SAND2012-1207 C

Sandia National Laboratories

Template-based Generic Programming Techniques for Finite Element Assembly

Roger Pawlowski, Eric Cyr, Eric Phipps, Andrew Salinger and John Shadid
Sandia National Laboratories

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

National Nuclear Security Administration

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

Template-based Generic Programming

Engine must be flexible, extensible, maintainable and EFFICIENT!

Sandia National Laboratories

DAG-based Assembly

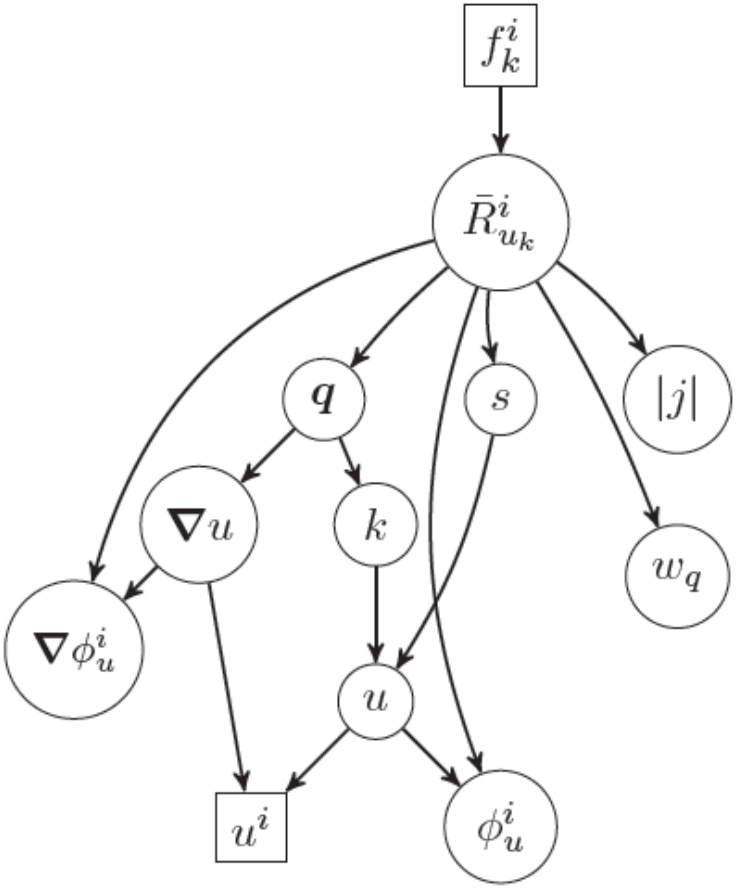
- Widely used idea in both research and production codes. Codes leveraging this:
  - Albany: Salinger
  - Amanzi: Moulton
  - Charon/Drekar/Panzer: Pawlowski and Cyr
  - SIERRA/Aria: Notz, …
  - Uintah: Berzins and Sutherland

P. K. Notz, R. P. Pawlowski, and J. C. Sutherland, **Graph-Based Software Design for Managing Complexity and Enabling Concurrency in Multiphysics PDE Software,** ACM Transactions on Mathematical Software, Vol. 39, No. 1 (2012).

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!

