Kokkos Trilinos 学习

曾祥磊

2025 年 7 月 5 日

目录

1 Kokkos 学习

1.1 Kokkos Tutorial

为了打代码和公式方便，使用 LaTeX 进行排版。书接上文，在完成了 Kokkos 的安装和使用后，我们已经能够正常的使用 Kokkos 进行并行计算。从教程开始学习 Kokkos 的使用方法，gemm 算法，SIMD 指令，CrsMatrix，SparseMatrix，CG，GMRES ，欧拉流的 FVM 求解。

Kokkos 目前用的人不多，但在开源 HPC 套件里面逐步适配，目前我了解和使用过的是 Trilinos(sandia 老本家)，分子动力学软件 LAMMPS (想不到吧，这也是 sandia 本家的)，Petsc 套件的 GPU 支持。这帮人里面有不少 c++ 委员会的人，拉起了一个提案，为在 c23 标准下为 c++ 的 STL 提供一个多维数组的接口，实现矩阵向量和张量计算，包含在 std::mdspan 中。

如果你这两个都用过，会非常惊奇的发现 (这俩一帮人写的) 用法和模板参数几乎一致。如下为 mdspan 和 kokkos 的核心模板参数，基本上包括参数类型，多维数组的长度，排布方式，访问方式。

```
template <class T,
          class Extents,
          class LayoutPolicy = std::layout_right,
          class AccessorPolicy = std::default_accessor<T>>
```

```
template <class DataType
      [, class LayoutType]
      [, class MemorySpace]
      [, class MemoryTraits]>
class View
{
  ...
};
```

既然都搞定矩阵表示了，为什么不把矩阵向量计算顺带也搞定。巧了，这帮人也是这么想的，除了 mdspan 提案，还将在 c26 标准中，实现 blas 的所有功能，包含在 std::linalg 头文件中，作为 mdspan 的延申。

1.1.1 Tutorial Step 1

Kokkos 的练习代码包括两个部分，第一部分为原始的没有改为并行的代码，第二部分为改为并行后的修正代码，在此将原始代码保存为注释，修改后代码为正常运行代码。

```
#include <limits>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <Kokkos_Core.hpp>

// 检查命令行输入的N, M, S是否符合要求, nrepeat表示其重复计算次数
void checkSizes(int &N, int &M, int &S, int &nrepeat);

int main(int argc, char *argv[])
{
    int N = -1;
    int M = -1;
    int S = -1;
    int nrepeat = 100;

    // 命令行参数读取
    for (int i = 0; i < argc; i++)
    {
    if ((strcmp(argv[i], "-N") == 0) || (strcmp(argv[i], "-Rows") == 0))
    {
        N = pow(2, atoi(argv[++i]));
        std::cout << "  User N is " << N << std::endl;
    }
    else if ((strcmp(argv[i], "-M") == 0) || (strcmp(argv[i], "-Columns") == 0))
    {
        M = pow(2, atof(argv[++i]));
        std::cout << "  User M is " << M << std::endl;
    }
    else if ((strcmp(argv[i], "-S") == 0) || (strcmp(argv[i], "-Size") == 0))
    {
        S = pow(2, atof(argv[++i]));
        std::cout << "  User S is " << S << std::endl;
    }
    else if (strcmp(argv[i], "-nrepeat") == 0)
    {
        nrepeat = atoi(argv[++i]);
    }
    else if ((strcmp(argv[i], "-h") == 0) || (strcmp(argv[i], "-help") == 0))
    {
        std::cout << "y^T*A*x Options: " << std::endl;
        std::cout << "-Rows (-N) <int>:      rows (default: 4096) " << std::endl;
        std::cout << "-Columns (-M) <int>:   columns (default: 1024) " << std::endl;
```

```
46          std::cout << "-Size (-S) <int>:       matrix (default: N*M) " << std::endl;
47          std::cout << "-nrepeat <int>:         repeats (default: 100) "<< std::endl;
48          std::cout << "-help (-h):             print this message " << std::endl;
49          exit(1);
50      }
51      }
52
53      // 检查参数是否合规
54      checkSizes(N, M, S, nrepeat);
55
56      // Kokkos环境初始化，和MPI_Initialize用法一致，所以代码都需要在
57      // Initialize和Finalize之间
58      Kokkos::initialize(argc, argv);
59      {
60      // 检测可用并行空间
61      // EXERCISE give-away: Choose an Execution Space.
62      // using ExecSpace = Kokkos::Serial;
63      // using ExecSpace = Kokkos::Threads;
64      // using ExecSpace = Kokkos::OpenMP;
65      // using ExecSpace = Kokkos::Cuda;
66
67      // EXERCISE: Choose device memory space.
68      // using MemSpace = Kokkos::HostSpace;
69      // using MemSpace = Kokkos::CudaSpace;
70      // using MemSpace = Kokkos::CudaUVMSpace;
71
72      #ifdef KOKKOS_ENABLE_CUDA
73      #define MemSpace Kokkos::CudaSpace
74      #endif
75
76      #ifdef KOKKOS_ENABLE_HIP
77      #define MemSpace Kokkos::Experimental::HIPSpace
78      #endif
79
80      #ifdef KOKKOS_ENABLE_OPENMPTARGET
81      #define MemSpace Kokkos::OpenMPTargetSpace
82      #endif
83
84      #ifndef MemSpace
85      #define MemSpace Kokkos::HostSpace
86      #endif
87
88      // 并行空间
89      using ExecSpace = MemSpace::execution_space;
90
91      // EXERCISE give-away: Use a RangePolicy.
92      // using range_policy = Kokkos::RangePolicy<ExecSpace>;
93      // 执行策略，在此模板只指定了并行空间
94      using range_policy = Kokkos::RangePolicy<ExecSpace>;
```

```
95
96      // EXERCISE give-away: Choose a Layout.
97      // EXERCISE: When exercise is correctly implemented, then
98      //           either layout will generate the correct answer.
99      //           However, performance will be different!
100
101     // using Layout = Kokkos::LayoutLeft;
102     using Layout = Kokkos::LayoutRight;
103
104     // Allocate y, x vectors and Matrix A on device.
105     // EXERCISE: Use MemSpace and Layout.
106     // Kokkos::View是Kokkos的通用的数组类型，类似std::shared_ptr
107     // 1D可以视为向量类型，2D视为矩阵类型
108     using ViewVectorType = Kokkos::View<double *, Layout, MemSpace>;
109     using ViewMatrixType = Kokkos::View<double **, Layout, MemSpace>;
110
111     // 定义需要的矩阵A和向量y，x
112     // 传入的字符串作为lable唯一的确定对应的实例，在debug中非常有效
113     ViewVectorType y("y", N);
114     ViewVectorType x("x", M);
115     ViewMatrixType A("A", N, M);
116
117     // CUDA中的数据不能被CPU访问和读取，如果需要进行改动，需要先创建
118     // CPU可访问的数据，也就是ViewVectorType::HostMirror
119     ViewVectorType::HostMirror h_y = Kokkos::create_mirror_view(y);
120     ViewVectorType::HostMirror h_x = Kokkos::create_mirror_view(x);
121     ViewMatrixType::HostMirror h_A = Kokkos::create_mirror_view(A);
122
123     // 初始化y
124     for (int i = 0; i < N; ++i)
125     {
126         h_y(i) = 1;
127     }
128
129     // 初始化x
130     for (int i = 0; i < M; ++i)
131     {
132         h_x(i) = 1;
133     }
134
135     // 初始化A
136     for (int j = 0; j < N; ++j)
137     {
138         for (int i = 0; i < M; ++i)
139         {
140         h_A(j, i) = 1;
141         }
142     }
143
```

```
144     // 由CPU可访问空间转移到对应的并行空间
145     Kokkos::deep_copy(y, h_y);
146     Kokkos::deep_copy(x, h_x);
147     Kokkos::deep_copy(A, h_A);
148 
149     // 计时器
150     Kokkos::Timer timer;
151 
152     for (int repeat = 0; repeat < nrepeat; repeat++)
153     {
154 
155         double result = 0;
156 
157         // 将yAx分解为y(j) * \sum_{i=0,M} A(j,i) * x(i)
158         // range_policy(start,end,...)指定起始和结束index
159         // KOKKOS_LAMBDA自动根据并行空间指定LAMBDA函数的初始化类型
160         Kokkos::parallel_reduce("yAx", range_policy(0, N),
161         KOKKOS_LAMBDA (int j, double &update) {
162         double temp2 = 0;
163 
164         for ( int i = 0; i < M; ++i ) {
165         temp2 += A( j, i ) * x( i );
166         }
167 
168         update += y( j ) * temp2; }, result);
169 
170         if (repeat == (nrepeat - 1))
171         {
172         std::cout << "  Computed result for " << N
173                     << " x " << M << " is " << result << std::endl;
174         }
175     }
176     // 计算时间
177     double time = timer.seconds();
178     // 计算带宽
179     double Gbytes = 1.0e-9 * double(sizeof(double) * (M + M * N + N));
180 
181     std::cout << "  N( " << N << " ) "
182               << "  M( " << M << " ) "
183               << "  nrepeat ( " << nrepeat << " ) "
184               << "  problem( " << Gbytes * 1000 << " MB ) "
185               << "  time( " << time << " s ) "
186               << "  bandwidth( " << Gbytes * nrepeat / time << " GB/s )" << std::
    endl;
187     }
188     // 结束Kokkos
189     Kokkos::finalize();
190 
191     return 0;
```

```
192 }
193
194 void checkSizes(int &N, int &M, int &S, int &nrepeat)
195 {
196
197     if (S == -1 && (N == -1 || M == -1))
198     {
199     S = pow(2, 22);
200     if (S < N)
201         S = N;
202     if (S < M)
203         S = M;
204     }
205
206     if (S == -1)
207     S = N * M;
208
209     if (N == -1 && M == -1)
210     {
211     if (S > 1024)
212     {
213         M = 1024;
214     }
215     else
216     {
217         M = S;
218     }
219     }
220
221     if (M == -1)
222     M = S / N;
223
224     if (N == -1)
225     N = S / M;
226 }
```

