Phalanx: An Flexible and Extensible Assembly/Field Evaluation Kernel For Handling Complexity in Simulation

Roger Pawlowski

Trilinos User Group Meeting
November 4, 2009

SAND2009-7348P
Acknowledgements

• Phalanx Contributors
  – Eric Phipps
  – Carter Edwards
  – Pat Notz

• Consulting
  – Roscoe Bartlett
  – Pavel Bochev
  – Dennis Ridzal
  – Andy Salinger
Motivation

• Assembly for general FE/FV PDE discretizations gets quite complex when supporting arbitrary equation sets.

• Issues to Address:
  – Compact and uniform user interface for extensibility
  – Flexibility to easily swap equation sets, material models and material properties while maintaining efficiency
  – Support for user defined data types
  – Support for embedded technology
  – Good OO-design/Code reuse

• Generalization and unification of:
  – Expression Manager in SIERRA/Aria
  – Variable Manager in Charon
Overview

• Phalanx is a local field evaluation kernel designed for assembly of arbitrary equation sets (i.e. evaluating residuals and Jacobians).
  – Equation sets, material models might change drastically

• Decompose a complex problem into a number of simpler problems with managed dependencies
  – Supports rapid development and extensibility
  – Consistent evaluation of fields as dependencies change

• Phalanx supports arbitrary user defined data types and evaluation types through template metaprogramming.
  – Flexibility for direct integration with user applications
  – Provides extensive support for embedded technology such as automatic differentiation for sensitivities and uncertainty quantification.

• Efficient evaluation of fields using worksets and memory management for efficient use of cache.
Complex Dependency Chains

**Momentum**
\[
\frac{\partial (\rho v)}{\partial t} + \nabla \cdot (\rho v \otimes v + T) - \rho g = 0
\]

**Continuity**
\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = 0
\]

**Energy**
\[
\rho C_p \frac{\partial T}{\partial t} + \nabla \cdot (\rho C_p T v - q) + s = 0
\]

**Species**
\[
\rho \frac{\partial y_i}{\partial t} + \nabla \cdot (\rho v y_i + j_k) + W_i \omega_i = 0 \quad i = 1 \ldots N_{sp}
\]

\[
T = P I + \frac{2}{3} \mu (\nabla \cdot u) I - \mu (\nabla u + \nabla u^T)
\]

\[
q = -k \nabla T
\]

\[
j_k = \rho y_k \frac{1}{x_k W} \sum_{j=1}^{K} W_j D_{kj} \nabla x_j
\]

- Complexity spirals when we add new equations, operators, use subsets, etc...
  - Swap eqns/models at runtime (no complex if/switch statements)
- **Automatically adjust the dependency tree**
  \[
  \rho(T, P) \rightarrow \rho(y_i, P, T)
  \]
- Separate fields so that they are evaluated only once (density)
Idea: Evaluators (Expressions)

Material Property Library
- Density
- Heat Capacity
- Diffusion Coefficients
- Viscosity

Field Manager
- void registerEvaluator(TemplateMgr&)
- void registerEvaluator<Residual>(...)
- void registerEvaluator<Jacobian>(...)

Physics/Equation Set
- Navier Stokes Equation Residuals
- Energy Conservation Equation Residual
- Species Conservation Equation Residual

Solver
- Gather DOFs
- Scatter Residual

Evaluators can be registered anytime before postRegistrationSetup()
Evaluator Anatomy

\[ \rho \frac{\partial y_i}{\partial t} + \nabla \cdot (\rho \nu y_i) + \nabla \cdot j_k + W_i \omega_i \rightarrow F_n = \int_{\Omega} \nabla \phi_n \cdot j_k \, d\Omega \]

- Evaluates one or more fields
- Depends on one or more fields
- CTOR: Size the fields
- Setup Method: Get pointers to memory
- Evaluate Method: Evaluate the field(s)
- Intrepid package does the final integration
Idea: Chain of Evaluators

- Phalanx FieldManager will
  - Determine which evaluators to call
  - The order to call the evaluators for consistency
  - Perform the evaluation on a workset

```csharp
field_manager.evaluateFields<Residual>(workset);
field_manager.evaluateFields<Jacobian>(workset);
```
Evaluators: Simple for Users

- We must simplify interfaces for analysts to implement
  - Don’t expose entire equation set to users
  - Hide advanced c++ (i.e. templating) from analysts looking to add new equations and material models
  - Don’t have to know about derivatives/solver techniques

\[ q = -\rho k \nabla T \]

```cpp
PHX_EVALUATOR_CLASS(EnergyFlux)
Field< MyVector<ScalarT> > flux;
Field< ScalarT > density;
Field< ScalarT > k;
Field< MyVector<ScalarT> > grad_temp;
int points_per_cell;
PHX_EVALUATOR_CLASS_END
```

```cpp
PHX_EVALUATE_FIELDS(EnergyFlux, workset)
{
    int size = workset.num_cells * points_per_cell;
    for (int i = 0; i < size; ++i)
        flux[i] = -density[i] * k[i] * grad_temp[i];
}
```

Skipped CTOR (field sizing), Setup (get pointers to memory)
Workset/Memory Management

• Break work up into worksets
  – Chunk of cells in finite element/volume calculation
• Memory allocation of all fields of all scalar types for an evaluation type is done in a single contiguous array!
  – Possibly fit all fields in cache
  – User defined allocators (template parameter in Traits)
• Leverage BLAS in evaluators
What does this buy you?

