Template-based Generic Programming Techniques for Finite Element Assembly

Roger Pawlowski, Eric Cyr, Eric Phipps, Andrew Salinger and John Shadid
Sandia National Laboratories
Challenges in Multiphysics Simulation

Physics Model Complexity

• Solving multiphysics PDE systems generates complexity:
  – Complex interdependent coupled physics
  – Multiple proposed mathematical models
  – Different numerical formulations (e.g. space-time discretizations)
• Supporting multiplicity in models and solution techniques often leads to complex code with complicated logic and fragile software designs

Analysis Beyond Forward Simulation

• Forward solves are not enough – we want to explore complex solution spaces:
  – Simultaneous analysis and design adds requirements (typically sensitivities)
  – Do not burden analysts/physics experts with analysis algorithm requirements: i.e. programming sensitivities for implicit solvers, optimization, stability, bifurcation analysis and UQ

Engine must be flexible, extensible, maintainable and EFFICIENT!
Challenges in Multiphysics Simulation

Physics Model Complexity

• Solving multiphysics PDE systems generates complexity:
  – Complex interdependent coupled physics
  – Different numerical formulations (e.g. space-time discretizations)
• Supporting multiplicity in models and solution techniques often leads to complex code with complicated logic and fragile software designs

Directed Acyclic Graph-based Assembly

Analysis Beyond Forward Simulation

• Forward solves are not enough – we want to explore complex solution spaces:
  – Simultaneous analysis and design adds requirements (typically

Template-based Generic Programming

requirements: i.e. programming sensitivities for implicit solvers, optimization, stability, bifurcation analysis and UQ

Engine must be flexible, extensible, maintainable and EFFICIENT!
DAG-based Assembly

- Widely used idea in both research and production codes. Codes leveraging this:
  - Albany: Salinger
  - Amanzi: Moulton
  - Charon/Drekar/Panzer: Pawlowski and Cyr
  - SIERRA/Aria: Notz, …
  - Uintah: Berzins and Sutherland

Lightweight DAG-based Expression Evaluation

- Decompose a complex model into a graph of simple kernels (functors)

- Supports rapid development, separation of concerns and extensibility.

- A node in the graph evaluates one or more fields:
  - Declare fields to evaluate
  - Declare dependent fields
  - Function to perform evaluation

- Separation of data (Fields) and kernels (Expressions) that operate on the data
  - Fields are accessed via multidimensional array interface

- Can use for asynchronous task management on node!
Navier-Stokes Example

• Graph-based equation description
  – Automated runtime dependency tracking (Topological sort to order the evaluations)
  – Each node is a point of extension that can be swapped out
  – Easy to add equations
  – Easy to change models
  – Easy to test in isolation
  – User controlled granularity
  – No unique decomposition

\[
R_T^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \left( (\rho C_p v \cdot \nabla T - H_v) \phi_T^i - q \cdot \nabla \phi_T^i \right) w_q|j| = 0
\]

\[
R_{vk}^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \left( \rho v \cdot \nabla \phi_v^i + \sigma : \nabla (\phi_v^i e_k) \right) w_q|j| = 0
\]

\[
R_p^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \nabla \cdot v \phi_p^i w_q|j| = 0
\]

• Multi-core research:
  – Spatial vs algorithmic decomposition
  – Kernel launch: fused vs separate
Analysis Beyond Forward Simulation

- Model problem

\[ f(\dot{x}, x, p) = 0, \quad \dot{x}, x \in \mathbb{R}^n, \quad p \in \mathbb{R}^m, \quad f : \mathbb{R}^{2n+m} \rightarrow \mathbb{R}^n \]

- Direct to steady-state, implicit time-stepping, linear stability analysis

\[ \left( \alpha \frac{\partial f}{\partial \dot{x}} + \beta \frac{\partial f}{\partial x} \right) \Delta x = -f \]

- Steady-state sensitivity analysis

\[ f(x^*, p) = 0, \quad s^* = g(x^*, p) \implies \]

\[ \frac{ds^*}{dp} = -\frac{\partial g}{\partial x}(x^*, p) \left( \frac{\partial f}{\partial x}(x^*, p) \right)^{-1} \frac{\partial f}{\partial p}(x^*, p) + \frac{\partial g}{\partial p}(x^*, p) \]

- Bifurcation analysis

\[ f(x, p) = 0, \quad \sigma(x, p) = 0, \quad \sigma = -u^T Jv, \quad \frac{\partial \sigma}{\partial x} = -u^T \frac{\partial}{\partial x}(Jv), \quad \frac{\partial \sigma}{\partial p} = -u^T \frac{\partial}{\partial p}(Jv), \]

\[
\begin{bmatrix}
J & 0 \\
b^T & a^T
\end{bmatrix}
\begin{bmatrix}
v \\
s_1
\end{bmatrix}
=
\begin{bmatrix}
0 \\
1
\end{bmatrix},
\begin{bmatrix}
J^T & b \\
a^T & 0
\end{bmatrix}
\begin{bmatrix}
u \\
s_1
\end{bmatrix}
=
\begin{bmatrix}
0 \\
1
\end{bmatrix}
\]
Template-based Generic Programming (TBGP)

