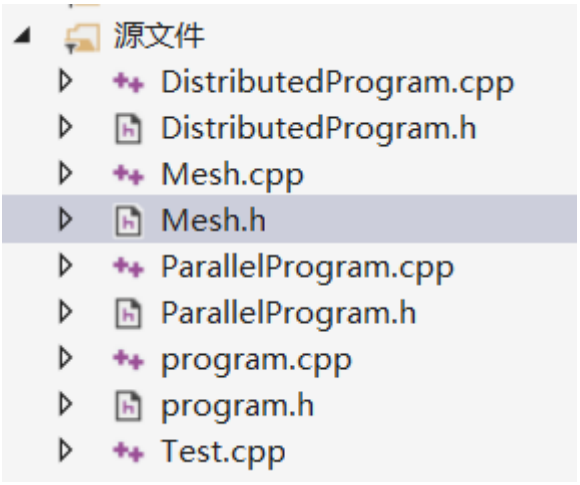
课程报告

概述

将本学期进行的实验编写为了一个 mesh 项目。其内容包括：对自由格式、vtk 格式、emd 格式的文件的读，读取后保存为 elements、maps、datas（由着 3 种元素构成 mesh），能够写为 emd 文件。能够对 mesh 进行核函数的操作，包括串行和并行两种方式。

实验过程

整体框架

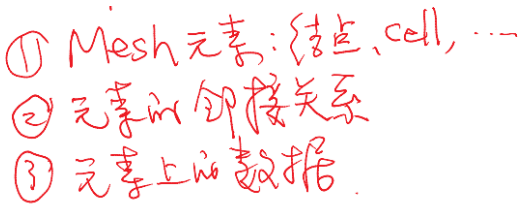


整个项目由以上文件组成，Mesh 类封装了 elements、maps、datas 等构成 mesh 的成员变量以及读写函数，program 类封装了对 mesh 进行的串行核函数操作，ParallelProgram 类封装了对 mesh 进行的并行核函数操作（使用所有线程共享 global mesh 的方法）DistributedProgram 类封装了对分布式 mesh 进行并行核函数操作（使用 partitonTo，partitionFrom 函数实现对 mesh 的划分）。

Mesh 类

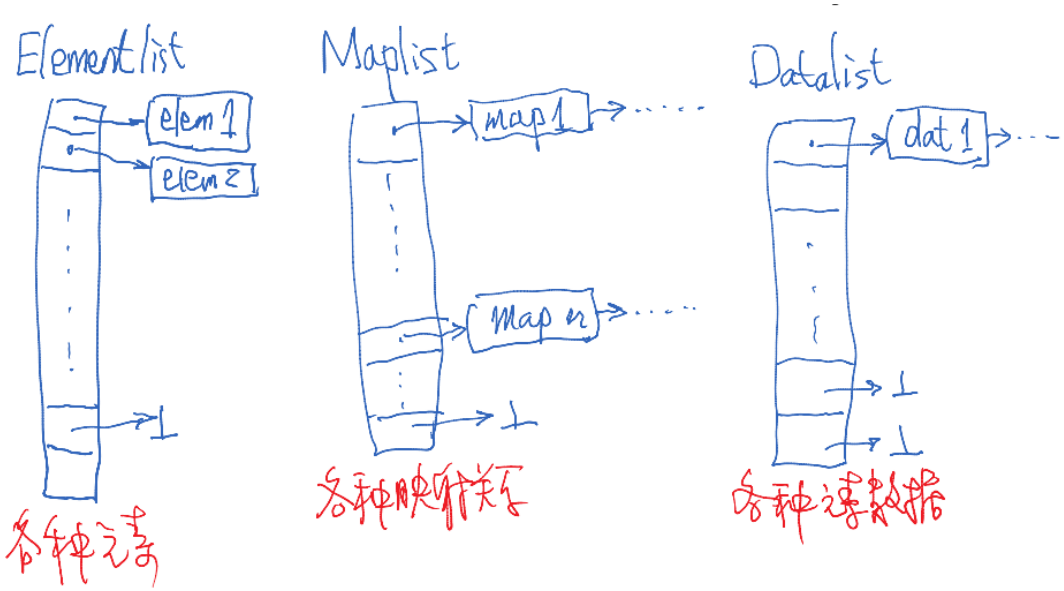
1. elements、maps、datas 构成 mesh

mesh 可以看成**由 3 要素组成:**



上面说的就是 element map data。

为了更好的存储管理 element map data，我们创建 3 个 list 来存储它们：



这里使用数组来创建 list，也就是说 list 是固定大小的，对于未存入元素的位置，置为 null。每个 list 中存储指针。

这 3 个 list 组成了完整的 mesh 文件。

element 的数据结构如下：

```
1. typedef struct {
2.   int index;         /* index */
3.   int size;          /* number of elements in set */
4.   char const name[16]; /* name of set */
5. }  elements;
```

可以看到，element 由 其在 list 中的下标 index、element 的数量、element 集的名称 组成

map 的数据结构如下：

```
1. typedef struct {
2.   int index;         /* index */
3.   Elements from,       /* set pointed from */
4.    to;             /* set pointed to */
5.   int dim,           /* dimension of pointer */
6.    * map;          /* array defining pointer */
7.   char const name[16]; /* name of pointer */
8. } map;
```

可以看到，map 由 其在 list 中的下标 index、映射关系中的两个 element（这里指的是指针）、映射关系的维度、映射关系集合、map 集的名称 组成

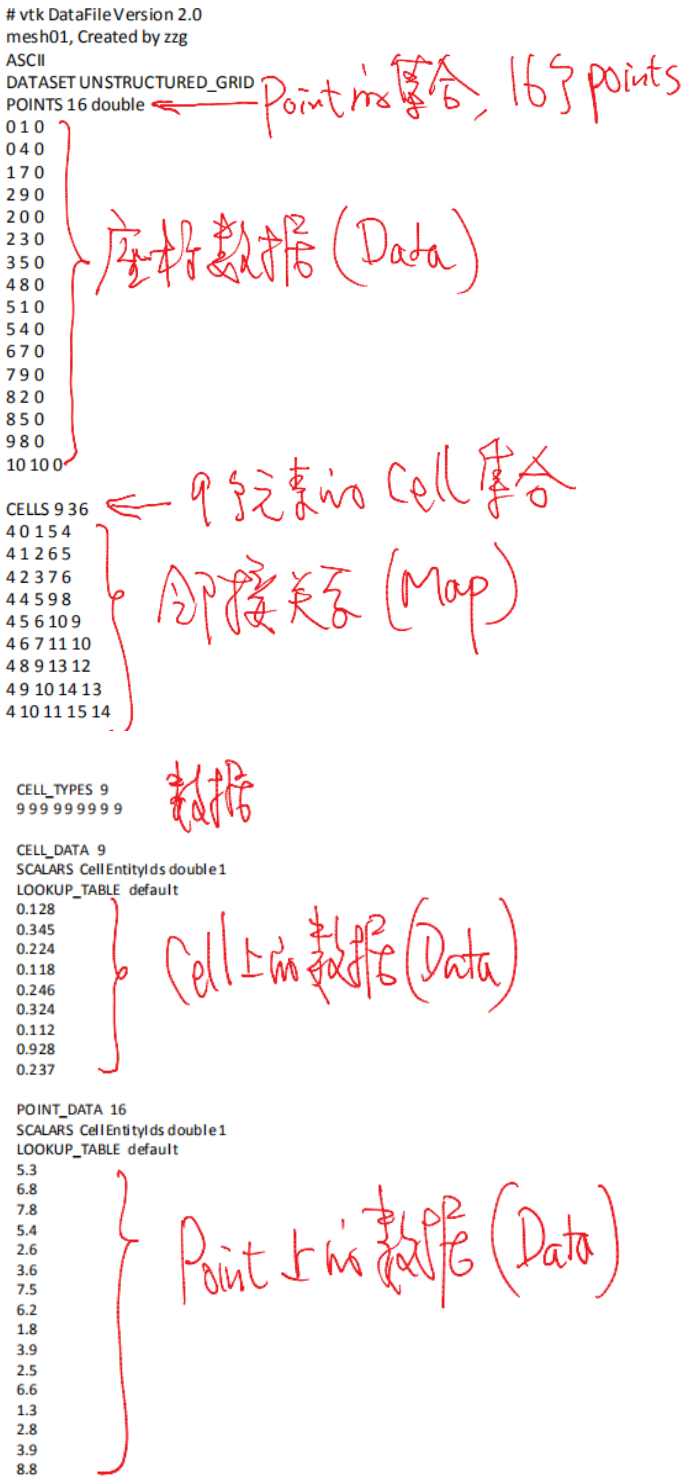
data 的数据结构如下：

```
1. typedef struct {
2.   int index;         /* index */
3.   Elements set;        /* set on which data is defined */
```

```
4.  int dim,          /* dimension of data */
5.   size;         /* size of each element in dataset */
6.  char* data;       /* data */
7.  char const type[16],name[16];        /* name of dataset */
8. } dat;
```

可以看到，data 由 其在 list 中的下标 index、对应描述的 element（这里指的是指针）、数据的维度、单个数据的大小、数据集、data 集的名称 组成

