# 课程报告

19335015 陈恩婷

## 各种机制的设计和实现的思路和方法

### 1. MESH 的数据结构

Mesh 的数据结构主要有两种，一种是直接用数组存储，一种是构建 Mesh 类，创建 Elements、Maps、Data 等结构体成员来进行存储。

#### a. 数组存储

数组存储的方式主要实现于 Mesh.cpp 的 ReadRaw、WriteRaw、readvtk、savevtk 函数中，主要需要处理的整数与数组包括：

```
int  nnode, ncell, nedge, nbedge;
int* becell, * ecell, * bound, * bedge, * edge, * cell, *cell2edge;
real* x;
int  npoint, ncell,cellsize;
int* cells, *celltype;
double* xyz;
int * q;
```

注意这里的数组为指针形式，需要为其动态分配内存。

#### b. 使用 Mesh 类中的结构体成员存储

结构体成员存储的方式主要实现于 Mesh.cpp 中 Mesh 的成员函数中，Mesh 类的数据成员和各结构体的声明如下：

```
typedef double real;
typedef struct {
    int index;             /* index */
    int size;              /* number of elements in set */
    char const name[ 30 ]; /* name of set */
} elements;

typedef elements* Elements;

typedef struct {
    int index;             /* index */
    Elements from,         /* set pointed from */
        to;                /* set pointed to */
    int dim,               /* dimension of pointer */
        *map;              /* array defining pointer */
    char const name[ 30 ]; /* name of pointer */
} map;

typedef map* Map;
```

```
typedef struct {
    int index;            /* index */
    Elements set;         /* set on which data is defined */
    int dim,              /* dimension of data */
        size;             /* size of each element in dataset */
    char* data;           /* data */
    char const type[ 16 ], /* datatype */
        name[ 30 ];       /* name of dataset */
} dat;

typedef dat* Data;

class Mesh {
   public:
    int element_list_size, map_list_size, dat_list_size,
        element_list_index, map_list_index, dat_list_index;
    Elements* element_list;
    Map* map_list;
    Data* dat_list;
```

Mesh.cpp 中还实现了从简单的数组存储方式转换为结构体存储的方法，主要代码如下：

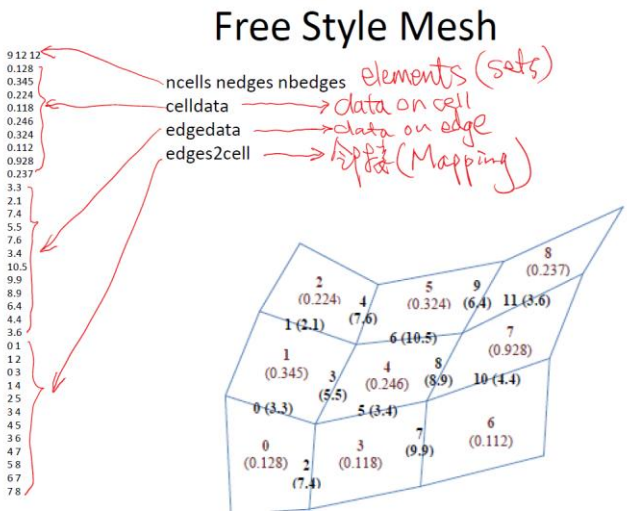
```
Elements Mesh::makeElements(int ele_size, char const* ele_name) {
    element_list[element_list_index] = (Elements)malloc(sizeof(elements));
    strcpy((char*)(element_list[element_list_index]->name), ele_name);
    element_list[element_list_index]->size = ele_size;
    element_list[element_list_index]->index = element_list_index;
    element_list_index++;
    return element_list[element_list_index-1];
}
Map Mesh::makeMap(Elements map_from, Elements map_to, int map_dim, int* map_map,
char const* map_name) {
    int arr_cnt = map_from->size;
    map_list[map_list_index] = (Map)malloc(sizeof(map));
    map_list[map_list_index]->index = map_list_index;
    map_list[map_list_index]->from = map_from;
    map_list[map_list_index]->to = map_to;
    map_list[map_list_index]->dim = map_dim;
    //map_list[map_list_index]->map = map_map;
    map_list[map_list_index]->map = (int*)malloc(sizeof(int) * map_dim * arr_cnt);
    for (int i = 0; i < arr_cnt * map_dim; i++)
        map_list[map_list_index]->map[i] = map_map[i];
    strcpy((char*)(map_list[map_list_index]->name), map_name);
    map_list_index++;
    return map_list[map_list_index - 1];

}
Data Mesh::makeData(Elements data_set, int data_dim, char const* data_type, char*
data_data, char const* data_name) {
    Data data = (Data)malloc(sizeof(dat));
    data->index = dat_list_index;
    data->set = data_set;
    data->dim = data_dim;
```

```
    //data->size = 8;   //xxxxxxxxxxx    ////////////////////////////
    //data->data = data_data;
    if (!strcmp(data_type, "int")) {
        data->size = 4;
        int* t = (int*)malloc(sizeof(int) * data->set->size * data->dim);
        for (int j = 0; j < data->set->size * data->dim; j++)
            t[j] = ((int*)data_data)[j];
        data->data = (char*)t;
    }
    if (!strcmp(data_type, "double")) {
        data->size = 8;
        double* t = (double*)malloc(sizeof(double) * data->set->size * data->dim);
        for (int j = 0; j < data->set->size * data->dim; j++)
            t[j] = ((double*)data_data)[j];
        data->data = (char*)t;
    }
    strcpy((char*)(data->name), data_name);
    strcpy((char*)(data->type), data_type);
    dat_list[dat_list_index++] = data;
    return data;
}
```