1.1.2 Tutorial Step 2

这个教程主要介绍不同的储存顺序对于计算性能的影响，尤其是对于不同的架构而言，对于 CPU 来说，由于其有缓存机制，每次读取都不会单独读取一个，而是一次性读取相邻内存上的多个值放入缓存，那么在计算矩阵向量乘法的时候，在遍历矩阵的列时，每次读取多个列值进行计算，整行计算完成后，步进到下一行。对于 GPU 来说，由于存在合并访问机制，理想情况下合并效为 100，否则会浪费大部分的带宽，所以为列优先 (GPU 部分比较胡编乱造，只能说看个乐)。

```
#include <Kokkos_Core.hpp>
#include <Kokkos_Timer.hpp>
#include <iostream>
#include <cstdio>
// 并行空间
using MemSpace = Kokkos::CudaSpace;

// 二维的数组类型，LayoutLeft -> 列优先，LayoutRight -> 行优先
// 对于CPU来说，矩阵Axb的过程中会大量读取和b中的值，为了提高
// 计算效率，如果Axb中读取一次就可以得到A(i,j)->A(i,j+n)无疑
// 会大大提高计算效率(Cache)，对应的是行优先。进一步，如果一次将这读取的
// n个数一起计算，会有极为可观的收益(SIMD)
// 对GPU来说，GPU核心非常多，由于存在合并访问，行优先会浪费很多的带宽，
// 一般是列优先
using left_type = Kokkos::View<double **, Kokkos::LayoutLeft, MemSpace>;
using right_type = Kokkos::View<double **, Kokkos::LayoutRight, MemSpace>;

using view_type = Kokkos::View<double *, MemSpace>;

// a.extent(0)，a.extent(1)指的是View的维数大小，NxM的N和M
template <class ViewType>
struct init_view
{
    ViewType a;
    init_view(ViewType a_) : a(a_) {}

    using size_type = typename ViewType::size_type;

    KOKKOS_INLINE_FUNCTION
    void operator()(const typename ViewType::size_type i) const
    {
    for (size_type j = 0; j < static_cast<size_type>(a.extent(1)); ++j)
    {
        a(i, j) = 1.0 * a.extent(0) * i + 1.0 * j;
    }
    }
};

template <class ViewType1, class ViewType2>
struct contraction
```

```
{
    view_type a;
    typename ViewType1::const_type v1;
    typename ViewType2::const_type v2;
    contraction(view_type a_, ViewType1 v1_, ViewType2 v2_)
        : a(a_), v1(v1_), v2(v2_) {}

    using size_type = typename view_type::size_type;

    KOKKOS_INLINE_FUNCTION
    void operator()(const view_type::size_type i) const
    {
    for (size_type j = 0; j < static_cast<size_type>(a.extent(1)); ++j)
    {
        a(i) = v1(i, j) * v2(j, i);
    }
    }
};

// 向量内积
struct dot
{
    view_type a;
    dot(view_type a_) : a(a_) {}
    using value_type = double;
    KOKKOS_INLINE_FUNCTION
    void operator()(const view_type::size_type i, double &lsum) const
    {
    lsum += a(i) * a(i);
    }
};

int main(int argc, char *argv[])
{
    Kokkos::initialize(argc, argv);
    {
    int size = 10000;
    view_type a("A", size);

    // Define two views with LayoutLeft and LayoutRight.
    left_type l("L", size, 10000);
    right_type r("R", size, 10000);

    // Initialize the data in the views.
    Kokkos::parallel_for(size, init_view<left_type>(l));
    Kokkos::parallel_for(size, init_view<right_type>(r));
    // MPI_Barrier
    Kokkos::fence();

```

```
90      Kokkos::Timer time1;
91      Kokkos::parallel_for(size, contraction<left_type, right_type>(a, l, r));
92      Kokkos::fence();
93      double sec1 = time1.seconds();
94
95      double sum1 = 0;
96      Kokkos::parallel_reduce(size, dot(a), sum1);
97      Kokkos::fence();
98
99      Kokkos::Timer time2;
100     Kokkos::parallel_for(size, contraction<right_type, left_type>(a, r, l));
101     Kokkos::fence();
102     double sec2 = time2.seconds();
103
104     double sum2 = 0;
105     Kokkos::parallel_reduce(size, dot(a), sum2);
106
107     printf("Result Left/Right %f Right/Left %f (equal result: %i)\n", sec1,
108             sec2, sum2 == sum1);
109     }
110     Kokkos::finalize();
111 }
```

1.1.3 Tutorial Step 3 - GEMM and Euler Flow

这个案例主要是使用 GEMM 算法快速计算矩阵和矩阵的乘法，主要看不懂的点集中在模板化，和一堆这个作者自己定义的类型上面，之前用 Trilinos 的时候也是差不多这样坐牢，纯纯的坐牢，想不做是不可能的，感觉不如 Petsc 使用起来上手。其次就是 CRS 矩阵的具体实现过程，如果有兴趣可以参考注释看一下，sandia 实验室的代码都一个样子，封装完成后使用 List 进行具体实现。