我们需要存储的数据如下：（以 vtk 格式为例）



points 部分为 points 的坐标数据，属于 data
cells 部分为每个 cell 对应的 points，为 cell 到 points 的映射关系，属于 map
cell_types 部分描述的是 cell 的类型，属于 cell 的 data
cell_data 和 point_data 顾名思义，属于 data
element 指的是 mesh 的元素，包括 cells、points，这里我们将 cells 的数量 ncell 存储到 cell 的 element 中，points 的数量 npoint 存储到 points 的 element 中

存储数据的话，就面临着**使用二维数组还是一维数组存储**的问题。
实验中，统一使用一维数组来表示二维数组，这因为二维数组不好和指针强行转化，在参数传递上也很麻烦，如果用动态方法分配二维数组，比较麻烦。
而一维数组可以轻松转换成二维数组—只需在下标表示上动用些技巧即可，所以我们均选择使用一维数组。
例如，对于 mxn 的二维数组，一维数组的下标 i*n+j 表示二维数组的第 i 行第 j 列。

存储的时候需要注意，**所有数据都需要进行深拷贝**而不是浅拷贝。若是单个数据，则直接进行赋值即为深拷贝；若是数组，则需要注意。
这里的数组元素为 map 集、data 集、name。
name 为 string，使用 strcpy 实现深拷贝：

```
1. strcpy((char*)(ele->name), ele_name);
```

map 集，我们需要用 malloc 为其申请内存空间，再一个元素一个元素复制过去：

```
1. map_list[map_list_index]->map = (int*)malloc(sizeof(int) *
   map_dim * arr_cnt);
2.  for (int i = 0; i < arr_cnt * map_dim; i++)
3.   map_list[map_list_index]->map[i] = map_map[i];
```

data 集同理：

```
1. if (!strcmp(data_type, "int")) {
2.   data->size = 4;
3.   int* t = (int*)malloc(sizeof(int) * data->set->size * dat
   a->dim);
4.   for (int j = 0; j < data->set->size * data->dim; j++)
5.    t[j] = ((int*)data_data)[j];
6.   data->data = (char*)t;
7.  }
8.  if (!strcmp(data_type, "double")) {
9.   data->size = 8;
10.  double* t = (double*)malloc(sizeof(double) * data->set->s
   ize * data->dim);
11.  for (int j = 0; j < data->set->size * data->dim; j++)
12.   t[j] = ((double*)data_data)[j];
13.  data->data = (char*)t;
14. }
```

注意到，我们对 data 集的存储做了一些小处理。
我们首先按照 data 的 type，来正常进行对应数据类型的深拷贝，然后将 data 集赋值为 char*

强制转换后的深拷贝数组。
这是因为 **dat 结构体中的 data 可能是不同种类型的数据**，如 int、double。为了能兼容所有的数据类型，dat 的定义中，将成员变量 data 定义为 char*类型，这是因为说到底这些数据类型的不同，体现在一个数据占几个字节，而 char 占 1 个字节，所有数据类型都可以化成 char 类型。
在要用到这些数据的时候，再将 char 类型转换为原来的数据类型即可。例如，计算 sum 的时候：

```
double sum = 0;
if (dcell2->size == 4) {
 int* temp = (int*)dcell2->data;
 for (int i = 0; i < ncell; i++) {
  sum += temp[i];
 }
}
else if (dcell2->size == 8) {
 double* temp = (double*)dcell2->data;
 for (int i = 0; i < ncell; i++) {
  sum += temp[i];
 }
}
```

首先根据 dat 的 size 判断数据类型是什么，然后将 data 转换为该数据类型再使用即可。

2.读取自由格式文件

使用 fscanf 函数和 fprintf 函数实现对文本文件的读写。

fscanf()函数用于将文件流中的数据格式化输入，其原型为：
int fscanf(FILE * stream, char *format [, argument ,...]);
其中，stream 为文件指针，format 为格式化字符串，argument 为格式化控制符对应的参数。
fscanf()函数根据指定的格式(format)，将输入流(stream)的数据存入地址(argument)
例如，读取 header 的时候，需要读取的 header 为 4 个 int 类型整数，中间用空格相隔，所以 format 使用“%d %d %d %d”即可，读取后，存储到 nnode、ncell、nedge、nbedge 中。
fscanf 函数返回成功匹配和赋值的个数，所以，若 fscanf 返回的数不为 4，则表示读取出错，此时 return false

文本文件的写操作和读操作类似，用 fprintf 代替 fscanf 即可
fprintf 函数原型：
int fprintf (FILE* stream, const char*format, [argument])
其中，stream 为文件指针，format 为格式化字符串，argument 为格式化控制符对应的参数。
fprintf()函数根据指定的格式(format)，向输出流(stream)写入数据(argument)。

read 函数的具体规划：
使用一个 `Mesh::readraw` 函数进行总的 read 操作（外部调用此函数进行 read），在该函数中，再调用 read 各个部分的函数，完成对各个部分的 read。

我们需要自己创建 cells 等数组来存储数据（因为 mesh 包含的成员变量只有 element map data），读取完数据后，再用数据创建相应的 element、map、data，由此来构 mesh。
创建好的 mesh 的各个 list 存储的数据如下：

```
s.makeElements(nnode, "nodes");   //0
s.makeElements(nedge, "edges");   //1
s.makeElements(nbedge, "bedges"); //2
s.makeElements(ncell, "cells");   //3
```

注意这里代码的顺序代表了其存储的顺序！

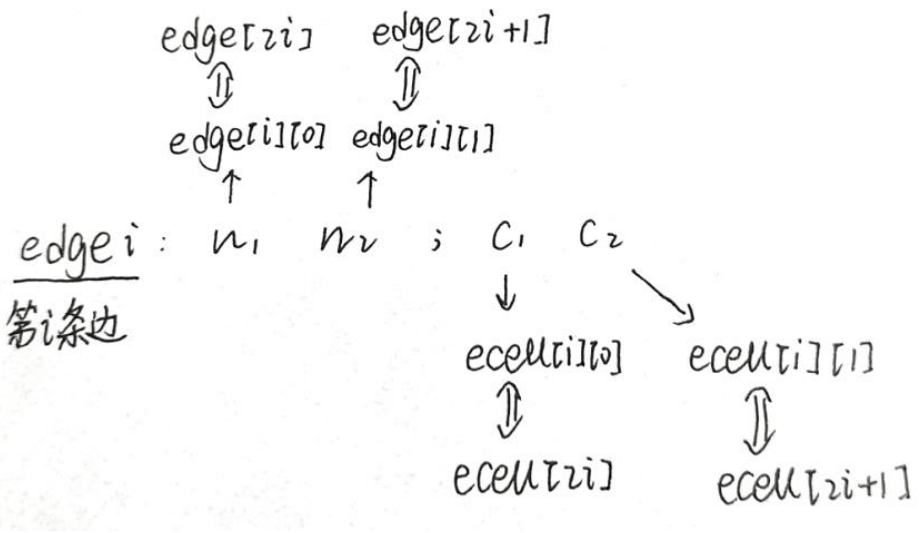
```
s.makeMap(s.element_list[1], s.element_list[0], 2, edge , "pedge");    0
s.makeMap(s.element_list[1], s.element_list[3], 2, ecell, "pecell");   1
s.makeMap(s.element_list[2], s.element_list[0], 2, bedge, "pbedge");   2
s.makeMap(s.element_list[2], s.element_list[3], 1, becell, "pbecell"); 3
s.makeMap(s.element_list[3], s.element_list[0], 4, cell, "pcell");     4
```

```
s.makeData(s.element_list[2], 1, "int", (char*)bound, "p_bound"); 0
s.makeData(s.element_list[0], 3, "double", (char*)x, "p_x");      1
s.makeData(s.element_list[3], 4, "double", (char*)q, "p_q");      2
s.makeData(s.element_list[3], 4, "double", (char*)qold, "p_qold"); 3
s.makeData(s.element_list[3], 1, "double", (char*)adt, "p_adt");  4
s.makeData(s.element_list[3], 4, "double", (char*)res, "p_res");  5
```

3.读取 vtk 格式文件

read 函数的具体规划：
使用一个 Mesh::readvtk 函数进行总的 read 操作(外部调用此函数进行 read)，在该函数中，再调用 read 各个部分的函数，完成对各个部分的 read。
我们需要自己创建 cells、xyz 等数组来存储数据（因为 mesh 包含的成员变量只有 element map data），读取完数据后，再用数据创建相应的 element、map、data，由此来构 mesh。

vtk 文件中没有 edge 和 bedge 的信息，我们需要从 cell 中获取边的信息来构建 edge 和 bedge。
首先，我们先理清边和 edge、ecell 的关系。
自由格式中，edge 为 nedge*2 大小的二维数组，每行存储一条边对应的 2 个 nodes；ecell 为 nedge*2 大小的二维数组，每行存储该行编号的边对应的 2 个 cells。也就是说，edge 和 ecell 的每一行是一一对应的，都用于描述同一条边，关系图示：



那么如何由 cells 获得 edge 和 ecell 呢?
cells 中每行存了一个 cell 的 4 个 points：n1 n2 n3 n4，由这 4 个 points 我们可以获得这个 cell 的 4 条边：<n1,n2><n2,n3><n3,n4><n4,n1>
那么这个 cell 就是这 4 条边的一个 ecell。

