Discretization and Assembly using Intrepid2 and Tpetra

M. Perego

Acknowledgments:


European Trilinos User Group Meeting, ETH, Zurich, Switzerland, June 11, 2019
Outline

• Brief intro to Trilinos Discretization Tools

• Essential Trilinos tools for assembling high-order finite elements in your application
  • Intrepid2: basis functions, quadrature rules, orientations, projections
  • DofManager: mesh topology and global numbering of DoFs
  • Tpetra: global assembly tools
  • Poisson Example

• Other Discretization Tools:
  • Sacado: automatic differentiation
  • Phalanx: DAG-based expression evaluation
  • Panzer: FE assembly and solution of nonlinear multi-physics problems

• Current developments
Trilinos Discretization Tools Overview

Local Node
- **Intrepid2**: FE Basis Library
- **Shards**: Cell/Element Topology
- **Phalanx**: DAG Assembly: flexibility/complexity
- **Sacado**: Automatic differentiation scalar types
- **Kokkos**: Performance portability

MPI Related
- **Panzer** (Multiphysics Assembly and Utilities)
  - **DoF Manager**: Global Indexing for mixed bases, mixed equations (CG and DG)
    - Connection Manager: Mesh DB abstraction
  - **Disc-FE**: Multiphysics assembly, Mixed Eq Sets, Mixed Bases, BCs, Compatible discretizations, Projections, Model Evaluators (NOX/Tempus/LOCA/ROL)
    - **Workset Builder**: Mesh over-decomposition for AMT
    - **Linear Object Factory**: Epetra/Tpetra/Thyra
- **Tpetra**: distributed linear algebra objects (vectors/matrices), hybrid parallelism
- **Thyra**: mathematical definition of vectors, vector spaces and linear operators
Trilinos Discretization Product Overview

Local Node
- **Intrepid2**: FE Basis Library
- **Shards**: Cell/Element Topology
- **Phalanx**: DAG Assembly: flexibility/complexity
- **Sacado**: Automatic differentiation scalar types
- **Kokkos**: Performance portability

MPI Related
- **Panzer** (Multiphysics Assembly and Utilities)
  - **DoF Manager**: Global Indexing for mixed bases, mixed equations (CG and DG)
    - Connection Manager: Mesh DB abstraction
  - **Disc-FE**: Multiphysics assembly, Mixed Eq Sets, Mixed Bases, BCs, Compatible discretizations, Projections, ModelEvaluators (NOX/Tempus/LOCA/ROL)
    - Workset Builder: Mesh over-decomposition for AMT
    - Linear Object Factory: Epetra/Tpetra/Thyra
- **Tpetra**: distributed linear algebra objects (vectors/matrices), hybrid parallelism
- **Thyra**: mathematical definition of vectors, vector spaces and linear operators
Intrepid2 package

Intrepid (INteroperable Tools for Rapid dEveloPment of compatIble Discretizations)

Trilinos package for advanced discretizations of Partial Differential Equations (PDEs)

Provides compatible high-order discretizations for finite elements,
Supports hybrid discretizations: FEM, FV and FD

Intrepid2

Refactoring of Intrepid for performance portability, using Kokkos.
Early developers (stage 1): I. Demeshko, A. Delora

Current developers (stage 2): K. Kim, M. Perego, N. Roberts, N. Ellingwood

Stage 1: fully compatibility with Intrepid interface, adoption of Kokkos

Stage 2: no backward compatibility but same mathematical/logical interface.
Use of Kokkos dynamic rank views.
Tools for Local Assembly of Compatible finite elements

De Rham complex

\[ \mathbb{R} \rightarrow \text{HGRAD} \quad \text{grad} \quad \text{HGRAD} \rightarrow \text{HCURL} \quad \text{curl} \quad \text{HCURL} \rightarrow \text{HDIV} \quad \text{div} \quad \text{HDIV} \rightarrow \text{HVOl} \rightarrow 0. \]

\[ \Phi_G^* : H(\text{grad}, \widehat{\kappa}) \rightarrow H(\text{grad}, \kappa) \]
\[ \Phi_C^* : H(\text{curl}, \widehat{\kappa}) \rightarrow H(\text{curl}, \kappa) \]
\[ \Phi_D^* : H(\text{div}, \widehat{\kappa}) \rightarrow H(\text{div}, \kappa) \]
\[ \Phi_V^* : H(\text{vol}, \widehat{\kappa}) \rightarrow H(\text{vol}, \kappa) \]

\[ \Phi_G^*(\widehat{u}) = \widehat{u} \circ F\kappa^{-1} \]
\[ \Phi_C^*(\widehat{u}) = ((DF\kappa)^{-T} \cdot \widehat{u}) \circ F\kappa^{-1} \]
\[ \Phi_D^*(\widehat{u}) = (J\kappa^{-1}DF\kappa \cdot \widehat{u}) \circ F\kappa^{-1} \]
\[ \Phi_V^*(\widehat{u}) = (J\kappa^{-1}\widehat{u}) \circ F\kappa^{-1} \]