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



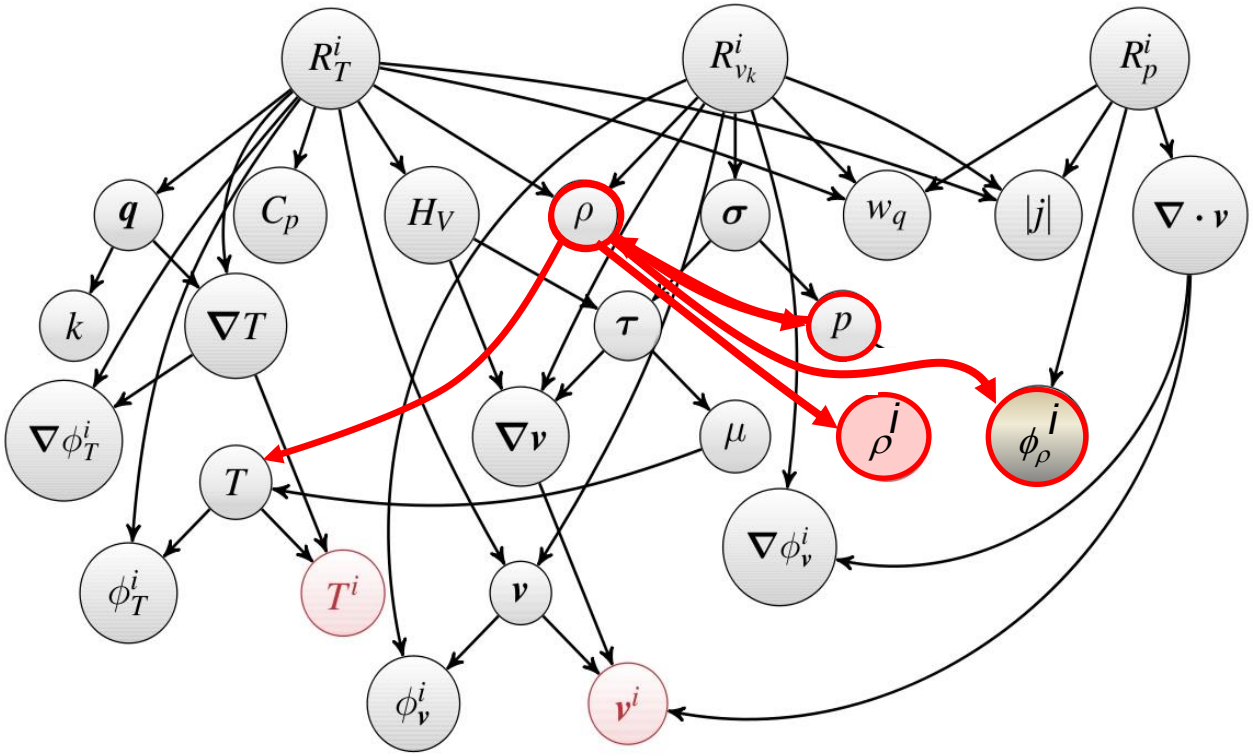
Navier-Stokes Example

- 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
- Multi-core research:
  - Spatial vs algorithmic decomposition
  - Kernel launch: fused vs separate

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

$$R_{v_k}^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \left[\rho \boldsymbol{v} \cdot \nabla \boldsymbol{v} \phi_{\boldsymbol{v}}^i + \boldsymbol{\sigma} : \nabla \left(\phi_{\boldsymbol{v}}^i \boldsymbol{e}_k \right) \right] w_q |j| = 0$$

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



Analysis Beyond Forward Simulation

- Model problem

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

- 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

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

$$\begin{bmatrix} J & a \\ b^T & 0 \end{bmatrix} \begin{bmatrix} v \\ s_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} J^T & b \\ a^T & 0 \end{bmatrix} \begin{bmatrix} u \\ s_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Sandia National Laboratories

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

`double`	`Fad<double>`
Operation	Forward AD rule
$c = a \pm b$	$\dot{c} = \dot{a} \pm \dot{b}$
$c = ab$	$\dot{c} = a\dot{b} + \dot{a}b$
$c = a/b$	$\dot{c} = (\dot{a} - c\dot{b})/b$
$c = a^r$	$\dot{c} = ra^{r-1}\dot{a}$
$c = \sin(a)$	$\dot{c} = \cos(a)\dot{a}$
$c = \cos(a)$	$\dot{c} = -\sin(a)\dot{a}$
$c = \exp(a)$	$\dot{c} = c\dot{a}$
$c = \log(a)$	$\dot{c} = \dot{a}/a$

$$\dot{u} := \frac{du}{dz}$$

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

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

$dx/dz = V$

Seeding/initializing V

For J: $V = \mathbf{I}$

For Jw: $V = w$

Sandia National Laboratories

TBGP Example

$$f_0 = 2x_0 + x_1^2$$
$$f_1 = x_0^3 + sin(x_1)$$

```
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]);
}
```

```
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]);
}
```

```
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]);
}
```

```
double* x;
double* f;
…
computeF(x,f);
```

```
DFad<double>* x;
DFad<double>* f;
…
computeF(x,f);
```

Same accuracy as writing analytic derivative:
No differencing error involved!

Sandia National Laboratories

Example Scalar Types

(Trilinos Stokhos and Sacado: E. Phipps)

Evaluation Types		Scalar Types
• **Residual**	$F(x,p)$	`double`
• **Jacobian**	$J = \frac{\partial F}{\partial x}$	`DFad<double>`
• **Hessian**	$\frac{\partial^2 F}{\partial x_i \partial x_j}$	`DFad< DFad<double> >`
• **Parameter Sensitivities**	$\frac{\partial F}{\partial p}$	`DFad<double>`
• **Jv**	Jv	`DFad<double>`
• **Stochastic Galerkin Residual**		`PCE<double>`
• **Stochastic Galerkin Jacobian**		`DFad< PCE<double> >`

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: $\dot{u} + \nabla \cdot \mathbf{q} + s = 0 \qquad \mathbf{q} = -k\nabla u$

Galerkin Weak form ignoring boundary terms for simplicity:

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

FEM Basis: $$u = \sum_{i=1}^{N_u} \phi_u^i u^i$$

Residual Equation:

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

Sandia National Laboratories

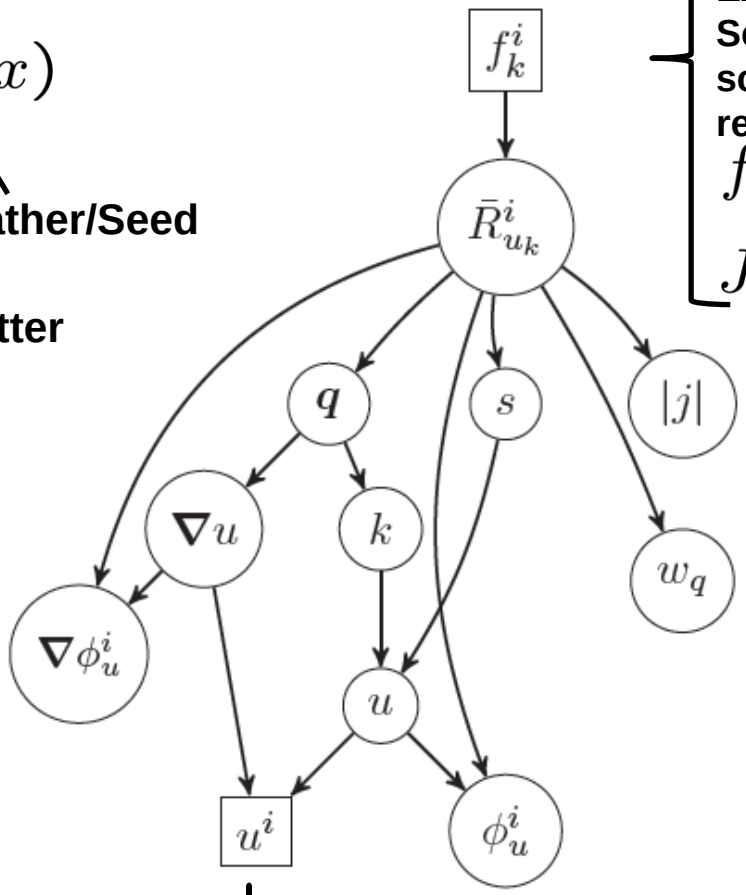
TBGP + DAG: Global Evaluation

$$f(x) = \sum_{k=1}^{N_w} S_k^f \bar{R}_{u_k}^i (G_k^f x)$$

Break mesh into worksets of elements

Extract/Scatter

Gather/Seed



Extract values from Scalar type and scatter to global residual

$$f_k \leftarrow S_k^f \bar{R}_{u_k}^i$$

$$J_k \leftarrow S_k^J \bar{R}_{u_k}^i$$

$$u_k \leftarrow G_k^f x$$

$$u_k \leftarrow G_k^J x$$

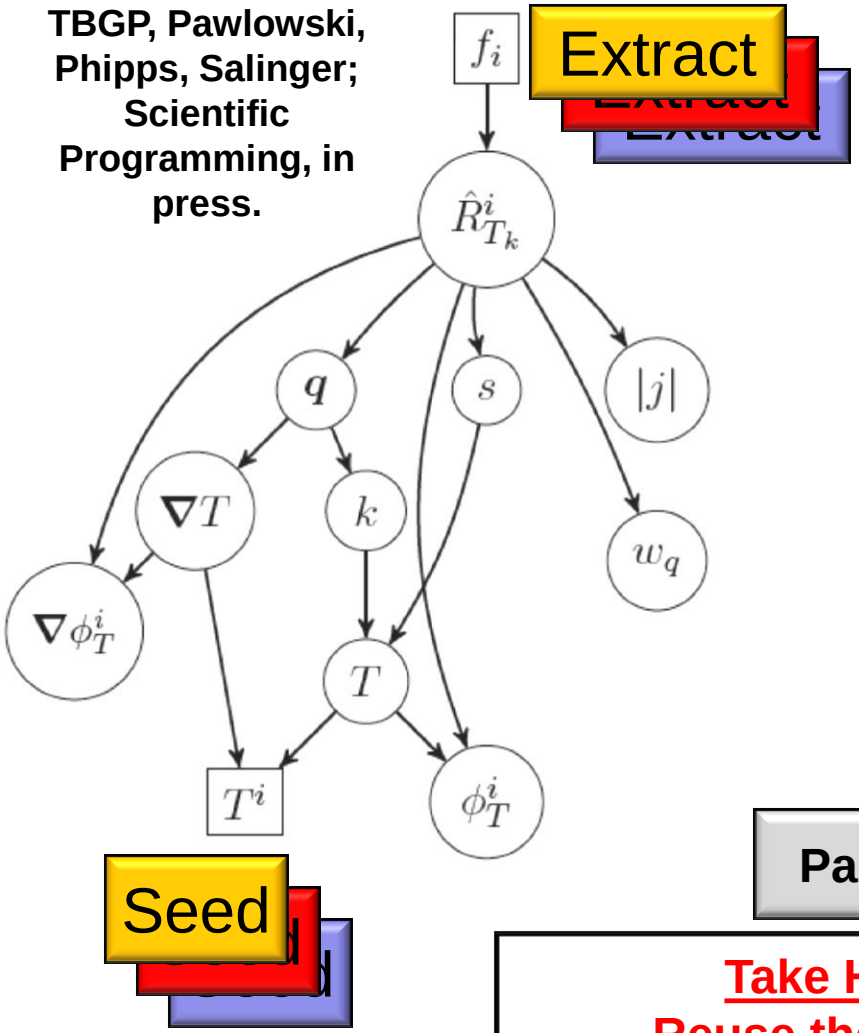
Gather solution values and seed Scalar type

- 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

Sandia National Laboratories

Handling Complexity in Analysis Requirements

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



$$f(x) = \sum_{k=1}^{N_w} f_k = \sum_{k=1}^{N_w} Q_k^T \hat{R}^i_{T_k}(P_k x)$$

$$\hat{R}^i_T = \sum_{e=1}^{N_e}\sum_{q=1}^{N_q}\left[-\nabla\phi^i_T \cdot \boldsymbol{q} + \phi^i_T s\right] w_q |j| = 0$$

Evaluation Type	Scalar Type
$f(x,p)$	`double`
$J = \dfrac{\partial f}{\partial x}$	`DFad<double>`
$\dfrac{\partial^2 f}{\partial x_i \partial x_j}$	`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!

