第三次作业报告

李晨昊 2017011466

2019-4-9

目录

1 运行方式 **2**

2 Q1 **2**

2.1 解题思路 2
2.2 关键程序 2
2.3 测试结果 2

3 Q2 **3**

3.1 解题思路 3
3.2 关键程序 4
3.3 测试结果 5

4 Q3 **5**

4.1 解题思路 5
4.2 关键程序 6
4.3 测试结果 6

5 Q4 **8**

5.1 解题思路 8
5.2 关键程序 8
5.3 测试结果 10

1 运行方式

直接执行 `make` 即可分别编译 `1.c`，`2.c`，`3.cpp`，`4.cpp` 得到四个可执行文件。其中 1 与 2 均可直接用 `srun -n [n proc] -l [1|2]` 运行，3 和 4 需要提供为程序提供一个命令行参数表示矩阵大小 `srun -n [n proc] -l [3|4] [matrix size]`，程序内部会对这个参数是否合法做出检查。

2 Q1

2.1 解题思路

本题要求每个进程生成有 10 个元素的随机数组，按顺序拼接成 10p 个元素的全局数组，计算这个全局数组的 10p 个前缀和，存在 p 个进程中。最后，将前缀和传输给 0 号进程输出。

使用 `MPI_Scan` 来从各个进程收集局部的前缀和信息。先在每个进程中计算出局部的前缀和，再调用 `MPI_Scan` 计算从开头到本进程的数字之和。此值减去本进程的所有数字之和，即是本进程的计算出的前缀和距离全局前缀和的差距，将本进程的计算结果都加上这个差距后，汇总到 0 号进程输出即可。

2.2 关键程序

```
// 计算局部前缀和
for (int i = 1; i < LOCAL_N; ++i) {
  a[i] += a[i - 1];
}
int pre;
// 计算从开头到本进程的数字之和
MPI_Scan(&a[LOCAL_N - 1], &pre, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
// pre - a[LOCAL_N - 1] 即是本进程之前的所有数字之和
pre -= a[LOCAL_N - 1];
// 加上这个值后，本进程计算出的值变成全局前缀和
for (int i = 0; i < LOCAL_N; ++i) {
  a[i] += pre;
}
```

2.3 测试结果

由于本题的设定，数据的规模本身是正比于进程数目的，因此无法对于一个固定的数据规模来计算不同进程数下的加速比。由于加速比 (一般而言) 总是小于核数，因此本题在较多进程

上的运行速度一定慢于较少进程，只用考察速度之比与 1 差距多少即可。

为了方便时间测定，去掉了程序中的输出，并且让每个进程生成 10^7 个随机数而非 10 个。注意表中的与 `p=1` 的速度比并不是平时讨论的加速比。

时间消耗表格如下

p	耗时	与 p=1 的时间比	与 p=1 的速度比
1	0.212s	1.000	1.000
2	0.337s	1.590	1.258
3	0.396s	1.868	1.606
4	0.424s	2.000	2.000
5	0.395s	1.863	2.684
6	0.416s	1.962	3.058
7	0.438s	2.066	3.388
8	0.452s	2.132	3.752
9	0.472s	2.226	4.042
10	0.490s	2.311	4.327
11	0.516s	2.434	4.519
12	0.536s	2.528	4.746
13	0.563s	2.656	4.895
14	0.591s	2.788	5.022
15	0.604s	2.849	5.265
16	0.627s	2.958	5.410
17	0.646s	3.047	5.579
18	0.668s	3.151	5.713
19	0.694s	3.274	5.804
20	0.711s	3.354	5.963
21	0.740s	3.491	6.016
22	0.752s	3.547	6.202
23	0.772s	3.642	6.316
24	0.812s	3.830	6.266

3 Q2

3.1 解题思路

本题要求完成 `Find_bins` 函数和 `Which_bin` 函数。

Which_bin 的作用为在一个已排序的数组中寻找某个位置，使得某值位于两个数组元素之间。其实本来在本题的设定中，可以直接使用一步除法就得到这个结果的，但是既然要求这样做，我也没太大必要来优化，直接采用朴素的线性查找即可。

Find_bins 的作用为为每个进程局部的桶统计计数，这一步只需要调用 Which_bin 即可。完成之后，使用 MPI_Reduce 将所有进程的统计结果求和。