我们可以遍历整个 cells，那么我们就可以获得所有的边和对应的 ecell。
这些边中有重复的，而所有的边在遍历过程中都至多出现 2 次（因为一个边最多与 2 个 cell 关联）。
当一条边第一次出现时，我们将赋予这条边一个编号 i，将它的 2 个 points 存入 edge 的第 i 行，将本次的 cell 存在 ecell 的第 i 行的第一个位置。当一条边第二次出现的时候，我们要能判断出来它已出现过，并获得它的编号 i，然后将本次的 cell 存在 ecell 的第 i 行的第二个位置。
需要注意的是，一条边的 edge 和 ecell 的行数是对应的（可以参考上图）。

那么我们需要能判断出来一条边是否已经出现过，并能将边（由 2 个 points 表示）与边的编号对应起来。这里我使用 map<pair<int,int>,int>来实现，将<n1,n2>映射到边的编号。

这里还需要注意一个问题，nedge 我们是不知道的，但是要存储数据就要先为 edge 和 ecell 分配内存，这就需要知道 nedge。
对于这个问题我思考了两种方法来解决：

1. 为 edge 和 ecell 分配足够大的空间：2*npoint*(npoint-1)*sizeof(int)，由排列组合可知，nedge 最大为 npoint*(npoint-1)。
2. 先使用 map 按照上述算法遍历一边 cell，只不过这边遍历中不存储数据。在本次遍历中，记录编号数，即可获得 nedge。

第一种方法是牺牲了空间，第二种方法是牺牲了时间。经过比较，我决定采用第 2 种方法，这是因为我们要处理的数据一般量很大，npoint^2 是一个很大的数，这样会导致 cell 排列紧密（导致边数少）时浪费的空间较多。而第二种方法只是由 1 次遍历增加到了 2 次，时间复杂度的阶数上并未发生改变，这部分的时间复杂度仍然为 O（ncell）。（当然，重复遍历 2 遍使用了 2 个 map，也就是说增加了一个 map 的空间开销，但是这和方法 1 的空间开销相比起来是微不足道的）

另外一个要注意的点是 pair<int,int>中，两个数据是有顺序性的，但是在我们获得边的过程中，<n1,n2><n2,n1>实际上是表示一条边。所以，在向 map 中插入一条边时，要同时插入<n1,n2>和<n2,n1>，避免出现类似于，<n1,n2>在 map 中存在，但是查询<n2,n1>显示 map 中没有，就认为这条边是第一次出现的情况。

获得 edge、ecell 的代码如下：

```
1. //获得 edge、ecell
2.  map<pair<int, int>, int> m1,m2;//存储<边的两个 point，边号>
3.  int en = 0;//边的编号
4. //第 1 次循环：
5.  for (int i = 0;i < ncell;i++) {//遍历每个 cell
6.   for (int j = 0;j < 4;j++) {
7.    int n1, n2;//获取边的 2 个 point
8.    n1 = cells[4*i+j];
9.    n2 = cells[4*i+((j+1) % 4)];
10.   pair<int, int> t1(n1, n2),t2(n2,n1);
11.   if (m1.find(t1) == m1.end() && m1.find(t2) == m1.end())
   {//该边是第一次出现
12.    m1[t1] = en;
13.    m1[t2] = en;
14.    en++;
15.   }
16.  }
17. }
18.
19. nedge = en;
20. edge = (int*)malloc(2 * nedge * sizeof(int));
21. ecell = (int*)malloc(2 * nedge * sizeof(int));
22. for (int i = 0;i < 2 * nedge;i++) {
23.  ecell[i] = -1;
24. }
25.//第 2 次循环：
26. en = 0;
27. for (int i = 0;i < ncell;i++) {//遍历每个 cell
28.  for (int j = 0;j < 4;j++) {
29.   int n1, n2;//获取边的 2 个 point
30.   n1 = cells[4*i+j];
31.   n2 = cells[4*i+((j + 1) % 4)];
32.   pair<int, int> t1(n1, n2),t2(n2,n1);
33.   if (m2.find(t1) != m2.end() || m2.find(t2) != m2.end())
   {//该边已出现过，map 中已存入该边
34.    int e = m2[t1];//获得该边的编号
35.    ecell[2 * e + 1] = i;
36.   }
```

```
37.    else {//该边第一次出现
38.     m2[t1] = en;
39.     m2[t2] = en;
40.     edge[2 * en] = n1;
41.     edge[2 * en + 1] = n2;
42.     ecell[2 * en] = i;
43.     en++;
44.    }
45.   }
46. }
```

下面，由 cells 数组获得边界信息（bedge、becell、bound）。
实际上，只要找到哪条边是边界边，获得它的编号即可，bedge 和 becell 中所需的数据由边界边的编号都能在 edge 和 ecell 中获得。

那么如何找到哪条边是边界边？
由 ecell 便可得知。因为边界边只和一个 cell 关联，所以我们只要找出哪个 ecell 的一行中只存了一个 cell 即可。我们对 ecell 进行初始化，将其初始化为-1：

```
1.  for (int i = 0;i < 2 * nedge;i++) {
2.   ecell[i] = -1;
3.  }
```

这样，通过判断是不是-1 即可判断该位置有没有存入 cell。由于我们存 ecell 的时候，是先存在 ecell 的每一行的第 1 个位置，再存第 2 个位置，那么，若 ecell 的第 i 行的第 2 个位置为-1（即 ecell[2*i+1]==-1），则表示编号为 i 的边为边界边。将编号为 i 的边的 edge 信息和 ecell 信息赋值给 bedge、becell 即可。

同样的，存在我们不知道 nbedge 的问题，所以无法为 bedge 和 becell 分配内存。这里我们也是选择遍历 2 遍来解决这个问题。

获得 bedge、becell 的代码如下：

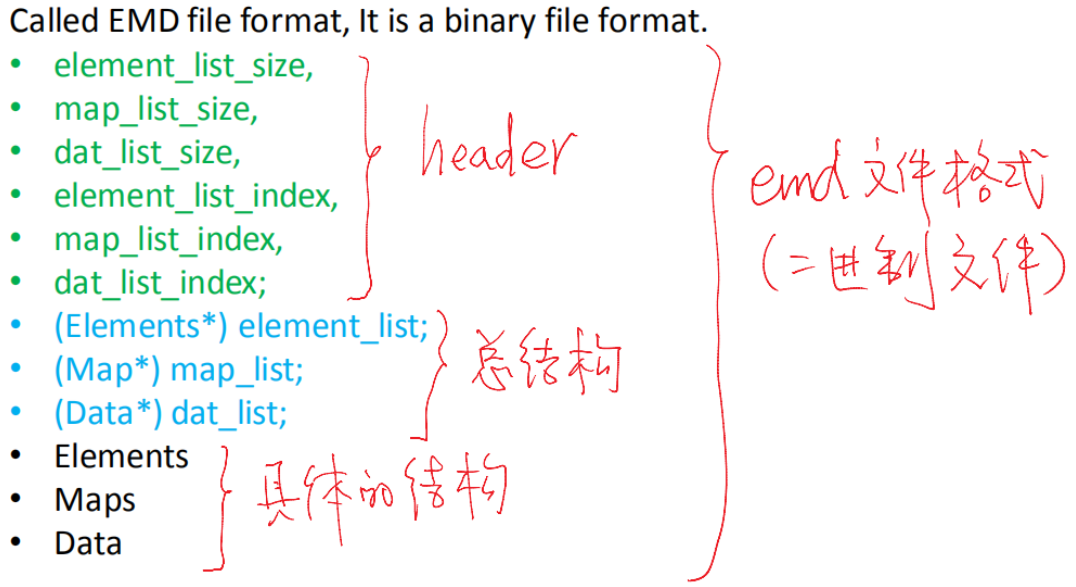
```
1.  //获得bedge、becell
2.  //通过：若是边界边，那么对应的ecell只有一个，该边号下第二个
   ecell为初始值-1，来判断出哪个边号是边界边
3.  int ben = 0;
4. //第1次循环：
5.  for (int i = 0;i < nedge;i++) {
6.   if (ecell[2 * i + 1] == -1) ben++;
7.  }
8.  nbedge = ben;
9.  bedge = (int*)malloc(2 * nbedge * sizeof(int));
10. becell = (int*)malloc(nbedge * sizeof(int));
11.//第2次循环：
12. int k = 0;
13. for (int i = 0;i < nedge;i++) {
```

```
14.   if (ecell[2 * i + 1] == -1) {
15.     bedge[2 * k] = edge[2 * i];
16.     bedge[2 * k+1] = edge[2 * i+1];
17.     becell[k] = ecell[2 * i];
18.     k++;
19.   }
20. }
```

关于 bound 的获取，思考了很久也没有想出获得的办法。个人觉得 bound 应该是事先给出来的，无法通过 vtk 中已知的数据获得。故这里，不存储 bound。

4.emd 格式文件的读写

emd 格式文件中直接存储的就是 elements maps datas，所以我们直接进行读取，就可构建 mesh。写的时候也是，直接将 mesh 的内容按下面的格式写。