虽然到现在我还是对 Kokkos 这东西一脸懵逼，但还是要挖个大坑，计划搞明白这个使用 Kokkos，FVM 和 DG 计算时变欧拉流的代码，目前已经能够完全跑起来，使用 paraview 进行后处理，可能得补以下 FVM 的东西，DG 之前就看过。在这个过程中连带这学一下 Kokkos 的稀疏矩阵和一些抽象的 simd 指令。

```
#include <limits>
#include <cstdio>
#include <cstdlib>
#include <cstring>

#include <Kokkos_Core.hpp>
#include <KokkosSparse_CrsMatrix.hpp>
#include <KokkosSparse_spgemm.hpp>

void checkSizes(int &N)
{
    // If N is undefined, set it to 2^10 = 1024.
    if (N == -1)
    N = 1024;

    printf("  Number of Rows N = %d, Number of Cols N = %d, Total nnz = %d\n", N, N,
     2 + 3 * (N - 2) + 2);

    // Check sizes.
    if (N < 0)
    {
    printf("  Sizes must be greater than 0.\n");
    exit(1);
    }
}

// 这模板外一堆加上参数传入过程，一眼丁真，遮沙避风了
template <typename crsMat_t>
void makeSparseMatrix(
    typename crsMat_t::StaticCrsGraphType::row_map_type::non_const_type &ptr,
    typename crsMat_t::StaticCrsGraphType::entries_type::non_const_type &ind,
    typename crsMat_t::values_type::non_const_type &val,
    typename crsMat_t::ordinal_type &numRows,
    typename crsMat_t::ordinal_type &numCols,
    typename crsMat_t::size_type &nnz,
    const int whichMatrix)
```

```
36 {
37     // 此处解释一下CRS矩阵的基本构成，首先是row_map用于储存每一行上的第一个非零元的
       位置
38     // 注意这里说的非零元的位置和下面的index都不是指的在稀疏矩阵(A(i,j))，而是在非零
       元数组
39     // 中的位置，从0到N，就是稀疏模板
40     typedef typename crsMat_t::StaticCrsGraphType::row_map_type::non_const_type
       ptr_type;
41     // 其次需要包括一个在每列上非零元的位置，一般为index，或者entry(非零元入口)
42     typedef typename crsMat_t::StaticCrsGraphType::entries_type::non_const_type
       ind_type;
43     // 最后就是一个平平无奇的数组，从0开始到N结束，按照每一行进行记录每个非零元
44     typedef typename crsMat_t::values_type::non_const_type val_type;
45     // 这个就是index
46     typedef typename crsMat_t::ordinal_type lno_t;
47     // int
48     typedef typename crsMat_t::size_type size_type;
49     // 存储的数据类型
50     typedef typename crsMat_t::value_type scalar_t;
51
52     using Kokkos::HostSpace;
53     using Kokkos::MemoryUnmanaged;
54     using Kokkos::View;
55
56     if (whichMatrix == 0)
57     {
58     // 这里按照行来填充一个矩阵，矩阵的主对角线为2，主对角线两侧的对角线为-1
59     // 其余为0的一个方阵
60     numCols = numRows;
61     // 非零元的个数
62     nnz = 2 + 3 * (numRows - 2) + 2;
63     // 这里多声明一个位置是为了存储最大的列数量，用于提前给其他算法提供一个
64     // 最大的内存分配大小
65     size_type *ptrRaw = new size_type[numRows + 1];
66     // 列索引向量
67     lno_t *indRaw = new lno_t[nnz];
68     // 非零元的值
69     scalar_t *valRaw = new scalar_t[nnz];
70
71     scalar_t two = 2.0;
72     scalar_t mone = -1.0;
73
74     // Add rows one-at-a-time
75     for (int i = 0; i < (numRows + 1); i++)
76     {
77         // [2, -1, ... ,0]
78         if (i == 0)
79         {
80         ptrRaw[0] = 0;
```

```
81          indRaw[0] = 0;
82          indRaw[1] = 1;
83          valRaw[0] = two;
84          valRaw[1] = mone;
85          }
86          else if (i == numRows)
87          {
88          ptrRaw[numRows] = nnz;
89          }
90          // [0, ..., -1, 2]
91          else if (i == (numRows - 1))
92          {
93          ptrRaw[i] = 2 + 3 * (i - 1);
94          indRaw[2 + 3 * (i - 1)] = i - 1;
95          indRaw[2 + 3 * (i - 1) + 1] = i;
96          valRaw[2 + 3 * (i - 1)] = mone;
97          valRaw[2 + 3 * (i - 1) + 1] = two;
98          }
99          // [0, ..., -1, 2, -1, ..., 0]
100         else
101         {
102         ptrRaw[i] = 2 + 3 * (i - 1);
103         indRaw[2 + 3 * (i - 1)] = i - 1;
104         indRaw[2 + 3 * (i - 1) + 1] = i;
105         indRaw[2 + 3 * (i - 1) + 2] = i + 1;
106         valRaw[2 + 3 * (i - 1)] = mone;
107         valRaw[2 + 3 * (i - 1) + 1] = two;
108         valRaw[2 + 3 * (i - 1) + 2] = mone;
109         }
110     }
111
112     // 用填充好的三个向量初始Kokkos::View
113     // 创建View
114     ptr = ptr_type("ptr", numRows + 1);
115     ind = ind_type("ind", nnz);
116     val = val_type("val", nnz);
117
118     // 用HostMirror将之前填充好的内容进行封装
119     typename ptr_type::HostMirror::const_type ptrIn(ptrRaw, numRows + 1);
120     typename ind_type::HostMirror::const_type indIn(indRaw, nnz);
121     typename val_type::HostMirror::const_type valIn(valRaw, nnz);
122
123     // 转移到View上
124     Kokkos::deep_copy(ptr, ptrIn);
125     Kokkos::deep_copy(ind, indIn);
126     Kokkos::deep_copy(val, valIn);
127
128     delete[] ptrRaw;
129     delete[] indRaw;
```

```
130     delete[] valRaw;
131     }
132     else
133     { // whichMatrix != 0
134     std::ostringstream os;
135     os << "Invalid whichMatrix value " << whichMatrix
136         << ".  Valid value(s) include " << 0 << ".";
137     throw std::invalid_argument(os.str());
138     }
139 }
140
141 template <typename crsMat_t>
142 crsMat_t makeCrsMatrix(int numRows)
143 {
144     typedef typename crsMat_t::StaticCrsGraphType graph_t;
145     typedef typename graph_t::row_map_type::non_const_type lno_view_t;
146     typedef typename graph_t::entries_type::non_const_type lno_nnz_view_t;
147     typedef typename crsMat_t::values_type::non_const_type scalar_view_t;
148     typedef typename crsMat_t::ordinal_type lno_t;
149     typedef typename crsMat_t::size_type size_type;
150
151     lno_view_t ptr;
152     lno_nnz_view_t ind;
153     scalar_view_t val;
154     lno_t numCols;
155     size_type nnz;
156
157     const int whichMatrix = 0;
158     makeSparseMatrix<crsMat_t>(ptr, ind, val, numRows, numCols, nnz, whichMatrix);
159     // 使用填充好的View对矩阵进行初始化
160     return crsMat_t("A", numRows, numCols, nnz, val, ptr, ind);
161 }
162
163 int main(int argc, char *argv[])
164 {
165     // Use current time as seed for random generator
166     srand(time(0));
167
168     int N = -1; // number of rows 2^10
169
170     // Read command line arguments.
171     for (int i = 0; i < argc; i++)
172     {
173     if (strcmp(argv[i], "-N") == 0)
174     {
175         N = atoi(argv[++i]);
176         printf("  User N is %d\n", N);
177     }
178     else if ((strcmp(argv[i], "-h") == 0) || (strcmp(argv[i], "-help") == 0))
```

```
179     {
180         printf("  SpGEMM (C=A*A) Options:\n");
181         printf("  -N <int>:      determines number of rows (columns) (default: 2^10
        = 1024)\n");
182         printf("  -help (-h):            print this message\n\n");
183         exit(1);
184     }
185     }
186
187     // Check sizes.
188     checkSizes(N);
189
190     Kokkos::initialize(argc, argv);
191     {
192     // Typedefs
193     typedef double scalar_type;
194     typedef int ordinal_type;
195     typedef int size_type;
196     typedef Kokkos::DefaultExecutionSpace device_type;
197     typedef KokkosSparse::CrsMatrix<scalar_type, ordinal_type, device_type, void,
        size_type> crs_matrix_type;
198
199     // 创建稀疏矩阵A
200     crs_matrix_type A = makeCrsMatrix<crs_matrix_type>(N);
201
202     // 这东西是sandia的人写的，如果之前有用过Trilinos的可以直接把这个看成一个
        ParameterList
203     typedef KokkosKernels::Experimental::KokkosKernelsHandle<size_type, ordinal_type
        , scalar_type,
204                                                               typename device_type
        ::execution_space,
205                                                               typename device_type
        ::memory_space,
206                                                               typename device_type
        ::memory_space>
207         KernelHandle;
208
209     KernelHandle kh;
210
211     // Set parameters in the handle
212     kh.set_team_work_size(16);
213     kh.set_dynamic_scheduling(true);
214     // kh.set_verbose(true);
215
216     // 指定gemm的具体实现方式
217     std::string myalg("SPGEMM_KK_MEMORY");
218     KokkosSparse::SPGEMMAlgorithm spgemm_algorithm = KokkosSparse::
        StringToSPGEMMAlgorithm(myalg);
219     // 创建这个指定的gemm计算器
```

```
220     kh.create_spgemm_handle(spgemm_algorithm);
221
222     crs_matrix_type C;
223
224     Kokkos::Timer timer;
225
226     // 大型稀疏矩阵的计算和求解基本分为两个部分，symbolic和numeric
227     // symbolic可以认为是进行分析步，前处理
228     // numeric是真正意义上在进行计算
229     // C = A * A
230     KokkosSparse::spgemm_symbolic(kh, A, false, A, false, C);
231
232     Kokkos::fence();
233     double symbolic_time = timer.seconds();
234     timer.reset();
235     // EXERCISE: Call the numeric phase
236     // EXERCISE hint: KokkosSparse::spgemm_numeric(...)
237     KokkosSparse::spgemm_numeric(kh, A, false, A, false, C);
238
239     Kokkos::fence();
240     double numeric_time = timer.seconds();
241
242     // Destroy the SpGEMM handle
243     kh.destroy_spgemm_handle();
244
245     // Print results (problem size, time, number of iterations and final norm
    residual).
246     printf("    Results: N( %d ), overall spgemm time( %g s ), symbolic time( %g s )
    , numeric time( %g s )\n",
247             N, symbolic_time + numeric_time, symbolic_time, numeric_time);
248     }
249
250     Kokkos::finalize();
251
252     return 0;
253 }
```

1.1.4 Tutorial Step 4 - Dual View and BlockJacobi

正常的数据类型为 Kokkos::View<double*,MemSpace,...>，为了使得这套代码能够自动适用于 CPU 和 GPU，使用 Kokkos::DualView<double *> 进行计算，这个可以分为两个部分理解，如果都在 HostSpace 空间中，那么 DualView 就是原始 View 的一个别名，指向同一地址；如果在 DeviceSpace，将创建一个在 HostSpace 的引用，让 CPU 访问。

以下为头文件 functors.hpp 中的内容

```
#include <Kokkos_Core.hpp>
#include <Kokkos_DualView.hpp>

typedef Kokkos::DualView<double *> view_type;
const double density_0 = 1;
const double temperature_0 = 300;

template <class ExecutionSpace>
struct ComputePressure
{

    static constexpr double gasConstant = 1;

    typedef ExecutionSpace execution_space;
    // std::conditional类似三目运算符, i == j ? 1 : 0
    // std::is_same<ExecutionSpace, Kokkos::DefaultExecutionSpace>::value 当前的ExecutionSpace和DefaultExecutionSpace
    // 是否一致,是则返回view_type::memory_space,否则则返回view_type::host_mirror_space
    typedef typename std::conditional<std::is_same<ExecutionSpace, Kokkos::DefaultExecutionSpace>::value,
                                      view_type::memory_space, view_type::host_mirror_space>::type memory_space;

    // scalar_array_type = double *
    // const_data_type = const double *
    Kokkos::View<view_type::scalar_array_type, view_type::array_layout,
                 memory_space>
        pressure;
    Kokkos::View<view_type::const_data_type, view_type::array_layout,
                 memory_space, Kokkos::MemoryRandomAccess>
        temperature;
    Kokkos::View<view_type::const_data_type, view_type::array_layout,
                 memory_space, Kokkos::MemoryRandomAccess>
        density;

    // 初始化方法
    ComputePressure(view_type dv_pressure, view_type dv_temperature, view_type dv_density)
    {
        // 获得正确的符合当前运行空间的View
```

```
37          pressure = dv_pressure.template view<memory_space>();
38          density = dv_density.template view<memory_space>();
39          temperature = dv_temperature.template view<memory_space>();
40          // 将Device上的View和Host上的View进行同步,类似MPI_Bcast(&data,Comm)
41          dv_pressure.sync<memory_space>();
42          dv_temperature.sync<memory_space>();
43          dv_density.sync<memory_space>();
44          // 表明dv_pressure被改动过
45          dv_pressure.modify<memory_space>();
46      }
47
48      //  p = \rho * gasConst * T
49      KOKKOS_INLINE_FUNCTION
50      void operator()(const int i) const
51      {
52          pressure(i) = density(i) * gasConstant * temperature(i);
53      }
54  };
55
56  // 剩下两个类似
57  template <class ExecutionSpace>
58  struct ComputeInternalEnergy
59  {
60
61      static constexpr double C_v = 1;
62
63      typedef ExecutionSpace execution_space;
64
65      typedef typename std::conditional<std::is_same<ExecutionSpace, Kokkos::
        DefaultExecutionSpace>::value,
66                                        view_type::memory_space, view_type::
        host_mirror_space>::type memory_space;
67
68      Kokkos::View<view_type::scalar_array_type, view_type::array_layout, memory_space
        > energy;
69
70      Kokkos::View<view_type::const_data_type, view_type::array_layout, memory_space,
        Kokkos::MemoryRandomAccess> temperature;
71
72      ComputeInternalEnergy(view_type dv_energy, view_type dv_temperature)
73      {
74          energy = dv_energy.template view<memory_space>();
75          temperature = dv_temperature.template view<memory_space>();
76
77          dv_energy.sync<memory_space>();
78          dv_temperature.sync<memory_space>();
79
80          // Mark energy as modified
81          dv_energy.modify<memory_space>();
```

```
82      }
83
84      KOKKOS_INLINE_FUNCTION
85      void operator()(const int i) const
86      {
87          energy(i) = C_v * temperature(i);
88      }
89  };
90
91  template <class ExecutionSpace>
92  struct ComputeEnthalpy
93  {
94
95      typedef ExecutionSpace execution_space;
96
97      typedef typename std::conditional<std::is_same<ExecutionSpace, Kokkos::
        DefaultExecutionSpace>::value,
98                                        view_type::memory_space, view_type::
        host_mirror_space>::type memory_space;
99
100     Kokkos::View<view_type::scalar_array_type, view_type::array_layout, memory_space
        > enthalpy;
101
102     Kokkos::View<view_type::const_data_type, view_type::array_layout, memory_space,
        Kokkos::MemoryRandomAccess> density;
103     Kokkos::View<view_type::const_data_type, view_type::array_layout, memory_space,
        Kokkos::MemoryRandomAccess> pressure;
104     Kokkos::View<view_type::const_data_type, view_type::array_layout, memory_space,
        Kokkos::MemoryRandomAccess> energy;
105
106     ComputeEnthalpy(view_type dv_enthalpy, view_type dv_energy, view_type
        dv_pressure, view_type dv_density)
107     {
108
109         enthalpy = dv_enthalpy.view<memory_space>();
110         density = dv_density.view<memory_space>();
111         pressure = dv_pressure.view<memory_space>();
112         energy = dv_energy.view<memory_space>();
113
114         dv_density.sync<memory_space>();
115         dv_pressure.sync<memory_space>();
116         dv_energy.sync<memory_space>();
117
118         // Mark enthalpy as modified
119         dv_enthalpy.modify<memory_space>();
120     }
121
122     KOKKOS_INLINE_FUNCTION
123     void operator()(const int i) const
```

```
124     {
125         enthalpy(i) = energy(i) + pressure(i) / density(i);
126     }
127 };
```