3.2 关键程序

```
int Which_bin(float data, float bin_maxes[], int bin_count,
  float min_meas) {
  if (min_meas <= data && data < bin_maxes[0]) {
    return 0; // 特判 data 位于第一个桶内
  } else {
    // 线性查找
    for (int i = 1; i < bin_count; ++i) {
      if (bin_maxes[i - 1] <= data && data < bin_maxes[i]) {
        return i;
      }
    }
    return bin_count - 1; // 特判 data 比所有元素都大 (其实这不会发生)
  }
}

void Find_bins(
      int bin_counts[]      /* out */,
      float local_data[]    /* in  */,
      int loc_bin_cts[]     /* out */,
      int local_data_count  /* in  */,
      float bin_maxes[]     /* in  */,
      int bin_count         /* in  */,
      float min_meas        /* in  */,
      MPI_Comm comm){
  // 统计每个桶的计数
  for (int i = 0; i < local_data_count; ++i) {
    ++loc_bin_cts[Which_bin(local_data[i], bin_maxes, bin_count, min_meas)];
  }
  // 全局求和
```

```
  MPI_Reduce(loc_bin_cts, bin_counts, bin_count, MPI_INT, MPI_SUM, 0, comm);
}
```

3.3 测试结果

运行 `srun -n 10 ./2`，输入 `10 0 100 400`，得到

```
0.000-10.000:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
10.000-20.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20.000-30.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
30.000-40.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
40.000-50.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
50.000-60.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
60.000-70.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
70.000-80.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
80.000-90.000:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
90.000-100.000: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

可见结果比较均匀，是合理的。输出的结果与指定的进程数目无关，基本证明实现是正确的。

4 Q3

4.1 解题思路

本题要求将矩阵按列分配到不同进程，这就导致每个进程分配到的内存并不是连续的。经测试，循环分配每行元素虽然可行，但是效率较低，因此考虑采用 MPI 的类型定义接口，实现在一次 `MPI_Scatter` 中将矩阵的所有元素分配出去。

这里使用 `MPI_Type_vector` 和 `MPI_Type_create_resized` 来得到所需的类型。达到的效果是，定义这样一种类型：它由 `n` 块连续的 `n / nproc` 个 `double` 构成，相应 `double` 块间的间隔为 `n * sizeof(double)`，利用这种类型的指针遍历内存的步长为 `n / nproc * sizeof(double)`。这样以来，在原始的矩阵上 (row-major 连续存储)，利用这种类型步进 `nproc` 次就遍历了所有的元素，这样的分配是符合要求的。

我认为到此为止，应该是可以使用 `MPI_Scatter` 来分配矩阵元素的。然而因为一些目前无法解释的原因，这会在某些数据下发生 segment fault，因此只能改用 `MPI_Scatterv` 来实现。

按列分配了矩阵元素后，再均分向量元素，这样每个进程就得到了自己计算需要的所有数据。它的计算结果是包括最终结果的每个分量，但都只是部分和，因此最后对中间结果进行 `MPI_Reduce` 即可。

4.2 关键程序

类型定义部分

```
MPI_Datatype vec_t, col_t;
MPI_Type_vector(n, each, n, MPI_DOUBLE, &vec_t);
MPI_Type_create_resized(vec_t, 0, each * sizeof(double), &col_t);
MPI_Type_commit(&col_t);
std::unique_ptr<int[]> size(new int[nproc]), off(new int[nproc]);
for (int i = 0; i < nproc; ++i) {
  size[i] = 1;
  off[i] = i;
}
```

数据分发部分

```
// 按列分配数组元素
MPI_Scatterv(mat_vec.first.get(), size.get(), off.get(),
  col_t, loc_mat.get(), n * each, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// 分块分配向量元素
MPI_Scatter(mat_vec.second.get(), each, MPI_DOUBLE,
  loc_vec.get(), each, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

计算部分

```
std::unique_ptr<double[]> y(new double[n]);
for (int i = 0; i < n; ++i) {
  double tmp = 0.0;
  for (int j = 0; j < each; ++j) {
    tmp += loc_mat[i * each + j] * loc_vec[j];
  }
  y[i] = tmp;
}
```

4.3 测试结果

考虑数据分发，时间/加速比表格如下

p	n=3600	n=7200	n=14400	n=28800
1	0.0395s/1.000	0.1563s/1.000	0.6401s/1.000	2.6686s/1.000
2	0.0423s/1.071	0.1506s/1.038	0.5847s/1.095	2.4319s/1.097