\{ HGRAD_transform_VALUE() \}
\{ HGRAD_transform_GRAD(), HCUrL_transform_VALUE() \}
\{ HCUrL_transform_CURL(), HDIV_transform_VALUE() \}
\{ HDIV_transform_DIV(), HVOl_transform_VALUE() \}

\[ F\kappa : \widehat{\kappa} \rightarrow \kappa \]

Courtesy: P. Bochev
Choice of multidimensional array

Intrepid2 work with multi-dimensional arrays: e.g.,

```c
double basisVals[F][P];  // C - # of cells
double basisGradVals[F][P][D];  // N - # of nodes per cell
double jacMat[C][P][D][D];  // D - spatial dimension (2 or 3)
```

Operations often done on batches of cells

It is natural to declare these array as `Kokkos::view`

However, in some cases the rank of the arrays might not be known at compile time. Solution: `Kokkos::DynRankView`

<table>
<thead>
<tr>
<th>BASIS</th>
<th>INPUT</th>
<th>VALUE</th>
<th>GRAD</th>
<th>CURL</th>
<th>DIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>HGRAD</td>
<td>(P, D)</td>
<td>(F, P)</td>
<td>(F, P, D)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H CURL</td>
<td>(P, D)</td>
<td>(F, P, D)</td>
<td></td>
<td>(F, P, D)</td>
<td></td>
</tr>
<tr>
<td>HDIV</td>
<td>(P, D)</td>
<td>(F, P, D)</td>
<td></td>
<td></td>
<td>(F, P)</td>
</tr>
</tbody>
</table>

Rank of differential operators evaluated at quadrature points
Properties of Kokkos::DynRankView

// 1. Construction of a dynamic rank view
//     - rank is deduced from the number of dimensions arguments
Kokkos::DynRanView<value_type, exec_space> a("a", 3, 9);

// 2. Same subview syntax
auto part = Kokkos::subview(a, 0, Kokkos::ALL());  // part = a(1, :);

// 3 Inter-operability with Kokkos static view
Kokkos::View<value_type**, exec_space> b("b", 3, 9);

b = a;  // okay
a = b;  // error, a static view cannot be converted to DynRankView

Kokkos::deep_copy(b, a);  //okay
Kokkos::deep_copy(a, b);  //okay
Advantages of using Kokkos::DynRankView

/* This functor computes the following three contractions
outputFields(cl, lf, rf) = leftFields(cl, lf, qp) * rightFields(cl, rf, qp);
outputFields(cl, lf, rf) = leftFields(cl, lf, qp, i) * rightFields(cl, rf, qp, i);
outputFields(cl, lf, rf) = leftFields(cl, lf, qp, i, j) * rightFields(cl, rf, qp, i, j);

Using a view, one would need to replicate the code for the scalar, vector and matrix cases. */

KOKKOS_INLINE_FUNCTION
void operator()(ordinal_type iter) const {
  ordinal_type cl, lf, rf;
  unrollIndex(cl, lf, rf, iter, outputFields.dimesion(0), outputFields.dimesion(1), outputFields.dimesion(2));

  // member variables outputFields, leftFields, rightFields
  auto result = Kokkos::subview( outputFields, cl, lf, rf );

  // extra ranks ignored when computing subviews (only w/ DynRankView)
  const auto left = Kokkos::subview( leftFields, cl, lf, Kokkos::ALL(), Kokkos::ALL(), Kokkos::ALL() );
  const auto right = Kokkos::subview( rightFields, cl, rf, Kokkos::ALL(), Kokkos::ALL(), Kokkos::ALL() );

  const ordinal_type npts = left.dimension(0);
  const ordinal_type iend = left.dimension(1);
  const ordinal_type jend = left.dimension(2);

  value_type tmp(0);
  for (ordinal_type qp = 0; qp < npts; ++qp)
    for (ordinal_type i = 0; i < iend; ++i)
      for (ordinal_type j = 0; j < jend; ++j)
        tmp += left(qp, i, j)*right(qp, i, j);
  result() = result() + tmp;
}
Assembly of element stiffness matrix

\[ A_{ij} = \int_{\kappa} \nabla v_i \cdot \nabla v_j \, d\mathbf{x} \]
\[ \int_{\hat{\kappa}} ((DF_{\kappa})^T \nabla \hat{\mathbf{v}}_i) \cdot ((DF_{\kappa})^T \nabla \hat{\mathbf{v}}_j) \, J d\hat{\mathbf{x}} \]
\[ \sum_q ((DF_{\kappa})^T \nabla \hat{\mathbf{v}}_{i,q}) \cdot ((DF_{\kappa})^T \nabla \hat{\mathbf{v}}_{j,q}) \, J_q w_q \]