Sandia National Laboratories

Node (functor) Example

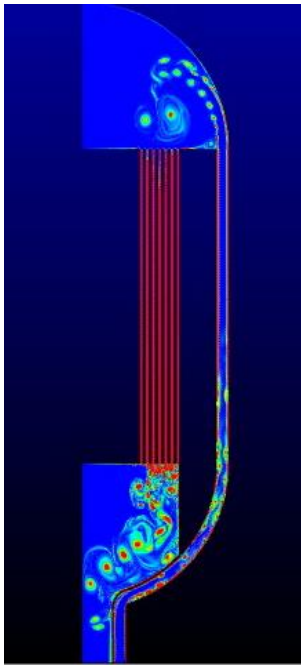
```
template<typename EvalT, typename Traits>
class NonlinearSource : public PHX::EvaluatorWithBaseImpl<Traits>,
                        public PHX::EvaluatorDerived<EvalT, Traits> {
public:
  NonlinearSource(const Teuchos::ParameterList& p);
  void postRegistrationSetup(typename Traits::SetupData d, 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;
};
```

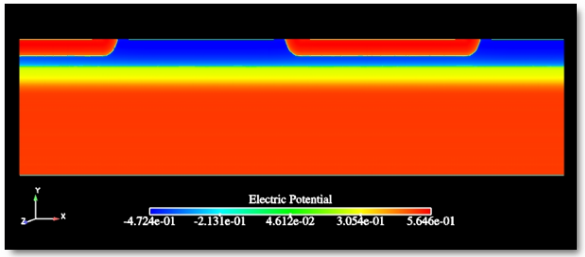
Node (functor) Example

```
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);
}
```

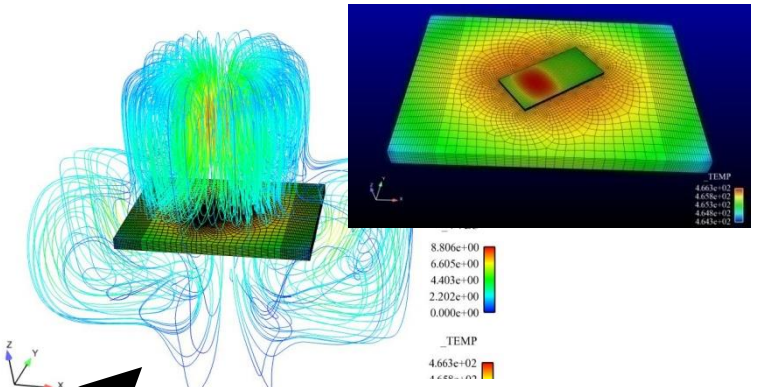
Rapid Development of New Physics

(Single driver and collection of interchangeable evaluators)