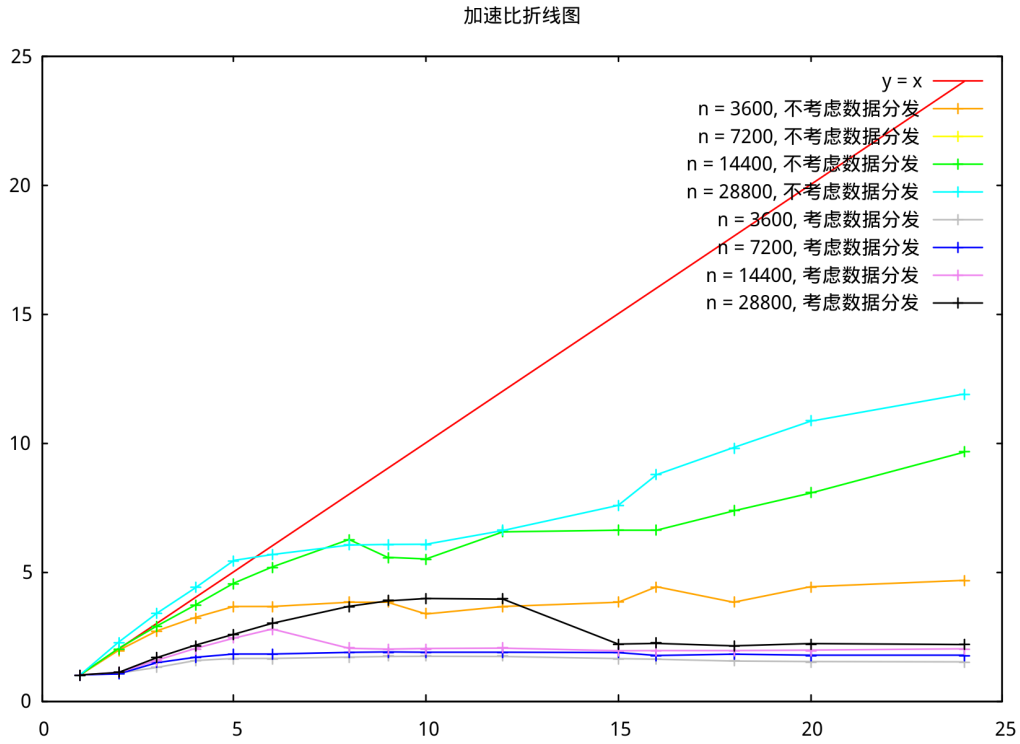
p	n=3600	n=7200	n=14400	n=28800
3	0.0306s/1.291	0.1061s/1.473	0.4078s/1.570	1.5989s/1.669
4	0.0254s/1.555	0.0930s/1.681	0.3158s/2.027	1.2397s/2.153
5	0.0241s/1.639	0.0864s/1.809	0.2642s/2.422	1.0355s/2.577
6	0.0241s/1.639	0.0862s/1.813	0.2304s/2.778	0.8889s/3.002
8	0.0234s/1.688	0.0833s/1.876	0.3149s/2.032	0.7288s/3.662
9	0.0230s/1.717	0.0825s/1.894	0.3196s/2.002	0.6896s/3.870
10	0.0229s/1.724	0.0830s/1.883	0.3156s/2.028	0.6728s/3.966
12	0.0230s/1.717	0.0831s/1.880	0.3142s/2.037	0.6780s/3.936
15	0.0242s/1.632	0.0837s/1.867	0.3298s/1.940	1.2153s/2.196
16	0.0246s/1.605	0.0890s/1.756	0.3295s/1.942	1.2026s/2.219
18	0.0257s/1.536	0.0867s/1.802	0.3291s/1.945	1.2537s/2.129
20	0.0260s/1.519	0.0886s/1.764	0.3264s/1.961	1.2059s/2.213
24	0.0262s/1.507	0.0884s/1.768	0.3174s/2.016	1.2251s/2.178

不考虑数据分发，时间/加速比表格如下

p	n=3600	n=7200	n=14400	n=28800
1	0.0084s/1.000	0.0328s/1.000	0.1368s/1.000	0.6466s/1.000
2	0.0043s/1.953	0.0171s/1.918	0.0675s/2.027	0.2852s/2.267
3	0.0031s/2.710	0.0119s/2.756	0.0472s/2.898	0.1905s/3.394
4	0.0026s/3.231	0.0092s/3.565	0.0369s/3.707	0.1479s/4.372
5	0.0023s/3.652	0.0081s/4.049	0.0300s/4.560	0.1189s/5.438
6	0.0023s/3.652	0.0079s/4.152	0.0264s/5.182	0.1141s/5.667
8	0.0022s/3.818	0.0072s/4.556	0.0219s/6.247	0.1072s/6.032
9	0.0022s/3.818	0.0072s/4.556	0.0246s/5.561	0.1067s/6.060
10	0.0025s/3.360	0.0075s/4.373	0.0249s/5.494	0.1066s/6.066
12	0.0023s/3.652	0.0073s/4.493	0.0209s/6.545	0.0980s/6.598
15	0.0022s/3.818	0.0061s/5.377	0.0207s/6.609	0.0854s/7.571
16	0.0019s/4.421	0.0058s/5.655	0.0207s/6.609	0.0738s/8.762
18	0.0022s/3.818	0.0056s/5.857	0.0186s/7.355	0.0659s/9.812
20	0.0019s/4.421	0.0052s/6.308	0.0170s/8.047	0.0597s/10.831
24	0.0018s/4.667	0.0044s/7.455	0.0142s/9.634	0.0544s/11.886