## 2. MESH 的文件读写

Mesh 的文件存储形式一共有四种，分别是 freestyle(.dat)、VTK(.vtk)、ASCII EMD(.emd)和二进制 EMD(.emd)。其中前两种针对的是简单数组形式的 Mesh 存储结构，后两种是针对类的结构体成员形式的存储结构。前三种为文本文件格式，主要用 malloc、fscanf、fprintf 等语言机制实现，最后一种为二进制文件格式主要用 malloc、fread、fwrite 等语言机制实现。下面附上每种方法的简要介绍和部分实现代码。

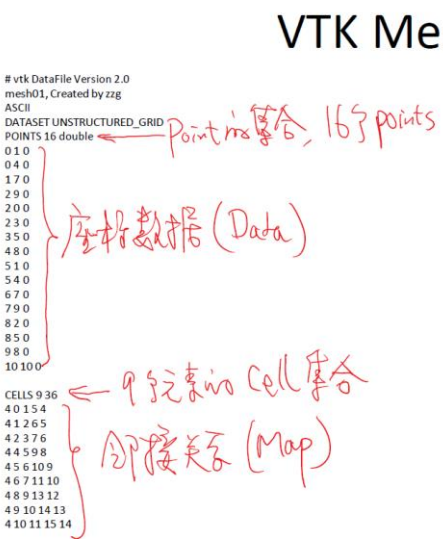
### a. Freestyle Mesh

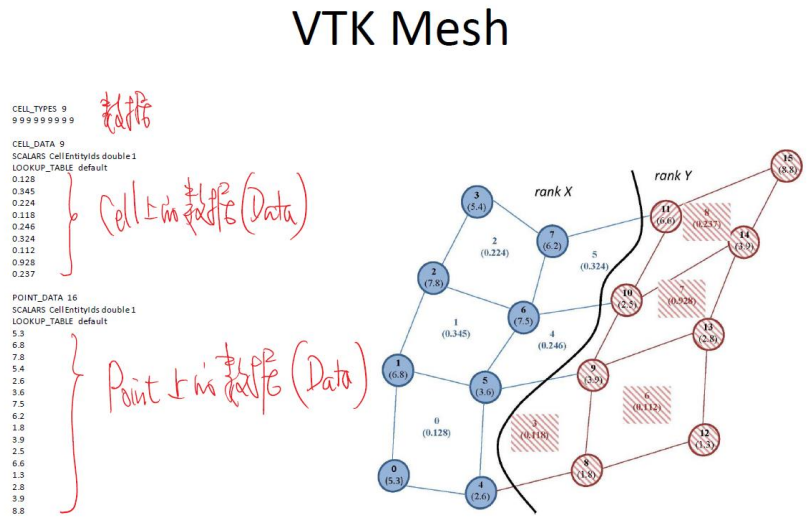


核心代码如下：

```
bool Mesh2Dquad::read(const char* fileName) {
    printf("reading in grid \n");
    FILE* fp;
    if ((fp = fopen(fileName, "r")) == NULL) {
        printf("can't open file\n");
        return 0;
    }
    readHeader(fp);
    init();
    readNode(fp);
    readCell(fp);
    readEdge(fp);
    readBEdge(fp);
    fclose(fp);
    return true;
}
bool Mesh2Dquad::readHeader(FILE* fp) {
    if ( fscanf( fp, "%d %d %d %d \n", &nnode, &ncell, &nedge, &nbedge ) != 4 ) {
        printf( "error reading from new_grid.dat\n" );
        exit( -1 );
    }
    cell = (int*)malloc( 4 * ncell * sizeof( int ) );