以下为主文件中的内容

```
1  #include <iostream>
2  #include "functors.hpp"
3
4  /*
5      这个案例主要是使用DualView实现在CPU和GPU上自动识别，提高不同平台之间的可移植性
6      相比于之前的CRS稀疏矩阵来说还是过于正常了，主要头文件中主要包括三个结构体，
7      实现计算能量，压力，焓。
8  */
9
10 // 初始化密度和温度
11 void load_state(view_type density, view_type temperature);
12 // 计算压力
13 void compute_pressure(view_type pressure, view_type density, view_type temperature);
14 // 计算能量
15 void compute_internal_energy(view_type energy, view_type temperature);
16 // 计算焓值
17 void compute_enthalpy(view_type enthalpy, view_type energy, view_type pressure,
       view_type density);
18 // 检查计算结果
19 void check_results(view_type pressure, view_type energy, view_type enthalpy);
20
21 int main(int narg, char *arg[])
22 {
23
24     std::cout << "initializing kokkos....." << std::endl;
25
26     Kokkos::initialize(narg, arg);
27
28     std::cout << "......done." << std::endl;
29     {
30     const int size = 1000000;
31
32     // 创建DualView,初始化和View类似
33     view_type pressure("pressure", size);
34     view_type density("density", size);
35     view_type temperature("temperature", size);
36     view_type energy("energy", size);
37     view_type enthalpy("enthalpy", size);
38
39     load_state(density, temperature);
40
41     // this section of code is supposed to mimic the structure of a time loop in a
42     // more complex physics app
```

```
43      const size_t maxSteps = 1;
44      for (size_t step = 0; step < maxSteps; ++step)
45      {
46          compute_pressure(pressure, density, temperature);
47          compute_internal_energy(energy, temperature);
48          compute_enthalpy(enthalpy, energy, pressure, density);
49      }
50
51      check_results(pressure, energy, enthalpy);
52      }
53
54      Kokkos::finalize();
55  }
56
57  // 初始化压力和温度View
58  void load_state(view_type density, view_type temperature)
59  {
60      // Host View Mirror
61      view_type::t_host h_density = density.h_view;
62      view_type::t_host h_temperature = temperature.h_view;
63
64      // extent(0)表示View这个多维数组的各个方向的维度
65      // Kokkos::View<double*, MemSpace>,对应一维数组,extent(0)为向量长度
66      // Kokkos::View<double**, MemSpace>,对应二维数组,extent(0)和extent(1)为两个方向
        的长度
67      for (view_type::size_type j = 0; j < h_density.extent(0); ++j)
68      {
69      h_density(j) = density_0;
70      h_temperature(j) = temperature_0;
71      }
72
73      // 标记为被改动，这样sync同步的时候会实际的复制数组
74      density.modify<view_type::host_mirror_space>();
75      temperature.modify<view_type::host_mirror_space>();
76  }
77
78  // 三个结构体的包装
79  void compute_pressure(view_type pressure, view_type density, view_type temperature)
80  {
81      const int size = pressure.extent(0);
82
83      Kokkos::parallel_for(size, ComputePressure<view_type::execution_space>(pressure,
         temperature, density));
84      Kokkos::fence();
85  }
86
87  void compute_internal_energy(view_type energy, view_type temperature)
88  {
89
```

```
90      const int size = energy.extent(0);
91      Kokkos::parallel_for(size, ComputeInternalEnergy<view_type::execution_space>(
        energy, temperature));
92      Kokkos::fence();
93  }
94
95  void compute_enthalpy(view_type enthalpy, view_type energy, view_type pressure,
        view_type density)
96  {
97
98      const int size = enthalpy.extent(0);
99
100     Kokkos::parallel_for(size, ComputeEnthalpy<view_type::execution_space>(enthalpy,
         energy, pressure, density));
101     Kokkos::fence();
102 }
103
104 // 检查计算结果
105 void check_results(view_type dv_pressure, view_type dv_energy, view_type dv_enthalpy
        )
106 {
107
108     const double R = ComputePressure<view_type::host_mirror_space>::gasConstant;
109     const double thePressure = R * density_0 * temperature_0;
110
111     const double cv = ComputeInternalEnergy<view_type::host_mirror_space>::C_v;
112     const double theEnergy = cv * temperature_0;
113
114     const double theEnthalpy = theEnergy + thePressure / density_0;
115
116     auto pressure = dv_pressure.h_view;
117     auto energy = dv_energy.h_view;
118     auto enthalpy = dv_enthalpy.h_view;
119
120     dv_pressure.sync<view_type::host_mirror_space>();
121     dv_energy.sync<view_type::host_mirror_space>();
122     dv_enthalpy.sync<view_type::host_mirror_space>();
123
124     double pressureError = 0;
125     double energyError = 0;
126     double enthalpyError = 0;
127     const int size = energy.extent(0);
128     for (int i = 0; i < size; ++i)
129     {
130     pressureError += (pressure(i) - thePressure) * (pressure(i) - thePressure);
131     energyError += (energy(i) - theEnergy) * (energy(i) - theEnergy);
132     enthalpyError += (enthalpy(i) - theEnthalpy) * (enthalpy(i) - theEnthalpy);
133     }
134
```

```
135     std::cout << "pressure error = " << pressureError << std::endl;
136     std::cout << "energy error = " << energyError << std::endl;
137     std::cout << "enthalpy error = " << enthalpyError << std::endl;
138 }
```

1.1.5 Kokkos - Euler Equation

开始挖坑，主要是最近比较闲（划掉），开始准备准备面试啥的，把之前的分析和代数都翻出来看看，以及材力，弹性力学，但流体主要是看个乐子，买了本 DG 的书到现在还没翻过，本着买了不看是浪费的原则，打算认证的学习一下，为了能够不是盲目的两眼抹黑。

使用 Kokkos 和 Deal.II 两个库进行学习，github 上有这两个库的 Euler 气体方程的计算案例，Kokkos 使用的是 FVM 进行求解（主要 FEM 的自由度处理比较抽象，还有 RT 单元的基函数），Deal.II 使用的是 DG 进行计算 (除了电磁的单元) 没见过用到 5 阶单元的，自由度数量非常多，下面的图是 9.2M 自由度跑出来的结果，在 3 马赫下的圆柱扰流，据作者自己说服务器跑了一天才搞定，具体的两个项目可以点击查看 Kokkos FVM Euler以及 Deal.II DG Euler/Compressible NV

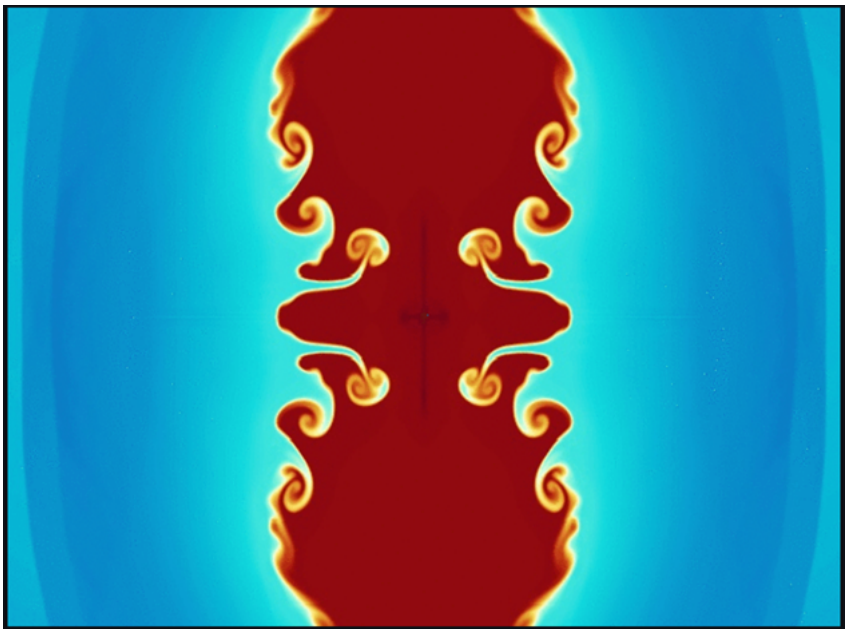

图 1: FVM-Euler 方程计算结果

Kokkos 具体的技术其实主要是利用这个架构，真正意义上的预处理，GMG,AMG, 之类的东西是没有的，基本上就是一个 SOR，BlockJacobi 作为预处理，简单看了一下，规整网格，写了一个读取配置的，输出 vti 的，好处是可以开启 SIMD 加速，以及可以在 GPU 上跑，但其实我觉得没有好的预处理感觉规模稍微大点就是一眼寄。