加速比折线图如下。可以发现并行效率在 n 较大而 p 较小的时候可以略微超过 1，我认为这

与 cache 有关。举例来说，假设某个任务需要对一个数组迭代多次，而这个数组无法完整装入 per core cache(L1/L2)，那么每次迭代的时候都会发生 cache 缺失 (也不一定需要访存，可能在共享的 L3 cache 处解决)；但是如果将任务分配到两个核上，这个数组的两半可能可以分别装入 per core cache，这样除了第一次迭代，后面的迭代都不会发生 cache 缺失，这样每个核的耗时都可能会小于原本耗时的一半，并行效率就超过了 1。



5 Q4

5.1 解题思路

与 Q3 类似，本题需要自定义数据类型才能一次把矩阵分配到各个子进程中。除此之外，观察得每个子矩阵在原始矩阵中的偏移量的差值并不是常数，因此需要使用 `MPI_Scatterv` 来指定子块的大小和偏移量。

对于向量的分配，这里简单的把它广播到每个进程中，让它们自己选择自己需要的部分。

5.2 关键程序

类型定义部分

```
MPI_Datatype vec_t, block_t;
MPI_Type_vector(each, each, n, MPI_DOUBLE, &vec_t);
MPI_Type_create_resized(vec_t, 0, each * sizeof(double), &block_t);
MPI_Type_commit(&block_t);
std::unique_ptr<int[]> size(new int[nproc]), off(new int[nproc]);
for (int i = 0; i < sq_nproc; ++i) {
  for (int j = 0; j < sq_nproc; ++j) {
    // 每个进程接收到一个块
    size[i * sq_nproc + j] = 1;
    // 这里的偏移量是和 MPI_Type_create_resized 中制定的大小一起考虑的
    // 换算成字节的偏移量，是 (i * n + j) * each * sizeof(double)
    off[i * sq_nproc + j] = i * n + j;
  }
}
```

数据分发部分

```
// 分块分配矩阵
MPI_Scatterv(mat.get(), size.get(), off.get(), block_t, loc_mat.get(),
  each * each, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// 广播向量
MPI_Bcast(vec.get(), n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

计算部分

```
// 注意后面的花括号，这里就把 y 清零了
std::unique_ptr<double[]> y(new double[n]{});
// 自身只使用 vec 的一部分，只填 y 的一部分，根据进程序号计算具体是哪一段
int vec_off = pid % sq_nproc * each, y_off = pid / sq_nproc * each;
for (int i = 0; i < each; ++i) {
  double tmp = 0.0;
  for (int j = 0; j < each; ++j) {
    tmp += loc_mat[i * each + j] * vec[vec_off + j];
  }
  y[y_off + i] = tmp;
}
```

5.3 测试结果

考虑数据分发，时间/加速比表格如下

p	n=3600	n=7200	n=14400	n=28800
1	0.0394s/1.000	0.1591s/1.000	0.6341s/1.000	2.6252s/1.000
4	0.0262s/1.504	0.0804s/1.979	0.3091s/2.051	1.2312s/2.132
9	0.0223s/1.767	0.0442s/3.600	0.1631s/3.888	0.6327s/4.149
16	0.0234s/1.684	0.0842s/1.890	0.1400s/4.529	0.5583s/4.702

不考虑数据分发，时间/加速比表格如下

p	n=3600	n=7200	n=14400	n=28800
1	0.0084s/1.000	0.0333s/1.000	0.1404s/1.000	0.6461s/1.000
4	0.0026s/3.231	0.0094s/3.543	0.0371s/3.784	0.1528s/4.228
9	0.0026s/3.231	0.0069s/4.826	0.0272s/5.162	0.0968s/6.675
16	0.0019s/4.421	0.0056s/5.946	0.0207s/6.783	0.0748s/8.638

加速比折线图如下