    edge = (int*)malloc( 2 * nedge * sizeof( int ) );
    ecell = (int*)malloc( 2 * nedge * sizeof( int ) );
    bedge = (int*)malloc( 2 * nbedge * sizeof( int ) );
    becell = (int*)malloc( nbedge * sizeof( int ) );
    bound = (int*)malloc( nbedge * sizeof( int ) );

    x = (real*)malloc( 2 * nnode * sizeof( real ) );
    cell2edge = (int*)malloc( 4 * ncell * sizeof( int ) );
    for ( int i = 0; i < 4 * ncell; ++i ) {
        cell2edge[ i ] = -1;
    }
    return 1;
}
bool Mesh2Dquad::readNode(FILE* fp) {
    for ( int n = 0; n < nnode; n++ ) {
        if ( fscanf( fp, "%lf %lf \n", &x[ 2 * n ], &x[ 2 * n + 1 ] ) != 2 ) {
            printf( "error reading from new_grid.dat\n" );
            exit( -1 );
        }
    }
    return 1;
}
```

b. VTK Mesh




主要使用的语言机制有 malloc、fprintf 和 fscanf 等库函数。

## malloc

```
Defined in header <stdlib.h>
void* malloc( size_t size );
```

## fprintf

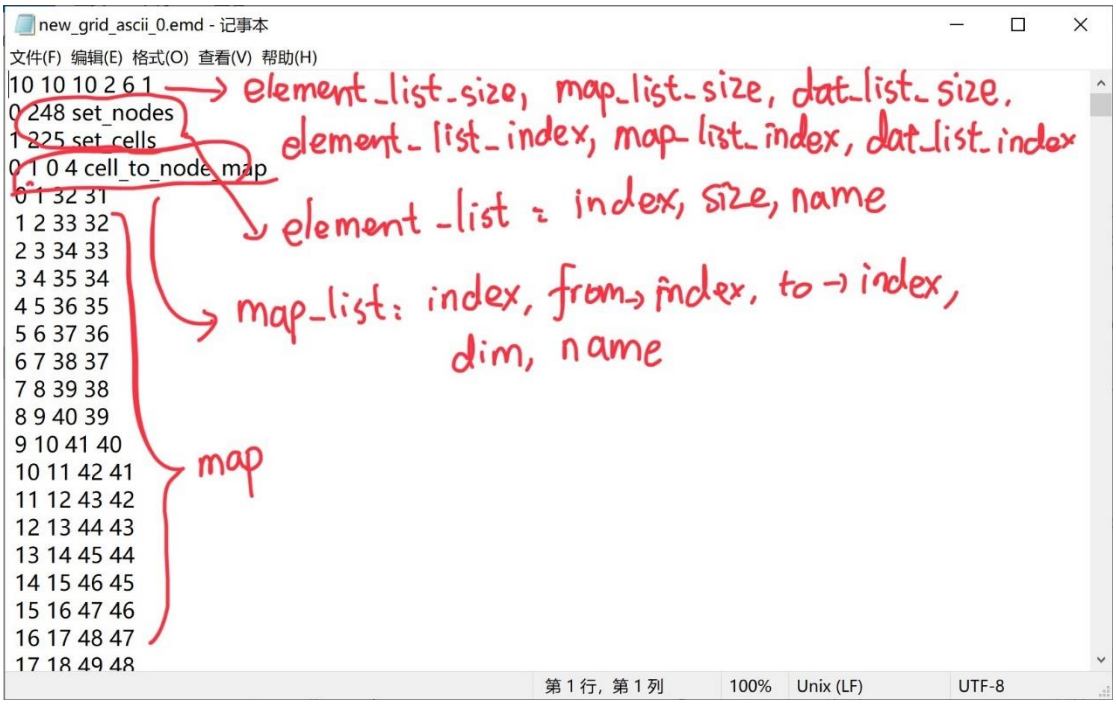
```
int fprintf ( FILE * stream, const char * format, ... );
```