- 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, RAD
  - Stochastic Galerkin: PCE
  - Multipoint: Ensemble

<table>
<thead>
<tr>
<th>Operation</th>
<th>Forward AD rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a \pm b$</td>
<td>$\dot{c} = \dot{a} \pm \dot{b}$</td>
</tr>
<tr>
<td>$c = ab$</td>
<td>$\dot{c} = a\dot{b} + \dot{a}b$</td>
</tr>
<tr>
<td>$c = a/b$</td>
<td>$\dot{c} = (\dot{a} - \dot{b}b)/b$</td>
</tr>
<tr>
<td>$c = a^r$</td>
<td>$\dot{c} = r\dot{a}a^{r-1}$</td>
</tr>
<tr>
<td>$c = \sin(a)$</td>
<td>$\dot{c} = \cos(a)\dot{a}$</td>
</tr>
<tr>
<td>$c = \cos(a)$</td>
<td>$\dot{c} = -\sin(a)\dot{a}$</td>
</tr>
<tr>
<td>$c = \exp(a)$</td>
<td>$\dot{c} = c\dot{a}$</td>
</tr>
<tr>
<td>$c = \log(a)$</td>
<td>$\dot{c} = \dot{a}/a$</td>
</tr>
</tbody>
</table>

Fad:

$$\frac{df}{dx}(x_0) V$$

$V \in \mathbb{R}^{n \times p}$

$$\frac{dx}{dz} = V$$

Seeding/initializing $V$

For J: $V = I$

For Jw: $V = w$
TBGP Example

\[ f_0 = 2x_0 + x_1^2 \]
\[ f_1 = x_0^3 + \sin(x_1) \]

```cpp
void computeF(double* x, double* f)
{
    f[0] = 2.0 * x[0] + x[1] * x[1];
    f[1] = x[0] * x[0] * x[0] + sin(x[1]);
}
```

```cpp
template<typename ScalarT>
void computeF(ScalarT* x, ScalarT* f)
{
    f[0] = 2.0 * x[0] + x[1] * x[1];
    f[1] = x[0] * x[0] * x[0] + sin(x[1]);
}
```

```cpp
void computeJ(double* x, double* J)
{
    // J(0,0)
    J[0] = 2.0;
    // J(0,1)
    J[1] = 2.0 * x[1];
    // J(1,0)
    J[2] = 3.0 * x[0] * x[0];
    // J(1,1)
    J[3] = cos(x[1]);
}
```

Same accuracy as writing analytic derivative:
No differencing error involved!
Example Scalar Types
(Trilinos Stokhos and Sacado: E. Phipps)

<table>
<thead>
<tr>
<th>Evaluation Types</th>
<th>Scalar Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Residual</strong> ( F(x, p) )</td>
<td>double</td>
</tr>
<tr>
<td><strong>Jacobian</strong> ( J = \frac{\partial F}{\partial x} )</td>
<td>DFad&lt;double&gt;</td>
</tr>
<tr>
<td><strong>Hessian</strong> ( \frac{\partial^2 F}{\partial x_i \partial x_j} )</td>
<td>DFad&lt;DFad&lt;double&gt; &gt;</td>
</tr>
<tr>
<td><strong>Parameter Sensitivities</strong> ( \frac{\partial F}{\partial p} )</td>
<td>DFad&lt;double&gt;</td>
</tr>
<tr>
<td><strong>Jv</strong> ( Jv )</td>
<td>DFad&lt;double&gt;</td>
</tr>
<tr>
<td><strong>Stochastic Galerkin Residual</strong></td>
<td>PCE&lt;double&gt;</td>
</tr>
<tr>
<td><strong>Stochastic Galerkin Jacobian</strong></td>
<td>DFad&lt;PCE&lt;double&gt; &gt;</td>
</tr>
</tbody>
</table>

1. All evaluation types are compiled into single library and managed at runtime from a non-template base class via a template manager.
2. Not tied to double (can do arbitrary precision)
3. Can mix multiple scalar types in any evaluation type.
4. Can specialize any node: Write analytic derivatives for performance!
TBGP in Multiphysics
PDE Assembly

PDE Equation: \[\ddot{u} + \nabla \cdot q + s = 0\]

Galerykin Weak form ignoring boundary terms for simplicity:

\[R_u^i = \int_\Omega \left[ \phi_u^i \dot{u} - \nabla \phi_u^i \cdot q + \phi_u^i s \right] d\Omega\]