使用 fread 函数和 fwrtie 函数实现对二进制文件的读写。

自由格式和 vtk 格式都是文本文件，而 emd 文件为二进制文件，要用不同的函数来进行读写。

fread 的函数原型如下：

size_t fread(void*buffer,size_t size,size_t count,FILE*stream);

1.buffer:　是读取的数据存放的内存的指针，

（可以是数组，也可以是新开辟的空间）

2.size:　是每次读取的字节数

3.count:　是读取的次数

4.stream:　是要读取的文件的指针

fwrite 的函数原型如下：

size_t fwrite(void*buffer,size_ size,size_t count,FILE*stream)

1.buffer：是一个指向用于保存数据的内存位置的指针

(是一个指针，对于 fwrite 来说，是要获取数据的地址)
2.size:　是每次读取的字节数
3.count:　是读取的次数
4.stream:　是数据写入的流（目标指针的文件）

由于读二进制文件需要事先知道变量的大小，而 name 的长度是不固定的，所以需要找出能获得 name 大小的方法。
这里我思考了 2 种方法:
1.为 element、dat、map 设置新的变量 name_size 来指明 name 的大小，这样，先读 name_size，就能知道 name 的大小，就可以读取 name 了。
2.将 name 的类型从 char const *改为确定大小的 char 数组：`char name[16]`，这样 name 的大小即为 sizeof(char)*16，即可轻松读取。
对于上述两种方法，个人认为将 name 设置为确定大小的 char 数组比较好。从内存方面考虑，设置一个大小固定的 char 数组，会浪费一小部分空间，但是额外存储 name_size 也需要一部分开销，这方面没有很大差别。从时间方面考虑，设置为固定大小数组方便了读写，无需别的操作，快捷简便，而设置 name_size 还需要读写此变量的操作。从功能方面考虑，name_size 变量的意义除了为了读写，对于之后可能对 mesh 的操作，没有什么作用，显得冗余（若需要，可以由 strlen(name)获得），破坏了对 mesh 类的成员变量的设计的精简性。所以这里采取的是设置 name 为大小为 16 的 char 数组。对于 dat 的 type 变量，也是相似的操作，设置为大小为 16 的 char 数组。

此外，需要注意**深浅拷贝**的问题。
一是，在 make 函数中，为 name 赋值时，需要使用 strcpy，简单的让两个 char*相等属于浅拷贝，函数结束后，局部变量释放，很可能指向的地址已被释放。
二是，写入文件时，需要将数据集中的数据写入，不能仅写入地址，因为地址很可能随后会被释放或者变换，找不到数据。

5.init 函数、构造函数、析构函数

Init 函数的作用是初始化 mesh 对象，需要做到，释放之前使用的内存，开辟新的内存和赋予变量初值。
需要注意的是要 free 干净，从小到大 free，需要 free：map 中的 map 指针、dat 中的 data 指针->各个 list 中的各个元素（指针）->各个 list。
Init 函数在 readfromfile 函数中调用。
Init 函数的代码如下:

```
1.    void Mesh::init() {
2.   //清除上一个
3.   for (int i = 0;i < element_list_index;i++) {
4.    free(element_list[i]);
5.   }
6.   for (int i = 0;i < map_list_index;i++) {
7.    free(map_list[i]->map);
8.    free(map_list[i]);
9.   }
```

```
10. for (int i = 0;i < dat_list_index;i++) {
11.  free(dat_list[i]->data);
12.  free(dat_list[i]);
13. }
14. free(element_list);
15. free(map_list);
16. free(dat_list);
17. //创建新的
18. element_list_index = 0;
19. map_list_index = 0;
20. dat_list_index = 0;
21. element_list_size = 10;
22. map_list_size = 10;
23. dat_list_size = 10;
24. element_list = (Elements*)malloc(sizeof(Elements) * elemen
    t_list_size);
25. map_list = (Map*)malloc(sizeof(Map) * map_list_size);
26. dat_list = (Data*)malloc(sizeof(Data) * dat_list_size);
27. memset(map_list, 0, sizeof(Map) * map_list_size);
28. memset(element_list, 0, sizeof(Elements) * element_list_si
    ze);
29. memset(dat_list, 0, sizeof(Data) * dat_list_size);
30.}
```

mesh 的构造函数即 init 函数的“创建新的”的部分，析构函数即 init 函数的“清除上一个”部分。若构造函数输入了文件名作为参数，则需最后调用 readfromfile 函数，读取文件信息，构建 mesh：

```
1. Mesh::Mesh() {
2.  element_list_index = 0;
3.  map_list_index = 0;
4.  dat_list_index = 0;
5.  element_list_size = 10;
6.  map_list_size = 10;
7.  dat_list_size = 10;
8.  element_list = (Elements*)malloc(sizeof(Elements) * elemen
    t_list_size);
9.  map_list = (Map*)malloc(sizeof(Map) * map_list_size);
10. dat_list = (Data*)malloc(sizeof(Data) * dat_list_size);
11. memset(map_list, 0, sizeof(Map) * map_list_size);
12. memset(element_list, 0, sizeof(Elements) * element_list_si
    ze);
13. memset(dat_list, 0, sizeof(Data) * dat_list_size);
14.}
15.
```

```
16.Mesh::Mesh (char* fname) {
17. element_list_index = 0;
18. map_list_index = 0;
19. dat_list_index = 0;
20. element_list_size = 10;
21. map_list_size = 10;
22. dat_list_size = 10;
23. element_list = (Elements*)malloc(sizeof(Elements) * elemen
    t_list_size);
24. map_list = (Map*)malloc(sizeof(Map) * map_list_size);
25. dat_list = (Data*)malloc(sizeof(Data) * dat_list_size);
26. memset(map_list, 0, sizeof(Map) * map_list_size);
27. memset(element_list, 0, sizeof(Elements) * element_list_si
    ze);
28. memset(dat_list, 0, sizeof(Data) * dat_list_size);
29. readfromfile(fname);
30.}
31.
32.Mesh::~Mesh() {
33. for (int i = 0;i < element_list_index;i++) {
34.  free(element_list[i]);
35. }
36. for (int i = 0;i < map_list_index;i++) {
37.  free(map_list[i]->map);
38.  free(map_list[i]);
39. }
40. for (int i = 0;i < dat_list_index;i++) {
41.  free(dat_list[i]->data);
42.  free(dat_list[i]);
43. }
44. free(element_list);
45. free(map_list);
46. free(dat_list);
47.}
```

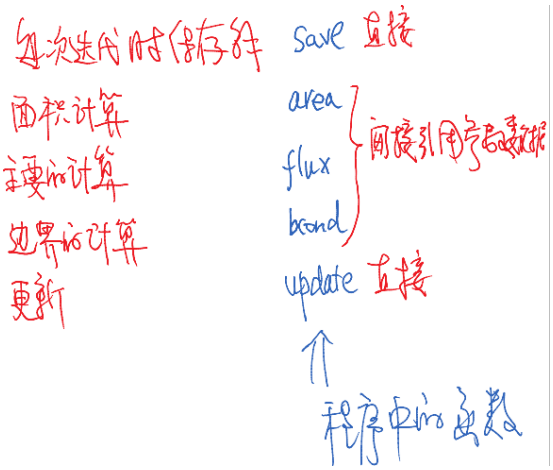
program 类

我们的操作对象是 mesh，故 Program 有 mesh 类型成员变量。这里注意，**由于涉及到 mesh 的析构问题，所以这里应为 mesh 的指针或者引用来作为成员变量**，所以 program 的构造函数传入的参数为 mesh 的引用：

```
Program(Mesh& ms);
```

核函数及其主要作用如下：

- save(save_soln)
- area(adt_calc)
- flux(res_calc)
- bcond(bres_calc)
- update



save 函数用于保存 cell 上的 4 个数据的旧值。
area 函数用于计算区域的大小。
flux 函数中对边进行计算。
bcond 函数中对边界进行计算。
update 函数用于更新 cell 上的数据。

下图为各个 element map data 的下标以及对应关系：

```
s.makeElements(nnode, "nodes");    //0
s.makeElements(nedge, "edges");    //1
s.makeElements(nbedge, "bedges"); //2
s.makeElements(ncell, "cells");    //3

s.makeMap(s.element_list[1], s.element_list[0], 2, edge , "pedge");     0
s.makeMap(s.element_list[1], s.element_list[3], 2, ecell, "pecell");    1
s.makeMap(s.element_list[2], s.element_list[0], 2, bedge, "pbedge");    2
s.makeMap(s.element_list[2], s.element_list[3], 1, becell, "pbecell"); 3
s.makeMap(s.element_list[3], s.element_list[0], 4, cell, "pcell");      4

s.makeData(s.element_list[2], 1, "int", (char*)bound, "p_bound");  0
s.makeData(s.element_list[0], 3, "double", (char*)x, "p_x");       1
s.makeData(s.element_list[3], 4, "double", (char*)q, "p_q");       2
s.makeData(s.element_list[3], 4, "double", (char*)qold, "p_qold"); 3
s.makeData(s.element_list[3], 1, "double", (char*)adt, "p_adt");   4
s.makeData(s.element_list[3], 4, "double", (char*)res, "p_res");   5
```