## fscanf

```
int fscanf ( FILE * stream, const char * format, ... );
```

fscanf 和 fprint 我在之前并不是很熟悉，不过它的主要语法和 printf、scanf 类似，比较容易上手。

## c. ASCII EMD




```
bool Mesh::writeElements(FILE * fp) {   ////////////////////
    // using fprintf
    for ( int i = 0; i < element_list_index; i++ ){
        fprintf(fp, "%d %d %s\n", element_list[i]->index, element_list[i]->size,
element_list[i]->name);
    }
    return 1;
}

bool Mesh::writeMaps(FILE * fp) {    ////////////////////////
    // using fprintf
    for (int i = 0; i < map_list_index; i++) {
        fprintf(fp, "%d %d %d %d %s\n", map_list[i]->index,
map_list[i]->from->index,
            map_list[i]->to->index, map_list[i]->dim, map_list[i]->name);
        for (int j = 0; j < map_list[i]->from->size; j++) {
            for (int k = 0; k < map_list[i]->dim; k++) {
                fprintf(fp, " %d", map_list[i]->map[j * map_list[i]->dim + k]);
            }
            fprintf(fp, "\n");
        }
        fprintf(fp, "\n");
    }
    return 1;
}
```

```
bool Mesh::writeData(FILE * fp) {    ////////////////
    // using fscanf
    for ( int i = 0; i < dat_list_index; i++ ){
        fprintf(fp, "%d %d %d %d %s\n", dat_list[i]->index, dat_list[i]->set->index,
            dat_list[i]->dim, dat_list[i]->size, dat_list[i]->name);
        if (dat_list[i]->size == 8 ){
            for ( int k = 0; k < dat_list[i]->set->size; k++ ){
                for ( int j = 0; j < dat_list[i]->dim; j++ ){
                    fprintf(fp, " %f", dat_list[i]->data[k * dat_list[i]->dim + j]);
                }
                fprintf(fp, "\n");
            }
        }
        else if (dat_list[i]->size == 4 ){
            for ( int k = 0; k < dat_list[i]->set->size; k++ ){
                for ( int j = 0; j < dat_list[i]->dim; j++ ){
                    fprintf(fp, " %d", dat_list[i]->data[k * dat_list[i]->dim + j]);
                }
                fprintf(fp, "\n");
            }
        }
        fprintf(fp, "\n");
    }
    return 1;
}
```

