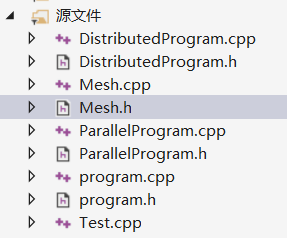
# 课程报告

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

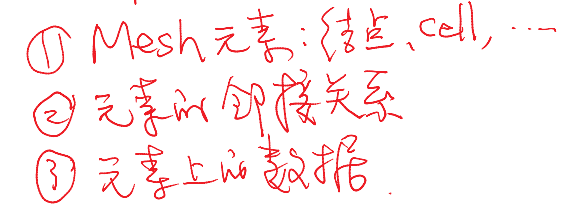
**整体框架**



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）。

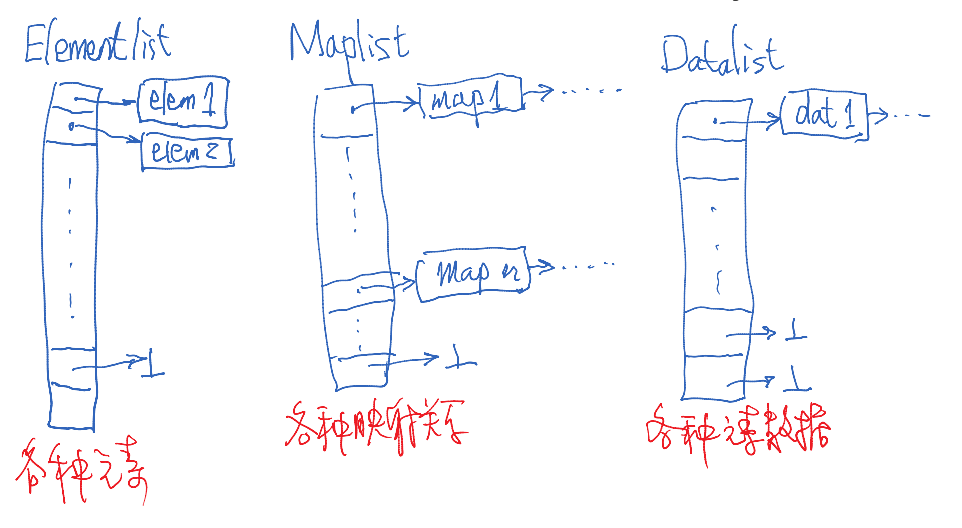
**Mesh类**

**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;

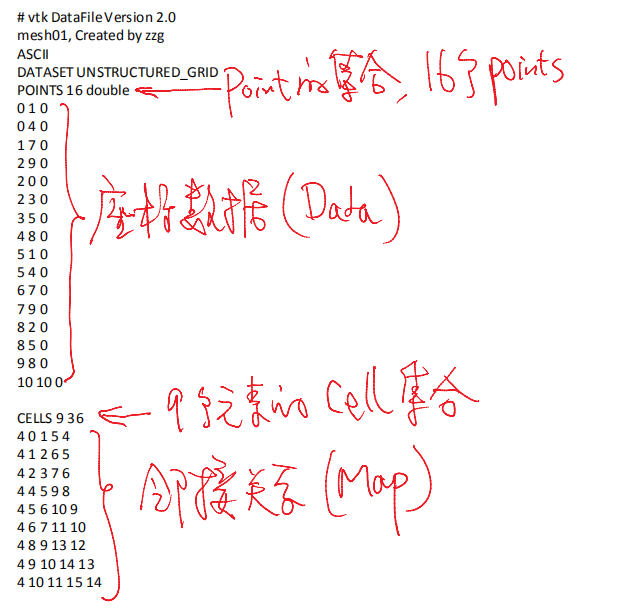
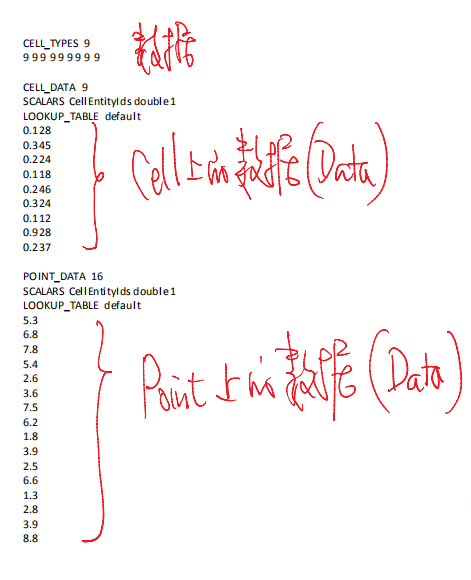
可以看到，map由 其在list中的下标index、映射关系中的两个element（这里指的是指针）、映射关系的维度、映射关系集合、map集的名称 组成