注意这里我们的顺序代表了其存储的顺序！

save：

save 函数是对单个 cell 的操作，其传入参数为 s.dat_list[2]->data 和 s.dat_list[3]->data，这两个 data 集的数据维度是 4，即一个 cell 对应 4 个数据，可以看作一行为一个 cell，有 4 个数据。则使用 (double*)(s.dat_list[2]->data) + 4 * i 和 (double*)(s.dat_list[3]->data) + 4 * i 来表示每个 cell 对应数据行的首地址。

这里要注意，由于 **dat 数据结构中，data 定义为 char*类型**，但是其中存储的不一定是 char 类型数据，本次实验中，所有的 data 中实际存储的都是 double 类型的数据，所以要对其**进行强制类型转换**。并且，是先将 data 的首地址转换为 double*类型，再进行加操作，以定位到所需数据行。

run 函数的迭代过程中，对 save 函数的迭代：

```
1.   for (int i = 0; i <n ; i++) {
2.    save((double*)(s.dat_list[2]->data) + 4 * i, (double*)(s
   .dat_list[3]->data) + 4 * i);
3.   }
```

area：

area 函数是对单个 cell 的操作。其中有对于 nodes 的操作，即对 cell 的 4 个 node 的操作，所以我们需要通过 `s.map_list[4]->map`，获得 cell 和 nodes 的 map。

由于一个 cell 对应 4 个 nodes，所以可以把这个 map 看作 4 个数据一行，表示一个 cell，所以对于下标为 i 的 cell 来说，其 4 个 nodes 的下标为：

```
s.map_list[4]->map[4*i]
s.map_list[4]->map[4 * i+1]
s.map_list[4]->map[4 * i+2]
s.map_list[4]->map[4 * i+3]
```

`s.dat_list[1]->data` 为 node 上的 xyz 数据，一个 node 对应 3 个数据，所以对于下标为 i 的 node，其数据行的首地址为`(double*)(s.dat_list[1]->data)+3*i`，这里是将 cell 对应的 4 个 nodes 的 xyz 坐标传入函数。

而 `s.dat_list[2]->data` 和 `s.dat_list[4]->data` 为 cell 上的数据，根据维度进行传入地址的处理即可，与 save 中操作类似。

run 函数的迭代过程中，对 area 函数的迭代：

```
1.   for (int k = 0; k < 2; k++) {
2.    for (int i = 0; i < n; i++) { //////////////
3.     area((double*)(s.dat_list[1]->data)+3*s.map_list[4]->ma
   p[4*i], (double*)(s.dat_list[1]->data) +3* s.map_list[4]->m
   ap[4 * i+1], (double*)(s.dat_list[1]->data) + 3 * s.map_lis
   t[4]->map[4 * i+2],
4.      (double*)(s.dat_list[1]->data )+ 3 * s.map_list[4]->ma
   p[4 * i+3], (double*)(s.dat_list[2]->data)+4*i, (double*)(s
   .dat_list[4]->data)+i);
5.    }
```

flux：

flux 函数是对单个 edge 的操作。仿照 area 中的操作，通过 map 获得 edge 和 nodes、cells 的关系，以获得 nodes 和 cells 的下标，再进行相应的加操作，获得对应数据行的首地址。逻辑和操作与 area 函数的参数传入类似，这里不再赘述。

run 函数的迭代过程中，对 flux 函数的迭代：

```
1.    for (int i = 0; i < m; i++) { //////////////
```

```
2.     flux((double*)(s.dat_list[1]->data)+3*s.map_list[0]->ma
   p[2*i], (double*)(s.dat_list[1]->data) + 3 * s.map_list[0]-
   >map[2 * i+1], (double*)(s.dat_list[2]->data )+ 4* s.map_li
   st[1]->map[2 * i ],
3.      (double*)(s.dat_list[2]->data) + 4 * s.map_list[1]->ma
   p[2 * i+1], (double*)(s.dat_list[4]->data)+ s.map_list[1]->
   map[2 * i], (double*)(s.dat_list[4]->data) + s.map_list[1]-
   >map[2 * i + 1],
4.      (double*)(s.dat_list[5]->data) + 4 * s.map_list[1]->ma
   p[2 * i ], (double*)(s.dat_list[5]->data )+ 4 * s.map_list[
   1]->map[2 * i + 1]);
5.    }
```

bcond：

bcond 函数是对单个 bedge 的操作。仿照 area 中的操作，通过 map 获得 edge 和 nodes、cells 的关系，以获得 nodes 和 cells 的下标，再进行相应的加操作，获得对应数据行的首地址。逻辑和操作与 area 函数的参数传入类似，这里不再赘述。

run 函数的迭代过程中，对 bcond 函数的迭代：

```
1.       for (int i = 0; i < bm; i++) {  ////////////////
2.     bcond((double*)(s.dat_list[1]->data) + 3 * s.map_list[2
   ]->map[2 * i], (double*)(s.dat_list[1]->data) + 3 * s.map_l
   ist[2]->map[2 * i+1], (double*)(s.dat_list[2]->data )+ 4 *
   s.map_list[3]->map[i],
3.      (double*)(s.dat_list[4]->data )+ s.map_list[3]->map[i]
   , (double*)(s.dat_list[5]->data )+ 4*s.map_list[3]->map[i],
    (int*)(s.dat_list[0]->data)+i);
4.    }
```

update：

update 函数是对单个 cell 的操作。仿照 save 函数的参数传入即可。

run 函数的迭代过程中，对 update 函数的迭代：

```
1.    for (int i = 0; i < n; i++) {
2.     update((double*)(s.dat_list[3]->data )+ 4 * i, (double*
   )(s.dat_list[2]->data) + 4 * i, (double*)(s.dat_list[5]->da
   ta) + 4 * i,
3.      (double*)(s.dat_list[4]->data )+ i, &rms);
4.    }
```

ParallelProgram 类

我们的操作对象是 mesh，故 ParallelProgram 有 mesh 类型成员变量。ParallelProgram 进行的工作和 Program 一样，只不过是以并行的方式进行。

老师给出的初始方案是，对于 cell node edge bedge 直接按照下标均分给各个线程去处理，只将被分到的 cell node edge bedge 传送给线程。**这样会出现的问题是线程缺少足够的数据信息，因为这些元素之间的关联关系并不是可以被下标均分的**，例如，会出现一个 cell 的 node 不在线程被分到的 node 中，那么该线程没有这个 node 的数据，则会出错。
对于这个问题，我想到了 2 个解决方法，一是线程若运行过程中发现缺少了什么数据，便向主线程索求（主线程拥有所有的数据），二是让所有线程都持有所有的数据。对于第一种方案，将大大增加通信开销，而第二种方案，虽然增加了内存的开销，但是还可以省去最初的主线程分配数据给线程的通信开销，故这里选择第二种方案进行修改。

我们**为全部的线程分配 global mesh**。这里的 global mesh 指的是完整的 mesh 的意思，而非全局变量的意思。每个线程都拥有完整的 mesh，存储在 global mesh 的相关变量中。

```
1. //Global Mesh
2.  int g_nnode = s.element_list[0]->size;
3.  int g_ncell = s.element_list[3]->size;
4.  int g_nedge = s.element_list[1]->size;
5.  int g_nbedge = s.element_list[2]->size;
6.  int* g_becell = 0, * g_ecell = 0, * g_bound = 0, * g_bedge
    = 0, * g_edge = 0,
7.   * g_cell = 0;
8.  double* g_x = 0, * g_q = 0, * g_qold = 0, * g_adt = 0, * g
   _res = 0;
9.
10. //local mesh
11. int nnode, ncell, nedge, nbedge;
12. int* becell, * ecell, * bound, * bedge, * edge, * cell;
13. double* x, * q, * qold, * adt, * res;
14. double rms, g_rms;
15.
16. initflow();//
17.
18. g_cell = (int*)(s.map_list[4]->map); //一个cell 4 个点
19. g_edge = (int*)(s.map_list[0]->map); //一条边 2 个点
20. g_ecell = (int*)(s.map_list[1]->map); //一个ecell 2 个点
21. g_bedge = (int*)(s.map_list[2]->map); //一条b 边2 个点
22. g_becell = (int*)(s.map_list[3]->map); //一个becell 1 个
   cell
23.
24. g_x = (double*)(s.dat_list[1]->data);//一个点三个坐标
25. g_q = (double*)(s.dat_list[2]->data);
26. g_qold = (double*)(s.dat_list[3]->data);
27. g_res = (double*)(s.dat_list[5]->data);
```

```
g_adt = (double*)(s.dat_list[4]->data);
g_bound = (int*)(s.dat_list[0]->data);

for (int n = 0; n < g_ncell; n++) { //ncell
 for (int m = 0; m < 4; m++) {
  g_q[4 * n + m] = qinf[m];
  g_res[4 * n + m] = 0.0f;
 }
}
//为了最后线程0 聚合所有数据
qold = (double*)malloc(4 * g_ncell * sizeof(double));
res = (double*)malloc(4 * g_ncell * sizeof(double));
adt = (double*)malloc(g_ncell * sizeof(double));

MPI_Barrier(MPI_COMM_WORLD);
```