DG 比较抽象，涉及到分块矩阵的预处理，正交补，以及这个方程本身就是个非线性方程，一般牛顿- 辛普森方程的雅可比矩阵还得处理一下，时变问题的 CN 离散方法，顺便一提这位直接不想写非线性的雅可比，使用是自动微分，主打一个看不懂，以及网格自适应（只能说还好不是 p 自适应）

顺便一说，这东西我是纯粹看个乐子，主打一手开心，顺便比较一下 FVM 和 DG 的差别，目前按照我的理解，FVM 是 0 阶的 DG，但我印象里面 DG 对于第一类边界条件是添加在弱形式里面，或者加上惩罚，不知道 FVM 的边界是什么情况。为了能够看明白这个东西，将会从比较简单的运输方程到斯托克斯到 ins，最后到 Euler，至于什么迎风修正，SUPG 之类后面再看看。

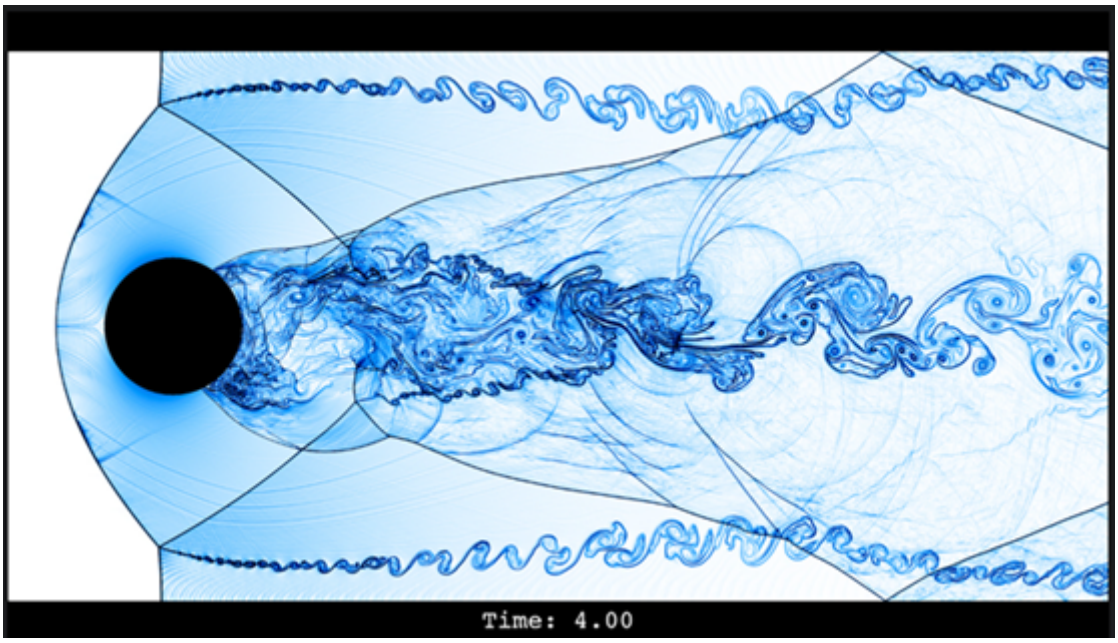

图 2: DG-Euler 方程计算结果

将从 DG 开始，看看这个运输方程是怎么解的，到时候会对比一下连续单元和 DG 单元的计算结果，看看表面的跳动怎么样。至于说流体力学是什么，我反正不知道，我只会方程离散到弱形式，为什么这样，以及一些结论可谓一窍不通，单纯的从方程到离散求解的过程。关于自动微分之前解非线性的时候用过一两回，我只能说用过的都说好，求什么变分导数，感觉不如残差线性化，尤其对于两相流这样的几个方程套在一起，再掺和一个水平集描述，可谓是群贤毕至，满汉全席，为了身心健康着想，还是自动微分实在，放弃智力，拥抱现代 c++ 模板。

最后看看能不能衔接上 occt 参数化建模优化一下经典的翅膀形状，又是最不想动脑的一集，直接自动微分，时变问题直接离散伴随，优化 LP，顺便看看能不能调一下 Tpetra 让代码在 GPU 上跑跑。

1.1.6 Kokkos - Linear advection equation

将从对流方程开始坐牢，这个方程的形式不算复杂，主要是使用连续 FEM 和离散 FEM 进行计算验证，众所周知，对于这个双曲方程来说，连续的 FEM 计算结果会出现非常的的震荡，并且这种震荡不会随着网格的细化而改善，主要原因是对流方程只约束了速度场的某一个方向导数上的值，但对于对应的正交方向没有约束，导致如果在边界或这某处出现震荡，这个震荡会沿着特征线扩散到起点处。

首先来看一下 SUPG 修正后的连续 FEM 计算方法，所有内容都只是图个乐，看个乐子。首先是方程形式：

$$\beta(x,y,z)\nabla u = f \qquad u = g\ , \quad u \in \partial\Omega_-$$

在上述的方程中，$\beta(x,y,z)$ 是一个向量场，会随着位置的变化而变化，边界 $\partial\Omega_-$ 是入口边界，满足条件是 $\{p \in \partial\Omega : \beta(p)\cdot\mathbf{n} < 0\}$，其实就是入口边界，然后开始离散化，SUPG 格式是迎风流线稳定，具体做法就是试探函数取为 $v + \delta\beta\nabla v$,，对于边界上的测试函数是完全看不明白，基本上就是推导半天得到的结论，取 $\beta\cdot\mathbf{n}v$ 作为边界测试函数，书上这么说，我也这么写，最不想看懂的一集。

所以上述说完基本上整个离散就结束了，最不想看懂的一集，别管怎么来的，先拿来用用看好不好，两边同时乘以测试函数并进行积分，如下：

$$\int_\Omega (v+\delta\beta\nabla v)\cdot\beta\nabla u - \int_{\partial\Omega_-}\beta\cdot\mathbf{n}vu = \int_\Omega (v+\delta\beta\nabla v)f - \int_{\partial\Omega_-}\beta\cdot\mathbf{n}vg$$

离散完成后就基本上按照上述进行计算即可，其实如果去掉边界的相，单纯的看原始方程的 SUPG 离散格式，可以倒退到原始的强形式为 $\beta\nabla v + \delta\nabla\cdot[\beta\cdot(\beta\cdot\nabla u)] = 0$ ，基本上就是多加了个扩散项，但又不是太大的扩散项，具体大小和网格的尺寸相关，也就是 $\delta(h)$

对于 DG 方法来说，还是需要需要稳定项，但只是在内部边界上，离散起来更为简洁，但是 DG 对于第一类边界条件要么加到弱形式里面，要么加入惩罚，处理起来比较抽象。离散格式如下：

$$\begin{aligned}\int_\Omega v\beta\nabla u &= \int_{\partial\Omega} v\beta\cdot\mathbf{n}u - \int_\Omega u\beta\nabla v \\ &= \int_{\partial\Omega_-} v\beta\cdot\mathbf{n}g + \int_{\partial\Omega_+} v\beta\cdot\mathbf{n}u + \int_{\partial\Omega_{inner}}[v]\beta\cdot\mathbf{n}u^{upwind} - \int_\Omega u\beta\nabla v\end{aligned}$$

2 Trilinos 学习

2.1 Trilinos Panzer

Trilinos 是和 Petsc 类似的 HPC 计算基本库，相比 Petsc 来说，Trilinos 项目更大，代码更加现代化，特指 c++ 模板一堆，这个有好有坏，学习成本比较高，但是功能非常强大，用户层面比较高，可以专注于功能的实现，不需要考虑不同架构上的性能问题 (最新一代的矩阵向量 Tpetra 建立在 Kokkos 上) 。Petsc 使用 c 语言实现，核心功能非常明确矩阵 Mat，向量 Vec，线性求解器 Ksp，非线性求解器 Sens，时间积分求解 Ts，优化求解 Tao，个人建议最好配置的时候外部配置一下 SUNDIALS，提供一系列时间积分求解方法以及切线敏度和伴随敏度分析。

以下是从官网上扣下来的几个模块的介绍，计算效率比手写的高出一大截 ()

- ARKODE, a solver with one-step methods for stiff, nonstiff, mixed stiff-nonstiff, and multirate ODE systems.
- CVODE, a solver with Adams and BDF methods for stiff and nonstiff ODE systems.
- CVODES, an extension of CVODE with forward and adjoint sensitivity analysis capabilities for stiff and nonstiff ODE systems.
- IDA, a solver with BDF methods for DAE systems.
- IDAS, an extension of IDA with forward and adjoint sensitivity analysis capabilities for DAE systems.
- KINSOL, a solver for nonlinear algebraic systems.