d. Binary EMD

二进制 EMD 的文件内容大概就是把前面的 ASCII VTK 的内容存成二进制文件。主要代码如下：

```
bool Mesh::writeHeaderbin(FILE * fp) {   ///////////////////////
    int header[6] = { element_list_size, map_list_size, dat_list_size,
            element_list_index, map_list_index, dat_list_index };
    int count = fwrite(header, sizeof(int), 6, fp);
    if (count != 6)
        return 0;
    return 1;
}

bool Mesh::writeElementsbin(FILE * fp) {   ////////////////////
    for (int i = 0; i < element_list_index; i++) {
        int count = fwrite(element_list[i], sizeof(elements), 1, fp);
        if (count != 1)
            return 0;
```

```
        }
        return 1;
    }

    bool Mesh::writeMapsbin(FILE * fp) {    ////////////////////////
        for (int i = 0; i < map_list_index; i++) {
            int temp[5] = { map_list[i]->index, map_list[i]->from->index,
                map_list[i]->to->index, map_list[i]->dim,strlen(map_list[i]->name) };
            int count = fwrite(temp, sizeof(int), 5, fp);
            if (count != 5)
                return 0;
            count = fwrite(map_list[i]->name, sizeof(char), temp[4], fp);
            if (count != temp[4])
                return 0;
            count = fwrite(map_list[i]->map, sizeof(int), map_list[i]->from->size *
    map_list[i]->dim, fp);
            if (count != map_list[i]->from->size * map_list[i]->dim)
                return 0;
        }
        return 1;
    }
```

# 各种问题的讨论和可能的其它设想

1. MakeMap 函数的实现

   makeMap 中，可以选择申请 arr_cnt 个 map 的空间，将数据分别存储与各个 map 中，或者只申请一个 map 的空间，将数据存储于它的 map 数据成员中。如下所示：

```
Map makeMap(Elements map_from, Elements map_to, int map_dim, int* map_map, char const*
map_name) {
    int arr_cnt = map_from->size;
    Map mapi = (Map)malloc(sizeof(map) );
    mapi->index = map_list_index;
    mapi->from = map_from;       mapi->to = map_to;
    mapi->dim = map_dim;  mapi->map = map_map;
    mapi->name = map_name;
    map_list[map_list_index++] = (Map)mapi;
    return (Map)mapi;
}
Map makeMap(Elements map_from, Elements map_to, int map_dim, int* map_map, char const*
map_name) {
    int arr_cnt = map_from->size;
    Map mapi = (Map)malloc(sizeof(map) *arr_cnt);
```

```
    for (int j= 0; j< arr_cnt; j++) {
            mapi[j].index = map_list_index;      mapi[j].from = map_from;
            mapi[j].to = map_to;          mapi[j].dim = map_dim;
            for (int i = 0; i < map_dim; i++)
                    mapi[j].map[i] = map_map[j * map_dim+i];
            mapi->name = map_name;
    }
    map_list[map_list_index++] = (Map)mapi;
    return (Map)mapi;
}
```

个人认为只申请一个 map 的方法比较好，因为这样节省了存储空间，也简化了申请和释放内存的工作量。

2. Mesh 相关的并行与分布式计算

主要问题：

a. 划分出的子 Mesh 所存储的编号是全局编号，不能用于子 Mesh 的查找

b. 每个进程所需要用到的数据不一定划分给了该进程

主要的解决思路：

1. 为每个进程申请足够的内存存放全局 Mesh
2. 先将全局 Mesh 里的数据重新编号，再划分给各个进程
3. 通过进程间通信的方式帮助每个进程获得需要的数据

本人之前尝试实现了重新编号的解决方案，主要思路按照如下思路实现：

```
double* new_x = (double*)malloc(4 * g_ncell * 3 * sizeof(double));

for (int i = 0; i < g_ncell; i++) { //////////////
memcpy(new_x + (4 * i + 0) * 3, g_x + (g_cell[4 * i + 0]) * 3, 3 * sizeof(double));
memcpy(new_x + (4 * i + 1) * 3, g_x + (g_cell[4 * i + 1]) * 3, 3 * sizeof(double));
memcpy(new_x + (4 * i + 2) * 3, g_x + (g_cell[4 * i + 2]) * 3, 3 * sizeof(double));
memcpy(new_x + (4 * i + 3) * 3, g_x + (g_cell[4 * i + 3]) * 3, 3 * sizeof(double));
}

memcpy(g_x, new_x, 4 * g_ncell * 3 * sizeof(double));

...

free(new_x);
```