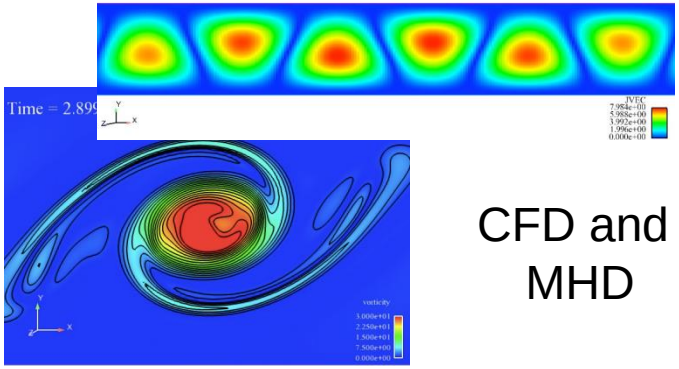
Semiconductor
Drift Diffusion



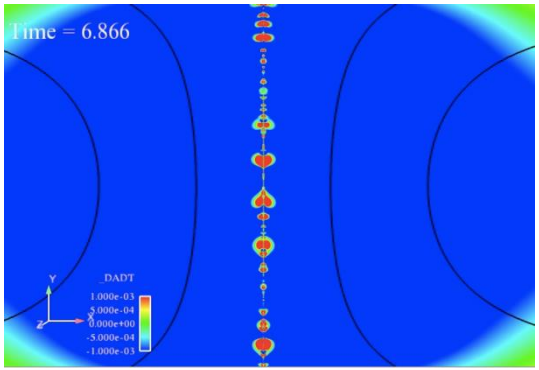
Chemicurrent

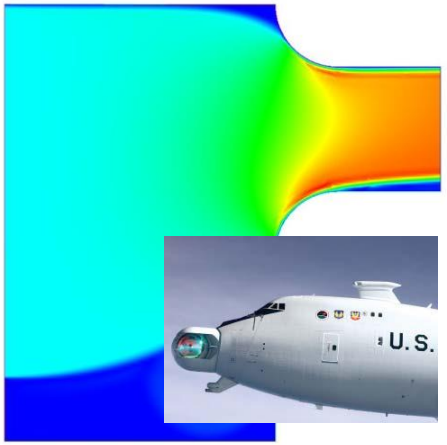
DAG Assembly +TBGP

NGNP Reactor



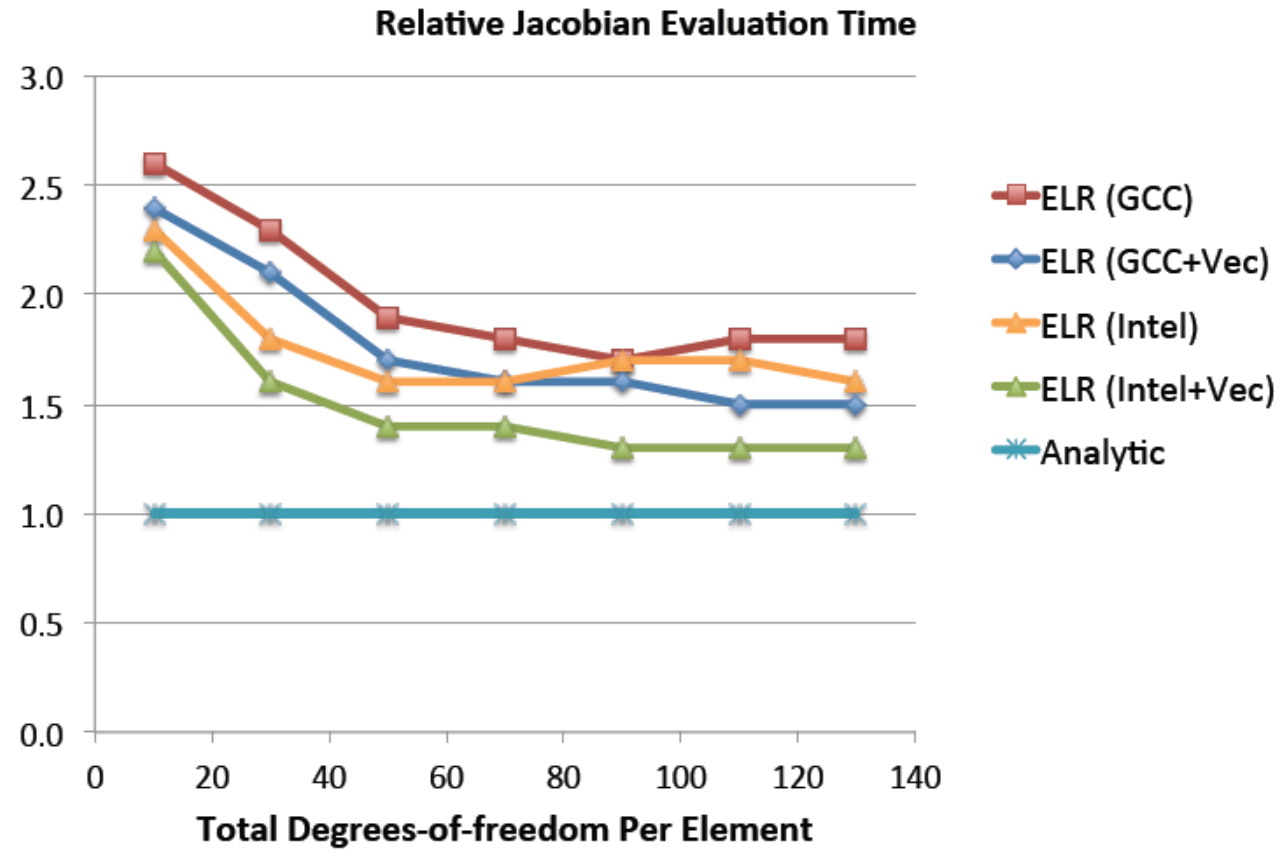
CFD and
MHD





Multi-phase
Chemically
Reacting Aerosol

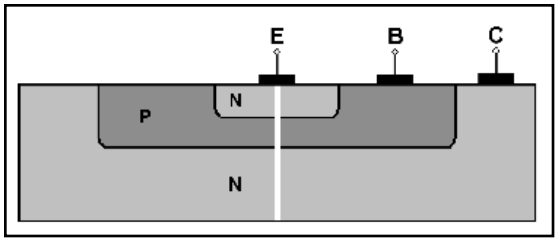
Jacobian Evaluation Efficiency



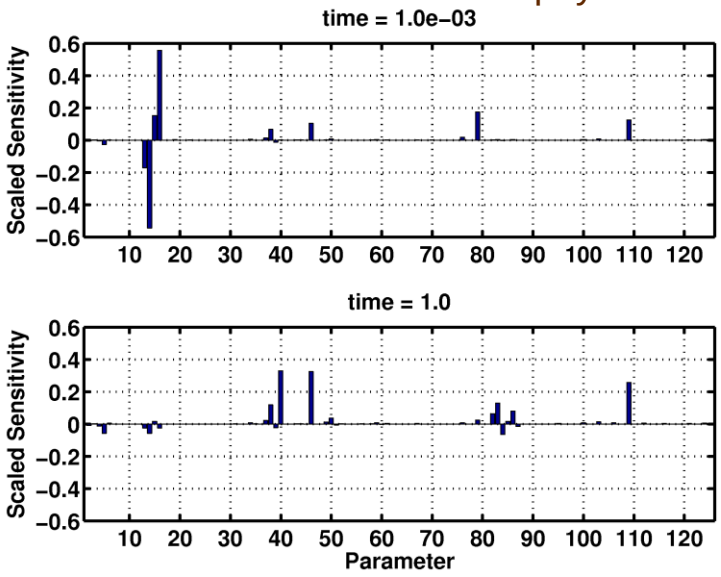
- Tremendous savings in development time
- Coding sensitivities is error prone and time consuming, especially when accounting for changing models/parameters!
- Vector intrinsics are hidden in the scalar types