Petsc 中 Ts 也提供许多时间积分求解器，如 BDF，θ 法，RK，IMEX 等，最近提供了伴随导数的计算，以及伪时间步进方法。

说了这么多 Petsc 相关的内容，Trilinos 里面有什么不同吗?，确实不同，上面所说的内容，Trilinos 里面都提供，甚至还有更多。Petsc 本身并不提供特征值计算，Slepc 作为 Petsc 的外延，提供基于子空间投影方法的特征值计算方案。以我的使用体验，低网格数下，Petsc 效率比 Trilinos 略高一点，但高网格和复杂方程下，Petsc 的编写成本会非常高，计算效率比 Trilinos 低不少。这里不是因为 Petsc 本身的计算效率不高，而是因为对于复杂 PDE 来说，计算 Jacobian 和求解线性系统非常耗时，手写代码效率一般。

其次还有一点，Trilinos 提供自动微分，在实现计算的同时，通过模板改变数值类型可以做到自动求解 Jacobian，只需要将弱形式编写完成，对应的 Jacobian 可以使用自动微分计算。最后，Petsc 的不提供有限元，有限体，有限差分等方法的实现，需要自己手动编写。上限自然是有的，但取决于个人的代码水平。如果你对 openmpi，openmp，simd 非常理解，能够避免并行计算的常见错误，提高计算效率，自然计算效率很高。但其实大伙都不怎么会，写出来的基本上也就是能跑的水平，至于性能不是重点，更关注方程和收敛性。这时 Trilinos 的优势就凸显出来。

2.1.1 Trilinos Tpetra MultiVector and CrsMatrix

这段时间一直没更新主要是在看 SUPG 和 DG，以及重新编译安装 Trilinos，之前安装的版本缺少一些内容，没有网格输入输出 (STK) 和自由度管理 (Panzer)。给我折磨坏了，最新 release 版本有问题，方程离散工具和自由度管理工具之间出现重复定义导致编译不通过，还得克隆下来 master 然后重新安装，目前应该是完全的安装好了。

可能会比较迷惑的点在与 Kokkos 和 Trilinos 为啥会混合在一起，实际上 Kokkos 是 Trilinos 的一个子项目，也是最核心的底层架构，主要负责矩阵向量表示，稀疏矩阵计算。Tpetra 就是以 Kokkos 为核心的线性代数核心包，因为网上的 Trilinos 介绍非常少，在此处简单的列举安装上的核心库的主要作用。

- Kokkos/KokkosKernels 核心底层
- Tpetra/Xpetra 线性代数表示/抽象接口
- Teuchos 提供参数文件/命令行参数，智能指针，MPI 接口
- Amesos2 提供外部的稀疏直接求解器，MUMPS,SuperLU,UMPACK
- Anasazi 特征值计算
- Belos 提供各种迭代方法，包括无矩阵接口
- Galeri 提供有限差分法离散
- ifpack2 提供一般的预处理，SOR，Jacobian 之类
- Intrepid2 提供旋度场，散度场，梯度场离散，基函数，正交规则
- ...

这里面有必要着重提及 Sacado 和 Muelu，后者是提供平滑聚类多重网格个几何多重网格的包，前者是用于实现自动微分功能 (尤其对于材料非线性，多相流，敏度分析这些东西)，你开始计算一个非线性材料的传热性能，辛辛苦苦推导出了对应的 Jacobian 方程，输入到对应的牛顿方法中，进行计算。你觉得这套方案一定能够适用其他问题，直到你遇到了多相流，参数敏度，你开始熬夜手搓，突然发现可以直接放弃大脑，拥抱现代 c++。

Sacado 提供正向直接微分和反向伴随微分，微分精度为机器精度，同时和 Tpetra 矩阵向量通过模板结合，可以直接在 cuda 上跑。什么你说不是手动推导的没有灵魂，确实没有灵魂，但你手动推导直接切线方程和伴随方程还没结束，这边可能就编写求解代码，边重载计算敏度了，折磨自己不如折磨机器 ()

如果你安装了 Sacado，以下是一个小测试案例

```
#include <Sacado.hpp>
#include <iostream>
using fad_double = Sacado::Fad::DFad<double>;
int main() {
  fad_double a,b,c;
```

```
  a = 1; b = 2;
  a.diff(0,2);  // Set a to be dof 0, in a 2-dof system.
  b.diff(1,2);  // Set b to be dof 1, in a 2-dof system.
  c = 2*a+cos(a*b);
  double *derivs = &c.fastAccessDx(0); // Access derivatives
  double a_value = 1, b_value = 2;
  double dc_da = 2 - sin(a_value * b_value) * b_value;
  double dc_db = - sin(a_value * b_value) * a_value;
  std::cout << std::setprecision(16) << "Sacado : dc/da = " << derivs[0] << ", dc/db
    =" << derivs[1] << std::endl;
  std::cout << std::setprecision(16)<< "Hands  : dc/da = " << dc_da << ", dc/db=" <<
     dc_db << std::endl;
}
```

其对应的输出结果如下：

$$Sacado: dc/da = 0.1814051463486366, dc/db = -0.9092974268256817$$

$$Hands: dc/da = 0.1814051463486366, dc/db = -0.9092974268256817$$

对于大型的离散问题来说，使用自动微分对整个矩阵进行计算似乎是不是非常的现实，但实际情况是，不会直接对整个稀疏矩阵进行计算，而是对每个单元刚度矩阵进行微分再进行组合，使用 c++ 模板重载实现在计算总体矩阵的时候同时计算 Jacobian 矩阵。对于参数敏度更是如此，放弃智力，拥抱现代 c++

以下代码是使用 Galeri 进行 Laplace 方程的有限差分离散，主要是需要搞明白矩阵向量的初始化。

```
#include "Galeri_XpetraMaps.hpp"
#include "Galeri_MatrixTraits.hpp"
#include "Galeri_XpetraMatrixTypes.hpp"
#include "Galeri_XpetraProblemFactory.hpp"
#include "Teuchos_DefaultComm.hpp"
#include "Teuchos_ParameterList.hpp"

#define GO long long
#define Scalar int
#define LO int
#define Node Tpetra::KokkosCompat::KokkosOpenMPWrapperNode

using namespace Galeri;
int main(int argc, char *argv[])
{
    using Teuchos::RCP;
    using Teuchos::rcp;
    typedef Tpetra::Map<LO, GO, Node> Tpetra_Map;
    typedef Tpetra::CrsMatrix<Scalar, LO, GO, Node> Tpetra_CrsMatrix;
    typedef Tpetra::MultiVector<Scalar, LO, GO, Node> Tpetra_MultiVector;
    typedef Teuchos::ScalarTraits<Scalar> ScalarTraits;
#ifdef HAVE_MPI
    MPI_Init(&argc, &argv);
```

```
#endif
    // Create comm
    RCP<const Teuchos::Comm<int>> comm = Teuchos::DefaultComm<int>::getComm();
    // Here we create the linear problem
    //
    //   Matrix * LHS = RHS
    //
    // with Matrix arising from a 5-point formula discretization.
    std::string mapType = "Cartesian2D";
    auto mapParameters = Teuchos::ParameterList("Tpetra::Map");
    // dimension of the problem is nx x ny
    mapParameters.set("nx", 10 * comm->getSize());
    mapParameters.set("ny", 10);
    // total number of processors is mx x my
    mapParameters.set("mx", comm->getSize());
    mapParameters.set("my", 1);
    mapParameters.print();
    auto out = Teuchos::getFancyOStream(Teuchos::rcpFromRef(std::cout));
    try
    {
        // Creation of the map
        auto map = RCP{Galeri::Xpetra::CreateMap<Scalar, GO, Tpetra_Map>(mapType,
    comm, mapParameters)};
        // Creation of linear problem
        auto problem = Galeri::Xpetra::BuildProblem<Scalar, LO, GO, Tpetra_Map,
    Tpetra_CrsMatrix, Tpetra_MultiVector>("Laplace2D", map, mapParameters);
        // Build Matrix and MultiVectors
        auto matrix = problem->BuildMatrix();
        auto LHS = rcp(new Tpetra_MultiVector(matrix->getDomainMap(), 1));
        auto RHS = rcp(new Tpetra_MultiVector(matrix->getRangeMap(), 1));
        auto ExactSolution = rcp(new Tpetra_MultiVector(matrix->getDomainMap(), 1));
        ExactSolution->randomize(0, 100);
        LHS->putScalar(ScalarTraits::zero());
        matrix->apply(*ExactSolution, *RHS);
        matrix->describe(*out, Teuchos::EVerbosityLevel::VERB_EXTREME);
        LHS->describe(*out, Teuchos::EVerbosityLevel::VERB_EXTREME);
        RHS->describe(*out, Teuchos::EVerbosityLevel::VERB_EXTREME);
        ExactSolution->describe(*out, Teuchos::EVerbosityLevel::VERB_EXTREME);
        // at this point any LinearSolver can be used which understands the Tpetra
    objects. For example: Amesos2 or Ifpack2
    }
    catch (Galeri::Exception &rhs)
    {
        if (comm->getRank() == 0)
        {
            cerr << "Caught exception: ";
            rhs.Print();
#ifdef HAVE_MPI
            MPI_Finalize();
```

```
70 #endif
71                 return (EXIT_FAILURE);
72             }
73         }
74 #ifdef HAVE_MPI
75         MPI_Finalize();
76 #endif
77         return (EXIT_SUCCESS);
78 }
```