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 \* data->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->size \* 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：

1. double sum = 0;
2. if (dcell2->size == 4) {
3. int\* temp = (int\*)dcell2->data;
4. for (int i = 0; i < ncell; i++) {
5. sum += temp[i];
6. }
7. }
8. else if (dcell2->size == 8) {
9. double\* temp = (double\*)dcell2->data;
10. for (int i = 0; i < ncell; i++) {
11. sum += temp[i];
12. }
13. }

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。

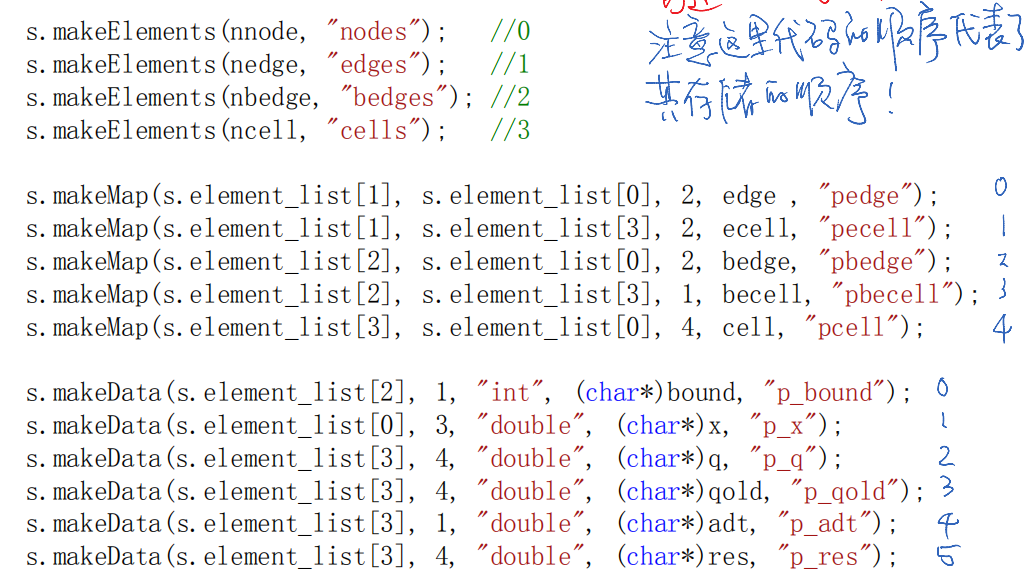
fprintf()函数根据指定的格式(format)，向输出流(stream)写入数据(argument)。