使用 ParallelProgram 的 mesh 变量 s 的各个 element map data 为 global 变量赋值。这里还为 qold 等变量分配了内存，这是为了最终在线程 0 聚合数据而使用的变量。最终，将每个线程负责的部分的数据汇集在线程 0 的 qold、res、adt 上，只汇集这 3 部分的数据是因为，在核函数的执行过程中，只有这 3 个数据发生了变换，故只汇集这 3 个部分的更新后的数据。

每个线程要在自己的完整的 mesh 上执行自己的任务部分的数据，那么我们需要获得其范围，这里定义 node_begin, node_end, cell_begin, cell_end, edge_begin, edge_end, bedge_begin, bedge_end，存储 node、cell、edge、bedge 操作的起始下标和结束下标。编写 compute_local_range 来计算 begin 和 end:

```
static void compute_local_range(int global_size, int mpi_comm_size,
 int mpi_rank, int& begin, int& end) {
 begin = 0;
 for (int i = 0; i < mpi_rank; i++) {
  begin += compute_local_size(global_size, mpi_comm_size, i);
 }
 end = begin + compute_local_size(global_size, mpi_comm_size, mpi_rank);
}
```

累加自己线程号之前的线程的 local_size 便可获得 begin，begin+本线程的 local_size=end。

在每个核函数操作后，都要进行同步，**使用 MPI_Barrier 来实现同步**:

```
for (int i = 0; i < ncell; i++) {
   area(
    g_x + (g_cell[4 * i]) * 3,
    g_x + (g_cell[4 * i + 1]) * 3,
    g_x + (g_cell[4 * i + 2]) * 3,
```

```
      g_x + (g_cell[4 * i + 3]) * 3,
      g_q + 4 * i,
      g_adt + i);
    }
  MPI_Barrier(MPI_COMM_WORLD);
```

这里的 MPI_Barrier 函数，是用于一个通信子中所有进程的同步，调用函数时进程将处于等待状态，直到通信子中所有进程都调用了该函数后才继续执行。在每个核函数的计算后都调用该函数，使得线程的计算阶段同步。

经过所有的计算后，**要将各线程的 rms 汇集到线程 0 的 g_rms 上，MPI_reduce 用于每个线程都发送相同数据量的数据给 root 线程，并进行归约操作**，这里是要求加和，于是为 MPI_SUM。

```
 MPI_Reduce(&rms,&g_rms,1,MPI_DOUBLE,MPI_SUM,0, MPI_COMM_WORLD);

 //聚合数据到线程0
 gather_double_array(qold, g_qold + 4 * cell_begin, comm_size, g_ncell, ncell, 4);
 gather_double_array(res, g_res + 4 * cell_begin, comm_size, g_ncell, ncell, 4);
 gather_double_array(adt, g_adt + cell_begin, comm_size, g_ncell, ncell, 1);

 if (my_rank == 0) {
  printf("ROOT: Total residual %10.5e \n", g_rms);
 }
```

各个线程将更新了的数据 g_qold、g_res、g_adt，汇集到线程 0 的 qold、res、adt 上。这里编写了函数 gather_double_array 来完成 double 类型数据的汇集：

```
static void gather_double_array(double* g_array, double* l_array,
 int comm_size, int g_size, int l_size,
 int elem_size) {
 int* sendcnts = (int*)malloc(comm_size * sizeof(int));
 int* displs = (int*)malloc(comm_size * sizeof(int));
 int disp = 0;

 for (int i = 0; i < comm_size; i++) {
  sendcnts[i] = elem_size * compute_local_size(g_size, comm_size, i);
 }
 for (int i = 0; i < comm_size; i++) {
  displs[i] = disp;
  disp = disp + sendcnts[i];
```

```
14. }
15. MPI_Gatherv(l_array, l_size*elem_size, MPI_DOUBLE, g_array
    , sendcnts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
16. free(sendcnts);
17. free(displs);
18.}
```

g_array 表示用于汇集的线程 0 的数组地址，l_array 表示线程的局部数据的起始地址。g_size 表示元素总数，l_size 表示线程局部元素数（注意，这两个 size 均描述的是 node\cell\edge\bedge 的大小），elem_size 表示数据维度。g_size*elem_size 为数据总量，l_size*elem_size 为线程局部数据量。例如，qold 是 cell 上的数据，则 g_size=g_ncell，l_size=ncell，由于一个 cell 上有 4 个 qold 数据，所以 elem_size=4。

由于各个线程负责的数据量可能不同，所以我们使用 MPI_Gatherv 来完成汇集工作，该函数可以从不同的线程接收不同数量的数据。MPI_Gatherv 的原型和参数说明如下：

int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs,MPI_Datatype recvtype, int root, MPI_Comm comm)

IN sendbuf 发送消息缓冲区的起始地址(可变)
IN sendcount 发送消息缓冲区中的数据个数(整型)
IN sendtype 发送消息缓冲区中的数据类型(句柄)
OUT recvbuf 接收消息缓冲区的起始地址(可变,仅对于根进程)
IN recvcounts 整型数组(长度为组的大小)，其值为从每个进程接收的数据个数(仅对于根进程)
IN displs 整数数组,每个入口 i 表示相对于 recvbuf 的位移,
此位移处存放着从进程 i 中接收的输入数据(仅对于根进程)
IN recvtype 接收消息缓冲区中数据类型(仅对于根进程)(句柄)
IN root 接收进程的序列号(句柄)
IN comm 通信子(句柄)

需要我们构建 recvcounts 和 displs 两个数组。由于接收每个线程的数据个数=每个线程发送的数据个数，所以我们直接定义 sendcnts 数组来获得每个线程发送的数据数。
通过 elem_size * compute_local_size(g_size, comm_size, i)获取线程 i 的 sendcnt。
displs 数组存放每个线程发送的数据在 recvbuf 中的位移，通过不断累加 sendcnt 即可获得每个线程的 displ。

DistributedProgram 类

我们的操作对象是 mesh，故 DistributedProgram 有 mesh 类型成员变量。DistributedProgram 进行的工作和 ParallelProgram 一样，但是这里是通过划分的方式来实现并行，将数据划分到每个线程上去执行。

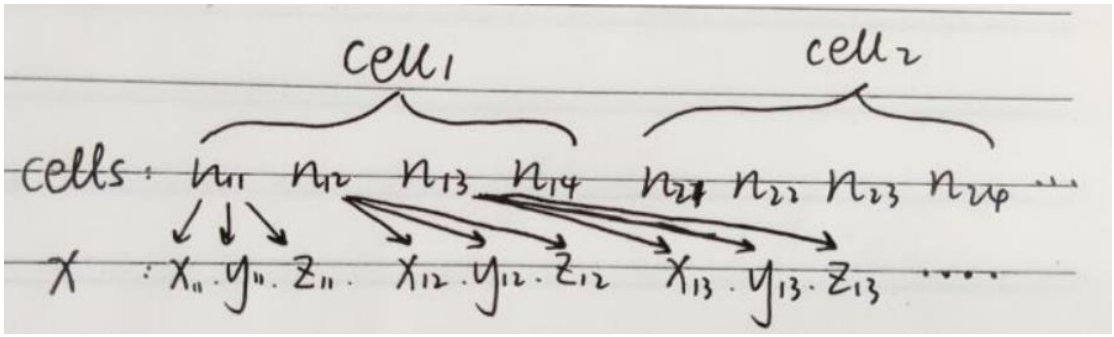
对于 mesh 该如何划分，这里考虑按照 cell 来划分，将 cell 均分给每个线程。
可以看作有以下 2 种情况：
1.若每个线程持有完整的 mesh，那么同 ParallelProgram 的做法相似，线程持有全部数据但

是仅执行自己的部分的任务。
2.若每个线程只持有部分 mesh，考虑一开始仅让线程持有自己部分的 cell 数组。

1.方法和 ParallelProgram 类似，故不再做重复工作。**下面我们实现 2.方法。**

首先一个问题，对于别的数据该如何处理呢？这里我想了两种解决方法:
(1)当后面执行核函数时，需要什么再让主线程发送过来数据。save 和 area 函数都是基于 cell 进行的，更新的是 cell 上的数据，和我们的划分很吻合，无需多做处理。flux 和 bcond 函数是基于 edge、bedge 进行的，可以看到，更新 res 的时候，采用的是加减的方式更新。我们知道，cell 之间肯定是有重叠的边，若不做处理，那么大多数边会被进行多次处理，而 res 将进行重复的加减，会导致结果不对（若对 res 的更新是赋值操作，则不会有问题）。所以我们这里可以考虑设置 2 个标识性数组，来标识每个 edge、bedge 是否被操作过了。由于我们这里是多线程操作，所以每个线程每次操作后、对标识性数组进行更新后，都要通过发送接收消息来达到同步。这里可以考虑专门另设一个线程，专门进行同步整合的工作，即由它来接收所有线程的消息，然后整合，然后再发送给所有的线程。
(2)将别的数据按照 cell 的顺序重新编排，这样便可以进行划分。

考虑到（1）方法执行时需要很大的通信开销，所以我们选择（2）来进行实现。我们通过编写 PartitionTo 和 PartitionFrom 2 个函数来实现划分。
向各个线程均分 cells，并将 x 数组（即 nodes 的 xyz）按照 cells 的顺序排列，示意图如下:

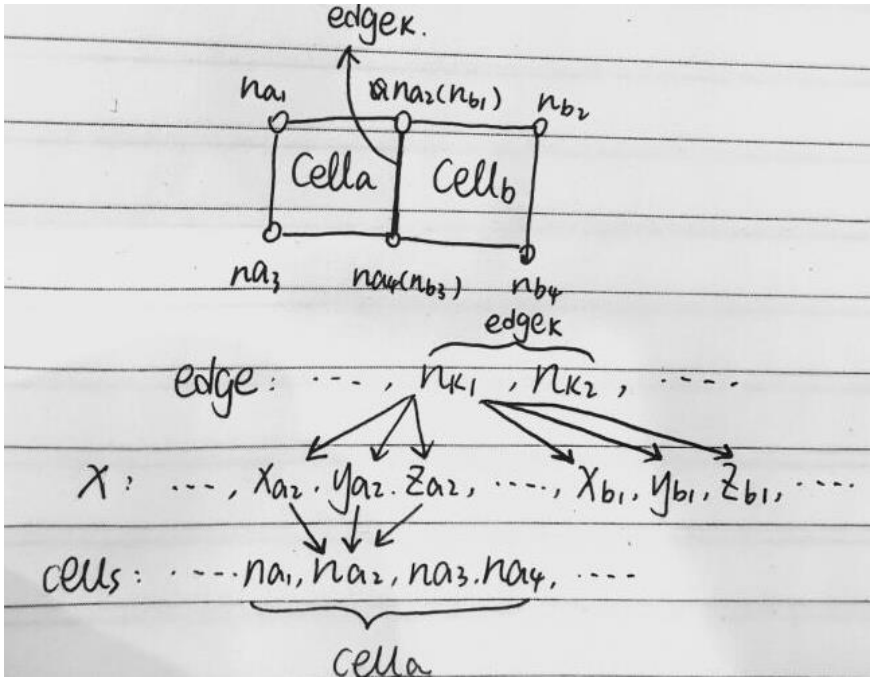


cells 中，存储的是每个 cell 对应的 nodes 编号，即按顺序存储着 cell1 的 4 个结点 n11 n12 n13 n14、cell2 的 4 个结点 n21 n22 n23 n24...
将 x 按照 cells 排列，则按照 n11 n12 n13 n14，n21 n22 n23 n24...结点的顺序，将其 xyz 存储在 x 中。
这便是 PartitonTo(cell,pcell,xyz)的任务。

在这样重排了 x 的顺序后，我们需要调整 edge 和 bedge。因为 edge 和 bedge 中存储的实际上是 edge/bedge 到 nodes 的 map，我们取 edge/bedge 的相关的两个结点的数据的时候，是通过这两个 map 中获得结点的编号（index）。现在结点进行了重排，原来的 index 定位到的已经不再是原来的点，所以我们需要对 edge bedge 做出调整。
这里的调整指的便是获得结点的新编号。这便是 PartitonFrom(edge,pedge,xyz) 和 PartitonFrom(bedge,pbedge,xyz)的任务。

为什么仅做上面的调整就可以呢？我们可以来看看我们需要执行的 4 个核函数。
save 函数，涉及到 q qold，这两个数据是 cell 上的数据，本身就是按照 cell 的编号来排序的，所以无需进行调整，均分即为按照 cell 划分。

area 函数，涉及到 x q adt，这是一个在 cell 上进行的函数，x 已按照 cell 划分调整，adt 也是按照 cell 分布的，无需特别调整。
flux 函数，这是一个在 edge 上进行的函数，涉及到 edge 的 node cell 上的数据，我们已对 edge 的 node 做出了调整，由于 node 已按 cell 划分了，所以 edge 关联的 2 个 cell 也在本划分块中。因为 edge 的 2 个 node 属于那 2 个 cell，所以 2 个 node 的出现代表 cell 也被划分在本线程中。这里可能不太好理解，我们画图说明下：



edge 的 node 在 x 中可能会对应多个 node 的 xyz（因为这里设计的 x 实际是冗余的，cell 之间有共同的点，而 x 在每个 cell 中都将该点 xyz 存入，就重复存了），这些 node 的 xyz 都对应到 cell 中的 node，故该 cell 也被划分入。

bcond 函数，这是一个在 bedge 上进行的函数，涉及到 bedge 的 node cell 上的数据，分析同 flux 函数，这里不再赘述。

update 函数，这是一个在 cell 上进行的函数，涉及到的 q qold res adt 都是按照 cell 分布的，所以无需特别调整。

PartitionTo 的实现

将 node 按 cell 划分，这部分的实现有 2 种选择：
1.按 cell 划分后的 node 不冗余。将 node 按 cell 中 node 出现的顺序新编号，若该出现之前已经新编号过了的 node，则直接将 cell 中 node 的编号换成前面编排的新编号。
2.按 cell 划分后的 node 冗余。对于每个 cell 中的 node，不管前面是否已经出现过，都按照 cell 的顺序给他新编号。

这里我选择第 2 种方法实现，因为第 1 种方法中，没办法将 node 划分到每个线程，而第二种方法可以实现将 node 划分到每个线程。
对于 cell 之间重复的点，我选择每个 cell 对应的 x 的部分，都存入重复点的 xyz。所以原来

分配的 x 的空间是不够的(仅够存 g_nnode 个 node 的 xyz)，这里设计一个数组 partiton_x，来存储按 cell 划分后的 x。

patition_x 的内存申请：
partition_x = (double*)malloc(3 * 4 * g_ncell * sizeof(double));
（有 g_ncell 个 cell，每个 cell 对应 4 个 nodes，每个 node 对应 3 个数据 xyz）

接下来实现 partitonTo 函数，在该函数中，我们需要做的是，遍历 cells 中的每个 node，在原来的 x 中找到该 node 的 xyz，放入 partiton_x 中，并更改 cells 中 node 编号为其在 partition_x 中的新编号：

```
1. int DistributedProgram::partitionTo(int* cells, int size, d
   ouble* p_x, double* partition)
2. {
3.  int index = 0;//partition 的下标
4.  for (int i = 0;i < size;i++) {//遍历cells
5.   int* cell = cells + 4 * i;
6.   for (int j = 0;j < 4;j++) {//遍历每个cell 的4 个点
7.    int node = *(cell + j);//node 编号
8.    *(cell + j) = index; //更改为新编号
9.    for (int k = 0;k < 3;k++) {//遍历每个点的xyz
10.    double data = *(p_x + 3 * node + k);
11.    partition[3*index+k] = data;
12.   }
13.   index++;
14.  }
15. }
16. return 0;
17.}
```

对该函数的调用：

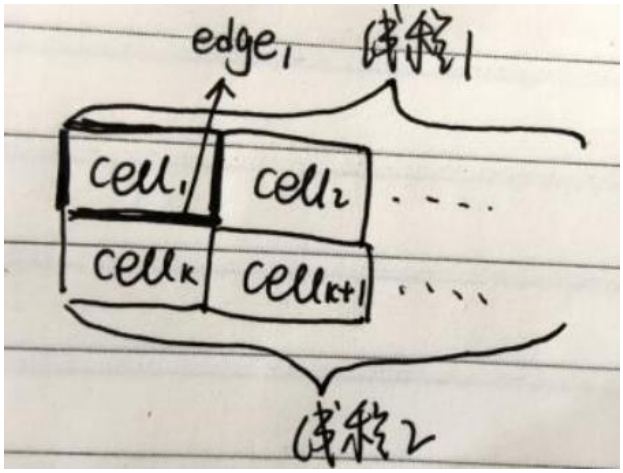
```
1. partitionTo(g_cell, g_ncell, g_x, partition_x);//将x
   （nodes）按照cells 排列
```

PartitionFrom 的实现

PartitionFrom 的任务是将 edge bedge 中 node 编号更换成新编号。
按照上述思路中，我们可以发现，这种思路下，**edge 中的 node 会对应很多个 cell 中的 node**，也就是说，edge 中每个 node 的新编号，是好几个编号的集合（至多是 4 个编号的集合）。
这里有一种实现思路，是创建一个 partiton_edge，其大小为 2 * g_nedge * 4* sizeof(int)，另外，我们还需要一个数组 index_num 记录每个 node 拥有多少个新编号。这里 partiton_edge 的结构可以看作是一个 2g_nedge*4 的矩阵，每行代表一个 node，每行有 4 列表示至多有 4 个编号。

这时，我们会遇到一个问题，那就是，如果均分划分 edge，那么涉及到的 nodes 和 cells 可能存在没有被划分到本线程的。

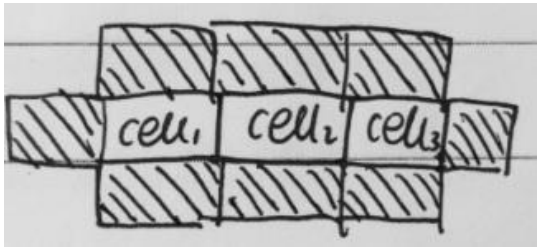
容易想到，那我们也将 edge 按照 cell 划分，允许冗余，将每个 cell 对应的 4 个 edge 按 cell 顺序存储。可是这样仍然存在问题，那就是，ecell 该如何处理。ecell 是和 edge 关联的 cell 的编号，**可能 edge 的另一个 ecell 不被划分到本线程**，如下图：



edge1 属于 cell1，被划分入线程 1 中，edge1 拥有 2 个 ecell：cell1 和 cellk，但是 cellk 属于线程 2，线程 1 不拥有 cellk 的数据。