Intrepid2 APIs
- Cubature::getCubature
- Basis::getValue
- Orientation::modifyBasis
- CellTools: setJacobian, setJacobianDet
- FunctionSpaceTool::applyTransform, computeCellMeasure
- FunctionSpaceTools::integrate
Assembly of element stiffness matrices

\[
A_{ij} = \int_{\kappa} \nabla v_i \cdot \nabla v_j \, dx \\
= \int_{\tilde{\kappa}} \left( (DF_\kappa)^{-T} \hat{\nabla} \hat{v}_i \right) \cdot \left( (DF_\kappa)^{-T} \hat{\nabla} \hat{v}_j \right) J d\tilde{x} \\
= \sum_q \left( (DF_\kappa)_q^{-T} \nabla \hat{v}_{i,q} \right) \cdot \left( (DF_\kappa)_q^{-T} \nabla \hat{v}_{j,q} \right) J_q w_q
\]

// Dimensions:
// C - # of cells
// N - # of nodes per cell
// D - spatial dimension (2 or 3)
// F - # of field DOFs
// P - # of integration points
Kokkos::DynRankView

cubPoints(P, D),
cubWeights(P),
refGradVals(F, P, D),
orientedGradVals(F, P, D)

// cell geometry
nodes(C, N, D),
jacMat(C, P, D, D),
jacInv(C, P, D, D),
jacDet(C, P),

// elements on physical coordinates
physGradVals(C, F, P, D),
elemStiff(C, F, F);
Assembly of element stiffness matrices

\[ A_{ij} = \int_{\kappa} \nabla v_i \cdot \nabla v_j \, dx \]

\[ \int_{\tilde{\kappa}} \left[ (DF_{\kappa})^{-T} \nabla \tilde{v}_i \right] \cdot \left[ (DF_{\kappa})^{-T} \nabla \tilde{v}_j \right] J d\tilde{x} \]

\[ \sum_q \left[ (DF_{\kappa})^{-T} \nabla \tilde{v}_{i,q} \right] \cdot \left[ (DF_{\kappa})^{-T} \nabla \tilde{v}_{j,q} \right] J_q w_q \]
In high-order finite elements DOFs of adjacent elements may not be ordered in the same way on the shared face. We need to enforce the matching of DOFs.

Solution: locally map the DOFs of a face or edge according to the \textit{global numbering} and transform the corresponding basis accordingly.

\[
\phi_i^E = \sum_j A_{i,j}^E \psi_j
\]

\[
\sum_j A_{i,j}^E \psi_j (E(\eta(\xi_k))) \cdot \mathbf{t}_E = \varphi_i(\xi_k) \cdot \hat{\mathbf{t}}
\]
Orientation Tools
(Hcurl case, 3d)

Possible orientations for triangles

\[ \sum_j A_{ij}^T \psi_j^T (T(\eta(\xi_k))) \cdot t_{T,l} = \varphi_i(\xi_k) \cdot \hat{t}_l \]

We store six matrices A, one for each orientation.
Orientation Tools
demonstration for HGRAD and HCURL problems

Problems implemented using Trilinos package Panzer.

Results show theoretical order of convergence.

Courtesy: K. Kim, E. Cyr
Projection Tools

Projection-based interpolation* into HGrad, HCurl, HDiv and HVol spaces:
- Optimal accuracy in $H^1$, $H^{\text{curl}}$, $H^{\text{div}}$, $L^2$ spaces.
- Global conformity, preserves commutativity of De Rahm diagram (compatible discretization)
- Local, computational-intensive operations, can take full advantage of modern architectures (up to 19 constrained optimizations per cell)

Projections are needed for
- accurately prescribing non trivial initial and boundary conditions
- avoid use of global L2 projection (requires global solve, does not preserve De Rahm properties)

Assembly of local stiffness matrices

NVIDIA Tesla P100
Chip: 56 SMs, 32 FP64 CUDA Cores/SM
Peak performance: 4.7 TFLOPs in double precision, 732+ GB/s (HBM)

<table>
<thead>
<tr>
<th>Workset size</th>
<th>Speed-up per entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>20480</td>
<td>7.8x</td>
</tr>
<tr>
<td>5120</td>
<td>4.5x</td>
</tr>
<tr>
<td>2560</td>
<td>1.8x</td>
</tr>
<tr>
<td>1280</td>
<td>1.1x</td>
</tr>
</tbody>
</table>

Courtesy: N. Roberts
Assembly, from local to global

We need two ingredients:

1. **Global numbering of mesh vertices** (required for basis orientation)

2. **Global numbering of DoFs**

1. is easy, all mesh generators provides that.

2. is usually not available, except for low-order nodal finite elements.