**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：



**3.读取vtk格式文件**

**read函数的具体规划：**

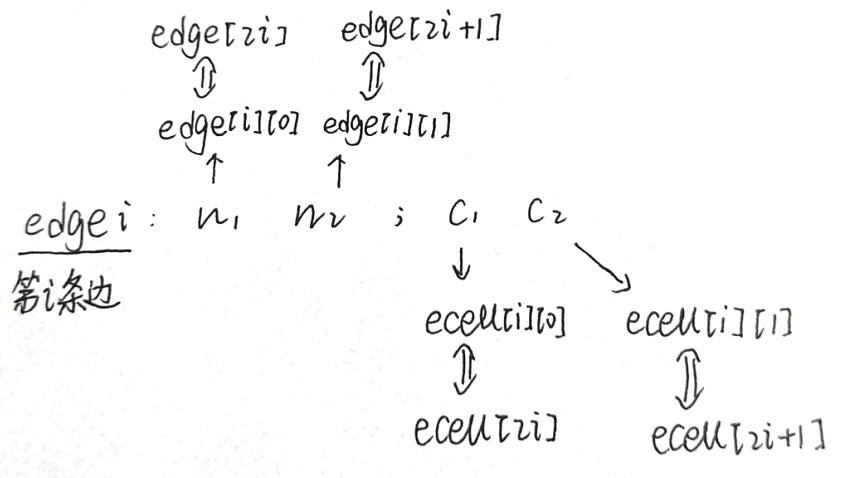
使用一个Mesh::readvtk函数进行总的read操作（外部调用此函数进行read），在该函数中，再调用read各个部分的函数，完成对各个部分的read。

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. nedge = en;
19. edge = (int\*)malloc(2 \* nedge \* sizeof(int));
20. ecell = (int\*)malloc(2 \* nedge \* sizeof(int));
21. for (int i = 0;i < 2 \* nedge;i++) {
22. ecell[i] = -1;
23. }
24. *//第2次循环：*
25. en = 0;
26. for (int i = 0;i < ncell;i++) {*//遍历每个cell*
27. for (int j = 0;j < 4;j++) {
28. int n1, n2;*//获取边的2个point*
29. n1 = cells[4\*i+j];
30. n2 = cells[4\*i+((j + 1) % 4)];
31. pair<int, int> t1(n1, n2),t2(n2,n1);
32. if (m2.find(t1) != m2.end() || m2.find(t2) != m2.end()) {*//该边已出现过，map中已存入该边*
33. int e = m2[t1];*//获得该边的编号*
34. ecell[2 \* e + 1] = i;
35. }
36. else {*//该边第一次出现*
37. m2[t1] = en;
38. m2[t2] = en;
39. edge[2 \* en] = n1;
40. edge[2 \* en + 1] = n2;
41. ecell[2 \* en] = i;
42. en++;
43. }
44. }
45. }

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。

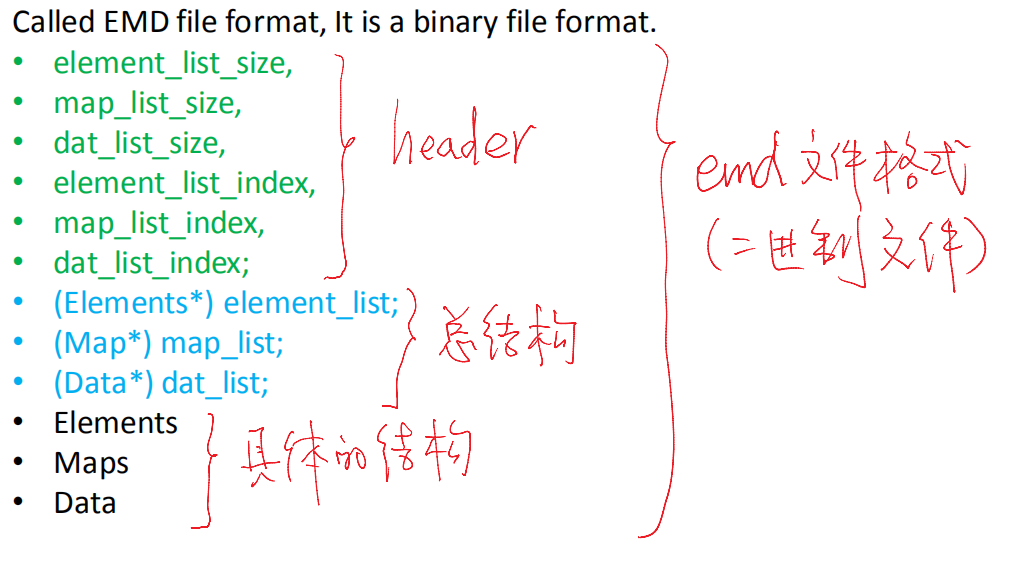
获得bedge、becell的代码如下：

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) \* element\_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\_size);
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：