2.1.2 Galeri, Belos && Ifpack2

感觉现在刚开始用这套代码，就是纯纯的缝合，先去对应的包里面找教程和案例，再把这些教程的内容复合在一起，得到需要的东西，刚开始先暂时不考虑组装的问题，使用 Geleri 有限差分构建需要的矩阵和向量，Belos 线性求解器，Ifpack2 预处理器。先简单实验一下.

```
// Ifpack2
#include <Ifpack2_Factory.hpp>
#include <Ifpack2_Preconditioner.hpp>

// Teuchos
#include <Teuchos_Assert.hpp>
#include <Teuchos_CommandLineProcessor.hpp>
#include <Teuchos_ParameterList.hpp>
#include <Teuchos_StandardCatchMacros.hpp>

// Tpetra
#include <Tpetra_Core.hpp>
#include <Tpetra_CrsMatrix.hpp>
#include <Tpetra_Map.hpp>
#include <Tpetra_MatrixIO.hpp>
#include <Tpetra_MultiVector.hpp>
#include <Tpetra_Operator.hpp>

// Belos
#include "BelosConfigDefs.hpp"
#include "BelosLinearProblem.hpp"
#include "BelosTFQMRSolMgr.hpp"
#include "BelosTpetraAdapter.hpp"

// Galeri
#include <Galeri_XpetraMaps.hpp>
#include <Galeri_XpetraMatrixTypes.hpp>
#include <Galeri_XpetraProblemFactory.hpp>

int main(int argc, char *argv[])
{
    // 浮点数类型
    using ST = typename Tpetra::MultiVector<>::scalar_type;
    // 局部自由度索引类型
    using LO = typename Tpetra::MultiVector<>::local_ordinal_type;
    // 全局自由度索引类型
    using GO = typename Tpetra::MultiVector<>::global_ordinal_type;
    // Kokkos 运行空间
    // 如果安装了CUDA,默认就在CUDA上跑
    // using NT = typename Tpetra::MultiVector<>::node_type;
    // 可以手动选择运行空间
    using NT = typename Tpetra::KokkosCompat::KokkosOpenMPWrapperNode;
    // 线性算子，矩阵的抽象封装
```

```
44      using OP = typename Tpetra::Operator<ST, LO, GO, NT>;
45      // 多重向量
46      using MV = typename Tpetra::MultiVector<ST, LO, GO, NT>;
47
48      // 矩阵稀疏模板
49      using tmap_t = Tpetra::Map<LO, GO, NT>;
50      // 稀疏矩阵
51      using tcrsmatrix_t = Tpetra::CrsMatrix<ST, LO, GO, NT>;
52
53      // 定义Belos统一的外部向量封装接口
54      using MVT = typename Belos::MultiVecTraits<ST, MV>;
55      // 定义Belos统一的外部矩阵封装接口
56      using OPT = typename Belos::OperatorTraits<ST, MV, OP>;
57
58      // 浮点数类型当前的具体类型
59      using MT = typename Teuchos::ScalarTraits<ST>::magnitudeType;
60      // 浮点数类型的抽象封装
61      using STM = typename Teuchos::ScalarTraits<MT>;
62
63      // 定义线性问题
64      using LinearProblem = typename Belos::LinearProblem<ST, MV, OP>;
65      // 定义预处理器
66      using Preconditioner = typename Ifpack2::Preconditioner<ST, LO, GO, NT>;
67      // 线性求解器
68      using TFQMRSolMgr = typename Belos::TFQMRSolMgr<ST, MV, OP>;
69
70      // 智能指针
71      using Teuchos::RCP;
72      using Teuchos::rcp;
73      // 参数文件
74      using Teuchos::ParameterList;
75
76      Tpetra::ScopeGuard MyScope(&argc, &argv);
77      {
78          RCP<const Teuchos::Comm<int>> comm = Tpetra::getDefaultComm();
79          // RCP<Teuchos::FancyOStream> Myout = Teuchos::getFancyOStream(Teuchos::
    rcpFromRef(std::cout));
80
81          // Flag 是否使用左预处理
82          bool leftPrec = true;
83          // Int 参数决定RHS的个数
84          int RhsNum = 1;
85          // 迭代容差
86          MT tol = STM::squareroot(STM::eps());
87
88          // 网格的点数
89          int nx = 100;
90
91          // 命令行参数控制
```

```
92          Teuchos::CommandLineProcessor cmdp(false, true);
93          cmdp.setOption("LeftPrec", "RightPrec", &leftPrec, "Preconditioner Type");
94          cmdp.setOption("RHSnum", &RhsNum, "Number Of RHS");
95          cmdp.setOption("nx", &nx, "Number Of Grid");
96          if (cmdp.parse(argc, argv) != Teuchos::CommandLineProcessor::
    PARSE_SUCCESSFUL)
97          {
98              return EXIT_FAILURE;
99          }
100
101         // 通过Galeri创建稀疏模板,并填充数据
102         ParameterList GaleriList;
103         GaleriList.set("n", nx * nx * nx);
104         GaleriList.set("nx", nx);
105         GaleriList.set("ny", nx);
106         GaleriList.set("nz", nx);
107         GaleriList.set("mx", (int)comm->getSize());
108         GaleriList.set("my", 1);
109         GaleriList.set("mz", 1);
110         // 稀疏模板
111         RCP<tmap_t> MatrixMap =
112             RCP{Galeri::Xpetra::CreateMap<LO, GO, tmap_t>("Cartesian3D", comm,
    GaleriList)};
113         // 稀疏矩阵类型
114         auto GaleriProblem =
115             Galeri::Xpetra::BuildProblem<ST, LO, GO, tmap_t, tcrsmatrix_t, MV>("
    Laplace3D", MatrixMap, GaleriList);
116         // 创建稀疏矩阵并计算
117         RCP<tcrsmatrix_t> MyMatrix = GaleriProblem->BuildMatrix();
118
119         // 创建右端项
120         RCP<MV> MyRHS = rcp(new MV(MatrixMap, RhsNum));
121         // 创建解向量
122         RCP<MV> MySolution = rcp(new MV(MatrixMap, RhsNum));
123         // 创建正确的解向量
124         RCP<MV> MyExSolution = rcp(new MV(MatrixMap, RhsNum));
125         // 将解向量随机化
126         MVT::MvRandom(*MyExSolution);
127         // RHS = A * EXSolution
128         OPT::Apply(*MyMatrix, *MyExSolution, *MyRHS);
129
130         // 设定Ifpack2 ILU预处理器
131         std::string precType = "RILUK";
132         RCP<Preconditioner> prec = Ifpack2::Factory::create<tcrsmatrix_t>(precType,
    MyMatrix);
133
134         TEUCHOS_ASSERT(prec != Teuchos::null);
135
136         // 预处理参数
```

```
137         int lFill = 2;
138         int overlap = 2;
139         ST absThresh = 0.0;
140         ST relThresh = 1.0;
141 
142         ParameterList precParams;
143         precParams.set("fact: iluk level-of-fill", lFill);
144         precParams.set("fact: iluk level-of-overlap", overlap);
145         precParams.set("fact: absolute threshold", absThresh);
146         precParams.set("fact: relative threshold", relThresh);
147         prec->setParameters(precParams);
148         // 初始化预处理器并计算具体值
149         prec->initialize();
150         prec->compute();
151 
152         int MaxIters = MyRHS->getGlobalLength() - 1;
153         // 设定线性求解器
154         ParameterList belosList;
155         belosList.set("Maximum Iterations", MaxIters);
156         belosList.set("Convergence Tolerance", tol);
157         belosList.set("Explicit Residual Test", true);
158         belosList.set("Verbosity",
159                       Belos::Errors +
160                       Belos::Warnings +
161                       Belos::TimingDetails +
162                       Belos::StatusTestDetails);
163 
164         // 定义线性问题
165         RCP<LinearProblem> MyProblem = rcp(new LinearProblem(MyMatrix,MySolution,
    MyRHS));
166         if (leftPrec)
167         {
168             MyProblem->setLeftPrec(prec);
169         }
170         else
171         {
172             MyProblem->setRightPrec(prec);
173         }
174 
175         bool set = MyProblem->setProblem();
176         if (set == false)
177         {
178             std::cout << std::endl
179                       << "ERROR:  Belos::LinearProblem failed to set up
    correctly!" << std::endl;
180             return EXIT_FAILURE;
181         }
182 
183         // 创建线性求解器
```

```
184         RCP<TFQMRSolMgr> MySolver = rcp(new TFQMRSolMgr(MyProblem,rcp(&belosList,
    false)));
185 
186         std::cout << "Solving..." << std::endl;
187         Belos::ReturnType ret = MySolver->solve();
188         std::cout << "Solve end" << std::endl;
189 
190         // 将求解的解和真解对比
191         bool badRes = false;
192         std::vector<ST> actualResids(RhsNum);
193         std::vector<ST> rhsNorm(RhsNum);
194 
195         // 残差向量
196         MV resids(MatrixMap, RhsNum);
197         // r = A*x - b
198         OPT::Apply(*MyMatrix, *MySolution, resids);
199         MVT::MvAddMv(-1.0, resids, 1.0, *MyRHS, resids);
200         //
201         MVT::MvNorm(resids, actualResids,Belos::TwoNorm);
202         MVT::MvNorm(*MyRHS, rhsNorm);
203         //
204         std::cout << "---------- Actual Residuals (normalized) ----------" << std::
    endl
205                   << std::endl;
206         for (int i = 0; i < RhsNum; i++)
207         {
208             ST actRes = actualResids[i] / rhsNorm[i];
209             std::cout << "Problem " << i << " : \t" << actRes << std::endl;
210             if (actRes > tol)
211                 badRes = true;
212         }
213         //
214         if (ret != Belos::Converged || badRes)
215         {
216             std::cout << std::endl
217                       << "ERROR:  Belos did not converge!" << std::endl;
218         }
219         else
220         {
221             std::cout << std::endl
222                       << "SUCCESS:  Belos converged!" << std::endl;
223         }
224     }
225     return EXIT_SUCCESS;
226 }
```

MPI/CUDA Run time	Serial	MPI NP 2	MPI NP 4
CUDA	4.78s	1.89s	1.43s
OpenMP	2.5s	1.94s	1.72s

表 1: MPI/Serial CUDA/OpenMP 运行时间

默认情况下 $nx = 100$ 且是三维问题，网格数量就是 nx^3，自由度数量也是差不多的量级。预处理器使用 ILU 预处理。以下是具体的细节时间输出

Timer Name	MinOverProcs	MeanOverProcs	MaxOverProcs
Op*x	0.1789	0.2399	0.2789
Prec*x	0.9508	0.9579	0.9669
TFQMRSolMgr	1.432	1.435	1.438
fillComplet	0.133	0.1361	0.1372
generation	0.08773	0.089	0.09294