• Consistent Evaluations: Dependencies are ensured to be up-to-date
• Evaluate each field once per cell
  – No recalculation of temporaries
• Flexible and Extensible: each simpler piece becomes an extension point that can be swapped out with different implementations
• Easier to craft code because each piece is simpler, more focused and easier to test in isolation
• Minimal interface: isolate users from bulk of assembly process
• Efficient: use of worksets
  – Block evaluations → Blas
  – Possibly fits into cache
Embedded Technology!!!

Field Manager is templated on Evaluation Type

Concept: Evaluation Types

- **Residual** $F(x, p)$
- **Jacobian** $J = \frac{\partial F}{\partial x}$
- **Tangent** $\frac{\partial F}{\partial p}$
- **Stochastic Galerkin Residual**
- **Stochastic Galerkin Jacobian**

Scalar Types

<table>
<thead>
<tr>
<th>Concept</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>Sacado::FAD::DFad&lt;double&gt;</td>
</tr>
<tr>
<td>Sacado::FAD::DFad&lt;double&gt;</td>
<td>Sacado::FAD::DFad&lt;double&gt;</td>
</tr>
<tr>
<td>Sacado::PCE::OrthogPoly&lt;double&gt;</td>
<td>Sacado::FAD::DFad&lt;double&gt;</td>
</tr>
<tr>
<td>Sacado::PCE::OrthogPoly&lt;double&gt;</td>
<td>Sacado::FAD::DFad&lt;double&gt;</td>
</tr>
</tbody>
</table>
Transformational PDE Assembly using Agile Components

**Take Home Message:**
1. Reuse the same code base
2. Never write Jacobians manually

**Packages / Libraries:**
- **Sacado:** Automatic Differentiation
- **Phalanx:** Field Manager, Evaluators
- **Intrepid:** Compatible Discretizations
- **iTAPS:** Mesh interface
struct MyTraits : public PHX::TraitsBase {

    typedef double RealType;
    typedef Sacado::Fad::DFad<double> FadType;

    struct Residual { typedef RealType ScalarT; };
    struct Jacobian { typedef FadType ScalarT; };

    typedef boost::mpl::vector<Residual, Jacobian> EvalTypes;

    // Residual (default scalar type is RealType)
    typedef boost::mpl::vector<
        RealType,
        MyVector<RealType>,
        MyMatrix<RealType>
    > ResidualDataTypes;

    // Jacobian (default scalar type is Fad<double>)
    typedef boost::mpl::vector<
        FadType,
        MyVector<FadType>,
        MyMatrix<FadType>
    > JacobianDataTypes;

    // Maps the key EvalType a vector of DataTypes
    typedef boost::mpl::map<
        boost::mpl::pair<Residual, ResidualDataTypes>,
        boost::mpl::pair<Jacobian, JacobianDataTypes>
    >::type EvalToDataMap;

    Declare Scalar Types
    Declare Evaluation Types
    Declare Residual Data Types
    Declare Jacobian Data Types
    Evaluation Types
    Maps Evaluation Types to Data Types
Multidimensional Arrays
(Shards – C. Edwards next)

Optional compile time checked access

```cpp
PHX_EVALUATOR_CLASS(EnergyFlux)

Field< MyVector<ScalarT> > flux;
Field< ScalarT > density;
Field< ScalarT > k;
Field< MyVector<ScalarT> > grad_temp;

int points_per_cell;

PHX_EVALUATOR_CLASS_END

PHX_EVALUATOR_CLASS(EnergyFlux)

MDField<ScalarT,Cell,QuadPoint,Dim> flux;
MDField<ScalarT,Cell,QuadPoint> density;
MDField<ScalarT,Cell,QuadPoint> dc;
MDField<ScalarT,Cell,QuadPoint,Dim> grad_temp;

int num_qp;
int num_dim;

PHX_EVALUATOR_CLASS_END

PHX_EVALUATE_FIELDS(EnergyFlux,workset)
{
    int size = workset.num_cells * points_per_cell;
    for (int i = 0; i < size; ++i)
        flux[i] = -density[i] * k[i] * grad_temp[i];
}

PHX_EVALUATE_FIELDS(EnergyFlux,workset)
{
    int num_cells = workset.num_cells;
    for (int cell = 0; cell < num_cells; ++cell)
        for (int qp = 0; qp < num_qp; ++qp)
            for (int dim = 0; dim < num_dim; ++dim)
                flux(cell,qp,dim) =
                    - density(cell,qp) * dc(cell,qp) * grad_temp(cell,qp,dim);
}
```
In closing...

• This package is very advanced
  – C++ templates and template metaprogramming
  – One developer will need to know templates to set up
  – Everyone else only needs to write evaluators (very minimal template code)

• Think hard before using
  – This is a hammer, its not right for every PDE code, especially if your equation set/models doesn’t change

• “My understanding keeps changing...” – Andy Salinger

• Don’t hesitate to ask for help!

Phalanx Summary

• Assembly kernel for cell based discretization of PDEs

• Breaks complex problems into simpler pieces
  – Automatically manage complex dependency chains
  – Easier to unit test
  – Don’t expose fully complex system to the user – only expose exactly what they need to write a user defined function

• Supports rapid development and extensibility
  – Easily swap evaluation routines
  – Easily swap dependency trees

• Arbitrary user defined data types and evaluation types: C++ Template metaprogramming

• Embedded technology support