Sensitivity Analysis Capability Demonstrated on the QASPR Simple Prototype

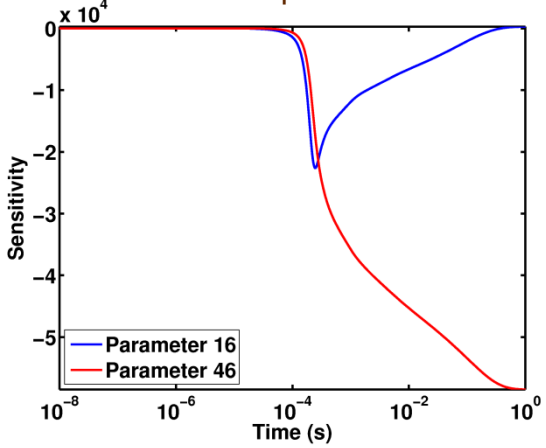
- Bipolar Junction Transistor
- Pseudo 1D strip (9x0.1 micron)
- Full defect physics
- 126 parameters



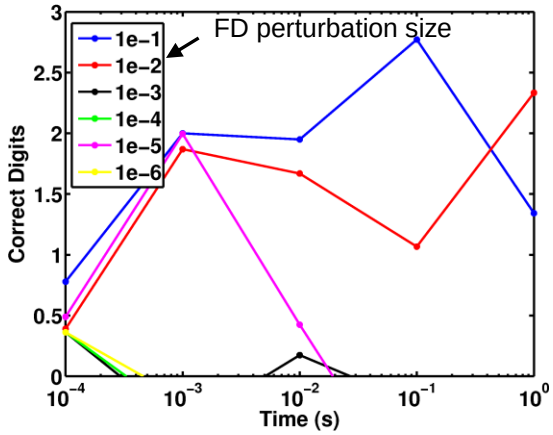
Sensitivities show dominant physics



Sensitivities computed at all times



1st-order Finite Difference Accuracy



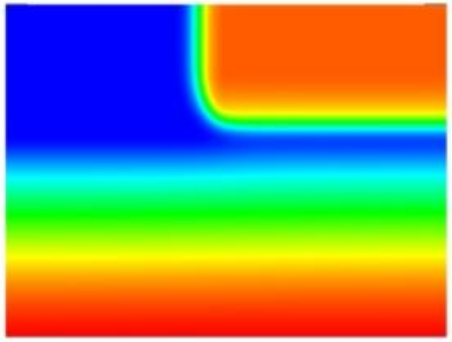
Comparison to FD:

- ✓ 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.**

cores	DOF	Jacobian time
256	7.93m	52.19
1024	31.5m	52.28
4096	126m	52.09
8192	253m	52.82
16384	504m	52.74
32768	1.01b	52.96
65536	2.01b	52.94



Example: JFNK
(2D Diffusion/Rxn System: 2 eqns)

- **JFNK (FD)**

$$Jv \approx \frac{F(x+\delta v) - F(x)}{\delta}$$

$$t \approx (\text{num\_Its}) * cost(F)$$

- **JFNK (AD)**
  - **Machine precision accurate**
  - **Ex: Solution varies 10^12 over domain**

$$Jv \leq 2.5 * cost(F)$$

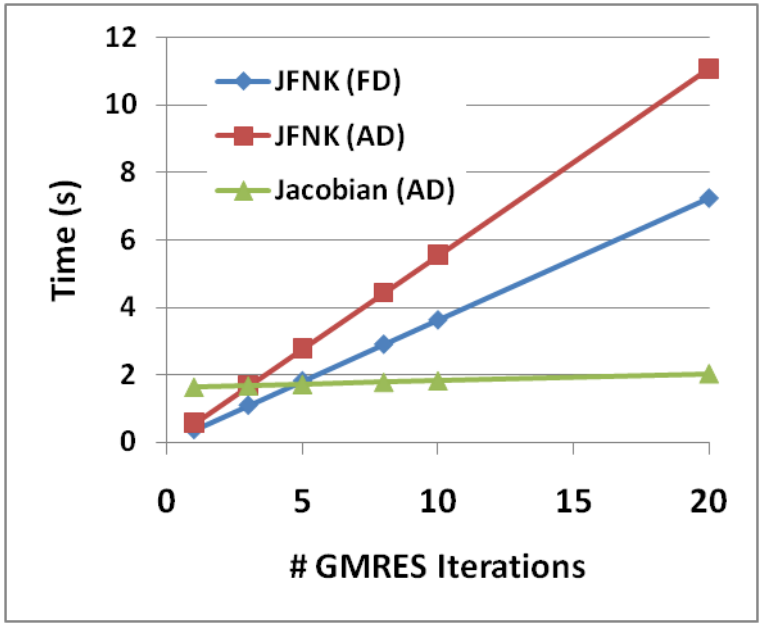
$$t \approx 1.53 * (num\_Its) * cost(F)$$