表 2: 具体运行时间

时间消耗最长的是预处理器的计算和预处理器和向量的计算，在 CUDA 上计算的优势就是可以大幅的加速矩阵向量乘法的计算效率，可以得到非常明显的加速效果，CPU 上由于不同节点之间的相互通信以及带宽的限制，加速效果具有明显的边际效应。在 800w 网格下的计算时间为

$$mpirun\ -np\ 4\ \&\&\ CUDA:\ 33.26s$$

$$mpirun\ -np\ 4\ \&\&\ OpenMP:\ 25.56s$$

此处 OpenMP 的计算速度快于 CUDA 的原因非常简单，(CUDA 内存太小了，任务管理器显示内存使用满了)

2.1.3 Panzer Laplace Example

Panzer 是 Trilinos 的复合物理场的统一管理包，可以耦合多个物理场，但这个使用起来比较抽象，这个模块设计的时候是考虑到了线性和非线性的问题，将一系列物理场进行耦合。如果还记得之前的自动微分包 Sacado, 在进行刚度矩阵计算的时候，会通过重载基类进行计算，计算的数值类型包括了三种 $double, Sacado :: Fad :: DFad < double >$，正常计算有限元问题的时候就使用 $double$，如果需要对问题进行非线性求解，就将计算类型的模板给定为 $Sacado :: Fad :: DFad < double >$，两者使用的是同一套模板函数，这个设计也就是之前提到的计算单元刚度矩阵的同时计算 $Jacobian$

Panzer 包括 $DOFManager, ConnManager, ClosureModelManager, FieldManager$，这四个的作用包括网格管理，自由度管理，物理场管理和求解模型管理。因为代码有点多，没法一次性放到 PDF 上，所以只挑选其中一部分代码进行解释网格和自由度先不看，以 Laplace 方程为例理解物理场是如何进行计算和离散的。以下说明以 Laplace 方程和时变 Laplace 方程举例。

```
1 struct Residual { typedef RealType ScalarT; };
```

$$\nabla \cdot (c(u) \cdot u) = Q$$

$$R = \nabla u \cdot c(u) \cdot \nabla v - Q$$

上述问题为一个非线性的 Laplace 问题，其中 $c(u)$ 表示和自变量 u 相关的分部场，R 为离散化的求解方程。

```
1 struct Jacobian { typedef FadType  ScalarT; };
```

$$\begin{aligned} J &= \frac{\partial R}{\partial u} \\ &= \lim_{\epsilon \to 0} \frac{1}{\epsilon}(R(u + \epsilon\delta u) - R(u)) \\ &= \lim_{\epsilon \to 0} \frac{1}{\epsilon}(\nabla(u + \epsilon\delta u) \cdot c(u + \epsilon\delta u) \cdot \nabla v - \nabla u \cdot c(u) \cdot \nabla v) \\ &= \nabla\delta u \cdot c(u) \cdot \nabla v + \nabla u \cdot c(u)_u \cdot \delta u \cdot \nabla v \end{aligned}$$

对于这个非线性问题来说，进行空间离散后会形成如 $A(u)u = Q$ 的形式，为了求解这个非线性问题，一般使用牛顿方法进行计算 $R(u) + J\delta u = 0$，这样迭代到收敛，类似的方法还有许多的变种，但一般这个就够用了。

```
1 struct Tangent  { typedef FadType  ScalarT; };
```

Tangent 敏度分析，以时变问题为例，p 为参数

$$\begin{aligned} \frac{\partial u}{\partial t} &= \nabla u \cdot c(p) \cdot \nabla v + Q \\ \frac{u^n - u^{n-1}}{\Delta t} &= \nabla u^{n-1} \cdot c(p) \cdot \nabla v + Q \\ (\frac{\partial u^n}{\partial p} - \frac{\partial u^{n-1}}{\partial p}) &= \Delta t \nabla u^{n-1} \cdot \frac{\partial c}{\partial p} \cdot \nabla v + \Delta t \nabla v \cdot c \cdot \nabla v \frac{\partial u^{n-1}}{\partial p} \end{aligned}$$

以简单的差分格式举例，通过给定 0 时刻下的敏度初值, 带入上述方程进行迭代可以计算得到最终时刻下的敏度，敏度扩散顺序和时间迭代顺序一致, 一般被称为 Tangent。和这个方程相对应的是伴随模型 Adjoint，其敏度扩散顺序和时间迭代顺序相反，这两中方案只是求导的顺序不

同，但是求解的结果是完全一致的，Tangent 适用于参数个数比较少，优化目标比较多的情况，Adjoint 适用于参数个数比较多，优化目标比较少的情况。

EvalT 为实际计算的类型，主要包括两种选择正常计算使用的浮点数类型 double,float 非线性问题使用的自动微分类型 FadType 这个类的初始化方法需要传入参数文件, 积分阶次, 网格数据和全局的自由度数据,*build_transient_support* 表示是否计算瞬态问题

```
template <typename EvalT>
Example::PoissonEquationSet<EvalT>::
    PoissonEquationSet(const Teuchos::RCP<Teuchos::ParameterList> &params,
                       const int &default_integration_order,
                       const panzer::CellData &cell_data,
                       const Teuchos::RCP<panzer::GlobalData> &global_data,
                       const bool build_transient_support)
: panzer::EquationSet_DefaultImpl<EvalT>(params,
                                         default_integration_order,
                                         cell_data,
                                         global_data,
                                         build_transient_support)
{
...
}
```

这个部分设计的比较抽象, 首先回忆一下正常的 Laplace 方程弱形式离散, 正如上面写的, 单元刚度矩阵为 $\nabla v \cdot c \cdot \nabla v$ 的双线性形式, 故此选定的基函数为 *HGrad*, 在添加完温度场后, 再添加对应的梯度场.

```
// 创建一个新的参数列表,给出默认的参数值,和传入的参数列表进行匹配
// 如果传入没有这个参数则会加入进去
Teuchos::ParameterList valid_parameters;
this->setDefaultValidParameters(valid_parameters);

valid_parameters.set("Model ID", "", "Closure model id associated with this equaiton
    set");
valid_parameters.set("Basis Type", "HGrad", "Type of Basis to use");
valid_parameters.set("Basis Order", 1, "Order of the basis");
valid_parameters.set("Integration Order", -1, "Order of the integration rule");

params->validateParametersAndSetDefaults(valid_parameters);
// 获取参数
std::string basis_type = params->get<std::string>("Basis Type");
int basis_order = params->get<int>("Basis Order");
int integration_order = params->get<int>("Integration Order");
std::string model_id = params->get<std::string>("Model ID");

// 这里声明创建一个TEMPERATURE的物理场
this->addDOF("TEMPERATURE", basis_type, basis_order, integration_order);
// 因为基函数类型为HGrad,这里需要温度场的梯度
this->addDOFGrad("TEMPERATURE");
// 如果计算瞬态问题,需要DTEMPERATURE_DT
```

```
if (this->buildTransientSupport())
  this->addDOFTimeDerivative("TEMPERATURE");
// 将边界条件,物理区域组合起来
this->addClosureModel(model_id);

this->setupDOFs();
```

我们再来看添加完成自由度之后, 如何实现离散化.Panzer 的封装程度非常高, 正常理解的组装过程可能是循环所有的网格, 在每个网格上根据基函数确定梯度值, 根据积分规则和阶数确定正交点, 循环所有正交点, 计算双线性形式, 再根据实际网格尺寸计算等参变换的 $Jacobian$ 矩阵和对应的行列式, 最后根据全局自由度索引提交到总刚度矩阵上.

这个过程是一般的有限元计算流程, 但如何让这个过程更快? 可以看到在参考单元上计算 $\nabla \cdot v$ 的时候, 所有单元 (如果不考虑混合网格) 是完全一致的, 可以把这部分分离出来, 所有网格只需要计算一次.Panzer 的离散过程就只需要指定基函数乘以那一部分即可, 其余过程可以自动进行.

```
// 积分阶次
RCP<IntegrationRule> ir = this->getIntRuleForDOF("TEMPERATURE");
// 基函数类型
RCP<BasisIRLayout> basis = this->getBasisIRLayoutForDOF("TEMPERATURE");
// 定义瞬态项算子
if (this->buildTransientSupport())
{
  // 残差R的名称,需要计算的v \dot Scalar中的Scalar对应的项
  string resName("RESIDUAL_TEMPERATURE"),
         valName("DXDT_TEMPERATURE");
  double multiplier(1);
  // 此处不需要给出v的具体形式,只要给出
  // v \dot Scalar中的Scalar对应的项即可
  // 相当于v \dot DXDT_TEMPERATURE
  RCP<Evaluator<Traits>> op =
      rcp(new Integrator_BasisTimesScalar<EvalT, Traits>
                  (EvaluatorStyle::CONTRIBUTES,
                   resName, valName, *basis, *ir, multiplier));
  // 创建这个算子
  this->template registerEvaluator<EvalT>(fm, op);
}

// 定义扩散项算子 \int \nabla T \cdot \nabla v
{
  double thermal_conductivity = 1.0;

  ParameterList p("Diffusion Residual");
  p.set("Residual Name", "RESIDUAL_TEMPERATURE");
  p.set("Flux Name", "GRAD_TEMPERATURE");
  p.set("Basis", basis);
  p.set("IR", ir);
  p.set("Multiplier", thermal_conductivity);
```

```
33   // \nabla v \cdot Vec,Vec给定为GRAD_TEMPERATURE
34   RCP<Evaluator<Traits>> op =
35         rcp(new Integrator_GradBasisDotVector<EvalT, Traits>(p));
36
37   this->template registerEvaluator<EvalT>(fm, op);
38 }
39
40 // 源项
41 {
42   string resName("RESIDUAL_TEMPERATURE"),
43          valName("SOURCE_TEMPERATURE");
44   double multiplier(-1);
45   // v \cdot Scalar, Scalar给定为Q
46   RCP<Evaluator<Traits>> op =
47       rcp(new Integrator_BasisTimesScalar<EvalT, Traits>
48                   (EvaluatorStyle::CONTRIBUTES,
49                   resName, valName, *basis, *ir, multiplier));
50   this->template registerEvaluator<EvalT>(fm, op);
51 }
```

2.1.4 Convection Equation Example