Ex 3.11

程序执行过程

1. 每个进程分别产生10个随机数。为使每个线程产生的随机数不同，将随机种子设为进程号+1.

```
srand(my_rank+1);
int a[10];
for (int i=0; i<n; i++)
    a[i] = rand() % 10;
```

2. 各个进程按进程号的顺序输出其生成的数组。由于是输出中间过程，没有采用统一发送至0号进程再输出的方式。

```
for (int i=0; i<comm_sz; i++) {
    if (i == my_rank) {
        printf("proc %d: ", my_rank);
        for (int j=0; j<10; j++)
            printf("%d ", a[j]);
        printf("\n");
        fflush(stdout);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

3. 各个线程计算本地的10个数的和 `local_sum`。

```
for (int i=0; i<n; i++)
    local_sum += a[i];
```

4. 对 `local_sum` 调用 `MPI_Scan` 获得全局前缀和。

```
MPI_Scan(&local_sum, &pre_sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

5. 计算本地的10个数的全局前缀和。

```
a[0] = pre_sum - local_sum + a[0];
for (int i=1; i<n; i++)
    a[i] += a[i-1];
```

6. 将计算结果发送至0号进程统一输出。

```
if (my_rank == 0) {
        int* output_buf = new int[comm_sz * n];
        MPI_Gather(a, n, MPI_INT, output_buf, n, MPI_INT, 0, MPI_COMM_WORLD);
        printf("sum\n");
        for (int i=0; i < comm_sz; i++) {
            for (int j=0; j<n; j++)
                printf("%d ", output_buf[i*n+j]);
            printf("\n");
        }
        delete[] output_buf;
    } else {
        MPI_Gather(a, n, MPI_INT, NULL, n, MPI_INT, 0, MPI_COMM_WORLD);
    }
```

测试结果

```
[2017011307@bootstraper mpi]$ mpicxx 3-11.cpp -o 3-11 && srun -n 4 3-11
proc 0: 3 6 7 5 3 5 6 2 9 1
proc 1: 0 9 8 5 1 8 4 7 5 7
proc 2: 6 5 8 0 5 0 2 6 1 4
proc 3: 1 3 4 6 3 7 5 3 8 5
sum
3 9 16 21 24 29 35 37 46 47
47 56 64 69 70 78 82 89 94 101
107 112 120 120 125 125 127 133 134 138
139 142 146 152 155 162 167 170 178 183
```

PA 3.1

程序执行过程

此题仅需要补充 `Find_bins` 与 `which_bin` 两个函数。

`Find_bins` 用以统计本地数据在各个区间的数量，将计算结果发送至0号进程。由于0号进程对各个进程发送的数据的处理方式是求和，数据发送采用 `MPI_Reduce` 函数。

```
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){
    int i;
    memset(loc_bin_cts, 0, sizeof(loc_bin_cts));
    for (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_INT, MPI_SUM, 0, comm);
}  /* Find_bins */
```

`which_bin`用以计算某个数据属于的桶的编号，书上给出线性查找与二分查找的方式。但由于每个桶所代表的区间相同，桶的编号可以O(1)算出。为了代码的清晰性，这里仅实现了线性查找。

```
int which_bin(float data, float bin_maxes[], int bin_count,
      float min_meas) {
    int i;
    for (i=0; i<bin_count-1; i++)
        if (data < bin_maxes[i])
            return i;
    return bin_count-1;
}
```

测试结果

```
[2017011307@bootstraper mpi]$ mpicc 3-1.c -o 3-1 -DDEBUG && srun -n 4 3-1

Enter the number of bins
5
Enter the minimum measurement
0
Enter the maximum measurement
5
Enter the number of data
20
Generated data:
   4.201 1.972 3.915 3.992 4.558 0.988 1.676 3.841 1.389 2.770 2.387 3.144 1
.824 2.567 4.761 4.581 3.179 3.586 0.708 3.035

0.000-1.000:    XX
1.000-2.000:    XXXX
2.000-3.000:    XXX
3.000-4.000:    XXXXXXX
4.000-5.000:    XXXX
```