Trilinos provides such capability in the **Panzer DoF Manager** sub-package
Panzer Degrees of Freedom Manager package

- ConnManager: abstraction for mesh topology connectivity (create it from a given mesh. Need to provide (global) vertices, edges, faces, cells and their connectivities)
- FieldPattern: pattern of field (i.e. what local DoFs are stored on vertices/edges/faces/cells for a specific field)
- DofManager: Computes global DoFs numbering and local-to-global maps for each field

- Handles multiple fields and physics blocks (w/ different fields on different blocks)
- Handles Continuous and Discontinuous Finite Elements
- Handles high-order, nodal and nonnodal finite elements

Fluid:
- Velocity (RT0, DoFs on edges)
- Pressure (Q0, DoFs on cells)
- Temperature (Q1, CG, DoFs on vertices)

Solid:
- Temperature (Q1, CG, DoFs on vertices)

<table>
<thead>
<tr>
<th>DoFs</th>
<th>fluid cell</th>
<th>solid cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Velocity</td>
<td>6,7,8,9</td>
<td></td>
</tr>
<tr>
<td>Pressure</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Temperature</td>
<td>0,1,3,2</td>
<td>1,4,2,5</td>
</tr>
</tbody>
</table>
Panzer Degrees of Freedom Manager package

- ConnManager: abstraction for mesh topology connectivity (create it from a given mesh. Need to provide (global) vertices, edges, faces, cells and their connectivities)
- FieldPattern: pattern of field (i.e. what local DoFs are stored on vertices/edges/faces/cells for a specific field)
- DofManager: Computes global DoFs numbering and local-to-global maps for each field
- Handles multiple fields and physics blocks (w/ different fields on different blocks)
- Handles Continuous and Discontinuous Finite Elements
- Handles high-order, nodal and nonnodal finite elements

Fluid:
- Velocity (Q2, CG, DoFs on vertices, edges, cells)
- Pressure (Q1, CG, DoFs on vertices)
- Temperature (Q1, CG, DoFs on vertices)

Solid:
- Temperature (Q1, CG, DoFs on vertices)

<table>
<thead>
<tr>
<th>DoFs</th>
<th>fluid cell</th>
<th>solid cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Velocity-x</td>
<td>0,4,12,8,18,20,22,24,26</td>
<td></td>
</tr>
<tr>
<td>Velocity-y</td>
<td>1,5,13,9,19,21,23,25,27</td>
<td></td>
</tr>
<tr>
<td>Pressure</td>
<td>2,6,14,10</td>
<td>7,16,11,17</td>
</tr>
<tr>
<td>Temperature</td>
<td>3,7,15,11</td>
<td>7,16,11,17</td>
</tr>
</tbody>
</table>
Panzer Degrees of Freedom Manager package

- ConnManager: abstraction for mesh topology connectivity (create it from a given mesh. Need to provide (global) vertices, edges, faces, cells and their connectivities)
- FieldPattern: pattern of field (i.e. what local DoFs are stored on vertices/edges/faces/cells for a specific field)
- DofManager: Computes global DoFs numbering and local-to-global maps for each field

- Handles multiple fields and physics blocks (w/ different fields on different blocks)
- Handles Continuous and Discontinuous Finite Elements
- Handles high-order, nodal and nonnodal finite elements

code snippet

```cpp
Panzer::DOFManager<int,int> dofManager = ...; // Build DOF manager from mesh topology

// Build Intrepid basis objects for Q2 (velocity) and Q1 (pressure and temperature)
RCP<Intrepid::Basis<double,FieldContainer> > q1_basis = rcp(new Intrepid::Basis_HGRAD_QUAD_C1_FEM);
RCP<Intrepid::Basis<double,FieldContainer> > q2_basis = rcp(new Intrepid::Basis_HGRAD_QUAD_C2_FEM);

// Build field patterns, these define the continuity requirements of the basis
RCP<const Panzer::IntrepidFieldPattern> q1_pattern = rcp(new Panzer::IntrepidFieldPattern(q1_basis));
RCP<const Panzer::IntrepidFieldPattern> q2_pattern = rcp(new Panzer::IntrepidFieldPattern(q2_basis));

// register fluid block fields
doFManager.addField("Fluid", "U", q2_pattern); // Velocity fields use Q2 basis
(doFManager.addField("Fluid", "V", q2_pattern);
doFManager.addField("Fluid", "Pres", q1_pattern); // Pressure and Temperature use Q1 basis
(doFManager.addField("Fluid", "Temp", q1_pattern); // register solid block fields
(doFManager.addField("Solid", "Temp", q1_pattern);

// Build global unknowns from specified fields (and patterns)
doFManager.buildGlobalUnknowns();
```
Poisson Assembly Example

Strong form:
\[
\begin{cases}
-\Delta u + u = f & \text{in } \Omega \\
\nabla u \cdot \mathbf{n} = 0 & \text{on } \partial\Omega
\end{cases}
\]

Weak form:
\[
\int_{\Omega} \nabla u \cdot \nabla \phi + \int_{\Omega} u \phi = \int_{\Omega} f \phi
\]

Discretization:
\[
A \mathbf{x} = \mathbf{b} \quad A_{ij} := \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j + \phi_i \phi_j, \quad b_i := \int_{\Omega} f \phi_i
\]

This example is going to become part of the Discretization tutorials.
// Get cell topology for base hexahedron
**typedef** shards::CellTopology CellTopology;
CellTopology hexa(shards::getCellTopologyData<shards::Hexahedron<8> >());