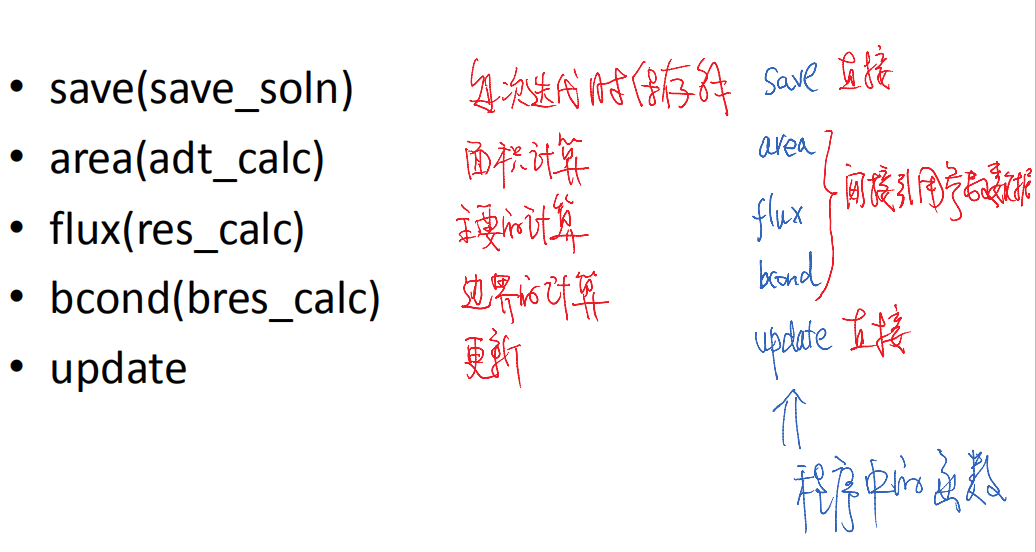
1. Mesh::Mesh() {
2. element\_list\_index = 0;
3. map\_list\_index = 0;
4. dat\_list\_index = 0;
5. element\_list\_size = 10;
6. map\_list\_size = 10;
7. dat\_list\_size = 10;
8. element\_list = (Elements\*)malloc(sizeof(Elements) \* element\_list\_size);
9. map\_list = (Map\*)malloc(sizeof(Map) \* map\_list\_size);
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\_size);
13. memset(dat\_list, 0, sizeof(Data) \* dat\_list\_size);
14. }
15. Mesh::Mesh (char\* fname) {
16. element\_list\_index = 0;
17. map\_list\_index = 0;
18. dat\_list\_index = 0;
19. element\_list\_size = 10;
20. map\_list\_size = 10;
21. dat\_list\_size = 10;
22. element\_list = (Elements\*)malloc(sizeof(Elements) \* element\_list\_size);
23. map\_list = (Map\*)malloc(sizeof(Map) \* map\_list\_size);
24. dat\_list = (Data\*)malloc(sizeof(Data) \* dat\_list\_size);
25. memset(map\_list, 0, sizeof(Map) \* map\_list\_size);
26. memset(element\_list, 0, sizeof(Elements) \* element\_list\_size);
27. memset(dat\_list, 0, sizeof(Data) \* dat\_list\_size);
28. readfromfile(fname);
29. }
30. Mesh::~Mesh() {
31. for (int i = 0;i < element\_list\_index;i++) {
32. free(element\_list[i]);
33. }
34. for (int i = 0;i < map\_list\_index;i++) {
35. free(map\_list[i]->map);
36. free(map\_list[i]);
37. }
38. for (int i = 0;i < dat\_list\_index;i++) {
39. free(dat\_list[i]->data);
40. free(dat\_list[i]);
41. }
42. free(element\_list);
43. free(map\_list);
44. free(dat\_list);
45. }