我主要探索了 area 计算部分的并行化，先创建 new_x 作为 renumber 后的 g_x 数组，填充好其数据后，再分配到各个进程的 x_for_area 数组中，计算好结果存到 q 和 adt 数组后，后续步骤再只用 0 号进程进行计算。

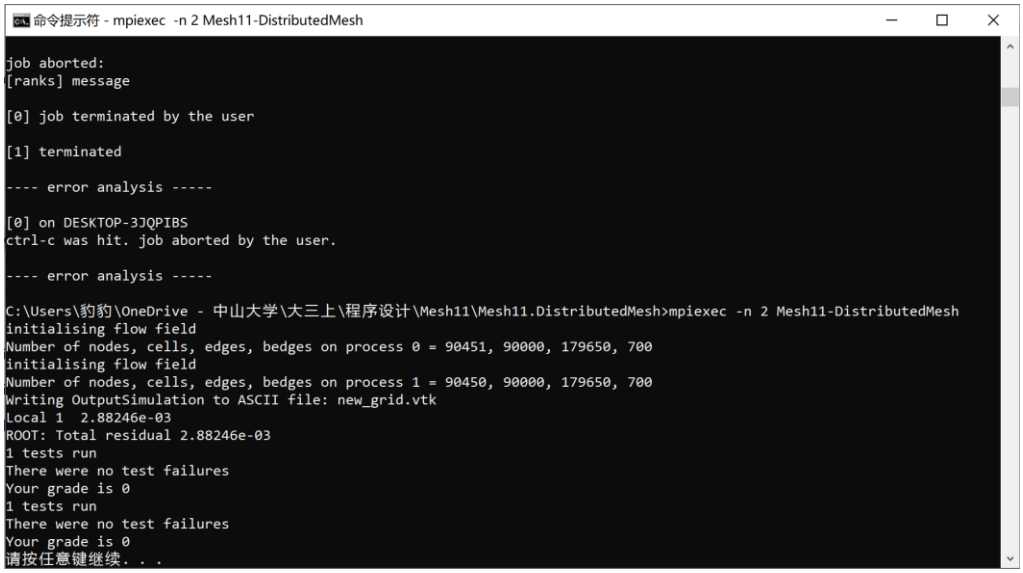
```
new_x = (double*)malloc(4 * g_ncell * 3 * sizeof(double));
for (int i = 0; i < g_ncell; i++) { //////////////
    memcpy(new_x + (4 * i + 0) * 3, g_x + (g_cell[4 * i + 0]) * 3, 3 * sizeof(double));
    memcpy(new_x + (4 * i + 1) * 3, g_x + (g_cell[4 * i + 1]) * 3, 3 * sizeof(double));
    memcpy(new_x + (4 * i + 2) * 3, g_x + (g_cell[4 * i + 2]) * 3, 3 * sizeof(double));
    memcpy(new_x + (4 * i + 3) * 3, g_x + (g_cell[4 * i + 3]) * 3, 3 * sizeof(double));
}
```