// build the topology
**auto** connManager = Teuchos::rcp(**new** panzer::unit_test::CartesianConnManager);
connManager->initialize(comm,
    global_ordinal_t(nx*px),
    global_ordinal_t(ny*py),
    global_ordinal_t(nz*pz),
    px,py,pz,bx,by,bz);

// **************** COMPUTE GLOBAL IDs OF VERTICES AND DOFs ************
// build the dof manager, and associate with the topology
**auto** dofManager = Teuchos::rcp(**new** panzer::DOFManager<local_ordinal_t, global_ordinal_t>);
dofManager->setConnManager(connManager,*comm.getRawMpiComm());

// add solution field to the element block
Teuchos::RCP< Intrepid2::Basis<DeviceSpaceType, scalar_t,scalar_t> > basis =
    Teuchos::rcp(**new** Basis_HGRAD_HEX_Cn_FEM<DeviceSpaceType,scalar_t,scalar_t>(degree));
**ordinal_type** basisCardinality = basis->getCardinality();
Teuchos::RCP<panzer::Intrepid2FieldPattern> fePattern =
    Teuchos::rcp(**new** panzer::Intrepid2FieldPattern(basis));
dofManager->addField("block-0_0_0",fePattern);

// compute global numbering
dofManager->buildGlobalUnknowns();
// Get coordinates of physical vertices
DynRankView physVertexes("", numOwnedElems, numNodesPerElem, dim);
{
    auto physVertexesHost = Kokkos::create_mirror_view(physVertexes);
    DynRankView refVertices("", numNodesPerElem, dim);
    Basis_HGRAD_HEX_C1_FEM<DeviceSpaceType, scalar_t, scalar_t> hexaLinearBasis;
    hexaLinearBasis.getDofCoords(refVertices);
    auto refVerticesHost = Kokkos::create_mirror_view(refVertices);
    Kokkos::deep_copy(refVerticesHost, refVertices);
    auto elemTriplet = connManager->getMyElementsTriplet();
    double h[3] = {hx, hy, hz};

    for (ordinal_type i=0; i<numOwnedElems; ++i) {
        elemTriplet = connManager->computeLocalElementGlobalTriplet(i,
            connManager->getMyElementsTriplet(),
            connManager->getMyOffsetTriplet());
        double offset[3] = {leftX + elemTriplet.x*hx+hx/2,
            leftY + elemTriplet.y*hy+hy/2,
            leftX + elemTriplet.z*hz+hz/2};

        for (ordinal_type j=0; j<numNodesPerElem; ++j) {
            for (ordinal_type k=0; k<dim; ++k)
              physVertexesHost(i,j,k) = offset[k]+h[k]/2.0*refVerticesHost(j,k);
        }
    }
    Kokkos::deep_copy(physVertexes, physVertexesHost);
}
// **************** COMPUTE ELEMENTS ORIENATIONS ****************************

//compute global ids of element vertices
DynRankViewIntHost elemNodesGID("", numOwnedElems, numNodesPerElem);
{
    for(ordinal_type i=0; i<numOwnedElems; ++i) {
        const auto GIDs = connManager->getConnectivity(i);
        for(ordinal_type j=0; j<numNodesPerElem; ++j) {
            elemNodesGID(i,j) = GIDs[j];
        }
    }
}

// compute orientations for cells (one time computation)
Kokkos::DynRankView<Orientation,DeviceSpaceType> elemOrts("elemOrts", numOwnedElems);
ots::getOrientation(elemOrts, elemNodesGID, hexa);

// **************** COMPUTE QUADRATURE POINTS ****************************
DefaultCubatureFactory cubFactory;
auto cellCub = cubFactory.create<DeviceSpaceType, scalar_t, scalar_t>(hexa.getBaseKey(), cubDegree);
ordinal_type numQPoints = cellCub->getNumPoints();
DynRankView quadPoints("", numQPoints, dim);
DynRankView weights("", numQPoints);
cellCub->getCubature(quadPoints, weights);
// ASSEMBLY OF LOCAL ELEMENT MATRICES

// Compute physical Dof Coordinates and Reference coordinates
DynRankView funAtQPoints(“”, numOwnedElems, numQPoints);
{
    DynRankView physQPoints(“”, numOwnedElems, numQPoints, dim);
    ct::mapToPhysicalFrame(physQPoints, quadPoints, physVertexes, basis->getBaseCellTopology());
    EvalRhsFunctor<DynRankView> functor;
    functor.funAtPoints = funAtQPoints;
    functor.points = physQPoints;
    Kokkos::parallel_for("loop for evaluating the rhs at quadrature points", numOwnedElems, functor);
}