PA 3.5

一、解题思路及关键程序

1. 程序执行过程

按列分配后，每个进程执行一个n行n/p列的矩阵与一个n/p行1列的向量的乘法，得到一个n行1列的向量。将这些向量求和可得最终答案。

我将此过程分拆为scatter、计算及gather、释放本地内存三步，分别对应三个函数。

1. scatter：由于`MPI_Scatter`在发送数据量较多时会崩溃，故使用`MPI_Scatterv`发送矩阵，在发送前将矩阵按列一维化。

```
void parallel_scatter(int n, double** matrix, double* vector, /* input */
        double** local_mat, double** local_vec)               /* output */ {
    int local_m = n/comm_sz;
    (*local_mat) = new double[local_m*n];
    (*local_vec) = new double[local_m];
    MPI_Scatter(vector, local_m, MPI_DOUBLE, *local_vec, local_m, MPI_DOUBLE,
0, comm);

    if (my_rank == 0) {
        double* buf = new double[n*n];
        int *sendcounts = new int[comm_sz];
        int *displs = new int[comm_sz];
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                buf[j*n+i] = matrix[i][j];
        for (int i=0; i<comm_sz; i++)
            sendcounts[i] = n*local_m;
        for (int i=0; i<comm_sz; i++)
            displs[i] = i*n*local_m;
        MPI_Scatterv(buf, sendcounts, displs, MPI_DOUBLE, *local_mat,
n*local_m, MPI_DOUBLE, 0, comm);
        delete[] buf;
        delete[] sendcounts;
        delete[] displs;
    } else {
        MPI_Scatterv(NULL, NULL, NULL, MPI_DOUBLE, *local_mat, n*local_m,
MPI_DOUBLE, 0, comm);
    }
}
```

2. 计算及gather: 每个进程单独计算其本地矩阵和本地向量的矩阵乘法，通过`MPI_Reduce`汇总至0号进程。

```
void parallel_solve(int n, double* local_mat, double* local_vec, double**
result) {
    double *local_ans = new double[n];
    memset(local_ans, 0, sizeof(double)*n);
    int local_m = n/comm_sz;
    for (int j=0; j<local_m; j++) {
        for (int i=0; i<n; i++) {
            local_ans[i] += local_mat[j*n+i] * local_vec[j];
        }
    }

    if (my_rank == 0)
        (*result) = new double[n];
    MPI_Reduce(local_ans, *result, n, MPI_DOUBLE, MPI_SUM, 0, comm);
    delete[] local_ans;
}
```

3. 因为输入矩阵大小不定，程序中大量使用堆上内存，需要及时释放。

```
void parallel_free(double* local_mat, double* local_vec) {
    delete[] local_mat;
    delete[] local_vec;
}
```

2.时间测试

分别测试了并行(scatter过程记入总时间)、并行(scatter过程不计入总时间)、串行的运行时间，采用`MPI_Wtime`进行计时，利用`MPI_Barrier`进行同步。最后，分别比较了两种并行算法与串行算法的输出结果（以保证结果的正确性）、加速比、并行效率。

用于计时的函数如下

```
void start_timing() {
    MPI_Barrier(comm);
    start_time = MPI_Wtime();
}

double end_timing() {
    double cur_time = MPI_Wtime();
    double used = cur_time - start_time;
    double reduced = -1;
    MPI_Reduce(&used, &reduced, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
    return reduced; // return -1 if my_rank != 0
}
```