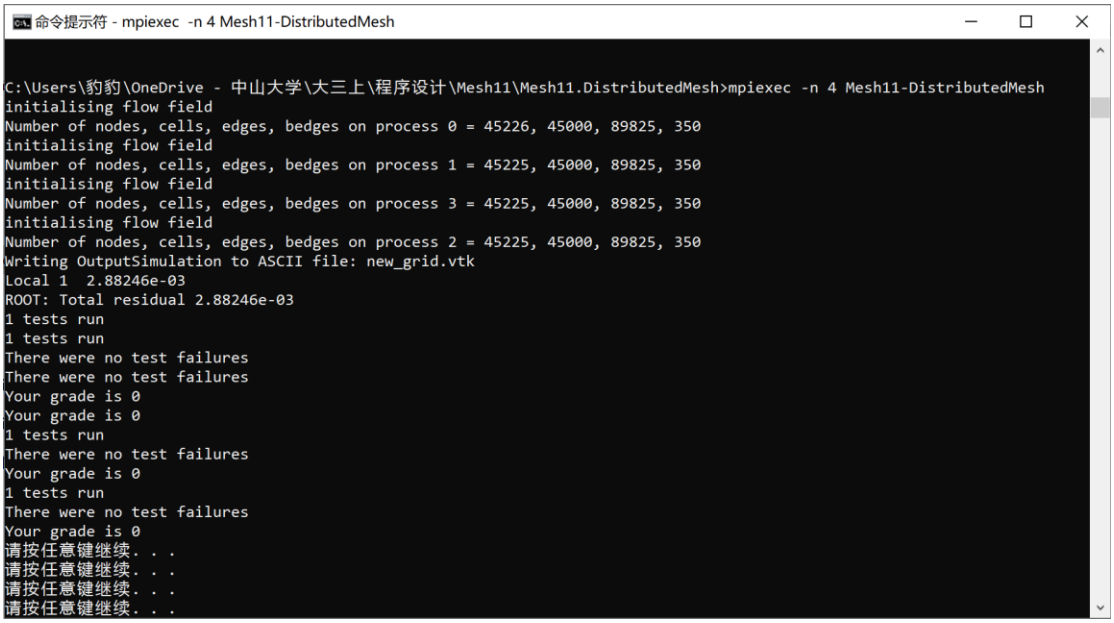
```
MPI_Barrier(MPI_COMM_WORLD);

for (int k = 0; k < 2; k++) {
    for (int i = 0; i < ncell; i++) { //////////////
        area(
            x_for_area + (4 * i) * 3,
            x_for_area + (4 * i + 1) * 3,
            x_for_area + (4 * i + 2) * 3,
            x_for_area + (4 * i + 3) * 3,
            q + 4 * i,
            adt + i);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    gather_double_array(g_q, q, comm_size, g_ncell, ncell, 4);
    gather_double_array(g_adt, adt, comm_size, g_ncell, ncell, 1);
```

为了保持并行化运行和串行程序运行结果一样，还需要修正这行代码：

```
rms = sqrt(rms / (double)g_ncell);
```


如图所示，串行程序的代码运行结果正确。以下是用 mpiexec 并行化运行的结果：






如图所示，并行程序运行结果正确。

3. 其他的问题和解决方法

   1. 读入文件时，同一行既有字符串又有整数会导致读入失败
      解决方法：仔细检查发现，由于文件中有空行，读取时需要跳过这些空行，否则实际读到的与预想的不是同一行，就会读不到想要的数据。

   2. 写入文件时，cells 数组每个数字占一行，没有像原文件一样一个 cell 占一行
      解决方法：celltype 中存储了每个 cell 是哪个类型，1 表示 cell 由两个 point 构成，3 表示由三个构成，5 表示由 4 个构成。根据这些信息。可以按照原文件的格式写入 cells 数组中的信息。

   3. 将文件中 mesh 的名字读到 dataset 字符串中时，由于文件中 mesh 的名字后面紧跟着一个逗号，读取时会将逗号也读进去
      解决方法：参考 stackoverflow 上面的回答，使用%[^,]就可以解决了

      If the order of your variables in the string is fixe, I mean It's always:

      ```
      string, string, string, int, float
      ```

      the use the following format specifier in `sscanf()`:

      ```
      int len = strlen(str);
      char a[len];
      char b[len];
      char c[len];
      unsigned long d;
      float e;

      sscanf(" %[^,] , %[^,] , %[^,] , %lu , %lf", a, b, c, &d, &e);
      ```

4. 实现过程中，测试运行时经常出现内存错误
   解决方案：这些大多是没有正确分配或者索引数组引起的。利用 Visual Studio 的调试功能，检查具体是哪一处内存出错，修改相关代码即可。

## 参考资料


1. https://stackoverflow.com/questions/16014859/sscanf-until-it-reaches-a-comma
2. https://en.cppreference.com/w/c/memory/malloc
3. https://www.cplusplus.com/reference/cstdio/fprintf/
4. https://www.cplusplus.com/reference/cstdio/fscanf/
5. https://www.cplusplus.com/reference/cstdio/fread/
6. https://www.cplusplus.com/reference/cstdio/fwrite/