// compute oriented basis functions at quadrature points
DynRankView basisValuesAtQPointsOriented(“”, numOwnedElems, basisCardinality, numQPoints);
DynRankView transformedBasisValuesAtQPointsOriented(“”, numOwnedElems, basisCardinality, numQPoints);
DynRankView basisValuesAtQPointsCells("inValues", numOwnedElems, basisCardinality, numQPoints);
DynRankView basisValuesAtQPoints("", basisCardinality, numQPoints);
basis->getValues(basisValuesAtQPoints, quadPoints);
rst::clone(basisValuesAtQPointsCells, basisValuesAtQPoints);

// modify basis values to account for orientations
ots::modifyBasisByOrientation(basisValuesAtQPointsOriented, basisValuesAtQPointsCells, elemOrts, basis.getRawPtr());

// transform basis values
fst::HGRADtransformVALUE(transformedBasisValuesAtQPointsOriented, basisValuesAtQPointsOriented);
DynRankView basisGradsAtQPointsOriented("", numOwnedElems, basisCardinality, numQPoints, dim);
DynRankView transformedBasisGradsAtQPointsOriented("", numOwnedElems, basisCardinality, numQPoints, dim);
DynRankView basisGradsAtQPointsCells("inValues", numOwnedElems, basisCardinality, numQPoints, dim);
DynRankView basisGradsAtQPoints("", basisCardinality, numQPoints, dim);
basis->getValues(basisGradsAtQPoints, quadPoints, OPERATOR_GRAD);
rst::clone(basisGradsAtQPointsCells, basisGradsAtQPoints);

// modify basis values to account for orientations
ots::modifyBasisByOrientation(basisGradsAtQPointsOriented,
  basisGradsAtQPointsCells,
  elemOrts,
  basis.getRawPtr());

// map basis functions to reference (oriented) element
DynRankView jacobianAtQPoints("", numOwnedElems, numQPoints, dim, dim);
DynRankView jacobianAtQPoints_inv("", numOwnedElems, numQPoints, dim, dim);
DynRankView jacobianAtQPoints_det("", numOwnedElems, numQPoints);
c::setJacobian(jacobianAtQPoints, quadPoints, physVertexes, hexa);
c::setJacobianInv (jacobianAtQPoints_inv, jacobianAtQPoints);

fst::HGRADtransformGRAD(transformedBasisGradsAtQPointsOriented, jacobianAtQPoints_inv, basisGradsAtQPointsOriented);
// compute integrals to assemble local matrices
DynRankView cellMat("cellMassMat", numOwnedElems, basisCardinality, basisCardinality),
    cellRhs("cellRhs", numOwnedElems, basisCardinality);

DynRankView weightedTransformedBasisValuesAtQPointsOriented("", numOwnedElems, basisCardinality, numQPoints);
DynRankView weightedTransformedBasisGradsAtQPointsOriented("", numOwnedElems, basisCardinality, numQPoints, dim);
DynRankView cellWeights("", numOwnedElems, numQPoints);
    rst::clone(cellWeights, weights);

    fst::multiplyMeasure(weightedTransformedBasisGradsAtQPointsOriented, cellWeights, transformedBasisGradsAtQPointsOriented);
    fst::multiplyMeasure(weightedTransformedBasisValuesAtQPointsOriented, cellWeights, transformedBasisValuesAtQPointsOriented);

    fst::integrate(cellMat, transformedBasisGradsAtQPointsOriented, weightedTransformedBasisGradsAtQPointsOriented);
    fst::integrate(cellMat, transformedBasisValuesAtQPointsOriented, weightedTransformedBasisValuesAtQPointsOriented, true);
    Kokkos::fence(); //make sure that funAtQPoints has been evaluated
    fst::integrate(cellRhs, funAtQPoints, weightedTransformedBasisValuesAtQPointsOriented);
Assembly into global matrix/vector

In order to compute $b_1$ and $b_2$ we need contributions from cells $K_0$ and $K_1$ that live on different procs.

Typical FE matrix and right-hand-side:

$$A_{ij} = \sum_{K \ni i,j} \int_K \mu \nabla \phi_j \cdot \nabla \phi_i,$$

Local FE matrix

$$A^K_{i,j} := \int_K \mu \nabla \psi_j \cdot \nabla \psi_i$$

Local FE vector

$$b_i = \sum_{K \ni i} \int_K f \phi_i$$

$A_{ij}$ and $b_i$ are computed locally for each cell $K$. In this example, $K_0$ is owned by $P_0$ and $K_1$ is owned by $P_1$. The nodes $P_0$ and $P_1$ are connected by shared cells $K_0$ and $K_1$.
Assembly into global matrix/vector

**Assembly**: Loop over cells and add the integral contributions into the corresponding entries of vectors/matrices.

There are different strategies to do so.

- **[type-1]** Loop over owned elements and sum into global entries. Tpetra will communicate at the fill-complete stage (most expensive).

- **[type-2]** Loop over owned elements and sum into owned and shared nodes. Need to store owned and overlap vectors/matrices and importing the overlap vectors/matrices into the owned ones.

- **[type-3]** Loop over owned and shared elements and sum contribution into owned nodes (potentially most efficient, no need of global communication, need access to shared cells, integrals are computed more than once on shared elements).
Assembly: Loop over cells and add the integral contributions into the corresponding entries of vectors/matrices.