- **Explicit Jacobian (AD generated)**
  - **Machine precision accurate**
  - **Complexity ideas allow for storing individual operators for preconditioning!**
  - **Larger memory requirements**

$$J(x) \leq 13 * cost(F)$$

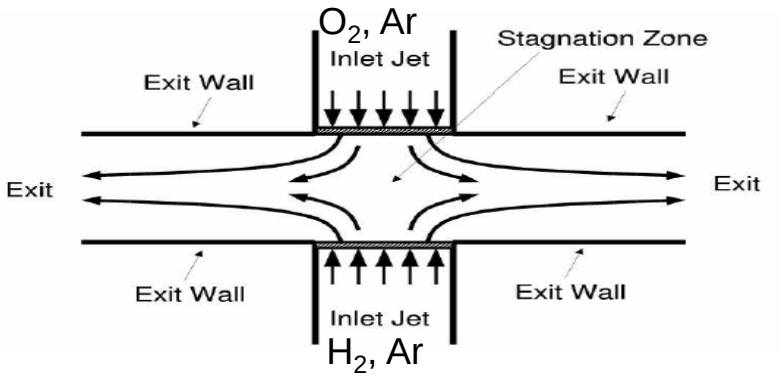
$$t \approx 4.45 + (num\_Its) * cost(Mv)$$

	Relative times	
JFNK (AD)	F(x)	1.00
Explicit J (AD)	J(x)	4.45
JFNK (AD)	Jv (AD)	1.53
Explicit J (AD)	Mv (matvec)	0.06

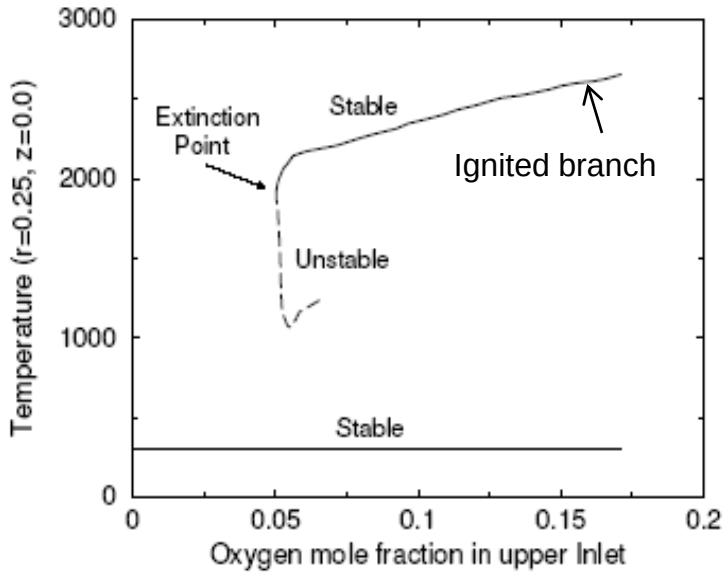


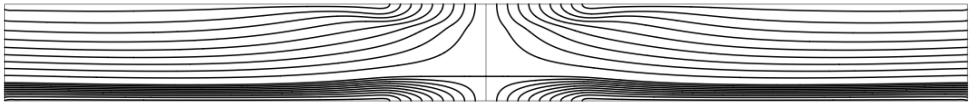
Sandia National Laboratories

Multiple-time-scale systems: Bifurcation Analysis of a Steady Reacting H_2, O_2, Ar, Opposed Flow Jet Reactor

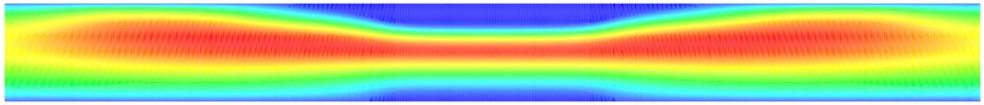


70 steady state reacting flow solves
(10 species, 19 reactions)

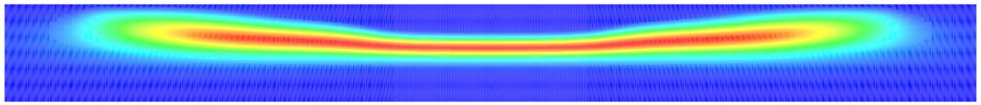




Streamlines



Temperature (Min. 300ºK, Max 2727ºK)



OH (Min. 0.0, Max 0.177)

