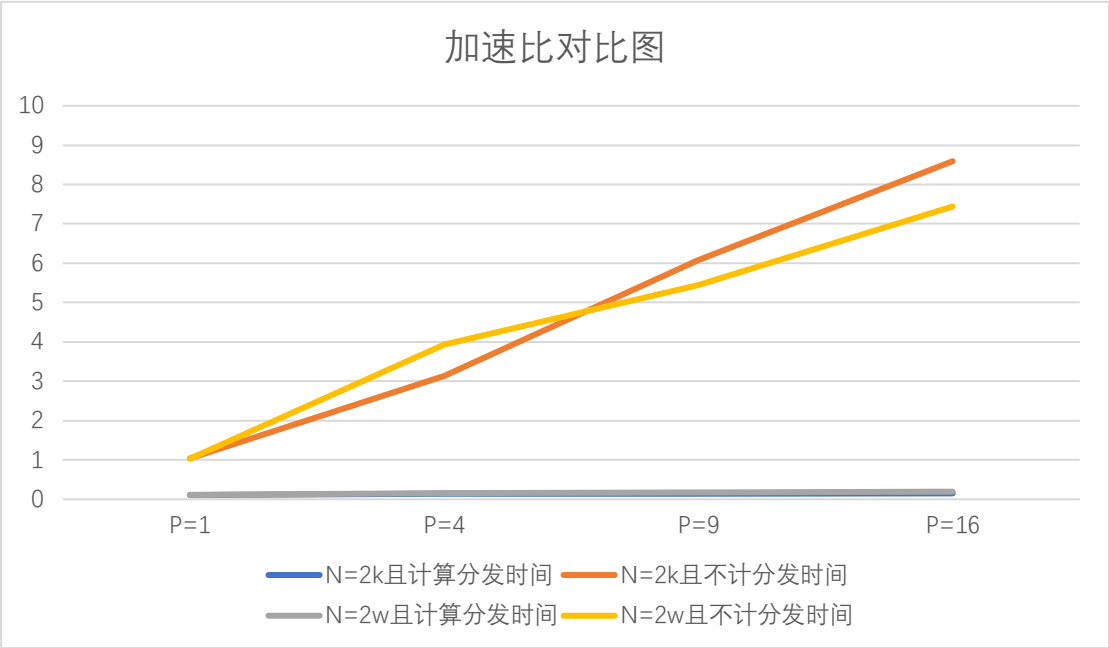


按子矩阵分块时的表格:

N=2000	分发时间/s	计算时间/s	总时间/s	计算分发时间的加速比	不计算分发时间的加速比
P=1	0.021149	0.002464	0.023613	0.108457	1.039367
P=4	0.01821	0.000812	0.019022	0.134003	3.139163
P=9	0.016716	0.000418	0.017134	0.148477	6.086124
P=16	0.016712	0.000296	0.017008	0.149518	8.59216

N=20000	分发时间/s	计算时间/s	总时间/s	计算分发时间的加速比	不计算分发时间的加速比
P=1	2.126727	0.255972	2.382699	0.110761	1.031015
P=4	1.723522	0.068600	1.792122	0.150504	3.931793
P=9	1.504013	0.050859	1.554872	0.178059	5.443658
P=16	1.39740	0.037323	1.434723	0.193580	7.441363

折线图:



结论：并行程序的效率基本都小于 1；在整个程序的运行时间中，分发时间是提升效率的瓶颈，如果去掉分发时间考虑加速比能够得到比较符合实际情况的值。