二、测试结果

时间的单位为秒，输出的10个数分别对应下表的十项

维数	进程数	串行时间	总时间	通信时间	计算时间	加速比(总时间)	并行效率(总时间)	加速比(计算时间)	并行效率(计算时间)
2000	1	0.011209	0.091980	0.071991	0.012597	0.121864	0.121864	0.889810	0.889810
2000	2	0.011203	0.087907	0.069467	0.006316	0.127442	0.063721	1.773705	0.886853
2000	4	0.011204	0.078379	0.067867	0.003428	0.142946	0.035737	3.268396	0.817099
2000	8	0.011197	0.076340	0.068898	0.001858	0.146674	0.018334	6.026434	0.753304
4000	1	0.044809	0.541196	0.489009	0.051185	0.082796	0.082796	0.875432	0.875432
4000	2	0.044820	0.518618	0.492727	0.025560	0.086422	0.043211	1.753505	0.876752
4000	4	0.044807	0.474169	0.460008	0.013585	0.094496	0.023624	3.298303	0.824576
4000	8	0.044789	0.473605	0.464740	0.007324	0.094570	0.011821	6.115401	0.764425
8000	1	0.193806	2.164732	2.008236	0.203431	0.089529	0.089529	0.952686	0.952686
8000	2	0.194612	2.086399	2.013634	0.103587	0.093277	0.046638	1.878727	0.939364
8000	4	0.189442	1.943611	1.938978	0.054390	0.097469	0.024367	3.483031	0.870758
8000	8	0.200794	1.889880	1.914521	0.029597	0.106247	0.013281	6.784258	0.848032
16000	1	0.858424	9.058253	8.212416	0.812396	0.094767	0.094767	1.056657	1.056657
16000	2	0.847408	8.690921	8.277667	0.407085	0.097505	0.048752	2.081649	1.040825
16000	4	0.813358	8.166832	7.957057	0.219704	0.099593	0.024898	3.702065	0.925516
16000	8	0.782454	7.965039	7.807775	0.116675	0.098236	0.012280	6.706275	0.838284

统计图见plot文件夹

PA 3.6

一、解题思路及关键程序

1. 程序执行过程

令$m = \sqrt{p}$

按块分配后，每个进程执行一个n/m行n/m列的矩阵与一个n/m行1列的向量的乘法，得到一个n/m行1列的向量。之后对这些向量进行求和与连接，得到最终答案。

我将此过程分拆为scatter、计算及gather、释放本地内存三步，分别对应三个函数。为实现方便，将计算同一行、计算同一列的进程分别编为行进程组和列进程组。

```
MPI_Init(NULL, NULL);
MPI_Comm_size(comm, &comm_sz);
MPI_Comm_rank(comm, &my_rank);
m = sqrt(comm_sz);
MPI_Comm_split(comm, my_rank%m, my_rank, &col_comm);
MPI_Comm_rank(col_comm, &row_rank);
MPI_Barrier(comm);
MPI_Comm_split(comm, my_rank/m+m, my_rank, &row_comm);
MPI_Comm_rank(row_comm, &col_rank);
```

1. scatter：

(1) 矩阵发送。由于`MPI_Scatter`在发送数据量较多时会崩溃，故使用`MPI_Scatterv`发送矩阵，在发送前将矩阵按块一维化。

(2) 向量发送

负责计算矩阵位于同一列的块的进程所需要的向量相同，故负责相同列的进程编为一个列进程组，0号进程先将向量`scatter`到每个列进程组的0号进程，然后每个列进程组的0号进程将收到的向量`boardcast`到同组的其他进程。

```
void parallel_scatter(int n, double** matrix, double* vector, double**
local_mat, double** local_vec) {
    int m = sqrt(comm_sz);
    int local_n = n/m;
    int local_size = local_n * local_n;

    (*local_mat) = new double[local_size];
    if (my_rank == 0) {
        double *buf = new double[n*n];
        for (int x=0; x<m; x++)
            for (int y=0; y<m; y++)
                for (int i=0; i<local_n; i++)
                    memcpy(buf+(x*m+y)*local_size+i*local_n,
matrix[x*local_n+i]+y*local_n, sizeof(double)*local_n);
        int *sendcounts = new int[comm_sz];
        int *displs = new int[comm_sz];
        for (int i=0; i<comm_sz; i++)
            sendcounts[i] = local_size;
        for (int i=0; i<comm_sz; i++)
            displs[i] = local_size*i;
        MPI_Scatterv(buf, sendcounts, displs, MPI_DOUBLE, *local_mat,
local_size, MPI_DOUBLE, 0, comm);
        delete[] buf;
        delete[] sendcounts;
        delete[] displs;
    } else {
        MPI_Scatterv(NULL, NULL, NULL, MPI_DOUBLE, *local_mat, local_size,
MPI_DOUBLE, 0, comm);
    }
    (*local_vec) = new double[local_n];
    if (row_rank == 0)
        MPI_Scatter(my_rank ? NULL : vector, local_n, MPI_DOUBLE, *local_vec,
local_n, MPI_DOUBLE, 0, row_comm);
    MPI_Bcast(*local_vec, local_n, MPI_DOUBLE, 0, col_comm);
}
```