**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。

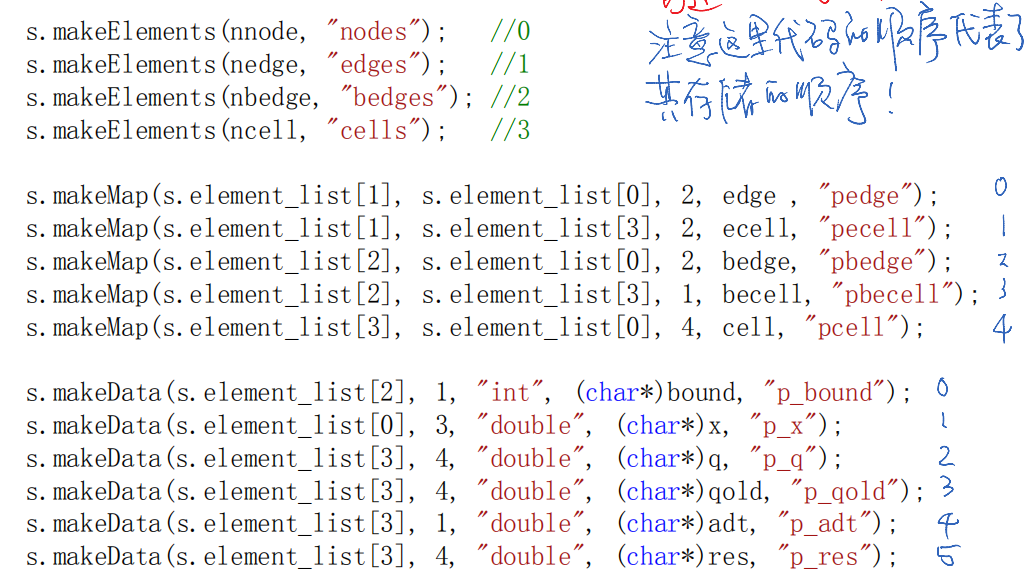
area函数用于计算区域的大小。

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：



**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]->map[4\*i], (double\*)(s.dat\_list[1]->data) +3\* s.map\_list[4]->map[4 \* i+1], (double\*)(s.dat\_list[1]->data) + 3 \* s.map\_list[4]->map[4 \* i+2],
4. (double\*)(s.dat\_list[1]->data )+ 3 \* s.map\_list[4]->map[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]->map[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\_list[1]->map[2 \* i ],
3. (double\*)(s.dat\_list[2]->data) + 4 \* s.map\_list[1]->map[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]->map[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\_list[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]->data) + 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. *//local mesh*
10. int nnode, ncell, nedge, nbedge;
11. int\* becell, \* ecell, \* bound, \* bedge, \* edge, \* cell;
12. double\* x, \* q, \* qold, \* adt, \* res;
13. double rms, g\_rms;
14. initflow();*//*
15. g\_cell = (int\*)(s.map\_list[4]->map); *//一个cell 4个点*
16. g\_edge = (int\*)(s.map\_list[0]->map); *//一条边 2个点*
17. g\_ecell = (int\*)(s.map\_list[1]->map); *//一个ecell 2个点*
18. g\_bedge = (int\*)(s.map\_list[2]->map); *//一条b边2个点*
19. g\_becell = (int\*)(s.map\_list[3]->map); *//一个becell 1个cell*
20. g\_x = (double\*)(s.dat\_list[1]->data);*//一个点三个坐标*
21. g\_q = (double\*)(s.dat\_list[2]->data);
22. g\_qold = (double\*)(s.dat\_list[3]->data);
23. g\_res = (double\*)(s.dat\_list[5]->data);
24. g\_adt = (double\*)(s.dat\_list[4]->data);
25. g\_bound = (int\*)(s.dat\_list[0]->data);
26. for (int n = 0; n < g\_ncell; n++) { *//ncell*
27. for (int m = 0; m < 4; m++) {
28. g\_q[4 \* n + m] = qinf[m];
29. g\_res[4 \* n + m] = 0.0f;
30. }
31. }
32. *//为了最后线程0聚合所有数据*
33. qold = (double\*)malloc(4 \* g\_ncell \* sizeof(double));
34. res = (double\*)malloc(4 \* g\_ncell \* sizeof(double));
35. adt = (double\*)malloc(g\_ncell \* sizeof(double));
36. 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：