对于这种情况，一种解决方法是，冗余的拥有 ecell 的数据。即，通过 cell 划分 edge 后，将每个 edge 的 2 个 ecell 按划分后的 edge 的顺序存储起来，再划分给每个线程。这样，保证了每个线程拥有足够的信息。

实际上，上述做法，就是让一个线程拥有周边所有与其 cells 相邻的 cells 的数据。一个简单的示意图如下：



一个线程拥有连续的 3 个 cell：cell1 cell2 cell3，那么它可能需要额外拥有阴影部分的 cell 的信息。

这时线程拥有的 cells 连续时的情况，若 cell 的分布是不连续的，那么线程需要持有更多的额外的 cells。最坏的情况，一个 cell 连带需要携带 4 个 cell 的额外数据。

一个简单的分析，一个线程被划分到的 cell 数目为 g_ncell/comm_size，按照上面说的一个 cell 连带 4 个 cell 的最坏情况，此时线程需要携带的 cell 数目为 5*g_ncell/comm_size。那么若并发的线程数小于等于 5，那么一个线程持有的数据量将大于等于整个 mesh 的数据量。当然，我们考虑的是最坏的情况，而且这种最坏的情况不可能发生在每个 cell 上，但是从此也可以看出，上述的实现方法不但实现起来繁琐，其最终实现的空间开销还和直接让线程持有全部数据差不多，并不是明智之举。

也许上述过程中讨论到的问题有更优的解决方法，我暂时没能想到。

重新审视上面的问题，可以发现，需要有多个编号就是因为若只持有第一个对应上的新编号，该 node 可能没有被划分入本线程。需要冗余拥有 ecell，因为线程可能没有拥有需要的 ecell。

所以我们这里简单的让 partiton_x 和 g_ecell、g_becell 每个线程都持有，来解决上面的问题。

让每个线程都获得 partiton_x：

```
1.  partition_x = (double*)malloc(3 * 4 * g_ncell * sizeof(dou
    ble));//用于存储 partition 后的 x，按 cell 对应
2.  partitionTo(g_cell, g_ncell, g_x, partition_x);//将 x
    （nodes）按照 cells 排列
3.  if (my_rank == 0) {
4.   partitionFrom(g_edge, g_nedge, 4 * g_ncell,g_cell);//由于
    x（nodes）换了新的顺序，所以原来的 node 编号失效了，要将旧的 node
    编号更换成新的
5.   partitionFrom(g_bedge, g_nbedge, 4 * g_ncell,g_cell);//由
    于 x（nodes）换了新的顺序，所以原来的 node 编号失效了，要将旧的
    node 编号更换成新的
6.  }
```

线程 0 读取文件创建好所有数据后，广播 g_ecell、g_becell：

```
1.  MPI_Bcast(g_ecell, 2 * g_nedge, MPI_INT, 0, MPI_COMM_WORLD
    );
2.  MPI_Bcast(g_becell, g_nbedge, MPI_INT, 0, MPI_COMM_WORLD);
```

使用公共数据的时候，需要转换下标，转换到线程的局部下标。这个通过下标加上在此线程之前的全部线程的 local size 之和即可：

```
1. for (int i = 0; i < nedge; i++) { //////////////
2.    int ii = 0;
3.    for (int k = 0;k < my_rank;k++) {
4.     ii+= compute_local_size(g_nedge, comm_size, k);
5.    }
6.    flux(
7.     partition_x + (edge[2 * i]) * 3,
8.     partition_x + (edge[2 * i + 1]) * 3,
9.     q + (g_ecell[2 * (ii+i)]) * 4,
10.    q + (g_ecell[2 * (ii + i) + 1]) * 4,
11.    adt + (g_ecell[2 * (ii + i)]),
12.    adt + (g_ecell[2 * (ii + i) + 1]),
13.    res + (g_ecell[2 * (ii + i)]) * 4,
14.    res + (g_ecell[2 * (ii + i) + 1]) * 4);
15.  }
16.  MPI_Barrier(MPI_COMM_WORLD);
17.
18.  for (int i = 0; i < nbedge; i++) {  ////////////////
```

```
19.    int ii = 0;
20.    for (int k = 0;k < my_rank;k++) {
21.     ii += compute_local_size(g_nbedge, comm_size, k);
22.    }
23.    bcond(
24.     partition_x + (bedge[2 * i]) * 3,
25.     partition_x + (bedge[2 * i + 1]) * 3,
26.     q + (g_becell[ii + i]) * 4,
27.     adt + (g_becell[ii+i]),
28.     res + (g_becell[ii+i]) * 4,
29.     bound + i);
30. }
```

这样，edge 和 bedge 新编号的获得，只需取第一个对应上的 node 的新编号即可。partitonFrom 函数的实现如下：

```
1. int partitionFrom(int* cells,int size,int size1, int* p_x)
2. {
3.  int index = 0;//partition 的下标
4.  for (int i = 0;i < size;i++) {//遍历每个边
5.   int* edge = cells + 2 * i;
6.   for (int j = 0;j < 2;j++) {//遍历每个边的node
7.    int old_node = *(edge + j);//旧的node 编号
8.    //查找新的编号,并更新
9.    for (int k = 0;k < size1;k++) {
10.    if (p_x[k] == old_node) *(edge + j) = k;
11.   }
12.  }
13. }
14. return 0;
15.}
```

对该函数的调用：

```
1.   partitionFrom(g_edge, g_nedge, 4 * g_ncell,g_cell);//由于x（nodes）换了新的顺序，所以原来的node 编号失效了，要将旧的node 编号更换成新的
2.   partitionFrom(g_bedge, g_nbedge, 4 * g_ncell,g_cell);//由于x（nodes）换了新的顺序，所以原来的node 编号失效了，要将旧的node 编号更换成新的
3.
```

实验结果

编写测试文件 Test.cpp：

```cpp
#include "Mesh.h"
#include "Program.h"
#include "ParallelProgram.h"
#include "DistributedProgram.h"
#include<iostream>

int main() {
 //test vtk read
 Mesh s;
s.readvtk("t1.vtk");

//test emd write
s.savetofile("new_grid.emd");

//test emd read
Mesh s2((char*)"new_grid.emd");

//test raw read
Mesh s3;
s3.readraw("new_grid.dat");

//test program
Mesh s4;
Program prog(s4);
prog.run(0);

//test distributed program
DistributedProgram prog2(s3);
int argc = 3;
char* argv[] = { {(char *)"mpiexec"},{(char*)"-n"},{(char*)"4"} };
prog2.run(argc,argv);

//test parallel program
ParallelProgram prog3(s3);
prog3.run(argc, argv);

return 0;
}
```

测试各个格式的读写和串行 program：

```
start read:t1.vtk
writing in grid
start read:new_grid.emd
start read:new_grid.dat
initialising flow field
 10   1.17422e-03
 20   1.12349e-03
 30   6.16318e-04
 40   3.65603e-04
 50   2.33023e-04
 60   1.57289e-04
 70   1.14031e-04
 80   8.73393e-05
 90   6.79479e-05
 100   5.24394e-05
```

可见运行正确

测试并行 program:

```
Number of nodes, cells, edges, bedges on process 0 = 124, 113, 217, 18
start read:t1.vtk
writing in grid
start read:new_grid.emd
start read:new_grid.dat
initialising flow field
 10   1.17422e-03
 20   1.12349e-03
 30   6.16318e-04
 40   3.65603e-04
 50   2.33023e-04
 60   1.57289e-04
 70   1.14031e-04
 80   8.73393e-05
 90   6.79479e-05
 100   5.24394e-05
initialising flow field
Number of nodes, cells, edges, bedges on process 2 = 124, 112, 216, 17
start read:t1.vtk
writing in grid
start read:new_grid.emd
start read:new_grid.dat
initialising flow field
 10   1.17422e-03
 20   1.12349e-03
 30   6.16318e-04
 40   3.65603e-04
 50   2.33023e-04
 60   1.57289e-04
 70   1.14031e-04
 80   8.73393e-05
 90   6.79479e-05
 100   5.24394e-05
initialising flow field
Number of nodes, cells, edges, bedges on process 3 = 124, 112, 216, 17
start read:t1.vtk
writing in grid
start read:new_grid.emd
start read:new_grid.dat
initialising flow field
 10   1.17422e-03
 20   1.12349e-03
 30   6.16318e-04
 40   3.65603e-04
 50   2.33023e-04
```

可见运行正确

总结

通过课程报告，将本学期编写的实验进行了一个汇总，编写成了一个 mesh 项目，能完成不同格式的文件的读写和存储，使用串行和并行的方式进行计算。在汇总的过程当中，让我又对本学期的内容进行了回顾，对于一些以前遇到的问题现在又有了新的看法，对于一些解答方法也能做出改进，对整个项目的掌握也越来越熟练。