There are different strategies to do so.

- **[type-1]** Loop over owned elements and sum into global entries. Tpetra will communicate at the fill-complete stage (most expensive).
- **[type-2]** Loop over owned elements and sum into owned and shared nodes. Need to store owned and overlap vectors/matrices and importing the overlap vectors/matrices into the owned ones.
- **[type-3]** Loop over owned and shared elements and sum contribution into owned nodes (potentially most efficient, no need of global communication, need access to shared cells, integrals are computed more than once on shared elements).
- **[type-1-FE]** It uses classes FECrsMultivector and FECrsMatrix. It is as easy to implement as type-1 but it implements an optimizes version of type-2 (aliasing memory). Similar to Epetra FE matrix and vector approach. (implemented by C. Siefert and G. Danielson)
Assembly into (Tpetra) global matrix/vector

Weak scaling, comparison of different assembly types for filling the Tpetra Graph (left) and matrix (right). (by C. Siefert and G. Danielson)

![Diagram showing weak scaling comparison]

- **[type-1]** Loop over owned elements and sum into global entries. Tpetra will communicate at the fill-complete stage (most expensive).
- **[type-2]** Loop over owned elements and sum into owned and shared nodes. Need to store owned and overlap vectors/matrices and importing the overlap vectors/matrices into the owned ones.
- **[type-3]** Loop over owned and shared elements and sum contribution into owned nodes (potentially most efficient, no need of global communication, need access to shared cells, integrals are computed more than once on shared elements)
- **[type-1-FE]** It uses classes FECrsMultivector and FECrsMatrix. It is as easy to implement as type-1 but it implements an optimizes version of type-2 (aliasing memory). Similar to Epetra FE matrix and vector approach. (implemented by C. Siefert and G. Danielson)
auto ugi = Teuchos::rcp_dynamic_cast<const panzer::UniqueGlobalIndexer<local_ordinal_t, global_ordinal_t> >(dofManager);

std::vector<global_ordinal_t> ownedIndices, ownedAndGhostedIndices;
ugi->getOwnedIndices(ownedIndices);
auto ownedMap = Teuchos::rcp(new map_t(T euchos::OrdinalTraits<global_ordinal_t>::invalid(),ownedIndices,0,comm_ptr));
ugi->getOwnedAndSharedIndices(ownedAndSharedIndices);
auto ownedAndShared_map = Teuchos::rcp(new const map_t(T euchos::OrdinalTraits<global_ordinal_t>::invalid(),ownedAndGhostedIndices,0,comm_ptr));

auto rowMap = ownedMap;
auto domainMap = ownedMap;
auto feGraph = Teuchos::rcp(new fe_graph_t(rowMap, ownedAndShared_map, 8*basisCardinality));

Teuchos::Array<global_ordinal_t> globalIdsInRow(basisCardinality);
auto elmtOffset = dofManager->getGIDFieldOffsets("eblock-0_0_0",0);

// fill graph
Tpetra::beginFill(*feGraph);
for(ordinal_type elemId=0; elemId<numOwnedElems; elemId++)
{
    // Populate globalIdsInRow:
    // - Copy the global node ids for current element into an array.
    std::vector<global_ordinal_t> elementGIDs;
dofManager->getElementGIDs(elemId, elementGIDs);
    for(ordinal_type nodeId=0; nodeId<basisCardinality; nodeId++)
    {
        globalIdsInRow[nodeId] = elementGIDs[elmtOffset[nodeId]];
    }
    // Add the contributions from the current row into the graph.
    for(ordinal_type nodeId=0; nodeId<basisCardinality; nodeId++)
    {
        feGraph->insertGlobalIndices(globalIdsInRow[nodeId], globalIdsInRow(nodeId));
    }
}
Tpetra::endFill(*feGraph);
auto A = Teuchos::rcp(new fe_matrix_t(feGraph));
auto b = Teuchos::rcp(new fe_multivector_t(domainMap, feGraph->getImporter(), 1));
Teuchos::Array<global_ordinal_t> columnLocalIds(basisCardinality);
Teuchos::Array<scalar_t> columnScalarValues(basisCardinality);  // scalar values for each column

//fill matrix
Tpetra::beginFill(*A,*b);
std::vector<global_ordinal_t> elementGIDs(basisCardinality);
auto elementLIDs = ugi->getLIDs();
for(ordinal_type elemId=0; elemId<numOwnedElems; elemId++)
{
    // Fill the global column ids array for this element
dofManager->getElementGIDs(elemId, elementGIDs);

    for(ordinal_type nodeId=0; nodeId<basisCardinality; nodeId++)
    columnLocalIds[nodeId] = A->getColMap()->getLocalElement(elementGIDs[elmtOffset[nodeId]]);