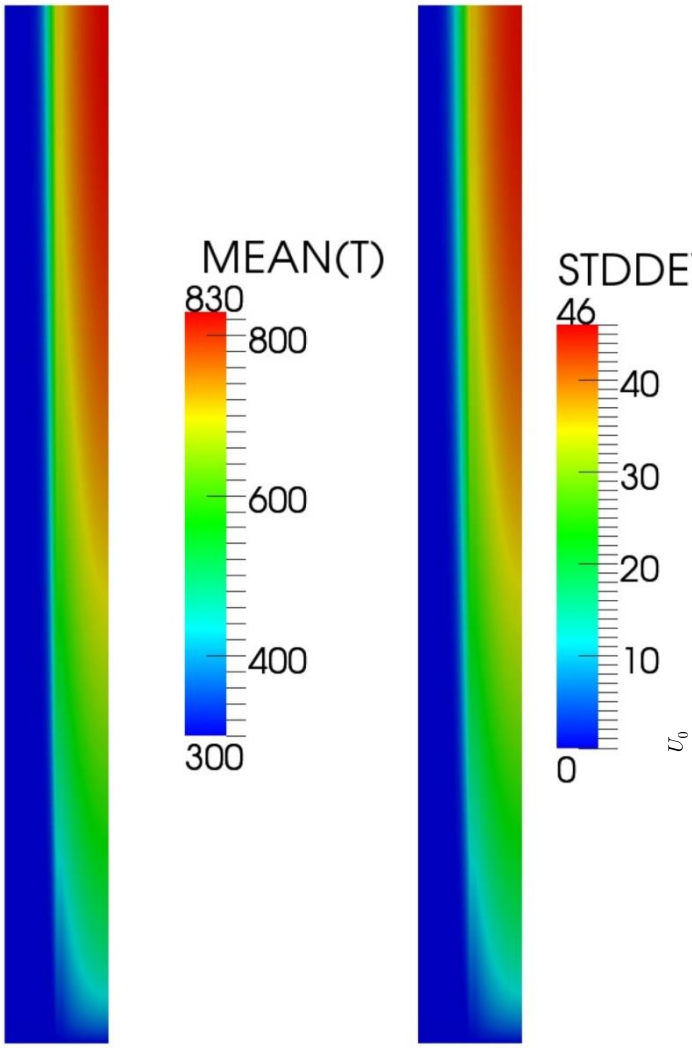
Approx. Physical Time scales (sec.):

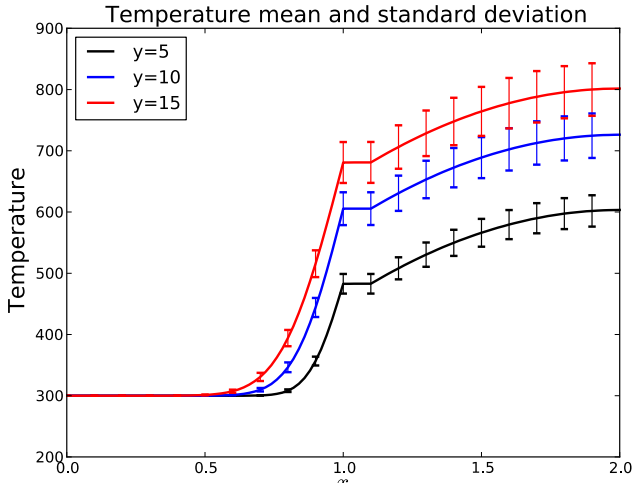
- Chemical kinetics: 10^{-12} to 10^{-4}
- Momentum diffusion: 10^{-6}
- Heat conduction: 10^{-6}
- Mass diffusion: 10^{-5} to 10^{-4}
- Convection: 10^{-5} to 10^{-4}
- Diffusion flame dynamics: (steady)

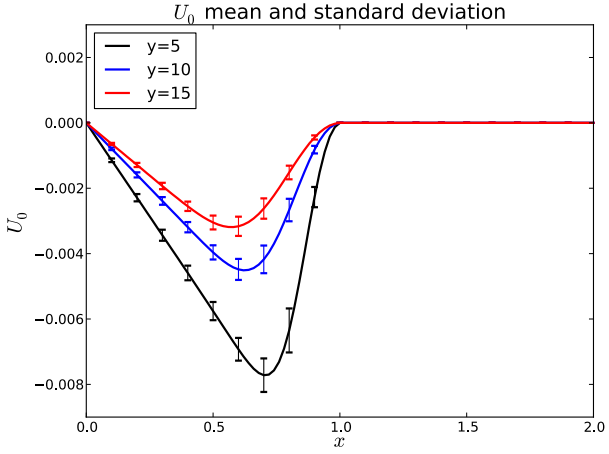
Sandia National Laboratories

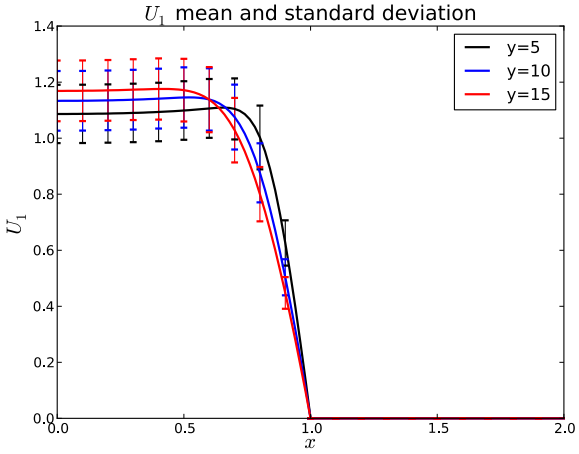
Embedded UQ in Drekar:

Rod to Fluid Heat Transfer









Sandia National Laboratories

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

Sandia National Laboratories

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

1. R. P. Pawlowski, E. T. Phipps and A. G. Salinger, **Automating Embedded Analysis Capabilities and Managing Software Complexity in Multiphysics Simulation, Part I:Template-based Generic Programming**, Scientific Programming 20 (2012) 197–219.
2. R. P. Pawlowski, E. T. Phipps, A. G. Salinger, S. J. Owen, C. M. Siefert and M. L. Staten, **Automating Embedded Analysis Capabilities and Managing Software Complexity in Multiphysics Simulation, Part II: Application to Partial Differential Equations**, Scientific Programming 20 (2012) 327–345.