1. static void compute\_local\_range(int global\_size, int mpi\_comm\_size,
2. int mpi\_rank, int& begin, int& end) {
3. begin = 0;
4. for (int i = 0; i < mpi\_rank; i++) {
5. begin += compute\_local\_size(global\_size, mpi\_comm\_size, i);
6. }
7. end = begin + compute\_local\_size(global\_size, mpi\_comm\_size, mpi\_rank);
8. }

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**：

1. for (int i = 0; i < ncell; i++) {
2. area(
3. g\_x + (g\_cell[4 \* i]) \* 3,
4. g\_x + (g\_cell[4 \* i + 1]) \* 3,
5. g\_x + (g\_cell[4 \* i + 2]) \* 3,
6. g\_x + (g\_cell[4 \* i + 3]) \* 3,
7. g\_q + 4 \* i,
8. g\_adt + i);
9. }
10. 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。

1. MPI\_Reduce(&rms,&g\_rms,1,MPI\_DOUBLE,MPI\_SUM,0, MPI\_COMM\_WORLD);
2. *//聚合数据到线程0*
3. gather\_double\_array(qold, g\_qold + 4 \* cell\_begin, comm\_size, g\_ncell, ncell, 4);
4. gather\_double\_array(res, g\_res + 4 \* cell\_begin, comm\_size, g\_ncell, ncell, 4);
5. gather\_double\_array(adt, g\_adt + cell\_begin, comm\_size, g\_ncell, ncell, 1);
6. if (my\_rank == 0) {
7. printf("ROOT: Total residual %10.5e \n", g\_rms);
8. }

**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：

1. static void gather\_double\_array(double\* g\_array, double\* l\_array,
2. int comm\_size, int g\_size, int l\_size,
3. int elem\_size) {
4. int\* sendcnts = (int\*)malloc(comm\_size \* sizeof(int));
5. int\* displs = (int\*)malloc(comm\_size \* sizeof(int));
6. int disp = 0;
7. for (int i = 0; i < comm\_size; i++) {
8. sendcnts[i] = elem\_size \* compute\_local\_size(g\_size, comm\_size, i);
9. }
10. for (int i = 0; i < comm\_size; i++) {
11. displs[i] = disp;
12. disp = disp + sendcnts[i];
13. }
14. MPI\_Gatherv(l\_array, l\_size\*elem\_size, MPI\_DOUBLE, g\_array, sendcnts, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
15. free(sendcnts);
16. free(displs);
17. }