    // For each node (row) on the current element:
    // - populate the values array
    for(ordinal_type nodeId=0; nodeId<basisCardinality; nodeId++)
    {
        global_ordinal_t localRowId = elementLIDs(elemId, elmtOffset[nodeId]);

        for(ordinal_type colId=0; colId<basisCardinality; colId++)
        columnScalarValues[colId] = cellMat(elemId, nodeId, colId);

        A->sumIntoLocalValues(localRowId, columnLocalIds, columnScalarValues);
        b->sumIntoLocalValue(columnLocalIds[nodeId], 0, cellRhs(elemId, nodeId));
    }
}
Tpetra::endFill(*A, *b);
Wall-clock time plots, single V100 GPU
(by J. Watkins)
Phalanx: Lightweight DAG-based Expression Evaluation

- Decompose a complex model into a graph of simple kernels (functors)
- A node in the graph evaluates one or more temporary fields (memory for flexibility)
- Runtime DAG construction of graph
- Supports rapid development, separation of concerns and extensibility.
- Achieves flexible multiphysics assembly
- Leverages Sacado scalar types for non-invasive Jacobian, Hessian, ...
- Combine kernels for performance

\[
R_u^i = \sum_q \left[ \phi_u^i u - \nabla \phi_u^i \cdot q + \phi_u^i s \right] j |\omega_q|
\]
with \( q = -k \nabla u \)
Sacado: Embedded (Template-based) Automatic Differentiation

- Implement equations templated on the scalar type
- Libraries provide new scalar types that overload the math operators to propagate embedded quantities
  - Expression templates for performance
  - Derivatives: FAD (forward AD), RAD
  - Hessians
  - Stochastic Galerkin: PCE
  - Multipoint: Ensemble (Stokhos)
- Analytic Values (NO FD involved)

In FE, for computing the Jacobian we take derivatives w.r.t. the solution vector. Derivative size is the #DoFs on each cell.

<table>
<thead>
<tr>
<th>f</th>
<th>( \text{df/dx} )</th>
<th>( \text{df/dy} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( y^2 )</td>
<td>2 y * dx/dy = 0</td>
<td>2 y * dx/dy = 2 y</td>
</tr>
<tr>
<td>( \sin(y^2) )</td>
<td>( \cos(y^2) ) dy^2 / dx = 0</td>
<td>( \cos(y^2) ) dy^2 / dy = ( \cos(y^2) ) 2y</td>
</tr>
</tbody>
</table>
Panzer: FE assembly (CG/DG) and solution of nonlinear multi-physics forward/inverse problems

Uses
- **Intrepid2**: FE Basis Library
- **Shards**: Cell/Element Topology
- **Phalanx**: DAG Assembly: flexibility/complexity
- **Sacado**: Automatic differentiation scalar types
- **Kokkos**: Performance portability
- **Tpetra**: distributed linear algebra objects (vectors/matrices), hybrid parallelism
- **Belos/AztecOO/ML/MueLu/Itpack/Itpack2**: linear solvers/preconditioners
- **NOX/LOCA**: nonlinear solvers / continuation algorithms
- **Tempus**: time discretization
- **ROL**: optimization algorithms
A note on Hierarchic Parallelism
(by R. Pawlowski)

- Single level parallelism is insufficient
- Does not expose enough Parallelism
- 3-level hierarchical parallelism shows significant improvement (league/team/vector)
- Key is to vectorize over FAD derivative dimension

Incompressible CFD Kernel in MHD library Drekar

\[ \int_e c \left( \vec{f}(x) \cdot \nabla \varphi_i(x) + s(x) \varphi_i(x) \right) \, dx \]
A note on Hierarchic Parallelism
(by R. Pawlowski)

Loop Hierarchy for AD

```cpp
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void IntegrateDiffusionTerm<EvalT,Traits>::operator()(
    const TeamPolicy<exec_space>::member_type& team) const
{
    const int cell = team.league_rank();
    parallel_for(TeamThreadRange(team,0,grad_basis.extent(2)), [&] (const int& basis) {
        for (int qp = 0; qp < static_cast<int>(grad_basis.extent(1)); ++qp) {
            for (int dim = 0; dim < static_cast<int>(grad_basis.extent(3)); ++dim)
                residual(cell,basis) += - weighted_grad_basis(cell,qp,basis,dim) * flux(cell,qp,dim);
        }
    });
}
```

Where is the parallel_for(ThreadVectorRange(...)); ?!?
→ Hidden in AD Scalar type – run vector loop over the AD derivative hidden dimension
And for non AD types? SIMD (not yet implemented)
Current/Future development

- Intrepid2
  - interface refactoring to allow optimizations for tensor-product basis functions, for low order basis and for affine elements
  - functor based interface, allowing fused operations
  - hierarchical basis

- Panzer
  - Use of Shared memory (to cache temporaries and reuse basis values),
  - hierarchic parallelism, SIMD for non-AD types.
  - Host-DAG vs Device DAG

- Tutorials

- Coming-soon COMPADRE: package for local meshless discretizations (tools for data transfer and PDE discretization based on Generalized Moving Least Squares)