FEM Basis:

\[u = \sum_{i=1}^{N_u} \phi_u^i u^i\]

Residual Equation:

\[\hat{R}_u^i = \sum_{e=1}^{N_E} \sum_{q=1}^{N_q} \left[ \phi_u^i \dot{u} - \nabla \phi_u^i \cdot q + \phi_u^i s \right] w_q |j| = 0\]
TBGP + DAG: Global Evaluation

\[ f(x) = \sum_{k=1}^{N_w} S^f_k \bar{R}^i_{uk}(G^f_k x) \]

- Break mesh into worksets of elements
- Only have to specialize two expressions for evaluation type:
  - Gather/Seed
  - Extract/Scatter
- All other code is reused
- Achieved separation of concerns!
- Machine precision accurate derivatives
- Kokos hides node specializations

Extract values from Scalar type and scatter to global residual
\[ f_k \leftarrow S^f_k \bar{R}^i_{uk} \]
\[ J_k \leftarrow S^J_k \bar{R}^i_{uk} \]
Handling Complexity in Analysis Requirements

TBGP, Pawlowski, Phipps, Salinger; Scientific Programming, in press.

\[
f(x) = \sum_{k=1}^{N_w} f_k = \sum_{k=1}^{N_w} Q_k^T \hat{R}_T^i (P_k x)
\]

\[
\hat{R}_T^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} [-\nabla \phi_T^i \cdot q + \phi_T^i s] w_q |j| = 0
\]

Evaluation Type

Scalar Type

\[
f(x, p)
\]

\[
J = \frac{\partial f}{\partial x}
\]

\[
\frac{\partial^2 f}{\partial x_i \partial x_j}
\]

double

DFad<double>

DFad<DFad<double> >

Param. Sens., Jv, Adjoint, PCE (SGF, SGJ), AP

Take Home Message:
Reuse the same code base!
Equations decoupled from algorithms!
Machine precision accuracy!
Node (functor) Example

template<typename EvalT, typename Traits>
class NonlinearSource : public PHX::EvaluatorWithBaseImpl<Traits>,
    public PHX::EvaluatorDerived<EvalT, Traits> {

public:
    NonlinearSource(const Teuchos::ParameterList& p);
    void postRegistrationSetup(typename Traits::SetupData d, PHX::FieldManager<Traits>& vm);
    void evaluateFields(typename Traits::EvalData d);
    void preEvaluate(typename Traits::PreEvalData d);
    void postEvaluate(typename Traits::PostEvalData d);

    KOKKOS_INLINE_FUNCTION
    void operator () (const int i) const;

private:
    typedef typename EvalT::ScalarT ScalarT;

    PHX::MDField<ScalarT,Cell,Point> source;
    PHX::MDField<const ScalarT,Cell,Point> density;
    PHX::MDField<const ScalarT,Cell,Point> temp;

    std::size_t cell_data_size;
};
Node (functor) Example

template<typename EvalT, typename Traits> NonlinearSource<EvalT, Traits>::
NonlinearSource(const Teuchos::ParameterList& p) : …
{
    this->addEvaluatedField(source);
    this->addDependentField(density);
    this->addDependentField(temp);
    this->setName("NonlinearSource");
}

template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void NonlinearSource<EvalT, Traits>::operator () (const int i) const
{
    for (int ip = 0; ip < density.dimension(1); ++ip)
        source(i,ip) = density(i,ip) * temp(i,ip) * temp(i,ip);
}

template<typename EvalT, typename Traits>
void NonlinearSource<EvalT, Traits>::
evaluateFields(typename Traits::EvalData d)
{
    Kokkos::parallel_for (d.num_cells, *this);
}
Rapid Development of New Physics
(Single driver and collection of interchangeable evaluators)
• Tremendous savings in development time
• Coding sensitivities is error prone and time consuming, especially when accounting for changing models/parameters!
• Vector intrinsics are hidden in the scalar types
Sensitivity Analysis Capability
Demonstrated on the QASPR Simple Prototype

• Bipolar Junction Transistor
• Pseudo 1D strip (9x0.1 micron)
• Full defect physics
• 126 parameters

Sensitivities show dominant physics

Comparison to FD:
✓ Sensitivities at all time points
✓ More accurate
✓ More robust
✓ 14x faster!
Large-Scale Semiconductor Device Simulations on IBM Blue Gene Platform (P. Lin)

- Generic programming (via AD tools) is applied at the element level, not globally.
- Weak scaling to 65k cores and two billion DOF: Jacobian evaluation via AD scales
- Using all four cores per node with MPI process on each core.