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。**

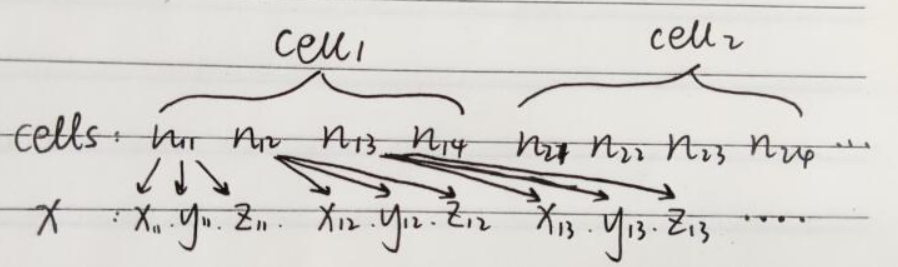
**首先一个问题，对于别的数据该如何处理呢？**这里我想了两种解决方法：

(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。

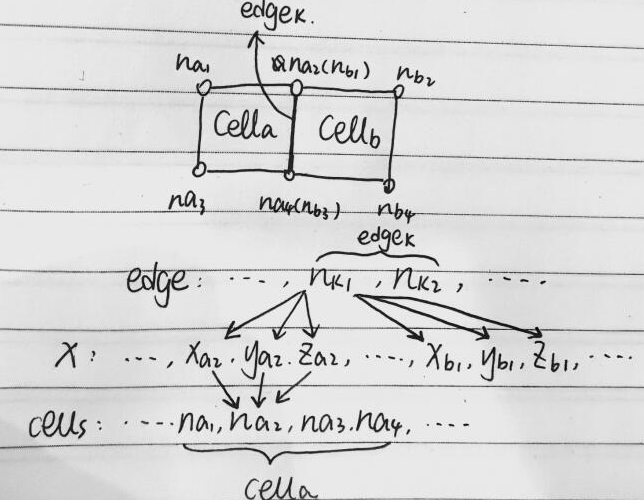
这里的调整指的便是获得结点的新编号。这便是PartitonFrom(edge,pedge,xyz)和PartitonFrom(bedge,pbedge,xyz)的任务。

**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, double\* 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**

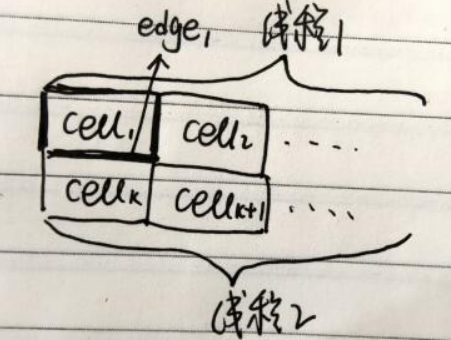
PartitionFrom的任务是将edge bedge中node编号更换成新编号。

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。**

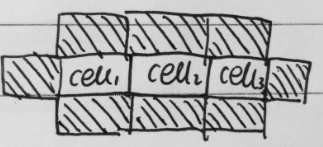
容易想到，那我们也将edge按照cell划分，允许冗余，将每个cell对应的4个edge按cell顺序存储。可是这样仍然存在问题，那就是，ecell该如何处理。ecell是和edge关联的cell的编号，**可能edge的另一个ecell不被划分到本线程**，如下图：



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(double));*//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. for (int i = 0; i < nbedge; i++) {  *//////////////////*
18. int ii = 0;
19. for (int k = 0;k < my\_rank;k++) {
20. ii += compute\_local\_size(g\_nbedge, comm\_size, k);
21. }
22. bcond(
23. partition\_x + (bedge[2 \* i]) \* 3,
24. partition\_x + (bedge[2 \* i + 1]) \* 3,
25. q + (g\_becell[ii + i]) \* 4,
26. adt + (g\_becell[ii+i]),
27. res + (g\_becell[ii+i]) \* 4,
28. bound + i);
29. }

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：

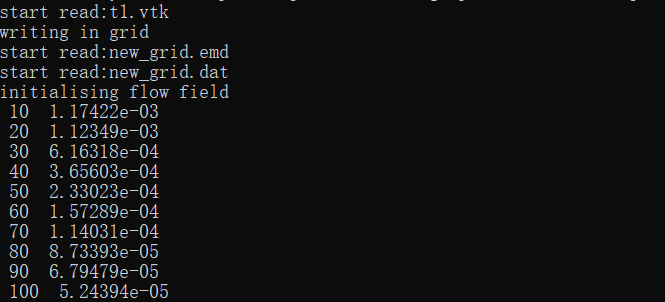
1. partitionFrom(g\_edge, g\_nedge, 4 \* g\_ncell,g\_cell);*//由于x（nodes）换了新的顺序，所以原来的node编号失效了，要将旧的node编号更换成新的*
2. partitionFrom(g\_bedge, g\_nbedge, 4 \* g\_ncell,g\_cell);*//由于x（nodes）换了新的顺序，所以原来的node编号失效了，要将旧的node编号更换成新的*