2. 计算及gather: 每个进程单独计算其本地矩阵和本地向量的矩阵乘法，通过`MPI_Reduce`汇总至所在行进程组的0号进程，这些进程再将行进程组内元素求和的结果`Gather`到所有进程的0号进程。

```
void parallel_solve(int n, double* local_mat, double* local_vec, double**
result) {
    int local_n = n/m;
    double *local_ans = new double[local_n];
    memset(local_ans, 0, sizeof(double)*local_n);
```

```
    for (int i=0; i<local_n; i++)
        for (int j=0; j<local_n; j++)
            local_ans[i] += local_mat[i*local_n+j]*local_vec[j];
    if (col_rank) {
        MPI_Reduce(local_ans, NULL, local_n, MPI_DOUBLE, MPI_SUM, 0, row_comm);
        goto end;
    }
    // reuse local_vec
    MPI_Reduce(local_ans, local_vec, local_n, MPI_DOUBLE, MPI_SUM, 0,
row_comm);
    if (row_rank) {
        MPI_Gather(local_vec, local_n, MPI_DOUBLE, NULL, local_n, MPI_DOUBLE,
0, col_comm);
        goto end;
    }
    (*result) = new double[n];
    MPI_Gather(local_vec, local_n, MPI_DOUBLE, *result, local_n, MPI_DOUBLE, 0,
col_comm);
end:
    delete[] local_ans;
}
```

3. 为输入矩阵大小不定，程序中大量使用堆上内存，需要及时释放。

```
void parallel_free(double* local_mat, double* local_vec) {
    delete[] local_mat;
    delete[] local_vec;
}
```

2.时间测试

此程序主要接口与PA3-5相同，故时间测试过程完全一致，此处略去代码。

二、测试结果

时间的单位为秒，输出的10个数分别对应下表的十项

维数	进程数	串行时间	总时间	通信时间	计算时间	加速比(总时间)	并行效率(总时间)	加速比(计算时间)	并行效率(计算时间)
2880	1	0.023212	0.065346	0.039562	0.025534	0.355216	0.355216	0.909064	0.909064
2880	4	0.025615	0.040102	0.032518	0.007038	0.638749	0.159687	3.639465	0.909866
2880	9	0.026370	0.032075	0.028361	0.003312	0.822139	0.091349	7.961705	0.884634
2880	16	0.025749	0.033598	0.029995	0.001917	0.766385	0.047899	13.432711	0.839544
5760	1	0.092803	0.261460	0.158614	0.102220	0.354941	0.354941	0.907877	0.907877
5760	4	0.102048	0.156712	0.127570	0.028321	0.651183	0.162796	3.603265	0.900816
5760	9	0.105206	0.127643	0.110500	0.013026	0.824220	0.091580	8.076618	0.897402
5760	16	0.102811	0.123880	0.114147	0.007410	0.829924	0.051870	13.874517	0.867157
11520	1	0.434127	1.048901	0.697273	0.407737	0.413888	0.413888	1.064723	1.064723
11520	4	0.511732	0.615950	0.534097	0.113306	0.830802	0.207700	4.516380	1.129095
11520	9	0.516152	0.477965	0.440540	0.052341	1.079895	0.119988	9.861334	1.095704
11520	16	0.482631	0.445913	0.414106	0.030011	1.082343	0.067646	16.081835	1.005115
23040	1	1.543371	4.485885	2.711762	1.626311	0.344051	0.344051	0.949001	0.949001
23040	4	1.757340	2.771027	2.127139	0.449785	0.634184	0.158546	3.907067	0.976767
23040	9	1.813664	2.220418	1.784283	0.209359	0.816812	0.090757	8.662931	0.962548
23040	16	1.836138	2.084168	1.698549	0.118316	0.880993	0.055062	15.518941	0.969934

统计图见plot文件夹