<table>
<thead>
<tr>
<th>cores</th>
<th>DOF</th>
<th>Jacobian time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>7.93m</td>
<td>52.19</td>
</tr>
<tr>
<td>1024</td>
<td>31.5m</td>
<td>52.28</td>
</tr>
<tr>
<td>4096</td>
<td>126m</td>
<td>52.09</td>
</tr>
<tr>
<td>8192</td>
<td>253m</td>
<td>52.82</td>
</tr>
<tr>
<td>16384</td>
<td>504m</td>
<td>52.74</td>
</tr>
<tr>
<td>32768</td>
<td>1.01b</td>
<td>52.96</td>
</tr>
<tr>
<td>65536</td>
<td>2.01b</td>
<td>52.94</td>
</tr>
</tbody>
</table>
Example: JFNK
(2D Diffusion/Rxn System: 2 eqns)

- **JFNK (FD)**
  \[ Jv \approx \frac{F(x + \delta v) - F(x)}{\delta} \]
  \[ t \approx (\text{num}_{\text{Its}}) \ast \text{cost}(F) \]

- **JFNK (AD)**
  - Machine precision accurate
  - Ex: Solution varies $10^{12}$ over domain
  \[ Jv \leq 2.5 \ast \text{cost}(F) \]
  \[ t \approx 1.53 \ast (\text{num}_{\text{Its}}) \ast \text{cost}(F) \]

- **Explicit Jacobian (AD generated)**
  - Machine precision accurate
  - Complexity ideas allow for storing individual operators for preconditioning!
  - Larger memory requirements
  \[ J(x) \leq 13 \ast \text{cost}(F) \]
  \[ t \approx 4.45 + (\text{num}_{\text{Its}}) \ast \text{cost}(Mv) \]

<table>
<thead>
<tr>
<th>Relative times</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(x)</td>
</tr>
<tr>
<td>J(x)</td>
</tr>
<tr>
<td>Jv (AD)</td>
</tr>
<tr>
<td>Mv (matvec)</td>
</tr>
</tbody>
</table>

![Graph showing relative times and GMRES iterations]
Multiple-time-scale systems: Bifurcation Analysis of a Steady Reacting H₂, O₂, Ar, Opposed Flow Jet Reactor

Approx. Physical Time scales (sec.):
- Chemical kinetics: $10^{-12}$ to $10^{-4}$
- Momentum diffusion: $10^{-6}$
- Heat conduction: $10^{-6}$
- Mass diffusion: $10^{-5}$ to $10^{-4}$
- Convection: $10^{-5}$ to $10^{-4}$
- Diffusion flame dynamics: (steady)
Embedded UQ in Drekar:
Rod to Fluid Heat Transfer
Issues

• Very flexible, maybe too much so?
  – Extreme flexibility allows you to shoot yourself in the foot!
  – Blind Application of TBGP can be inefficient (Minimize Scatter, AD sensitivities at the local element level)

• Efficient expression templates may require more recent compilers:
  – Gnu 4.6+, Intel 11+

• AD can be slower than hand coded derivatives
  – For implicit methods, assembly is usually not the bottleneck – inverting the Jacobian is the bottleneck
  – Adding new parameter sensitivities is difficult for (multiple) ever-changing physics models, …
  – Can use AD as first cut for Jacobian, then go back and replace terms with hand coded where appropriate
  – Development time spent debugging hand coded Jacobians is significant!

• Advanced C++ language features (templates) can be intimidating
  – Error reporting of templated code is improving
  – Expended significant effort to minimize/hide templates from node impls
Conclusions

DAG + TBGP:

• Exascale hardware → multiphysics → combinatorial explosion of sensitivity requirements.
  – Changing equation sets, formulations will change sensitivity requirements!

• We can write very advanced multiphysics software that is efficient, flexible and maintainable but templates are crucial

• Decoupling algorithms from equations is powerful:
  – We don’t write Jacobians anymore - enormous savings of manpower!

• Generic programming allows:
  – Segregation of technologies
  – Easily adaptive environment (from SE standpoint)

• Machine precision accuracy of required quantities is achieved

• Future: Integration of ATM for functional parallelism
Trilinos Tools for Supporting TBGP

- **Panzer**: Multiphysics assembly framework
- **Intrepid**: Discretizations tools for PDEs
  - Basis functions, quadrature rules, …
- **Phalanx**: DAG Assembly manager
  - DAG for multiphysics complexity
  - Explicitly manages fields/kernels for different evaluation/scalar types
- **Stokhos**: UQ Scalar Types
  - PCE and multipoint/ensemble scalar type classes/overloaded operators
  - Simultaneous ensemble propagation classes, overloaded operators
  - Tools and data structures for forming, solving embedded SG systems
- **Sacado**: AD Scalar types
  - AD scalar types
  - Parameter library – tools to manage model parameters
  - MPL – simple implementation of some metaprogramming constructs
- **Kokkos (shards mda deprecated)**
  - Multi-dimensional array for next-gen architectures