实验结果

编写测试文件Test.cpp：

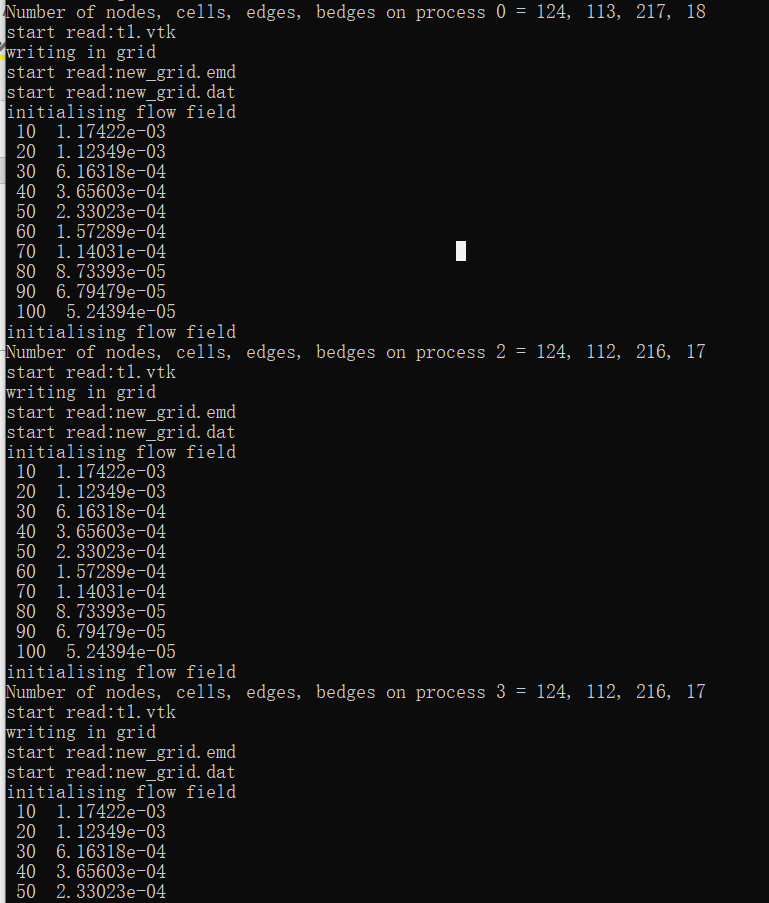
1. #include "Mesh.h"
2. #include "Program.h"
3. #include "ParallelProgram.h"
4. #include "DistributedProgram.h"
5. #include<iostream>
6. int main() {
7. *//test vtk read*
8. Mesh s;
9. s.readvtk("t1.vtk");
10. *//test emd write*
11. s.savetofile("new\_grid.emd");
12. *//test emd read*
13. Mesh s2((char\*)"new\_grid.emd");
14. *//test raw read*
15. Mesh s3;
16. s3.readraw("new\_grid.dat");
17. *//test program*
18. Mesh s4;
19. Program prog(s4);
20. prog.run(0);
21. *//test distributed program*
22. DistributedProgram prog2(s3);
23. int argc = 3;
24. char\* argv[] = { {(char \*)"mpiexec"},{(char\*)"-n"},{(char\*)"4"} };
25. prog2.run(argc,argv);
26. *//test parallel program*
27. ParallelProgram prog3(s3);
28. prog3.run(argc, argv);
29. return 0;
30. }

测试各个格式的读写和串行program：



可见运行正确

测试并行program：



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。